Getting Started with the Entity Framework 4.1 using ASP.NET MVC

Tom Dykstra

Step-by-Step

Microsoft

Getting Started with the Entity Framework 4.1 Using ASP.NET MVC

Tom Dykstra

Summary: In this book, you'll learn the basics of using Entity Framework Code First to display and edit data in an ASP.NET MVC application.

Category: Step-by-Step
Applies to: ASP.NET 4.0, MVC 3, Entity Framework 4.1, Visual Studio 2010
Source: ASP.NET site (link to source content)
E-book publication date: June 2012

Microsoft

Contents

Creating an Entity Framework Data Model for an ASP.NET MVC Application

The Contoso University sample web application demonstrates how to create ASP.NET MVC applications using the Entity Framework. The sample application is a website for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments.

This tutorial series explains the steps taken to build the Contoso University sample application. You can download the completed application or create it by following the steps in the tutorial. The tutorial shows examples in C#. The downloadable sample contains code in both C# and Visual Basic. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET Entity Framework forum or the Entity Framework and LINQ to Entities forum.

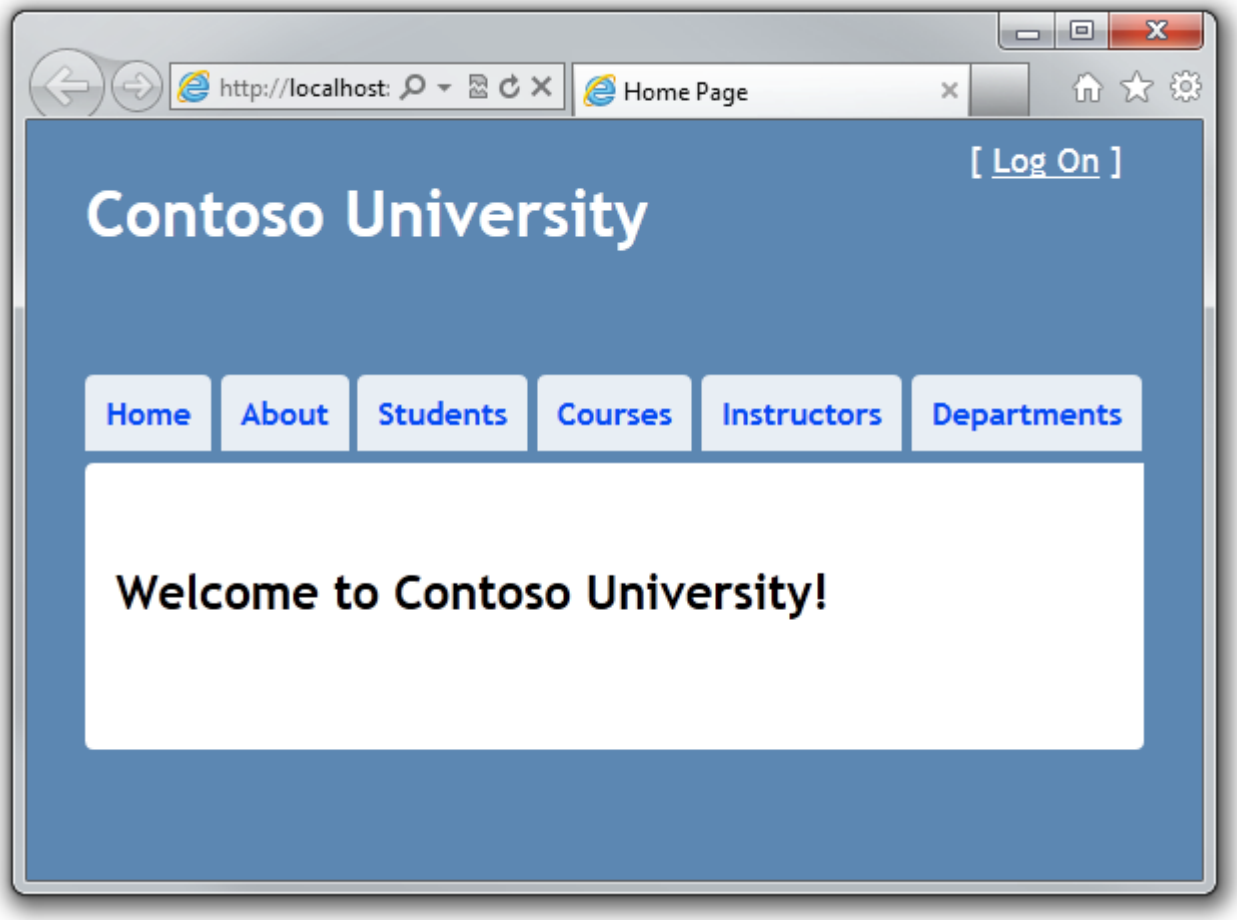
This tutorial series assumes you know how to work with ASP.NET MVC in Visual Studio. If you don't, a good place to start is a basic ASP.NET MVC Tutorial. If you prefer to work with the ASP.NET Web Forms model, see the Getting Started with the Entity Framework and Continuing with the Entity Framework tutorials.

Before you start, make sure you have the following software installed on your computer:

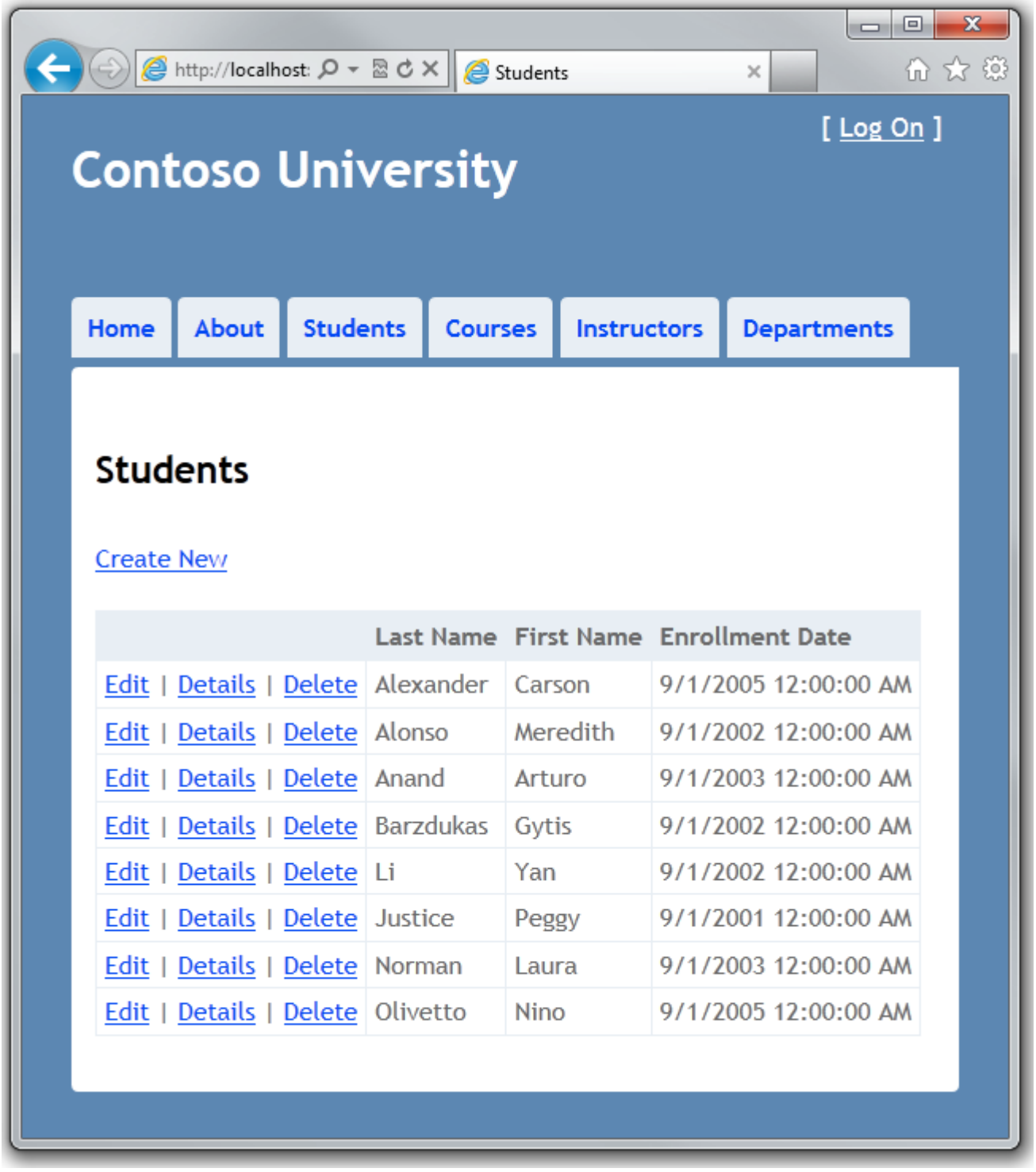
- Visual Studio 2010 SP1 or Visual Web Developer Express 2010 SP1 (If you use one of these links, the following items will be installed automatically.)
- ASP.NET MVC 3 Tools Update
- Microsoft SQL Server Compact 4.0
- Microsoft Visual Studio 2010 SP1 Tools for SQL Server Compact 4.0

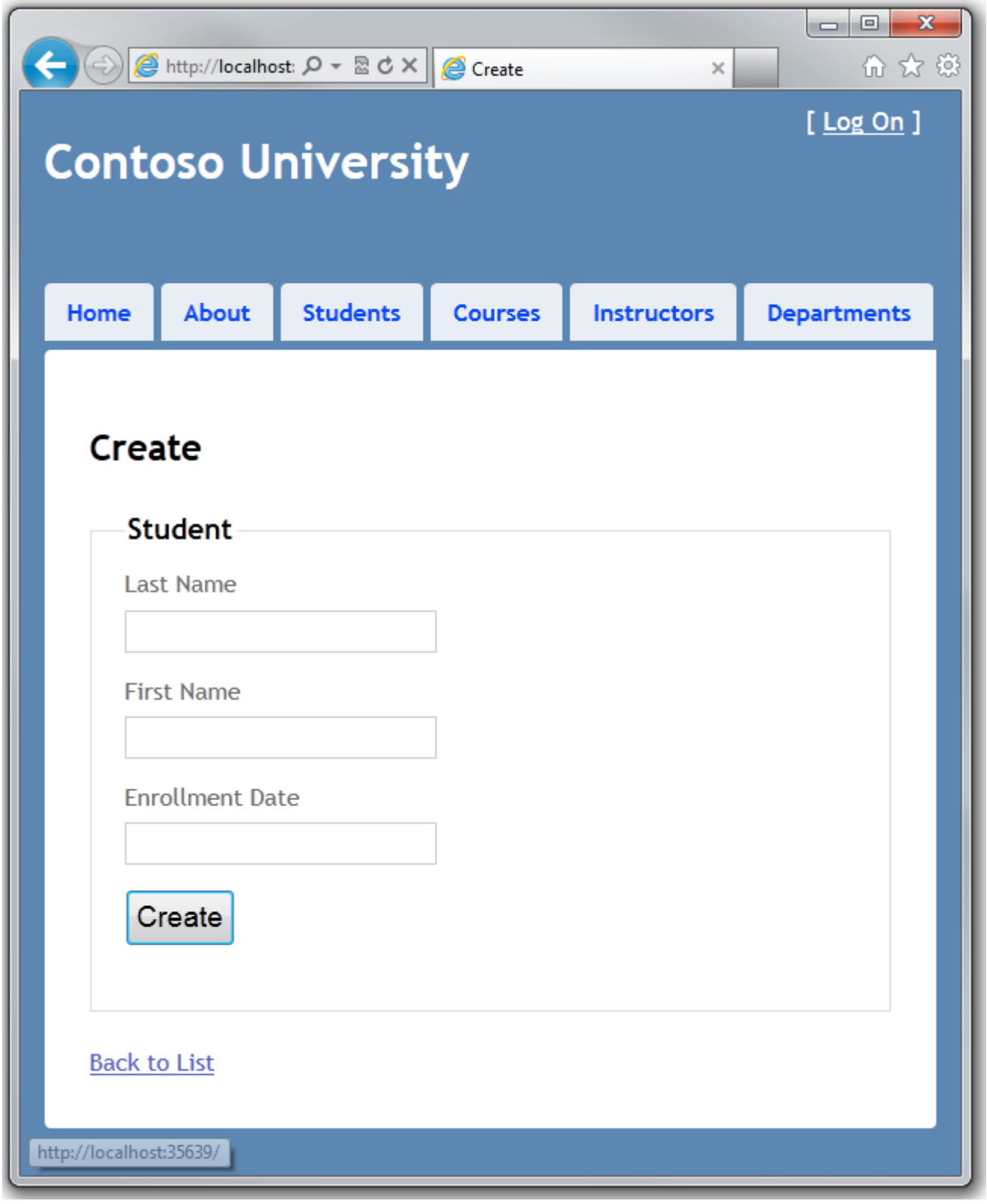
The Contoso University Web Application

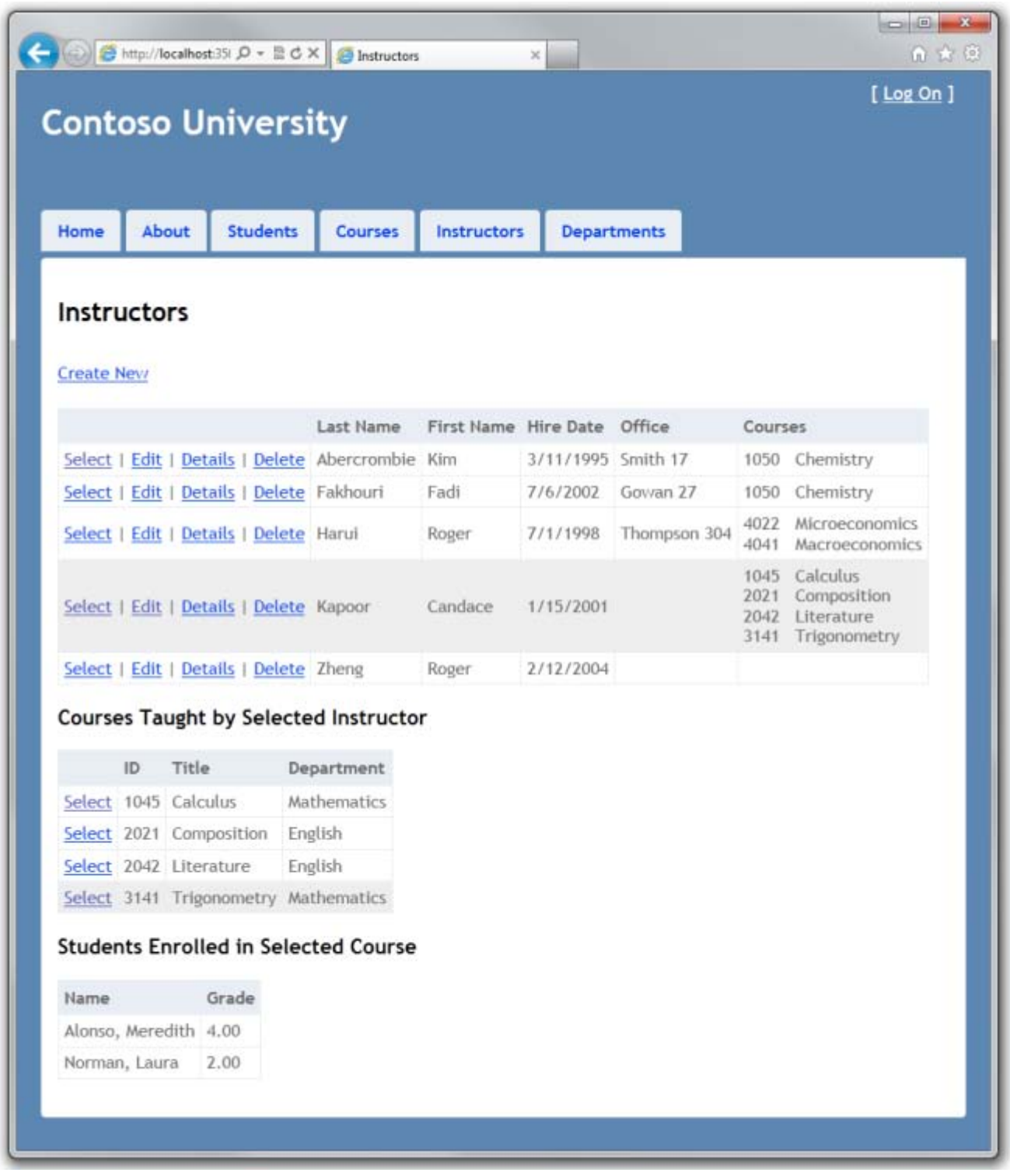
The application you'll be building in these tutorials is a simple university website.



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


http://localhost:
Students
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Students
Create New
Last Name
First Name
Enrollment Date
Edit | Details | Delete
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

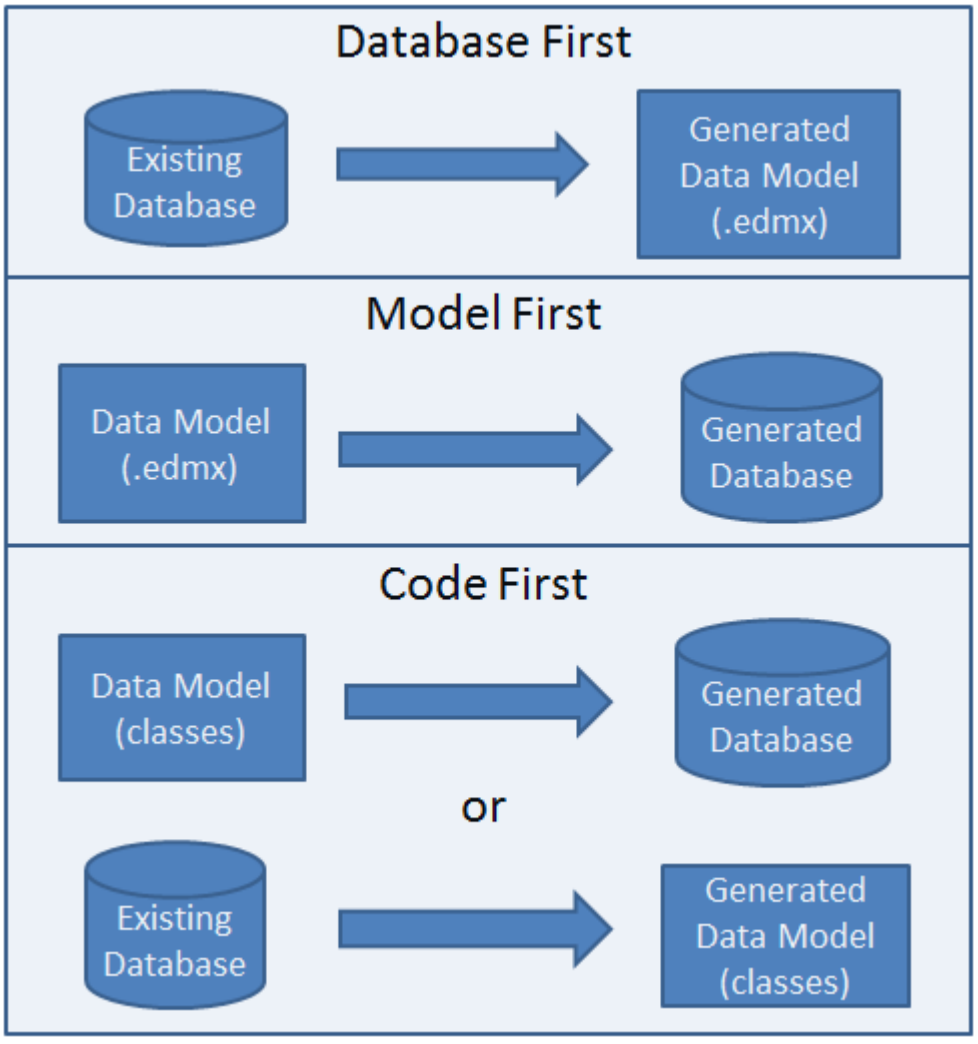

http://localhost:
Create
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Create
Student
Last Name
First Name
Enrollment Date
Create
Back to List
http://localhost:35639/



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.

Entity Framework Development Approaches

As shown in the following diagram, there are three ways you can work with data in the Entity Framework: *Database First*, *Model First*, and *Code First*.



Database First

If you already have a database, the Entity Framework can automatically generate a data model that consists of classes and properties that correspond to existing database objects such as tables and columns. The information about your database structure (*store schema*), your data model (*conceptual model*), and the mapping between them is stored in XML in an *.edmx* file. Visual Studio provides the Entity Framework designer, which is a graphical designer that you can use to display and edit the *.edmx* file. The Getting Started With the Entity Framework and Continuing With the Entity Frameworktutorial sets use Database First development.

Model First

If you don't yet have a database, you can begin by creating a model using the Entity Framework designer in Visual Studio. When the model is finished, the designer can generate DDL (*data definition language*) statements to create the database. This approach also uses an *.edmx* file to store model and mapping information. The What's New in the Entity Framework 4 tutorial includes a brief example of Model First development.

Code First

Whether you have an existing database or not, you can code your own classes and properties that correspond to tables and columns and use them with the Entity Framework without an *.edmx* file. That's why you sometimes see this approach called *code only*, although the official name is Code First. The mapping between the store schema and the conceptual model represented by your code is handled by convention and by a special mapping API. If you don't yet have a database, the Entity Framework can automatically create the database for you, or drop and re-create it if the model changes. This tutorial series uses Code First development.

The data access API that was developed for Code First is based on the `DbContext` class. This API can also be used with the Database First and Model First development workflows. For more information, see When is Code First not code first? on the Entity Framework team blog.

POCO (Plain Old CLR Objects)

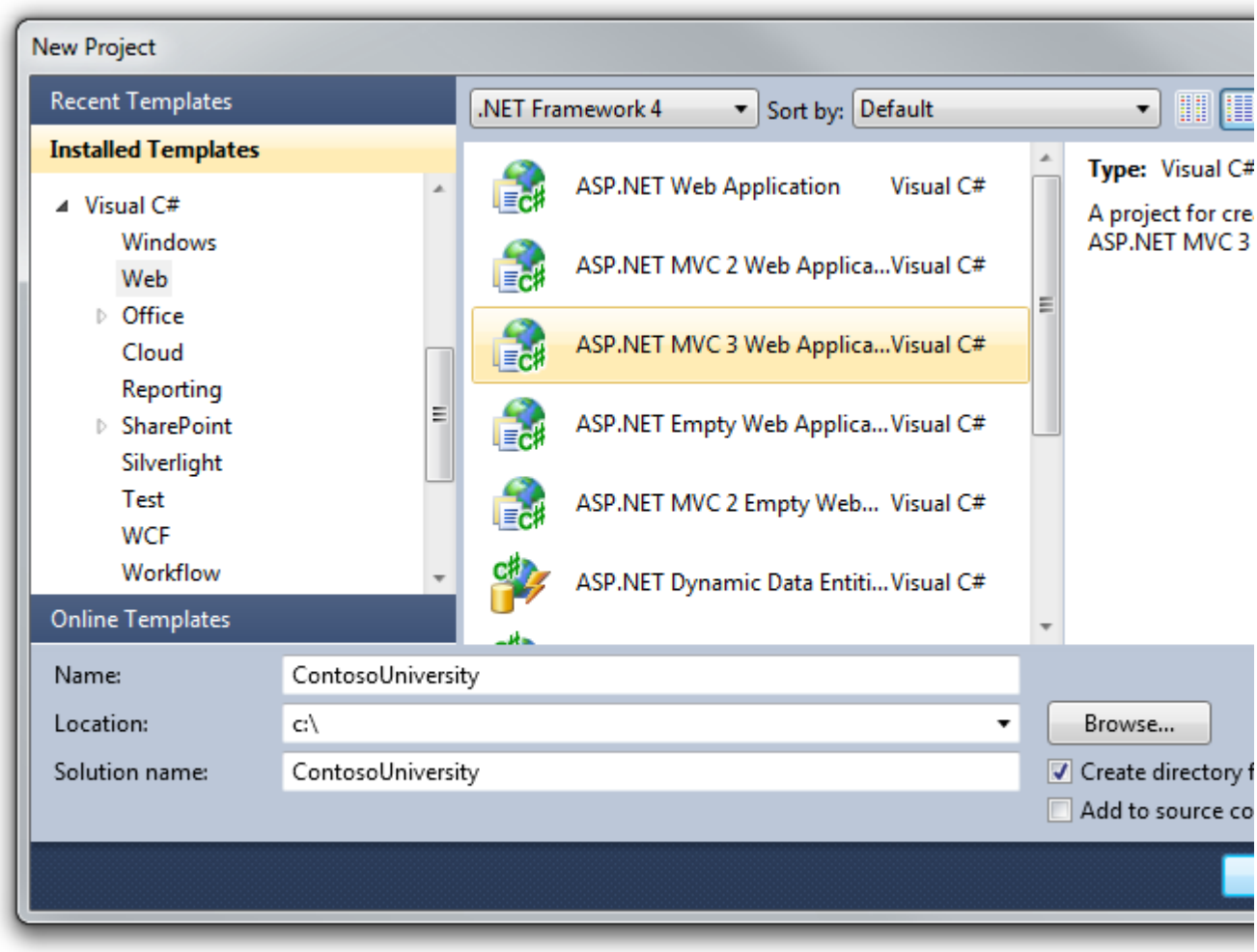
By default, when you use the Database First or Model First development approaches, the entity classes in your data model inherit from the EntityObject class, which provides them with Entity Framework functionality. This means that these classes technically aren't persistence ignorant and so don't conform fully to one of the requirements of domain-driven design. All development approaches of the Entity Framework can also work with POCO (*plain old CLR objects*) classes, which essentially means that they are persistence-ignorant because they don't inherit from the `EntityObject` class. In this tutorial you'll use POCO classes.

Creating an MVC Web Application

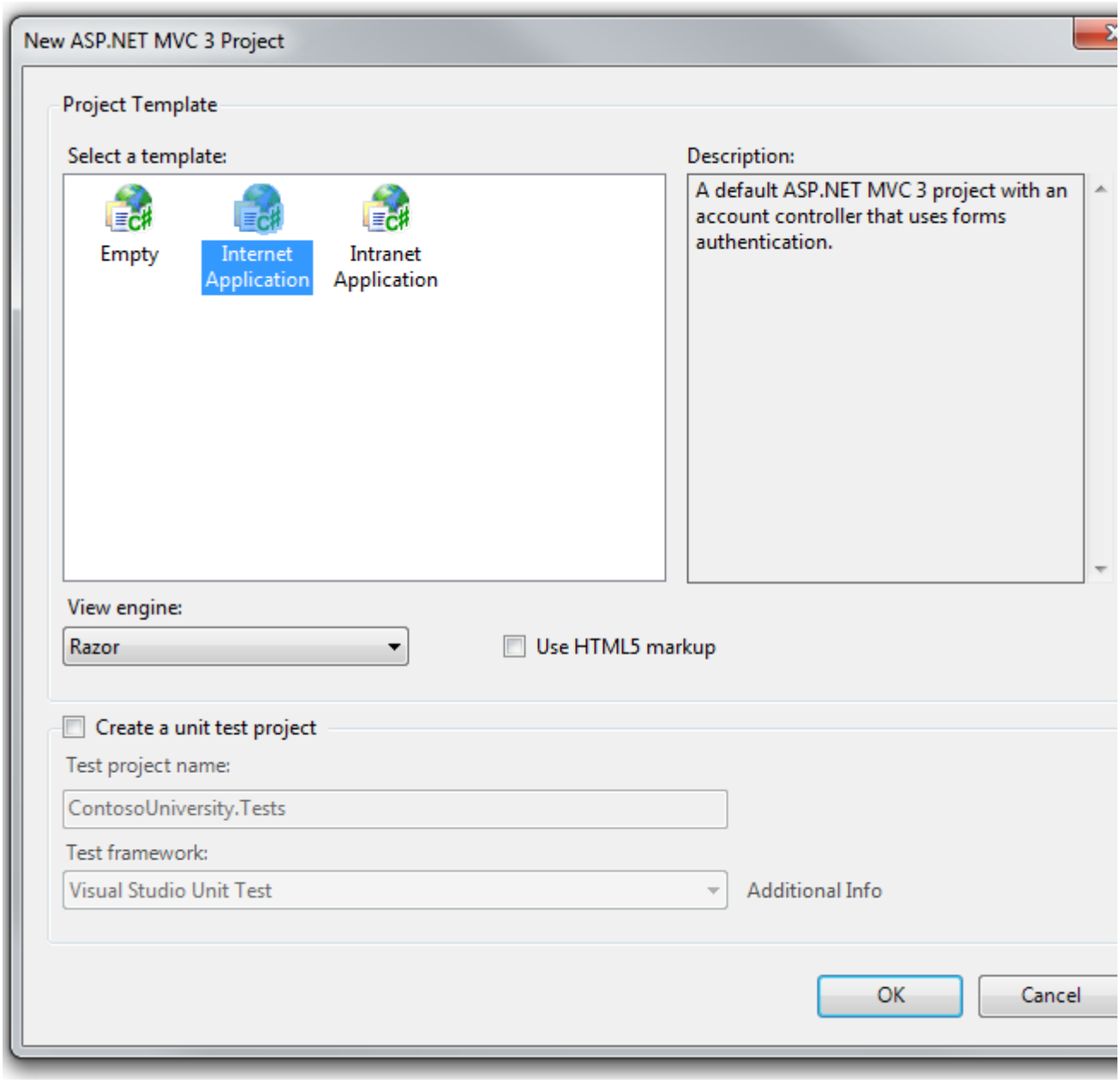
Before you start, make sure you have the following installed on your computer:

- Visual Studio 2010 SP1 or Visual Web Developer Express 2010 SP1 (If you use one of these links, the following items will be installed automatically.)
- ASP.NET MVC 3 Tools Update
- Microsoft SQL Server Compact 4.0
- Microsoft Visual Studio 2010 SP1 Tools for SQL Server Compact 4.0

Open Visual Studio and create a new project named "ContosoUniversity" using the **ASP.NET MVC 3 Web Application** template:



In the **New ASP.NET MVC 3 Project** dialog box select the **Internet Application** template and the **Razor** view engine, clear the **Create a unit test project** check box, and then click **OK**.



Setting Up the Site Style

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

In order to set up the Contoso University menu, in the *Views\Shared_Layout.cshtml* file, replace the existing `h1` heading text and the menu links, as shown in the following example:

```
<!DOCTYPE html>
<html>
```

```
<head>
<title>@ViewBag.Title</title>
<linkhref="@Url.Content("~/Content/Site.css")"rel="stylesheet"type="text/css"/>
<scriptsrc="@Url.Content("~/Scripts/jquery-
1.5.1.min.js")"type="text/javascript"></script>
</head>
<body>
<divclass="page">
<divid="header">
<divid="title">
<h1>Contoso University</h1>
</div>

<divid="logindisplay">
                @Html.Partial("_LogOnPartial")
</div>
<divid="menucontainer">
<ulid="menu">
<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>
<divid="main">
            @RenderBody()
</div>
<divid="footer">
</div>
</div>
</body>
</html>
```

In the *Views\Home\Index.cshtml* file, delete everything under the h2 heading.

In the *Controllers\HomeController.cs* file, replace "Welcome to ASP.NET MVC!" with "Welcome to Contoso University!"

In the *Content\Site.css* file, make the following changes in order to move the menu tabs to the left:

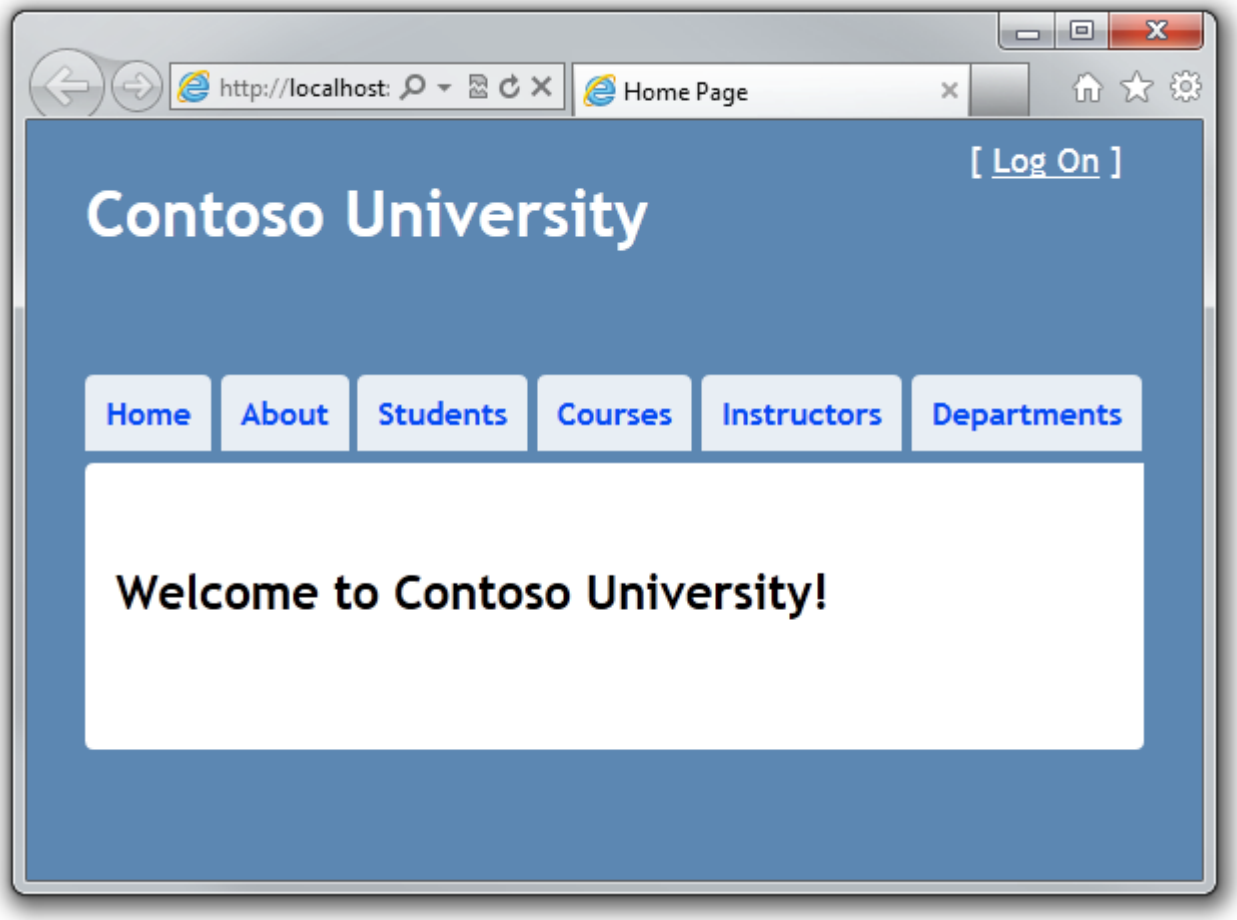
- In the definition for `#main`, add `clear: both;`, as shown in the following example:

```
#main
{
    clear: both;
    padding:30px30px15px30px;
    background-color:#fff;
    border-radius:4px000;
-webkit-border-radius:4px000;
-moz-border-radius:4px000;
}
```

- In the definition for `nav` and `#menucontainer`, add `clear: both; float: left;,` as shown in the following example:

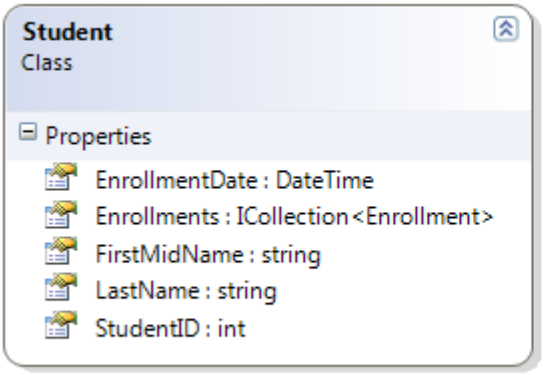
```
nav,
#menucontainer {
    margin-top:40px;
    clear: both;
float: left;
}
```

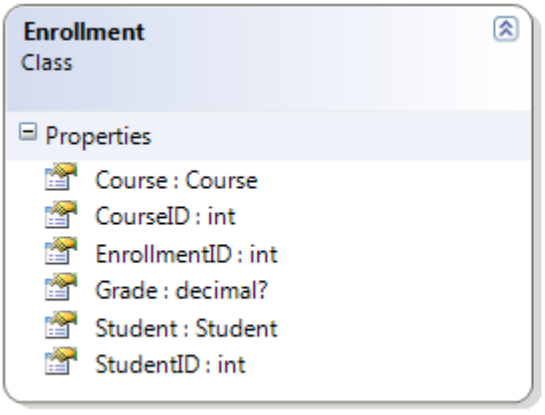
Run the site. You see the home page with the main menu.

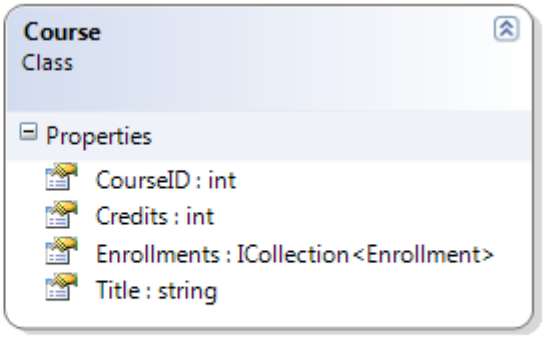


Creating the Data Model

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





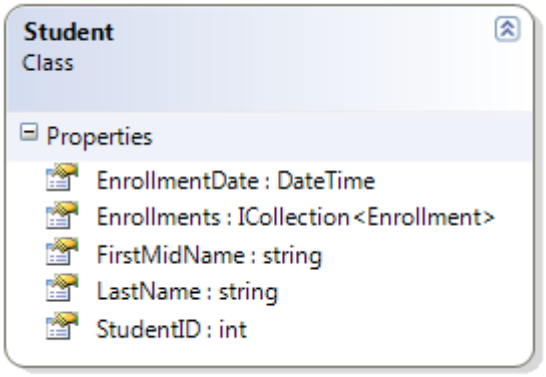


There's a one-to-many relationship between `Student` and `Enrollment` entities, and there's a one-to-many relationship between `Course` and `Enrollment` entities. In other words, a student can be enrolled in any number of courses, and a course can have any number of students enrolled in it.

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

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

The Student Entity



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

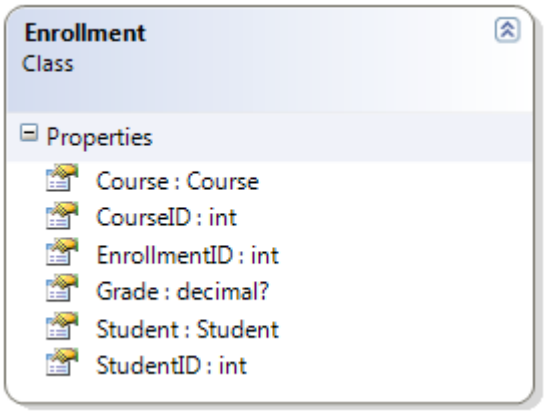
```
usingSystem;
usingSystem.Collections.Generic;

namespaceContosoUniversity.Models
{
publicclassStudent
{
publicintStudentID{get;set;}
publicstringLastName{get;set;}
publicstringFirstMidName{get;set;}
publicDateTimeEnrollmentDate{get;set;}
publicvirtualICollection<Enrollment>Enrollments{get;set;}
}
}
```

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

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

Navigation properties are typically defined as `virtual` so that they can take advantage of an Entity Framework function called *lazy loading*. (Lazy loading will be explained later.) If a navigation property can hold multiple entities (as in many-to-many or one-to-many relationships), its type must be `ICollection`.

The Enrollment Entity



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

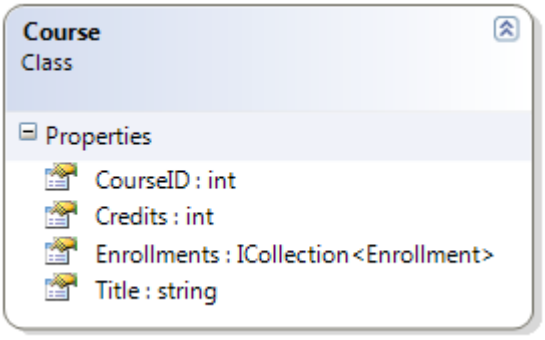
```
usingSystem;
usingSystem.Collections.Generic;

namespaceContosoUniversity.Models
{
publicclassEnrollment
{
publicintEnrollmentID{get;set;}
publicintCourseID{get;set;}
publicintStudentID{get;set;}
publicdecimal?Grade{get;set;}
publicvirtualCourseCourse{get;set;}
publicvirtualStudentStudent{get;set;}
}
}
```

The question mark after the `decimal` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade — null means a grade hasn't been assigned yet, while zero means a zero grade has been assigned.

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

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

The Course Entity



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

```
usingSystem;
usingSystem.Collections.Generic;

namespaceContosoUniversity.Models
{
publicclassCourse
{
publicintCourseID{get;set;}
publicstringTitle{get;set;}
publicintCredits{get;set;}
publicvirtualICollection<Enrollment>Enrollments{get;set;}
}
}
```

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

Creating 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 the code for this project, the class is named `SchoolContext`.

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

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.Data.Entity;
usingContosoUniversity.Models;
usingSystem.Data.Entity.ModelConfiguration.Conventions;

namespaceContosoUniversity.Models
{
publicclassSchoolContext:DbContext
{
publicDbSet<Student>Students{get;set;}
publicDbSet<Enrollment>Enrollments{get;set;}
publicDbSet<Course>Courses{get;set;}

protectedoverridevoidOnModelCreating(DbModelBuilder modelBuilder)
{
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
}
}
}
```

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

The statement in the `OnModelCreating` method prevents table names from being pluralized. If you didn't do this, the generated tables would be named `Students`, `Courses`, and `Enrollments`. Instead, the table names will be `Student`, `Course`, and `Enrollment`. Developers disagree about whether table names should be

pluralized or not. This tutorial uses the singular form, but the important point is that you can select whichever form you prefer by including or omitting this line of code.

(This class is in the Models namespace, because in some situations Code First assumes that the entity classes and the context class are in the same namespace.)

Setting the Connection String

You don't have to create a connection string. If you don't create one, the Entity Framework will automatically create a SQL Server Express database for you. In this tutorial, however, you'll work with SQL Server Compact, so you need to create a connection string to specify that.

Open the project *Web.config* file and add a new connection string to the `connectionStrings` collection, as shown in the following example. (Make sure you update the *Web.config* file in the root project folder. There's also a *Web.config* file is in the *Views* subfolder that you don't need to update.)

```
<addname="SchoolContext"connectionString="Data
Source=|DataDirectory|School.sdf"providerName="System.Data.SqlServerCe.4.0"/>
```

By default, the Entity Framework looks for a connection string named the same as the object context class. The connection string you've added specifies a SQL Server Compact database named *School.sdf* located in the *App_Data* folder.

Initializing 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 class that includes a method that the Entity Framework automatically calls after creating the database in order to populate it with test data. In this section you'll specify that the database should be dropped and re-created whenever the model changes.

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

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.Linq;
usingSystem.Web;
```

```
usingSystem.Data.Entity;
usingContosoUniversity.Models;

namespaceContosoUniversity.DAL
{
publicclassSchoolInitializer:DropCreateDatabaseIfModelChanges<SchoolContext>
{
protectedoverridevoidSeed(SchoolContext context)
{
var students =newList<Student>
{
newStudent{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("
2005-09-01")},
newStudent{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2
002-09-01")},
newStudent{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2003
-09-01")},
newStudent{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2
002-09-01")},
newStudent{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002-09-
01")},
newStudent{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("200
1-09-01")},
newStudent{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2003
-09-01")},
newStudent{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("200
5-09-01")}
};
            students.ForEach(s => context.Students.Add(s));
            context.SaveChanges();

var courses =newList<Course>
{
newCourse{Title="Chemistry",Credits=3,},
newCourse{Title="Microeconomics",Credits=3,},
newCourse{Title="Macroeconomics",Credits=3,},
newCourse{Title="Calculus",Credits=4,},
newCourse{Title="Trigonometry",Credits=4,},
```

```
newCourse{Title="Composition",Credits=3,},
newCourse{Title="Literature",Credits=4,}
};
            courses.ForEach(s => context.Courses.Add(s));
            context.SaveChanges();

var enrollments =newList<Enrollment>
{
newEnrollment{StudentID=1,CourseID=1,Grade=1},
newEnrollment{StudentID=1,CourseID=2,Grade=3},
newEnrollment{StudentID=1,CourseID=3,Grade=1},
newEnrollment{StudentID=2,CourseID=4,Grade=2},
newEnrollment{StudentID=2,CourseID=5,Grade=4},
newEnrollment{StudentID=2,CourseID=6,Grade=4},
newEnrollment{StudentID=3,CourseID=1},
newEnrollment{StudentID=4,CourseID=1,},
newEnrollment{StudentID=4,CourseID=2,Grade=4},
newEnrollment{StudentID=5,CourseID=3,Grade=3},
newEnrollment{StudentID=6,CourseID=4},
newEnrollment{StudentID=7,CourseID=5,Grade=2},
};
            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.

Make the following changes in the *Global.asax.cs* file to cause this initializer code to run when the application begins:

- Add `using` statements:

```
usingSystem.Data.Entity;
usingContosoUniversity.Models;
usingContosoUniversity.DAL;
```

- In the `Application_Start` method, call an Entity Framework method that runs the database initializer code:

```
Database.SetInitializer<SchoolContext>(newSchoolInitializer());
```

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` class). If there's a difference, the application drops and re-creates the database.

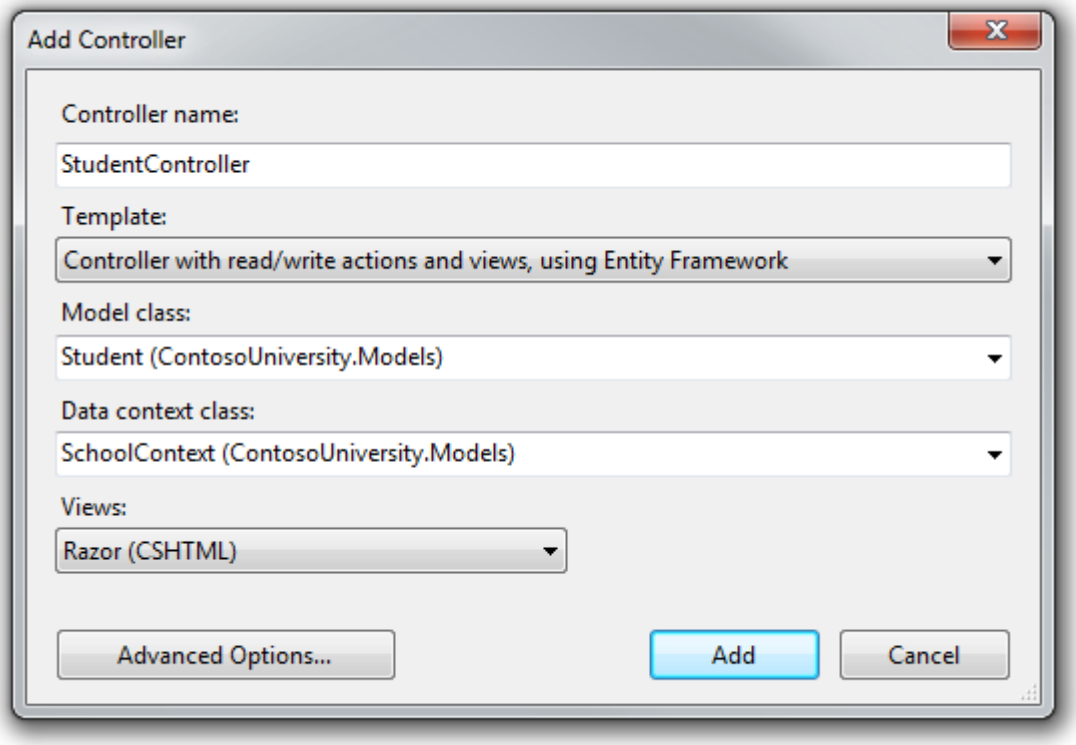
Note When you deploy an application to a production web server, you must remove code that seeds the database.

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.

Creating a Student Controller

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

- Controller name: **StudentController**.
- Template: **Controller with read/write actions and views, using Entity Framework**. (The default.)
- Model class: **Student (ContosoUniversity.Models)**. (If you don't see this option in the drop-down list, build the project and try again.)
- Data context class: **SchoolContext (ContosoUniversity.Models)**.
- Views: **Razor (CSHTML)**. (The default.)



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

```
privateSchoolContext db =newSchoolContext();
```

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

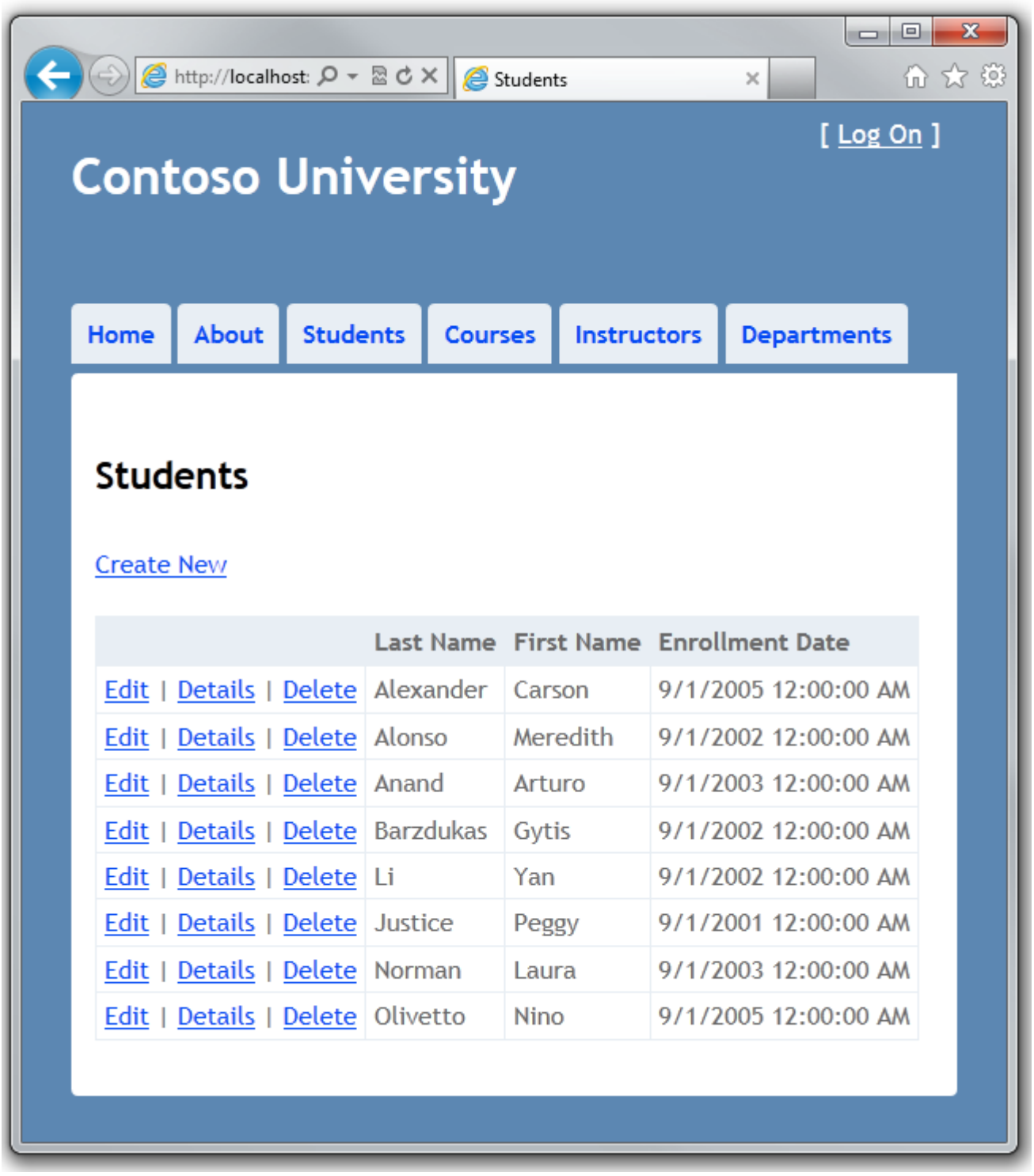
```
publicViewResultIndex()
{
returnView(db.Students.ToList());
}
```

The automatic scaffolding has also created a set of `Student` views. To customize the default headings and column order in the `Index` view, open *Views\Student\Index.cshtml* and replace the existing code with the following code:

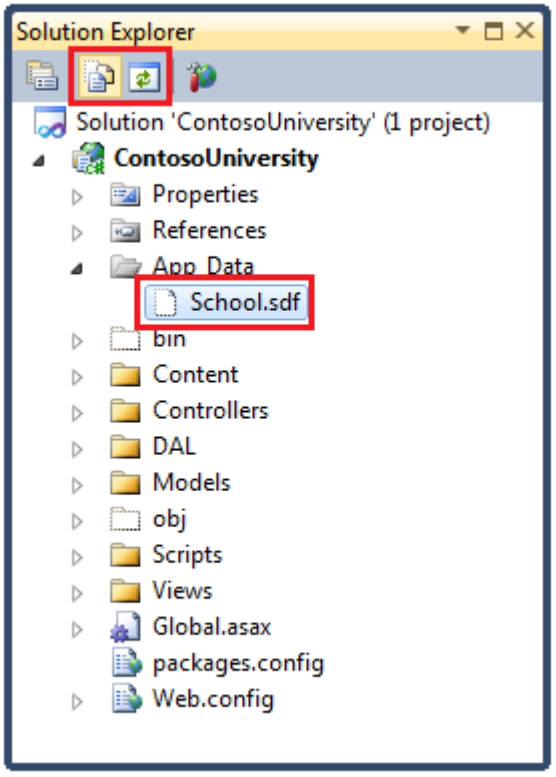
```
@modelIEnumerable<ContosoUniversity.Models.Student>
```

```
@{
ViewBag.Title="Students";
}

<h2>Students</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
<tr>
<th></th>
<th>Last Name</th>
<th>FirstName</th>
<th>Enrollment Date</th>
</tr>

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

</table>
```

Run the site, click the **Students** tab, and you see a list of students.

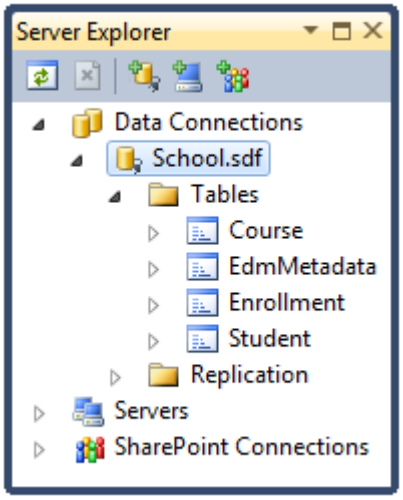


Close the browser. In **Solution Explorer**, select the **ContosoUniversity** project (make sure the project and not the solution is selected). Click **Show all Files** if you aren't already in that mode. Click **Refresh** and then expand the *App_Data* folder to see the *School.sdf* file.



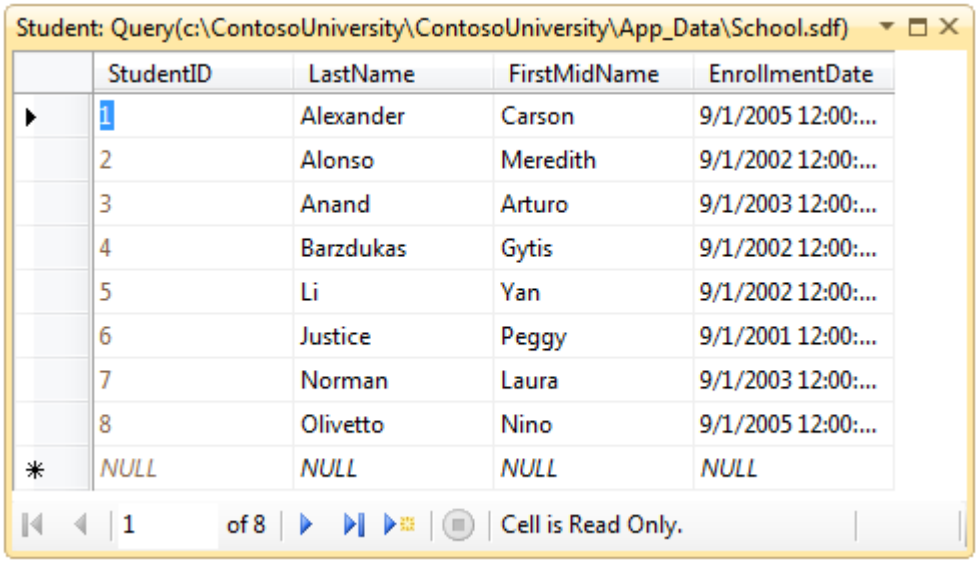
Double-click *School.sdf* to open **Server Explorer**. Then expand the **Tables** folder to see the tables that have been created in the database.

Note If you get an error when you double-click *School.sdf*, make sure you have installed **Visual Studio 2010 SP1 Tools for SQL Server Compact 4.0**. (For links to the software, see the list of prerequisites at the top of this page.) If you install the tools now, you'll have to close and re-open Visual Studio.

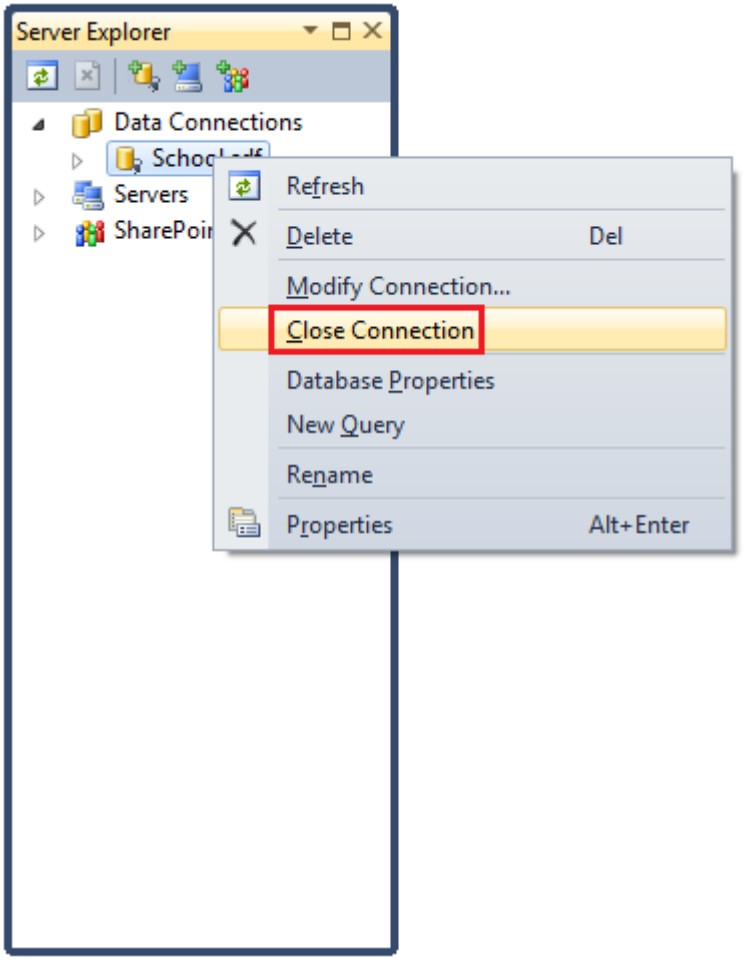


There's one table for each entity set, plus one additional table. The `EdmMetadata` table is used by the Entity Framework to determine when the model and the database are out of sync.

Right-click one of the tables and select **Show Table Data** to see the data that was loaded in the table by the `SchoolInitializer` class.



When you're finished, close the connection. (If you don't close the connection, you might get an error the next time you run the project).



Conventions

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

- The pluralized forms of entity class names are used as table names.
- Entity property names are used for column names.
- Entity properties that are named `ID` or *classname*`ID` are recognized as primary key properties.
- The Entity Framework connects to your database by looking for a connection string that has the same name as your context class (in this case, `SchoolContext`).

You've seen that conventions can be overridden (for example, you specified that table names shouldn't be pluralized), and you'll learn more about conventions and how to override them later.

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

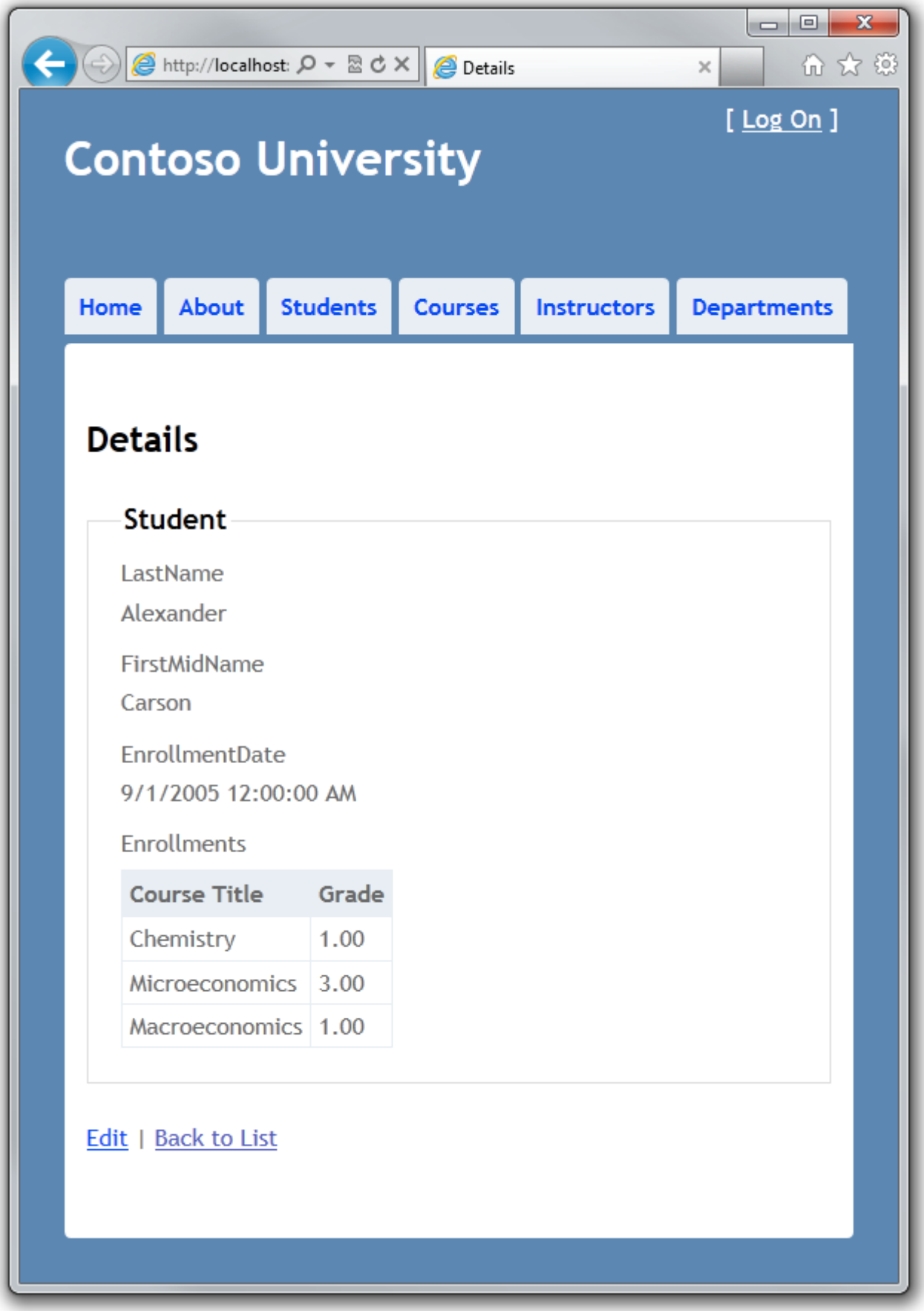
Links to other Entity Framework resources can be found at the end of this e-book.

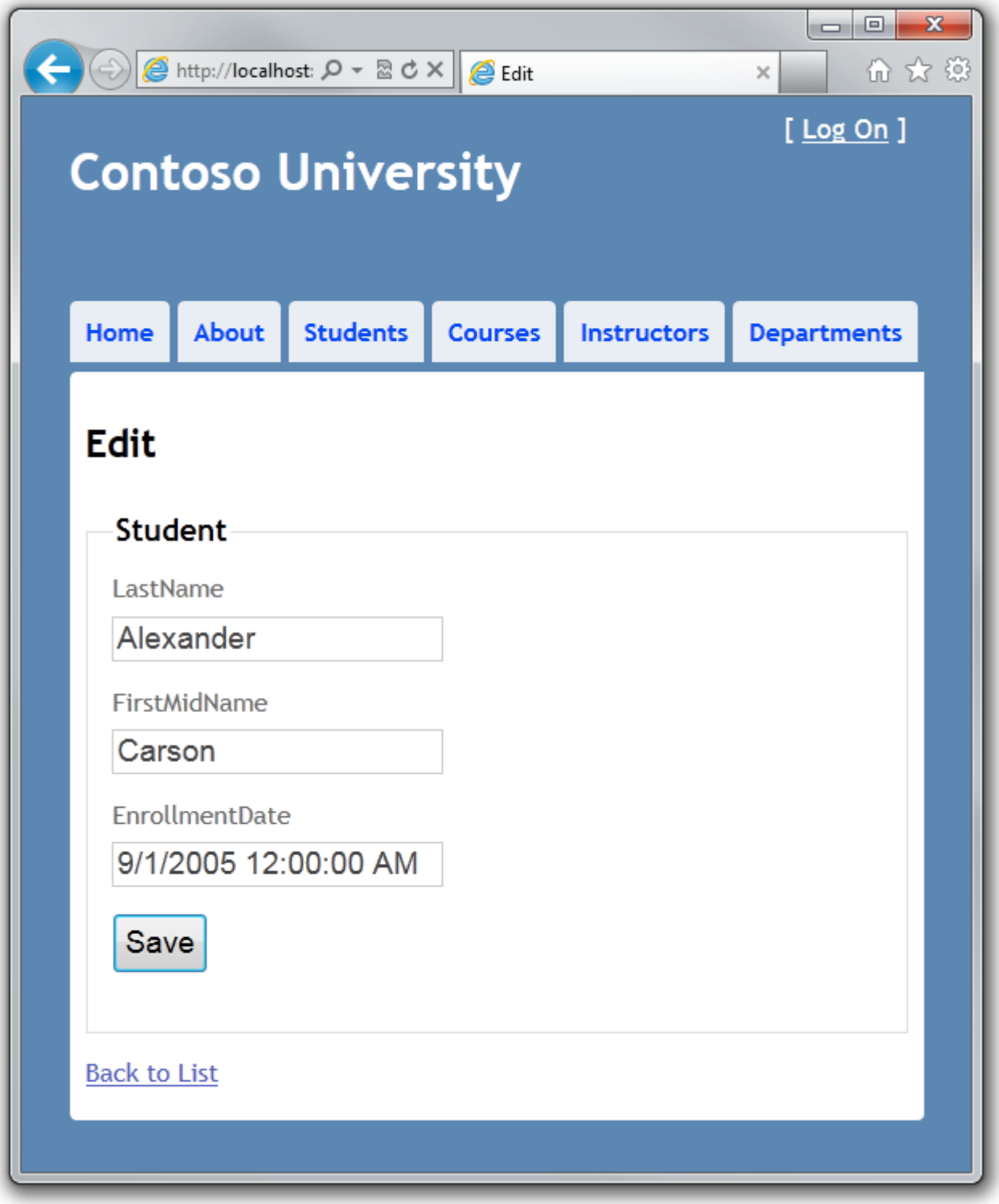
Implementing Basic CRUD Functionality

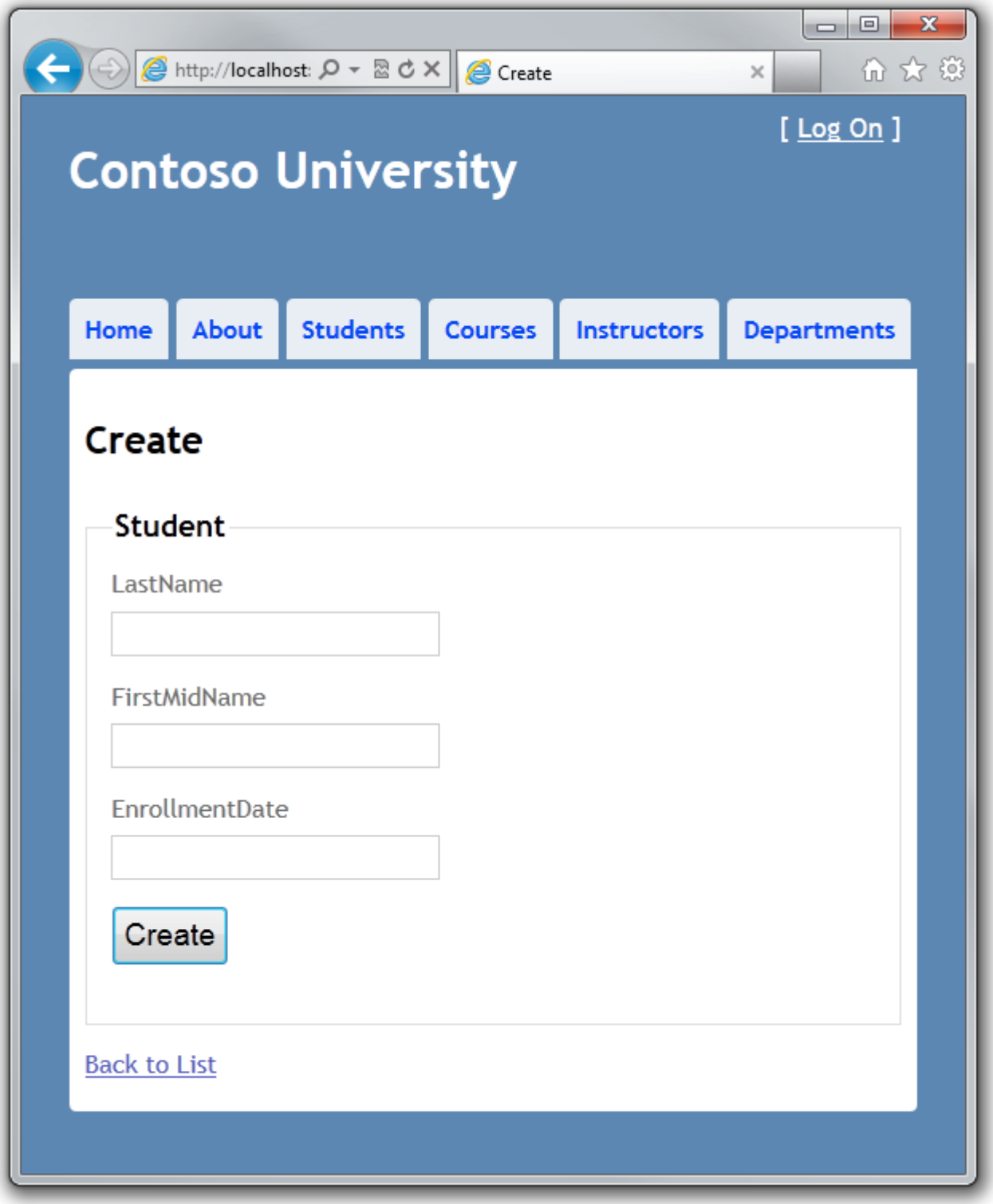
In the previous tutorial you created an MVC application that stores and displays data using the Entity Framework and SQL Server Compact. In this tutorial you will review and customize the CRUD (create, read, update, delete) code that the MVC scaffolding automatically creates for you in controllers and 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, you won't implement a repository until a later tutorial in this series.

In this tutorial, you will create the following web pages:


http://localhost:
Details
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Details
Student
LastName
Alexander
FirstMidName
Carson
EnrollmentDate
9/1/2005 12:00:00 AM
Enrollments
Course Title
Grade
Chemistry
1.00
Microeconomics
3.00
Macroeconomics
1.00
Edit | Back to List


http://localhost:
Edit
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Edit
Student
LastName
Alexander
FirstMidName
Carson
EnrollmentDate
9/1/2005 12:00:00 AM
Save
Back to List


http://localhost:
Create
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Create
Student
LastName
FirstMidName
EnrollmentDate
Create
Back to List

Creating a Details Page

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

In *Controllers\StudentController.cs*, the action method for the Details view resembles the following example:

```
publicViewResultDetails(int id)
{
Student student = db.Students.Find(id);
returnView(student);
}
```

The code uses the `Find` method to retrieve a single `Student` entity corresponding to the key value that's passed to the method as the `id` parameter. The `id` value comes from a query string in the **Details** hyperlink on the Index page.

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

```
<divclass="display-label">LastName</div>
<divclass="display-field">
    @Html.DisplayFor(model => model.LastName)
</div>
```

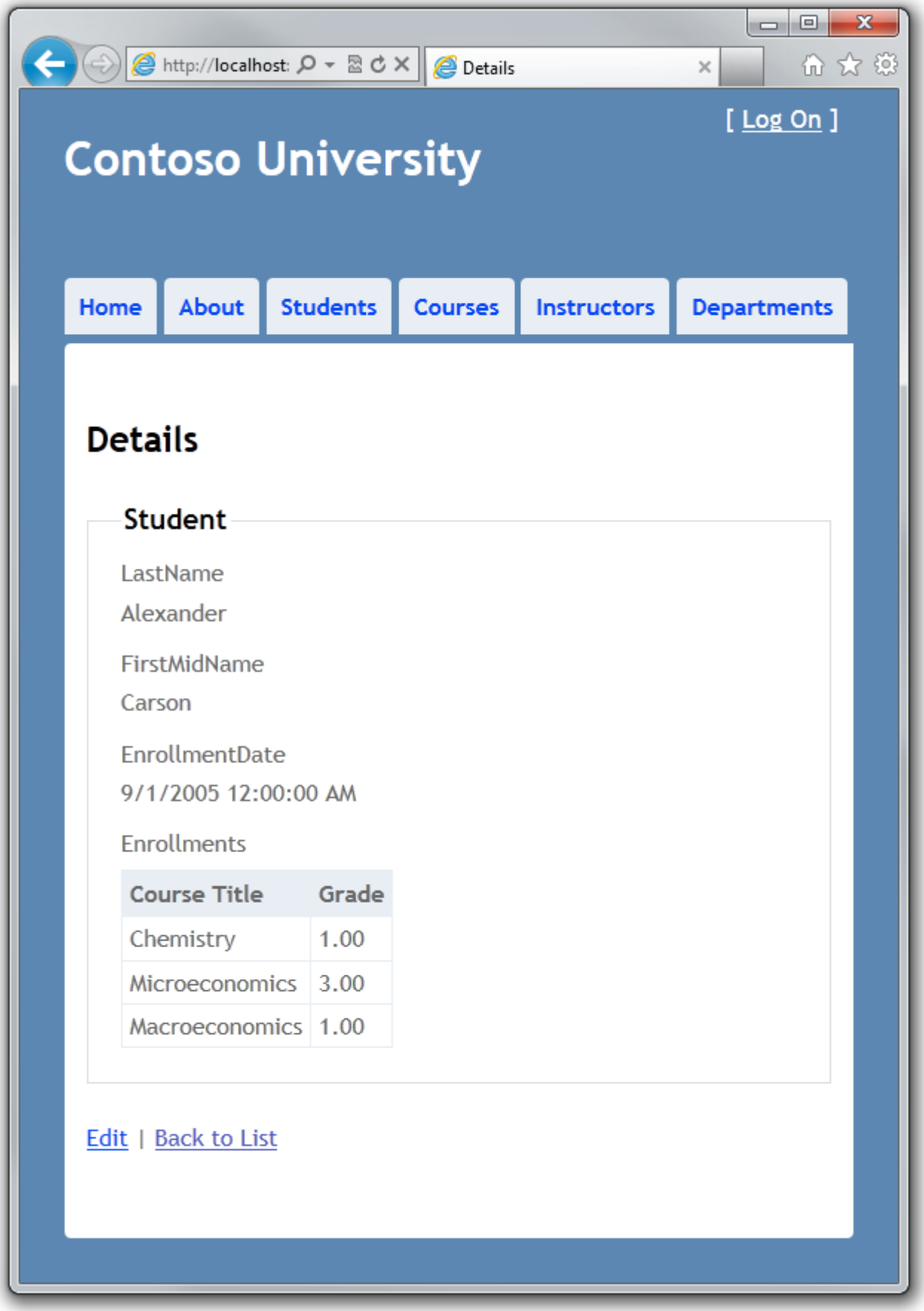
To display a list of enrollments, add the following code after the `EnrollmentDate` field, immediately before the closing `fieldset` tag:

```
<divclass="display-label">
    @Html.LabelFor(model => model.Enrollments)
</div>
<divclass="display-field">
<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>
</div>
```

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 first time you try to access that property, a 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.)

Run the page by selecting the **Students** tab and clicking a **Details** hyperlink. You see the list of courses:


http://localhost:
Details
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Details
Student
LastName
Alexander
FirstMidName
Carson
EnrollmentDate
9/1/2005 12:00:00 AM
Enrollments
Course Title
Grade
Chemistry
1.00
Microeconomics
3.00
Macroeconomics
1.00
Edit | Back to List

Creating a Create Page

In *Controllers\StudentController.cs*, replace the `HttpPostCreate` action method with the following code to add a `try-catch` block to the scaffolded method:

```
[HttpPost]
publicActionResultCreate(Student student)
{
try
{
if(ModelState.IsValid)
{
            db.Students.Add(student);
            db.SaveChanges();
returnRedirectToAction("Index");
}
}
catch(DataException)
{
//Log the error (add a variable name after DataException)
ModelState.AddModelError("","Unable to save changes. Try again, and if the problem
persists see your system administrator.");
}
returnView(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 .NET Framework 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.)

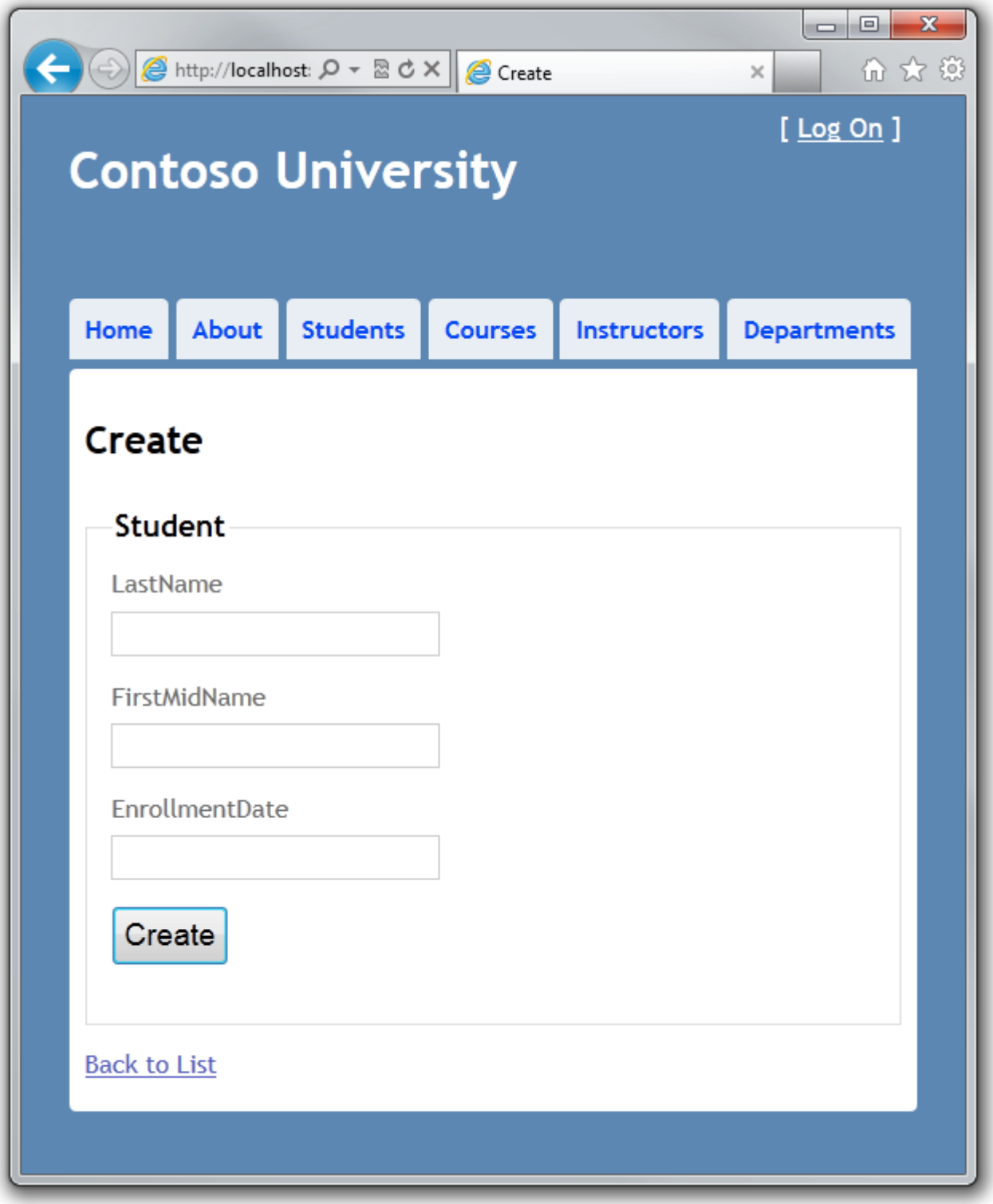
The `try-catch` block is the only difference between this code and what the automatic scaffolding created. If an exception that derives from `DataException` is caught while the changes are being saved, a generic error message is displayed. These kinds of errors are typically caused by something external to the application rather than a programming error, so the user is advised to try again. 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`. The following example shows the relevant code:

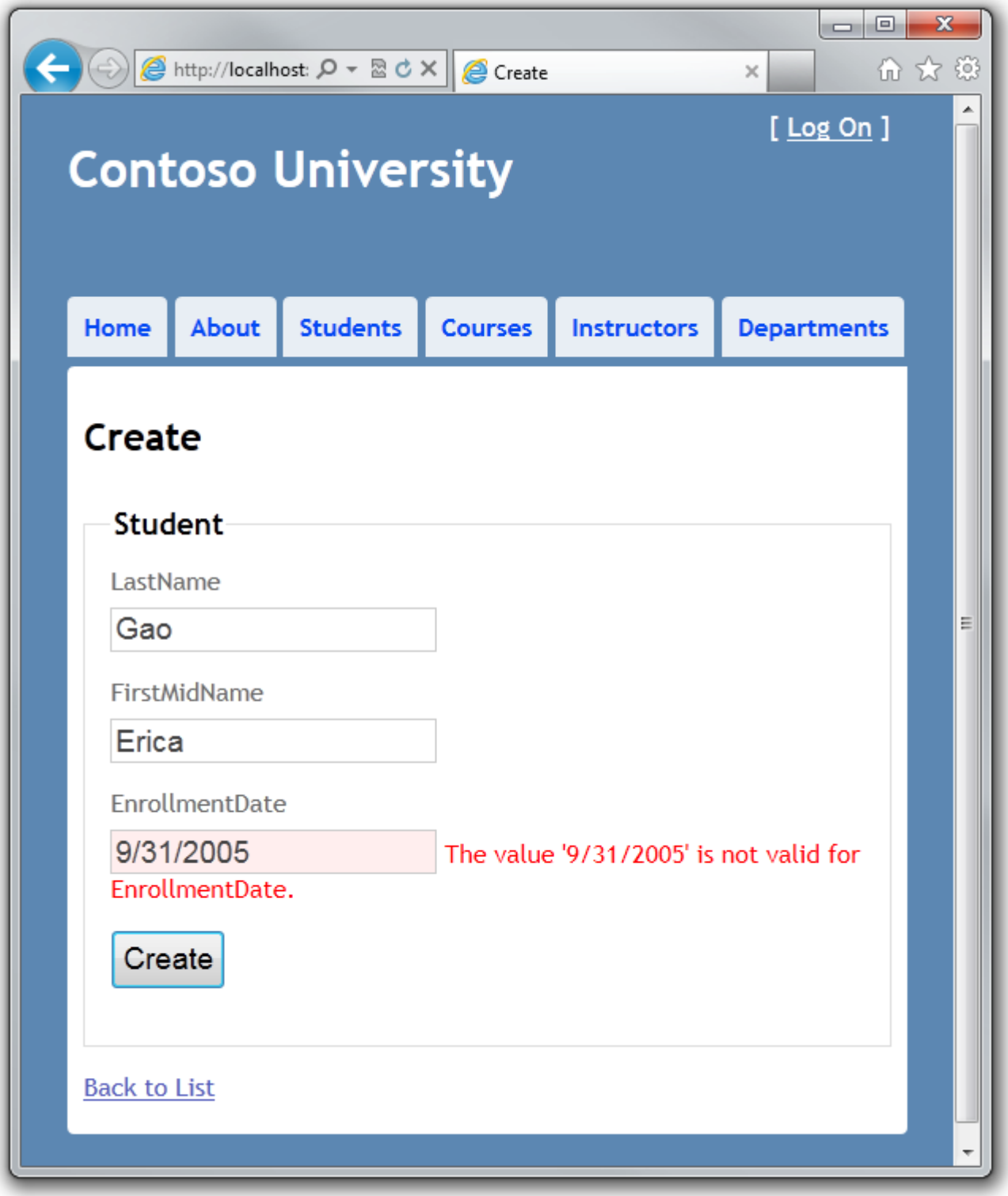
```
<divclass="editor-label">
    @Html.LabelFor(model => model.LastName)
</div>
<divclass="editor-field">
    @Html.EditorFor(model => model.LastName)
    @Html.ValidationMessageFor(model => model.LastName)
</div>
```

No changes are required in *Create.cshtml*.

Run the page by selecting the **Students** tab and clicking **Create New**.

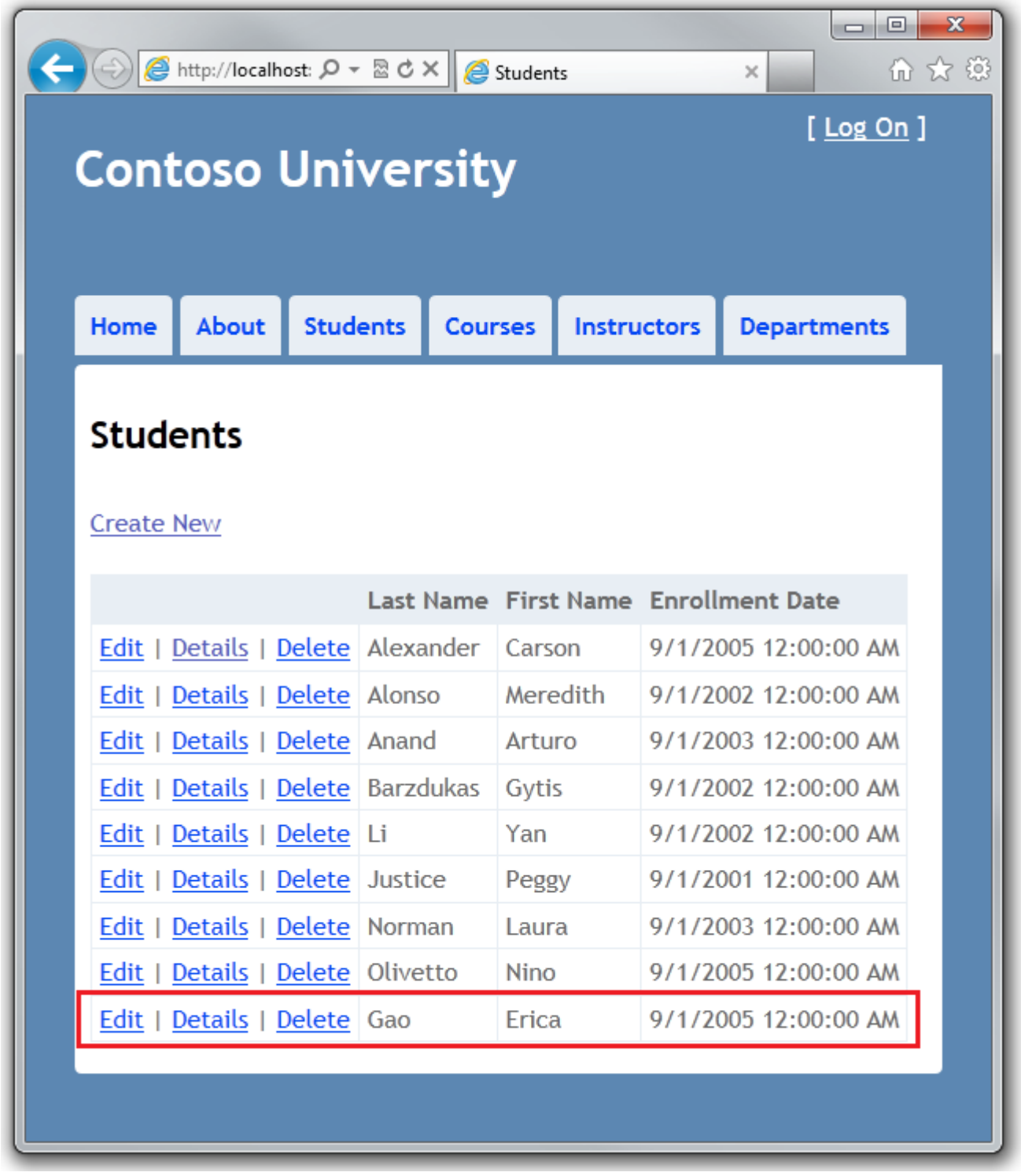


Data validation works by default. Enter names and an invalid date and click **Create** to see the error message.



In this case you're seeing client-side validation that's implemented using JavaScript. But server-side validation is also implemented. Even if client validation failed, bad data would be caught and an exception would be thrown in server code.

Change the date to a valid value such as 9/1/2005 and click **Create** to see the new student appear in the **Index** page.



Creating an Edit Page

In *Controllers\StudentController.cs*, the `HttpGetEdit` 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 `HttpPostEdit` action method with the following code to add a `try-catch` block:

```
[HttpPost]
publicActionResultEdit(Student student)
{
try
{
if(ModelState.IsValid)
{
            db.Entry(student).State=EntityState.Modified;
            db.SaveChanges();
returnRedirectToAction("Index");
}
}
catch(DataException)
{
//Log the error (add a variable name after DataException)
ModelState.AddModelError("","Unable to save changes. Try again, and if the problem
persists see your system administrator.");
}
returnView(student);
}
```

This code is similar to what you saw in the `HttpPostCreate` method. However, instead of adding the entity created by the model binder to the entity set, this code sets a flag on the entity that 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. (You will learn how to handle concurrency in the Handling Concurrency tutorial later in this series.)

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.

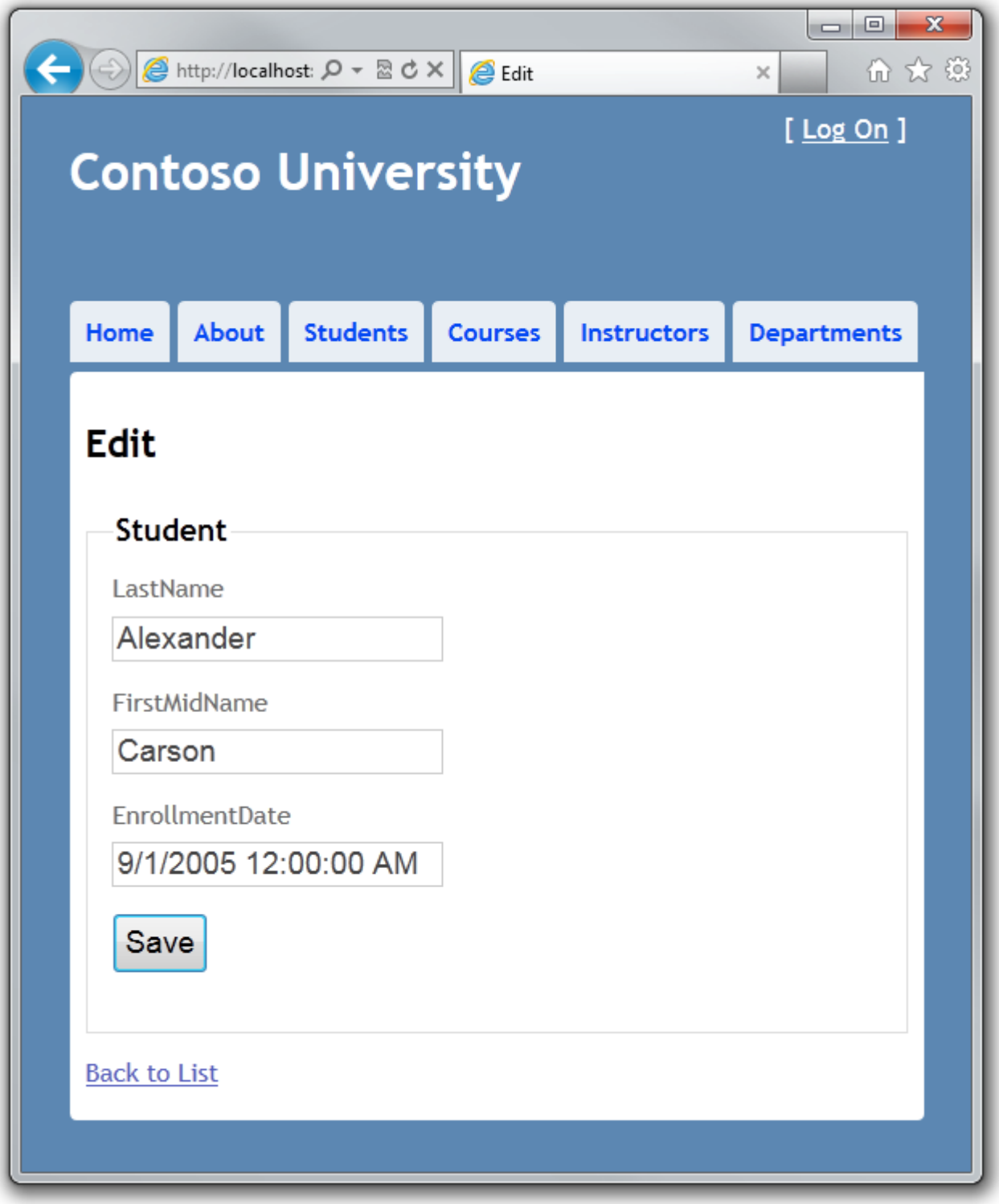
In a desktop application, state changes are typically set automatically. In this type of application, you read an entity and make changes to some of its property values. This causes its entity state to automatically be changed to `Modified`. Then when you call `SaveChanges`, the Entity Framework generates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

However, in a web application this sequence is interrupted, because the database context instance that reads an entity is disposed after a page is rendered. When the `HttpPostEdit` action method is called, this is the result of a new request and you have a new instance of the context, 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.

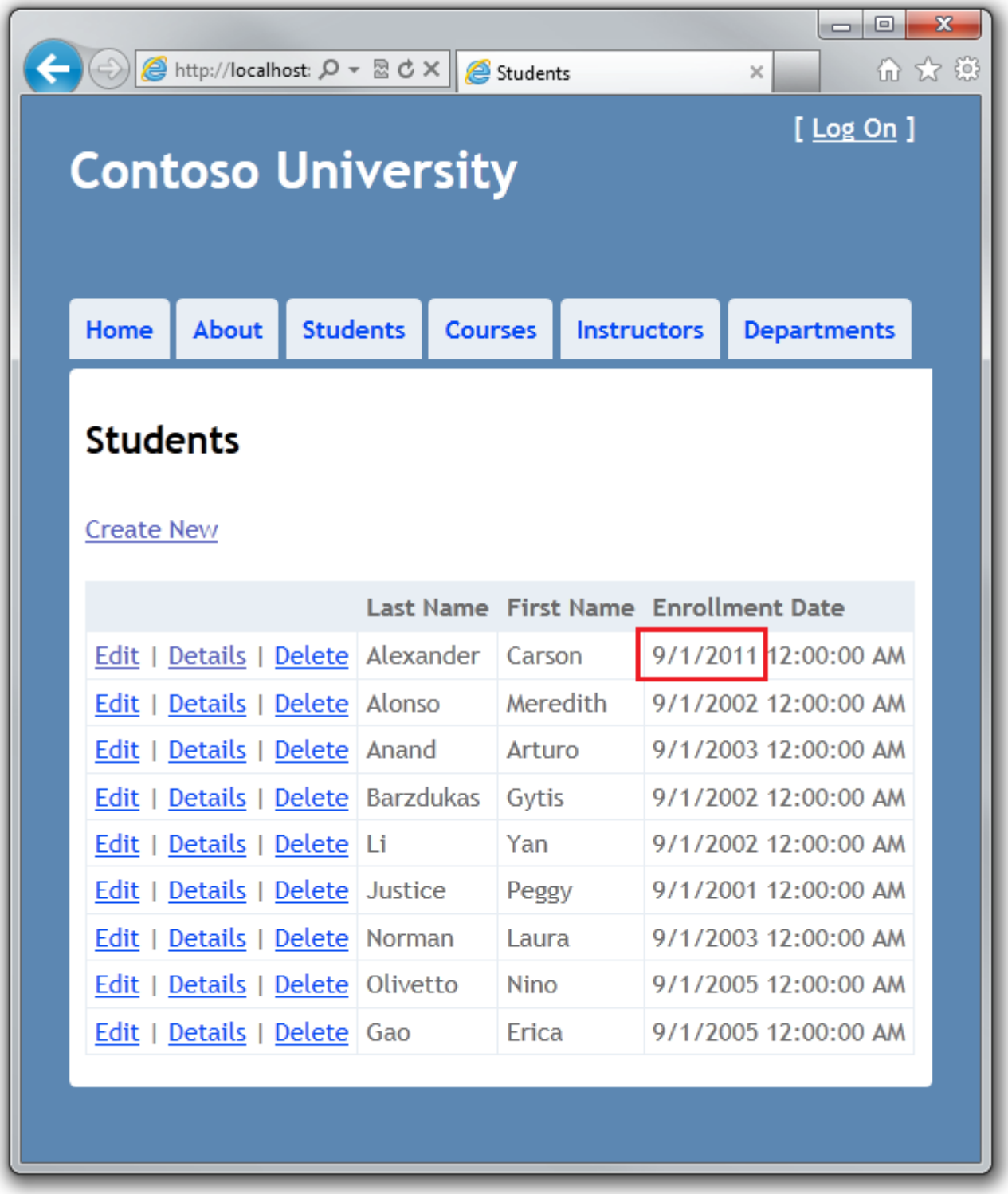
If you want the SQL `Update` statement to update only the fields that the user actually changed, you can save the original values in some way (such as hidden fields) so that they are available when the `HttpPostEdit` method is called. Then you can create a `Student` entity using the original values, 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 Add/Attach and Entity States and Local Data on the Entity Framework team blog.

The 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.



Creating a Delete Page

In *Controllers\StudentController.cs*, the template code for the `HttpGetDelete` 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 will 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 `HttpPostDelete` method is called and then that method actually performs the delete operation.

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

Replace the `HttpGetDelete` action method with the following code, which manages error reporting:

```
publicActionResultDelete(int id,bool? saveChangesError)
{
if(saveChangesError.GetValueOrDefault())
{
ViewBag.ErrorMessage="Unable to save changes. Try again, and if the problem persists
see your system administrator.";
}
returnView(db.Students.Find(id));
}
```

This code accepts an optional Boolean parameter that indicates whether it was called after a failure to save changes. This parameter is null (`false`) when the `HttpGetDelete` method is called in response to a page request. When it is called by the `HttpPostDelete` method in response to a database update error, the parameter is `true` and an error message is passed to the view.

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

```
[HttpPost,ActionName("Delete")]
publicActionResultDeleteConfirmed(int id)
{
```

```
try
{
Student student = db.Students.Find(id);
        db.Students.Remove(student);
        db.SaveChanges();
}
catch(DataException)
{
//Log the error (add a variable name after DataException)
returnRedirectToAction("Delete",
newSystem.Web.Routing.RouteValueDictionary{
{"id", id },
{"saveChangesError",true}});
}
returnRedirectToAction("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.

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 =newStudent(){StudentID= 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 `HttpGetDelete` method doesn't delete the data. Performing a delete operation in response to a GET request (or for that matter, performing an 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.

In *Views\Student\Delete.cshtml*, add the following code between the `h2` heading and the `h3` heading:

```
<pclass="error">@ViewBag.ErrorMessage</p>
```

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

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 Handling Concurrency tutorial later in this series.)

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 should see to it that the context instance is disposed. That is why you will find a Dispose method at the end of the `StudentController` class in *StudentController.cs*, as shown in the following example:

```
protectedoverridevoidDispose(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.

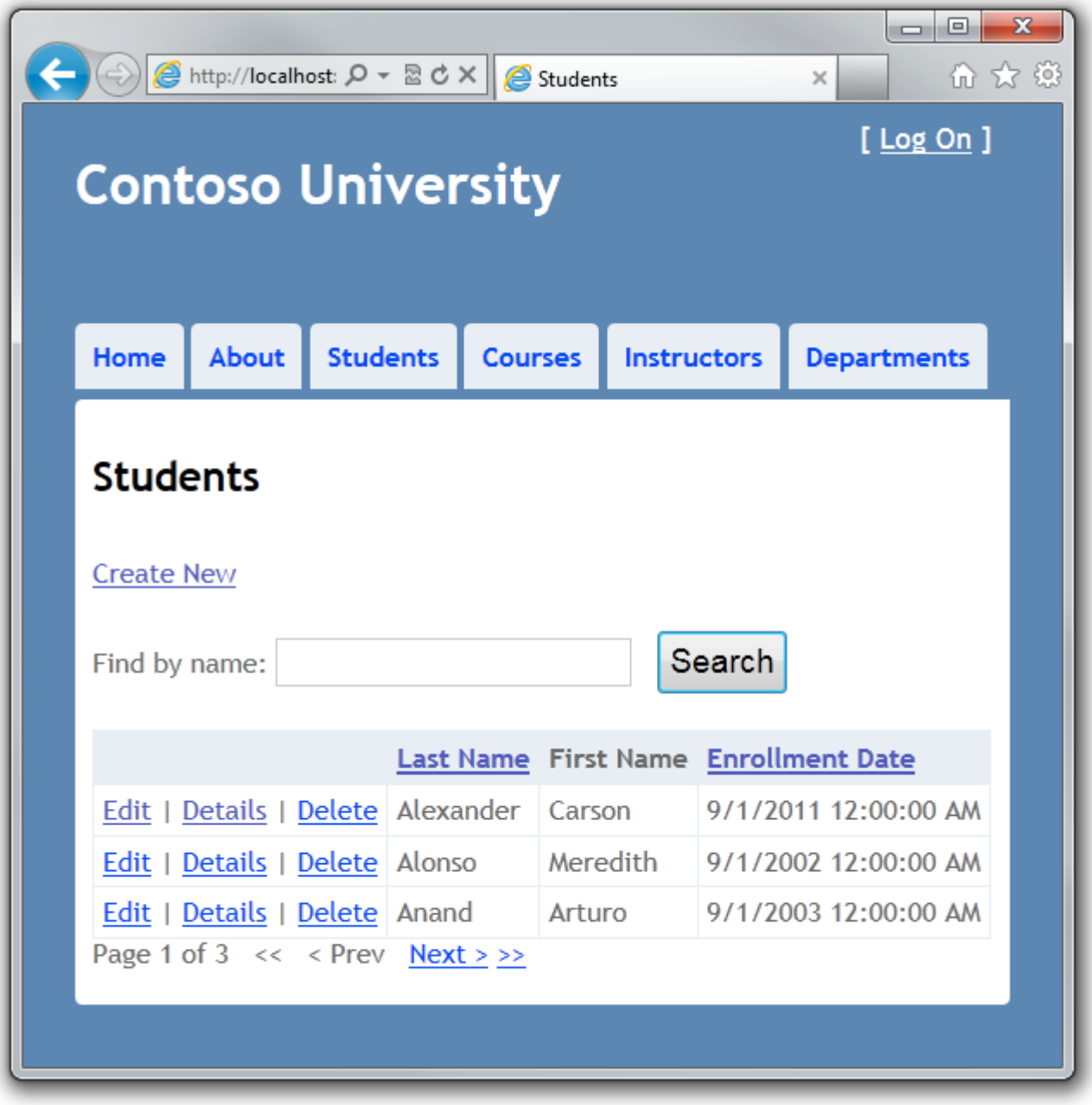
You now have a complete set of pages that perform simple CRUD operations for `Student` entities. In the next tutorial you'll expand the functionality of the Index page by adding sorting and paging.

Links to other Entity Framework resources can be found at the end of the last tutorial in this series.

Sorting, Filtering, and Paging

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.



Adding 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.

Adding Sorting Functionality to the Index Method

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

```
publicViewResultIndex(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;
}
returnView(students.ToList());
}
```

This code receives a `sortOrder` parameter from the query string in the URL, which 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 a space and the string "desc" to specify descending order.

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.

There are four possibilities, depending on how the data is currently sorted:

- If the current order is **Last Name** ascending, the **Last Name** link must specify **Last Name** descending, and the **Enrollment Date** link must specify **Date** ascending.
- If the current order is **Last Name** descending, the links must indicate **Last Name** ascending (that is, empty string) and **Date** ascending.
- If the current order is **Date** ascending, the links must indicate **Last Name** ascending and **Date** descending.
- If the current order is **Date** descending, the links must indicate **Last Name** ascending and **Date** 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.

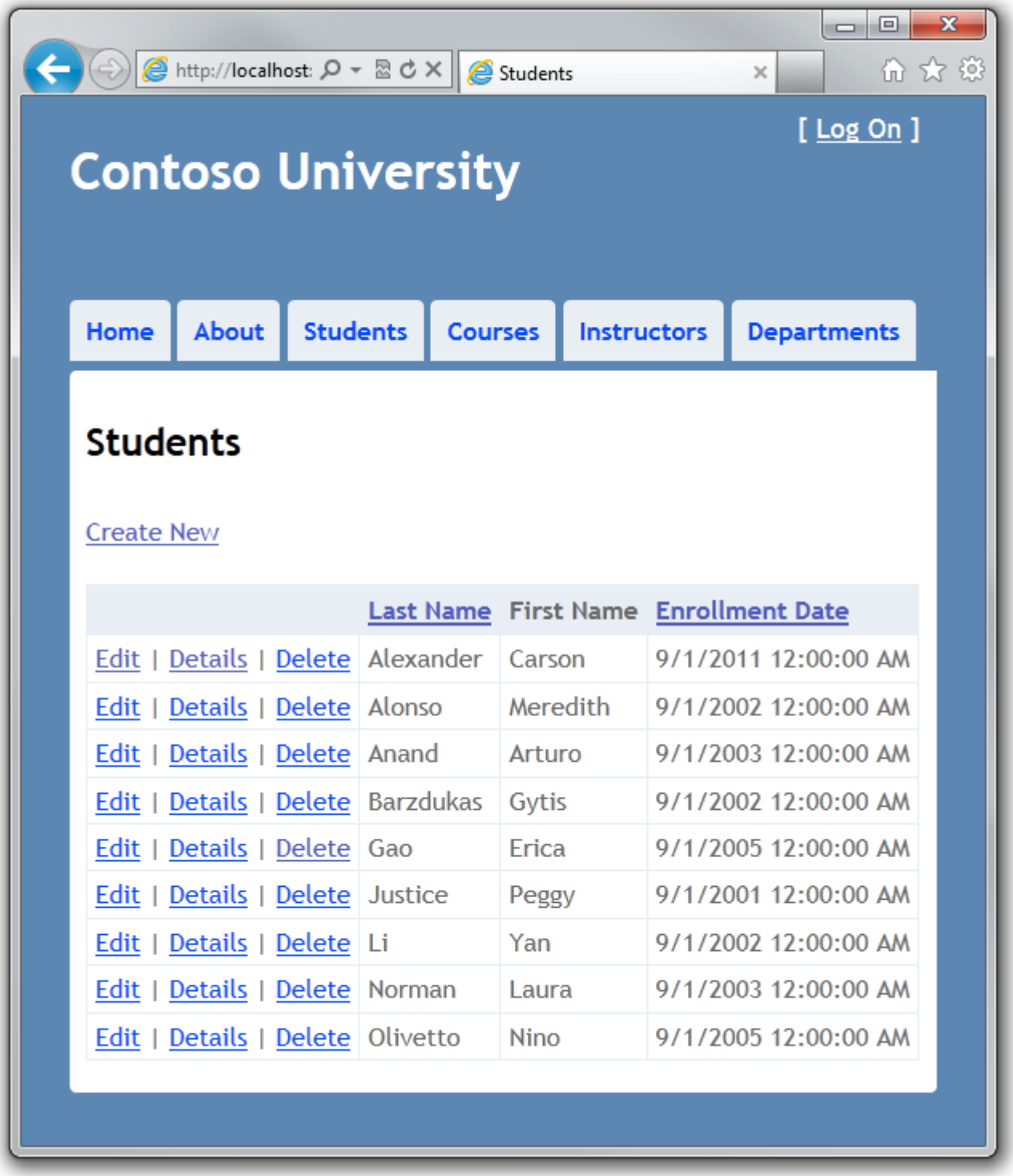
Adding Column Heading Hyperlinks to the Student Index View

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

```
<tr>
<th></th>
<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>
</tr>
```

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 column headings to verify that sorting works.



Adding a Search Box to the Students Index Page

To add filtering to the Student 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.

Adding Filtering Functionality to the Index Method

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

```
publicViewResultIndex(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;
}
```

```
returnView(students.ToList());
}
```

You've added a `searchString` parameter to the `Index` method. You've also added a `where` clause to the LINQ statement 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 later to the Index view.The statement that adds the `where` clause is executed only if there's a value to search for:

```
if(!String.IsNullOrEmpty(searchString))
{
    students = students.Where(s =>
s.LastName.ToUpper().Contains(searchString.ToUpper())
|| s.FirstMidName.ToUpper().Contains(searchString.ToUpper()));
}
```

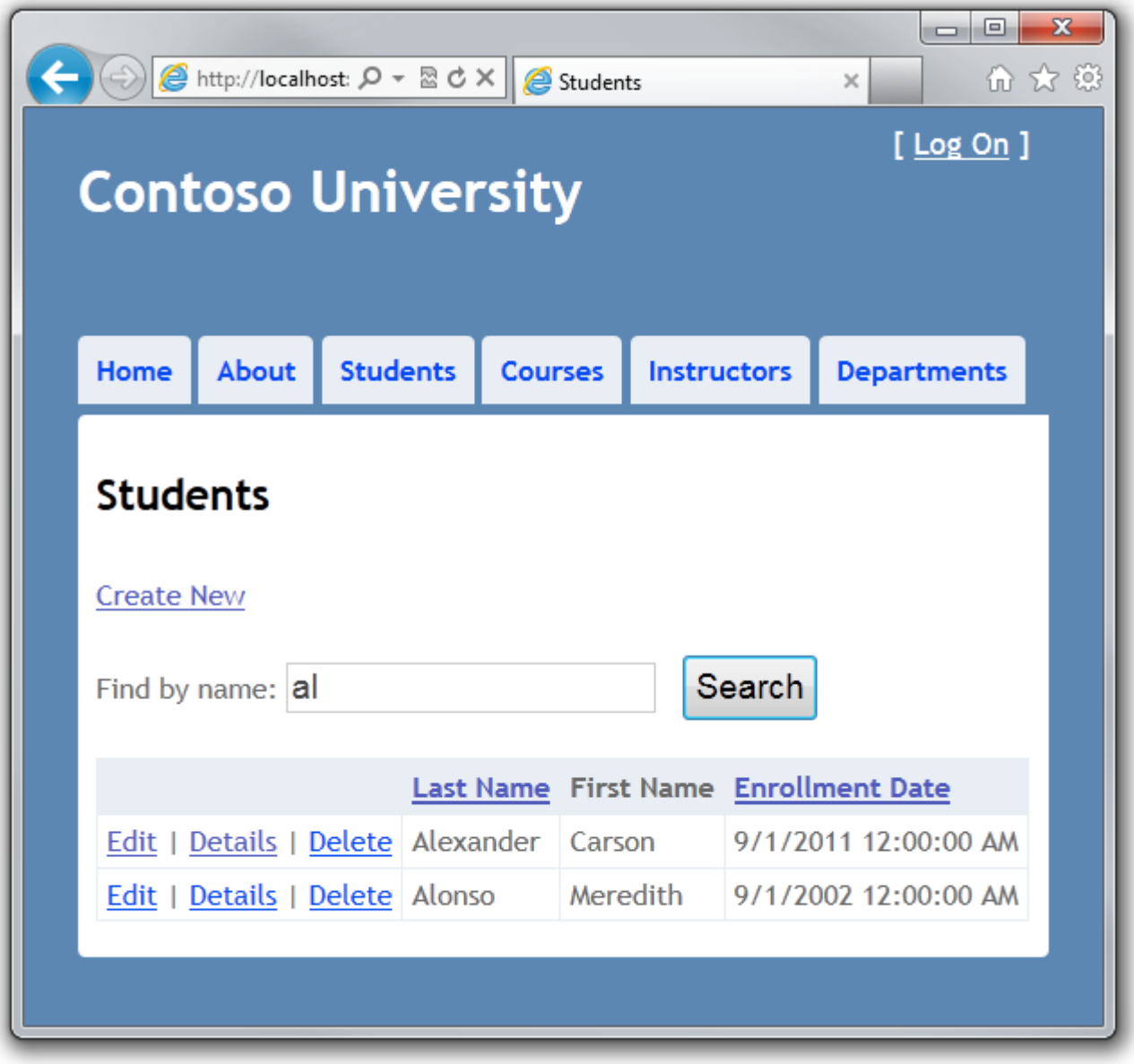
Note The .NET Framework implementation of the `Contains` method returns all rows when you pass an empty string to it, but the Entity Framework provider for SQL Server Compact 4.0 returns zero rows for empty strings. Therefore the code in the example (putting the `Where` statement inside an `if` statement) 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.)

Adding a Search Box to the Student Index View

In *Views\Student\Index.cshtml*, add a caption, a text box, and a **Search** button immediately before the opening `table` tag:

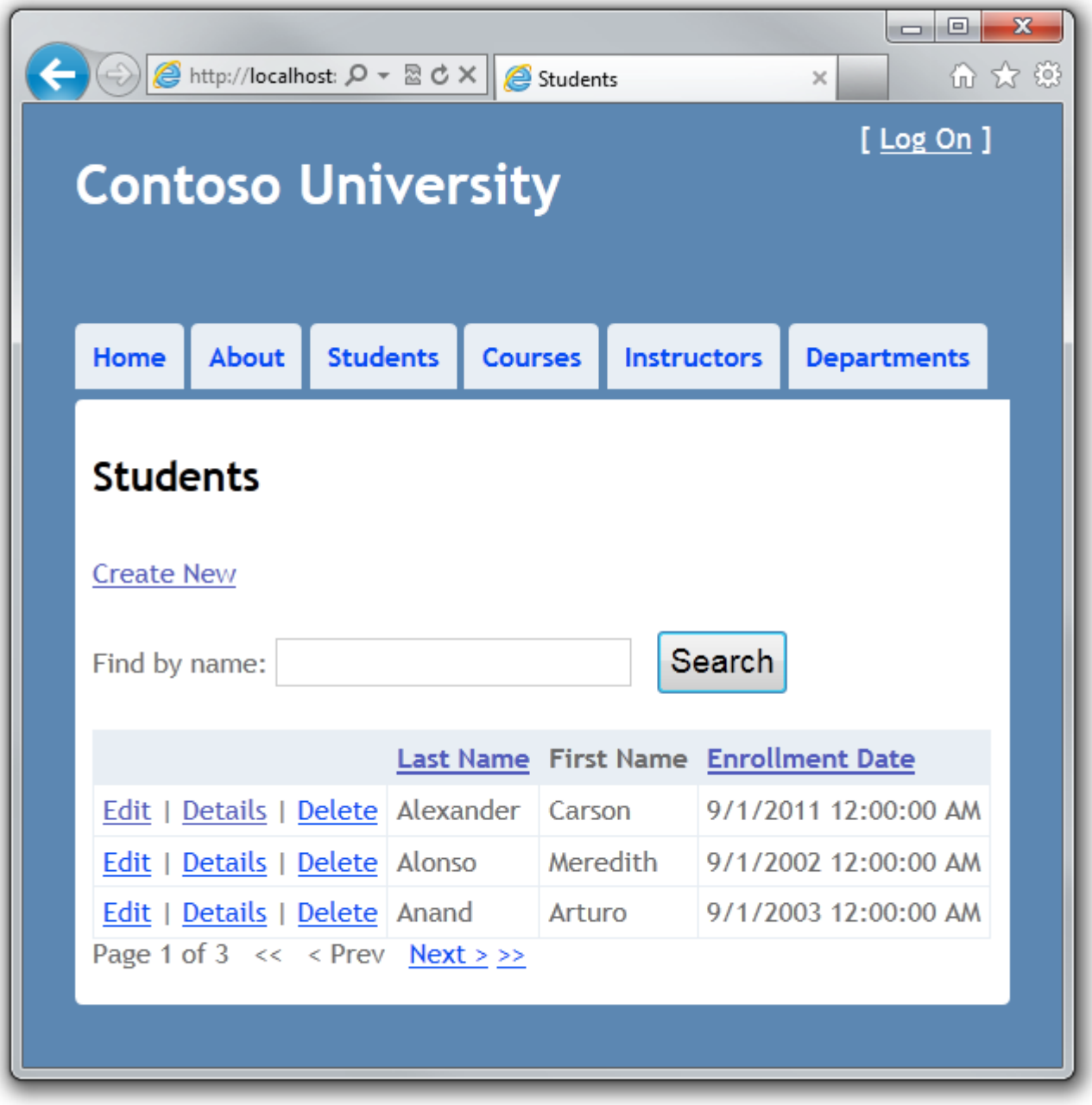
```
@using(Html.BeginForm())
{
<p>
Findby name:@Html.TextBox("SearchString")
<input type="submit" value="Search"/></p>
}
```

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



Adding Paging to the Students Index Page

To add paging to the Student Index page, you'll start by installing the **PagedList** NuGet package. Then you'll make additional changes in the `Index` method and add paging links to the `Index` view. The following illustration shows the paging links.

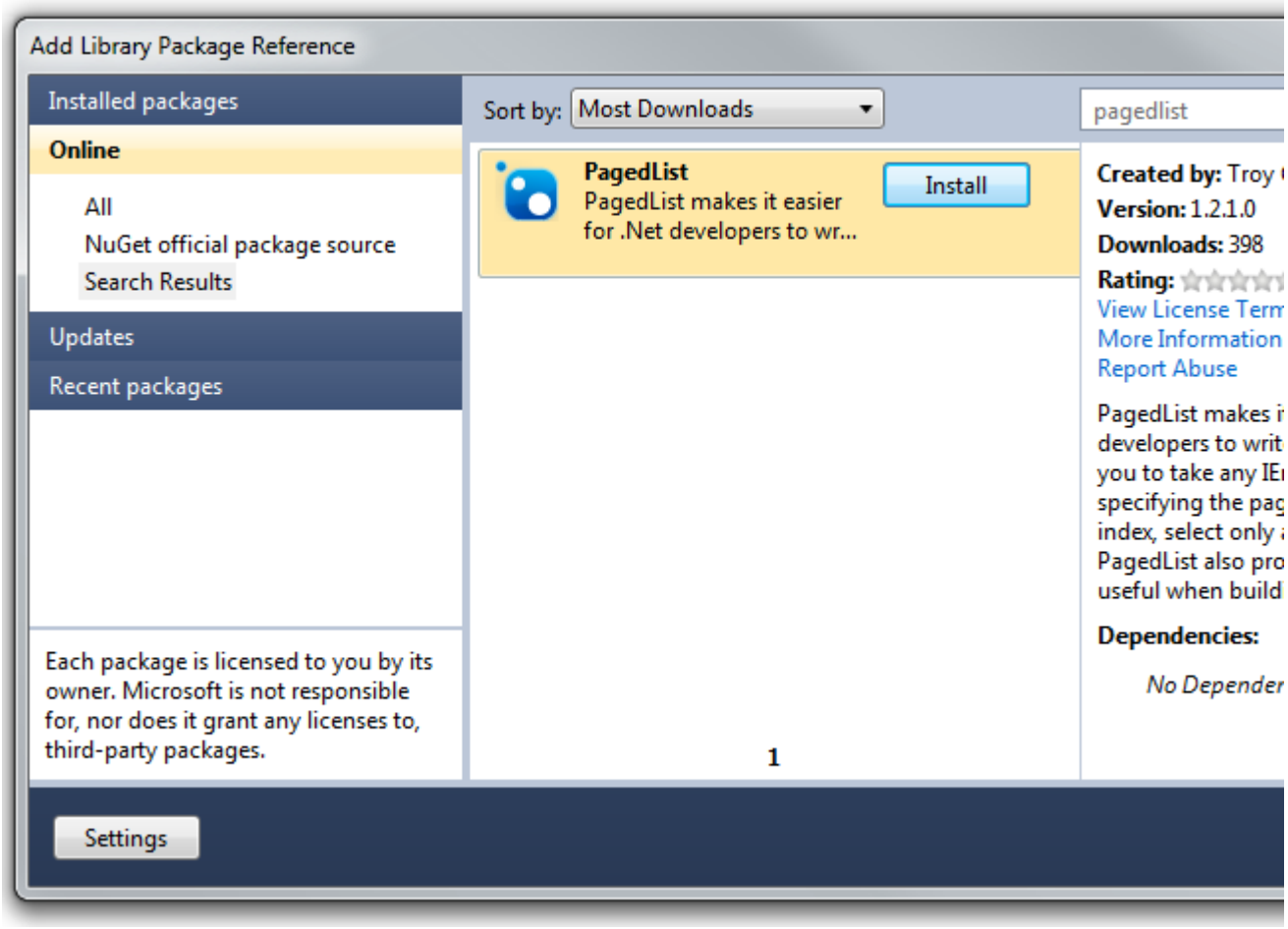


Installing the PagedList NuGet Package

The NuGet **PagedList** package installs a `PagedList` collection type. When you put query results in a `PagedList` collection, several properties and methods are provided that facilitate paging.

In Visual Studio, make sure the project (not the solution) is selected. From the **Tools** menu, select **Library Package Manager** and then **Add Library Package Reference**.

In the **Add Library Package Reference** dialog box, click the **Online** tab on the left and then enter "pagedlist" in the search box. When you see the **PagedList** package, click **Install**.



Adding Paging Functionality to the Index Method

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

```
usingPagedList;
```

Replace the `Index` method with the following code:

```
publicViewResultIndex(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(Request.HttpMethod=="GET")
{
                searchString = currentFilter;
}
else
{
                page =1;
}
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:
                    students = students.OrderBy(s => s.LastName);
break;
}

int pageSize =3;
```

```
int pageNumber =(page ??1);
returnView(students.ToPagedList(pageNumber, pageSize));
}
```

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

```
publicViewResultIndex(string sortOrder,string currentFilter,string searchString,int?
page)
```

The first time the page is displayed, or if the user hasn't clicked a paging link, the `page` variable is 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 `ViewBag` property provides the view with the current filter string, because this string must be restored to the text box when the page is redisplayed. In addition, the string must be included in the paging links in order to maintain the filter settings during paging. Finally, 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, hence the original page might not even exist anymore.

```
if(Request.HttpMethod=="GET")
{
                searchString = currentFilter;
}
else
{
                page =1;
}
ViewBag.CurrentFilter= searchString;
```

At the end of the method, the student query is converted to a `PagedList` instead of to a `List` so that it will be passed to the view in a collection that supports paging. This is the code:

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

The `ToPagedList` method takes a page number value. The two question marks represent an operator that 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.

Adding Paging Links to the Student Index View

In *Views\Student\Index.cshtml*, replace the existing code with the following code:

```
@modelPagedList.IPagedList<ContosoUniversity.Models.Student>

@{
ViewBag.Title="Students";
}

<h2>Students</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
@using(Html.BeginForm())
{
<p>
Findby name:@Html.TextBox("SearchString",ViewBag.CurrentFilterasstring)
<input type="submit" value="Search"/></p>
}
<table>
<tr>
<th></th>
<th>
        @Html.ActionLink("Last Name", "Index", new { sortOrder=ViewBag.NameSortParm,
```

```
currentFilter=ViewBag.CurrentFilter })
</th>
<th>
FirstName
</th>
<th>
        @Html.ActionLink("Enrollment Date", "Index", new { sortOrder =
ViewBag.DateSortParm, currentFilter = ViewBag.CurrentFilter })
</th>
</tr>

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

</table>

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

    @if (Model.HasPreviousPage)
    {
```

```
        @Html.ActionLink("<<", "Index", new { page = 1, sortOrder =
ViewBag.CurrentSort, currentFilter=ViewBag.CurrentFilter  })
        @Html.Raw(" ");
        @Html.ActionLink("< Prev", "Index", new { page = Model.PageNumber - 1,
sortOrder = ViewBag.CurrentSort, currentFilter=ViewBag.CurrentFilter  })
    }
    else
    {
        @:<<
        @Html.Raw(" ");
        @:< Prev
    }

    @if (Model.HasNextPage)
    {
        @Html.ActionLink("Next >", "Index", new { page = Model.PageNumber + 1,
sortOrder = ViewBag.CurrentSort, currentFilter=ViewBag.CurrentFilter  })
        @Html.Raw(" ");
        @Html.ActionLink(">>", "Index", new { page = Model.PageCount, sortOrder =
ViewBag.CurrentSort, currentFilter=ViewBag.CurrentFilter  })
    }
    else
    {
        @:Next >
        @Html.Raw(" ")
        @:>>
    }
</div>
```

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

The text box is initialized with the current search string so that the user can page through filter results without the search string disappearing:

```
Findby name:@Html.TextBox("SearchString",ViewBag.CurrentFilterasstring)
```

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})
```

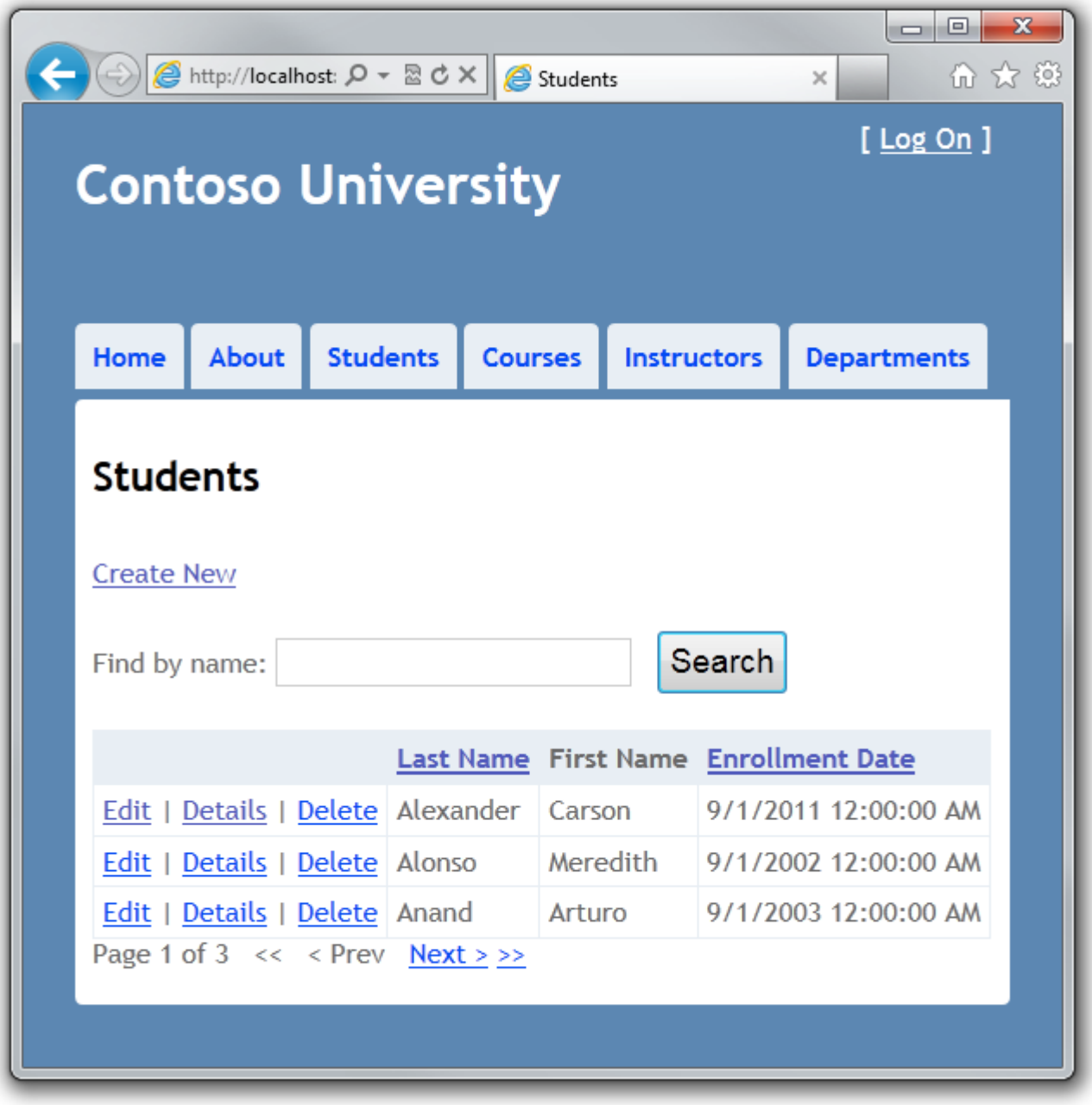
On one line at the bottom of the page, this code displays the following navigation UI:

Page *[current page number]* of *[total number of pages]*<<< Prev Next >>>

The **<<** symbol is a link to the first page, **< Prev** is a link to the previous page, and so on. If the user is currently on page 1, the links to move backward are disabled; similarly, if the user is on the last page, the links to move forward are disabled. Each paging link passes the new page number and the current sort order and search string to the controller in the query string. This lets you maintain the sort order and filter results during paging.

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.)

Run the page.



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.

Creating 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.

Creating the View Model

Create a *ViewModels* folder. In that folder, create *EnrollmentDateGroup.cs* and replace the existing code with the following code:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.ComponentModel.DataAnnotations;

namespaceContosoUniversity.ViewModels
{
publicclassEnrollmentDateGroup
{
[DisplayFormat(DataFormatString="{0:d}")]
publicDateTime?EnrollmentDate{get;set;}

publicintStudentCount{get;set;}
}
}
```

Modifying the Home Controller

In *HomeController.cs*, add the following `using` statements:

```
usingContosoUniversity.DAL;
usingContosoUniversity.Models;
usingContosoUniversity.ViewModels;
```

Add a class variable for the database context:

```
privateSchoolContext db =newSchoolContext();
```

Replace the `About` method with the following code:

```
publicActionResultAbout()
{
var data =from student in db.Students
group student by student.EnrollmentDateinto dateGroup
selectnewEnrollmentDateGroup()
{
EnrollmentDate= dateGroup.Key,
StudentCount= dateGroup.Count()
};
returnView(data);
}
```

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:

```
protectedoverridevoidDispose(bool disposing)
{
    db.Dispose();
base.Dispose(disposing);
}
```

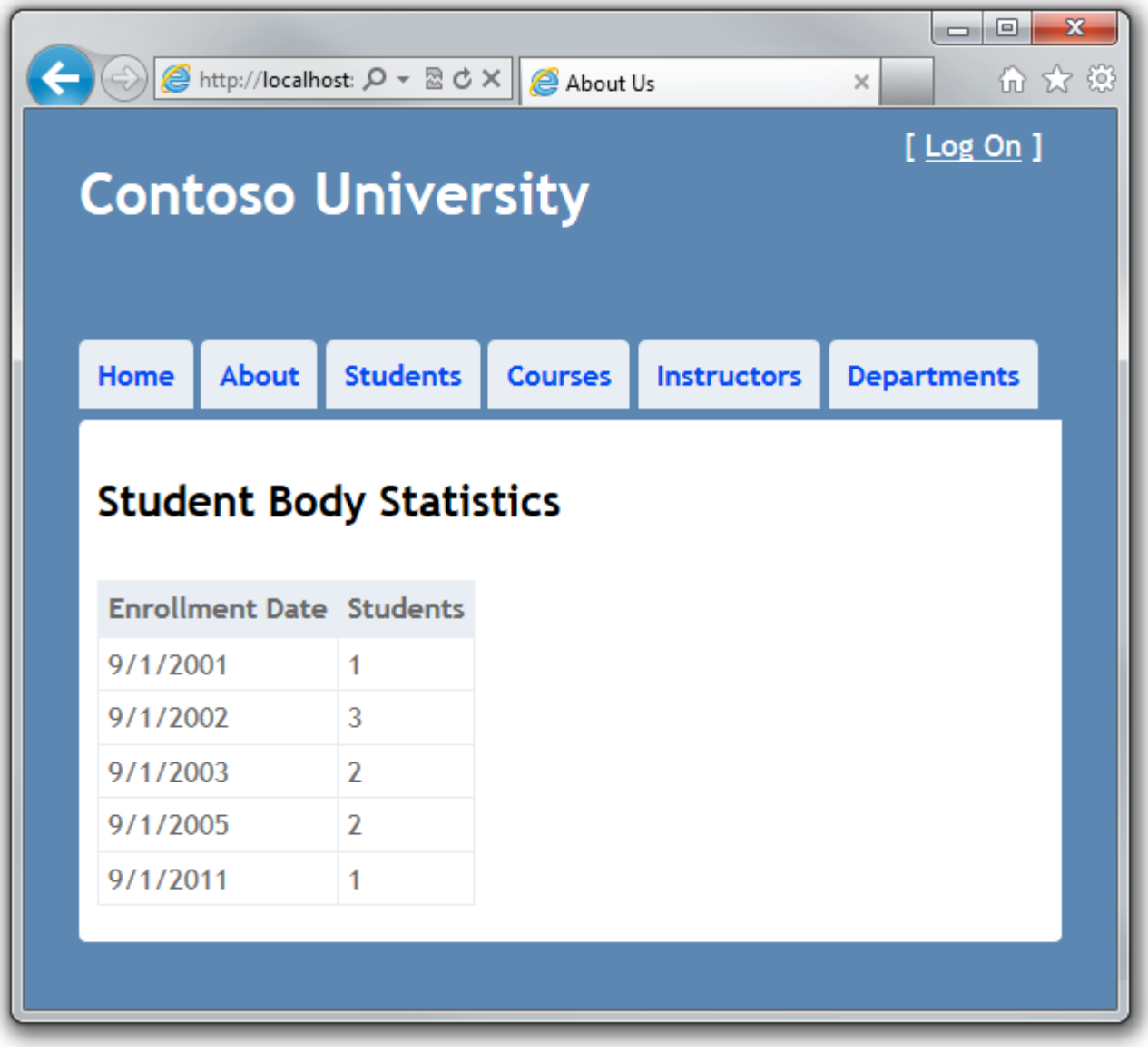
Modifying the About View

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

```
@modelIEnumerable<ContosoUniversity.ViewModels.EnrollmentDateGroup>

@{
ViewBag.Title="Student Body Statistics";
}

<h2>StudentBodyStatistics</h2>
```

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

@foreach(var item inModel){
<tr>
<td>
@String.Format("{0:d}", item.EnrollmentDate)
</td>
<td>
            @item.StudentCount
</td>
</tr>
}
</table>
```

Run the page. The count of students for each enrollment date is displayed in a table.



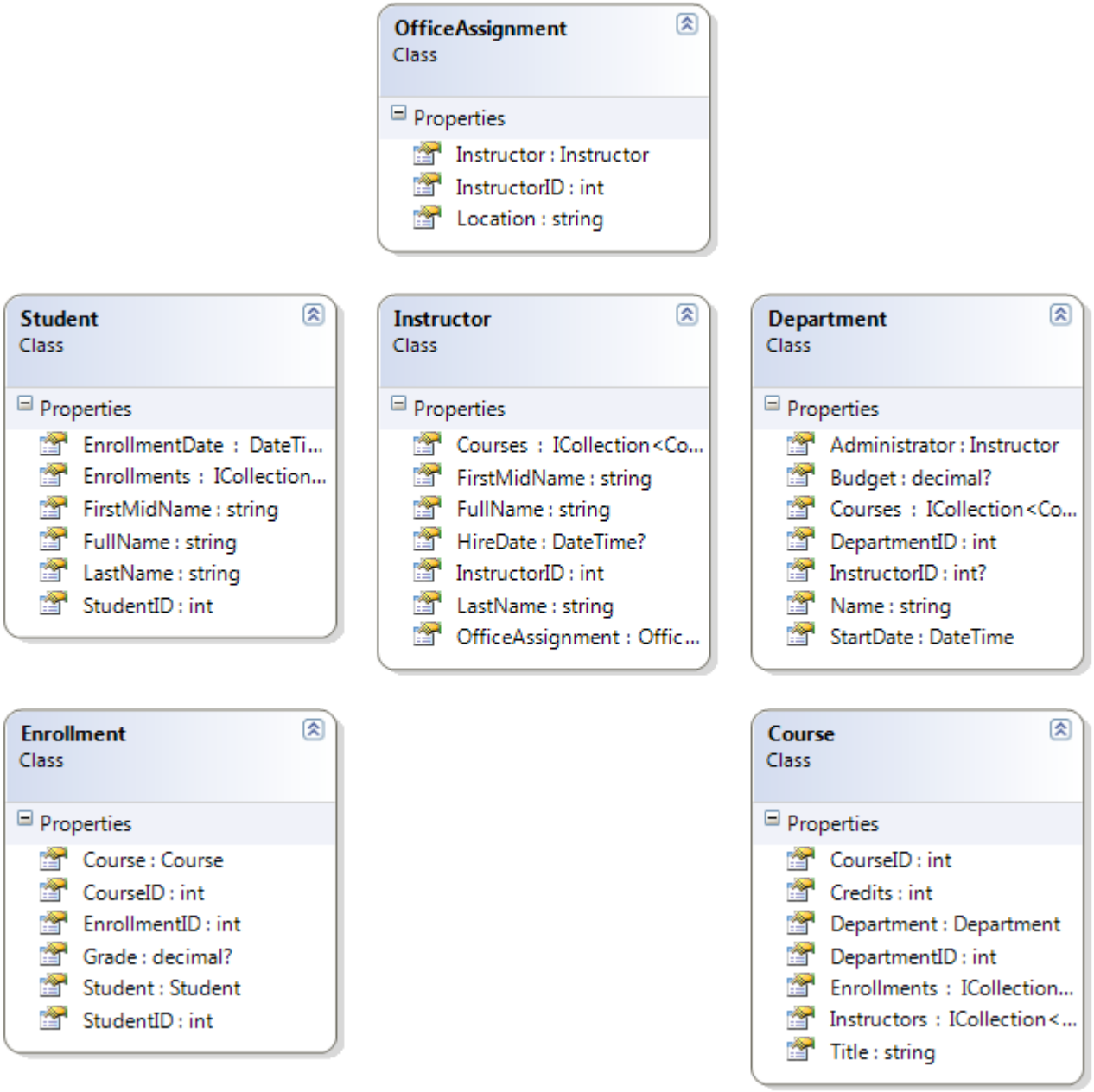
You've now seen how to create a data model and implement basic CRUD, sorting, filtering, paging, and grouping functionality. In the next tutorial you'll begin looking at more advanced topics by expanding the data model.

Links to other Entity Framework resources can be found at the end of the last tutorial in this series.

Creating a More Complex Data Model

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 make use of data annotation attributes to control the behavior of your model classes.

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



Using Attributes to Control Formatting, Validation, and Database Mapping

In this section you'll see examples of attributes you can add to model classes to specify formatting, validation, and database mapping. Then in 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 DisplayFormat Attribute

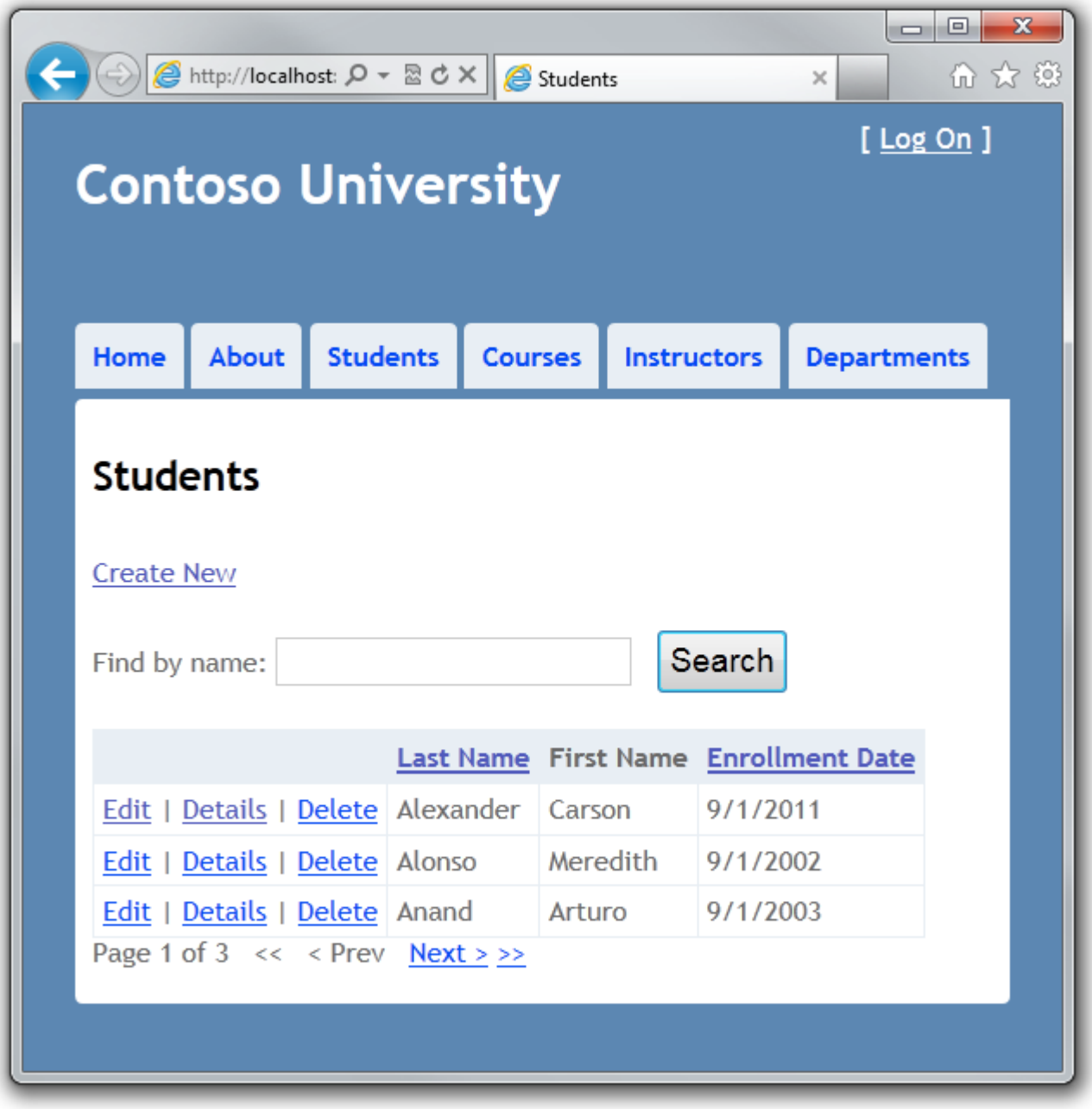
For student enrollment dates, all of the web pages currently display the time along with the date, although all you care about for this field is the date. By using data annotation attributes, you can make one code change that will fix the display format everywhere. To see an example of that, you'll add 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 a `DisplayFormat` attribute to the `EnrollmentDate` property, as shown in the following example:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.ComponentModel.DataAnnotations;

namespaceContosoUniversity.Models
{
publicclassStudent
{
publicintStudentID{get;set;}
publicstringLastName{get;set;}
publicstringFirstMidName{get;set;}

[DisplayFormat(DataFormatString="{0:d}",ApplyFormatInEditMode=true)]
publicDateTimeEnrollmentDate{get;set;}

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

The format string specifies that only a short date should be displayed for this property. The `ApplyFormatInEditMode` setting specifies that this 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.)

Run the Student Index page again and notice that times are no longer displayed for the enrollment dates. The same will be true if you run the other Student pages.



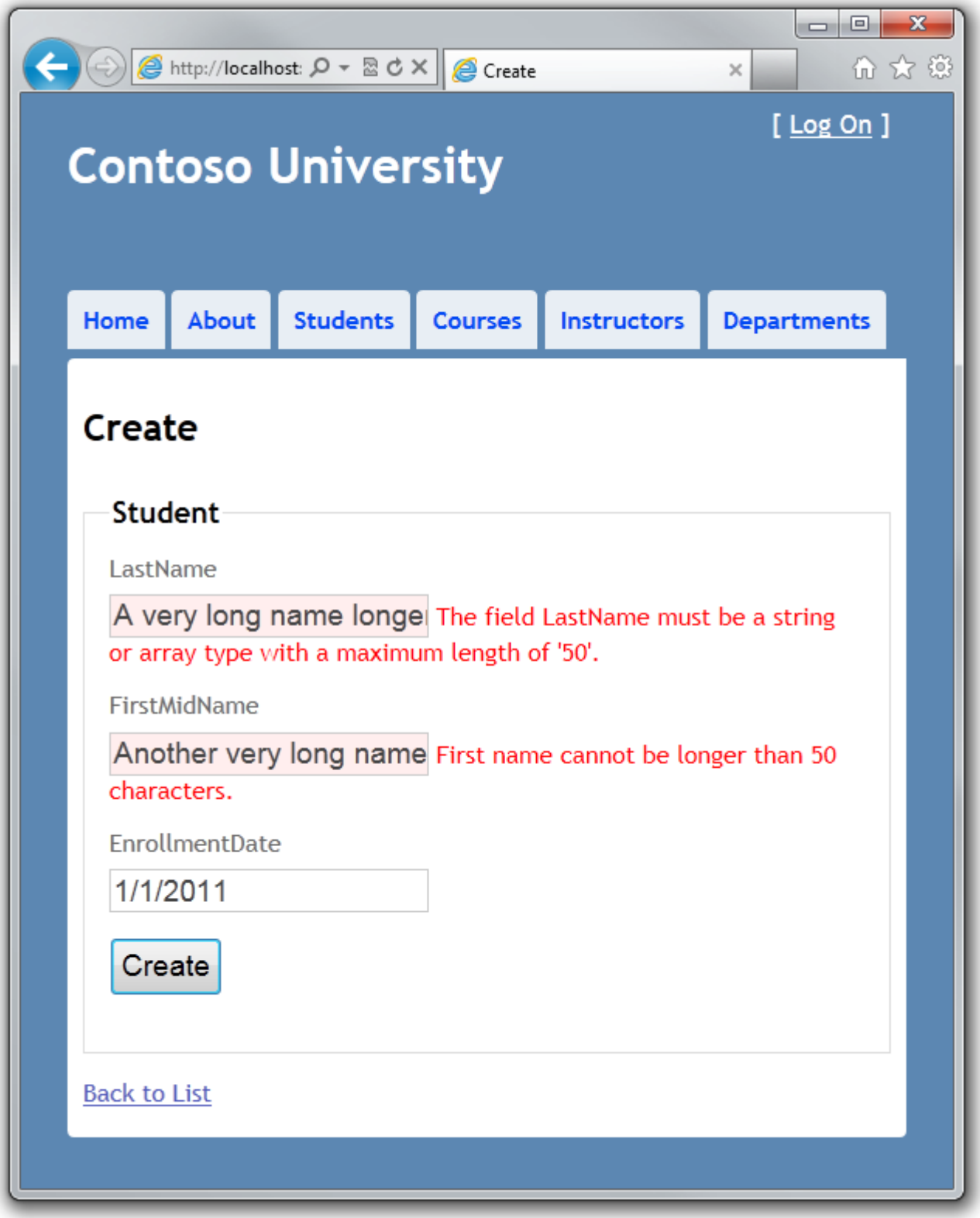
The MaxLength Attribute

You can also specify data validation rules and messages using attributes. Suppose you want to ensure that users don't enter more than 50 characters for a name. To add this limitation, add `Range` attributes to the `LastName` and `FirstMidName` properties, as shown in the following example:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.ComponentModel.DataAnnotations;

namespaceContosoUniversity.Models
{
publicclassStudent
{
publicintStudentID{get;set;}

[MaxLength(50)]
publicstringLastName{get;set;}

[MaxLength(50,ErrorMessage="First name cannot be longer than 50 characters.")]
publicstringFirstMidName{get;set;}

[DisplayFormat(DataFormatString="{0:d}",ApplyFormatInEditMode=true)]
publicDateTimeEnrollmentDate{get;set;}

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

If a user attempts to enter a last name that's too long, a default error message will be displayed. If a long first name is entered, the custom error message you specified will be displayed.

Run the Create page, enter two names longer than 50 characters, and click **Create** to see the error messages. (You'll have to enter a valid date in order to get past the date validation.)


http://localhost:
Create
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Create
Student
LastName
A very long name longe
The field LastName must be a string or array type with a maximum length of '50'.
FirstMidName
Another very long name
First name cannot be longer than 50 characters.
EnrollmentDate
1/1/2011
Create
Back to List

It's a good idea to always specify the maximum length for string properties. If you don't, when Code First creates the database, the corresponding columns will have the maximum length allowed for strings in the database, which would be an inefficient database table structure.

The Column Attribute

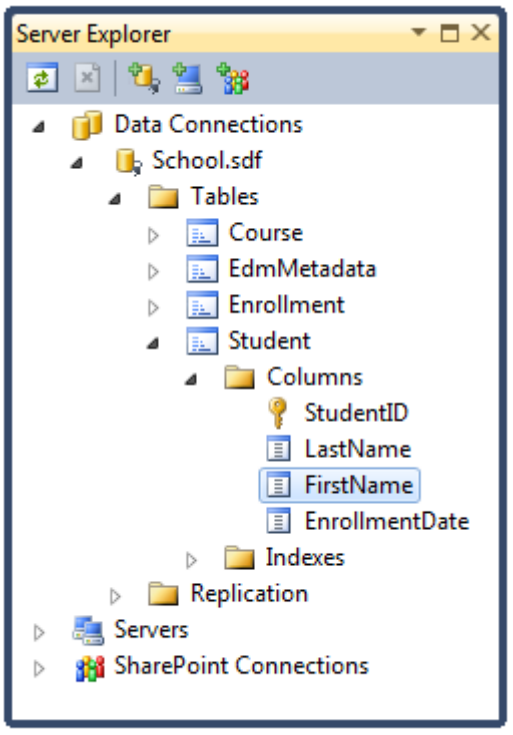
You can also use attributes to control how your classes and properties are mapped to the database. Suppose you had used the name `FirstMidName` for the first-name field because the field might also contain a middle name. But you want the database column to be named `FirstName`, because users who will 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 assumed to be the same as property names.)

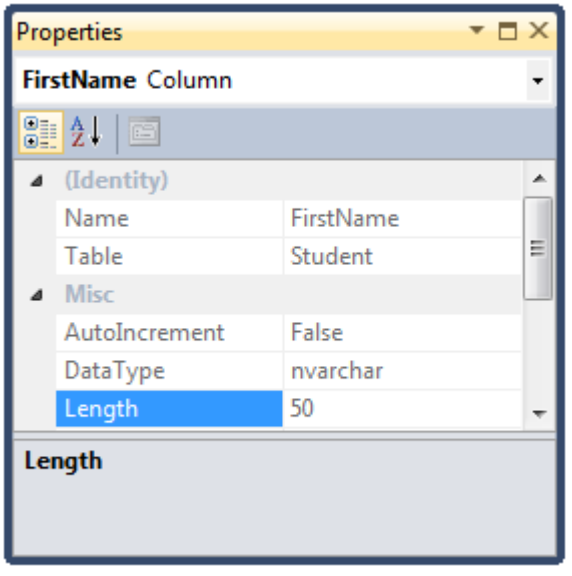
Add the column name attribute to the `FirstMidName` property, as shown in the following example:

```
[Column("FirstName")]
publicstringFirstMidName{get;set;}
```

Run the Student Index page again and you see that nothing has changed. (You can't just run the site and view the home page; you have to select the Student Index page because that causes the database to be accessed, which causes the database to be automatically dropped and re-created.) However, if you open the database in **Server Explorer** as you did earlier, you can expand the `Student` table to see that the column name is `FirstName`.



In the **Properties** window, you'll also notice that the name-related fields are defined as 50 characters in length, thanks to the `MaxLength` attributes you added earlier.

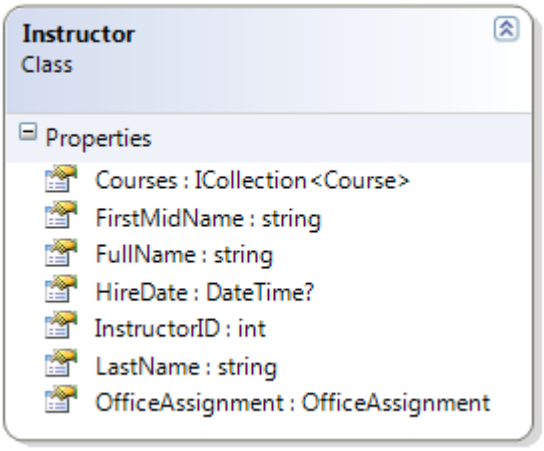


In most cases, you can also make mapping changes using method calls, as you'll see later in this tutorial.

In the following sections you'll make more use of data annotations attributes as you expand the `School` data model. In each section you'll create a class for an entity or modify a class that you created in the first tutorial.

Note If you try to compile before you finish creating all of these entity classes, you might get compiler errors.

Creating the Instructor Entity



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

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.ComponentModel.DataAnnotations;

namespaceContosoUniversity.Models
{
publicclassInstructor
{
publicInt32InstructorID{get;set;}

[Required(ErrorMessage="Last name is required.")]
[Display(Name="Last Name")]
[MaxLength(50)]
publicstringLastName{get;set;}

[Required(ErrorMessage="First name is required.")]
[Column("FirstName")]
[Display(Name="First Name")]
```

```
[MaxLength(50)]
publicstringFirstMidName{get;set;}

[DisplayFormat(DataFormatString="{0:d}",ApplyFormatInEditMode=true)]
[Required(ErrorMessage="Hire date is required.")]
[Display(Name="Hire Date")]
publicDateTime?HireDate{get;set;}

publicstringFullName
{
get
{
returnLastName+", "+FirstMidName;
}
}

publicvirtualICollection<Course>Courses{get;set;}
publicvirtualOfficeAssignmentOfficeAssignment{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 using inheritance to eliminate this redundancy.

The Required and Display Attributes

The attributes on the `LastName` property specify that it's a required field, that the caption for the text box should be "Last Name" (instead of the property name, which would be "LastName" with no space), and that the value can't be longer than 50 characters.

```
[Required(ErrorMessage="Last name is required.")]
[Display(Name="Last Name")]
[MaxLength(50)]
publicstringLastName{get;set;}
```

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.

```
publicstringFullName
{
get
{
returnLastName+", "+FirstMidName;
}
}
```

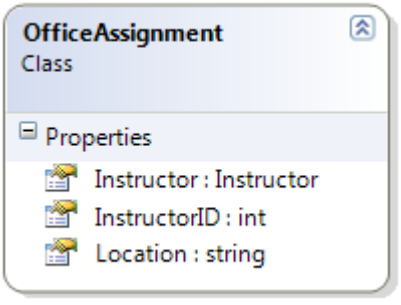
The Courses and OfficeAssignment Navigation Properties

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 be `ICollection`.

An instructor can teach any number of courses, so `Courses` is defined as a collection of `Course` entities. On the other hand, an instructor can only have one office, so `OfficeAssignment` is defined as a single `OfficeAssignment` entity (which may be null if no office is assigned).

```
publicvirtualICollection<Course>Courses{get;set;}
publicvirtualOfficeAssignmentOfficeAssignment{get;set;}
```

Creating the OfficeAssignment Entity



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

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.ComponentModel.DataAnnotations;

namespaceContosoUniversity.Models
{
publicclassOfficeAssignment
{
[Key]
publicintInstructorID{get;set;}

[MaxLength(50)]
[Display(Name="Office Location")]
publicstringLocation{get;set;}

publicvirtualInstructorInstructor{get;set;}
}
}
```

The Key Attribute

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

```
[Key]
publicintInstructorID{get;set;}
```

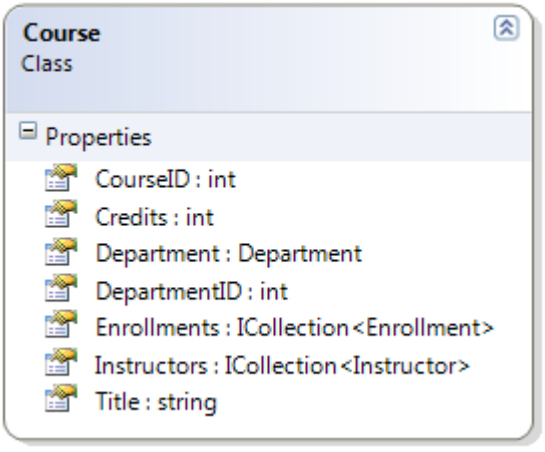
You can also use the `Key` attribute if the entity does have its own primary key but you want to name the property something different than *`classname`*`ID` or `ID`.

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). When an `Instructor` entity has a related `OfficeAssignment` entity, each entity will have a reference to the other one in its navigation property.

Modifying the Course Entity



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

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.ComponentModel.DataAnnotations;

namespaceContosoUniversity.Models
{
publicclassCourse
{
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name="Number")]
publicintCourseID{get;set;}

[Required(ErrorMessage="Title is required.")]
[MaxLength(50)]
publicstringTitle{get;set;}

[Required(ErrorMessage="Number of credits is required.")]
[Range(0,5,ErrorMessage="Number of credits must be between 0 and 5.")]
publicintCredits{get;set;}
```

```
[Display(Name="Department")]
publicintDepartmentID{get;set;}

publicvirtualDepartmentDepartment{get;set;}
publicvirtualICollection<Enrollment>Enrollments{get;set;}
publicvirtualICollection<Instructor>Instructors{get;set;}
}
}
```

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")]
publicintCourseID{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:

  ```
  publicintDepartmentID{get;set;}
  publicvirtualDepartmentDepartment{get;set;}
  ```

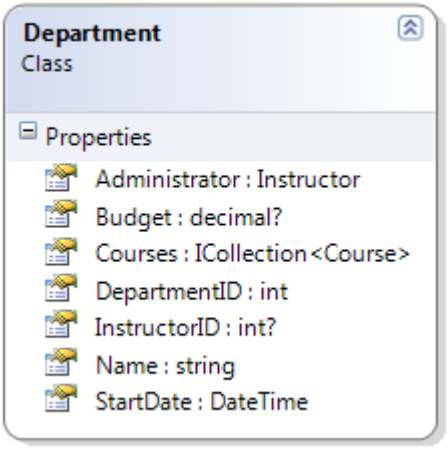
- A course can have any number of students enrolled in it, so there's an `Enrollments` navigation property:

  ```
  publicvirtualICollectionEnrollments{get;set;}
  ```

- A course may be taught by multiple instructors, so there's an `Instructors` navigation property:

```
publicvirtualICollection<Instructor>Instructors{get;set;}
```

Creating the Department Entity



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

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.ComponentModel.DataAnnotations;

namespaceContosoUniversity.Models
{
publicclassDepartment
{
publicintDepartmentID{get;set;}

[Required(ErrorMessage="Department name is required.")]
[MaxLength(50)]
publicstringName{get;set;}

[DisplayFormat(DataFormatString="{0:c}")]
[Required(ErrorMessage="Budget is required.")]
[Column(TypeName="money")]
publicdecimal?Budget{get;set;}
```

```
[DisplayFormat(DataFormatString="{0:d}",ApplyFormatInEditMode=true)]
[Required(ErrorMessage="Start date is required.")]
publicDateTimeStartDate{get;set;}

[Display(Name="Administrator")]
publicint?InstructorID{get;set;}

publicvirtualInstructorAdministrator{get;set;}
publicvirtualICollection<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")]
publicdecimal?Budget{get;set;}
```

This is normally not required, because the Entity Framework chooses the appropriate SQL Server data type based on the CLR type that you define for the property. The CLR `decimal` type would normally map to a SQL Server `decimal` type. But in this case you know that the column will be holding currency amounts, and the `money` data type is more appropriate for that.

Foreign Key and Navigation Properties

The foreign key and navigation properties reflect the following relationships:

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

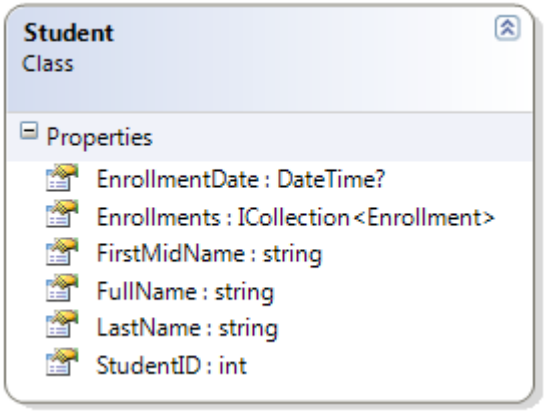
```
publicint?InstructorID{get;set;}
publicvirtualInstructorAdministrator{get;set;}
```

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

```
publicvirtualICollectionCourses{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 your initializer code runs. For example, if you didn't define the `Department.InstructorID` property as nullable, you'd get the following exception message when the initializer runs: "The referential relationship will result in a cyclical reference that's not allowed."

Modifying the Student Entity



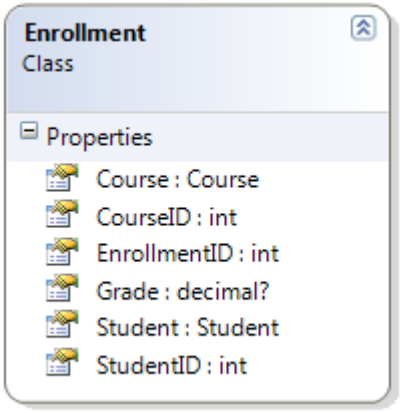
In *Models\Student.cs*, replace the code you added earlier with the following code:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.ComponentModel.DataAnnotations;

namespaceContosoUniversity.Models
{
publicclassStudent
{
publicintStudentID{get;set;}
```

```
[Required(ErrorMessage="Last name is required.")]
[Display(Name="Last Name")]
[MaxLength(50)]
publicstringLastName{get;set;}

[Required(ErrorMessage="First name is required.")]
[Column("FirstName")]
[Display(Name="First Name")]
[MaxLength(50)]
publicstringFirstMidName{get;set;}

[Required(ErrorMessage="Enrollment date is required.")]
[DisplayFormat(DataFormatString="{0:d}",ApplyFormatInEditMode=true)]
[Display(Name="Enrollment Date")]
publicDateTime?EnrollmentDate{get;set;}

publicstringFullName
{
get
{
returnLastName+", "+FirstMidName;
}
}

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

This code just adds attributes that you've now already seen in the other classes.

Modifying the Enrollment Entity



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

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.ComponentModel.DataAnnotations;

namespaceContosoUniversity.Models
{
publicclassEnrollment
{
publicintEnrollmentID{get;set;}

publicintCourseID{get;set;}

publicintStudentID{get;set;}

[DisplayFormat(DataFormatString="{0:#.#}",ApplyFormatInEditMode=true,NullDisplayText=
"No grade")]
publicdecimal?Grade{get;set;}

publicvirtualCourseCourse{get;set;}
publicvirtualStudentStudent{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:

```
publicintCourseID{get;set;}
publicvirtualCourseCourse{get;set;}
```

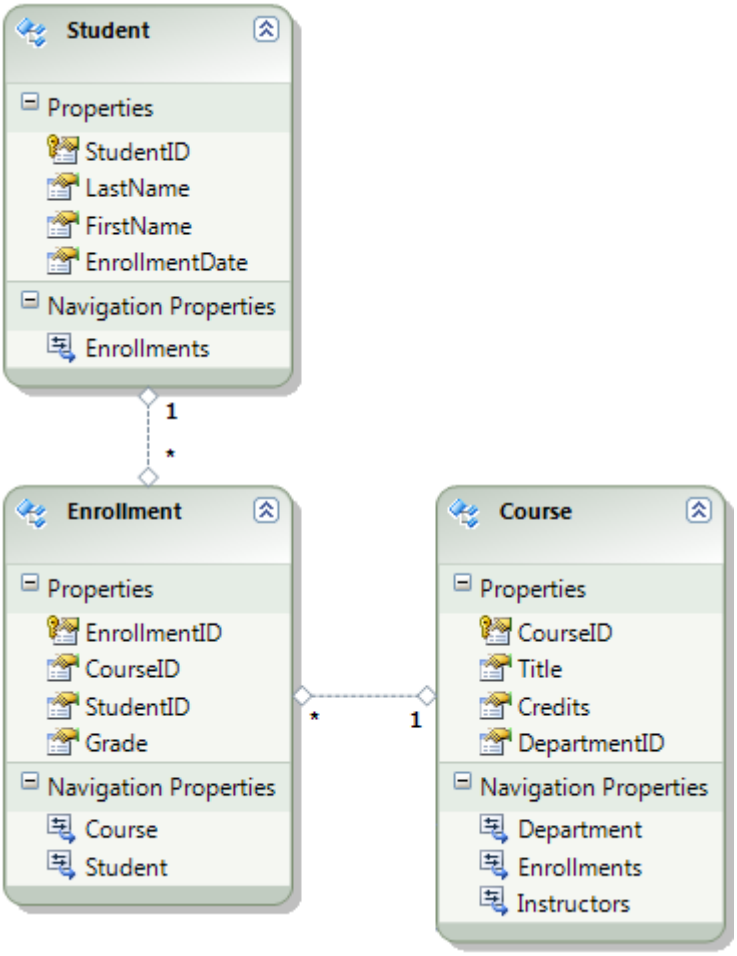
- An enrollment record is for a single student, so there's a `StudentID` foreign key property and a `Student` navigation property:

```
publicintStudentID{get;set;}
publicvirtualStudentStudent{get;set;}
```

Many-to-Many Relationships

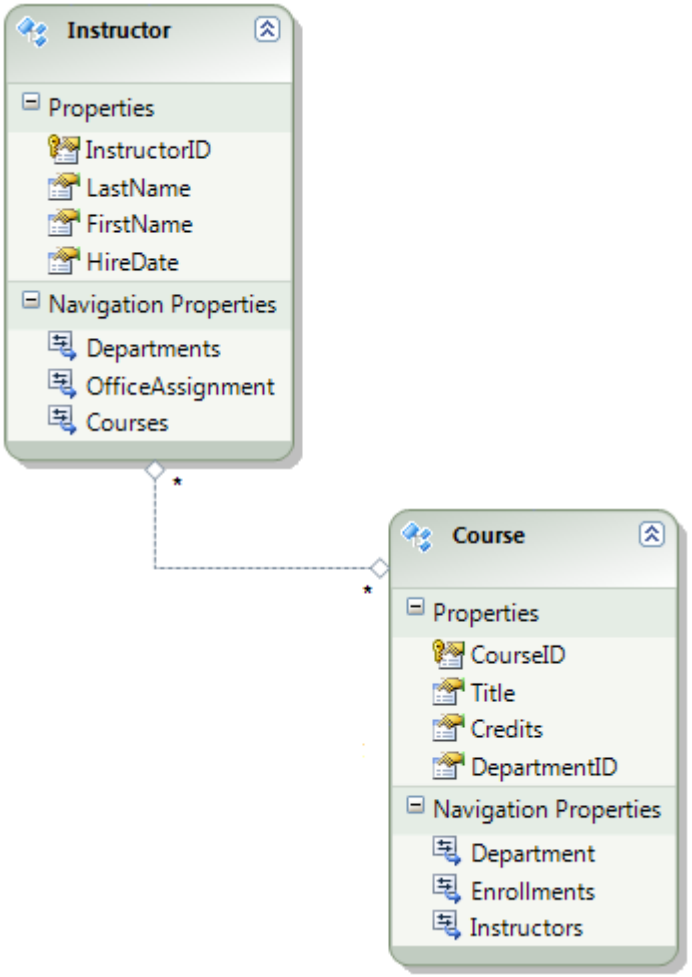
There's a many-to-many relationship between the `Student` and `Course` entities, and the `Enrollment` entity corresponds to a many-to-many join table *with payload* in the database. This means that the `Enrollment` table contains additional data besides foreign keys for the joined tables (in this case, 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 designer; creating the diagram isn't part of the tutorial, it's just being used here as an illustration.)

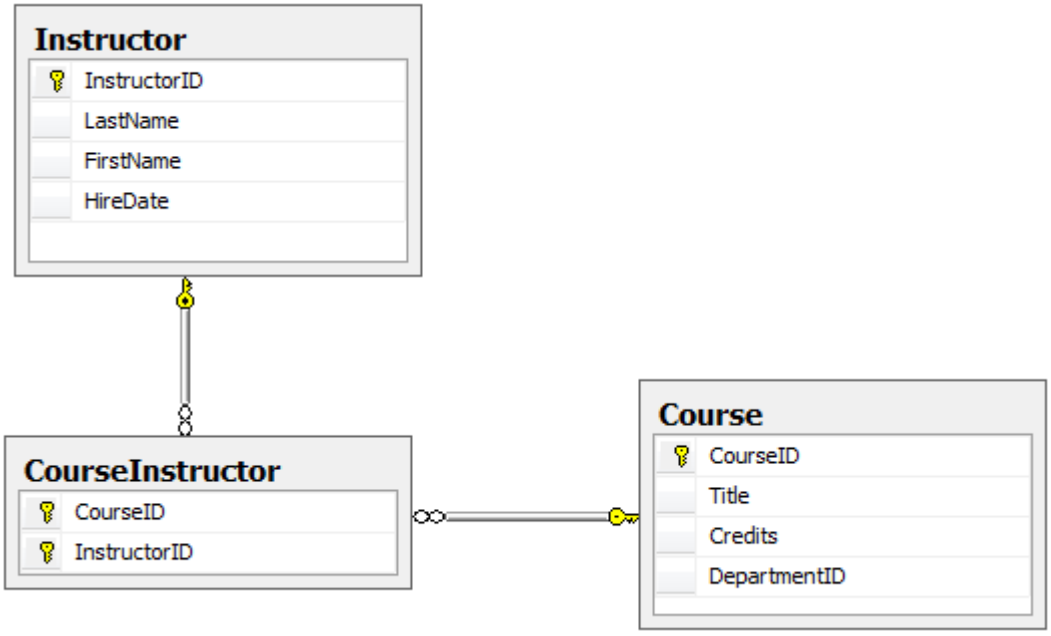


Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

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.

The DisplayFormat Attribute

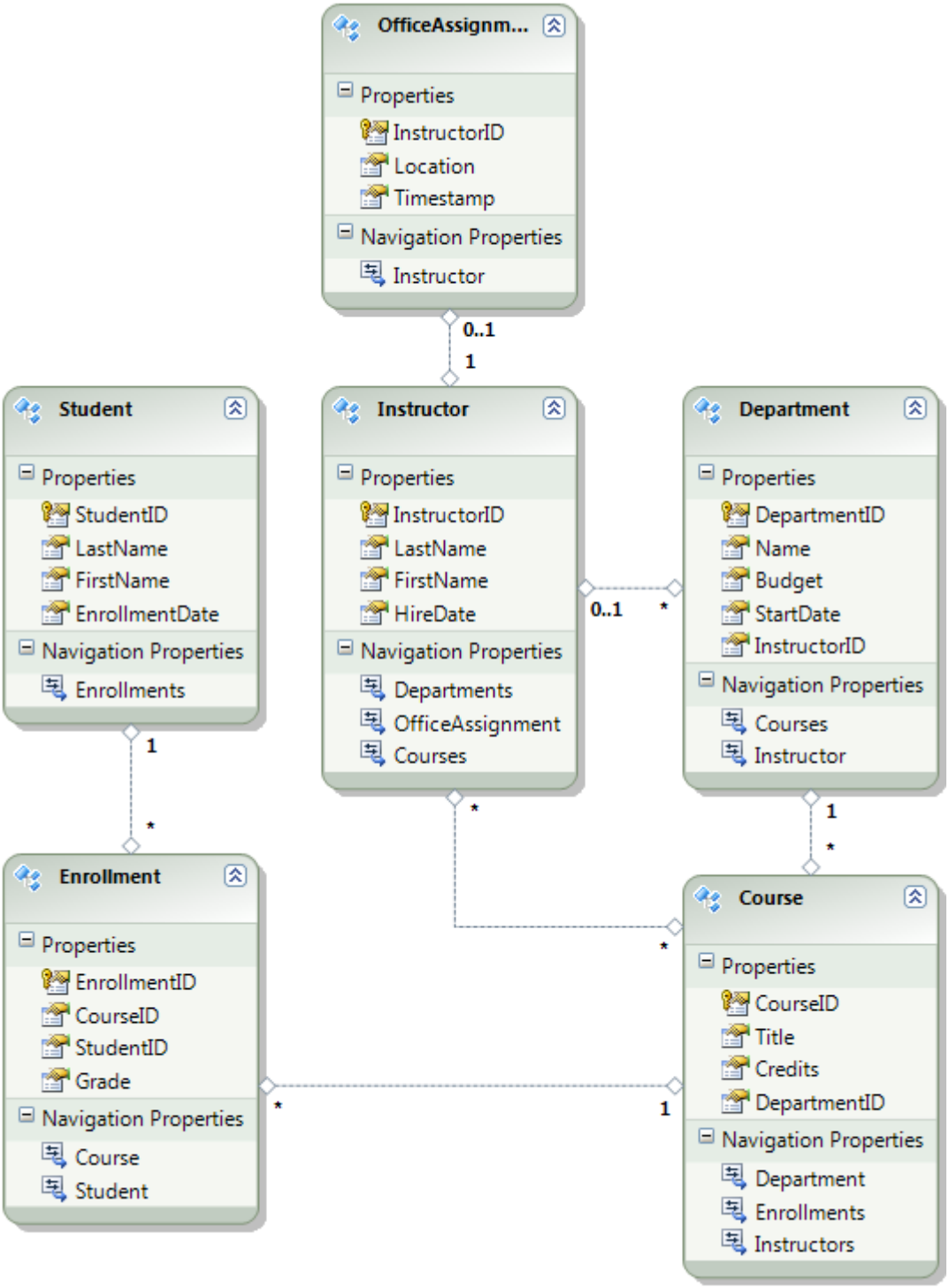
The `DisplayFormat` attribute on the `Grade` property specifies how the data will be formatted:

```
[DisplayFormat(DataFormatString="{0:#.#}",ApplyFormatInEditMode=true,NullDisplayText=
"No grade")]
publicdecimal?Grade{get;set;}
```

- The grade displays as two digits separated by a period — for example, "3.5" or "4.0".
- The grade is also displayed this way in edit mode (in a text box).
- If there's no grade (the question mark after `decimal` indicates that the property is nullable), the text "No grade" is displayed.

Entity Diagram Showing Relationships

The following illustration shows the diagram that the Entity Framework Database First designer creates for the 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.

Customizing 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.) In some cases you need to use methods rather than attributes because there's no attribute for a particular function. In other cases you can choose to use a method when both methods and attributes are available. (Some people prefer not to use attributes.)

Replace the code in *DAL\SchoolContext.cs* with the following code:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.Data.Entity;
usingContosoUniversity.Models;
usingSystem.Data.Entity.ModelConfiguration.Conventions;

namespaceContosoUniversity.Models
{
publicclassSchoolContext:DbContext
{
publicDbSet<Course>Courses{get;set;}
publicDbSet<Department>Departments{get;set;}
publicDbSet<Enrollment>Enrollments{get;set;}
publicDbSet<Instructor>Instructors{get;set;}
publicDbSet<Student>Students{get;set;}
publicDbSet<OfficeAssignment>OfficeAssignments{get;set;}

protectedoverridevoidOnModelCreating(DbModelBuilder modelBuilder)
{
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
            modelBuilder.Entity<Instructor>()
.HasOptional(p => p.OfficeAssignment).WithRequired(p => p.Instructor);
            modelBuilder.Entity<Course>()
.HasMany(c => c.Instructors).WithMany(i => i.Courses)
.Map(t => t.MapLeftKey("CourseID")
.MapRightKey("InstructorID")
.ToTable("CourseInstructor"));
            modelBuilder.Entity<Department>()
```

```
.HasOptional(x => x.Administrator);
}
}
}
```

The new statements in the `OnModelCreating` method specify the following relationships:

- A one-to-zero-or-one relationship between the `Instructor` and `OfficeAssignment` entities:

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

- A 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"));
```

- A zero-or-one-to-many relationship between the Instructor and Department tables. In other words, a department may or may not have an instructor assigned to it as administrator; the assigned administrator is represented by the `Department.Administrator` navigation property:

```
modelBuilder.Entity<Department>()
.HasOptional(x => x.Administrator);
```

For more details about what these "fluent API" statements are doing behind the scenes, see the Fluent API blog post on the ASP.NET User Education Team's blog.

Initializing the Database with Test Data

Earlier you created *DAL\SchoolInitializer.cs* to initialize your database with test data. Now replace the code in that file with the following code in order to provide test data for the new entities you've created.

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.Linq;
usingSystem.Web;
usingSystem.Data.Entity;
usingContosoUniversity.Models;

namespaceContosoUniversity.DAL
{
publicclassSchoolInitializer:DropCreateDatabaseIfModelChanges<SchoolContext>
{
protectedoverridevoidSeed(SchoolContext context)
{
var students =newList<Student>
{
newStudent{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("
2005-09-01")},
newStudent{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2
002-09-01")},
newStudent{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2003
-09-01")},
newStudent{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2
002-09-01")},
newStudent{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002-09-
01")},
newStudent{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("200
1-09-01")},
newStudent{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2003
-09-01")},
newStudent{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("200
5-09-01")}
};
            students.ForEach(s => context.Students.Add(s));
            context.SaveChanges();
```

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

var departments =newList<Department>
{
newDepartment{Name="English",Budget=350000,StartDate=DateTime.Parse("2007-09-
01"),InstructorID=1},
newDepartment{Name="Mathematics",Budget=100000,StartDate=DateTime.Parse("2007-09-
01"),InstructorID=2},
newDepartment{Name="Engineering",Budget=350000,StartDate=DateTime.Parse("2007-09-
01"),InstructorID=3},
newDepartment{Name="Economics",Budget=100000,StartDate=DateTime.Parse("2007-09-
01"),InstructorID=4}
};
            departments.ForEach(s => context.Departments.Add(s));
            context.SaveChanges();

var courses =newList<Course>
{
newCourse{CourseID=1050,Title="Chemistry",Credits=3,DepartmentID=3,Instructors=newLis
t<Instructor>()},
newCourse{CourseID=4022,Title="Microeconomics",Credits=3,DepartmentID=4,Instructors=n
ewList<Instructor>()},
newCourse{CourseID=4041,Title="Macroeconomics",Credits=3,DepartmentID=4,Instructors=n
ewList<Instructor>()},
```

```
newCourse{CourseID=1045,Title="Calculus",Credits=4,DepartmentID=2,Instructors=newList
<Instructor>()},
newCourse{CourseID=3141,Title="Trigonometry",Credits=4,DepartmentID=2,Instructors=new
List<Instructor>()},
newCourse{CourseID=2021,Title="Composition",Credits=3,DepartmentID=1,Instructors=newL
ist<Instructor>()},
newCourse{CourseID=2042,Title="Literature",Credits=4,DepartmentID=1,Instructors=newLi
st<Instructor>()}
};
            courses.ForEach(s => context.Courses.Add(s));
            context.SaveChanges();

            courses[0].Instructors.Add(instructors[0]);
            courses[0].Instructors.Add(instructors[1]);
            courses[1].Instructors.Add(instructors[2]);
            courses[2].Instructors.Add(instructors[2]);
            courses[3].Instructors.Add(instructors[3]);
            courses[4].Instructors.Add(instructors[3]);
            courses[5].Instructors.Add(instructors[3]);
            courses[6].Instructors.Add(instructors[3]);
            context.SaveChanges();

var enrollments =newList<Enrollment>
{
newEnrollment{StudentID=1,CourseID=1050,Grade=1},
newEnrollment{StudentID=1,CourseID=4022,Grade=3},
newEnrollment{StudentID=1,CourseID=4041,Grade=1},
newEnrollment{StudentID=2,CourseID=1045,Grade=2},
newEnrollment{StudentID=2,CourseID=3141,Grade=4},
newEnrollment{StudentID=2,CourseID=2021,Grade=4},
newEnrollment{StudentID=3,CourseID=1050},
newEnrollment{StudentID=4,CourseID=1050,},
newEnrollment{StudentID=4,CourseID=4022,Grade=4},
newEnrollment{StudentID=5,CourseID=4041,Grade=3},
newEnrollment{StudentID=6,CourseID=1045},
newEnrollment{StudentID=7,CourseID=3141,Grade=2},
};
            enrollments.ForEach(s => context.Enrollments.Add(s));
```

```
            context.SaveChanges();

var officeAssignments =newList<OfficeAssignment>
{
newOfficeAssignment{InstructorID=1,Location="Smith 17"},
newOfficeAssignment{InstructorID=2,Location="Gowan 27"},
newOfficeAssignment{InstructorID=3,Location="Thompson 304"},
};
            officeAssignments.ForEach(s => context.OfficeAssignments.Add(s));
            context.SaveChanges();
}
}
}
```

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

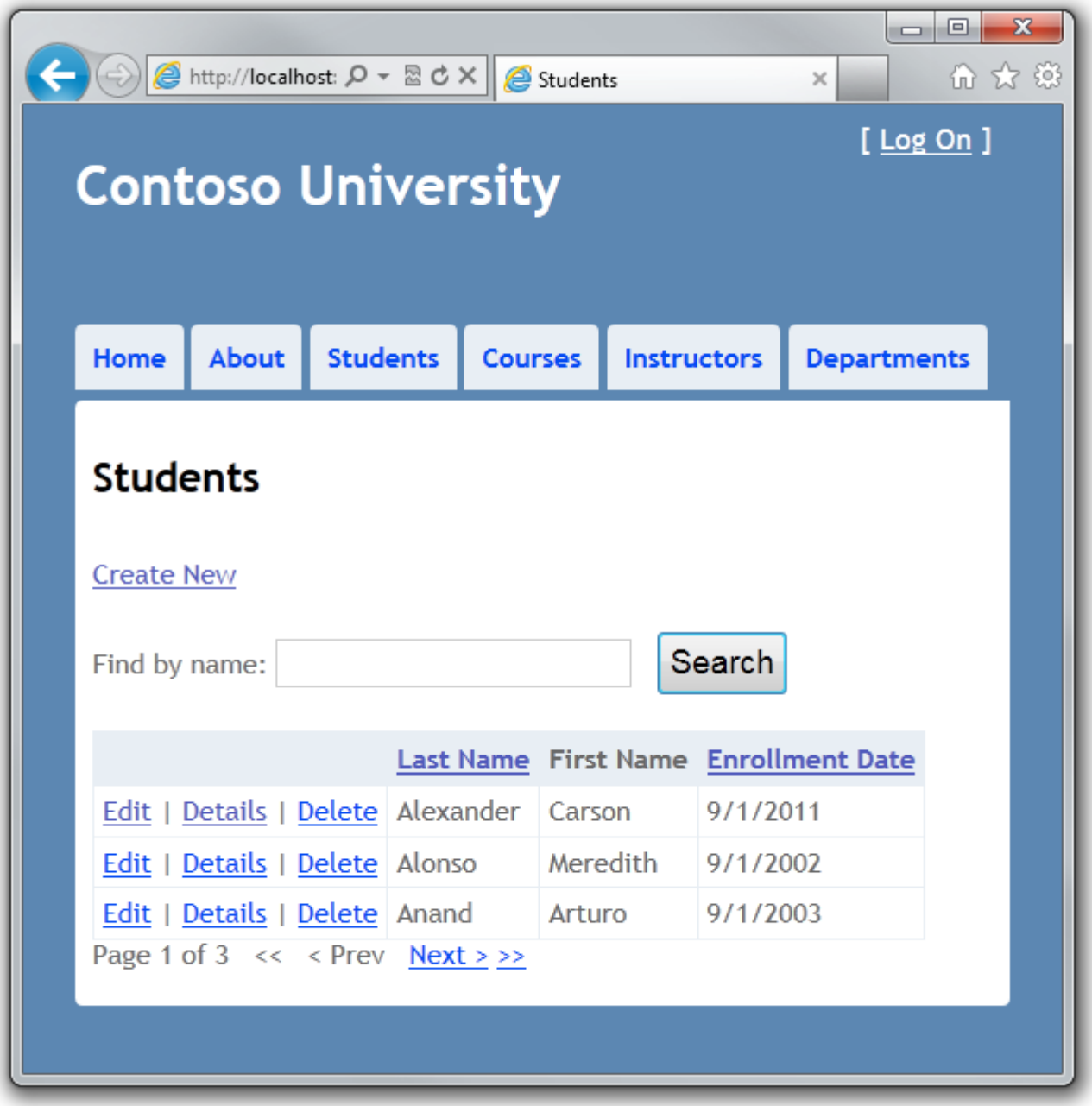
```
var courses =newList
{
newCourse{CourseID=1050,Title="Chemistry",Credits=3,DepartmentID=3,Instructors=newLis
t()},
...
};
courses.ForEach(s => context.Courses.Add(s));
context.SaveChanges();

courses[0].Instructors.Add(instructors[0]);
...
context.SaveChanges();
```

When you create a `Course` object, you initialize the `Instructors` navigation property as an empty collection using the code `Instructors = new List()`. 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.

Note Remember that when you deploy an application to a production web server, you must remove any code you've added to seed the database.

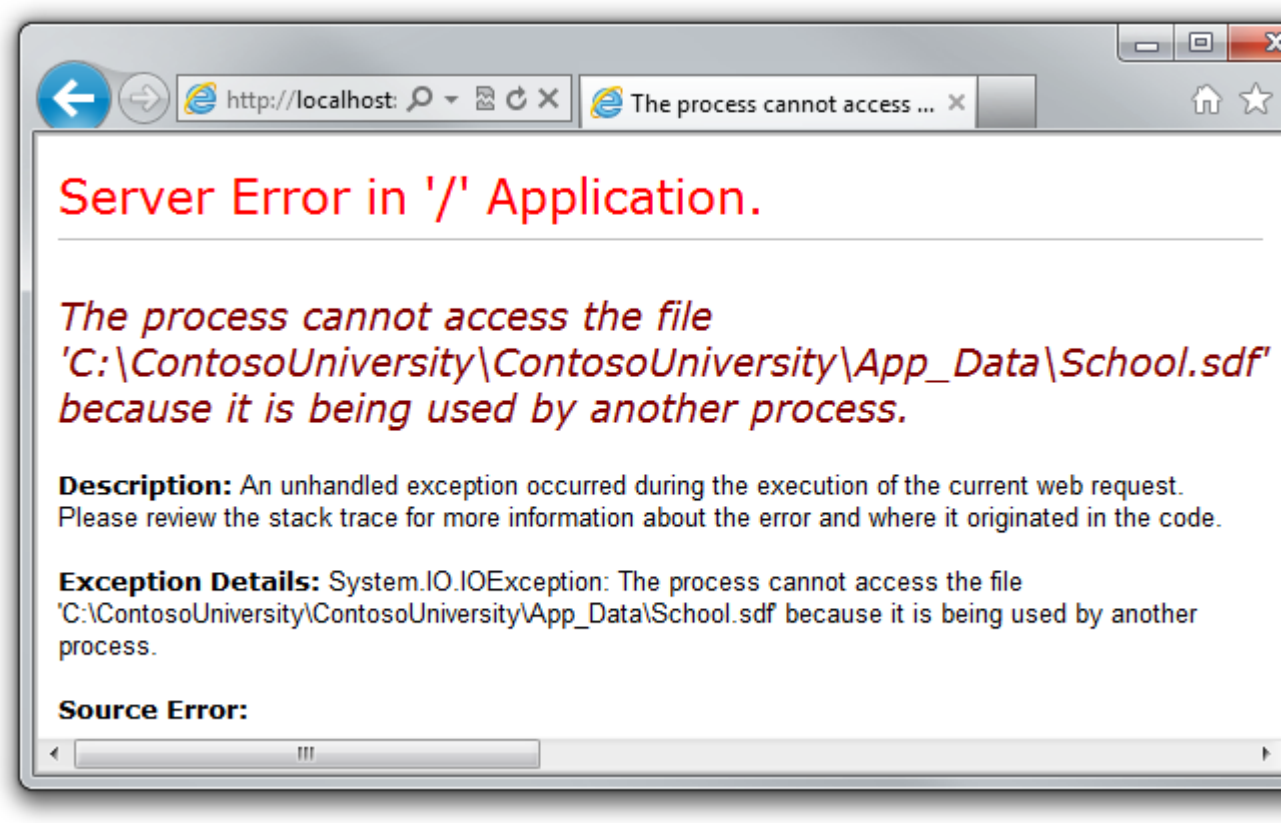
Dropping and Re-Creating the Database

Now run the site and select the Student Index page.

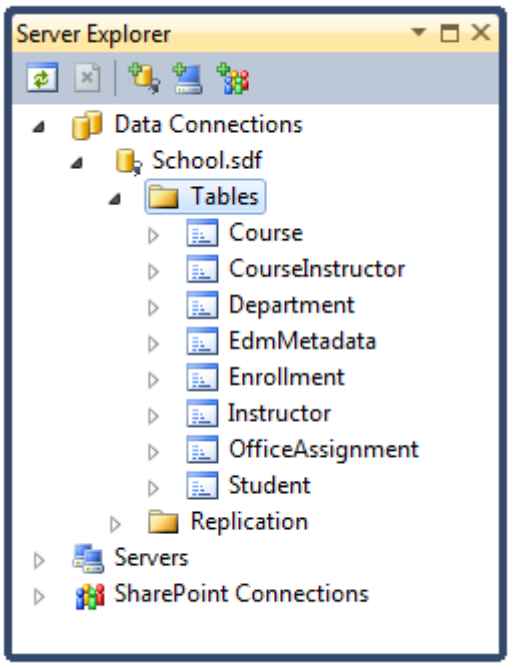


The page looks the same as it did before, but behind the scenes the database has been re-created.

If you don't see the Student Index page and instead you get an error that indicates that the *School.sdf* file is in use (see the following illustration), you need to reopen **Server Explorer** and close the connection to the database. Then try displaying the Student Index page again.

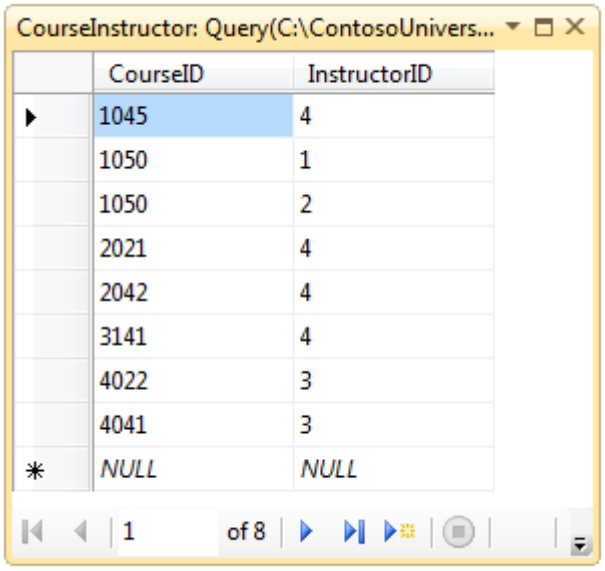


After viewing the Student Index page, open the database in **Server Explorer** as you did earlier, and expand the **Tables** node to see that all of the tables have been created.



Besides `EdmMetadata`, you see one table you didn't create a model class for: `CourseInstructor`. 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.



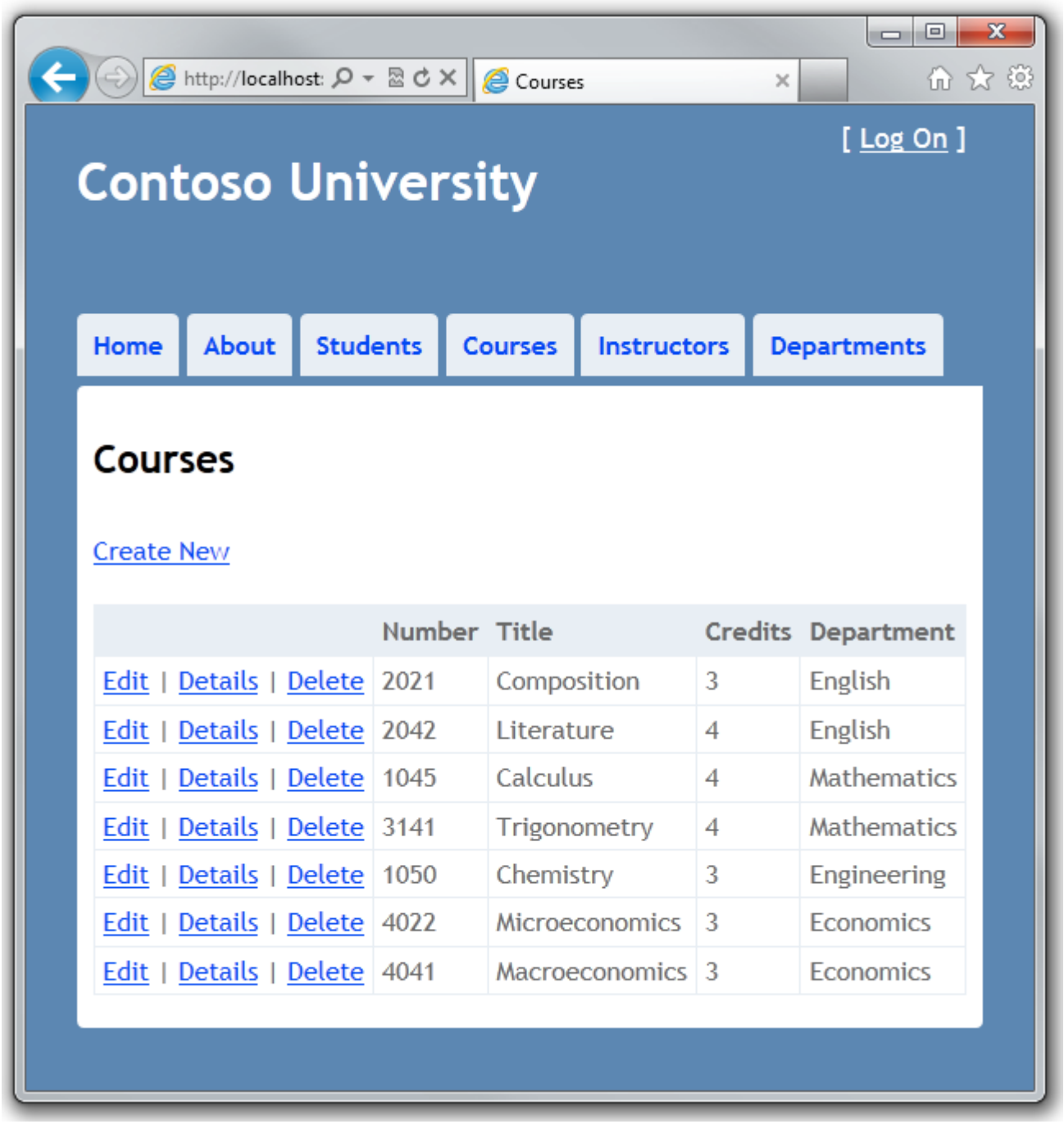
You now have a more complex data model and corresponding database. In the following tutorial you'll learn more about different ways to access related data.

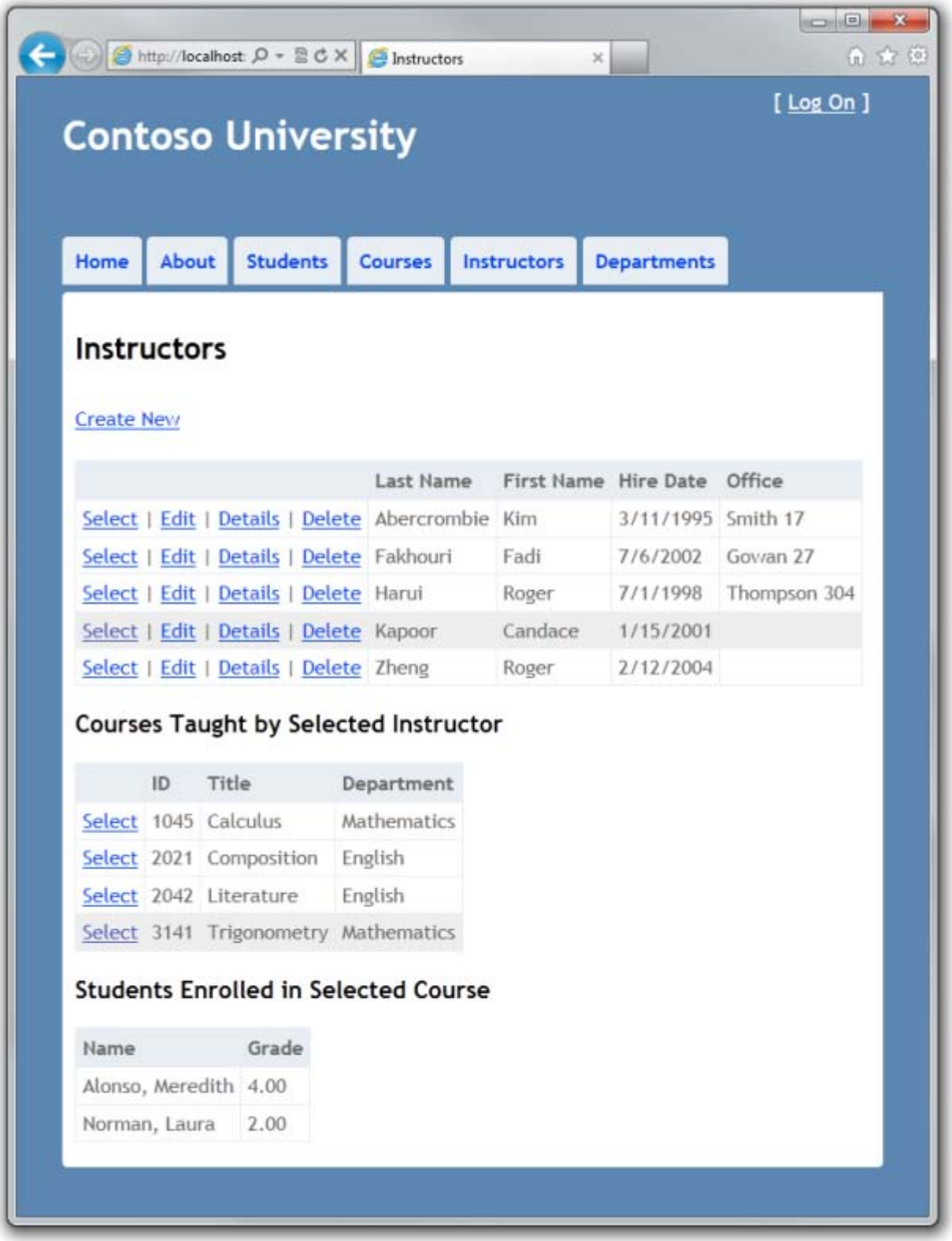
Links to other Entity Framework resources can be found at the end of the last tutorial in this series.

Reading Related Data

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.

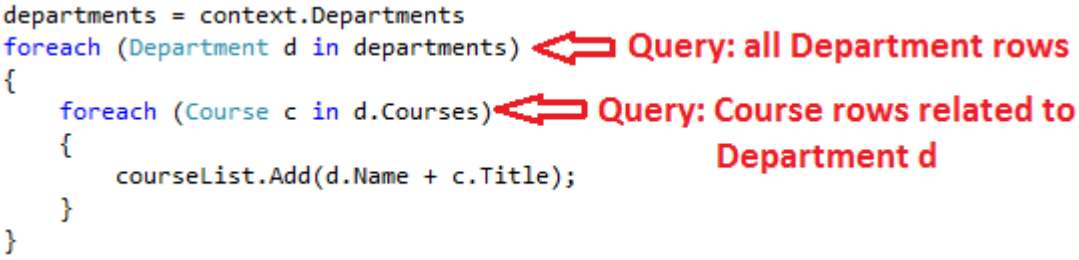




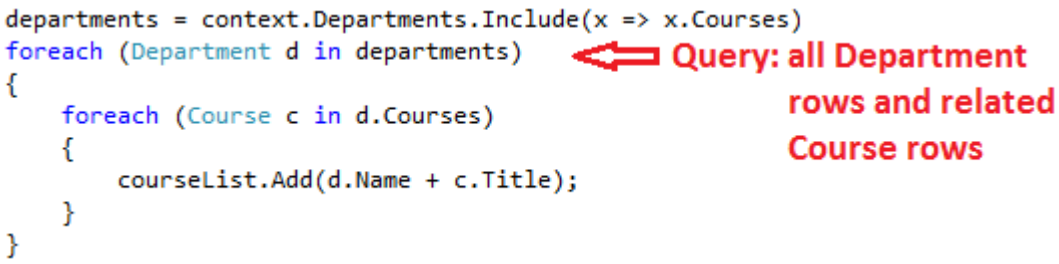
Lazy, Eager, and Explicit Loading of Related Data

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

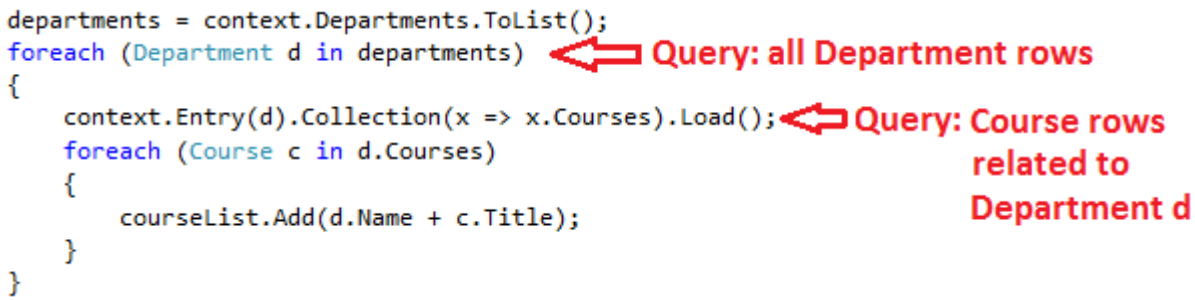
- *Lazy loading*. When the entity is first read, related data isn't retrieved. However, the first time you attempt to access a navigation property, the data required for that navigation property is automatically retrieved. This results in multiple queries sent to the database — one for the entity itself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



- *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.



- *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)`.)



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

In general, if you know you need related data for every entity retrieved, eager loading 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. The lazy loading and explicit loading examples would both result in eleven queries.

On the other hand, if you need to access an entity's navigation properties only infrequently or only for a small portion of a set of entities you're processing, lazy loading may be more efficient, because eager loading would retrieve more data than you need. Typically you'd use explicit loading only when you've turned lazy loading off. One scenario when you might turn lazy loading off is during serialization, when you know you don't need all navigation properties loaded. If lazy loading were on, all navigation properties would all be loaded automatically, because serialization accesses all properties.

The database context class performs lazy loading by default. There are two ways to turn off lazy loading:

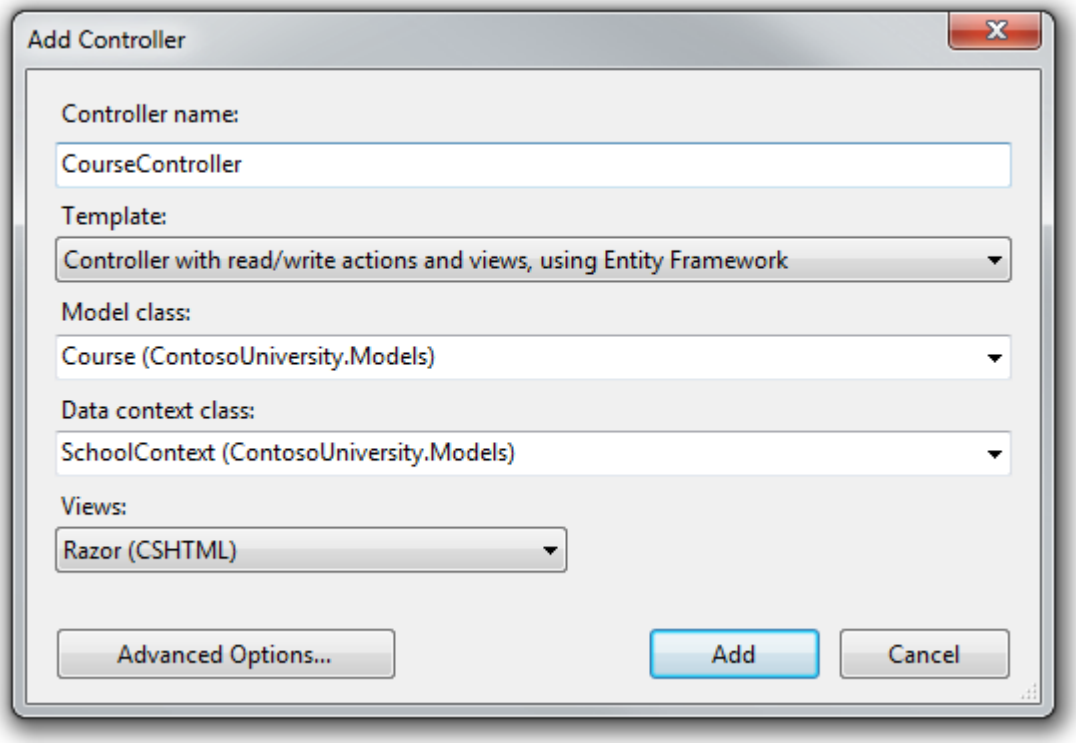
- For specific navigation properties, omit the `virtual` keyword when you declare the property.
- For all navigation properties, set `LazyLoadingEnabled` to `false`.

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), but it would work without errors if it relies on lazy loading. Temporarily disabling lazy loading is one way to discover where the code is relying on lazy loading, because without it the navigation properties will be null and the code will fail.

Creating a Courses Index 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 for the `Course` entity type, using the same options that you did earlier for the `Student` controller, as shown in the following illustration:



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

```
publicViewResultIndex()
{
var courses = db.Courses.Include(c => c.Department);
returnView(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 existing code with the following code:

```
@modelIEnumerable<ContosoUniversity.Models.Course>

@{
ViewBag.Title="Courses";
}

<h2>Courses</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
<tr>
<th></th>
<th>Number</th>
<th>Title</th>
<th>Credits</th>
<th>Department</th>
</tr>

@foreach(var item inModel){
<tr>
<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>
<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>
</tr>
}
</table>
```

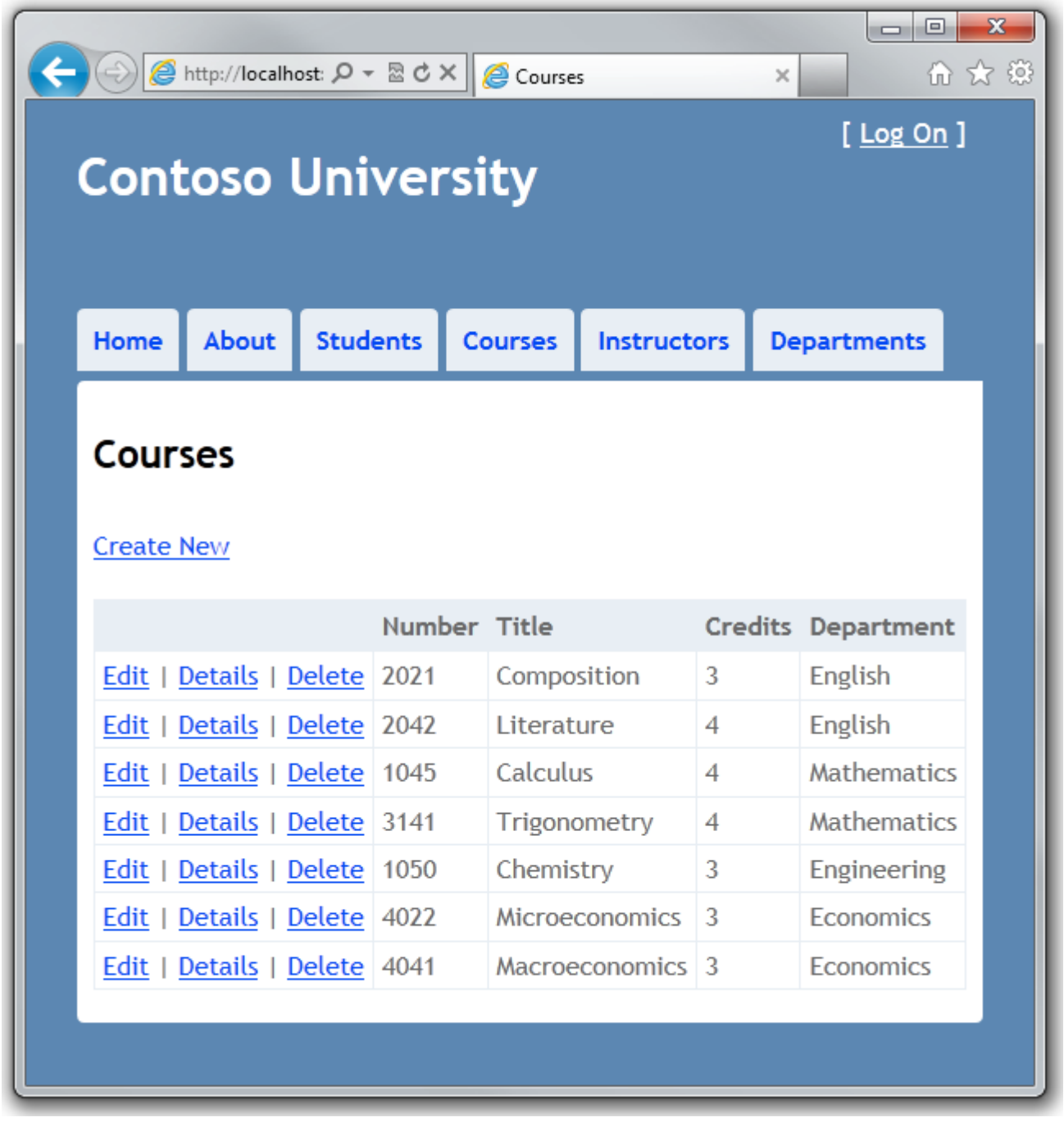
You've made the following changes to the scaffolded code:

- Changed the heading from **Index** to **Courses**.
- Moved the row links to the left.
- Added a column under the heading **Number** that shows the `CourseID` property value. (Primary keys aren't scaffolded because normally they are meaningless. However, in this case the primary key is meaningful and you want to show it.)
- Changed the last column heading from **DepartmentID** (the name of the foreign key to the `Department` entity) to **Department**.

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

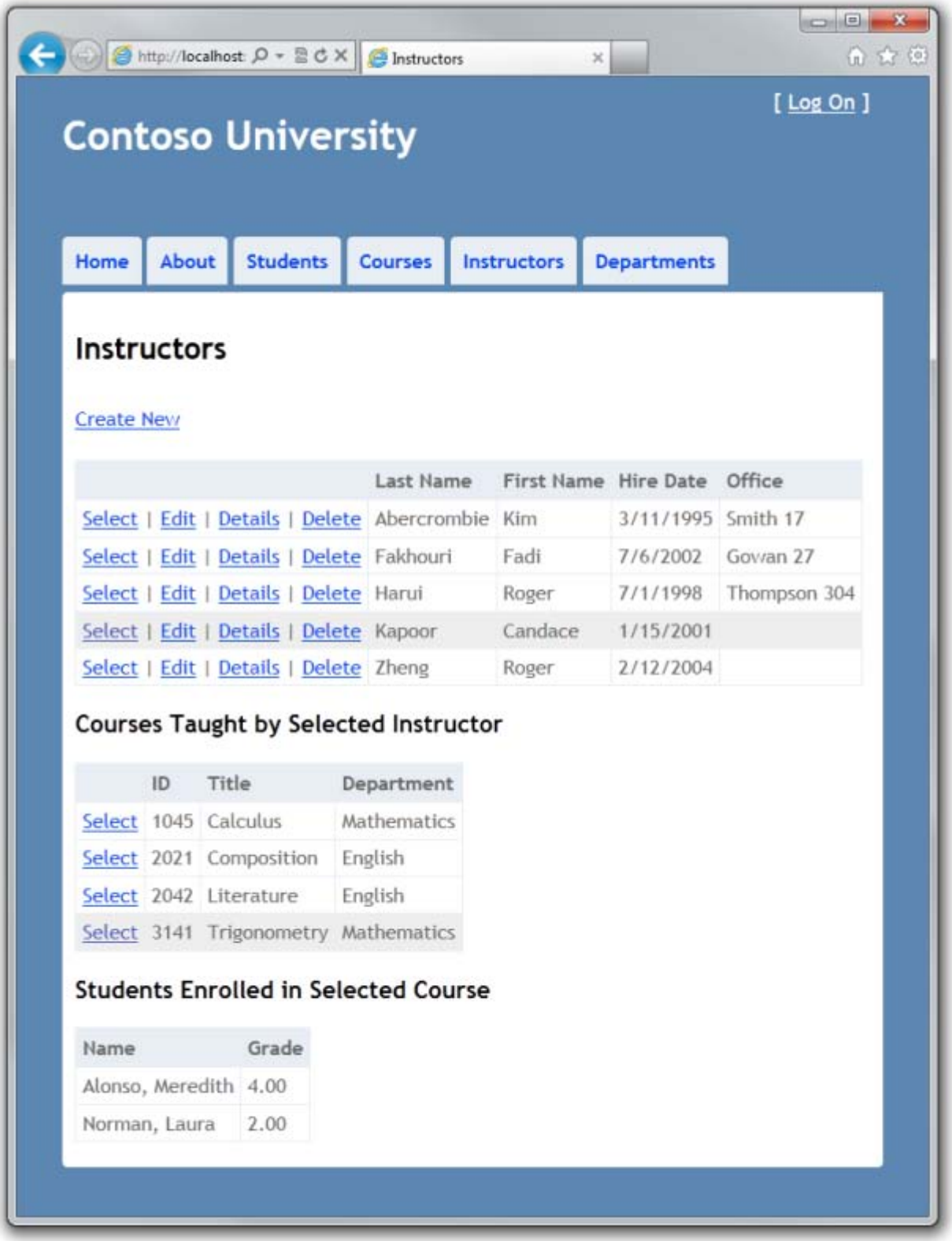
```
<td>
    @Html.DisplayFor(modelItem => item.Department.Name)
</td>
```

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



Creating an Instructors Index 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 Index page:



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 will 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.)

Creating a View Model for the Instructor Index View

The Instructor Index 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:

```
usingSystem;
usingSystem.Collections.Generic;
usingContosoUniversity.Models;

namespaceContosoUniversity.ViewModels
{
publicclassInstructorIndexData
{
publicIEnumerable<Instructor>Instructors{get;set;}
publicIEnumerable<Course>Courses{get;set;}
publicIEnumerable<Enrollment>Enrollments{get;set;}
}
}
```

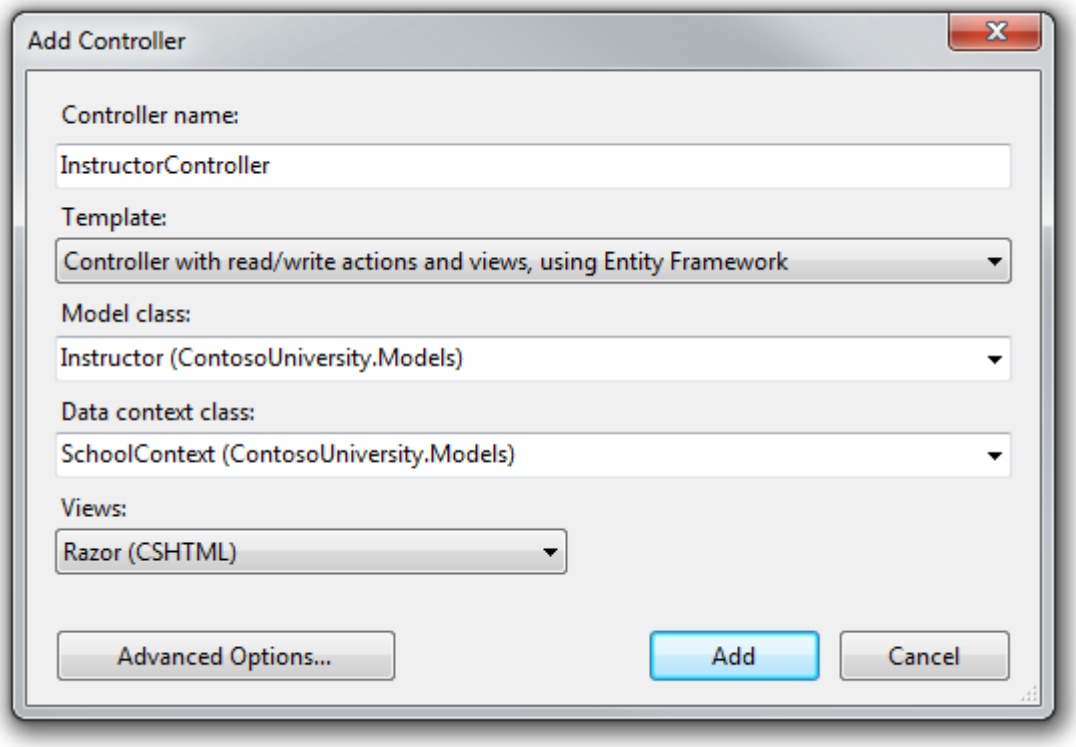
Adding a Style for Selected Rows

To mark selected rows you need a different background color. To provide a style for this UI, add the following code to the section marked `MISC` in *Content\Site.css*, as shown in the following example:

```
/* MISC
----------------------------------------------------------*/
.selectedrow
{
    background-color:#EEEEEE;
}
```

Creating the Instructor Controller and Views

Create a controller for the `Instructor` entity type, using the same options that you did earlier for the `Student` controller, as shown in the following illustration:



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

```
usingContosoUniversity.ViewModels;
```

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

```
publicViewResultIndex()
{
var instructors = db.Instructors.Include(i => i.OfficeAssignment);
returnView(instructors.ToList());
}
```

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

```
publicActionResultIndex(Int32? id,Int32? courseID)
{
var viewModel =newInstructorIndexData();
    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.InstructorID==
id.Value).Single().Courses;
}

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

returnView(viewModel);
}
```

The method accepts optional query string parameters that provide the ID values of the selected instructor and selected course, and passes all of the required data to the view. The query string 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:

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

This statement specifies eager loading for the `Instructor.OfficeAssignment` and the `Instructor.Courses` navigation property. For the related `Course` entities, eager loading is specified for the `Course.Department` navigation property by using the `Select` method within the `Include` method. The results are sorted by last name.

If an instructor 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.InstructorID==
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 null 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.InstructorID== id.Value)
```

Instead of:

```
.Where(I => i.InstructorID== 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;
}
```

Finally, the view model is returned to the view:

```
returnView(viewModel);
```

Modifying the Instructor Index View

In *Views\Instructor\Index.cshtml*, replace the existing code with the following code:

```
@modelContosoUniversity.ViewModels.InstructorIndexData

@{
ViewBag.Title="Instructors";
}

<h2>Instructors</h2>
```

```
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
<tr>
<th></th>
<th>Last Name</th>
<th>FirstName</th>
<th>Hire Date</th>
<th>Office</th>
</tr>
@foreach(var item inModel.Instructors)
{
string selectedRow ="";
if(item.InstructorID==ViewBag.InstructorID)
{
            selectedRow ="selectedrow";
}
<tr class="@selectedRow" valign="top">
<td>
@Html.ActionLink("Select","Index",new{ id = item.InstructorID})|
@Html.ActionLink("Edit","Edit",new{ id = item.InstructorID})|
@Html.ActionLink("Details","Details",new{ id = item.InstructorID})|
@Html.ActionLink("Delete","Delete",new{ id = item.InstructorID})
</td>
<td>
                @item.LastName
</td>
<td>
@item.FirstMidName
</td>
<td>
                @String.Format("{0:d}", item.HireDate)
</td>
<td>
@if(item.OfficeAssignment!=null)
{
```

```
@item.OfficeAssignment.Location
}
</td>
</tr>
}
</table>
```

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

- Changed the page title from **Index** to **Instructors**.
- Moved the row link columns to the left.
- Removed the **FullName** column.
- 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.)

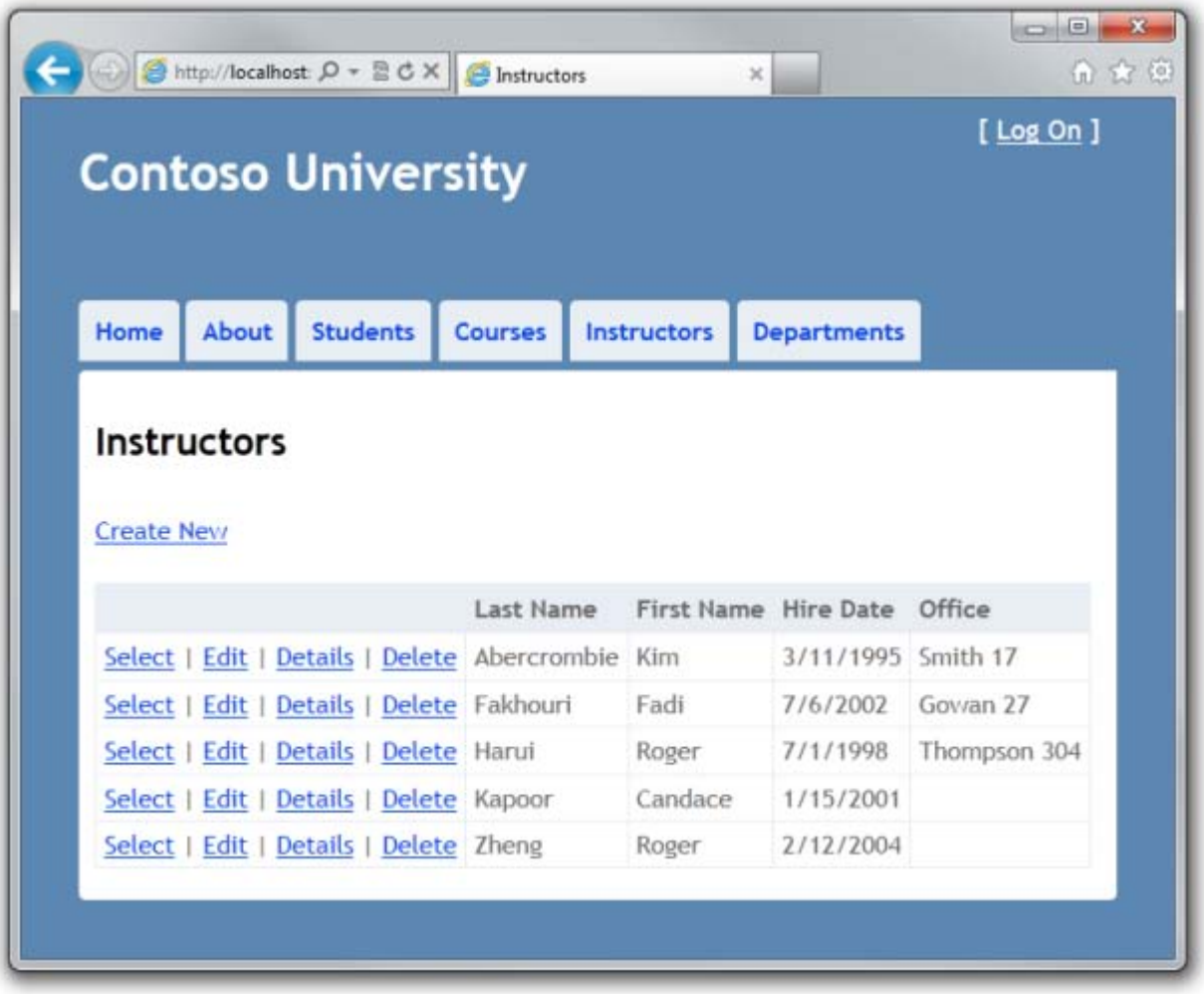
```
<td>
    @if (item.OfficeAssignment != null)
    {
        @item.OfficeAssignment.Location
    }
</td>
```

- Added code that will dynamically add `class="selectedrow"` to the `tr` element of the selected instructor. This sets a background color for the selected row using the CSS class that you created earlier. (The `valign` attribute will be useful in the following tutorial when you add a multirow column to the table.)

```
string selectedRow ="";
if(item.InstructorID==ViewBag.InstructorID)
{
    selectedRow ="selectedrow";
}
<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 page to see the list of instructors. The page displays the `Location` property of related `OfficeAssignment` entities and an empty table cell when there's no related `OfficeAssignment` entity.



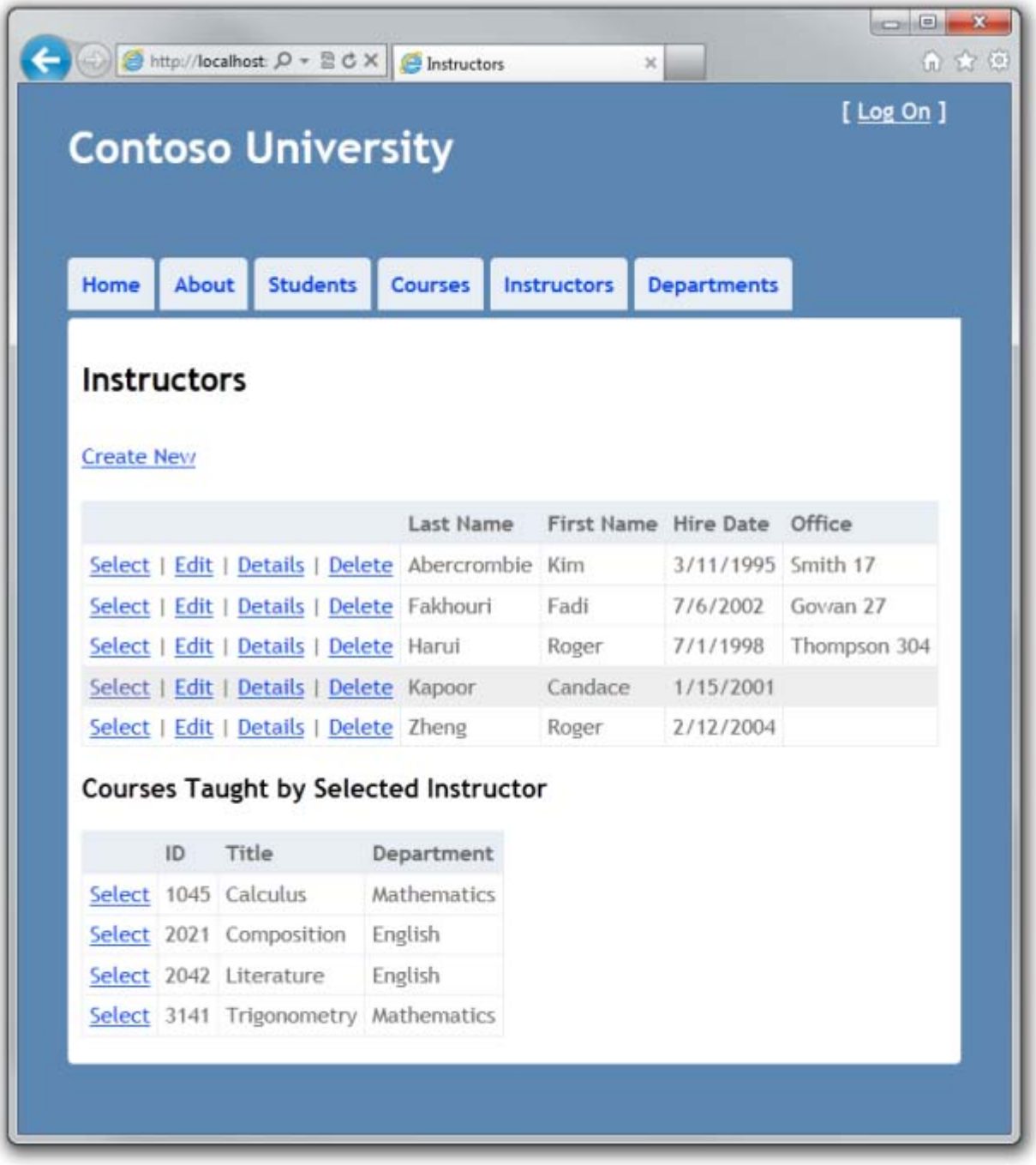
While you still have *Views\Instructor\Index.cshtml* open, after the `table` element, add the following code. This displays a list of courses related to an instructor when an instructor is selected.

```
@if(Model.Courses!=null)
{
<h3>CoursesTaughtbySelectedInstructor</h3>
<table>
<tr>
<th></th>
```

```
<th>ID</th>
<th>Title</th>
<th>Department</th>
</tr>

@foreach(var item inModel.Courses)
{
string selectedRow ="";
if(item.CourseID==ViewBag.CourseID)
{
            selectedRow ="selectedrow";
}
<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.



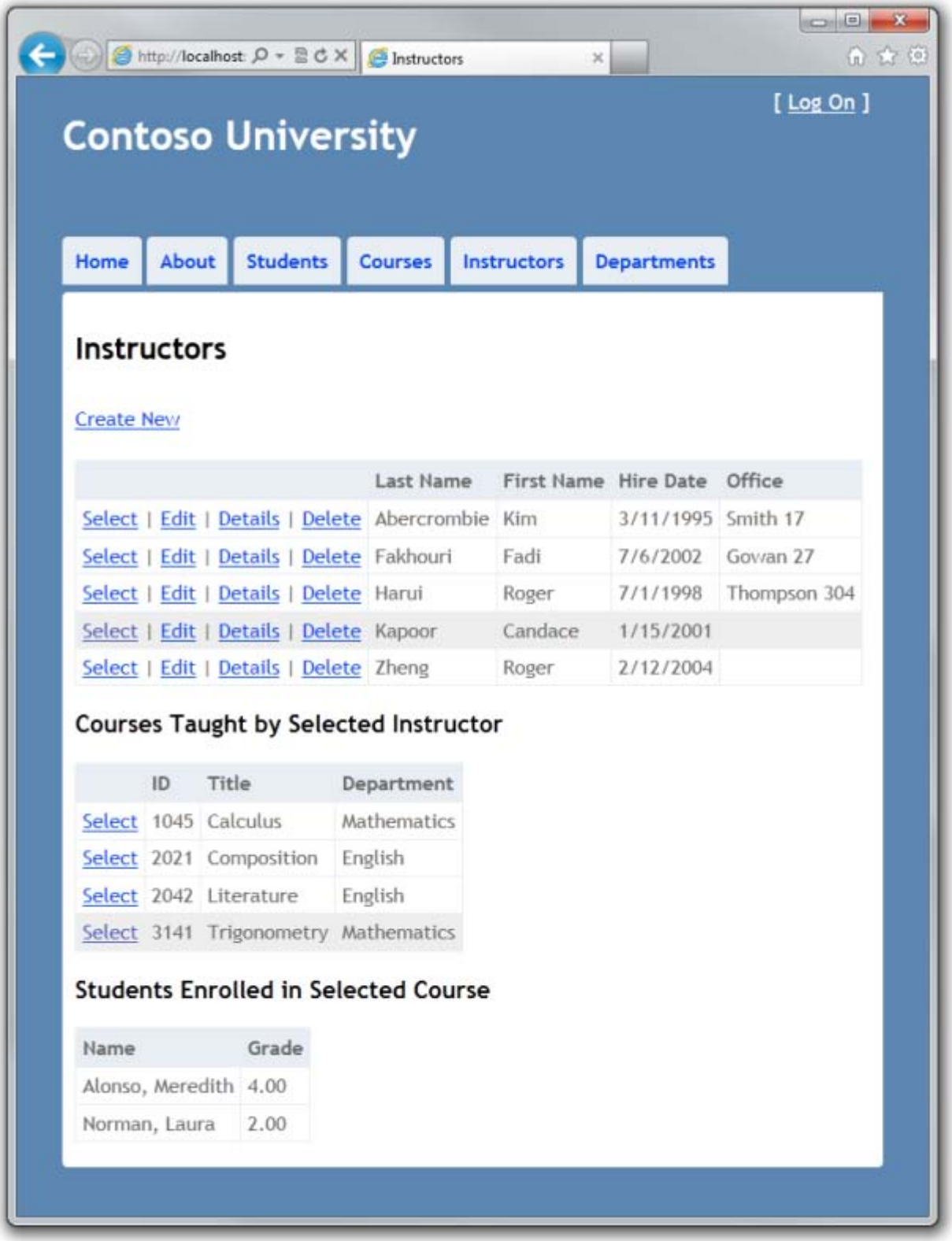
Note If the selected row isn't highlighted, click the **Refresh** button on your browser; this is sometimes required in order to reload the *.css* file.

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>
StudentsEnrolledinSelectedCourse</h3>
<table>
<tr>
<th>Name</th>
<th>Grade</th>
</tr>
@foreach(var item inModel.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. The `DisplayFor` helper is used so that null grades will display as "No grade", as specified in the `DisplayFormat` data annotation attribute for that field.

Run the page and select an instructor. Then select a course to see the list of enrolled students and their grades.



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:

```
publicActionResultIndex(Int32? id,Int32? courseID)
{
var viewModel =newInstructorIndexData();
    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.InstructorID==
id.Value).Single().Courses;
}
```

```
if(courseID !=null)
{
ViewBag.CourseID= courseID.Value;

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;
}

returnView(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. You can 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.

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.

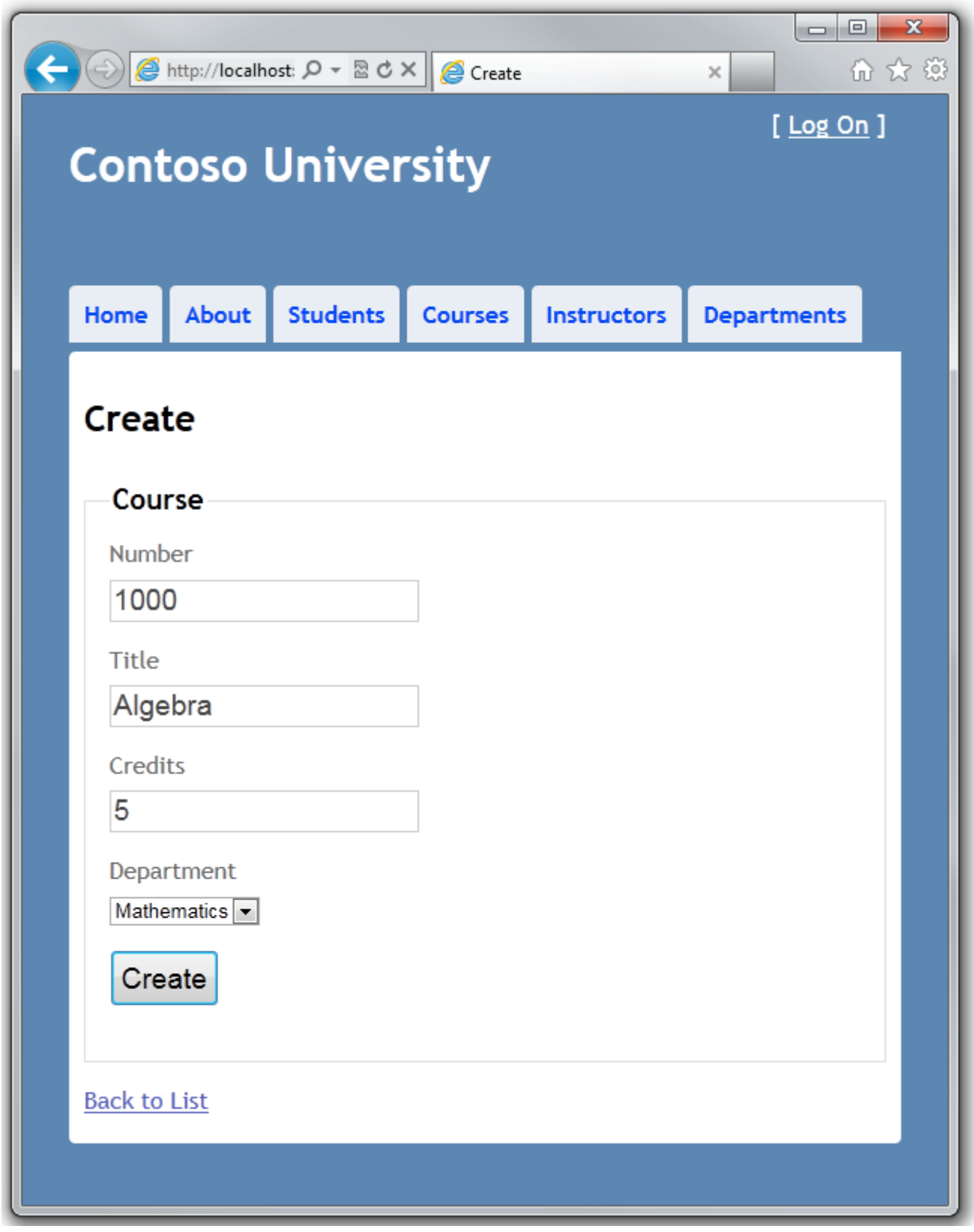
Links to other Entity Framework resources, can be found at the end of the last tutorial in this series.

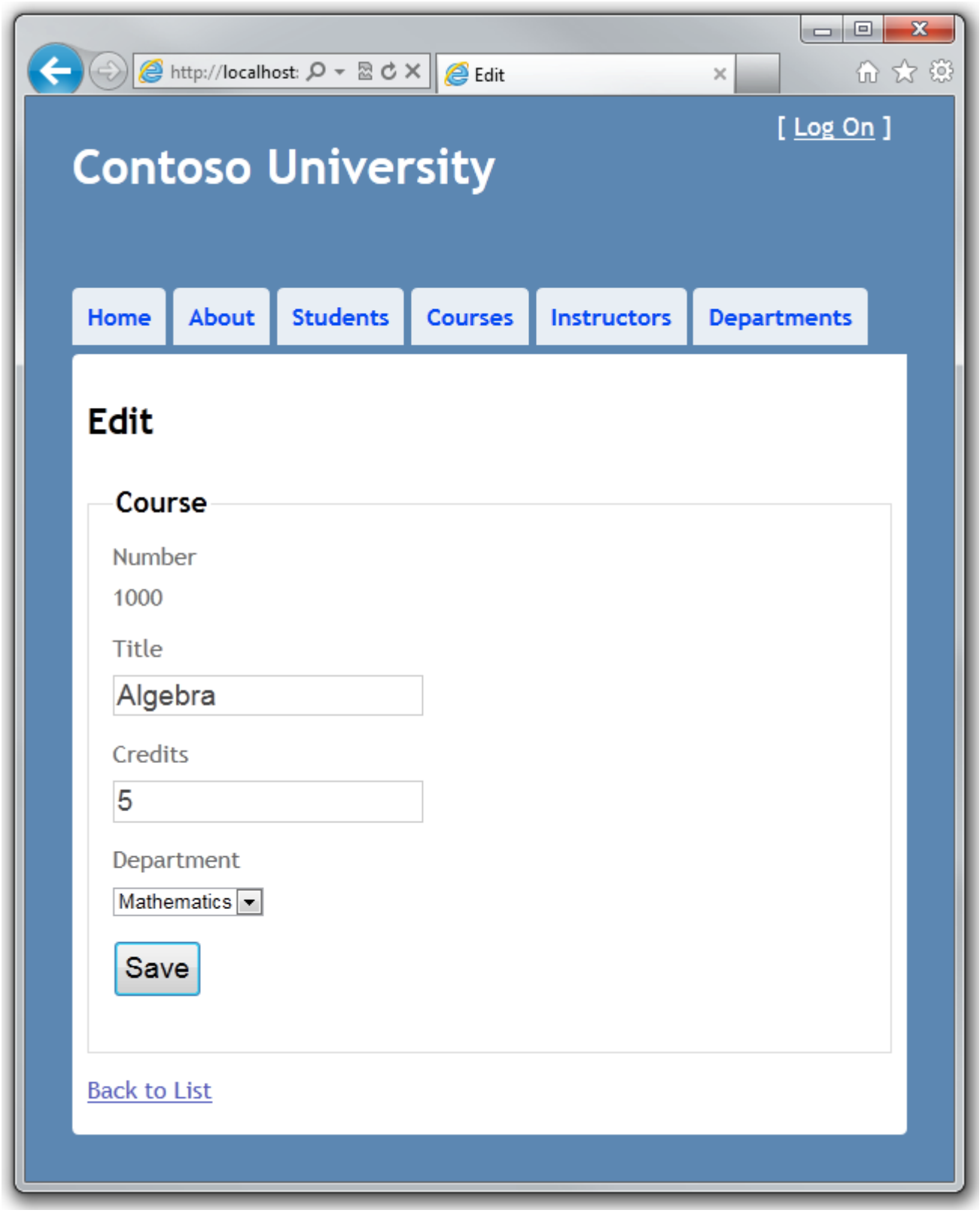
This page intentionally left blank

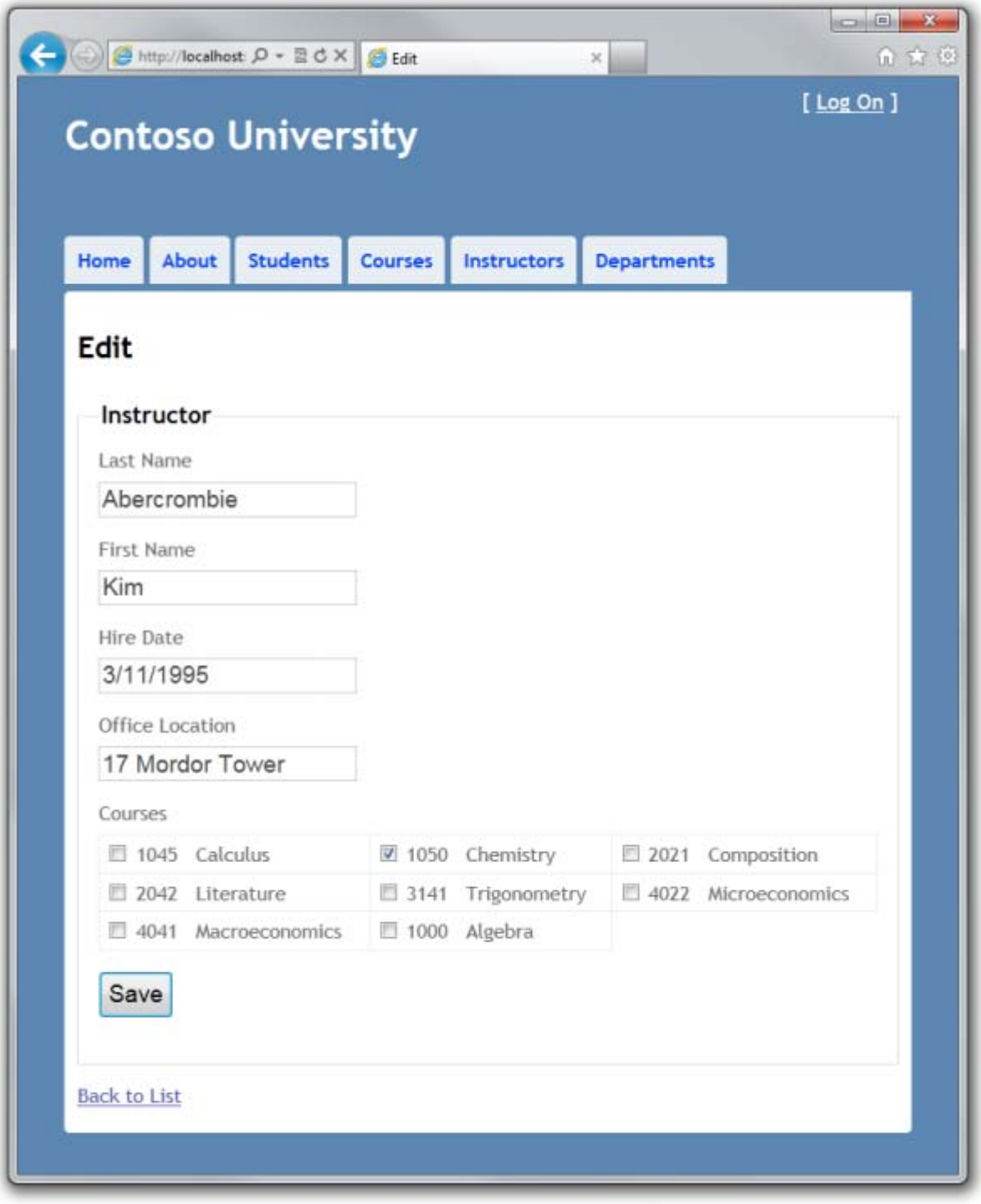
Updating Related Data

In the previous tutorial you displayed related data; in in this tutorial you'll update related data. For most relationships, this can be done by updating the appropriate foreign key fields. For many-to-many relationships, the Entity Framework doesn't expose the join table directly, so you must explicitly add and remove entities to and from the appropriate navigation properties.

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


http://localhost:
Create
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Create
Course
Number
1000
Title
Algebra
Credits
5
Department
Mathematics
Create
Back to List


http://localhost:
Edit
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Edit
Course
Number
1000
Title
Algebra
Credits
5
Department
Mathematics
Save
Back to List



Customizing 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

is 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 `Edit` and `Create` methods and replace them with the following code:

```
publicActionResultCreate()
{
PopulateDepartmentsDropDownList();
returnView();
}

[HttpPost]
publicActionResultCreate(Course course)
{
try
{
if(ModelState.IsValid)
{
            db.Courses.Add(course);
            db.SaveChanges();
returnRedirectToAction("Index");
}
}
catch(DataException)
{
//Log the error (add a variable name after DataException)
ModelState.AddModelError("","Unable to save changes. Try again, and if the problem
persists, see your system administrator.");
}
PopulateDepartmentsDropDownList(course.DepartmentID);
returnView(course);
}

publicActionResultEdit(int id)
{
Course course = db.Courses.Find(id);
PopulateDepartmentsDropDownList(course.DepartmentID);
```

```
returnView(course);
}

[HttpPost]
publicActionResultEdit(Course course)
{
try
{
if(ModelState.IsValid)
{
            db.Entry(course).State=EntityState.Modified;
            db.SaveChanges();
returnRedirectToAction("Index");
}
}
catch(DataException)
{
//Log the error (add a variable name after DataException)
ModelState.AddModelError("","Unable to save changes. Try again, and if the problem
persists, see your system administrator.");
}
PopulateDepartmentsDropDownList(course.DepartmentID);
returnView(course);
}

privatevoidPopulateDepartmentsDropDownList(object selectedDepartment =null)
{
var departmentsQuery =from d in db.Departments
orderby d.Name
select d;
ViewBag.DepartmentID=newSelectList(departmentsQuery,"DepartmentID","Name",
selectedDepartment);
}
```

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 a parameter that allows the caller to optionally specify the item that will be selected initially when the drop-down list is rendered.

The `HttpGetCreate` method calls the `PopulateDepartmentsDropDownList` method without setting the selected item, because for a new course the department is not established yet:

```
publicActionResultCreate()
{
PopulateDepartmentsDropDownList();
returnView();
}
```

The `HttpGetEdit` method sets the selected item, based on the ID of the department that is already assigned to the course being edited:

```
publicActionResultEdit(int id)
{
Course course = db.Courses.Find(id);
PopulateDepartmentsDropDownList(course.DepartmentID);
returnView(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(DataException)
{
//Log the error (add a variable name after DataException)
ModelState.AddModelError("","Unable to save changes. Try again, and if the problem
persists, see your system administrator.");
}
PopulateDepartmentsDropDownList(course.DepartmentID);
returnView(course);
```

This code ensures that when the page is redisplayed to show the error message, whatever department was selected stays selected.

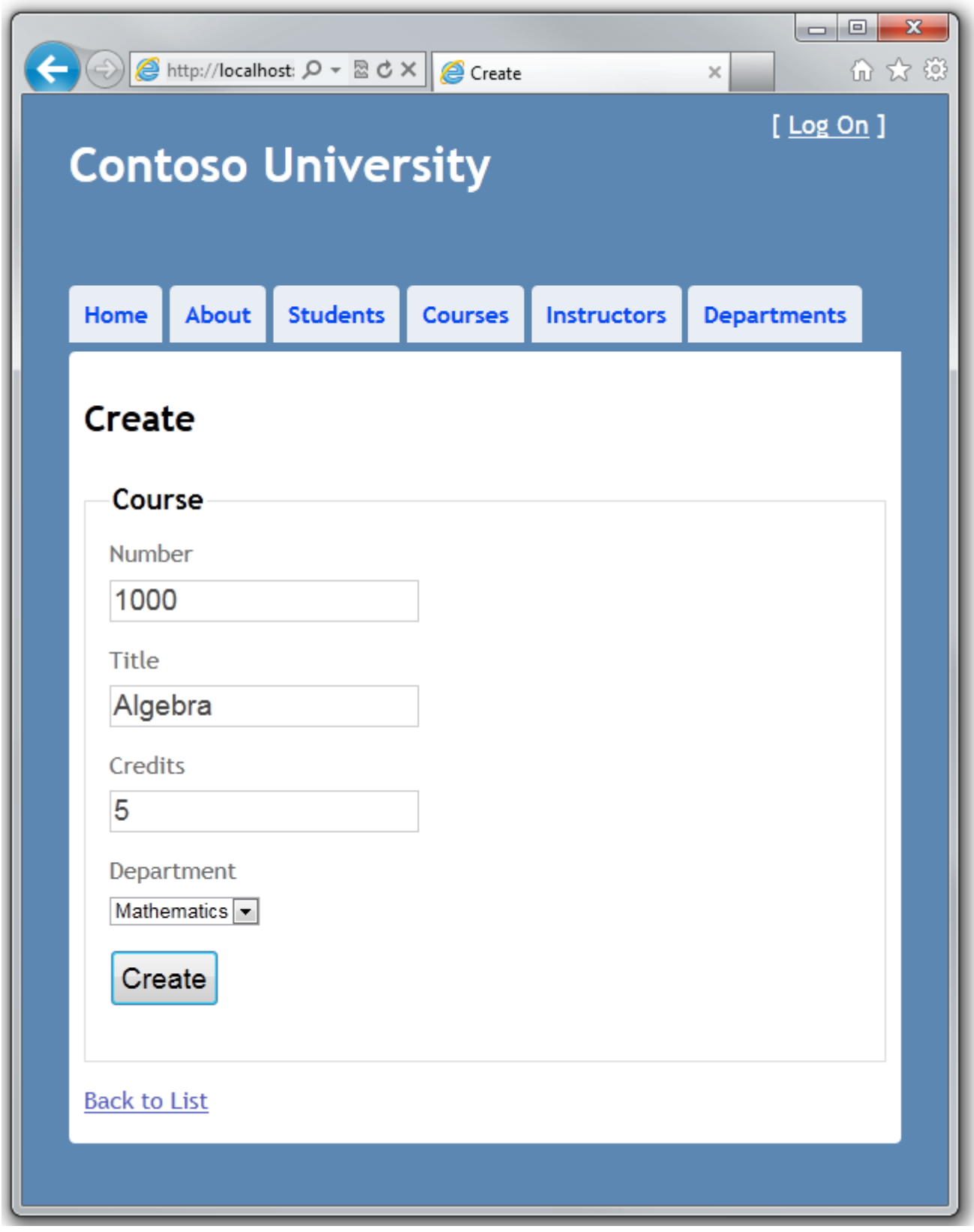
In *Views\Course\Create.cshtml*, add a new field before the **Title** field to allow the user to enter the course number. As explained in an earlier tutorial, primary key fields aren't scaffolded, but this primary key is meaningful, so you want the user to be able to enter the key value.

```
<divclass="editor-label">
    @Html.LabelFor(model => model.CourseID)
</div>
<divclass="editor-field">
    @Html.EditorFor(model => model.CourseID)
    @Html.ValidationMessageFor(model => model.CourseID)
</div>
```

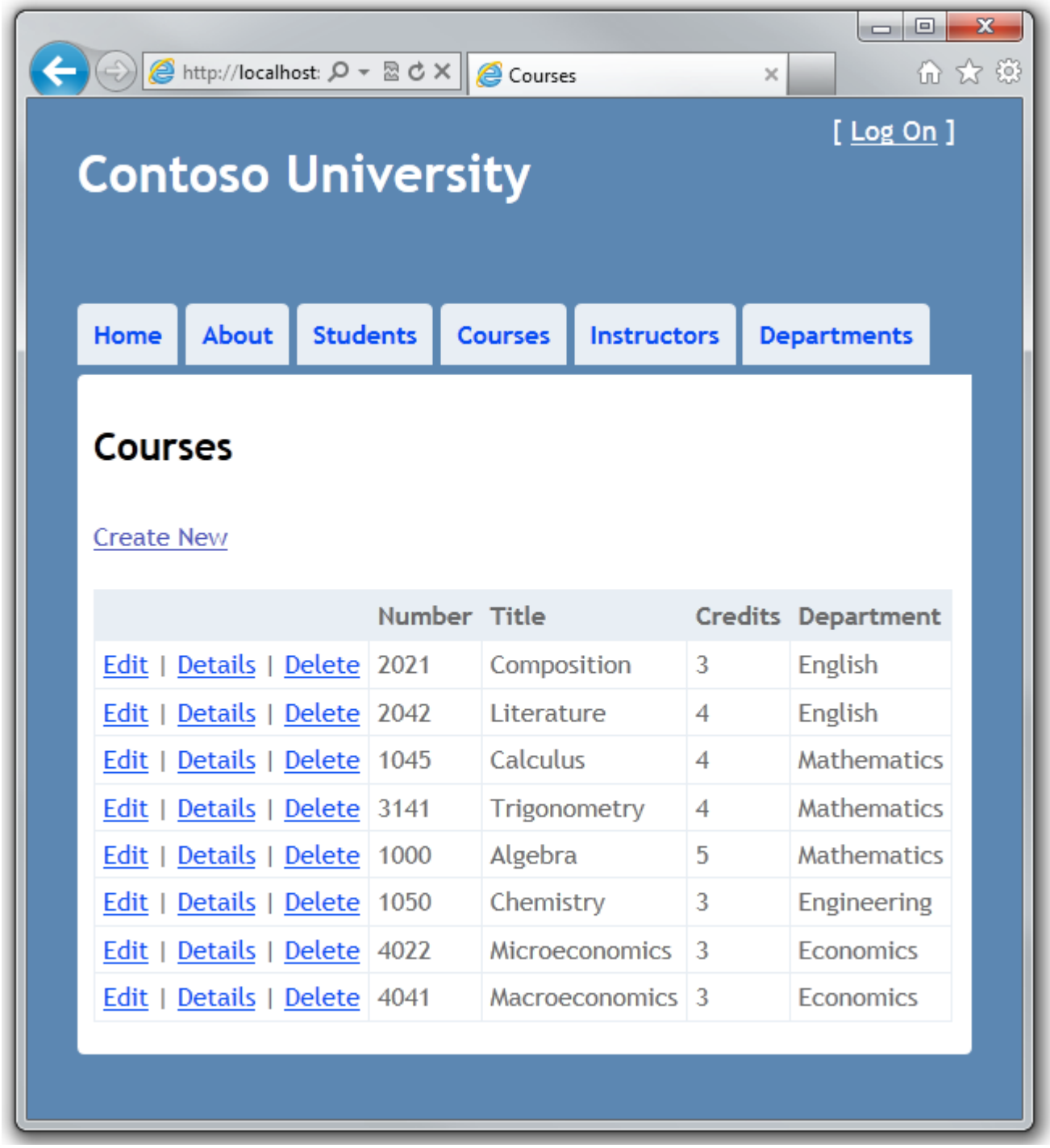
In *Views\Course\Edit.cshtml*, *Views\Course\Delete.cshtml*, and *Views\Course\Details.cshtml*, add a new field before the **Title** field to display the course number. Because it's the primary key, it's displayed, but it can't be changed.

```
<divclass="editor-label">
    @Html.LabelFor(model => model.CourseID)
</div>
<divclass="editor-field">
    @Html.DisplayFor(model => model.CourseID)
</div>
```

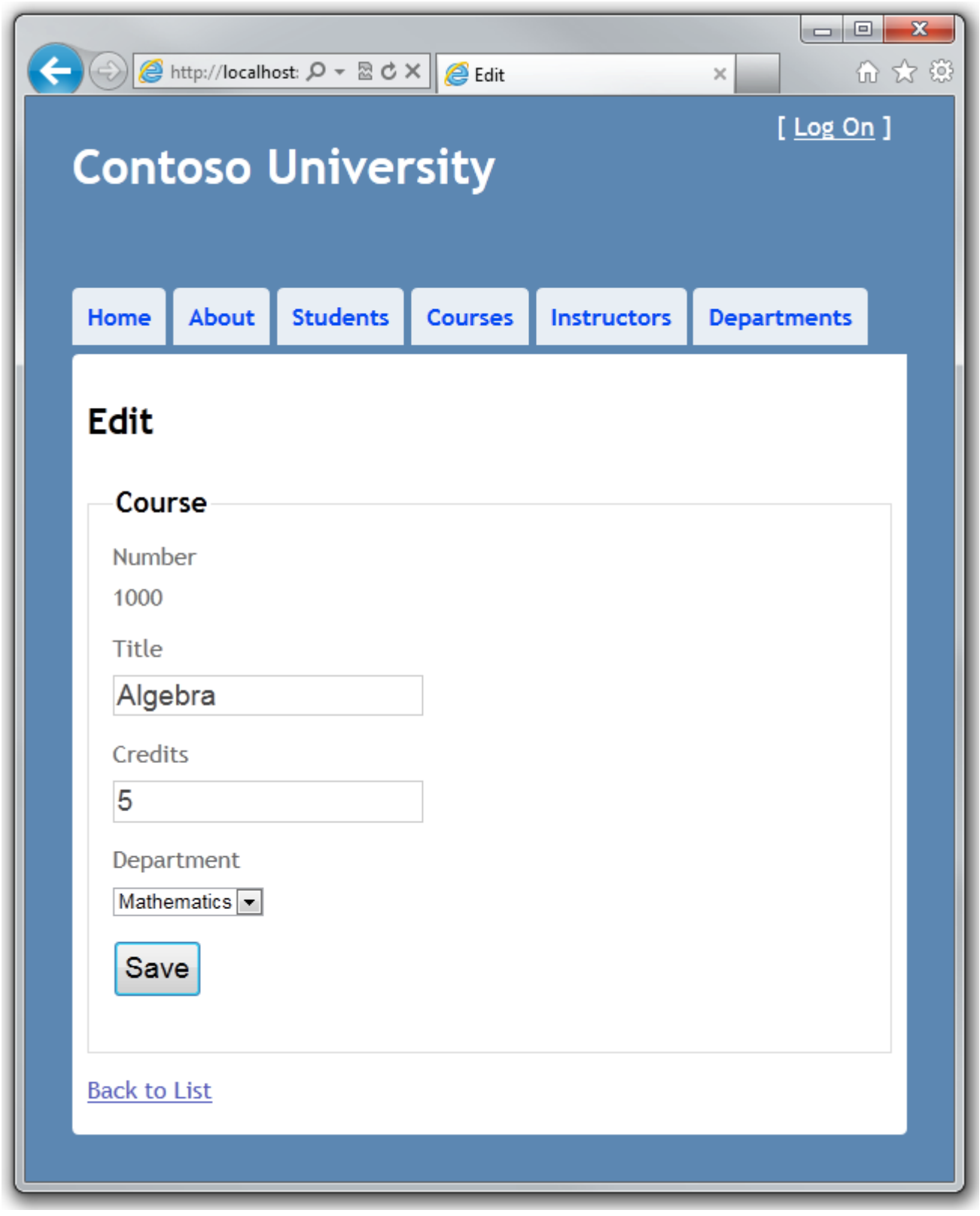
Run the **Create** page (display the Course Index page and click **Create New**) and enter data for a new course:


http://localhost:
Create
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Create
Course
Number
1000
Title
Algebra
Credits
5
Department
Mathematics
Create
Back to List

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.



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 `HttpGetEdit` method:

```
publicActionResultEdit(int id)
{
Instructor instructor = db.Instructors.Find(id);
ViewBag.InstructorID=newSelectList(db.OfficeAssignments,"InstructorID","Location",
instructor.InstructorID);
returnView(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:

```
publicActionResultEdit(int id)
{
Instructor instructor = db.Instructors
.Include(i => i.OfficeAssignment)
.Include(i => i.Courses)
.Where(i => i.InstructorID== id)
.Single();
returnView(instructor);
}
```

This code drops the `ViewBag` statement and adds eager loading for associated `OfficeAssignment` and `Course` entities. (You don't need `Courses` now, but you'll need it later.) 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 `HttpPostEdit` method with the following code. which handles office assignment updates:

```
[HttpPost]
publicActionResultEdit(int id,FormCollection formCollection)
{
var instructorToUpdate = db.Instructors
.Include(i => i.OfficeAssignment)
.Include(i => i.Courses)
.Where(i => i.InstructorID== id)
.Single();
if(TryUpdateModel(instructorToUpdate,"",null,newstring[]{"Courses"}))
{
try
{
if(String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment.Location))
{
                instructorToUpdate.OfficeAssignment=null;
}

            db.Entry(instructorToUpdate).State=EntityState.Modified;
            db.SaveChanges();

returnRedirectToAction("Index");
}
catch(DataException)
{
//Log the error (add a variable name after DataException)
ModelState.AddModelError("","Unable to save changes. Try again, and if the problem
persists, see your system administrator.");
returnView();
}
}
returnView(instructorToUpdate);
}
```

The code does the following:

- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment` and `Courses` navigation properties. This is the same as what you did in the `HttpGetEdit` method.
- Updates the retrieved `Instructor` entity with values from the model binder, excluding the `Courses` navigation property:

```
If(TryUpdateModel(instructorToUpdate,"",null,newstring[]{"Courses"}))
```

  (The second and third parameters specify no prefix on the property names and no list of properties to include.) If validation fails, `TryUpdateModel` returns `false`, and the code falls through to the `return View` statement at the end of the method.

- 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.

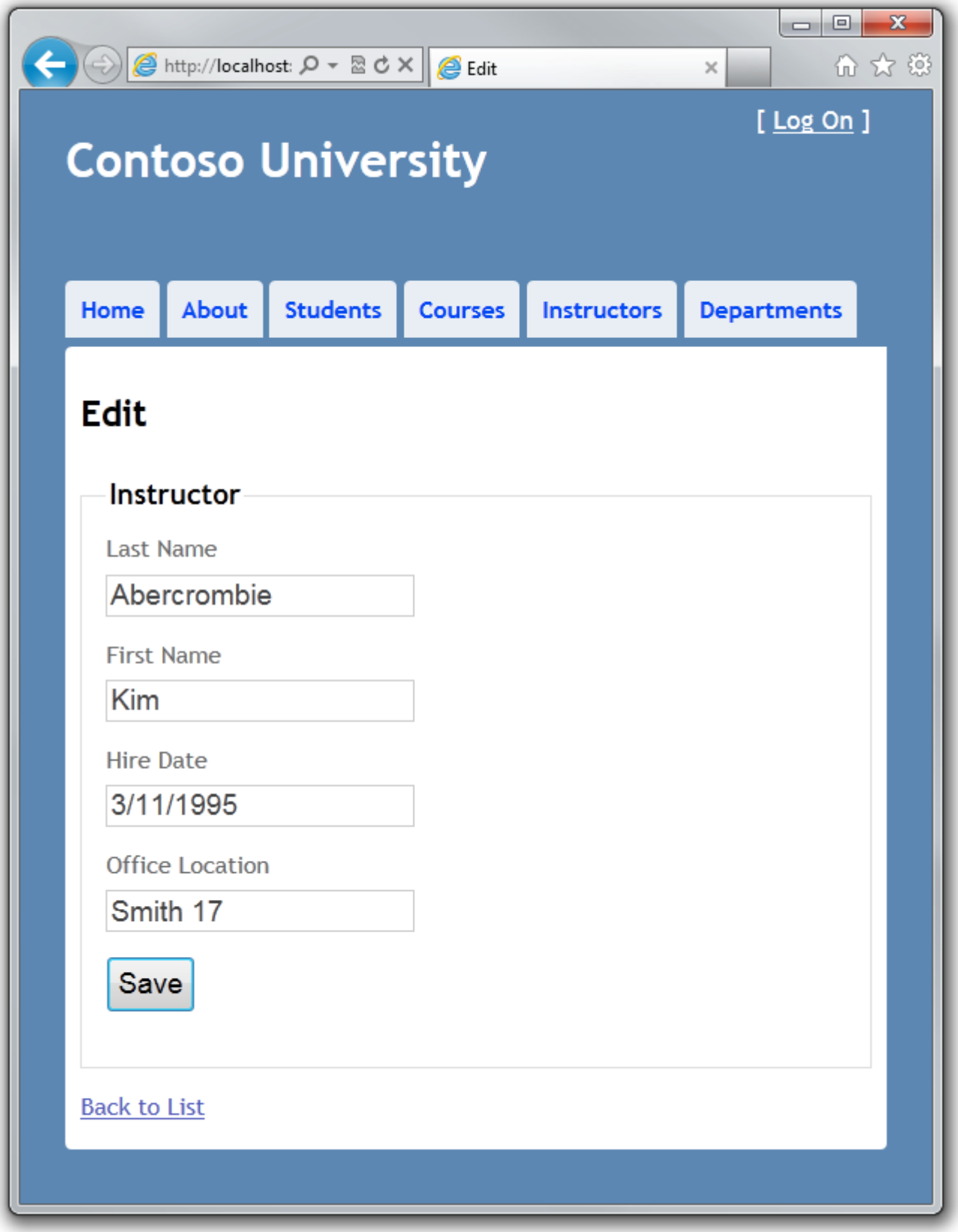
```
if(String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment.Location))
{
    instructorToUpdate.OfficeAssignment=null;
}
```

- Saves the changes to the database.

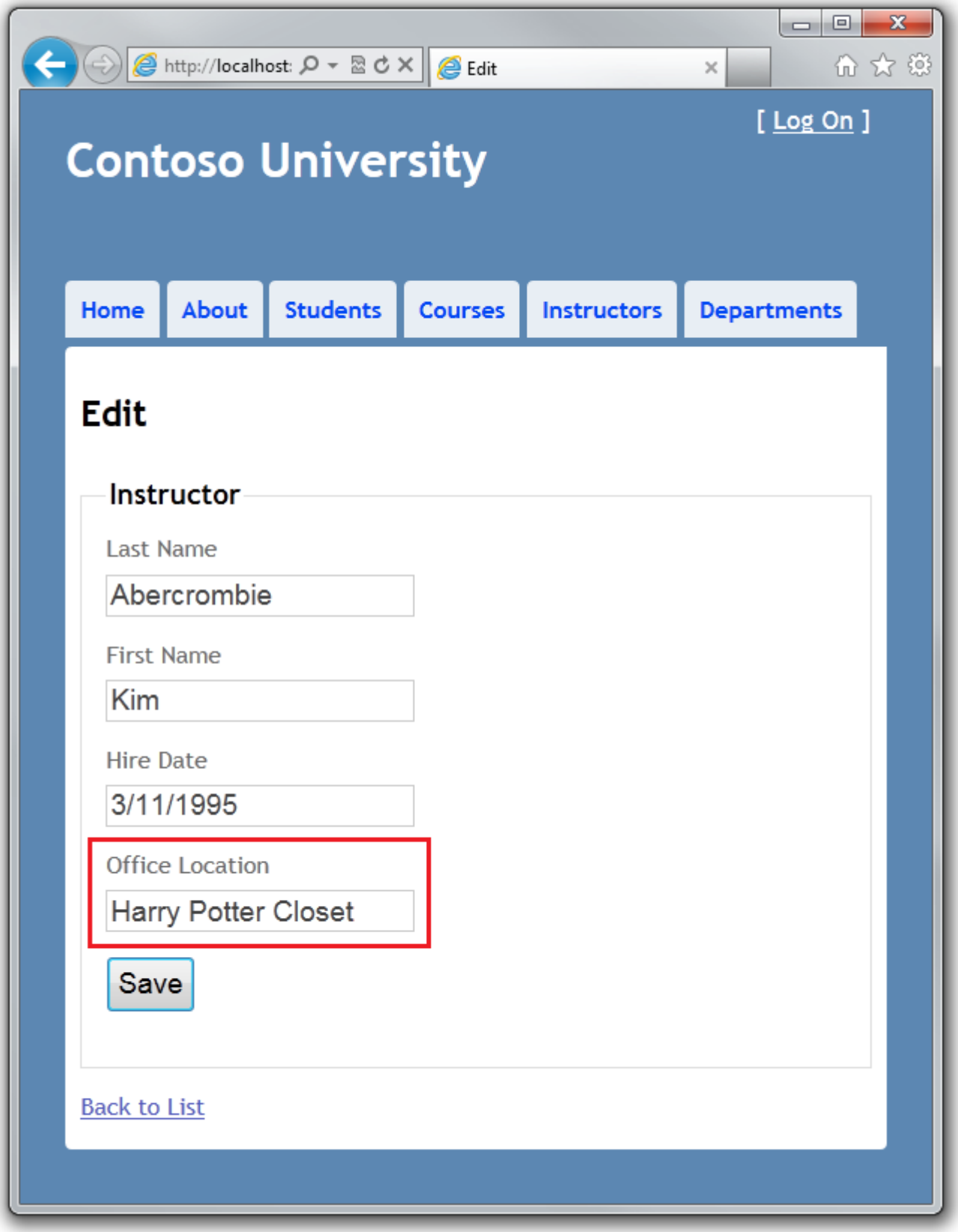
In *Views\Instructor\Edit.cshtml*, after the `div` elements for the **Hire Date** field, add a new field for editing the office location:

```
<divclass="editor-label">
    @Html.LabelFor(model => model.OfficeAssignment.Location)
</div>
<divclass="editor-field">
    @Html.EditorFor(model => model.OfficeAssignment.Location)
    @Html.ValidationMessageFor(model => model.OfficeAssignment.Location)
</div>
```

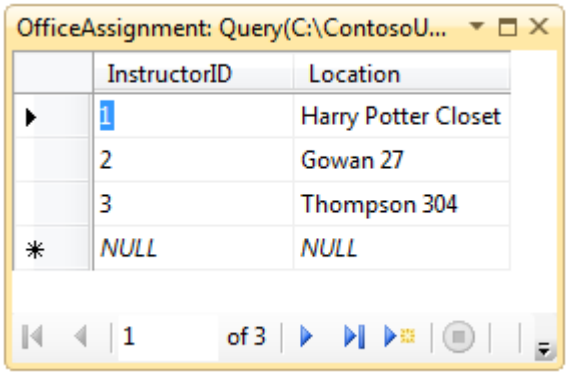
Run the page (select the **Instructors** tab and then click **Edit** on an instructor).


http://localhost:
Edit
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Edit
Instructor
Last Name
Abercrombie
First Name
Kim
Hire Date
3/11/1995
Office Location
Smith 17
Save
Back to List

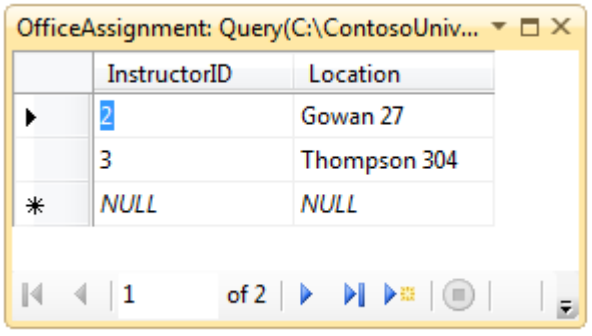
Change the **Office Location** and click **Save**.


http://localhost:
Edit
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Edit
Instructor
Last Name
Abercrombie
First Name
Kim
Hire Date
3/11/1995
Office Location
Harry Potter Closet
Save
Back to List

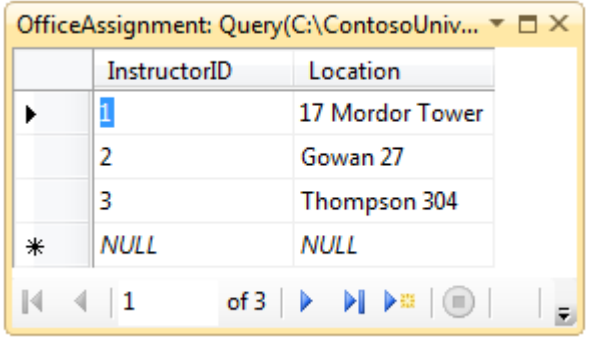
The new location appears on the Index page, and you can see the table row when you open the `OfficeAssignment` table in **Server Explorer**.



Return to the Edit page, clear the **Office Location** and click **Save**. The Index page shows a blank office location and **Server Explorer** shows that the row has been deleted.

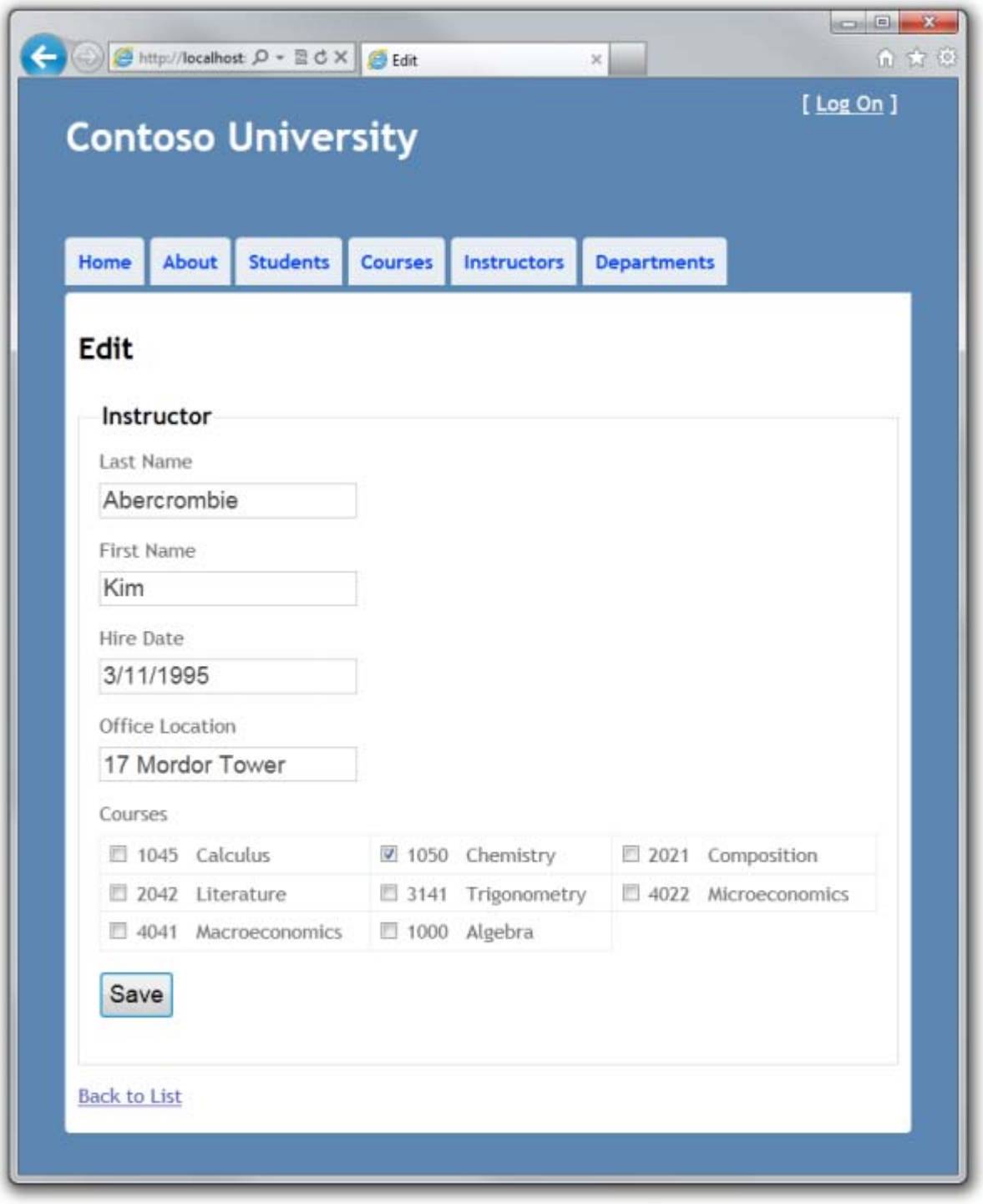


Return to the **Edit** page, enter a new value in the **Office Location** and click **Save**. The **Index** page shows the new location, and **Server Explorer** shows that a row has been created.



Adding Course Assignments to the Instructor Edit Page

Instructors may teach any number of courses. You'll now 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:



The relationship between the `Course` and `Instructor` entities is many-to-many, which means you do not have direct access to the join table or foreign key fields. Instead, you will 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 probably would 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:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.ComponentModel.DataAnnotations;

namespaceContosoUniversity.ViewModels
{
publicclassAssignedCourseData
{
publicintCourseID{get;set;}
publicstringTitle{get;set;}
publicboolAssigned{get;set;}
}
}
```

In *InstructorController.cs*, in the `HttpGetEdit` method, call a new method that provides information for the check box array using the new view model class, as shown in the following example:

```
publicActionResultEdit(int id)
{
Instructor instructor = db.Instructors
.Include(i => i.OfficeAssignment)
.Include(i => i.Courses)
.Where(i => i.InstructorID== id)
.Single();
```

```
PopulateAssignedCourseData(instructor);
returnView(instructor);
}

privatevoidPopulateAssignedCourseData(Instructor instructor)
{
var allCourses = db.Courses;
var instructorCourses =newHashSet<int>(instructor.Courses.Select(c => c.CourseID));
var viewModel =newList<AssignedCourseData>();
foreach(var course in allCourses)
{
        viewModel.Add(newAssignedCourseData
{
CourseID= course.CourseID,
Title= course.Title,
Assigned= instructorCourses.Contains(course.CourseID)
});
}
ViewBag.Courses= viewModel;
}
```

The code in the new 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 of courses that are assigned to the instructor is set to `true`. 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 `HttpPostEdit` method with the following code, which calls a new method that updates the `Courses` navigation property of the `Instructor` entity.

```
[HttpPost]
publicActionResultEdit(int id,FormCollection formCollection,string[] selectedCourses)
{
var instructorToUpdate = db.Instructors
.Include(i => i.OfficeAssignment)
```

```
.Include(i => i.Courses)
.Where(i => i.InstructorID== id)
.Single();
if(TryUpdateModel(instructorToUpdate,"",null,newstring[]{"Courses"}))
{
try
{
if(String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment.Location))
{
                instructorToUpdate.OfficeAssignment=null;
}

UpdateInstructorCourses(selectedCourses, instructorToUpdate);

            db.Entry(instructorToUpdate).State=EntityState.Modified;
            db.SaveChanges();

returnRedirectToAction("Index");
}
catch(DataException)
{
//Log the error (add a variable name after DataException)
ModelState.AddModelError("","Unable to save changes. Try again, and if the problem
persists, see your system administrator.");
}
}
PopulateAssignedCourseData(instructorToUpdate);
returnView(instructorToUpdate);
}

privatevoidUpdateInstructorCourses(string[] selectedCourses,Instructor
instructorToUpdate)
{
if(selectedCourses ==null)
{
        instructorToUpdate.Courses=newList<Course>();
return;
}
```

```
var selectedCoursesHS =newHashSet<string>(selectedCourses);
var instructorCourses =newHashSet<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);
}
}
}
}
```

If no check boxes were selected, the code in `UpdateInstructorCourses` initializes the `Courses` navigation property with an empty collection:

```
if(selectedCourses ==null)
{
    instructorToUpdate.Courses=newList();
return;
}
```

The code then loops through all courses in the database. 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 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:

```
<divclass="editor-field">
<table>
<tr>
            @{
                int cnt = 0;
                List<ContosoUniversity.ViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                foreach (var course in courses) {
                    if (cnt++ % 3 == 0) {
                        @:  </tr><tr>
                    }
                    @: <td>
<inputtype="checkbox"
```

```
name="selectedCourses"
value="@course.CourseID"
                                @(Html.Raw(course.Assigned ? "checked=\"checked\"" :
"")) />
                            @course.CourseID @:  @course.Title
                        @:</td>
                    }
                    @: </tr>
                }
</table>
</div>
```

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 already assigned to the instructor have `checked` attributes, which selects them.

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.

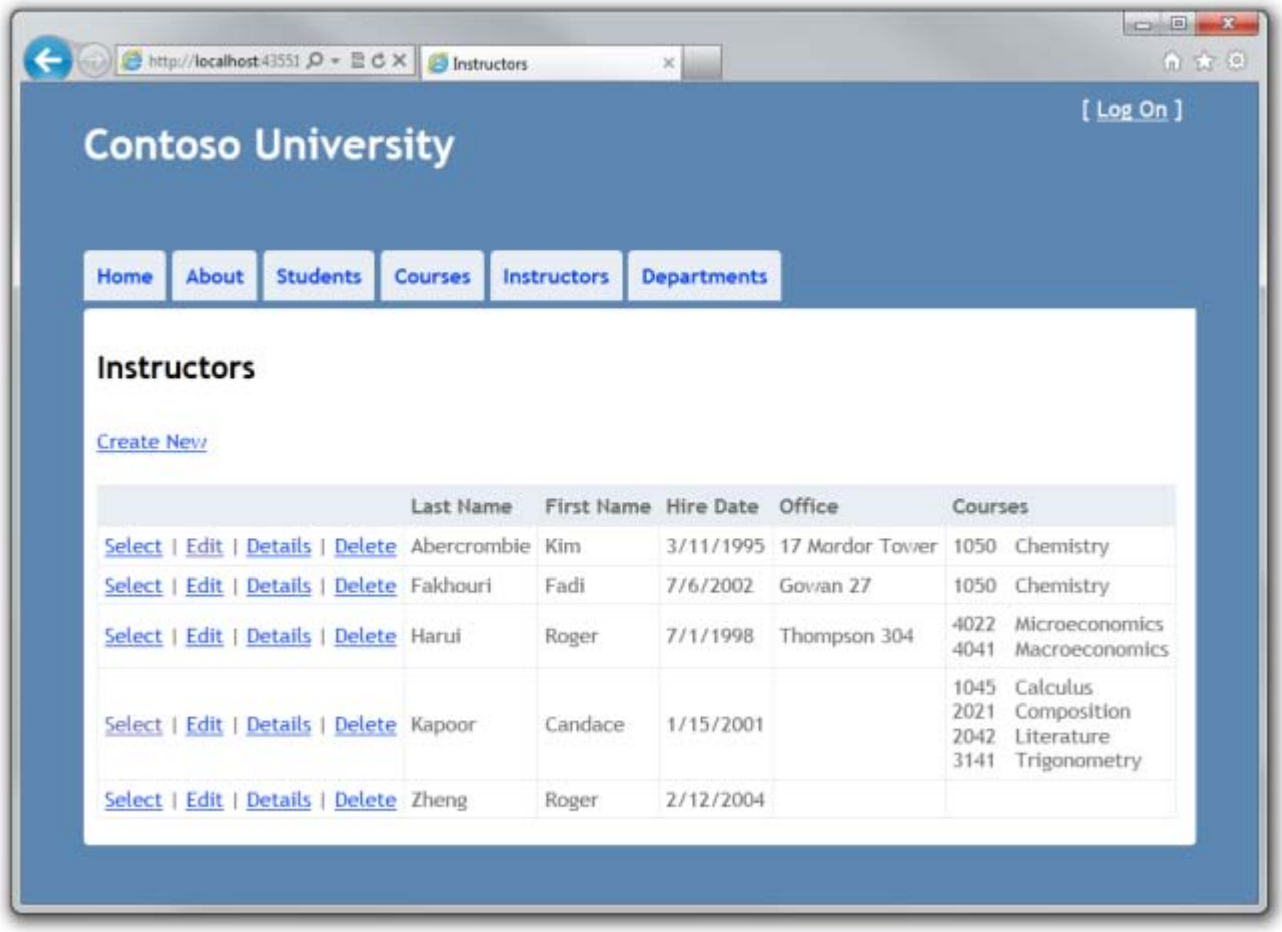
In *Views\Instructor\Index.cshtml*, add a `<th>Courses</th>` heading cell immediately following the `<th>Office</th>` heading, as shown in the following example:

```
<tr>
<th></th>
<th>Last Name</th>
<th>First Name</th>
<th>Hire Date</th>
<th>Office</th>
<th>Courses</th>
</tr>
```

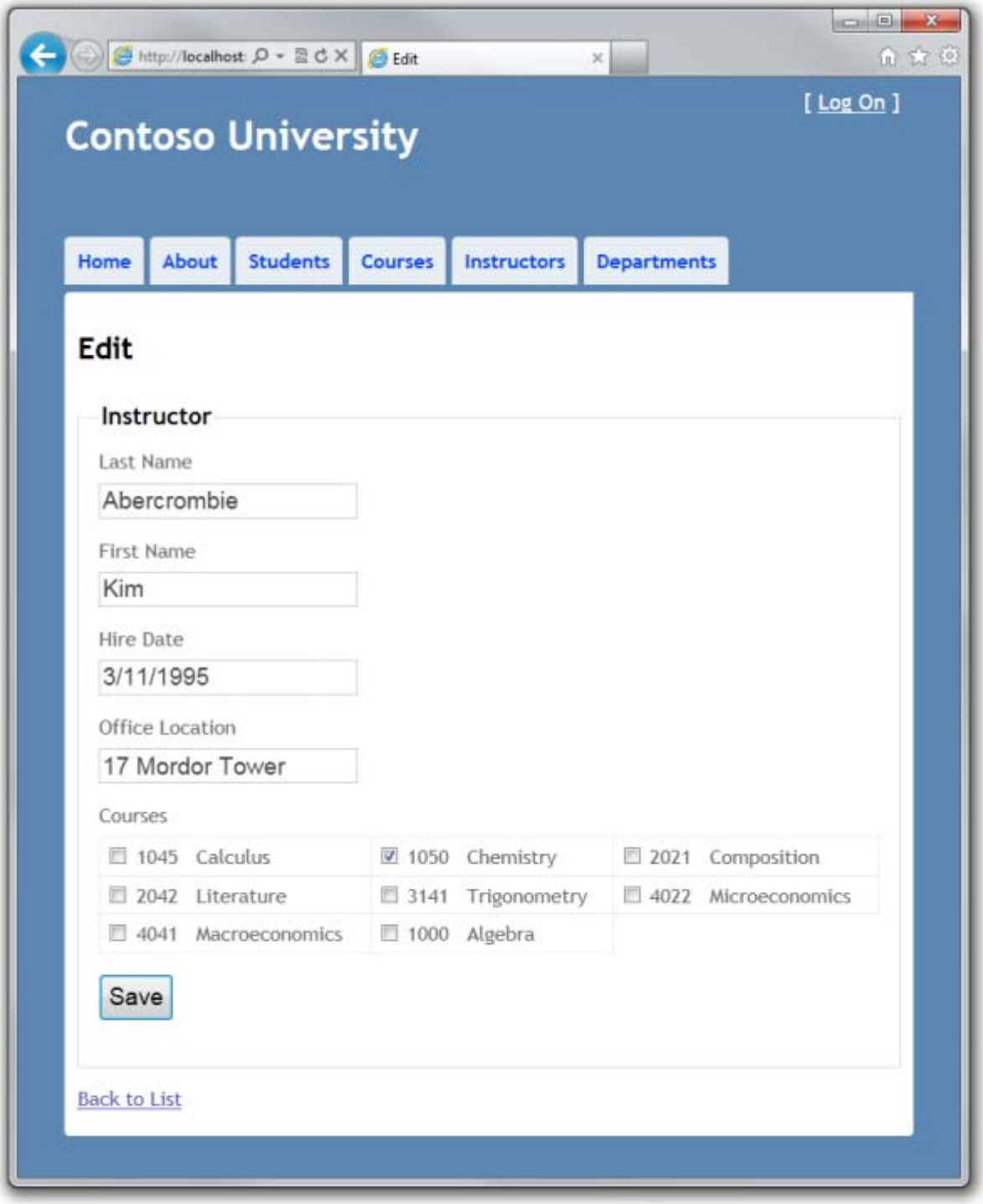
Then add a new detail cell immediately following the office location detail cell:

```
<td>
    @{
        foreach (var course in item.Courses)
        {
            @course.CourseID @:  @course.Title <br/>
        }
    }
</td>
```

Run the **Instructor Index** page to see the courses assigned to each instructor:



Click **Edit** on an instructor to see the Edit page.



Change some course assignments and click **Save**. The changes you make are reflected on the Index page.

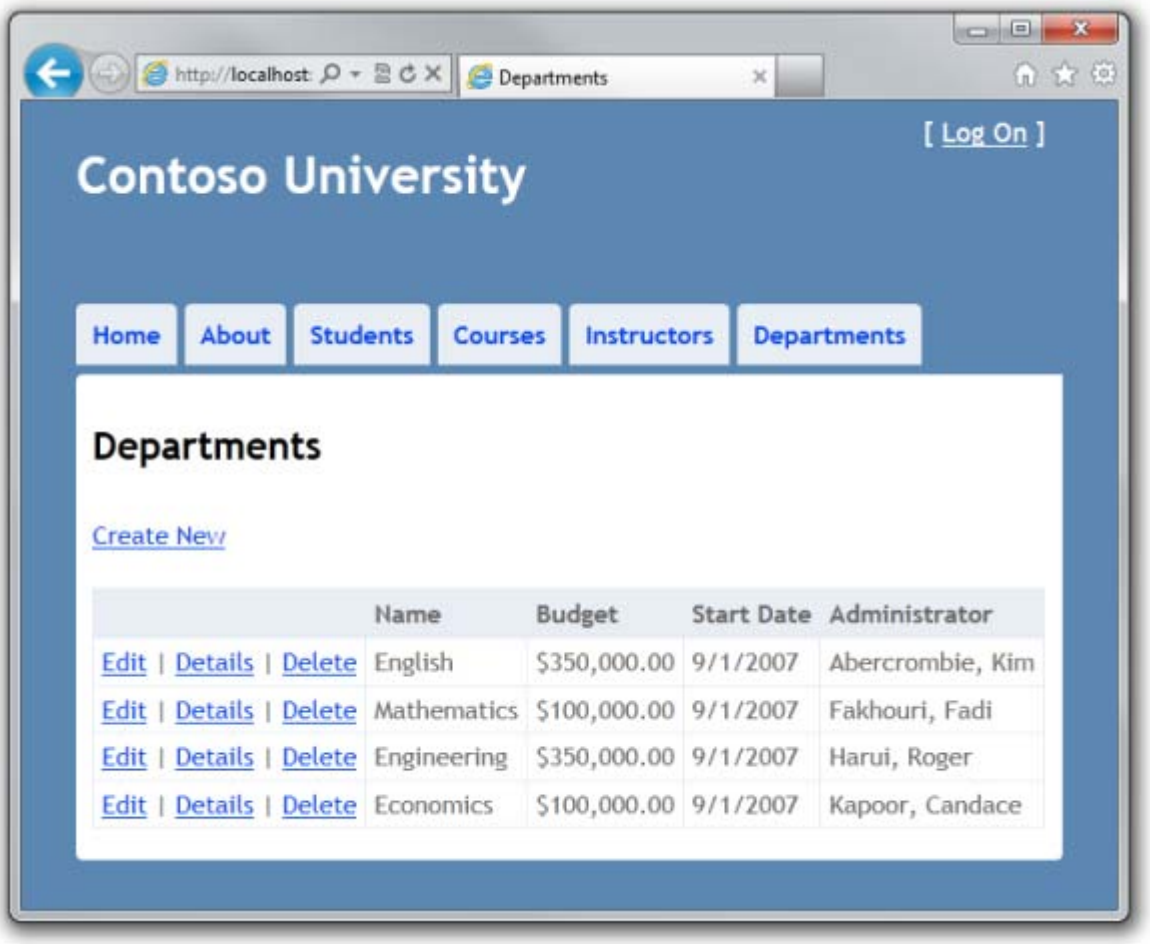
You have now completed this introduction to working with related data. So far in these tutorials you've done a full range of CRUD operations, but you haven't dealt with concurrency issues. The next tutorial will introduce

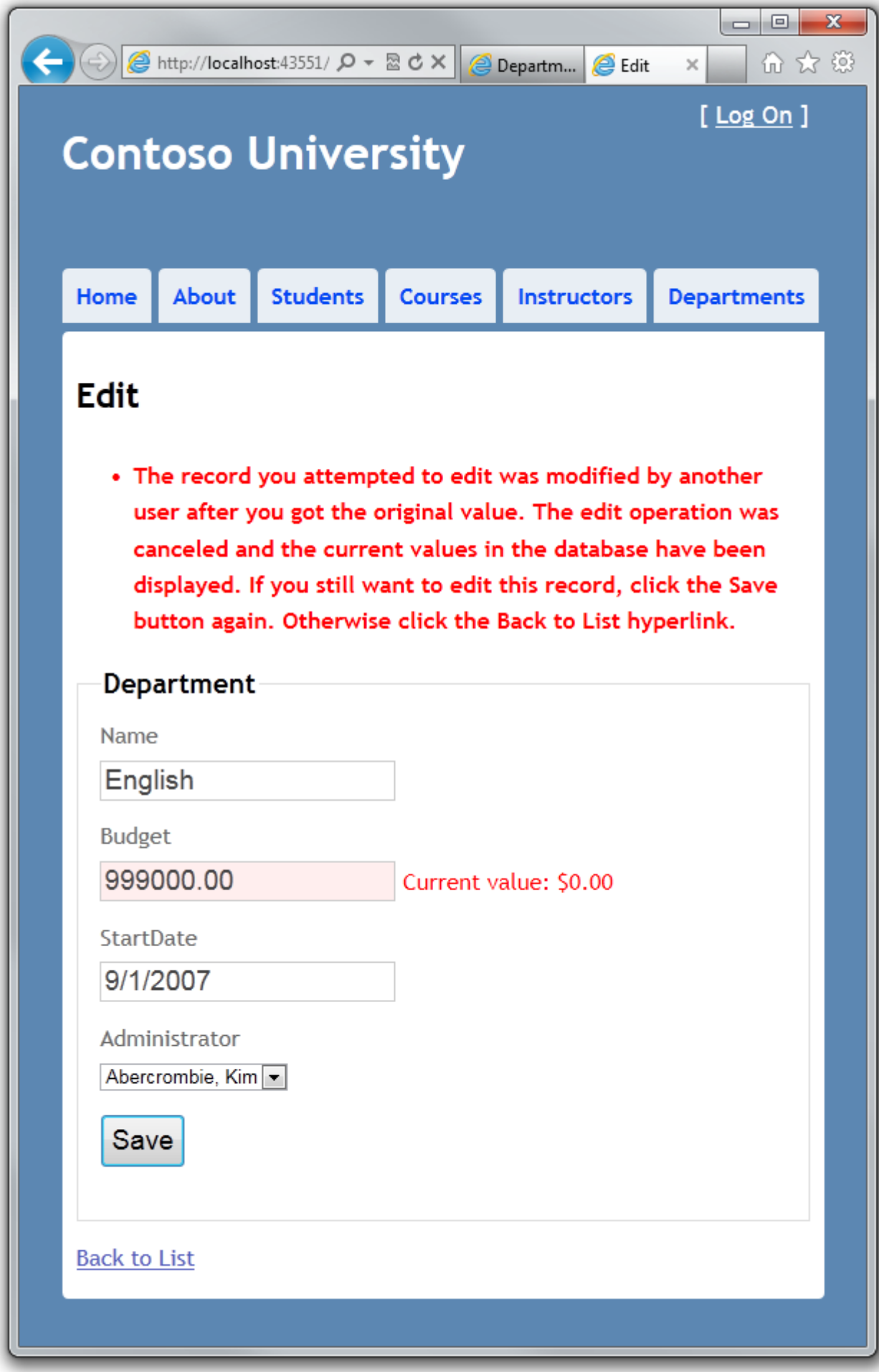
the topic of concurrency, explain options for handling it, and add concurrency handling to the CRUD code you've already written for one entity type.

Links to other Entity Framework resources, can be found at the end of the last tutorial in this series.

Handling Concurrency

In the previous two tutorials you worked with related data. This tutorial shows how to handle concurrency. You'll create web pages that work with the `Department` entity, and the pages that edit and delete `Department` entities will handle concurrency errors. The following illustrations show the Index and Delete pages, including some messages that are displayed if a concurrency conflict occurs.




http://localhost:43551/
Departm...
Edit
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Edit
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
Name
English
Budget
999000.00
Current value: $0.00
StartDate
9/1/2007
Administrator
Abercrombie, Kim
Save
Back to List

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 set up the Entity Framework to detect 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 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.

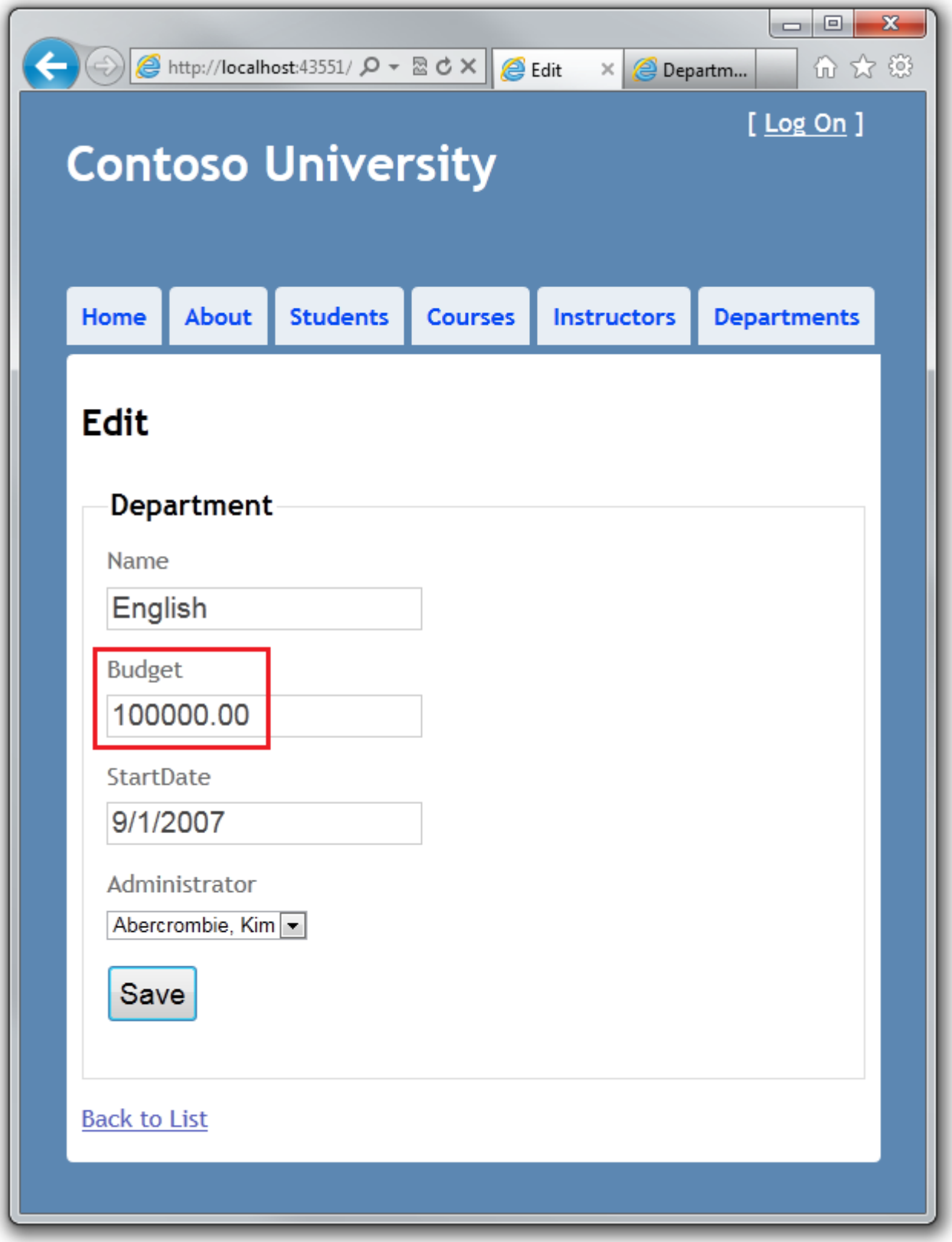
Pessimistic Concurrency (Locking)

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.

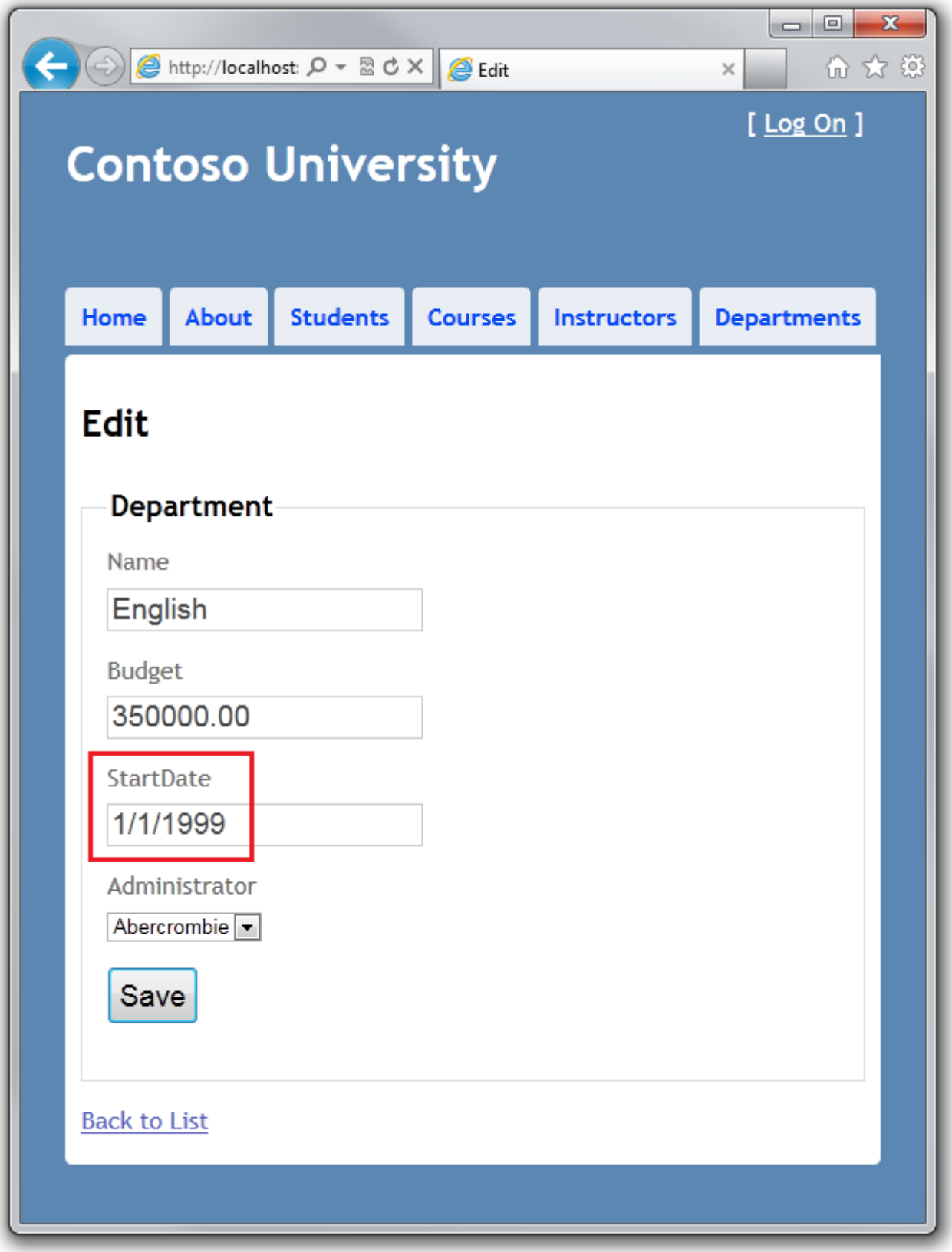
Managing locks has some disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases (that is, it doesn't scale well). For these reasons, not all database management systems 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 $100,000.00. (John administers a competing department and wants to free up money for his own department.)


http://localhost:43551/
Edit
Departm...
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Edit
Department
Name
English
Budget
100000.00
StartDate
9/1/2007
Administrator
Abercrombie, Kim
Save
Back to List

Before John clicks **Save**, Jane runs the same page and changes the **Start Date** field from 9/1/2007 to 1/1/1999. (Jane administers the History department and wants to give it more seniority.)


http://localhost:
Edit
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Edit
Department
Name
English
Budget
350000.00
StartDate
1/1/1999
Administrator
Abercrombie
Save
Back to List

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 1/1/999 and a budget of $100,000.00.

  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 values 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).

- You can let Jane's change overwrite John's change. The next time someone browses the English department, they'll see 1/1/1999 and the restored $350,000.00 value. This is called a *Client Wins* or *Last in Wins* scenario. (The client's values 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 `timestamp`, but it doesn't actually contain a date or time value. Instead, the value is a sequential number that's incremented each time the row is updated. (Therefore the same type can be called `rowversion` in recent versions of SQL Server.) In an `Update` or `Delete` command, the `Where` clause includes the original value of the tracking column. If the row being updated has been changed by another user, the value in that 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. This method is as effective as using a tracking column. However, 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 because it either requires server resources or must be included in the web page itself. Therefore this approach generally not recommended, and it isn't the method used in this tutorial.

In the remainder of this tutorial you'll add a tracking property to the `Department` entity, create a controller and views, and test to verify that everything works correctly.

Note If you were implementing concurrency without a tracking column, you would have to mark all non-primary-key properties in the entity for concurrency tracking by adding the `ConcurrencyCheck` attribute to them. That change would enable the Entity Framework to include all columns in the SQL `WHERE` clause of `UPDATE` statements.

Adding a Tracking Property to the Department Entity

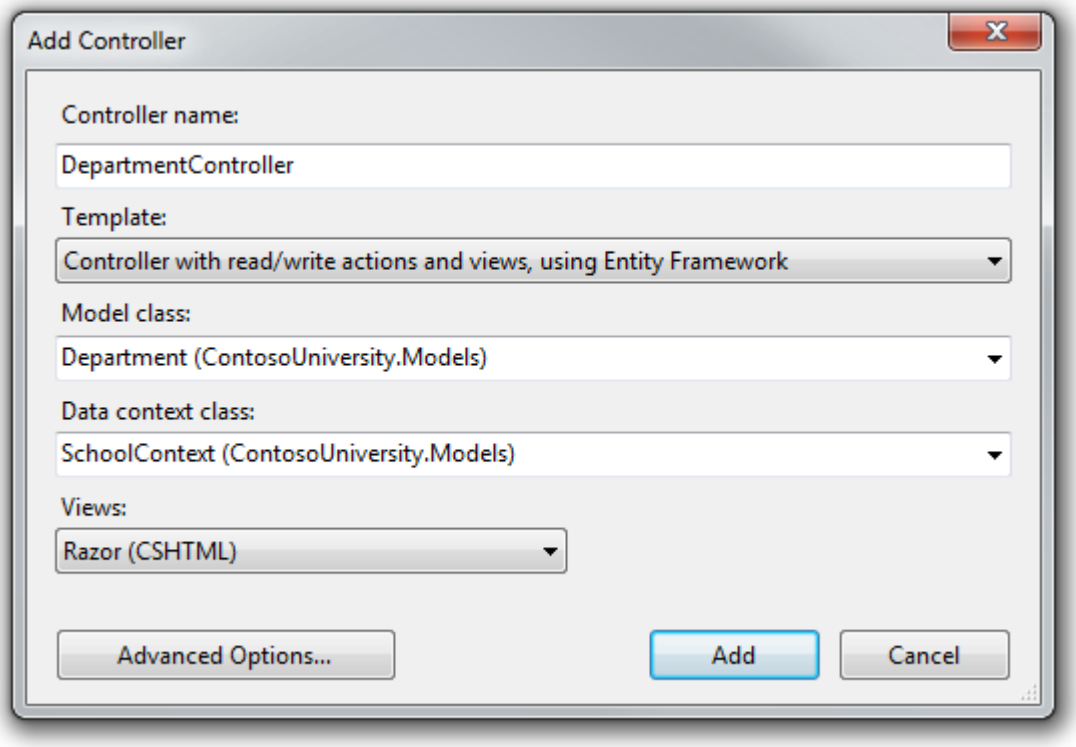
In *Models\Department.cs*, add a tracking property:

```
[Timestamp]
publicByte[]Timestamp{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.

Creating a Department Controller

Create a `Department` controller and views the same way you did the other controllers, using the following settings:



In *Controllers\DepartmentController.cs*, add a `using` statement:

```
usingSystem.Data.Entity.Infrastructure;
```

Change "LastName" to "FullName" everywhere in this file (four occurrences) so that the department administrator drop-down lists will contain the full name of the instructor rather than just the last name.

Replace the existing code for the `HttpPostEdit` method with the following code:

```
[HttpPost]
publicActionResultEdit(Department department)
{
try
{
```

```
if(ModelState.IsValid)
{
            db.Entry(department).State=EntityState.Modified;
            db.SaveChanges();
returnRedirectToAction("Index");
}
}
catch(DbUpdateConcurrencyException ex)
{
var entry = ex.Entries.Single();
var databaseValues =(Department)entry.GetDatabaseValues().ToObject();
var clientValues =(Department)entry.Entity;
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.Timestamp= databaseValues.Timestamp;
}
catch(DataException)
{
//Log the error (add a variable name after Exception)
ModelState.AddModelError(string.Empty,"Unable to save changes. Try again, and if the
problem persists contact your system administrator.");
}
```

```
ViewBag.InstructorID=newSelectList(db.Instructors,"InstructorID","FullName",
department.InstructorID);
returnView(department);
}
```

The view will store the original timestamp value in a hidden field. When the model binder creates the `department` instance, that object will have the original `Timestamp` 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 `Timestamp` value.

If zero rows are affected by the `UPDATE` command, the Entity Framework throws a `DbUpdateConcurrencyException` exception, and the code in the `catch` block gets the affected `Department` entity from the exception object. This entity has both the values read from the database and the new values entered by the user:

```
var entry = ex.Entries.Single();
var databaseValues =(Department)entry.GetDatabaseValues().ToObject();
var clientValues =(Department)entry.Entity;
```

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 `Timestamp` value of the `Department` object to the new value retrieved from the database. This new `Timestamp` 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 `Timestamp` property value, immediately following the hidden field for the `DepartmentID` property:

```
@Html.HiddenFor(model => model.Timestamp)
```

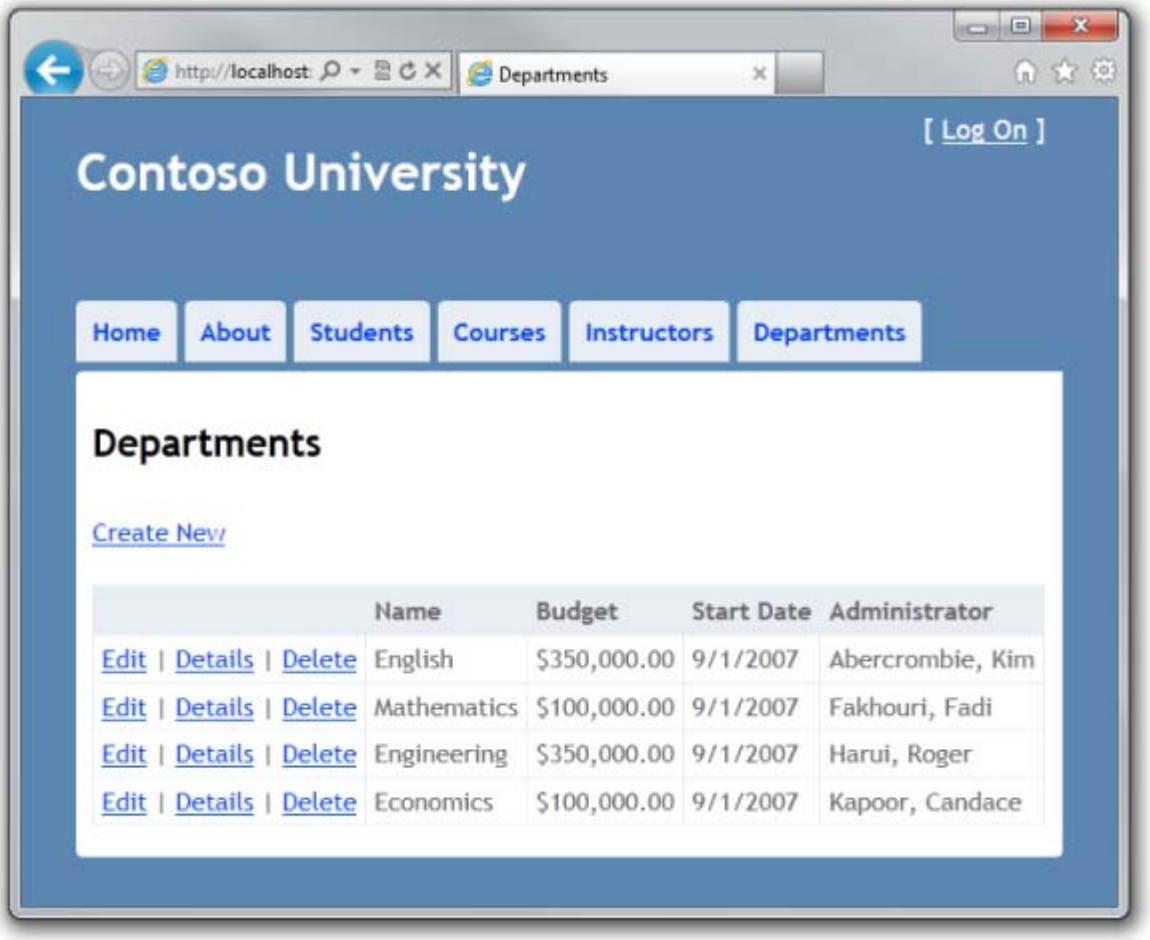
In *Views\Department\Index.cshtml*, replace the existing code with the following code to move row links to the left and change the page title and column headings to display `FullName` instead of `LastName` in the **Administrator** column:

```
@modelIEnumerable<ContosoUniversity.Models.Department>

@{
ViewBag.Title="Departments";
}

<h2>Departments</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
<tr>
<th></th>
<th>Name</th>
<th>Budget</th>
<th>Start Date</th>
<th>Administrator</th>
</tr>

@foreach(var item inModel){
<tr>
<td>
```

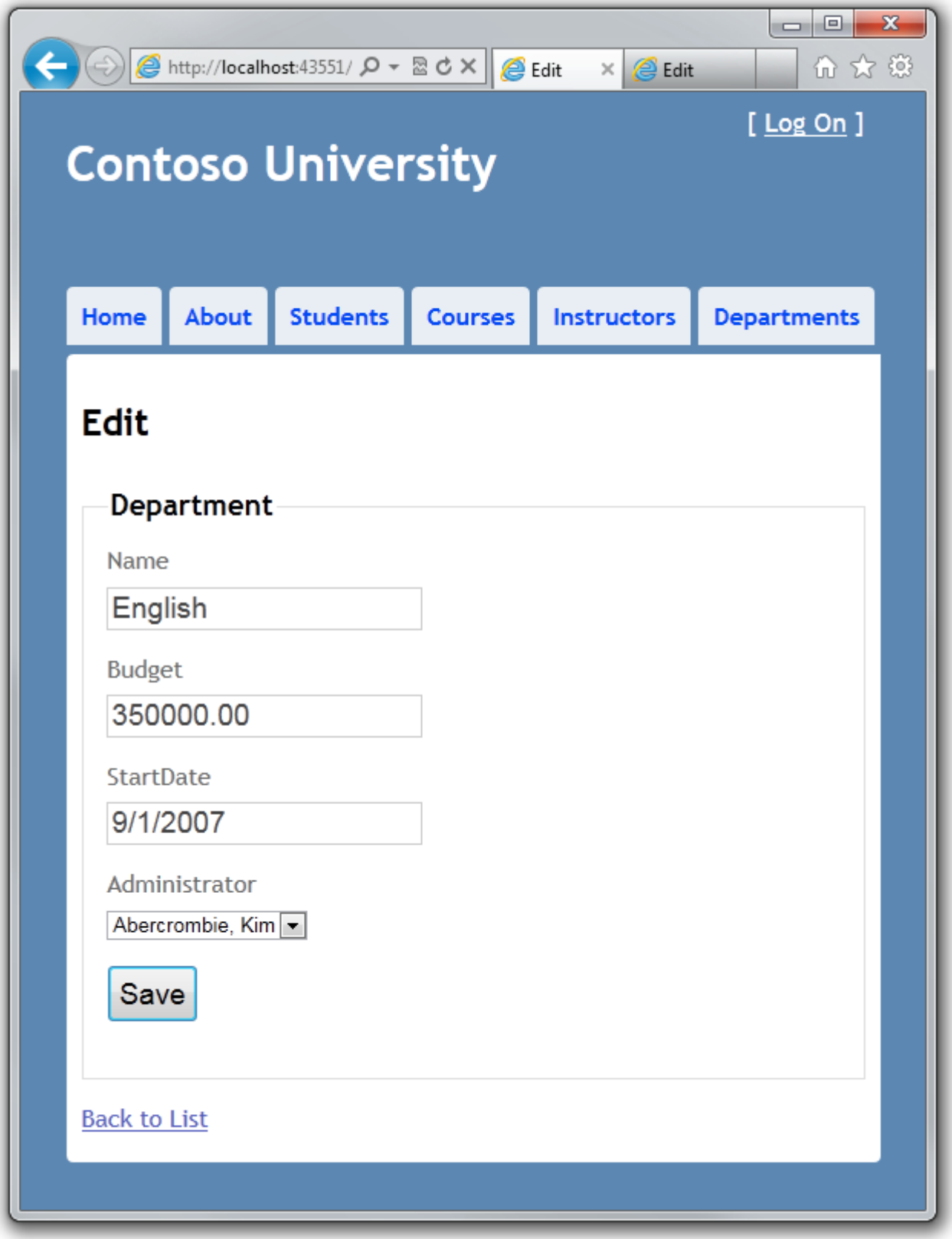
```
@Html.ActionLink("Edit","Edit",new{ id=item.DepartmentID})|
@Html.ActionLink("Details","Details",new{ id=item.DepartmentID})|
@Html.ActionLink("Delete","Delete",new{ id=item.DepartmentID})
</td>
<td>
            @Html.DisplayFor(modelItem => item.Name)
</td>
<td>
@Html.DisplayFor(modelItem => item.Budget)
</td>
<td>
            @Html.DisplayFor(modelItem => item.StartDate)
</td>
<td>
@Html.DisplayFor(modelItem => item.Administrator.FullName)
</td>
</tr>
}

</table>
```

Testing Optimistic Concurrency Handling

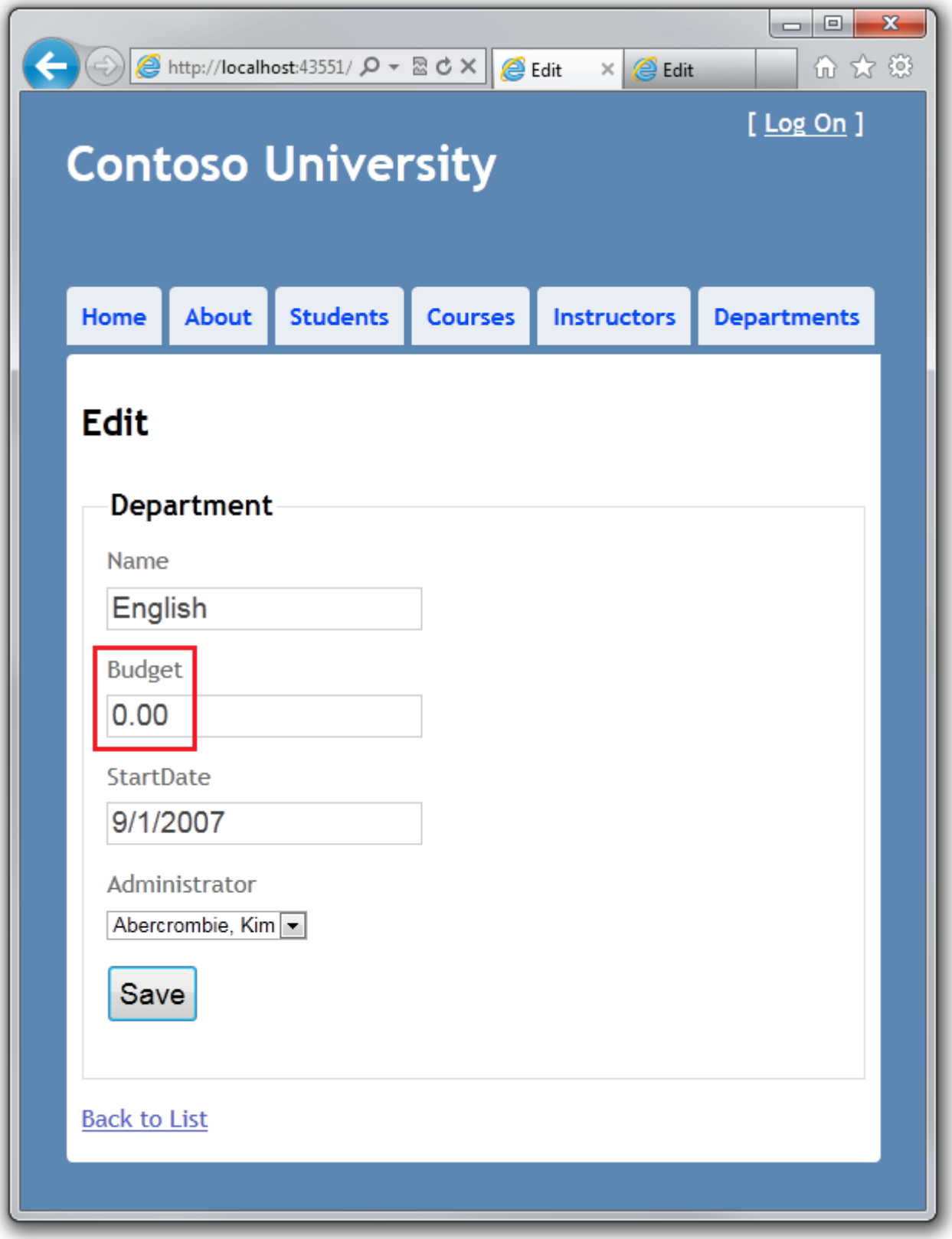
Run the site and click **Departments**:



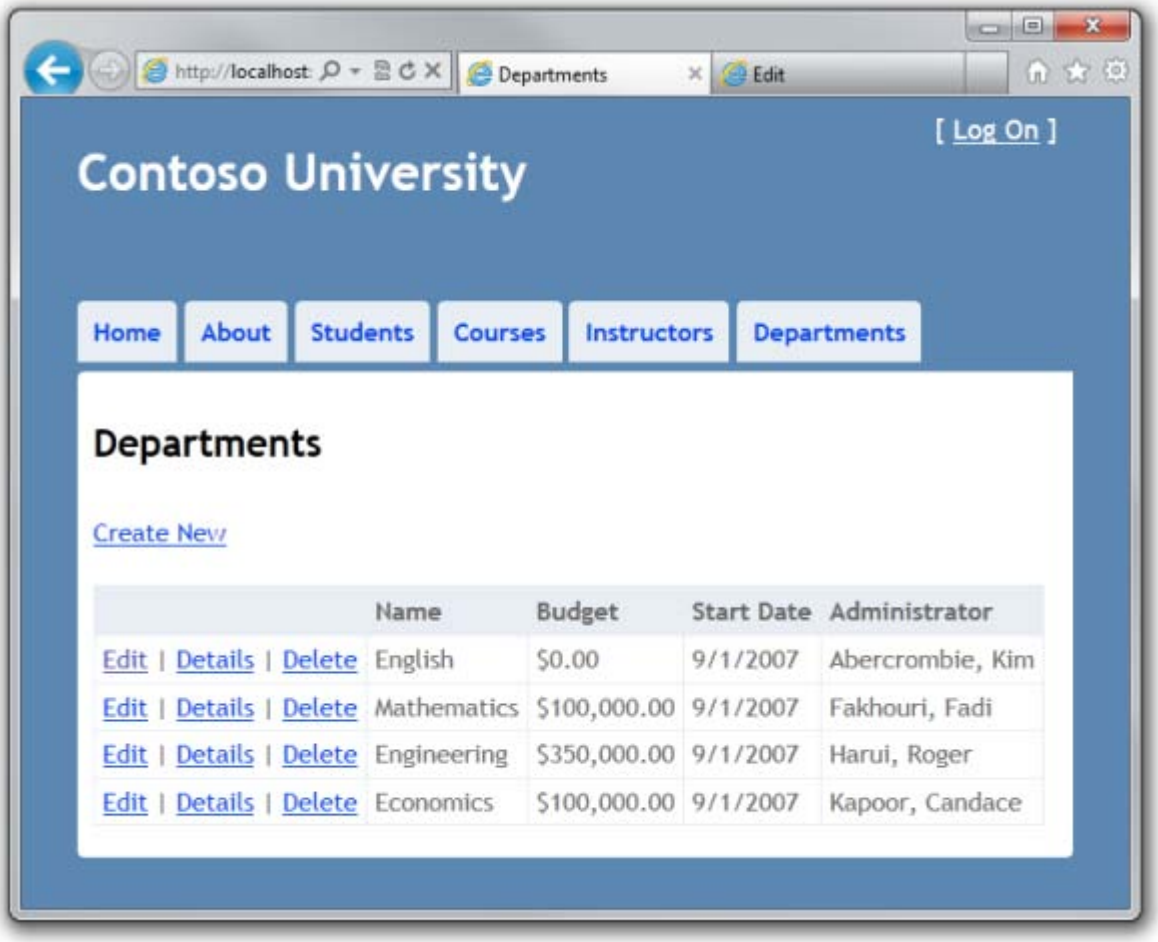
Click an **Edit** hyperlink and then open a new browser window and go to the same URL in that window. The windows display the same information.


http://localhost:43551/
Edit
Edit
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Edit
Department
Name
English
Budget
350000.00
StartDate
9/1/2007
Administrator
Abercrombie, Kim
Save
Back to List

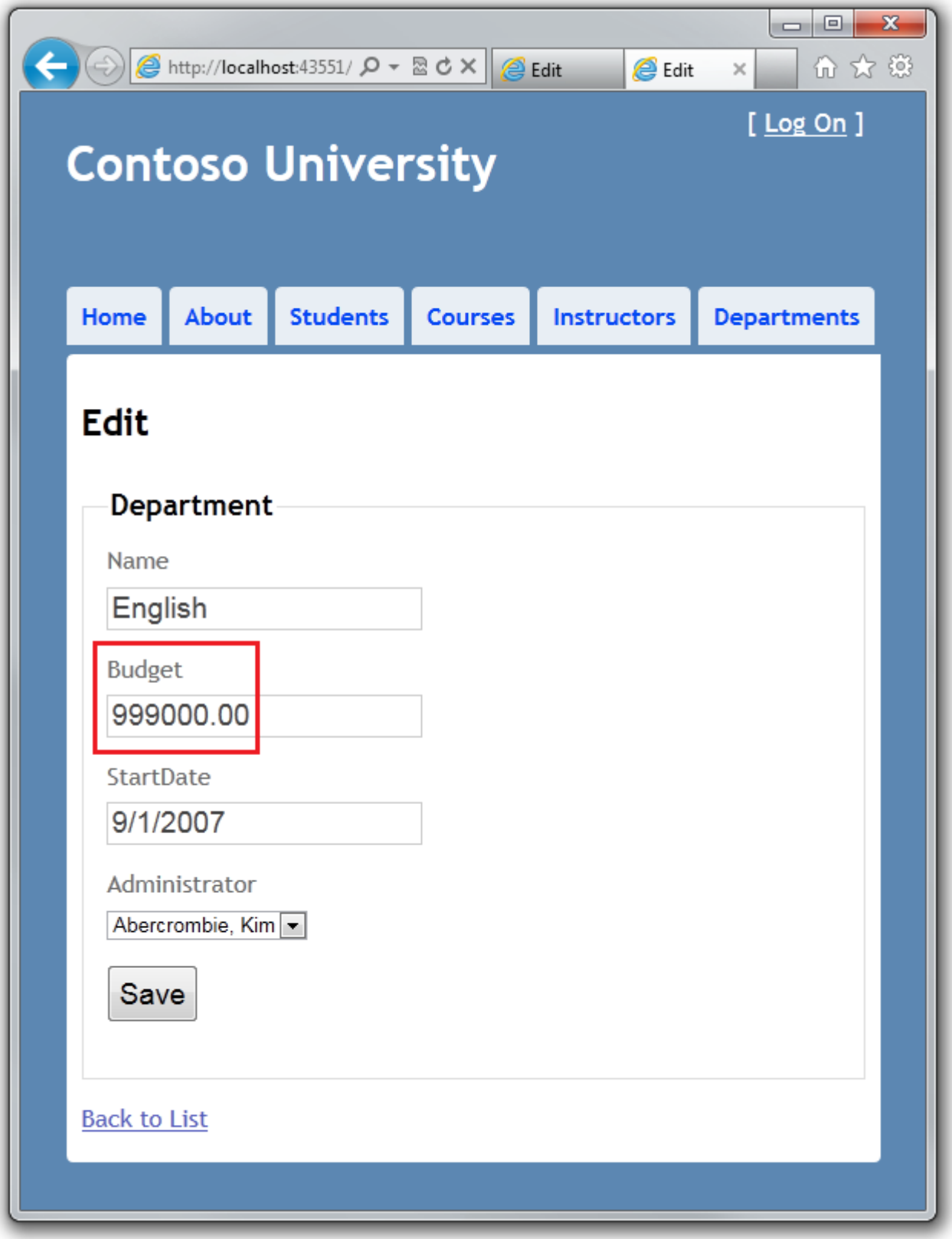
Change a field in the first browser window and click **Save**.


http://localhost:43551/
Edit
Edit
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Edit
Department
Name
English
Budget
0.00
StartDate
9/1/2007
Administrator
Abercrombie, Kim
Save
Back to List

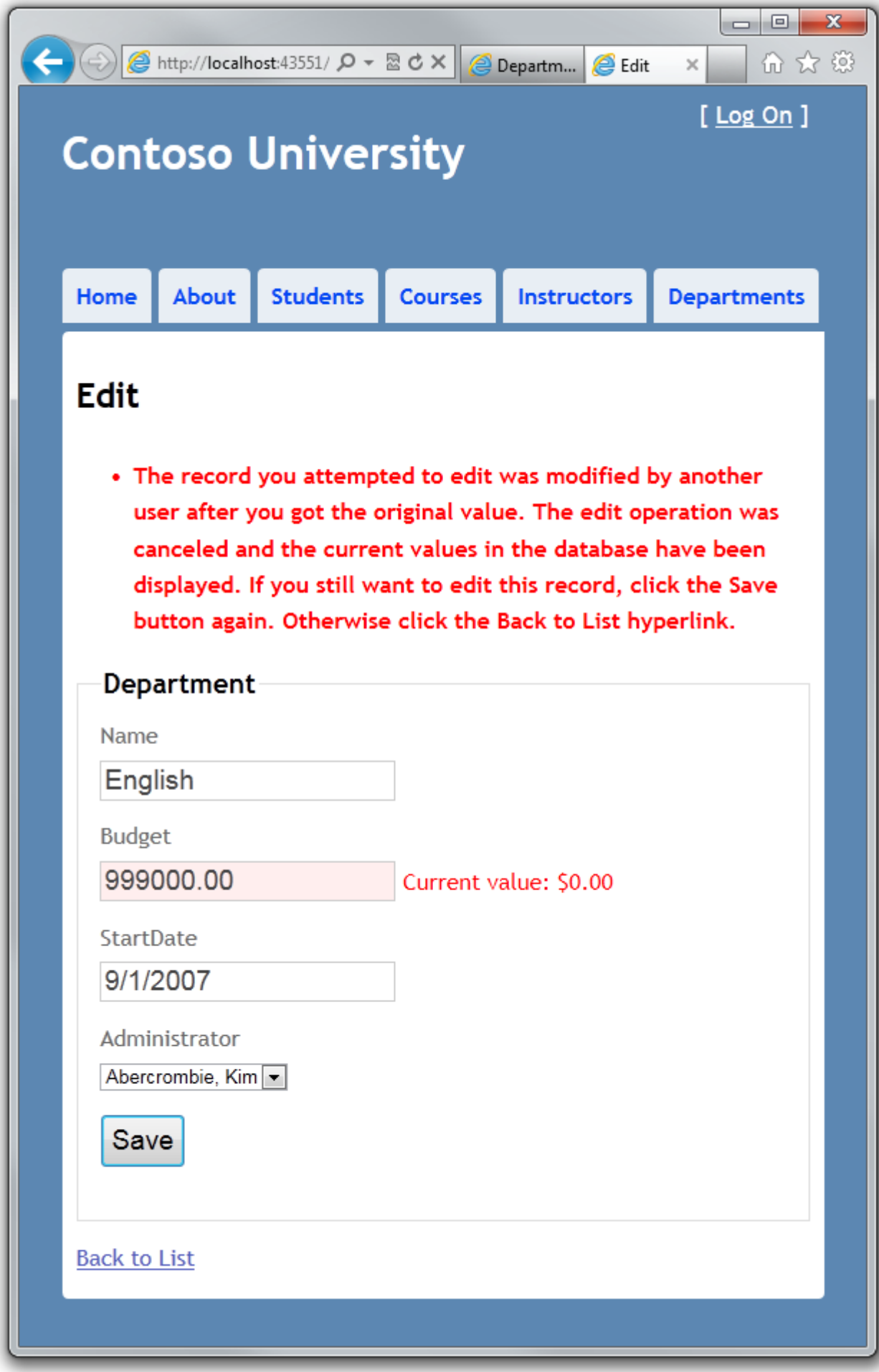
The browser shows the Index page with the changed value.



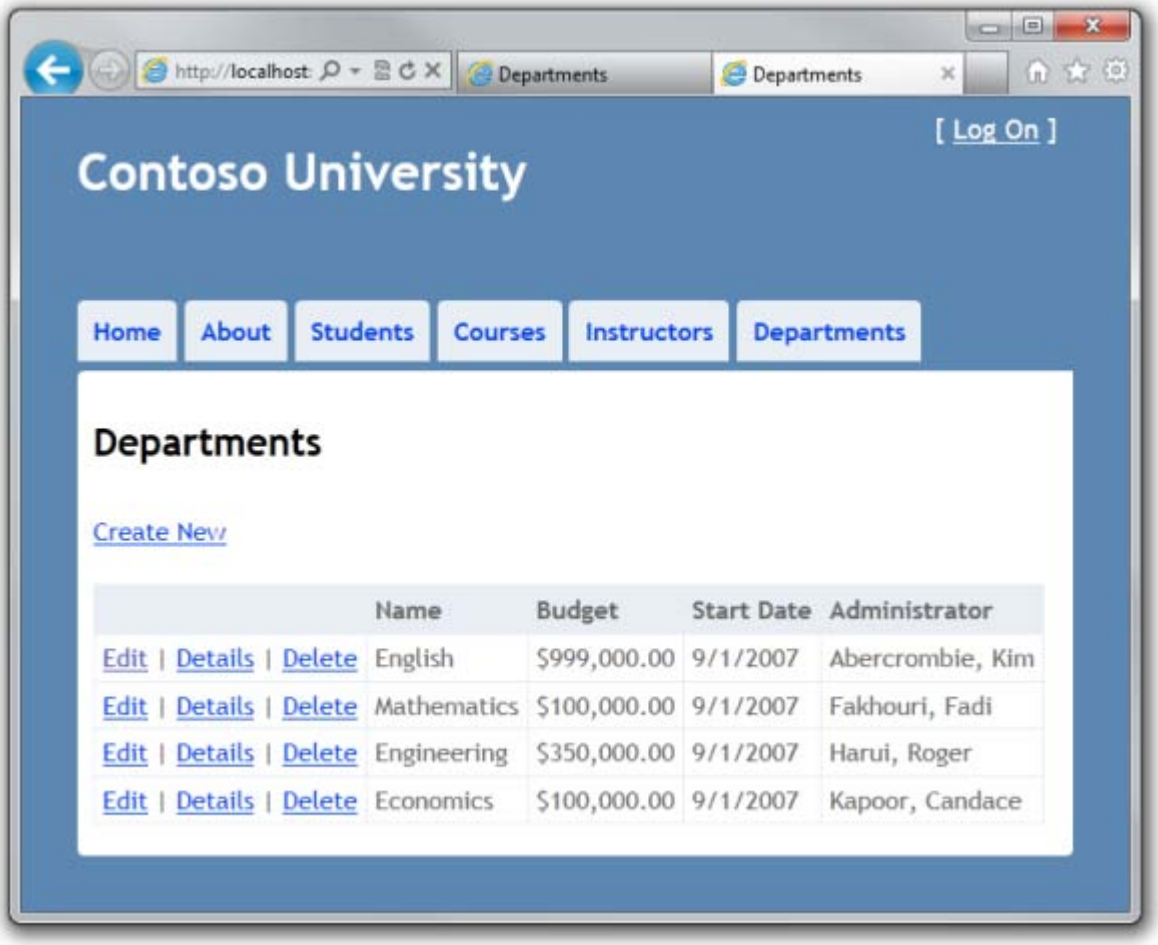
Change the same field to a different value in the second browser window.


http://localhost:43551/
Edit
Edit
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Edit
Department
Name
English
Budget
999000.00
StartDate
9/1/2007
Administrator
Abercrombie, Kim
Save
Back to List

Click **Save** in the second browser window. You see an error message:


http://localhost:43551/
Departm...
Edit
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Edit
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
Name
English
Budget
999000.00
Current value: $0.00
StartDate
9/1/2007
Administrator
Abercrombie, Kim
Save
Back to List

Click **Save** again. The value you entered in the second browser is saved in the database and you see that value when the Index page appears.



Adding a Delete Page

For the Delete page, the Entity Framework detects concurrency conflicts in a similar manner. When the `HttpGetDelete` method displays the confirmation view, the view includes the original `Timestamp` value in a hidden field. That value is then available to the `HttpPostDelete` 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 `Timestamp` 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 `HttpGetDelete` method is called with an error flag set to `true` in order to redisplay the confirmation page with an error message.

In *DepartmentController.cs*, replace the `HttpGetDelete` method with the following code:

```
publicActionResultDelete(int id,bool? concurrencyError)
{
if(concurrencyError.GetValueOrDefault())
{
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.";
}

Department department = db.Departments.Find(id);
returnView(department);
}
```

The method accepts an optional parameter that indicates whether the page is being redisplayed after a concurrency error. If this flag is `true`, error message text is sent to the view using a `ViewBag` property.

Replace the code in the `HttpPostDelete` method (named `DeleteConfirmed`) with the following code:

```
[HttpPost,ActionName("Delete")]
publicActionResultDeleteConfirmed(Department department)
{
try
{
        db.Entry(department).State=EntityState.Deleted;
        db.SaveChanges();
returnRedirectToAction("Index");
}
catch(DbUpdateConcurrencyException)
{
returnRedirectToAction("Delete",
newSystem.Web.Routing.RouteValueDictionary{{"concurrencyError",true}});
}
catch(DataException)
{
```

```
//Log the error (add a variable name after Exception)
ModelState.AddModelError(string.Empty,"Unable to save changes. Try again, and if the
problem persists contact your system administrator.");
returnView(department);
}
}
```

In the scaffolded code that you just replaced, this method accepted only a record ID:

```
publicActionResultDeleteConfirmed(int id)
```

You've changed this parameter to a `Department` entity instance created by the model binder. This gives you access to the `Timestamp` property value in addition to the record key.

```
publicActionResultDeleteConfirmed(Department department)
```

If a concurrency error is caught, the code redisplays 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 to make some formatting changes and add an error message field:

```
@modelContosoUniversity.Models.Department

@{
ViewBag.Title="Delete";
}

<h2>Delete</h2>

<p class="error">@ViewBag.ConcurrencyErrorMessage</p>

<h3>Are you sure you want to deletethis?</h3>
<fieldset>
<legend>Department</legend>
```

```
<div class="display-label">
        @Html.LabelFor(model => model.Name)
</div>
<div class="display-field">
@Html.DisplayFor(model => model.Name)
</div>

<div class="display-label">
        @Html.LabelFor(model => model.Budget)
</div>
<div class="display-field">
@Html.DisplayFor(model => model.Budget)
</div>

<div class="display-label">
        @Html.LabelFor(model => model.StartDate)
</div>
<div class="display-field">
@Html.DisplayFor(model => model.StartDate)
</div>

<div class="display-label">
        @Html.LabelFor(model => model.InstructorID)
</div>
<div class="display-field">
@Html.DisplayFor(model => model.Administrator.FullName)
</div>
</fieldset>
@using(Html.BeginForm()){
@Html.HiddenFor(model => model.DepartmentID)
@Html.HiddenFor(model => model.Timestamp)
<p>
<input type="submit" value="Delete"/>|
@Html.ActionLink("Back to List","Index")
</p>
}
```

This code adds an error message between the `h2` and `h3` headings:

```
<pclass="error">@ViewBag.ConcurrencyErrorMessage</p>
```

It replaces `LastName` with `FullName` in the `Administrator` field:

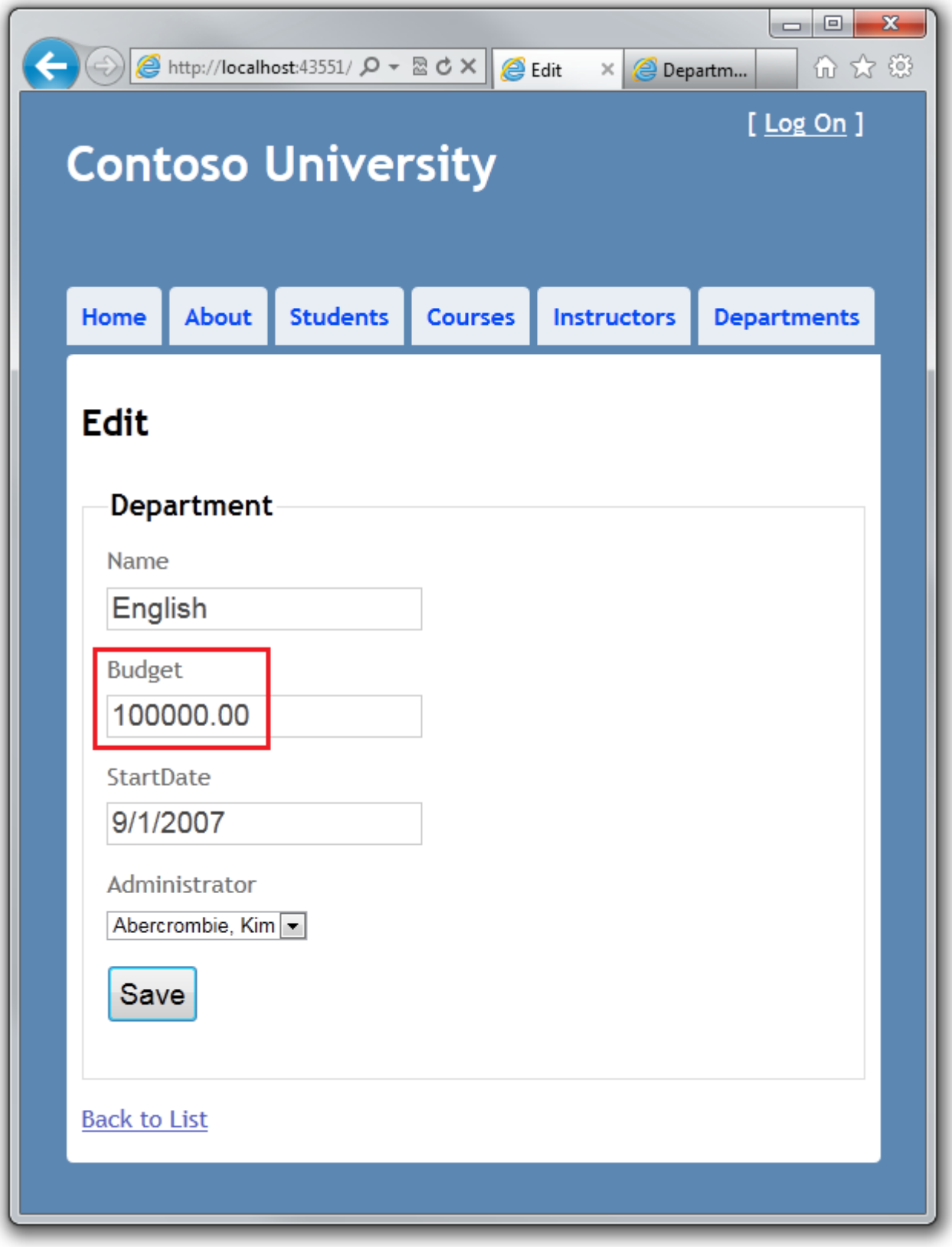
```
<divclass="display-label">
    @Html.LabelFor(model => model.InstructorID)
</div>
<divclass="display-field">
    @Html.DisplayFor(model => model.Administrator.FullName)
</div>
```

Finally, it adds hidden fields for the `DepartmentID` and `Timestamp` properties after the `Html.BeginForm` statement:

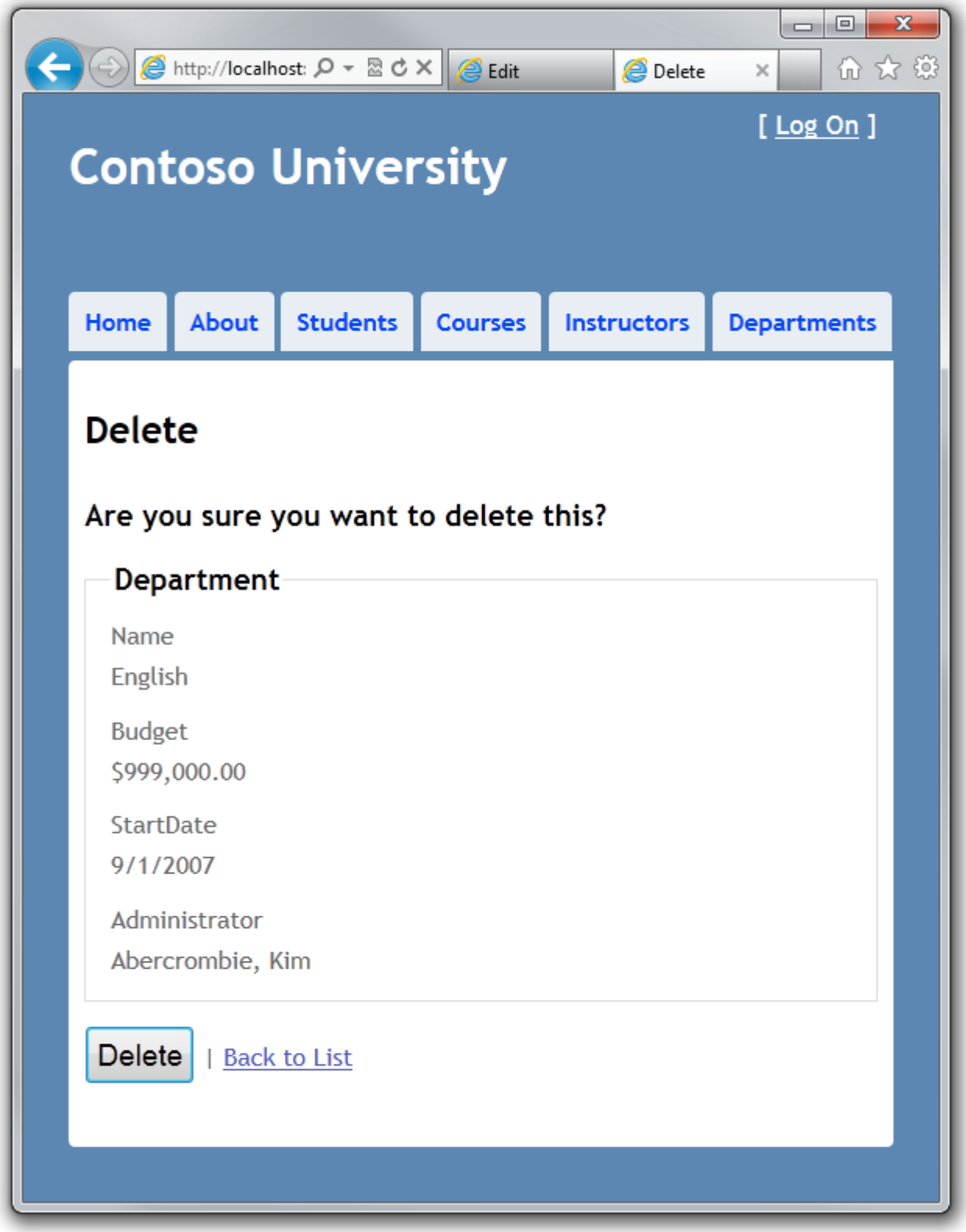
```
@Html.HiddenFor(model => model.DepartmentID)
@Html.HiddenFor(model => model.Timestamp)
```

Run the Departments Index page and open a second browser window with the same URL.

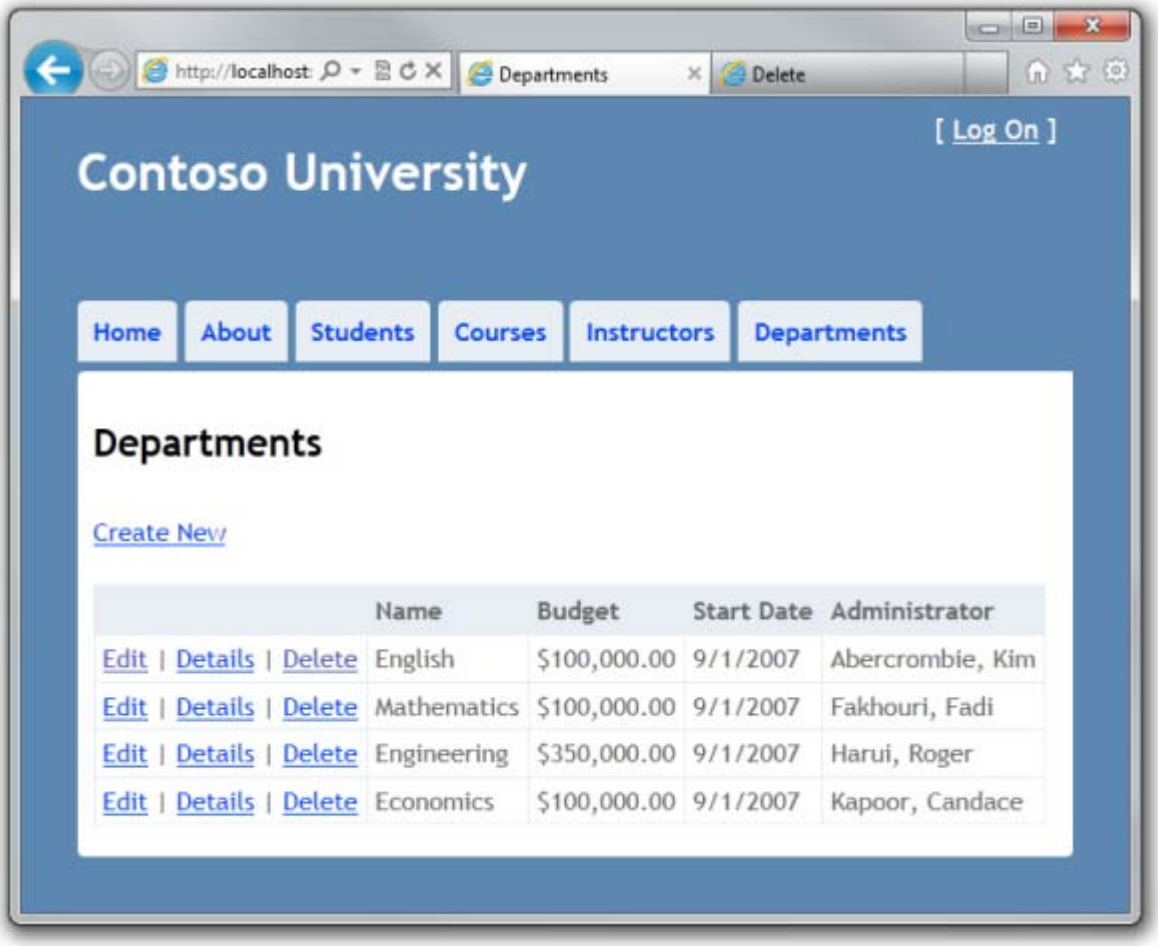
In the first window, click **Edit** on a department and change one of the values, but don't click **Save** yet:


http://localhost:43551/
Edit
Departm...
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Edit
Department
Name
English
Budget
100000.00
StartDate
9/1/2007
Administrator
Abercrombie, Kim
Save
Back to List

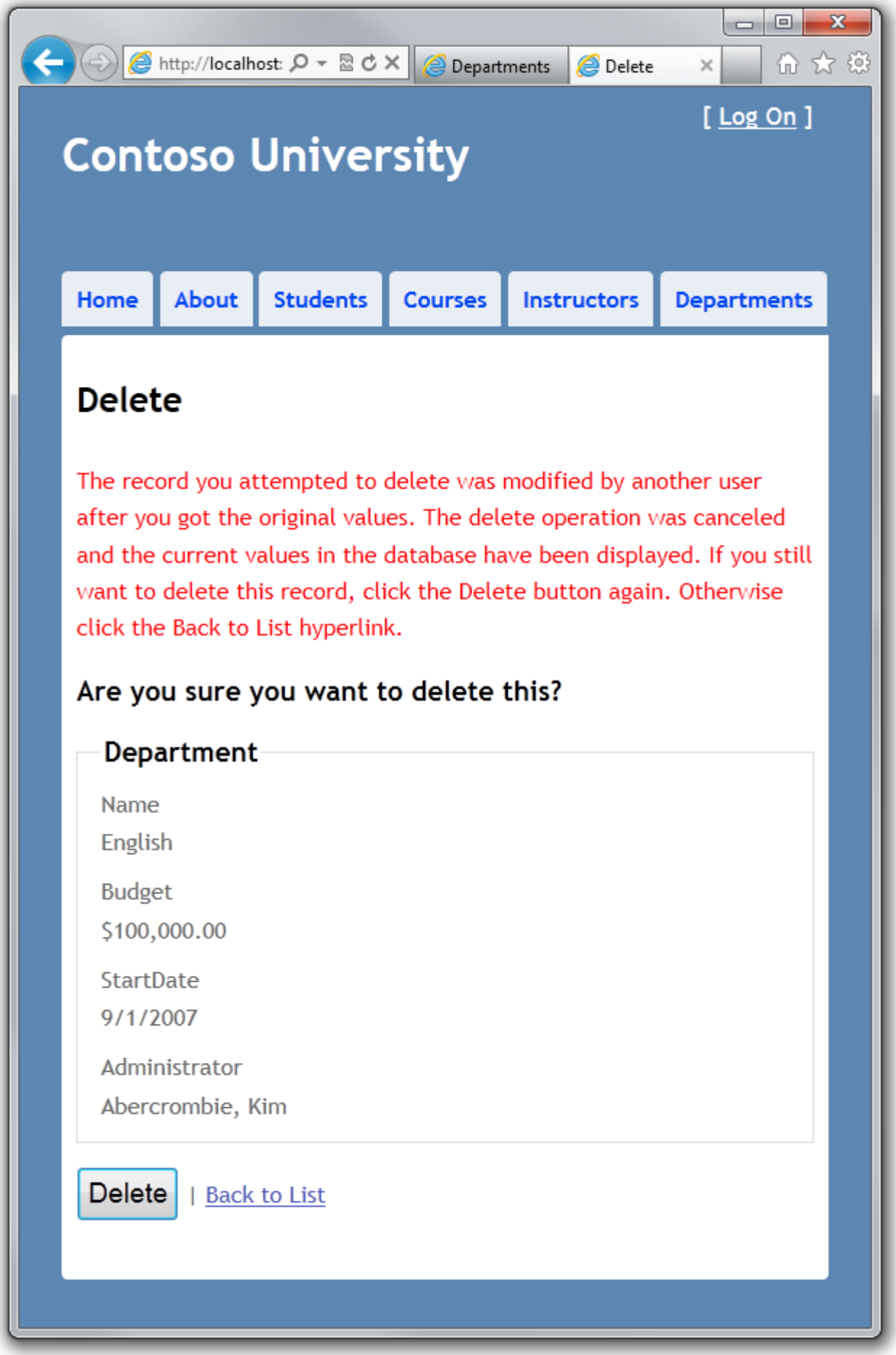
In the second window, select **Delete** on the same department. The Delete confirmation page appears.



Click **Save** in the first browser window. The Index page confirms the change.



Now click **Delete** in the second browser. You see the concurrency error message, and the Department values are refreshed with what's currently in the database.


http://localhost:
Departments
Delete
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Delete
The record you attempted to delete was modified by another user after you got the original values. The delete operation was canceled and the current values in the database have been displayed. If you still want to delete this record, click the Delete button again. Otherwise click the Back to List hyperlink.
Are you sure you want to delete this?
Department
Name
English
Budget
$100,000.00
StartDate
9/1/2007
Administrator
Abercrombie, Kim
Delete
| Back to List

If you click **Delete** again, you're redirected to the Index page, which shows that the department has been deleted.

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 the Entity Framework team blog. The next tutorial shows how to implement table-per-hierarchy inheritance for the `Instructor` and `Student` entities.

Links to other Entity Framework resources can be found at the end of the last tutorial in this series.

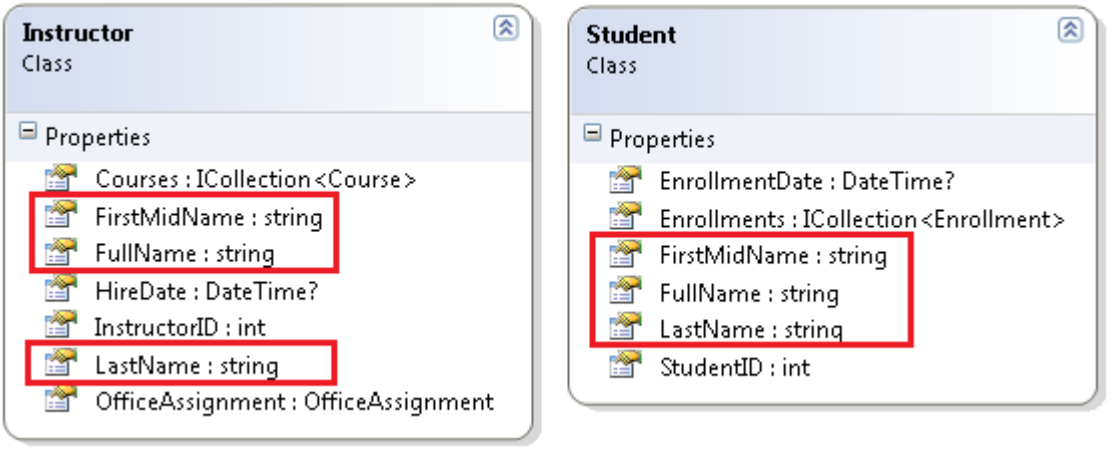
Implementing Inheritance

In the previous tutorial you handled concurrency exceptions. This tutorial will show you how to implement inheritance in the data model.

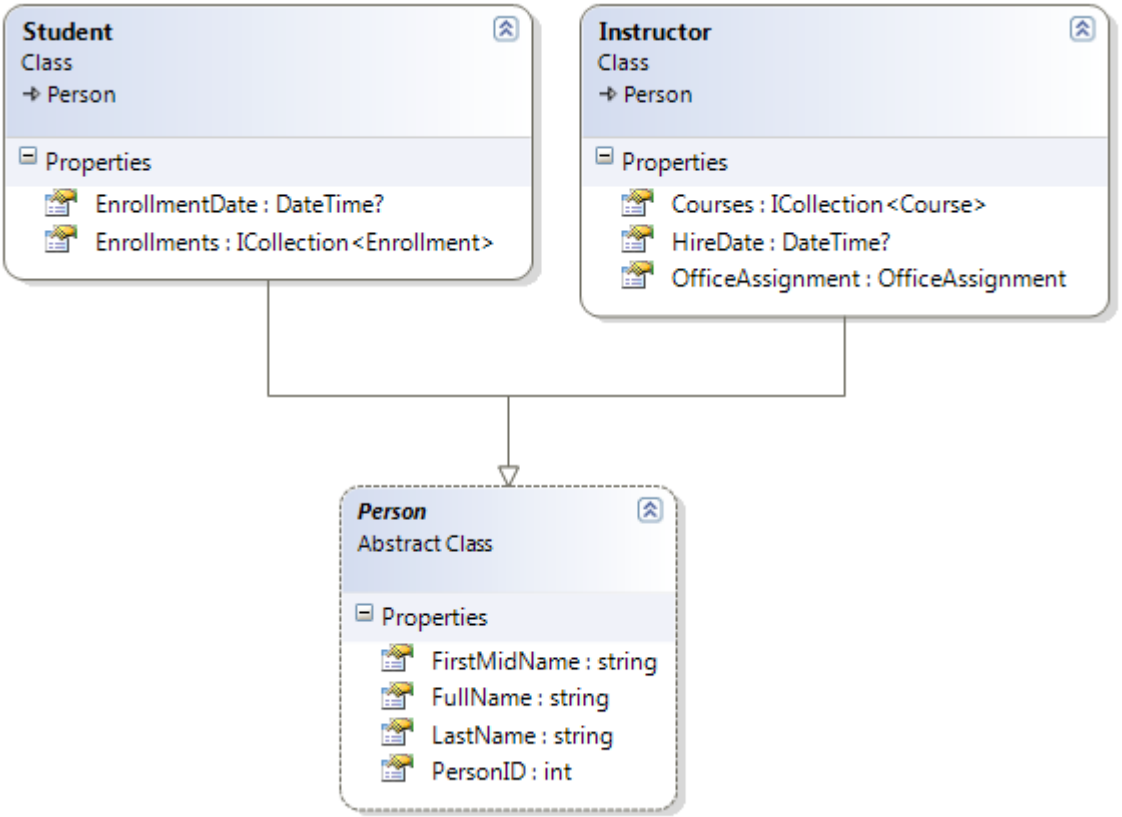
In object-oriented programming, you can use inheritance to eliminate redundant code. In this tutorial, you'll change the `Instructor` and `Student` classes so that they derive from a `Person` base class which contains properties such as `LastName` that are common to both instructors and students. You 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.

Table-per-Hierarchy versus Table-per-Type Inheritance

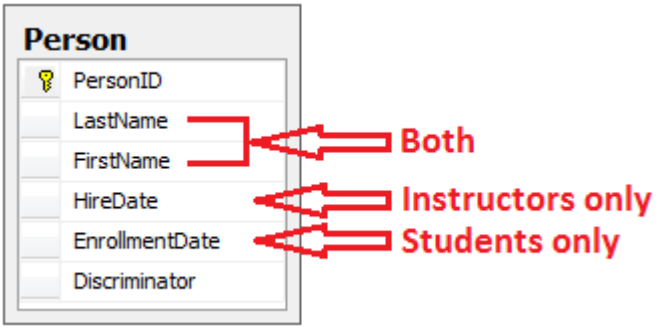
In object-oriented programming, you can use inheritance to make it easier to work with related classes. For example, the `Instructor` and `Student` classes in the `School` data model share several properties, which results in redundant code:



Suppose you want to eliminate the redundant code for the properties that are shared by the `Instructor` and `Student` entities. 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:

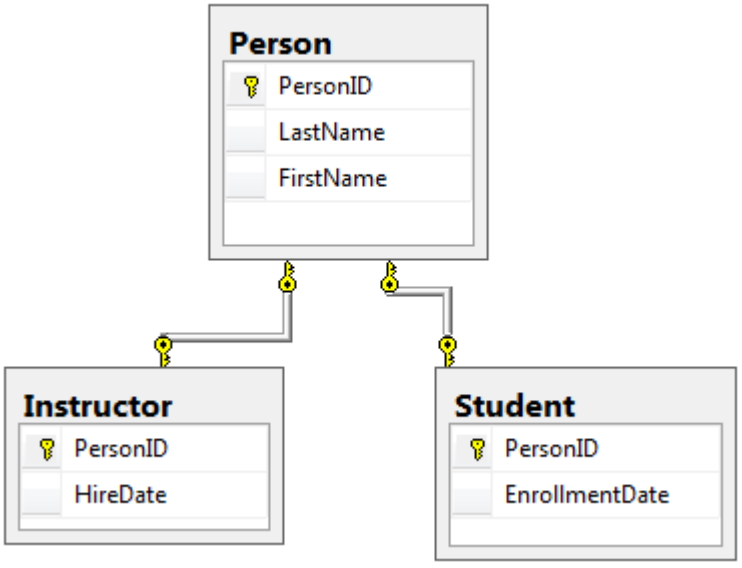


There are several ways this inheritance structure could be represented in the database. You could have a `Person` table that includes information about both students and instructors in a single table. Some of the columns could apply only to instructors (`HireDate`), some only to students (`EnrollmentDate`), some to both (`LastName`, `FirstName`). Typically you'd have a *discriminator* column to indicate which type each row represents. (In this case , 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.

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. You'll do that by performing the following steps:

- Create a `Person` class and change the `Instructor` and `Student` classes to derive from `Person`.
- Add model-to-database mapping code to the database context class.
- Change `InstructorID` and `StudentID` references throughout the project to `PersonID`.

Creating the Person Class

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

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.ComponentModel.DataAnnotations;

namespaceContosoUniversity.Models
{
```

```
publicabstractclassPerson
{
[Key]
publicintPersonID{get;set;}

[Required(ErrorMessage="Last name is required.")]
[Display(Name="Last Name")]
[MaxLength(50)]
publicstringLastName{get;set;}

[Required(ErrorMessage="First name is required.")]
[Column("FirstName")]
[Display(Name="First Name")]
[MaxLength(50)]
publicstringFirstMidName{get;set;}

publicstringFullName
{
get
{
returnLastName+", "+FirstMidName;
}
}
}
}
```

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:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.ComponentModel.DataAnnotations;

namespaceContosoUniversity.Models
{
publicclassInstructor:Person
{
```

```
[DisplayFormat(DataFormatString="{0:d}",ApplyFormatInEditMode=true)]
[Required(ErrorMessage="Hire date is required.")]
[Display(Name="Hire Date")]
publicDateTime?HireDate{get;set;}

publicvirtualICollection<Course>Courses{get;set;}

publicvirtualOfficeAssignmentOfficeAssignment{get;set;}
}
}
```

Make similar changes to *Student.cs*. The `Student` class will look like the following example:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.ComponentModel.DataAnnotations;

namespaceContosoUniversity.Models
{
publicclassStudent:Person
{
[Required(ErrorMessage="Enrollment date is required.")]
[DisplayFormat(DataFormatString="{0:d}",ApplyFormatInEditMode=true)]
[Display(Name="Enrollment Date")]
publicDateTime?EnrollmentDate{get;set;}

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

Adding the Person Entity Type to the Model

In *SchoolContext.cs*, add a `DbSet` property for the `Person` entity type:

```
publicDbSet<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 re-created, it will have a `Person` table in place of the `Student` and `Instructor` tables.

Changing InstructorID and StudentID to PersonID

In *SchoolContext.cs*, in the Instructor-Course mapping statement, change `MapRightKey("InstructorID")` to `MapRightKey("PersonID")`:

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

This change isn't required; it just changes the name of the InstructorID column in the many-to-many join table. If you left the name as InstructorID, the application would still work correctly.

Next, perform a global change (all files in the project) to change `InstructorID` to `PersonID` and `StudentID` to `PersonID`. Make sure that this change is case-sensitive. (Note that this demonstrates a disadvantage of the *`classname`*`ID` pattern for naming primary keys. If you had named primary keys ID without prefixing the class name, no renaming would be necessary now.)

Adjusting Primary Key Values in the Initializer

In *SchoolInitializer.cs*, the code currently assumes that primary key values for `Student` and `Instructor` entities are numbered separately. This is still correct for `Student` entities (they'll still be 1 through 8), but `Instructor` entities will now be 9 through 13 instead of 1 through 5, because the block of code that adds instructors comes after the one that adds students in the initializer class. Replace the code that seeds the `Department` and `OfficeAssignment` entity sets with the following code that uses the new ID values for instructors:

```
var departments =newList<Department>
{
newDepartment{Name="English",Budget=350000,StartDate=DateTime.Parse("2007-09-
01"),PersonID=9},
newDepartment{Name="Mathematics",Budget=100000,StartDate=DateTime.Parse("2007-09-
01"),PersonID=10},
newDepartment{Name="Engineering",Budget=350000,StartDate=DateTime.Parse("2007-09-
```

```
01"),PersonID=11},
newDepartment{Name="Economics",Budget=100000,StartDate=DateTime.Parse("2007-09-
01"),PersonID=12}
};

var officeAssignments =newList<OfficeAssignment>
{
newOfficeAssignment{PersonID=9,Location="Smith 17"},
newOfficeAssignment{PersonID=10,Location="Gowan 27"},
newOfficeAssignment{PersonID=11,Location="Thompson 304"},
};
```

Changing OfficeAssignment to Lazy Loading

The current version of the Entity Framework doesn't support eager loading for one-to-zero-or-one relationships when the navigation property is on the derived class of a TPH inheritance structure. This is the case with the `OfficeAssignment` property on the `Instructor` entity. To work around this, you'll remove the code you added earlier to perform eager loading on this property.

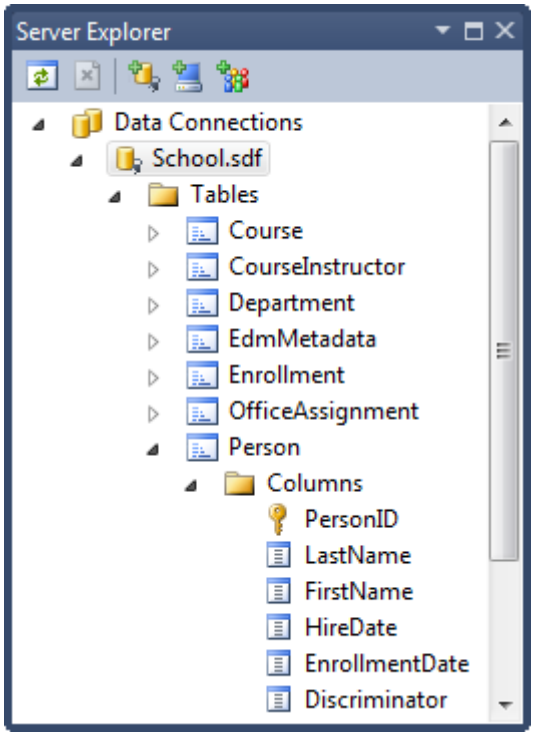
In *InstructorController.cs*, delete the three occurrences of the following line of code:

```
.Include(i => i.OfficeAssignment)
```

Testing

Run the site and try various pages. Everything works the same as it did before.

In **Solution Explorer**, double-click *School.sdf* to open the database in **Server Explorer**. Expand **School.sdf** 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, plus the discriminator column.



The following diagram illustrates the structure of the new School database:

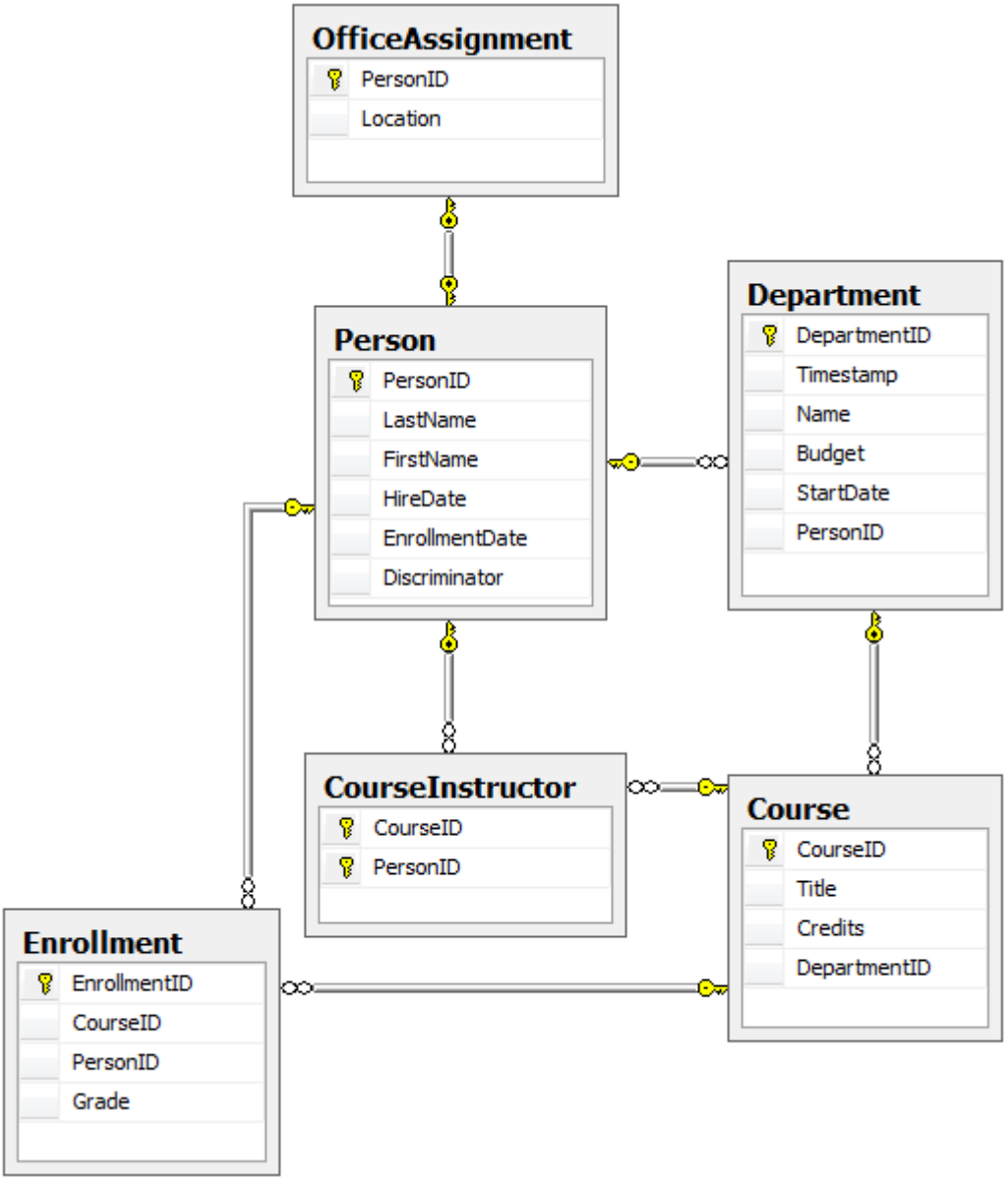


Table-per-hierarchy inheritance has now been implemented for the `Person`, `Student`, and `Instructor` classes. For more information about this and other inheritance structures, see Inheritance Mapping Strategies on Morteza Manavi's blog. In the next tutorial you'll see some ways to implement the repository and unit of work patterns.

Links to other Entity Framework resources can be found at the end of the last tutorial in this series.

Implementing the Repository and Unit of Work Patterns

In the previous tutorial you used inheritance to reduce redundant code in the `Student` and `Instructor` entity classes. In this tutorial you'll see some ways to use the repository and unit of work patterns for CRUD operations. As in the previous tutorial, in this one you'll change the way your code works with pages you already created rather than creating new pages.

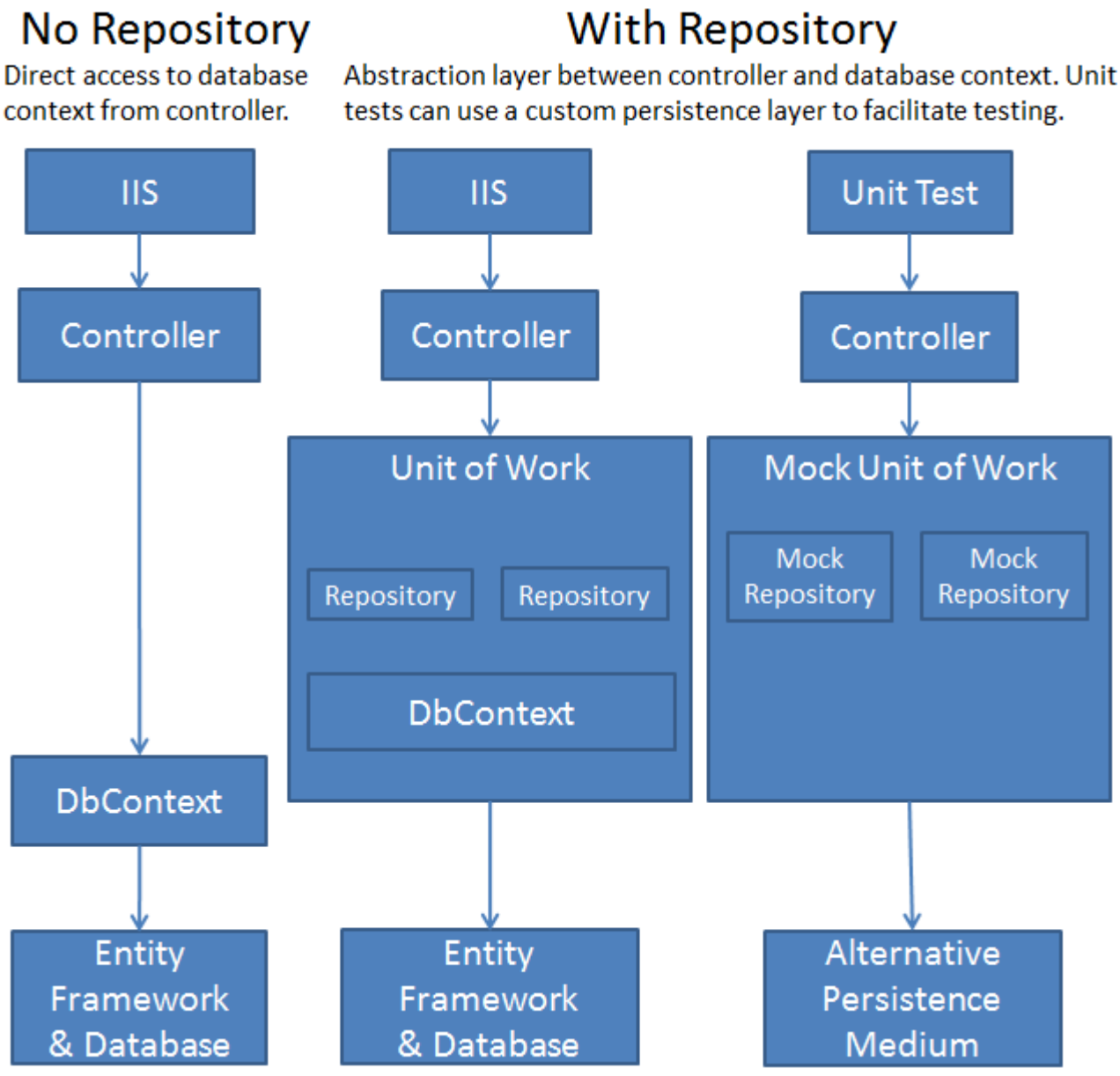
The Repository and Unit of Work Patterns

The repository and unit of work 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).

In this tutorial you'll implement a repository class for each entity type. For the `Student` entity type you'll create a repository interface and a repository class. When you instantiate the repository in your controller, you'll use the interface so that the controller will accept a reference to any object that implements the repository interface. When the controller runs under a web server, it receives a repository that works with the Entity Framework. When the controller runs under a unit test class, it receives a repository that works with data stored in a way that you can easily manipulate for testing, such as an in-memory collection.

Later in the tutorial you'll use multiple repositories and a unit of work class for the `Course` and `Department` entity types in the `Course` controller. The unit of work class coordinates the work of multiple repositories by creating a single database context class shared by all of them. If you wanted to be able to perform automated unit testing, you'd create and use interfaces for these classes in the same way you did for the `Student` repository. However, to keep the tutorial simple, you'll create and use these classes without interfaces.

The following illustration shows one way to conceptualize the relationships between the controller and context classes compared to not using the repository or unit of work pattern at all.



You won't create unit tests in this tutorial series. For an introduction to TDD with an MVC application that uses the repository pattern, see Walkthrough: Using TDD with ASP.NET MVC on the MSDN Library web site. For more information about the repository pattern, see Using Repository and Unit of Work patterns with Entity Framework 4.0 on the Entity Framework team blog and the Agile Entity Framework 4 Repository series of posts on Julie Lerman's blog.

Note There are many ways to implement the repository and unit of work patterns. You can use repository classes with or without a unit of work class. You can implement a single repository for all entity types, or one for each type. If you implement one for each type, you can use separate classes, a generic base class and derived classes, or an abstract base class and derived classes. You can include business logic in your repository or

restrict it to data access logic. You can also build an abstraction layer into your database context class by using `IDbSet` interfaces there instead of `DbSet` types for your entity sets. The approach to implementing an abstraction layer shown in this tutorial is one option for you to consider, not a recommendation for all scenarios and environments.

Creating the Student Repository Class

In the *DAL* folder, create a class file named *IStudentRepository.cs* and replace the existing code with the following code:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.Linq;
usingSystem.Web;
usingContosoUniversity.Models;

namespaceContosoUniversity.DAL
{
publicinterfaceIStudentRepository:IDisposable
{
IEnumerable<Student>GetStudents();
StudentGetStudentByID(int studentId);
voidInsertStudent(Student student);
voidDeleteStudent(int studentID);
voidUpdateStudent(Student student);
voidSave();
}
}
```

This code declares a typical set of CRUD methods, including two read methods — one that returns all `Student` entities, and one that finds a single `Student` entity by ID.

In the *DAL* folder, create a class file named *StudentRepository.cs* file. Replace the existing code with the following code, which implements the `IStudentRepository` interface:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.Linq;
```

```
usingSystem.Data;
usingContosoUniversity.Models;

namespaceContosoUniversity.DAL
{
publicclassStudentRepository:IStudentRepository,IDisposable
{
privateSchoolContext context;

publicStudentRepository(SchoolContext context)
{
this.context = context;
}

publicIEnumerable<Student>GetStudents()
{
return context.Students.ToList();
}

publicStudentGetStudentByID(int id)
{
return context.Students.Find(id);
}

publicvoidInsertStudent(Student student)
{
            context.Students.Add(student);
}

publicvoidDeleteStudent(int studentID)
{
Student student = context.Students.Find(studentID);
            context.Students.Remove(student);
}

publicvoidUpdateStudent(Student student)
{
            context.Entry(student).State=EntityState.Modified;
```

```
}

publicvoidSave()
{
            context.SaveChanges();
}

privatebool disposed =false;

protectedvirtualvoidDispose(bool disposing)
{
if(!this.disposed)
{
if(disposing)
{
                    context.Dispose();
}
}
this.disposed =true;
}

publicvoidDispose()
{
Dispose(true);
            GC.SuppressFinalize(this);
}
}
}
```

The database context is defined in a class variable, and the constructor expects the calling object to pass in an instance of the context:

```
privateSchoolContext context;

publicStudentRepository(SchoolContext context)
{
```

```
this.context = context;
}
```

You could instantiate a new context in the repository, but then if you used multiple repositories in one controller, each would end up with a separate context. Later you'll use multiple repositories in the `Course` controller, and you'll see how a unit of work class can ensure that all repositories use the same context.

The repository implements `IDisposable` and disposes the database context as you saw earlier in the controller, and its CRUD methods make calls to the database context in the same way that you saw earlier.

Changing the Student Controller to Use the Repository

In *StudentController.cs*, replace the code currently in the class with the following code:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.Data;
usingSystem.Data.Entity;
usingSystem.Linq;
usingSystem.Web;
usingSystem.Web.Mvc;
usingContosoUniversity.Models;
usingContosoUniversity.DAL;
usingPagedList;

namespaceContosoUniversity.Controllers
{
publicclassStudentController:Controller
{
privateIStudentRepository studentRepository;


publicStudentController()
{
this.studentRepository =newStudentRepository(newSchoolContext());
}
```

```
publicStudentController(IStudentRepository studentRepository)
{
this.studentRepository = studentRepository;
}


//
// GET: /Student/

publicViewResultIndex(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(Request.HttpMethod=="GET")
{
                searchString = currentFilter;
}
else
{
                page =1;
}
ViewBag.CurrentFilter= searchString;

var students =from s in studentRepository.GetStudents()
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;
}

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


//
// GET: /Student/Details/5

publicViewResultDetails(int id)
{
Student student = studentRepository.GetStudentByID(id);
returnView(student);
}

//
// GET: /Student/Create

publicActionResultCreate()
{
returnView();
}

//
// POST: /Student/Create
```

```
[HttpPost]
publicActionResultCreate(Student student)
{
try
{
if(ModelState.IsValid)
{
                    studentRepository.InsertStudent(student);
                    studentRepository.Save();
returnRedirectToAction("Index");
}
}
catch(DataException)
{
//Log the error (add a variable name after DataException)
ModelState.AddModelError("","Unable to save changes. Try again, and if the problem
persists see your system administrator.");
}
returnView(student);
}

//
// GET: /Student/Edit/5

publicActionResultEdit(int id)
{
Student student = studentRepository.GetStudentByID(id);
returnView(student);
}

//
// POST: /Student/Edit/5

[HttpPost]
publicActionResultEdit(Student student)
{
try
{
```

```
if(ModelState.IsValid)
{
                    studentRepository.UpdateStudent(student);
                    studentRepository.Save();
returnRedirectToAction("Index");
}
}
catch(DataException)
{
//Log the error (add a variable name after DataException)
ModelState.AddModelError("","Unable to save changes. Try again, and if the problem
persists see your system administrator.");
}
returnView(student);
}

//
// GET: /Student/Delete/5

publicActionResultDelete(int id,bool? saveChangesError)
{
if(saveChangesError.GetValueOrDefault())
{
ViewBag.ErrorMessage="Unable to save changes. Try again, and if the problem persists
see your system administrator.";
}
Student student = studentRepository.GetStudentByID(id);
returnView(student);
}


//
// POST: /Student/Delete/5

[HttpPost,ActionName("Delete")]
publicActionResultDeleteConfirmed(int id)
{
try
```

```
{
Student student = studentRepository.GetStudentByID(id);
                studentRepository.DeleteStudent(id);
                studentRepository.Save();
}
catch(DataException)
{
//Log the error (add a variable name after DataException)
returnRedirectToAction("Delete",
newSystem.Web.Routing.RouteValueDictionary{
{"id", id },
{"saveChangesError",true}});
}
returnRedirectToAction("Index");
}

protectedoverridevoidDispose(bool disposing)
{
            studentRepository.Dispose();
base.Dispose(disposing);
}
}
}
```

The controller now declares a class variable for an object that implements the `IStudentRepository` interface instead of the context class:

```
privateIStudentRepository studentRepository;
```

The default constructor creates a new context instance, and an optional constructor allows the caller to pass in a context instance.

```
publicStudentController()
{
this.studentRepository =newStudentRepository(newSchoolContext());
}
```

```
publicStudentController(IStudentRepository studentRepository)
{
this.studentRepository = studentRepository;
}
```

(If you were using *dependency injection*, or DI, you wouldn't need the default constructor because the DI software would ensure that the correct repository object would always be provided.)

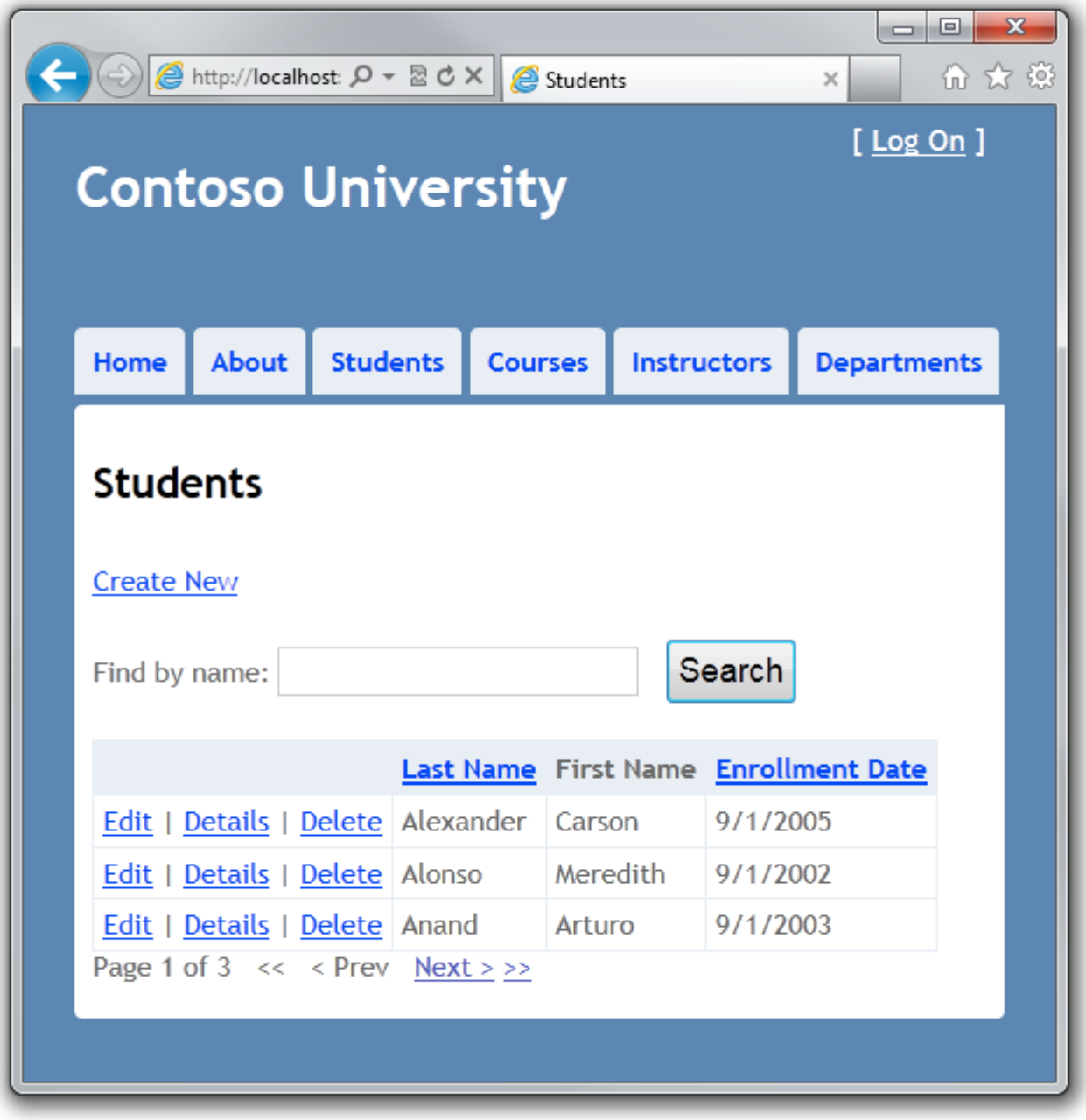
In the CRUD methods, the repository is now called instead of the context:

```
var students =from s in studentRepository.GetStudents()
select s;

Student student = studentRepository.GetStudentByID(id);

studentRepository.InsertStudent(student);
studentRepository.Save();

studentRepository.UpdateStudent(student);
studentRepository.Save();

studentRepository.DeleteStudent(id);
studentRepository.Save();
```

And the `Dispose` method now disposes the repository instead of the context:

```
studentRepository.Dispose();
```

Run the site and click the **Students** tab.



The page looks and works the same as it did before you changed the code to use the repository, and the other Student pages also work the same. However, there's an important difference in the way the `Index` method of the controller does filtering and ordering. The original version of this method contained the following code:

```
var students =from s in context.Students
select s;
if(!String.IsNullOrEmpty(searchString))
{
```

```
    students = students.Where(s =>
s.LastName.ToUpper().Contains(searchString.ToUpper())
|| s.FirstMidName.ToUpper().Contains(searchString.ToUpper()));
}
```

In the original version of the code, `students` is typed as an `IQueryable` object. The query isn't sent to the database until it's converted into a collection using a method such as `ToList`, which means that this `Where` method becomes a `WHERE` clause in the SQL query and is processed by the database. That in turn means that only the selected entities are returned by the database. However, as a result of changing `context.Students` to `studentRepository.GetStudents()`, the `students` variable after this statement is an `IEnumerable` collection that includes all students in the database. The end result of applying the `Where` method is the same, but now the work is done in memory on the web server and not by the database. For large volumes of data, this is likely to be inefficient. The following section shows how to implement repository methods that enable you to specify that this work should be done by the database.

You've now created an abstraction layer between the controller and the Entity Framework database context. If you were going to perform automated unit testing with this application, you could create an alternative repository class in a unit test project that implements `IStudentRepository`. Instead of calling the context to read and write data, this mock repository class could manipulate in-memory collections in order to test controller functions.

Implementing a Generic Repository and a Unit of Work Class

Creating a repository class for each entity type could result in a lot of redundant code, and it could result in partial updates. For example, suppose you have to update two different entity types as part of the same transaction. If each uses a separate database context instance, one might succeed and the other might fail. One way to minimize redundant code is to use a generic repository, and one way to ensure that all repositories use the same database context (and thus coordinate all updates) is to use a unit of work class.

In this section of the tutorial, you'll create a `GenericRepository` class and a `UnitOfWork` class, and use them in the `Course` controller to access both the `Department` and the `Course` entity sets. As explained earlier, to keep this part of the tutorial simple, you aren't creating interfaces for these classes. But if you were going to use them to facilitate TDD, you'd typically implement them with interfaces the same way you did the `Student` repository.

Creating a Generic Repository

In the *DAL* folder, create *GenericRepository.cs* and replace the existing code with the following code:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.Linq;
usingSystem.Data;
usingSystem.Data.Entity;
usingContosoUniversity.Models;
usingSystem.Linq.Expressions;

namespaceContosoUniversity.DAL
{
publicclassGenericRepository<TEntity>whereTEntity:class
{
internalSchoolContext context;
internalDbSet<TEntity> dbSet;

publicGenericRepository(SchoolContext context)
{
this.context = context;
this.dbSet = context.Set<TEntity>();
}

publicvirtualIEnumerable<TEntity>Get(
Expression<Func<TEntity,bool>> filter =null,
Func<IQueryable<TEntity>,IOrderedQueryable<TEntity>> orderBy =null,
string includeProperties ="")
{
IQueryable<TEntity> query = dbSet;

if(filter !=null)
{
                query = query.Where(filter);
}

foreach(var includeProperty in includeProperties.Split
(newchar[]{','},StringSplitOptions.RemoveEmptyEntries))
{
                query = query.Include(includeProperty);
}
```

```
if(orderBy !=null)
{
return orderBy(query).ToList();
}
else
{
return query.ToList();
}
}

publicvirtualTEntityGetByID(object id)
{
return dbSet.Find(id);
}

publicvirtualvoidInsert(TEntity entity)
{
            dbSet.Add(entity);
}

publicvirtualvoidDelete(object id)
{
TEntity entityToDelete = dbSet.Find(id);
Delete(entityToDelete);
}

publicvirtualvoidDelete(TEntity entityToDelete)
{
if(context.Entry(entityToDelete).State==EntityState.Detached)
{
                dbSet.Attach(entityToDelete);
}
            dbSet.Remove(entityToDelete);
}

publicvirtualvoidUpdate(TEntity entityToUpdate)
{
```

```
            dbSet.Attach(entityToUpdate);
            context.Entry(entityToUpdate).State=EntityState.Modified;
}
}
}
```

Class variables are declared for the database context and for the entity set that the repository is instantiated for:

```
internalSchoolContext context;
internalDbSet dbSet;
```

The constructor accepts a database context instance and initializes the entity set variable:

```
publicGenericRepository(SchoolContext context)
{
this.context = context;
this.dbSet = context.Set();
}
```

The `Get` method uses lambda expressions to allow the calling code to specify a filter condition and a column to order the results by, and a string parameter lets the caller provide a comma-delimited list of navigation properties for eager loading:

```
publicvirtualIEnumerable<TEntity>Get(
Expression<Func<TEntity,bool>> filter =null,
Func<IQueryable<TEntity>,IOrderedQueryable<TEntity>> orderBy =null,
string includeProperties ="")
```

The code `Expression<Func<TEntity, bool>> filter` means the caller will provide a lambda expression based on the `TEntity` type, and this expression will return a Boolean value. For example, if the repository is instantiated for the `Student` entity type, the code in the calling method might specify `student => student.LastName == "Smith"` for the `filter` parameter.

The code `Func<IQueryable<TEntity>, IOrderedQueryable<TEntity>> orderBy` also means the caller will provide a lambda expression. But in this case, the input to the expression is an `IQueryable` object for the

`TEntity` type. The expression will return an ordered version of that `IQueryable` object. For example, if the repository is instantiated for the `Student` entity type, the code in the calling method might specify `q => q.OrderBy(s => s.LastName)` for the `orderBy` parameter.

The code in the `Get` method creates an `IQueryable` object and then applies the filter expression if there is one:

```
IQueryable<TEntity> query = dbSet;

if(filter !=null)
{
    query = query.Where(filter);
}
```

Next it applies the eager-loading expressions after parsing the comma-delimited list:

```
foreach(var includeProperty in includeProperties.Split
(newchar[]{','},StringSplitOptions.RemoveEmptyEntries))
{
    query = query.Include(includeProperty);
}
```

Finally, it applies the `orderBy` expression if there is one and returns the results; otherwise it returns the results from the unordered query:

```
if(orderBy !=null)
{
return orderBy(query).ToList();
}
else
{
return query.ToList();
}
```

When you call the `Get` method, you could do filtering and sorting on the `IEnumerable` collection returned by the method instead of providing parameters for these functions. But the sorting and filtering work would then

be done in memory on the web server. By using these parameters, you ensure that the work is done by the database rather than the web server. An alternative is to create derived classes for specific entity types and add specialized `Get` methods, such as `GetStudentsInNameOrder` or `GetStudentsByName`. However, in a complex application, this can result in a large number of such derived classes and specialized methods, which could be more work to maintain.

The code in the `GetByID`, `Insert`, and `Update` methods is similar to what you saw in the non-generic repository. (You aren't providing an eager loading parameter in the `GetByID` signature, because you can't do eager loading with the `Find` method.)

Two overloads are provided for the `Delete` method:

```
publicvirtualvoidDelete(object id)
{
TEntity entityToDelete = dbSet.Find(id);
    dbSet.Remove(entityToDelete);
}

publicvirtualvoidDelete(TEntity entityToDelete)
{
    context.Entry(entityToDelete).State=EntityState.Deleted;
}
```

One of these lets you pass in just the ID of the entity to be deleted, and one takes an entity instance. As you saw in the Handling Concurrency tutorial, for concurrency handling you need a `Delete` method that takes an entity instance that includes the original value of a tracking property.

This generic repository will handle typical CRUD requirements. When a particular entity type has special requirements, such as more complex filtering or ordering, you can create a derived class that has additional methods for that type.

Creating the Unit of Work Class

The unit of work class serves one purpose: to make sure that when you use multiple repositories, they share a single database context. That way, when a unit of work is complete you can call the `SaveChanges` method on that instance of the context and be assured that all related changes will be coordinated. All that the class needs is a `Save` method and a property for each repository. Each repository property returns a repository instance that has been instantiated using the same database context instance as the other repository instances.

In the *DAL* folder, create a class file named *UnitOfWork.cs* and replace the existing code with the following code:

```
usingSystem;
usingContosoUniversity.Models;

namespaceContosoUniversity.DAL
{
publicclassUnitOfWork:IDisposable
{
privateSchoolContext context =newSchoolContext();
privateGenericRepository<Department> departmentRepository;
privateGenericRepository<Course> courseRepository;

publicGenericRepository<Department>DepartmentRepository
{
get
{

if(this.departmentRepository ==null)
{
this.departmentRepository =newGenericRepository<Department>(context);
}
return departmentRepository;
}
}

publicGenericRepository<Course>CourseRepository
{
get
{

if(this.courseRepository ==null)
{
this.courseRepository =newGenericRepository<Course>(context);
}
return courseRepository;
}
```

```
}

publicvoidSave()
{
            context.SaveChanges();
}

privatebool disposed =false;

protectedvirtualvoidDispose(bool disposing)
{
if(!this.disposed)
{
if(disposing)
{
                    context.Dispose();
}
}
this.disposed =true;
}

publicvoidDispose()
{
Dispose(true);
            GC.SuppressFinalize(this);
}
}
}
```

The code creates class variables for the database context and each repository. For the `context` variable, a new context is instantiated:

```
privateSchoolContext context =newSchoolContext();
privateGenericRepository<Department> departmentRepository;
privateGenericRepository<Course> courseRepository;
```

Each repository property checks whether the repository already exists. If not, it instantiates the repository, passing in the context instance. As a result, all repositories share the same context instance.

```
publicGenericRepository<Department>DepartmentRepository
{
get
{

if(this.departmentRepository ==null)
{
this.departmentRepository =newGenericRepository<Department>(context);
}
return departmentRepository;
}
}
```

The `Save` method calls `SaveChanges` on the database context.

Like any class that instantiates a database context in a class variable, the `UnitOfWork` class implements `IDisposable` and disposes the context.

Changing the Course Controller to use the UnitOfWork Class and Repositories

Replace the code you currently have in *CourseController.cs* with the following code:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.Data;
usingSystem.Data.Entity;
usingSystem.Linq;
usingSystem.Web;
usingSystem.Web.Mvc;
usingContosoUniversity.Models;
usingContosoUniversity.DAL;

namespaceContosoUniversity.Controllers
{
```

```
publicclassCourseController:Controller
{
privateUnitOfWork unitOfWork =newUnitOfWork();

//
// GET: /Course/

publicViewResultIndex()
{
var courses = unitOfWork.CourseRepository.Get(includeProperties:"Department");
returnView(courses.ToList());
}

//
// GET: /Course/Details/5

publicViewResultDetails(int id)
{
Course course = unitOfWork.CourseRepository.GetByID(id);
returnView(course);
}

//
// GET: /Course/Create

publicActionResultCreate()
{
PopulateDepartmentsDropDownList();
returnView();
}

[HttpPost]
publicActionResultCreate(Course course)
{
try
{
if(ModelState.IsValid)
{
```

```
                    unitOfWork.CourseRepository.Insert(course);
                    unitOfWork.Save();
returnRedirectToAction("Index");
}
}
catch(DataException)
{
//Log the error (add a variable name after DataException)
ModelState.AddModelError("","Unable to save changes. Try again, and if the problem
persists, see your system administrator.");
}
PopulateDepartmentsDropDownList(course.DepartmentID);
returnView(course);
}

publicActionResultEdit(int id)
{
Course course = unitOfWork.CourseRepository.GetByID(id);
PopulateDepartmentsDropDownList(course.DepartmentID);
returnView(course);
}

[HttpPost]
publicActionResultEdit(Course course)
{
try
{
if(ModelState.IsValid)
{
                    unitOfWork.CourseRepository.Update(course);
                    unitOfWork.Save();
returnRedirectToAction("Index");
}
}
catch(DataException)
{
//Log the error (add a variable name after DataException)
ModelState.AddModelError("","Unable to save changes. Try again, and if the problem
```

```
persists, see your system administrator.");
}
PopulateDepartmentsDropDownList(course.DepartmentID);
returnView(course);
}

privatevoidPopulateDepartmentsDropDownList(object selectedDepartment =null)
{
var departmentsQuery = unitOfWork.DepartmentRepository.Get(
                orderBy: q => q.OrderBy(d => d.Name));
ViewBag.DepartmentID=newSelectList(departmentsQuery,"DepartmentID","Name",
selectedDepartment);
}

//
// GET: /Course/Delete/5

publicActionResultDelete(int id)
{
Course course = unitOfWork.CourseRepository.GetByID(id);
returnView(course);
}

//
// POST: /Course/Delete/5

[HttpPost,ActionName("Delete")]
publicActionResultDeleteConfirmed(int id)
{
Course course = unitOfWork.CourseRepository.GetByID(id);
            unitOfWork.CourseRepository.Delete(id);
            unitOfWork.Save();
returnRedirectToAction("Index");
}

protectedoverridevoidDispose(bool disposing)
{
            unitOfWork.Dispose();
```

```
base.Dispose(disposing);
}
}
}
```

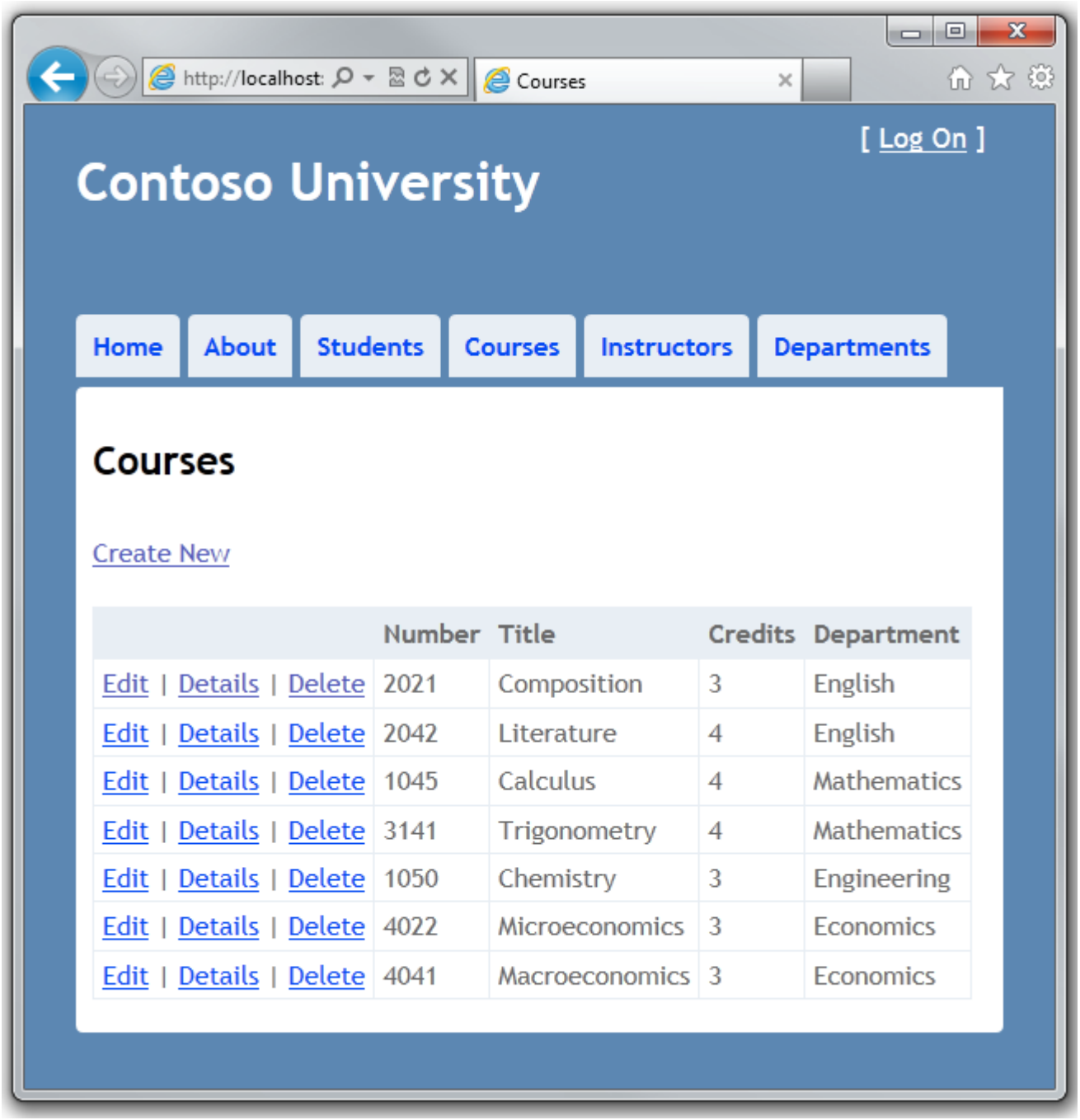
This code adds a class variable for the `UnitOfWork` class. (If you were using interfaces here, you wouldn't initialize the variable here; instead, you'd implement a pattern of two constructors just as you did for the `Student` repository.)

```
privateUnitOfWork unitOfWork =newUnitOfWork();
```

In the rest of the class, all references to the database context are replaced by references to the appropriate repository, using `UnitOfWork` properties to access the repository. The `Dispose` method disposes the `UnitOfWork` instance.

```
var courses = unitOfWork.CourseRepository.Get(includeProperties:"Department");
// ...
Course course = unitOfWork.CourseRepository.GetByID(id);
// ...
unitOfWork.CourseRepository.Insert(course);
unitOfWork.Save();
// ...
Course course = unitOfWork.CourseRepository.GetByID(id);
// ...
unitOfWork.CourseRepository.Update(course);
unitOfWork.Save();
// ...
var departmentsQuery = unitOfWork.DepartmentRepository.Get(
    orderBy: q => q.OrderBy(d => d.Name));
// ...
Course course = unitOfWork.CourseRepository.GetByID(id);
// ...
unitOfWork.CourseRepository.Delete(id);
unitOfWork.Save();
// ...
unitOfWork.Dispose();
```

Run the site and click the **Courses** tab.



The page looks and works the same as it did before your changes, and the other Course pages also work the same.

You have now implemented both the repository and unit of work patterns. You have used lambda expressions as method parameters in the generic repository. For more information about how to use these expressions with

an `IQueryable` object, see IQueryable(T) Interface (System.Linq) in the MSDN Library. In the next tutorial you'll learn how to handle some advanced scenarios.

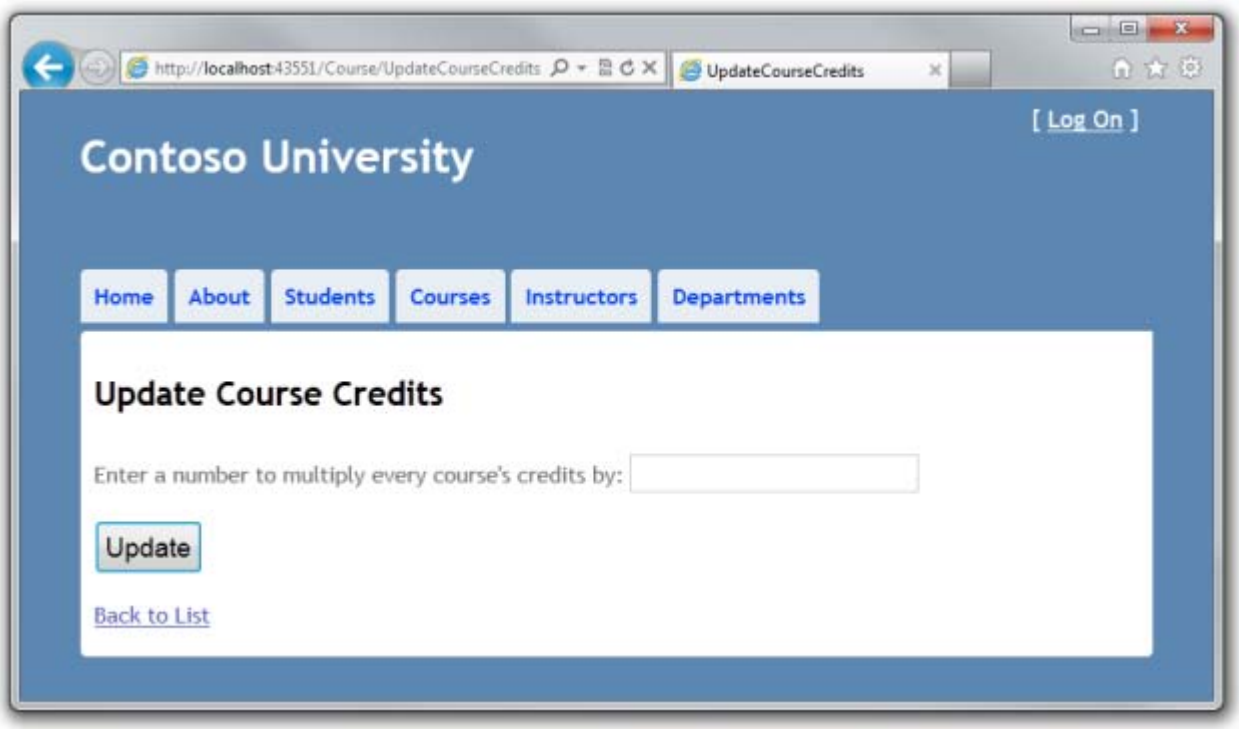
Links to other Entity Framework resources can be found at the end of the last tutorial in this series.

Advanced Scenarios

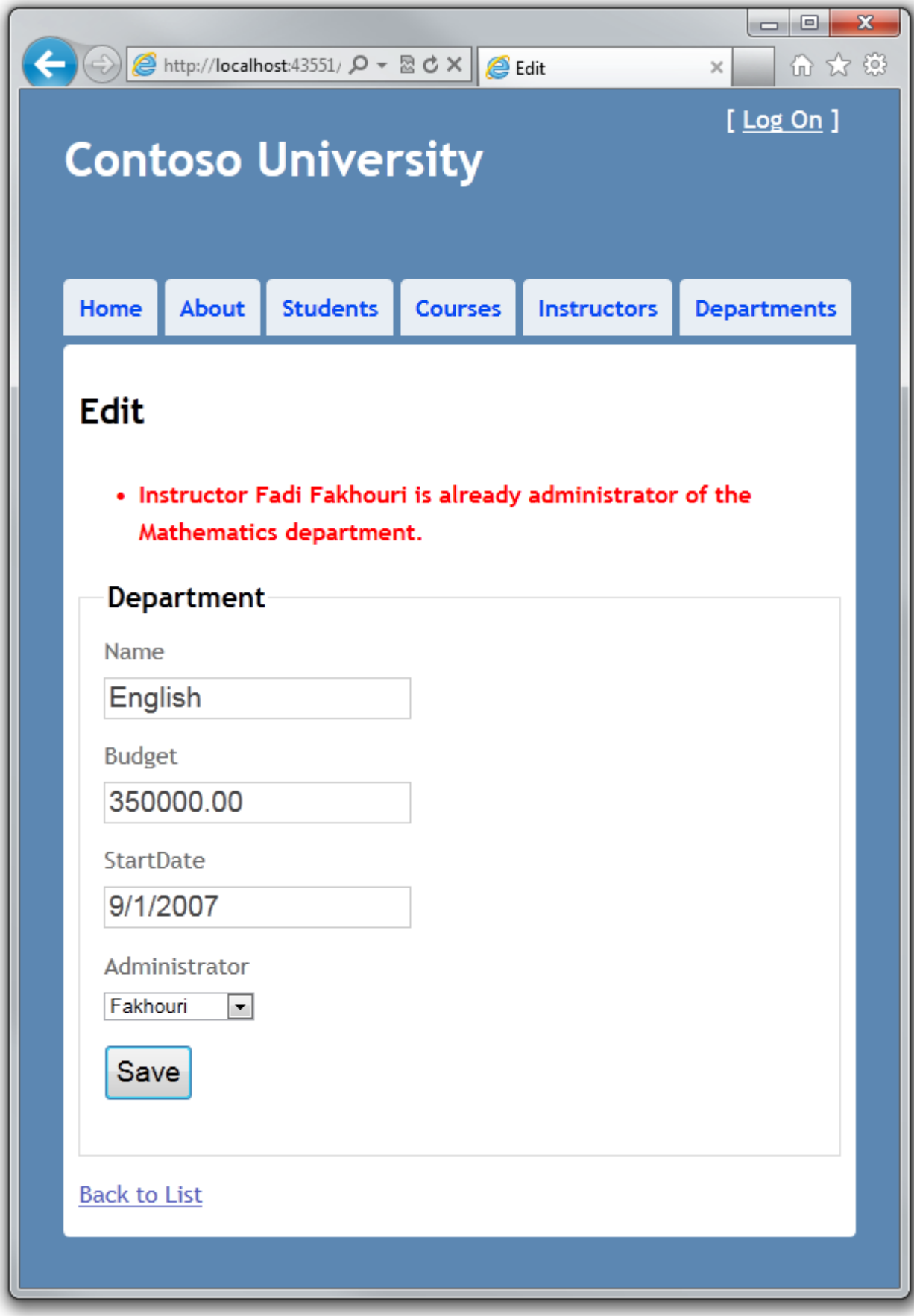
In the previous tutorial you implemented the repository and unit of work patterns. This tutorial covers the following topics:

- Performing raw SQL queries.
- Performing no-tracking queries.
- Examining queries sent to the database.
- Working with proxy classes.
- Disabling automatic detection of changes.
- Disabling validation when saving changes.

For most of these you will 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:


http://localhost:43551/
Edit
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Edit
Instructor Fadi Fakhouri is already administrator of the Mathematics department.
Department
Name
English
Budget
350000.00
StartDate
9/1/2007
Administrator
Fakhouri
Save
Back to List

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 `DbDatabase.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 `DbDatabase.SqlCommand` 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

Suppose you want the `GenericRepository` class to provide additional filtering and sorting flexibility without requiring that you create a derived class with additional methods. One way to achieve that would be to add a method that accepts a SQL query. You could then specify any kind of filtering or sorting you want in the controller, such as a `Where` clause that depends on a joins or subquery. In this section you'll see how to implement such a method.

Create the `GetWithRawSql` method by adding the following code to *GenericRepository.cs*:

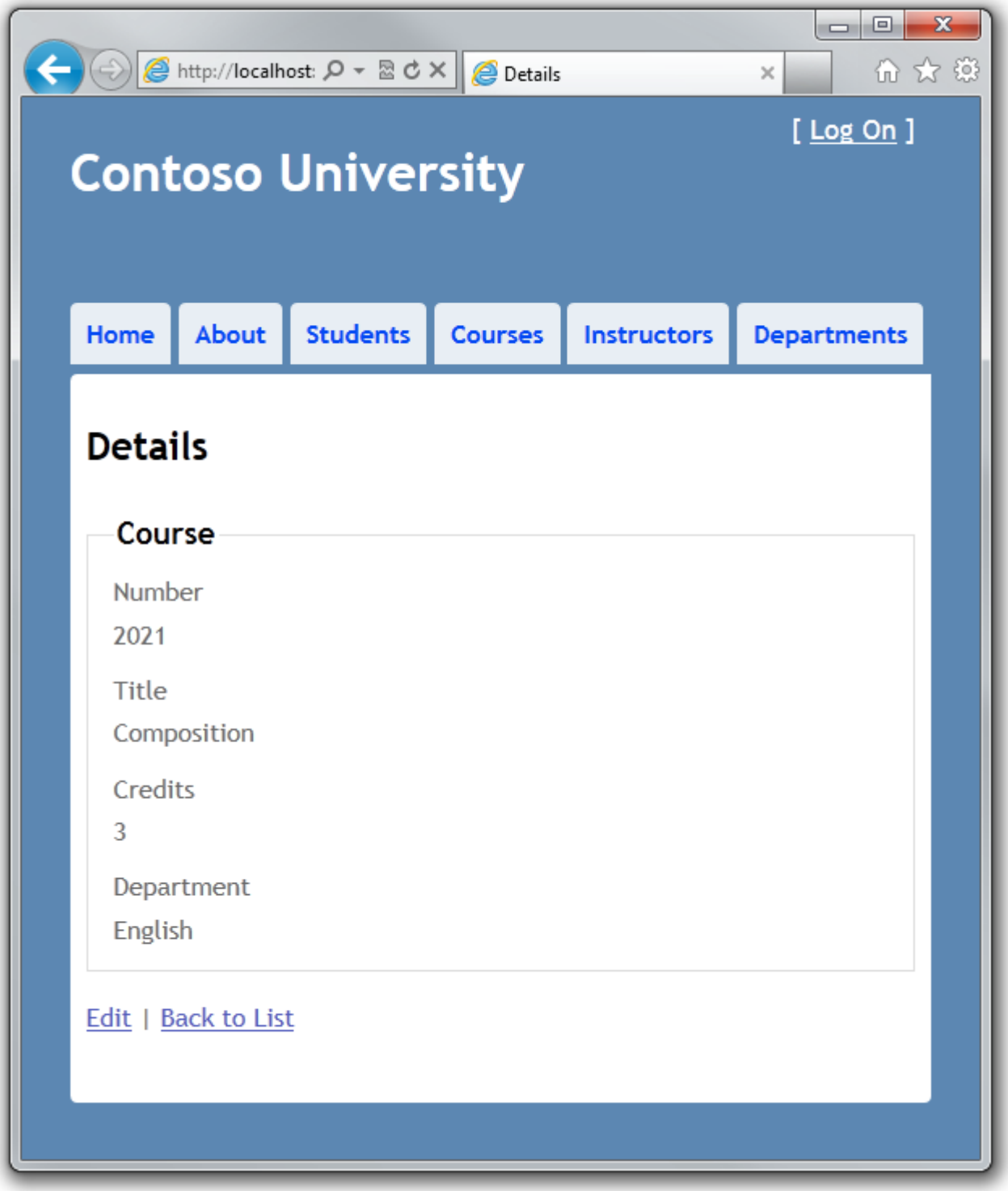
```
publicvirtualIEnumerable<TEntity>GetWithRawSql(string query,paramsobject[]
parameters)
{
return dbSet.SqlQuery(query, parameters).ToList();
}
```

In *CourseController.cs*, call the new method from the `Details` method, as shown in the following example:

```
publicActionResultDetails(int id)
{
var query ="SELECT * FROM Course WHERE CourseID = @p0";
returnView(unitOfWork.CourseRepository.GetWithRawSql(query, id).Single());
}
```

In this case you could have used the `GetByID` method, but you're using the `GetWithRawSql` method to verify that the `GetWithRawSQL` method works.

Run the Details page to verify that the select query works (select the **Course** tab and then **Details** for one course).



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:

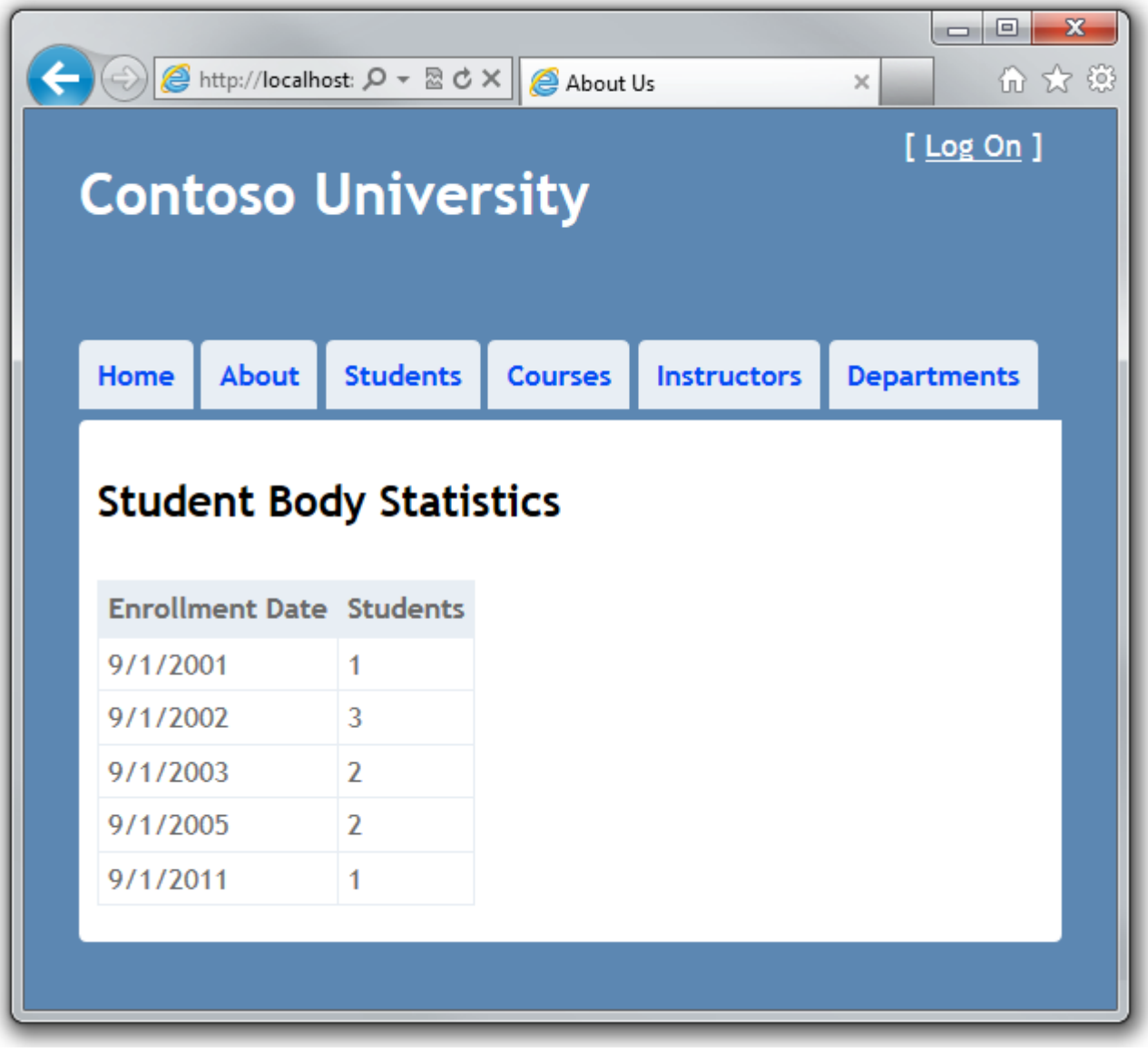
```
var data =from student in db.Students
group student by student.EnrollmentDateinto dateGroup
selectnewEnrollmentDateGroup()
{
EnrollmentDate= dateGroup.Key,
StudentCount= dateGroup.Count()
};
```

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 the following code:

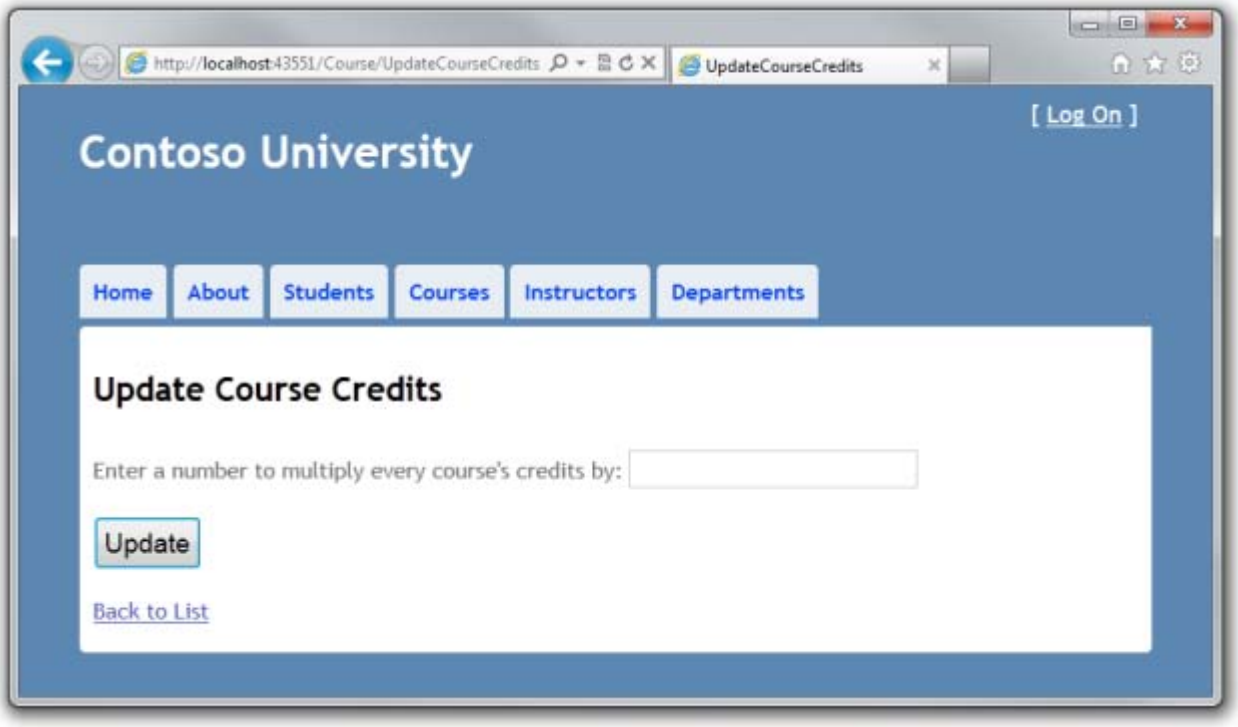
```
var query ="SELECT EnrollmentDate, COUNT(*) AS StudentCount "
+"FROM Person "
+"WHERE EnrollmentDate IS NOT NULL "
+"GROUP BY EnrollmentDate";
var data = db.Database.SqlQuery<EnrollmentDateGroup>(query);
```

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 allows 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 the previous tutorial you used the generic repository to read and update `Course` entities in the `Course` controller. For this bulk update operation, you need to create a new repository method that isn't in the generic repository. To do that, you'll create a dedicated `CourseRepository` class that derives from the `GenericRepository` class.

In the *DAL* folder, create *CourseRepository.cs* and replace the existing code with the following code:

```
usingSystem;
usingContosoUniversity.Models;

namespaceContosoUniversity.DAL
{
publicclassCourseRepository:GenericRepository<Course>
{
publicCourseRepository(SchoolContext context)
:base(context)
{
}

publicintUpdateCourseCredits(int multiplier)
{
```

```
return context.Database.ExecuteSqlCommand("UPDATE Course SET Credits = Credits *
{0}", multiplier);
}

}
}
```

In *UnitOfWork.cs*, change the `Course` repository type from `GenericRepository<Course>` to `CourseRepository:`

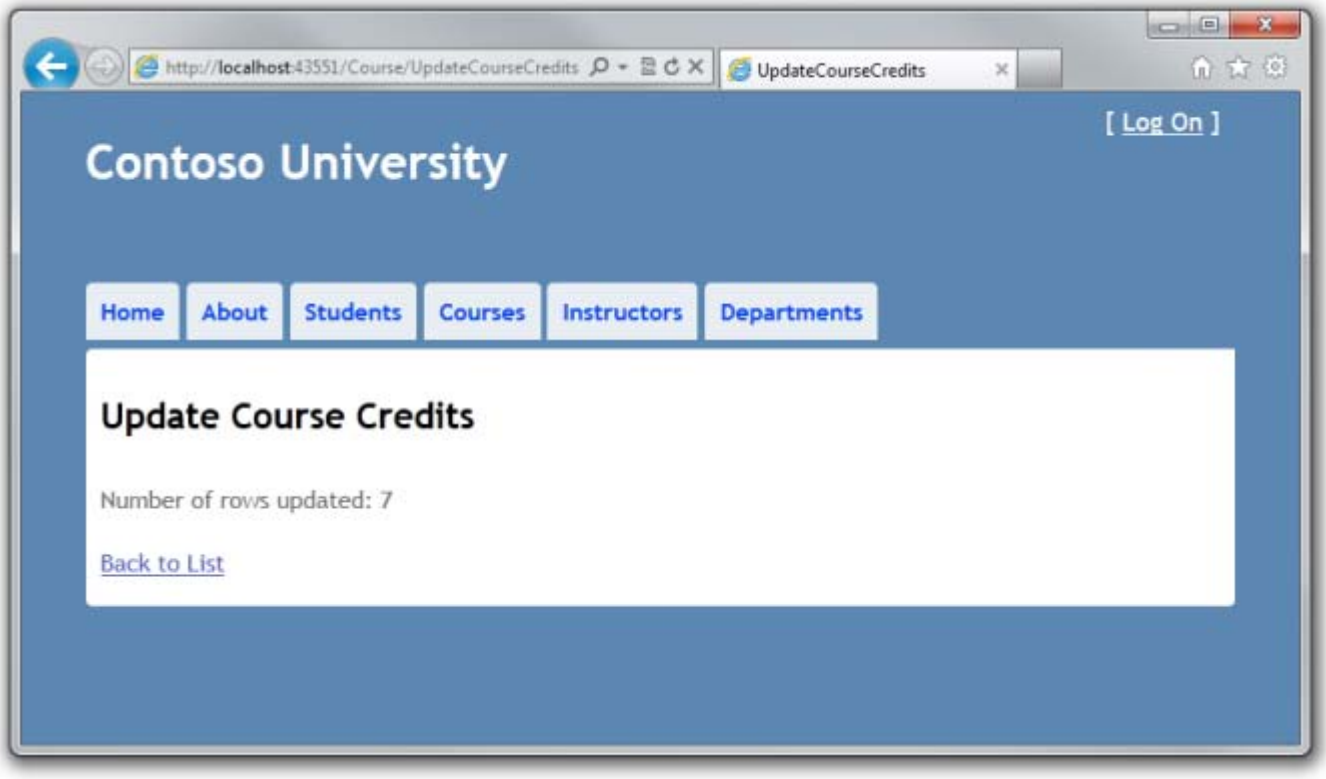
```
privateCourseRepository courseRepository;

publicCourseRepositoryCourseRepository
{
get
{

if(this.courseRepository ==null)
{
this.courseRepository =newCourseRepository(context);
}
return courseRepository;
}
}
```

In *CourseContoller.cs*, add an `UpdateCourseCredits` method:

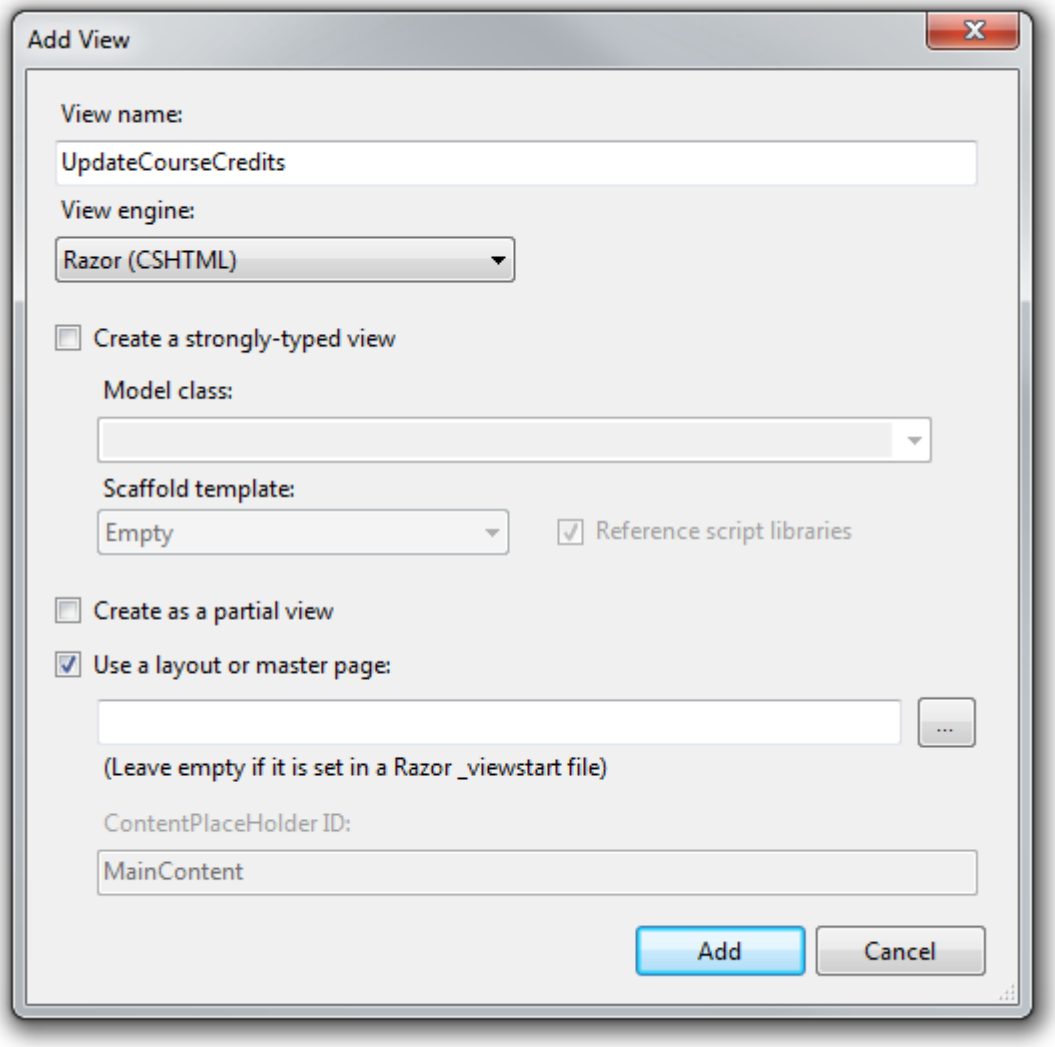
```
publicActionResultUpdateCourseCredits(int? multiplier)
{
if(multiplier !=null)
{
ViewBag.RowsAffected=
unitOfWork.CourseRepository.UpdateCourseCredits(multiplier.Value);
}
returnView();
}
```

This method will be used for both `HttpGet` and `HttpPost`. When the `HttpGetUpdateCourseCredits` method runs, the `multiplier` variable will be null and the view will display an empty text box and a submit button, as shown in the preceding illustration.

When the **Update** button is clicked and the `HttpPost` method runs, `multiplier` will have the value entered in the text box. The code then calls the repository `UpdateCourseCredits` method, which returns the number of affected rows, and that value is stored in the `ViewBag` object. When the view receives the number of affected rows in the `ViewBag` object, it displays that number instead of the text box and submit button, as shown in the following illustration:



Create a view in the *Views\Course* folder for the Update Course Credits page:

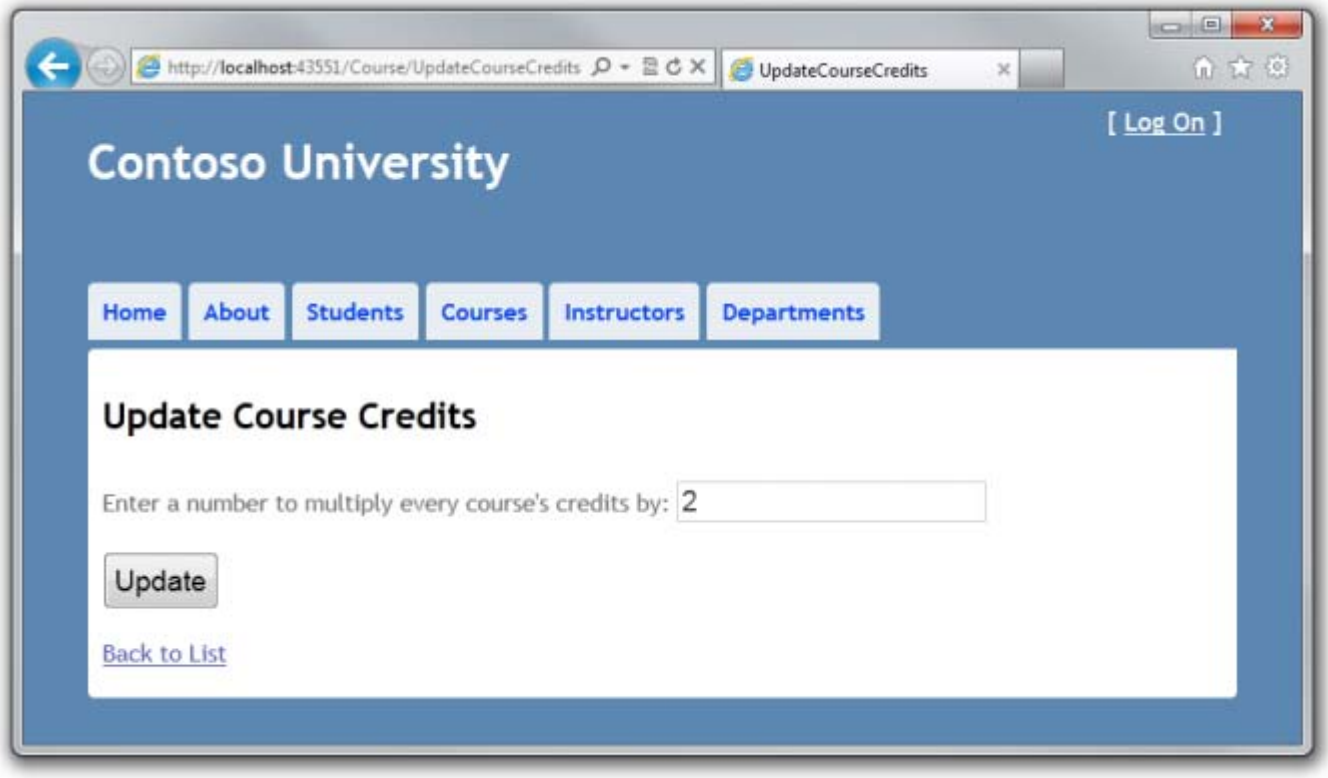


In *Views\Course\UpdateCourseCredits.cshtml*, replace the existing code with the following code:

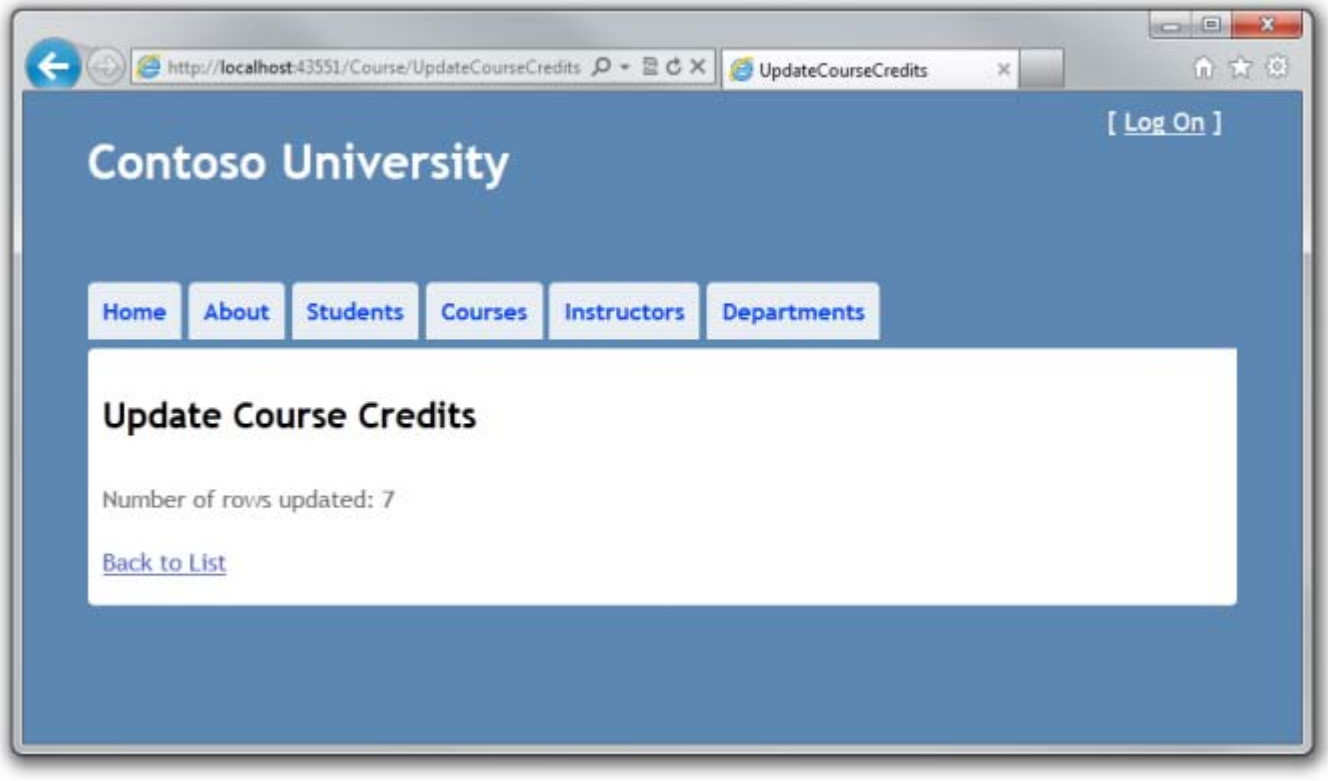
```
@modelContosoUniversity.Models.Course

@{
ViewBag.Title="UpdateCourseCredits";
}

<h2>UpdateCourseCredits</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>
```

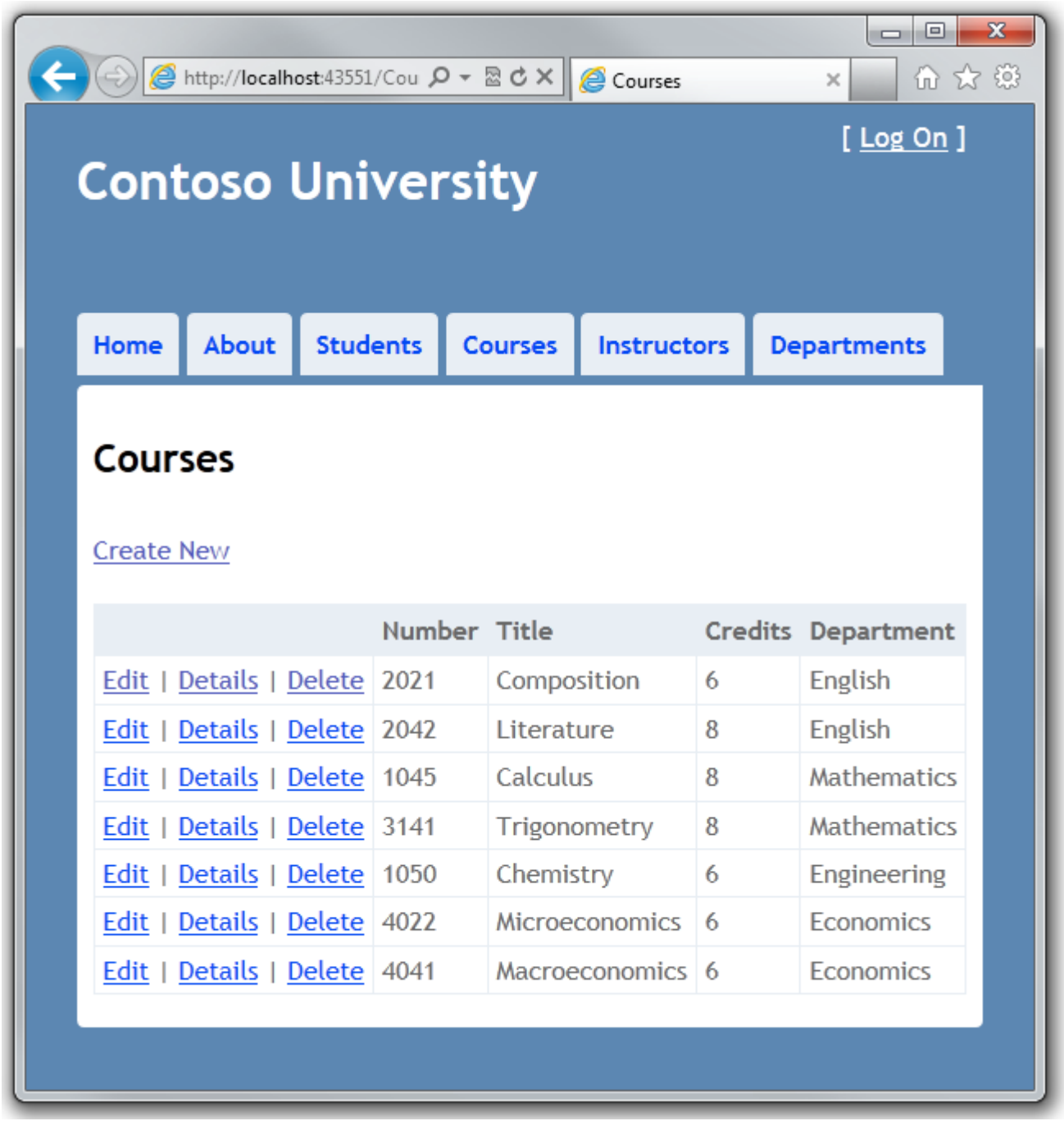
Run the page by selecting the **Courses** tab, then adding "/UpdateCourseCredits" to the end of the URL in the browser's address bar (for example: *http://localhost:50205/Course/UpdateCourseCredits*). Enter a number in the text box:



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 the Entity Framework team blog.

No-Tracking Queries

When a database context retrieves database 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 specify whether the context tracks entity objects for a query by using the `AsNoTracking` method. Typical scenarios in which you might want to do that include the following:

- The 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 prevent this from happening 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.

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:

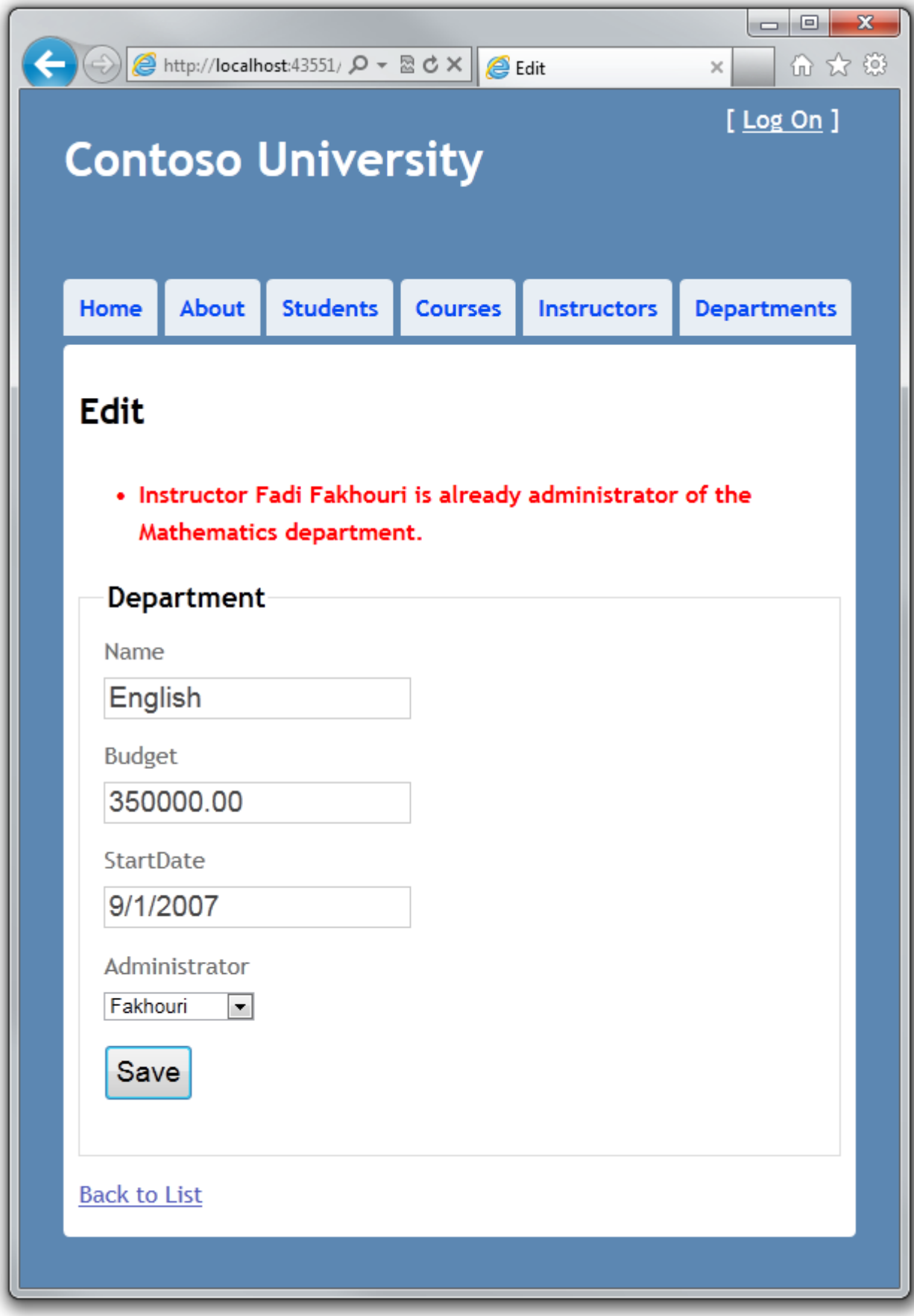
```
privatevoidValidateOneAdministratorAssignmentPerInstructor(Department department)
{
if(department.PersonID!=null)
{
var duplicateDepartment = db.Departments
.Include("Administrator")
.Where(d => d.PersonID== department.PersonID)
.FirstOrDefault();
if(duplicateDepartment !=null&& duplicateDepartment.DepartmentID!=
department.DepartmentID)
{
var 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);
}
}
}
```

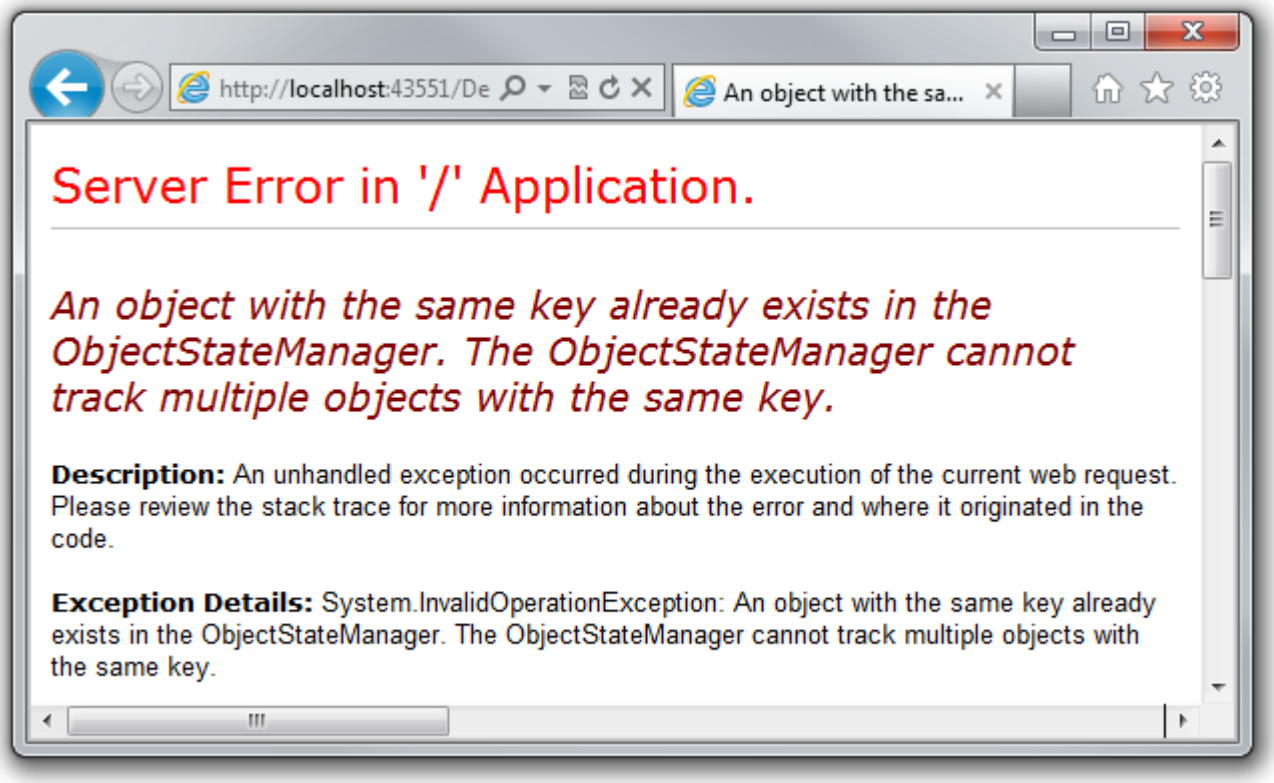
Add code in the `try` block of the `HttpPostEdit` method to call this new method if there are no validation errors. The `try` block now looks like the following example:

```
if(ModelState.IsValid)
{
ValidateOneAdministratorAssignmentPerInstructor(department);
}
if(ModelState.IsValid)
{
    db.Entry(department).State=EntityState.Modified;
    db.SaveChanges();
returnRedirectToAction("Index");
}
```

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:


http://localhost:43551/
Edit
[Log On]
Contoso University
Home
About
Students
Courses
Instructors
Departments
Edit
Instructor Fadi Fakhouri is already administrator of the Mathematics department.
Department
Name
English
Budget
350000.00
StartDate
9/1/2007
Administrator
Fakhouri
Save
Back to List

Now run the Department Edit page again and this time change the **Budget** amount. When you click **Save**, you see an error page:



The exception error message is "`An object with the same key already exists in the ObjectStateManager. The ObjectStateManager cannot track multiple objects with the same key.`" 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. Because that's the department being edited, no error is reported. As a result of this read operation, however, 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, but that fails 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 example:

```
var 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 Queries Sent to the Database

Sometimes it's helpful to be able to see the actual SQL queries that are sent to the database. To do this, you can examine a query variable in the debugger or call the query's `ToString` method. To try this out, you'll look at a simple query and then look at what happens to it as you add options such eager loading, filtering, and sorting.

In *Controllers/CourseController*, replace the `Index` method with the following code:

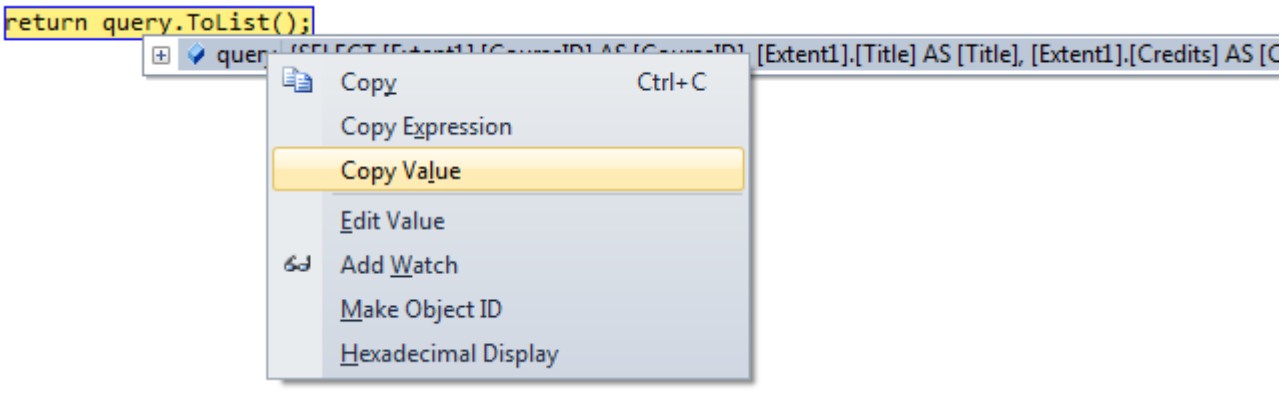
```
publicViewResultIndex()
{
var courses = unitOfWork.CourseRepository.Get();
returnView(courses.ToList());
}
```

Now set a breakpoint in *GenericRepository.cs* on the `return query.ToList();` and the return orderBy(query).ToList(); statements of the `Get` method. 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 Compact. 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]}
```

Queries can be too long to display in the debugging windows in Visual Studio. To see the entire query, you can copy the variable value and paste it into a text editor:



Now you'll add a drop-down list to the Course 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:

```
publicActionResultIndex(int?SelectedDepartment)
{
var departments = unitOfWork.DepartmentRepository.Get(
        orderBy: q => q.OrderBy(d => d.Name));
ViewBag.SelectedDepartment=newSelectList(departments,"DepartmentID","Name",SelectedDe
partment);

int departmentID =SelectedDepartment.GetValueOrDefault();
returnView(unitOfWork.CourseRepository.Get(
        filter: d =>!SelectedDepartment.HasValue|| d.DepartmentID== departmentID,
        orderBy: q => q.OrderBy(d => d.CourseID),
        includeProperties:"Department"));
}
```

The method receives the selected value of the drop-down list in the `SelectedDepartment` parameter. If nothing is selected, this parameter will be null.

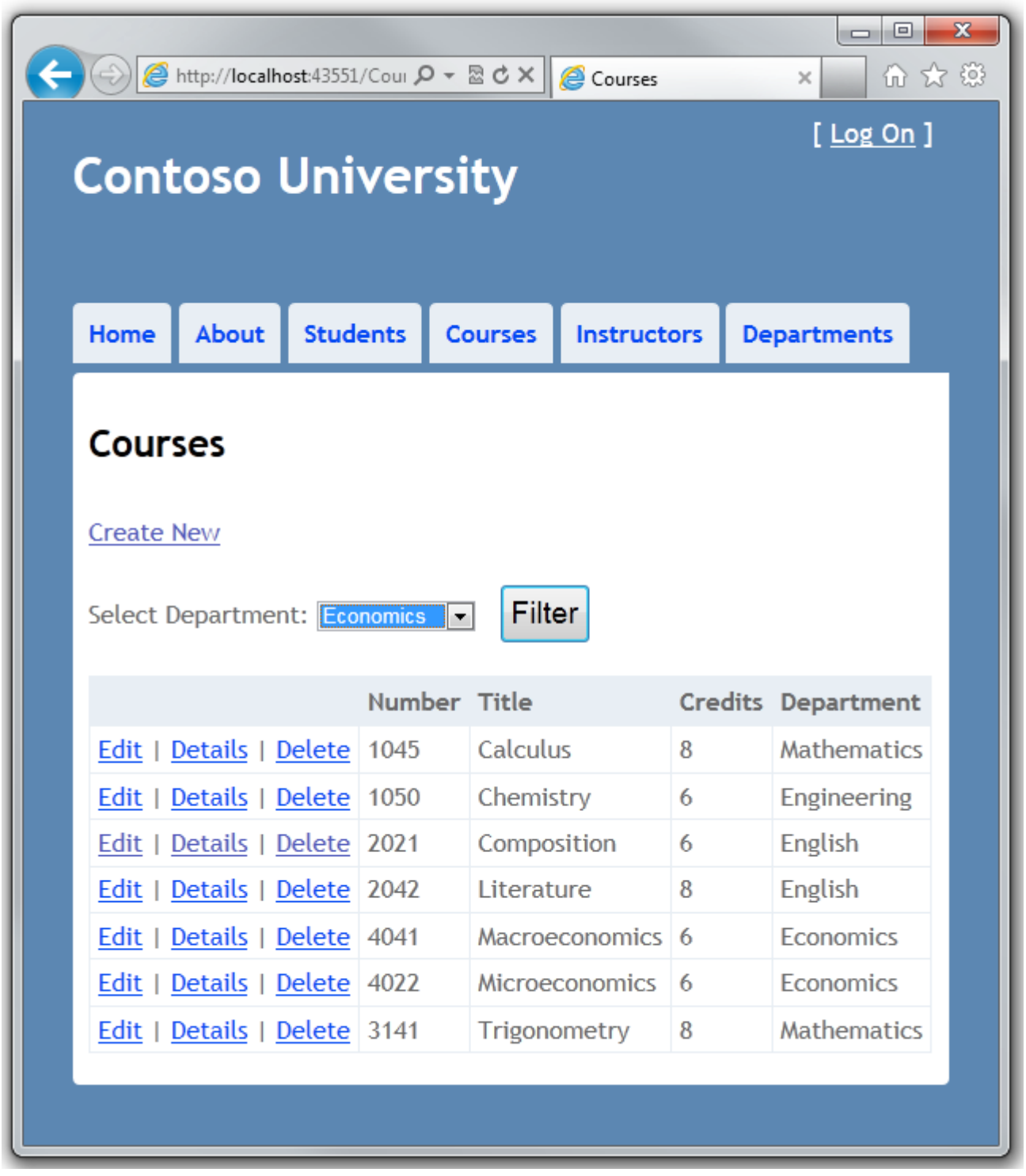
A `SelectList` collection containing all departments is passed to the view for the drop-down list. The parameters passed to the `SelectList` constructor specify the value field name, the text field name, and the selected item.

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>SelectDepartment:@Html.DropDownList("SelectedDepartment","All")
<input type="submit" value="Filter"/></p>
}
```

With the breakpoints still set in the `GenericRepository` class, run the Course Index page. Continue through the first two 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
[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].[PersonID] AS [PersonID],
[Extent2].[Timestamp] AS [Timestamp]
FROM  [Course] AS [Extent1]
INNER JOIN [Department] AS [Extent2] ON
[Extent1].[DepartmentID]=[Extent2].[DepartmentID]
WHERE (@p__linq__0 IS NULL) OR ([Extent1].[DepartmentID]=@p__linq__1)}
```

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.

Working with 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. This proxy overrides some virtual properties of the entity to insert hooks for performing actions automatically when the property is accessed. For example, this mechanism is used to support lazy loading of relationships.

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, serializing non-proxy instances might be more efficient than serializing proxy instances.
- 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 the Entity Framework team blog.

Disabling Automatic Detection of Changes

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 was 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 the Entity Framework team blog.

Disabling Validation When Saving Changes

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 the Entity Framework team blog.

Links to Entity Framework Resources

This completes this series of tutorials on using the Entity Framework in an ASP.NET MVC application. For more information about the Entity Framework, see the following resources:

- Introduction to the Entity Framework 4.1 (Code First)
- The Entity Framework Code First Class Library API Reference
- Entity Framework FAQ
- The Entity Framework Team Blog
- Entity Framework in the MSDN Library
- Entity Framework in the MSDN Data Developer Center
- Entity Framework Forums on MSDN
- Julie Lerman's blog
- Code First DataAnnotations Attributes
- Maximizing Performance with the Entity Framework in an ASP.NET Web Application
- Profiling Database Activity in the Entity Framework
- Entity Framework Power Tools

The following posts on the Entity Framework Team Blog provide more information about some of the topics covered in these tutorials:

- Fluent API Samples. How to customize mapping using fluent API method calls.
- Connections and Models. How to connect to different types of databases.
- Pluggable Conventions. How to change conventions.
- Finding Entities. How to use the `Find` method with composite keys.
- Loading Related Entities. Additional options for eager, lazy, and explicit loading.
- Load and AsNoTracking. More on explicit loading.

Many of the blog posts listed here are for the CTP5 version of Entity Framework Code First. Most of the information in them remains accurate, but there are some changes between CTP5 and the officially released version of Code First.

For information about how to use LINQ with the Entity Framework, see LINQ to Entities in the MSDN Library.

For a tutorial that uses more MVC features and uses the Entity Framework, see MVC Music Store.

For information about how to deploy your web application after you've built it, see ASP.NET Deployment Content Map in the MSDN Library.