Getting Started with Entity Framework 6 Code First using MVC 5

Tom Dykstra, Rick Anderson

Step By Step, Guide

Microsoft

Advanced Entity Framework 6 Scenarios for an MVC 5 Web Application (12 of 12)

In the previous tutorial you implemented table-per-hierarchy inheritance. This tutorial includes introduces several topics that are useful to be aware of when you go beyond the basics of developing ASP.NET web applications that use Entity Framework Code First. Step-by-step instructions walk you through the code and using Visual Studio for the following topics:

- Performing raw SQL queries
- Performing no-tracking queries
- Examining SQL sent to the database

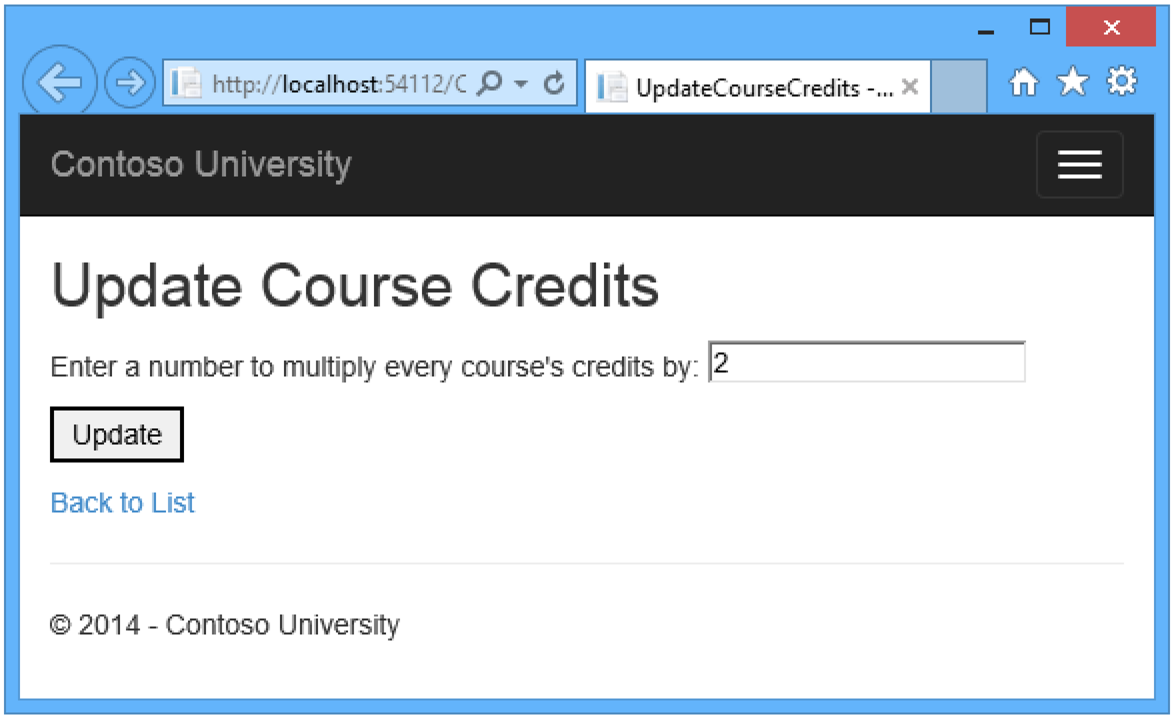
The tutorial introduces several topics with brief introductions followed by links to resources for more information:

- Repository and unit of work patterns
- Proxy classes
- Automatic change detection
- Automatic validation
- EF tools for Visual Studio
- Entity Framework source code

The tutorial also includes the following sections:

- Summary
- Acknowledgments
- A note about VB
- Common errors, and solutions or workarounds for them

For most of these topics, you'll work with pages that you already created. To use raw SQL to do bulk updates you'll create a new page that updates the number of credits of all courses in the database:

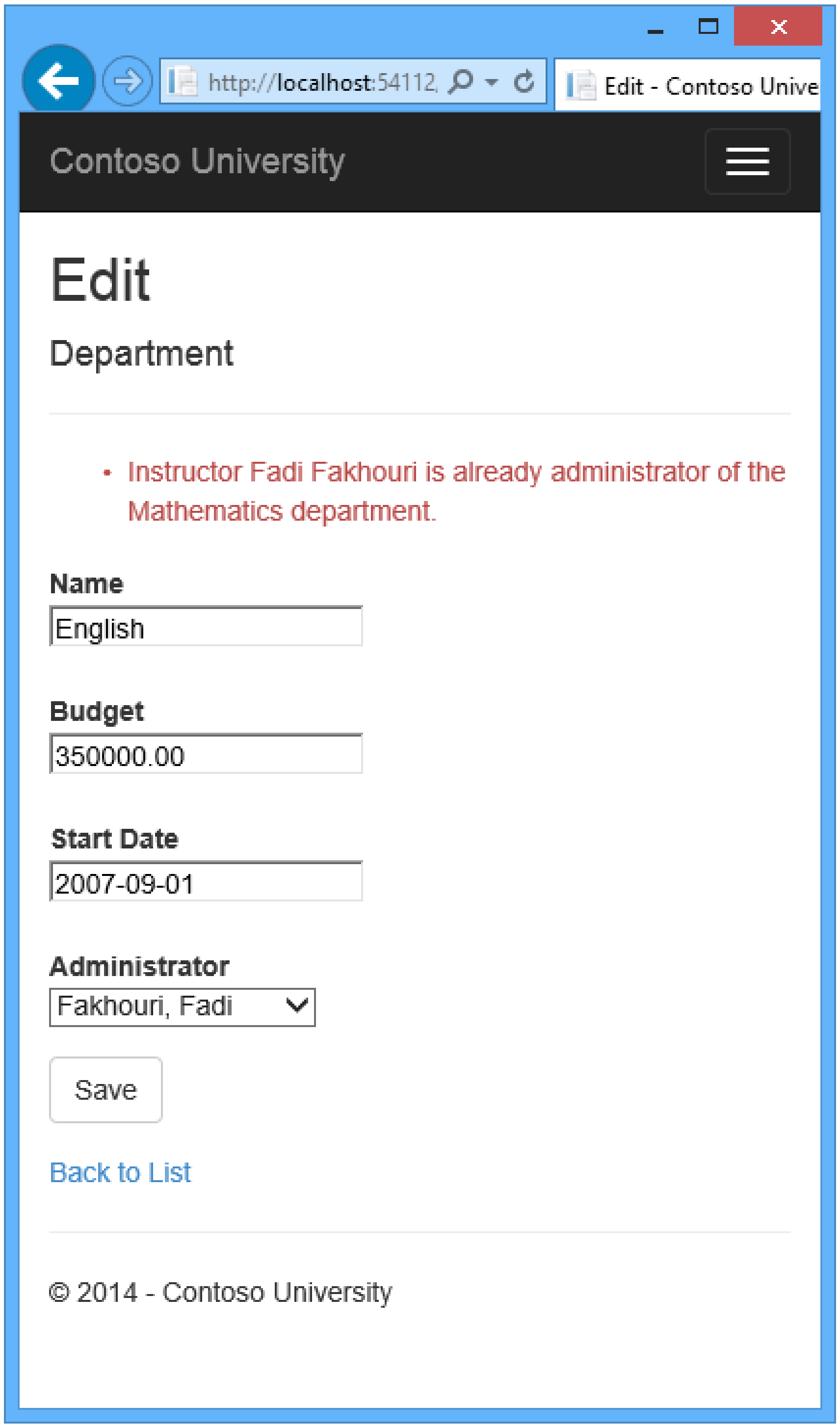


Update Course Credits

Enter a number to multiply every course's credits by: 2

© 2014 - Contoso University

And to use a no-tracking query you'll add new validation logic to the Department Edit page:



Contoso University

Edit

Department

- Instructor Fadi Fakhouri is already administrator of the Mathematics department.

Name: English

Budget: 350000.00

Start Date: 2007-09-01

Administrator: Fakhouri, Fadi

© 2014 - Contoso University

Performing Raw SQL Queries

The Entity Framework Code First API includes methods that enable you to pass SQL commands directly to the database. You have the following options:

- Use the DbSet.SqlQuery method for queries that return entity types. The returned objects must be of the type expected by the `DbSet` object, and they are automatically tracked by the database context unless you turn tracking off. (See the following section about the AsNoTracking method.)
- Use the Database.SqlQuery method for queries that return types that aren't entities. The returned data isn't tracked by the database context, even if you use this method to retrieve entity types.
- Use the Database.ExecuteSqlCommand for non-query commands.

One of the advantages of using the Entity Framework is that it avoids tying your code too closely to a particular method of storing data. It does this by generating SQL queries and commands for you, which also frees you from having to write them yourself. But there are exceptional scenarios when you need to run specific SQL queries that you have manually created, and these methods make it possible for you to handle such exceptions.

As is always true when you execute SQL commands in a web application, you must take precautions to protect your site against SQL injection attacks. One way to do that is to use parameterized queries to make sure that strings submitted by a web page can't be interpreted as SQL commands. In this tutorial you'll use parameterized queries when integrating user input into a query.

Calling a Query that Returns Entities

The DbSet<TEntity> class provides a method that you can use to execute a query that returns an entity of type `TEntity`. To see how this works you'll change the code in the `Details` method of the `Department` controller.

In *DepartmentController.cs*, replace the `db.Departments.Find` method call with a `db.Departments.SqlQuery` method call, as shown in the following highlighted code:

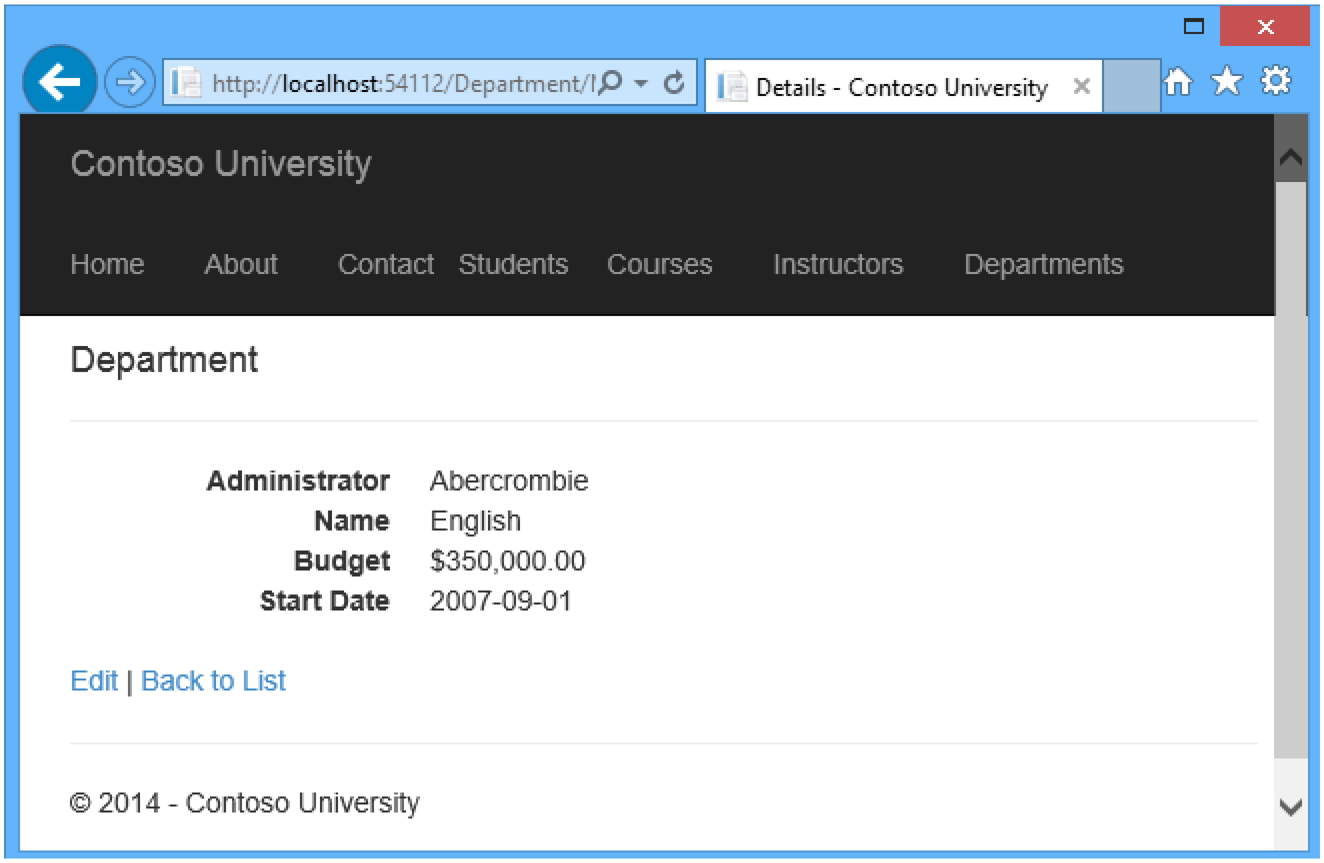
```
public async Task<ActionResult> Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    // Commenting out original code to show how to use a raw SQL query.
    //Department department = await db.Departments.FindAsync(id);

    // Create and execute raw SQL query.
    string query = "SELECT * FROM Department WHERE DepartmentID = @p0";
    Department department = await db.Departments.SqlQuery(query, 
id).SingleOrDefaultAsync();

    if (department == null)
    {
        return HttpNotFound();
    }
    return View(department);
}
```

To verify that the new code works correctly, select the **Departments** tab and then **Details** for one of the departments.



Department

Administrator	Abercrombie
Name	English
Budget	$350,000.00
Start Date	2007-09-01

Edit | Back to List

© 2014 - Contoso University

Calling a Query that Returns Other Types of Objects

Earlier you created a student statistics grid for the About page that showed the number of students for each enrollment date. The code that does this in *HomeController.cs* uses LINQ:

```
var data = from student in db.Students
           group student by student.EnrollmentDate into dateGroup
           select new EnrollmentDateGroup()
           {
               EnrollmentDate = dateGroup.Key,
               StudentCount = dateGroup.Count()
           };
```

Suppose you want to write the code that retrieves this data directly in SQL rather than using LINQ. To do that you need to run a query that returns something other than entity objects, which means you need to use the Database.SqlQuery method.

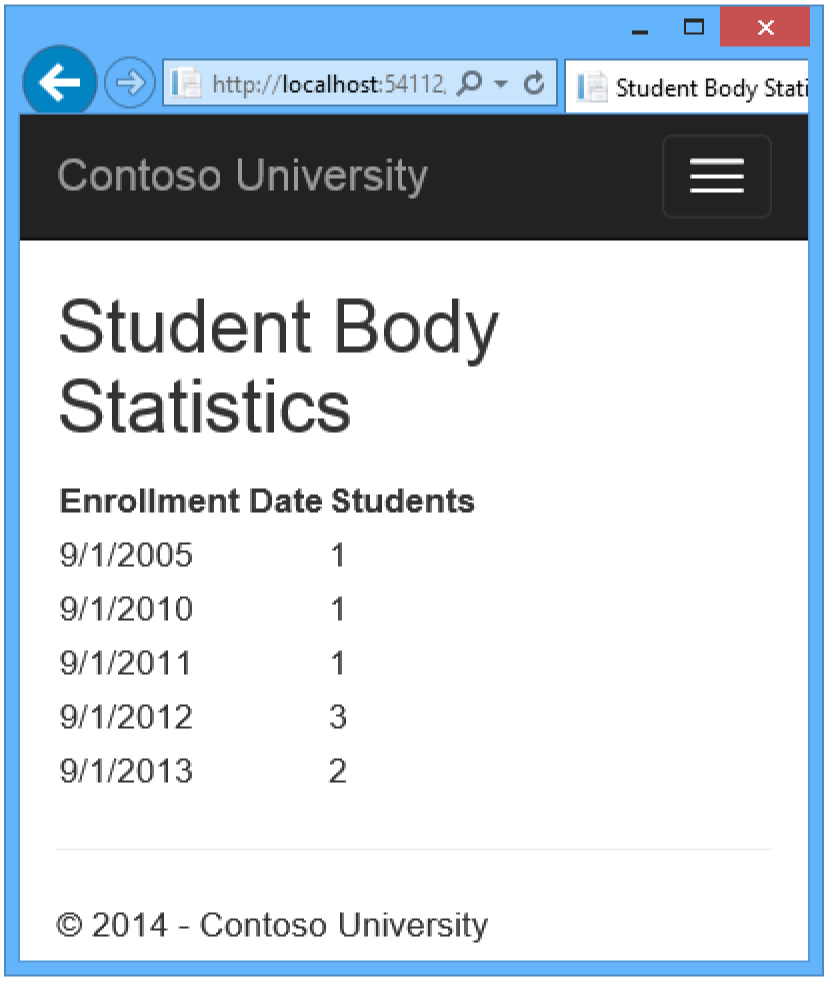
In *HomeController.cs*, replace the LINQ statement in the `About` method with a SQL statement, as shown in the following highlighted code:

```
public ActionResult About()
{
    // Commenting out LINQ to show how to do the same thing in SQL.
    //IQueryable<EnrollmentDateGroup> = from student in db.Students
    //             group student by student.EnrollmentDate into dateGroup
    //             select new EnrollmentDateGroup()
    //             {
    //                 EnrollmentDate = dateGroup.Key,
    //                 StudentCount = dateGroup.Count()
    //             };

    // SQL version of the above LINQ code.
    string query = "SELECT EnrollmentDate, COUNT(*) AS StudentCount "
        + "FROM Person "
        + "WHERE Discriminator = 'Student' "
        + "GROUP BY EnrollmentDate";
    IEnumerable<EnrollmentDateGroup> data =
db.Database.SqlQuery<EnrollmentDateGroup>(query);

    return View(data.ToList());
}
```

Run the About page. It displays the same data it did before.



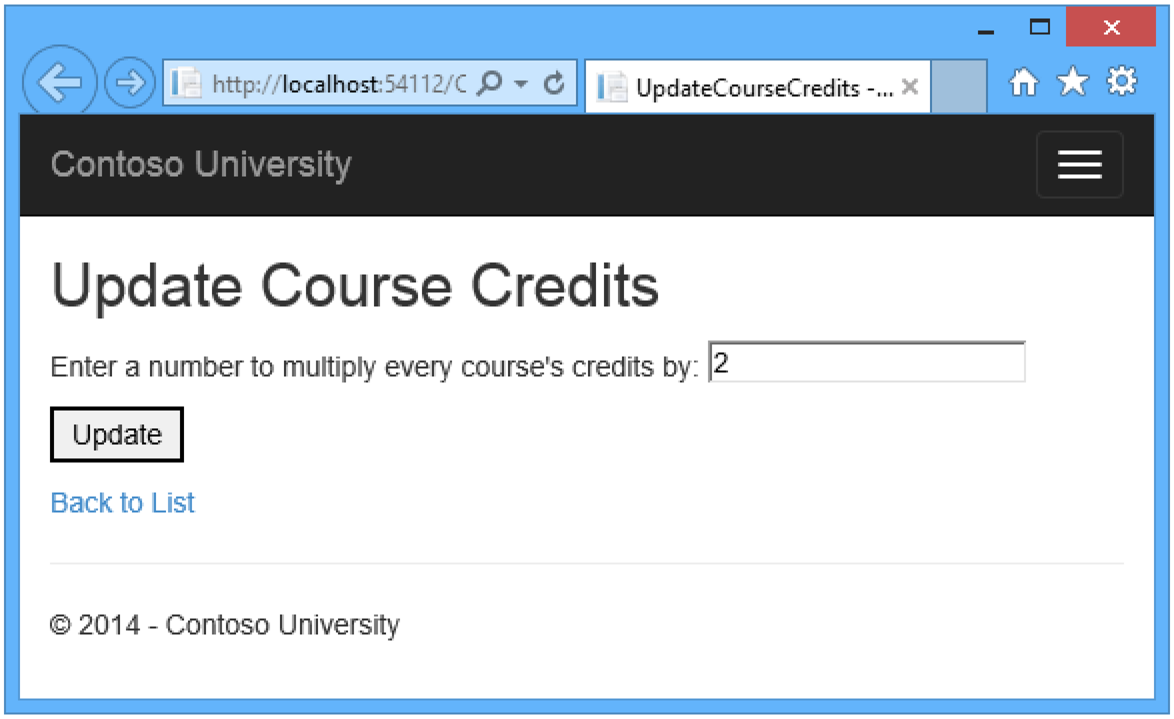
Student Body Statistics

Enrollment Date	Students
9/1/2005	1
9/1/2010	1
9/1/2011	1
9/1/2012	3
9/1/2013	2

© 2014 - Contoso University

Calling an Update Query

Suppose Contoso University administrators want to be able to perform bulk changes in the database, such as changing the number of credits for every course. If the university has a large number of courses, it would be inefficient to retrieve them all as entities and change them individually. In this section you'll implement a web page that enables the user to specify a factor by which to change the number of credits for all courses, and you'll make the change by executing a SQL `UPDATE` statement. The web page will look like the following illustration:



Update Course Credits

Enter a number to multiply every course's credits by: 2

© 2014 - Contoso University

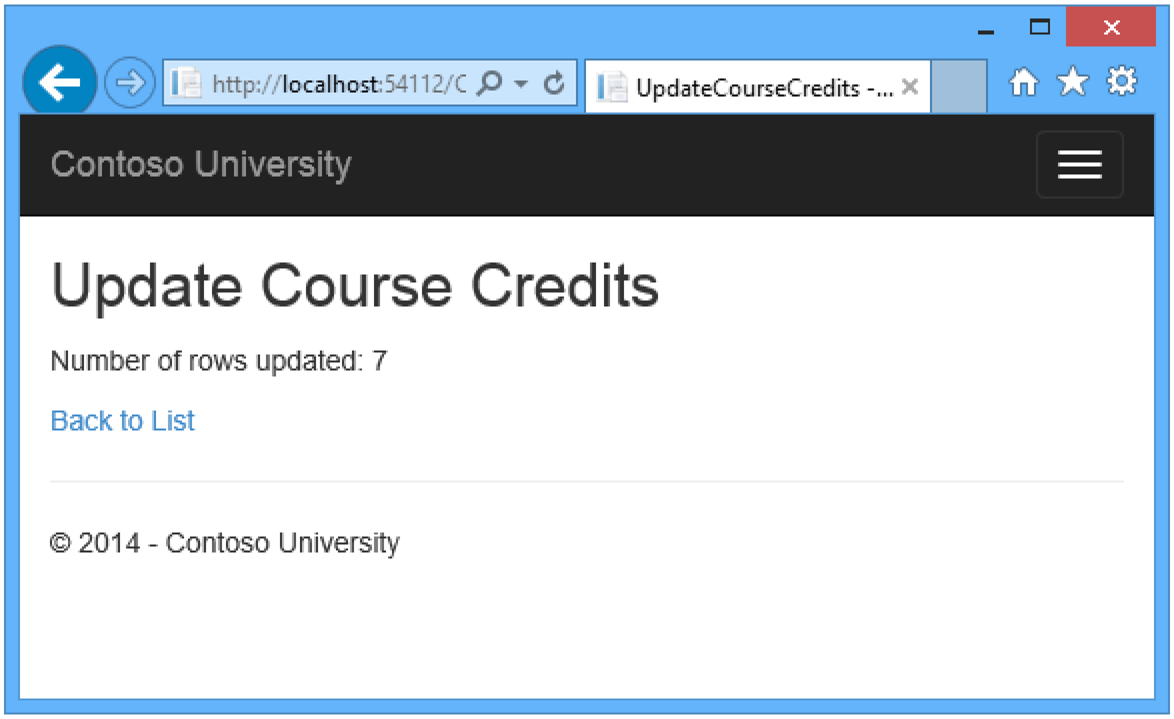
In *CourseContoller.cs*, add `UpdateCourseCredits` methods for `HttpGet` and `HttpPost`:

```
public ActionResult UpdateCourseCredits()
{
    return View();
}

[HttpPost]
public ActionResult UpdateCourseCredits(int? multiplier)
{
    if (multiplier != null)
    {
        ViewBag.RowsAffected = db.Database.ExecuteSqlCommand("UPDATE Course 
SET Credits = Credits * {0}", multiplier);
    }
    return View();
}
```

When the controller processes an `HttpGet` request, nothing is returned in the `ViewBag.RowsAffected` variable, and the view displays an empty text box and a submit button, as shown in the preceding illustration.

When the **Update** button is clicked, the `HttpPost` method is called, and `multiplier` has the value entered in the text box. The code then executes the SQL that updates courses and returns the number of affected rows to the view in the `ViewBag.RowsAffected` variable. When the view gets a value in that variable, it displays the number of rows updated instead of the text box and submit button, as shown in the following illustration:



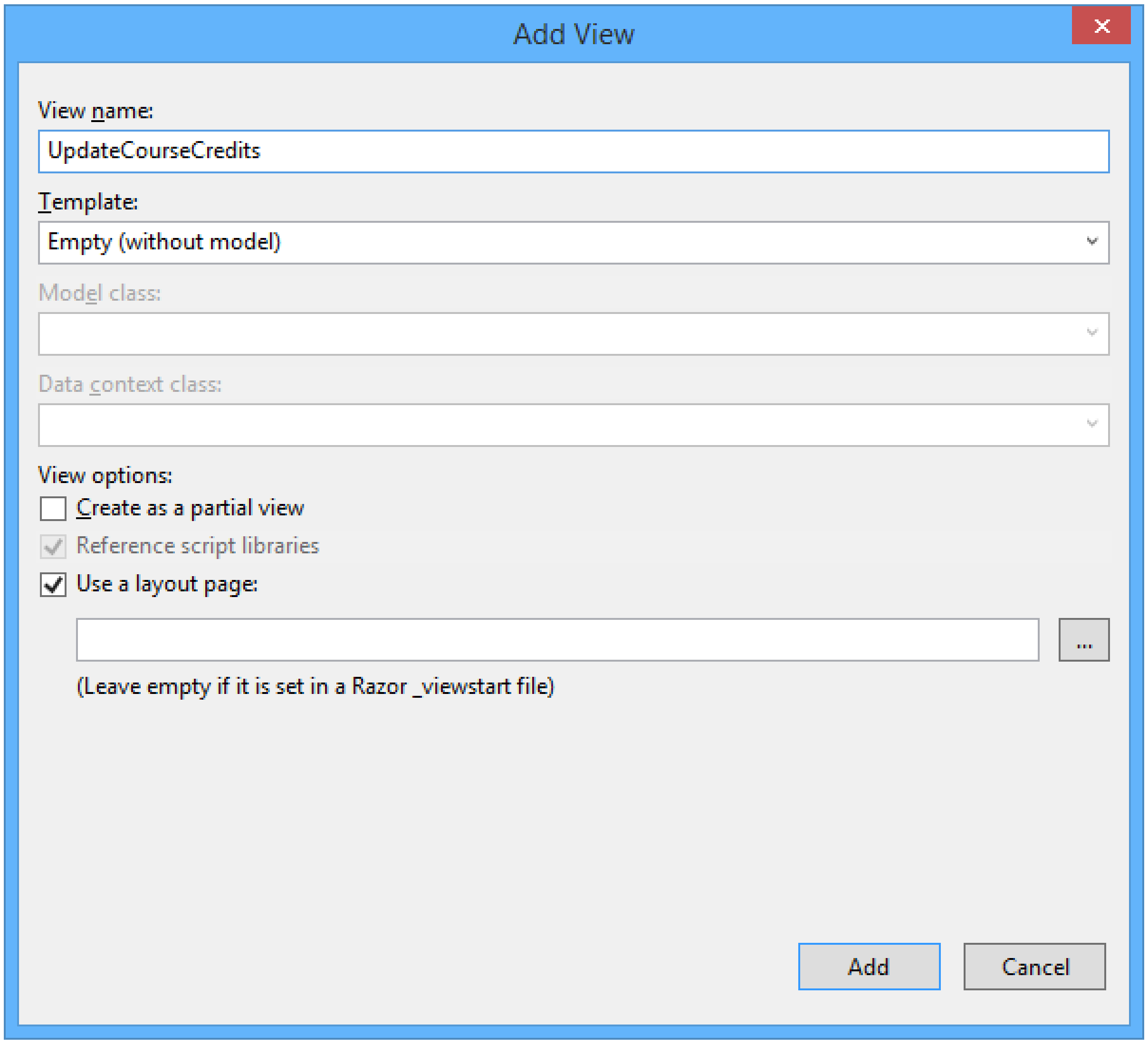
Update Course Credits

Number of rows updated: 7

Back to List

© 2014 - Contoso University

In *CourseController.cs*, right-click one of the `UpdateCourseCredits` methods, and then click **Add**.


Add View

View name:
UpdateCourseCredits

Template:
Empty (without model)

Model class:

Data context class:

View options:
- [ ] Create as a partial view
- [x] Reference script libraries
- [x] Use a layout page:

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

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

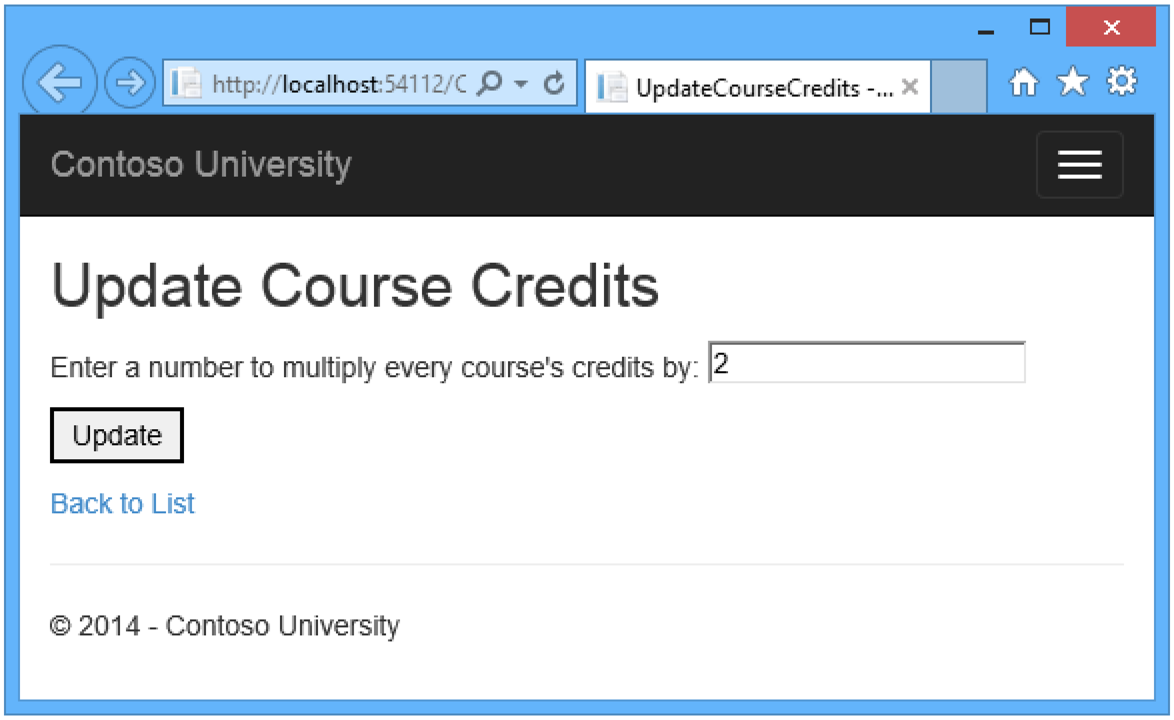
```
@model ContosoUniversity.Models.Course

@{
    ViewBag.Title = "UpdateCourseCredits";
}

<h2>Update Course Credits</h2>

@if (ViewBag.RowsAffected == null)
{
    using (Html.BeginForm())
    {
        <p>
            Enter a number to multiply every course's credits by:
@Html.TextBox("multiplier")
        </p>
        <p>
            <input type="submit" value="Update" />
        </p>
    }
}
@if (ViewBag.RowsAffected != null)
{
    <p>
        Number of rows updated: @ViewBag.RowsAffected
    </p>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

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

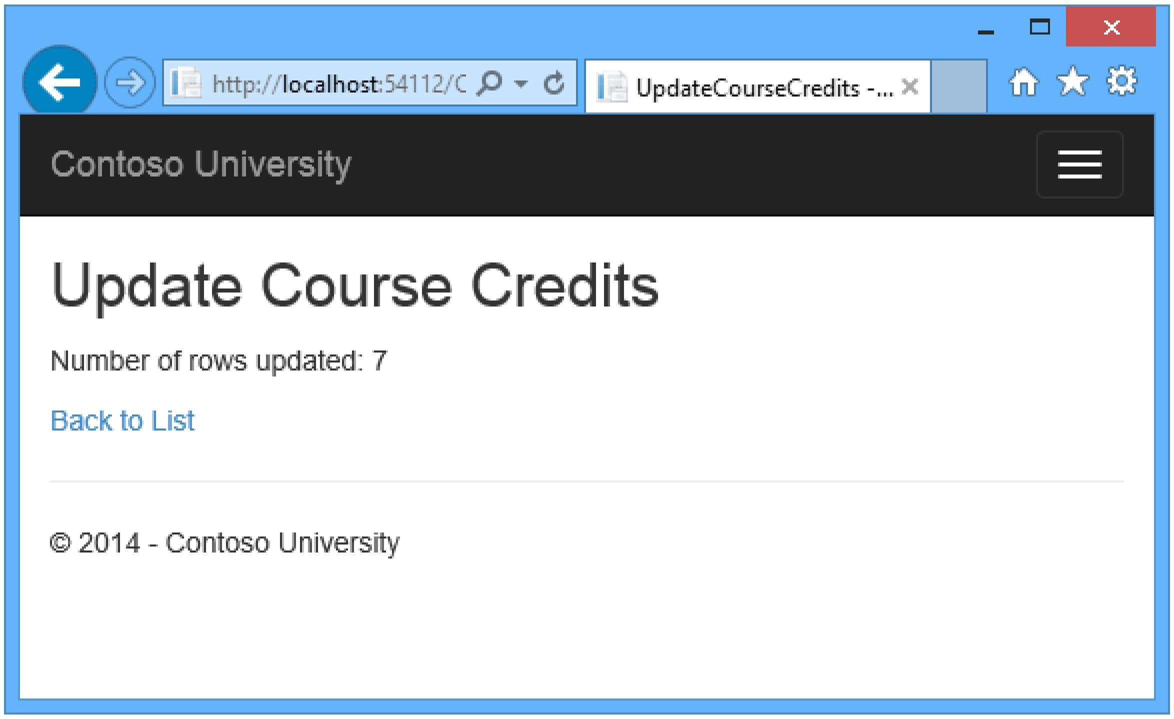


Update Course Credits

Enter a number to multiply every course's credits by: 2

© 2014 - Contoso University

Click **Update**. You see the number of rows affected:



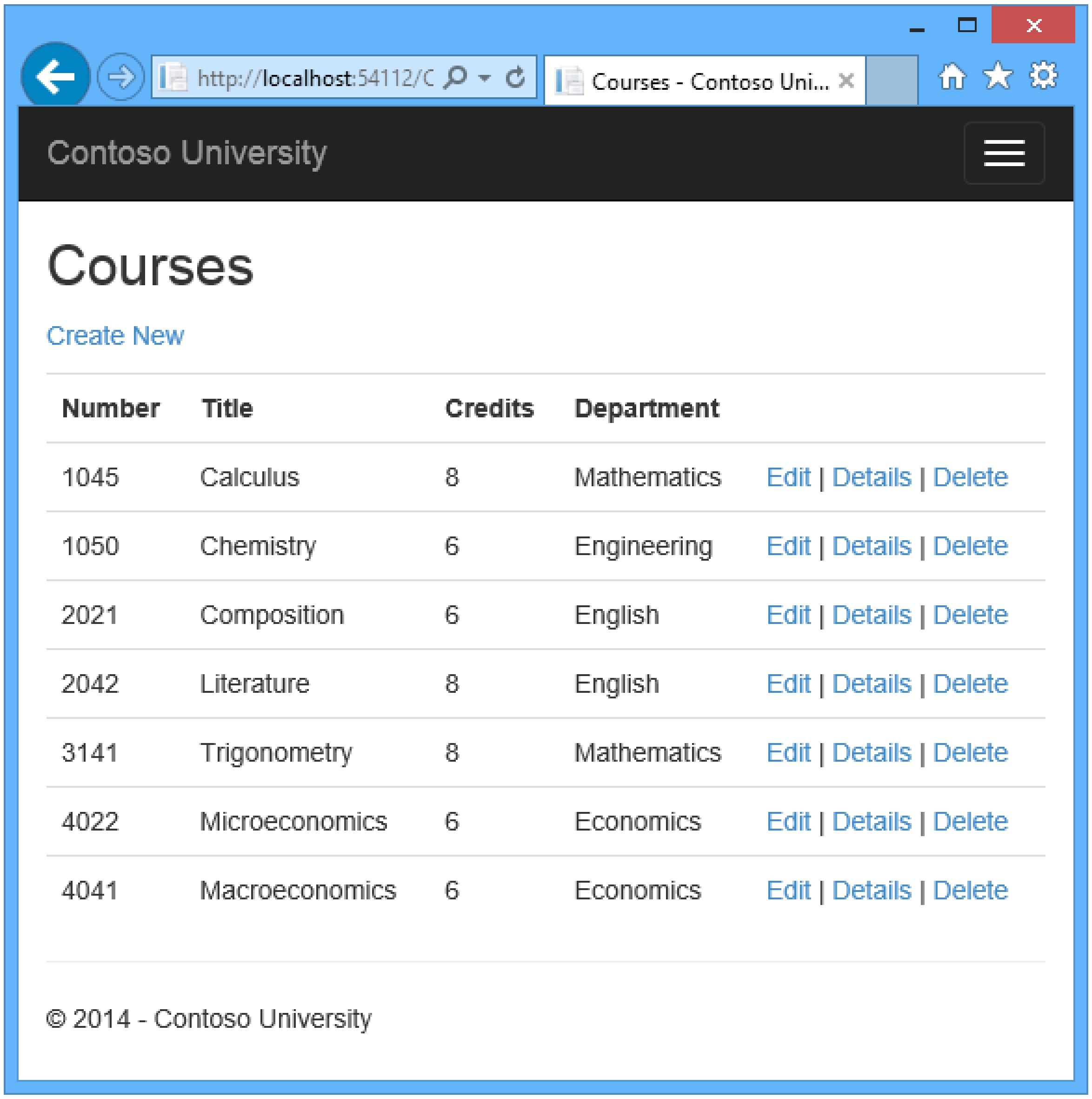
Update Course Credits

Number of rows updated: 7

Back to List

© 2014 - Contoso University

Click **Back to List** to see the list of courses with the revised number of credits.



Courses

Create New

Number	Title	Credits	Department	
1045	Calculus	8	Mathematics	Edit | Details | Delete
1050	Chemistry	6	Engineering	Edit | Details | Delete
2021	Composition	6	English	Edit | Details | Delete
2042	Literature	8	English	Edit | Details | Delete
3141	Trigonometry	8	Mathematics	Edit | Details | Delete
4022	Microeconomics	6	Economics	Edit | Details | Delete
4041	Macroeconomics	6	Economics	Edit | Details | Delete

© 2014 - Contoso University

For more information about raw SQL queries, see Raw SQL Queries on MSDN.

No-Tracking Queries

When a database context retrieves table rows and creates entity objects that represent them, by default it keeps track of whether the entities in memory are in sync with what's in the database. The data in memory acts as a cache and is used when you update an entity. This caching is often unnecessary in a web application because context instances are typically short-lived (a new one is created and disposed for each request) and the context that reads an entity is typically disposed before that entity is used again.

You can disable tracking of entity objects in memory by using the AsNoTracking method. Typical scenarios in which you might want to do that include the following:

- A query retrieves such a large volume of data that turning off tracking might noticeably enhance performance.
- You want to attach an entity in order to update it, but you earlier retrieved the same entity for a different purpose. Because the entity is already being tracked by the database context, you can't attach the entity that you want to change. One way to handle this situation is to use the `AsNoTracking` option with the earlier query.

In this section you'll implement business logic that illustrates the second of these scenarios. Specifically, you'll enforce a business rule that says that an instructor can't be the administrator of more than one department. (Depending on what you've done with the Departments page so far, you might already have some departments that have the same administrator. In a production application you would apply a new rule to existing data also, but for this tutorial that isn't necessary.)

In *DepartmentController.cs*, add a new method that you can call from the `Edit` and `Create` methods to make sure that no two departments have the same administrator:

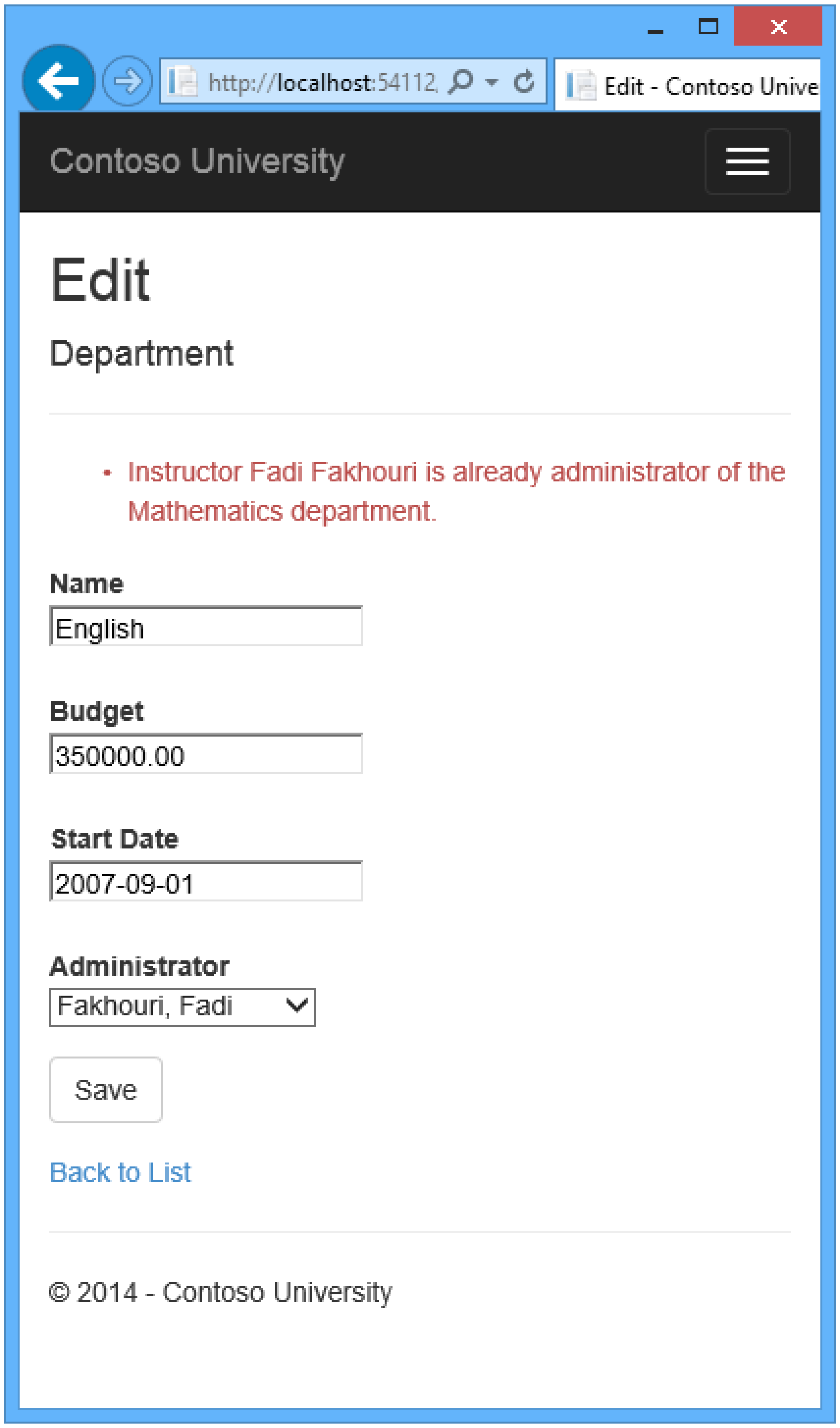
```
private void ValidateOneAdministratorAssignmentPerInstructor(Department 
department)
{
    if (department.InstructorID != null)
    {
        Department duplicateDepartment = db.Departments
            .Include("Administrator")
            .Where(d => d.InstructorID == department.InstructorID)
            .FirstOrDefault();
        if (duplicateDepartment != null && duplicateDepartment.DepartmentID 
!= department.DepartmentID)
        {
            string errorMessage = String.Format(
                "Instructor {0} {1} is already administrator of the {2} 
department.",
                duplicateDepartment.Administrator.FirstMidName,
                duplicateDepartment.Administrator.LastName,
                duplicateDepartment.Name);
            ModelState.AddModelError(string.Empty, errorMessage);
        }
    }
}
```

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

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(
   [Bind(Include = "DepartmentID, Name, Budget, StartDate, RowVersion, 
PersonID")]
    Department department)
{
   try
   {
      if (ModelState.IsValid)
      {
         ValidateOneAdministratorAssignmentPerInstructor(department);
      }

      if (ModelState.IsValid)
      {
         db.Entry(department).State = EntityState.Modified;
         db.SaveChanges();
         return RedirectToAction("Index");
      }
   }
   catch (DbUpdateConcurrencyException ex)
   {
      var entry = ex.Entries.Single();
      var clientValues = (Department)entry.Entity;
```

Run the Department Edit page and try to change a department's administrator to an instructor who is already the administrator of a different department. You get the expected error message:



Edit

Department

- Instructor Fadi Fakhouri is already administrator of the Mathematics department.

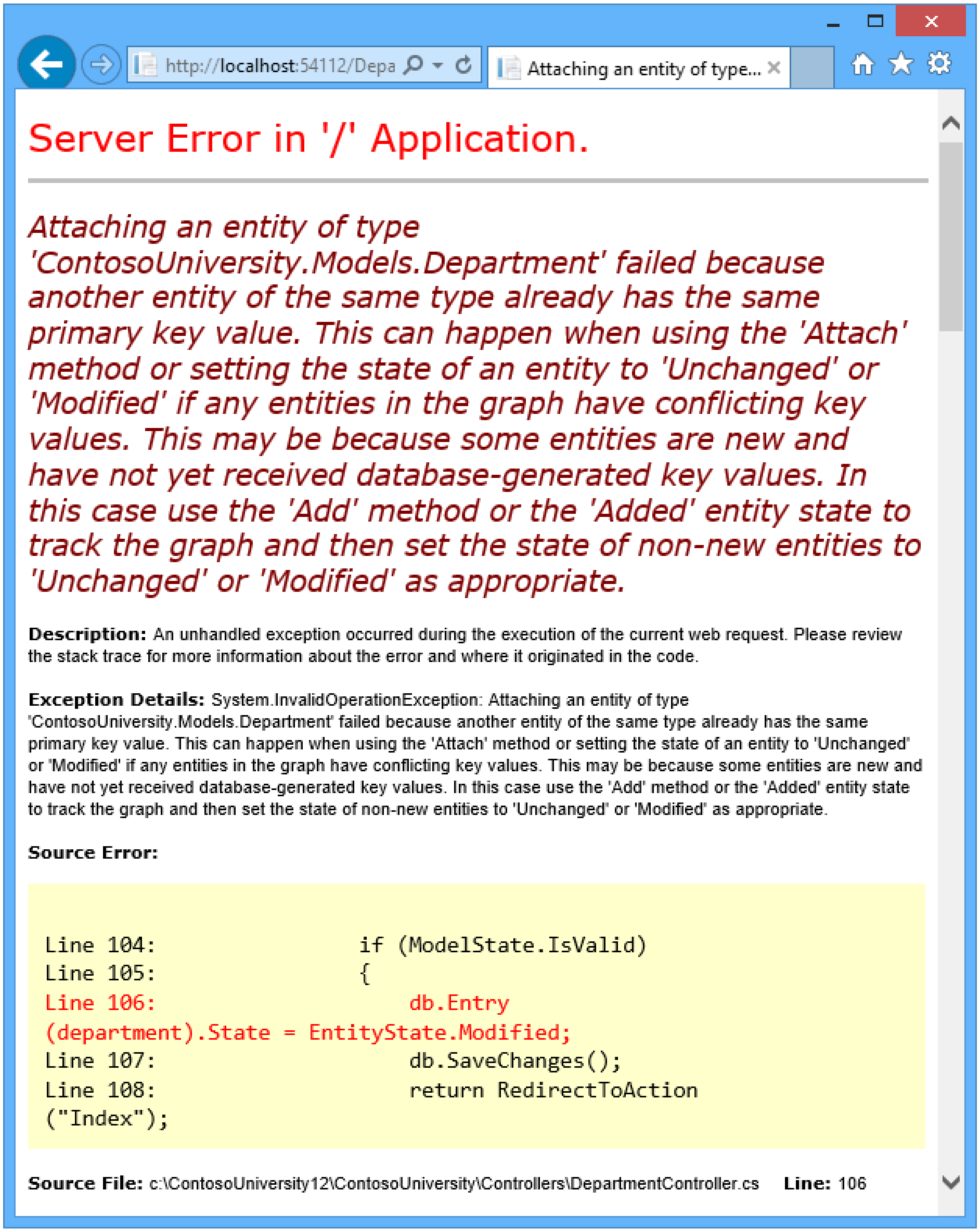
Name: English

Budget: 350000.00

Start Date: 2007-09-01

Administrator: Fakhouri, Fadi

Now run the Department Edit page again and this time change the **Budget** amount. When you click **Save**, you see an error page that results from the code you added in `ValidateOneAdministratorAssignmentPerInstructor`:



Server Error in '/' Application.

Attaching an entity of type 'ContosoUniversity.Models.Department' failed because another entity of the same type already has the same primary key value. This can happen when using the 'Attach' method or setting the state of an entity to 'Unchanged' or 'Modified' if any entities in the graph have conflicting key values. This may be because some entities are new and have not yet received database-generated key values. In this case use the 'Add' method or the 'Added' entity state to track the graph and then set the state of non-new entities to 'Unchanged' or 'Modified' as appropriate.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.InvalidOperationException: Attaching an entity of type 'ContosoUniversity.Models.Department' failed because another entity of the same type already has the same primary key value. This can happen when using the 'Attach' method or setting the state of an entity to 'Unchanged' or 'Modified' if any entities in the graph have conflicting key values. This may be because some entities are new and have not yet received database-generated key values. In this case use the 'Add' method or the 'Added' entity state to track the graph and then set the state of non-new entities to 'Unchanged' or 'Modified' as appropriate.

Source Error:

```
Line 104:                if (ModelState.IsValid)
Line 105:                {
Line 106:                    db.Entry
(department).State = EntityState.Modified;
Line 107:                    db.SaveChanges();
Line 108:                    return RedirectToAction
("Index");
```

Source File: c:\ContosoUniversity12\ContosoUniversity\Controllers\DepartmentController.cs   **Line:** 106

The exception error message is:

Attaching an entity of type 'ContosoUniversity.Models.Department' failed because another entity of the same type already has the same primary key value. This can happen when using the 'Attach' method or setting the state of an entity to 'Unchanged' or 'Modified' if any entities in the graph have conflicting key values. This may be because some entities are new and have not yet received database-generated key values. In this case use the 'Add' method or the 'Added' entity state to track the graph and then set the state of non-new entities to 'Unchanged' or 'Modified' as appropriate.

This happened because of the following sequence of events:

- The `Edit` method calls the `ValidateOneAdministratorAssignmentPerInstructor` method, which retrieves all departments that have Kim Abercrombie as their administrator. That causes the English department to be read. As a result of this read operation, the English department entity that was read from the database is now being tracked by the database context.
- The `Edit` method tries to set the `Modified` flag on the English department entity created by the MVC model binder, which implicitly causes the context to try to attach that entity. But the context can't attach the entry created by the model binder because the context is already tracking an entity for the English department.

One solution to this problem is to keep the context from tracking in-memory department entities retrieved by the validation query. There's no disadvantage to doing this, because you won't be updating this entity or reading it again in a way that would benefit from it being cached in memory.

In *DepartmentController.cs*, in the `ValidateOneAdministratorAssignmentPerInstructor` method, specify no tracking, as shown in the following:

```
Department duplicateDepartment = db.Departments
   .Include("Administrator")
   .Where(d => d.PersonID == department.PersonID)
   .AsNoTracking()
   .FirstOrDefault();
```

Repeat your attempt to edit the **Budget** amount of a department. This time the operation is successful, and the site returns as expected to the Departments Index page, showing the revised budget value.

Examining SQL sent to the database

Sometimes it's helpful to be able to see the actual SQL queries that are sent to the database. In an earlier tutorial you saw how to do that in interceptor code; now you'll see some ways to do it without writing interceptor code. To try this out, you'll look at a simple query and then look at what happens to it as you add options such eager loading, filtering, and sorting.

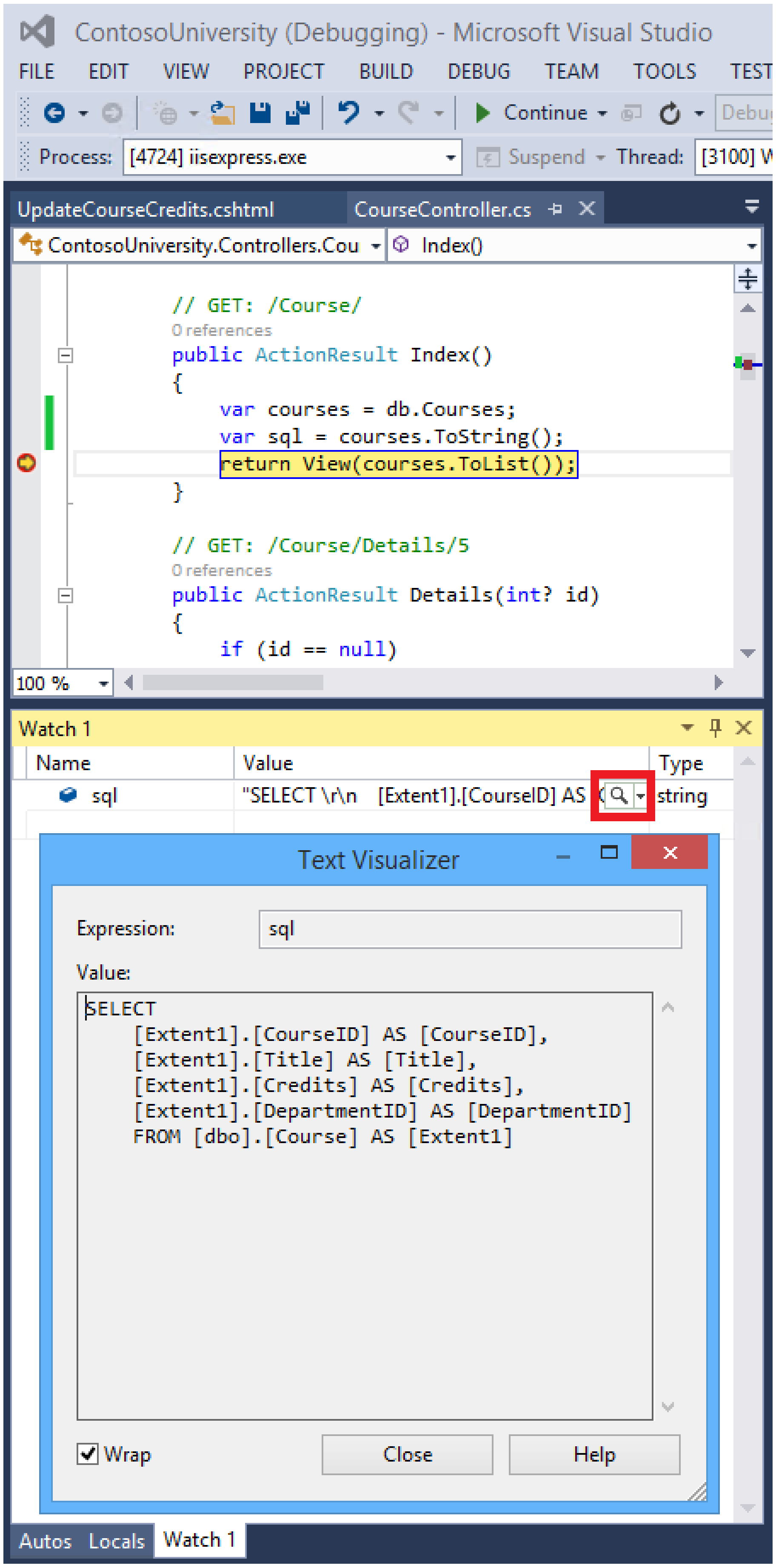
In *Controllers/CourseController*, replace the `Index` method with the following code, in order to temporarily stop eager loading:

```
public ActionResult Index()
{
    var courses = db.Courses;
    var sql = courses.ToString();
    return View(courses.ToList());
}
```

Now set a breakpoint on the `return` statement (F9 with the cursor on that line). Press F5 to run the project in debug mode, and select the Course Index page. When the code reaches the breakpoint, examine the `query` variable. You see the query that's sent to SQL Server. It's a simple `Select` statement.

```
{SELECT
[Extent1].[CourseID] AS [CourseID],
[Extent1].[Title] AS [Title],
[Extent1].[Credits] AS [Credits],
[Extent1].[DepartmentID] AS [DepartmentID]
FROM [Course] AS [Extent1]}
```

Click the magnifying class to see the query in the **Text Visualizer**.


ContosoUniversity (Debugging) - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST
Continue
Process: [4724] iisexpress.exe
Suspend
Thread: [3100]
UpdateCourseCredits.cshtml
CourseController.cs
ContosoUniversity.Controllers.Cou
Index()
100 %
Watch 1
Text Visualizer
Wrap
Close
Help
Autos Locals Watch 1

```
        // GET: /Course/
        0 references
        public ActionResult Index()
        {
            var courses = db.Courses;
            var sql = courses.ToString();
            return View(courses.ToList());
        }

        // GET: /Course/Details/5
        0 references
        public ActionResult Details(int? id)
        {
            if (id == null)
```

Name	Value	Type
sql	"SELECT \r\n [Extent1].[CourseID] AS (	string

Expression: sql

Value:

```
SELECT
    [Extent1].[CourseID] AS [CourseID],
    [Extent1].[Title] AS [Title],
    [Extent1].[Credits] AS [Credits],
    [Extent1].[DepartmentID] AS [DepartmentID]
    FROM [dbo].[Course] AS [Extent1]
```

Now you'll add a drop-down list to the Courses Index page so that users can filter for a particular department. You'll sort the courses by title, and you'll specify eager loading for the `Department` navigation property.

In *CourseController.cs*, replace the `Index` method with the following code:

```
public ActionResult Index(int? SelectedDepartment)
{
    var departments = db.Departments.OrderBy(q => q.Name).ToList();
    ViewBag.SelectedDepartment = new SelectList(departments, "DepartmentID",
"Name", SelectedDepartment);
    int departmentID = SelectedDepartment.GetValueOrDefault();

    IQueryable<Course> courses = db.Courses
        .Where(c => !SelectedDepartment.HasValue || c.DepartmentID ==
departmentID)
        .OrderBy(d => d.CourseID)
        .Include(d => d.Department);
    var sql = courses.ToString();
    return View(courses.ToList());
}
```

Restore the breakpoint on the `return` statement.

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

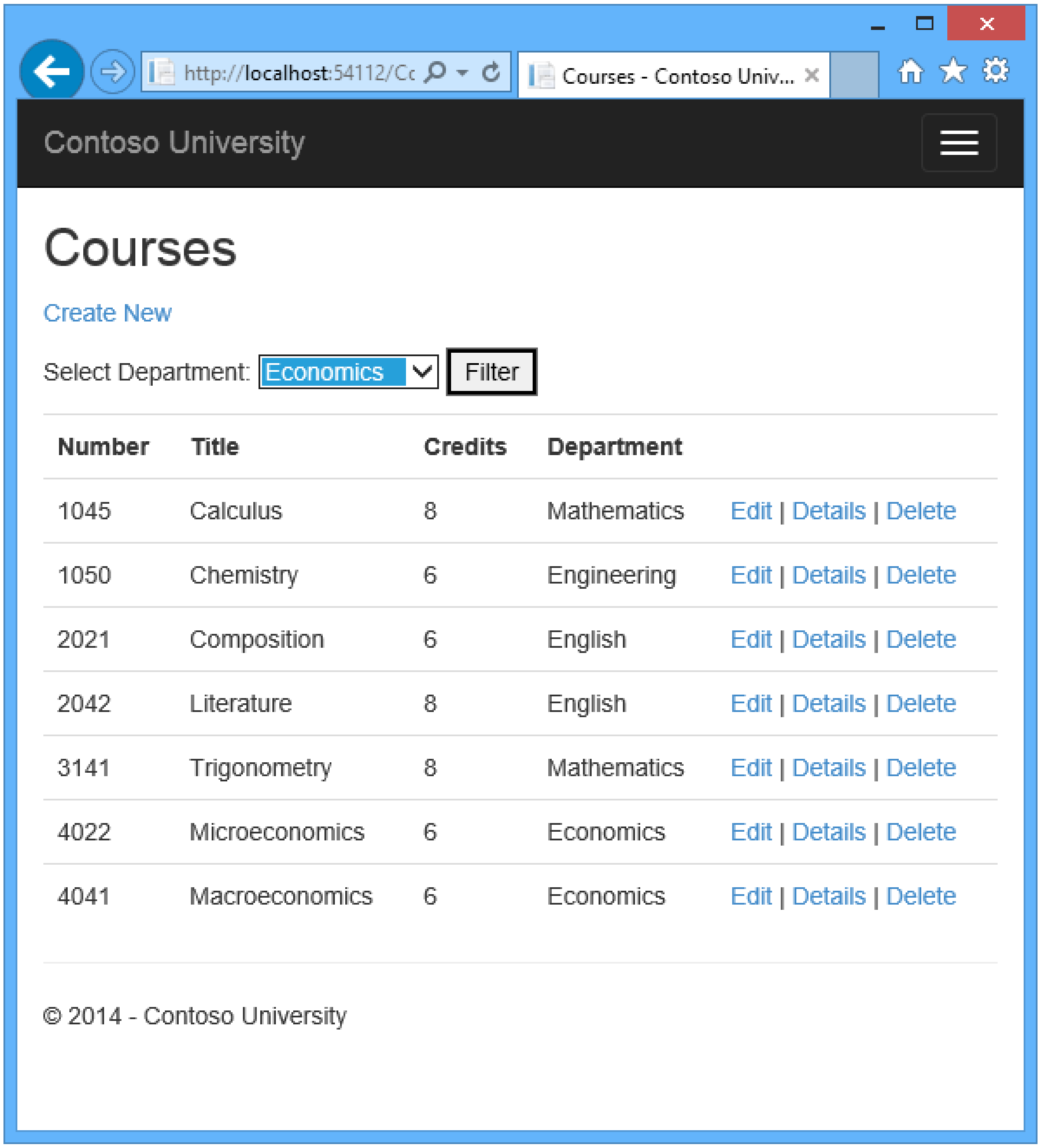
A `SelectList` collection containing all departments is passed to the view for the drop-down list. The parameters passed to the `SelectList` constructor specify the value field name, the text field name, and the selected item.

For the `Get` method of the `Course` repository, the code specifies a filter expression, a sort order, and eager loading for the `Department` navigation property. The filter expression always returns `true` if nothing is selected in the drop-down list (that is, `SelectedDepartment` is null).

In *Views\Course\Index.cshtml*, immediately before the opening `table` tag, add the following code to create the drop-down list and a submit button:

```
@using (Html.BeginForm())
{
    <p>Select Department: @Html.DropDownList("SelectedDepartment","All")
    <input type="submit" value="Filter" /></p>
}
```

With the breakpoint still set, run the Course Index page. Continue through the first times that the code hits a breakpoint, so that the page is displayed in the browser. Select a department from the drop-down list and click **Filter**:



Contoso University

Courses

Create New

Select Department: Economics [Filter]

Number	Title	Credits	Department	
1045	Calculus	8	Mathematics	Edit | Details | Delete
1050	Chemistry	6	Engineering	Edit | Details | Delete
2021	Composition	6	English	Edit | Details | Delete
2042	Literature	8	English	Edit | Details | Delete
3141	Trigonometry	8	Mathematics	Edit | Details | Delete
4022	Microeconomics	6	Economics	Edit | Details | Delete
4041	Macroeconomics	6	Economics	Edit | Details | Delete

© 2014 - Contoso University

This time the first breakpoint will be for the departments query for the drop-down list. Skip that and view the `query` variable the next time the code reaches the breakpoint in order to see what the `Course` query now looks like. You'll see something like the following:

```
SELECT
    [Project1].[CourseID] AS [CourseID],
    [Project1].[Title] AS [Title],
    [Project1].[Credits] AS [Credits],
    [Project1].[DepartmentID] AS [DepartmentID],
    [Project1].[DepartmentID1] AS [DepartmentID1],
    [Project1].[Name] AS [Name],
    [Project1].[Budget] AS [Budget],
    [Project1].[StartDate] AS [StartDate],
    [Project1].[InstructorID] AS [InstructorID],
    [Project1].[RowVersion] AS [RowVersion]
    FROM ( SELECT
        [Extent1].[CourseID] AS [CourseID],
        [Extent1].[Title] AS [Title],
        [Extent1].[Credits] AS [Credits],
        [Extent1].[DepartmentID] AS [DepartmentID],
        [Extent2].[DepartmentID] AS [DepartmentID1],
        [Extent2].[Name] AS [Name],
        [Extent2].[Budget] AS [Budget],
        [Extent2].[StartDate] AS [StartDate],
        [Extent2].[InstructorID] AS [InstructorID],
        [Extent2].[RowVersion] AS [RowVersion]
        FROM  [dbo].[Course] AS [Extent1]
        INNER JOIN [dbo].[Department] AS [Extent2] ON
[Extent1].[DepartmentID] = [Extent2].[DepartmentID]
        WHERE @p__linq__0 IS NULL OR [Extent1].[DepartmentID] = @p__linq__1
    )  AS [Project1]
    ORDER BY [Project1].[CourseID] ASC
```

You can see that the query is now a `JOIN` query that loads `Department` data along with the `Course` data, and that it includes a `WHERE` clause.

Remove the `var sql = courses.ToString()` line.

Repository and unit of work patterns

Many developers write code to implement the repository and unit of work patterns as a wrapper around code that works with the Entity Framework. These patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD). However, writing additional code to implement these patterns is not always the best choice for applications that use EF, for several reasons:

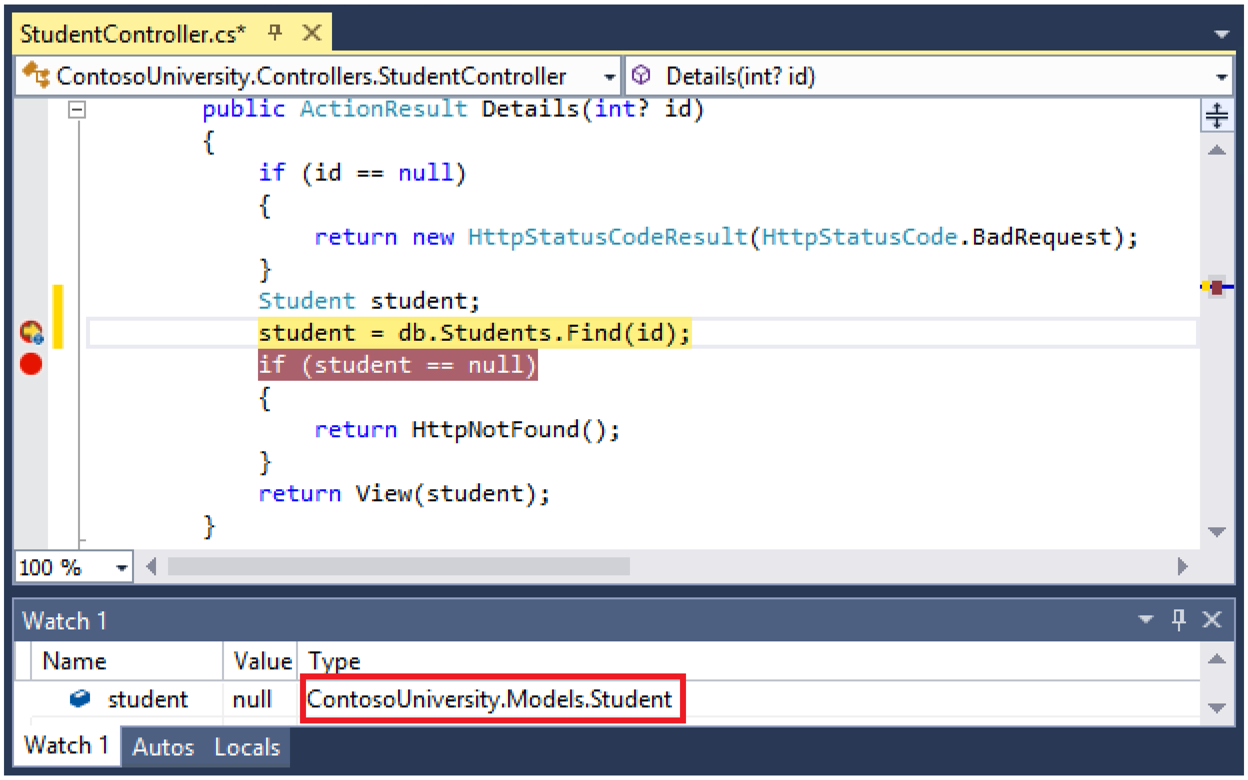
- The EF context class itself insulates your code from data-store-specific code.
- The EF context class can act as a unit-of-work class for database updates that you do using EF.
- Features introduced in Entity Framework 6 make it easier to implement TDD without writing repository code.

For more information about how to implement the repository and unit of work patterns, see the Entity Framework 5 version of this tutorial series. For information about ways to implement TDD in Entity Framework 6, see the following resources:

- How EF6 Enables Mocking DbSets more easily
- Testing with a mocking framework
- Testing with your own test doubles

Proxy classes

When the Entity Framework creates entity instances (for example, when you execute a query), it often creates them as instances of a dynamically generated derived type that acts as a proxy for the entity. For example, see the following two debugger images. In the first image, you see that the `student` variable is the expected `Student` type immediately after you instantiate the entity. In the second image, after EF has been used to read a student entity from the database, you see the proxy class.



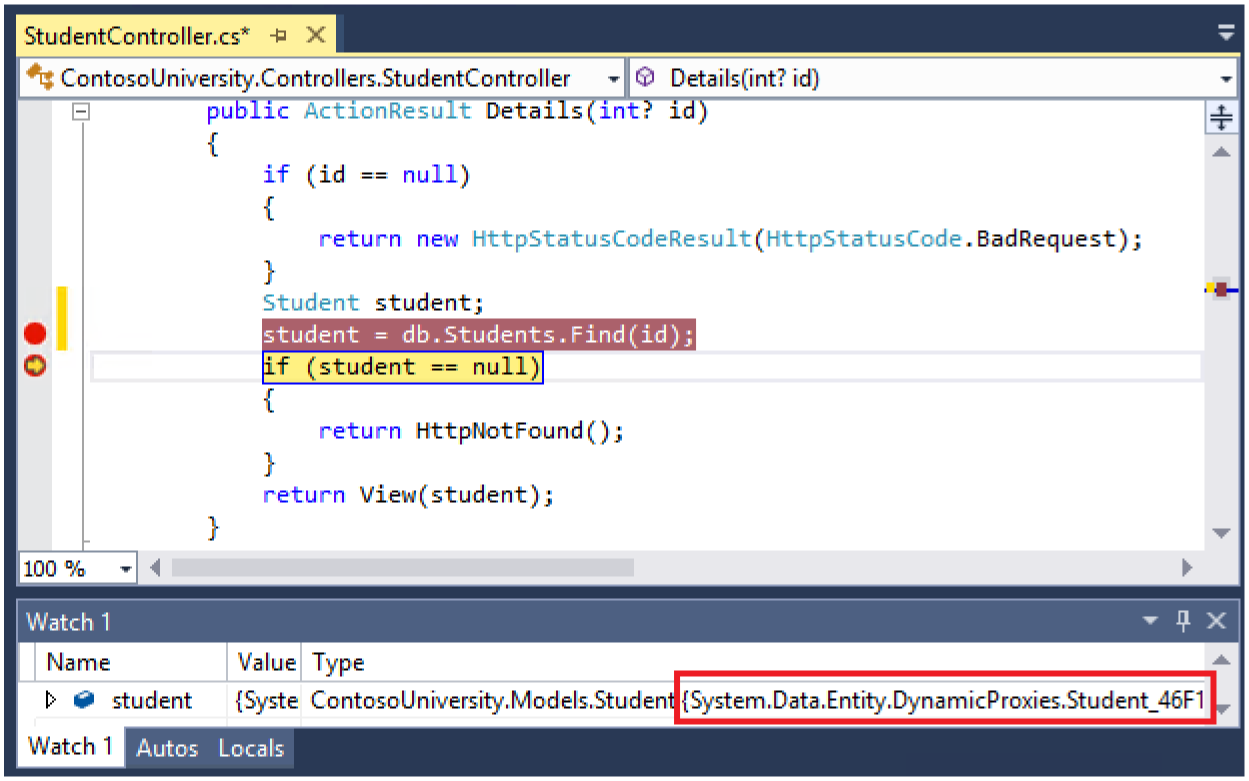
StudentController.cs*

ContosoUniversity.Controllers.StudentController | Details(int? id)

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

Watch 1

Name	Value	Type
student	null	ContosoUniversity.Models.Student



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

```
Name       Value   Type
student    {Syste  ContosoUniversity.Models.Student {System.Data.Entity.DynamicProxies.Student_46F1
```

This proxy class overrides some virtual properties of the entity to insert hooks for performing actions automatically when the property is accessed. One function this mechanism is used for is lazy loading.

Most of the time you don't need to be aware of this use of proxies, but there are exceptions:

- In some scenarios you might want to prevent the Entity Framework from creating proxy instances. For example, when you're serializing entities you generally want the POCO classes, not the proxy classes. One way to avoid serialization problems is to serialize data transfer objects (DTOs) instead of entity objects, as shown in the Using Web API with Entity Framework tutorial. Another way is to disable proxy creation.
- When you instantiate an entity class using the `new` operator, you don't get a proxy instance. This means you don't get functionality such as lazy loading and automatic change tracking. This is typically okay; you generally don't need lazy loading, because you're creating a new entity that isn't in the database, and you generally don't need change tracking if you're explicitly marking the entity as `Added`. However, if you do need lazy loading and you need change tracking, you can create new entity instances with proxies using the Create method of the `DbSet` class.
- You might want to get an actual entity type from a proxy type. You can use the GetObjectType method of the `ObjectContext` class to get the actual entity type of a proxy type instance.

For more information, see Working with Proxies on MSDN.

Automatic change detection

The Entity Framework determines how an entity has changed (and therefore which updates need to be sent to the database) by comparing the current values of an entity with the original values. The original values are stored when the entity is queried or attached. Some of the methods that cause automatic change detection are the following:

- `DbSet.Find`
- `DbSet.Local`
- `DbSet.Remove`
- `DbSet.Add`
- `DbSet.Attach`
- `DbContext.SaveChanges`
- `DbContext.GetValidationErrors`
- `DbContext.Entry`
- `DbChangeTracker.Entries`

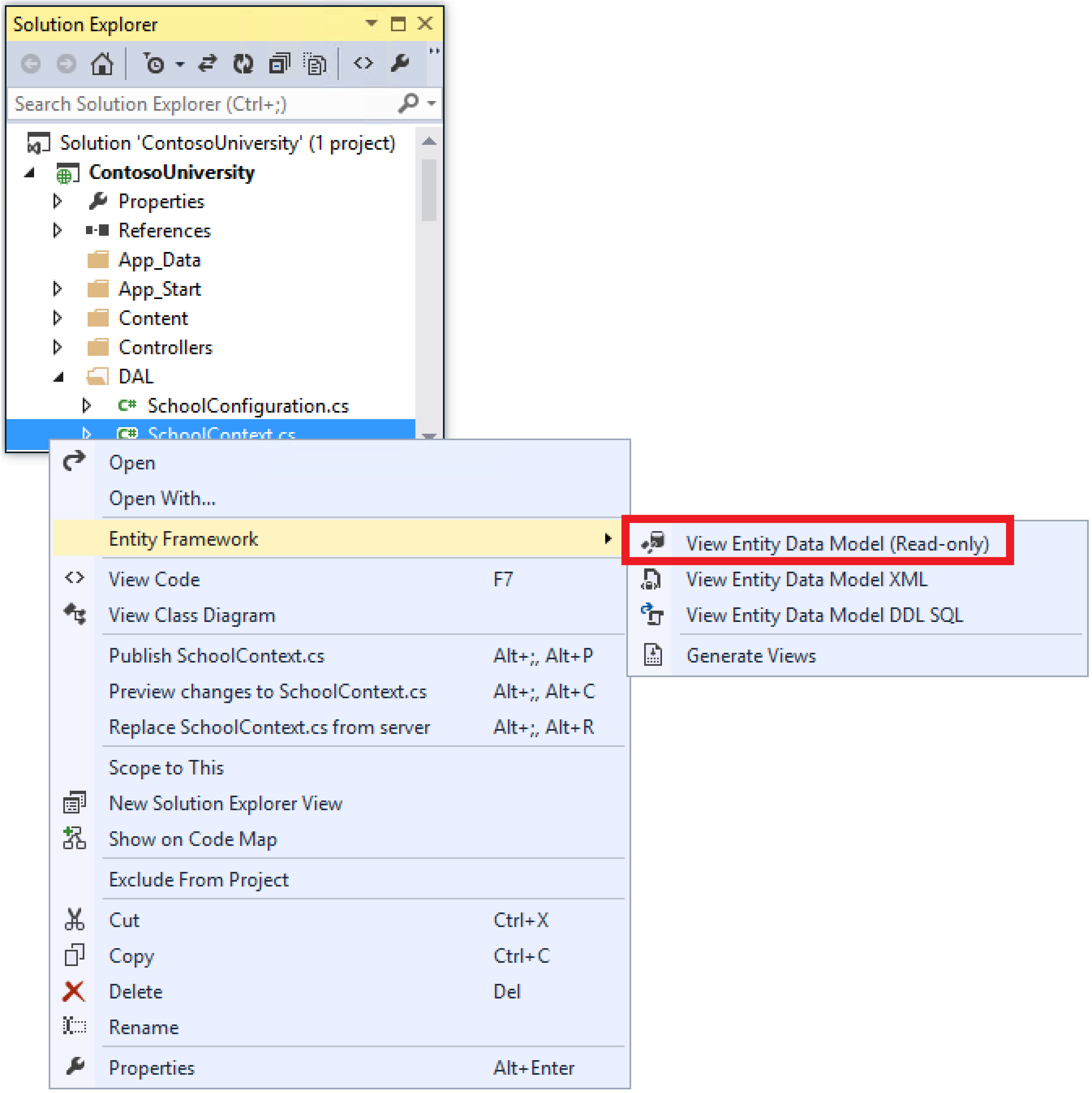
If you're tracking a large number of entities and you call one of these methods many times in a loop, you might get significant performance improvements by temporarily turning off automatic change detection using the AutoDetectChangesEnabled property. For more information, see Automatically Detecting Changes on MSDN.

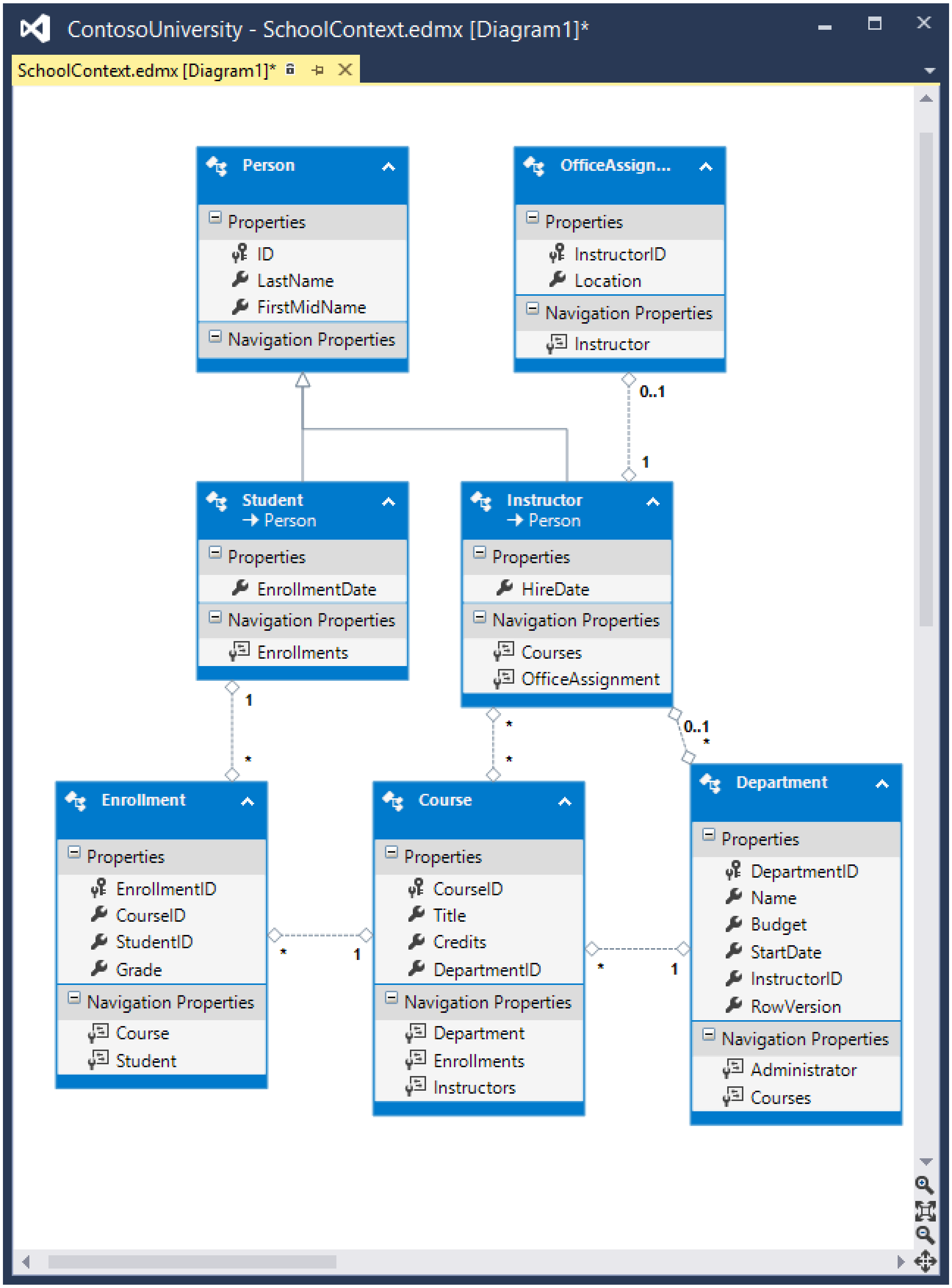
Automatic validation

When you call the `SaveChanges` method, by default the Entity Framework validates the data in all properties of all changed entities before updating the database. If you've updated a large number of entities and you've already validated the data, this work is unnecessary and you could make the process of saving the changes take less time by temporarily turning off validation. You can do that using the ValidateOnSaveEnabled property. For more information, see Validation on MSDN.

Entity Framework Power Tools

Entity Framework Power Tools is a Visual Studio add-in that was used to create the data model diagrams shown in these tutorials. The tools can also do other function such as generate entity classes based on the tables in an existing database so that you can use the database with Code First. After you install the tools, some additional options appear in context menus. For example, when you right-click your context class in **Solution Explorer**, you get an option to generate a diagram. When you're using Code First you can't change the data model in the diagram, but you can move things around to make it easier to understand.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ContosoUniversity' (1 project)
ContosoUniversity
Properties
References
App_Data
App_Start
Content
Controllers
DAL
SchoolConfiguration.cs
SchoolContext.cs
Open
Open With...
Entity Framework
View Code F7
View Class Diagram
Publish SchoolContext.cs Alt+;, Alt+P
Preview changes to SchoolContext.cs Alt+;, Alt+C
Replace SchoolContext.cs from server Alt+;, Alt+R
Scope to This
New Solution Explorer View
Show on Code Map
Exclude From Project
Cut Ctrl+X
Copy Ctrl+C
Delete Del
Rename
Properties Alt+Enter
View Entity Data Model (Read-only)
View Entity Data Model XML
View Entity Data Model DDL SQL
Generate Views


ContosoUniversity - SchoolContext.edmx [Diagram1]*
SchoolContext.edmx [Diagram1]*
Person
Properties
ID
LastName
FirstMidName
Navigation Properties
OfficeAssign...
Properties
InstructorID
Location
Navigation Properties
Instructor
0..1
1
Student → Person
Properties
EnrollmentDate
Navigation Properties
Enrollments
Instructor → Person
Properties
HireDate
Navigation Properties
Courses
OfficeAssignment
1
*
*
*
0..1
*
Enrollment
Properties
EnrollmentID
CourseID
StudentID
Grade
Navigation Properties
Course
Student
Course
Properties
CourseID
Title
Credits
DepartmentID
Navigation Properties
Department
Enrollments
Instructors
Department
Properties
DepartmentID
Name
Budget
StartDate
InstructorID
RowVersion
Navigation Properties
Administrator
Courses
*
1
*
1

Entity Framework source code

The source code for Entity Framework 6 is available at http://entityframework.codeplex.com/. Besides source code, you can get nightly builds, issue tracking, feature specs, design meeting notes, and more. You can file bugs, and you can contribute your own enhancements to the EF source code.

Although the source code is open, Entity Framework is fully supported as a Microsoft product. The Microsoft Entity Framework team keeps control over which contributions are accepted and tests all code changes to ensure the quality of each release.

Summary

This completes this series of tutorials on using the Entity Framework in an ASP.NET MVC application. For more information about how to work with data using the Entity Framework, see the EF documentation page on MSDN and ASP.NET Data Access - Recommended Resources.

For more information about how to deploy your web application after you've built it, see ASP.NET Web Deployment - Recommended Resources in the MSDN Library.

For information about other topics related to MVC, such as authentication and authorization, see the ASP.NET MVC - Recommended Resources.

Acknowledgments

- Tom Dykstra wrote the original version of this tutorial, co-authored the EF 5 update, and wrote the EF 6 update. Tom is a senior programming writer on the Microsoft Web Platform and Tools Content Team.
- Rick Anderson (twitter @RickAndMSFT) did most of the work updating the tutorial for EF 5 and MVC 4 and co-authored the EF 6 update. Rick is a senior programming writer for Microsoft focusing on Azure and MVC.
- Rowan Miller and other members of the Entity Framework team assisted with code reviews and helped debug many issues with migrations that arose while we were updating the tutorial for EF 5 and EF 6.

VB

When the tutorial was originally produced for EF 4.1, we provided both C# and VB versions of the completed download project. Due to time limitations and other priorities we have not done that for this version. If you build a VB project using these tutorials and would be willing to share that with others, please let us know.

Common errors, and solutions or workarounds for them

Cannot create/shadow copy

Error Message:

Cannot create/shadow copy '<filename>' when that file already exists.

Solution

Wait a few seconds and refresh the page.

Update-Database not recognized

Error Message (from the `Update-Database` command in the PMC):

The term 'Update-Database' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again.

Solution

Exit Visual Studio. Reopen project and try again.

Validation failed

Error Message (from the `Update-Database` command in the PMC):

Validation failed for one or more entities. See 'EntityValidationErrors' property for more details.

Solution

One cause of this problem is validation errors when the `Seed` method runs. See Seeding and Debugging Entity Framework (EF) DBs for tips on debugging the `Seed` method.

HTTP 500.19 error

Error Message:

HTTP Error 500.19 - Internal Server Error
The requested page cannot be accessed because the related configuration data for the page is invalid.

Solution

One way you can get this error is from having multiple copies of the solution, each of them using the same port number. You can usually solve this problem by exiting all instances of Visual Studio, then restarting the project you're working on. If that doesn't work, try changing the port number. Right click on the project file and then click properties. Select the **Web** tab and then change the port number in the **Project Url** text box.

Error locating SQL Server instance

Error Message:

A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify that the instance name is correct and that SQL Server is configured to allow remote connections. (provider: SQL Network Interfaces, error: 26 - Error Locating Server/Instance Specified)

Solution

Check the connection string. If you have manually deleted the database, change the name of the database in the construction string.