

Getting Started with Entity Framework 6 Code First using MVC 5

Tom Dykstra, Rick Anderson

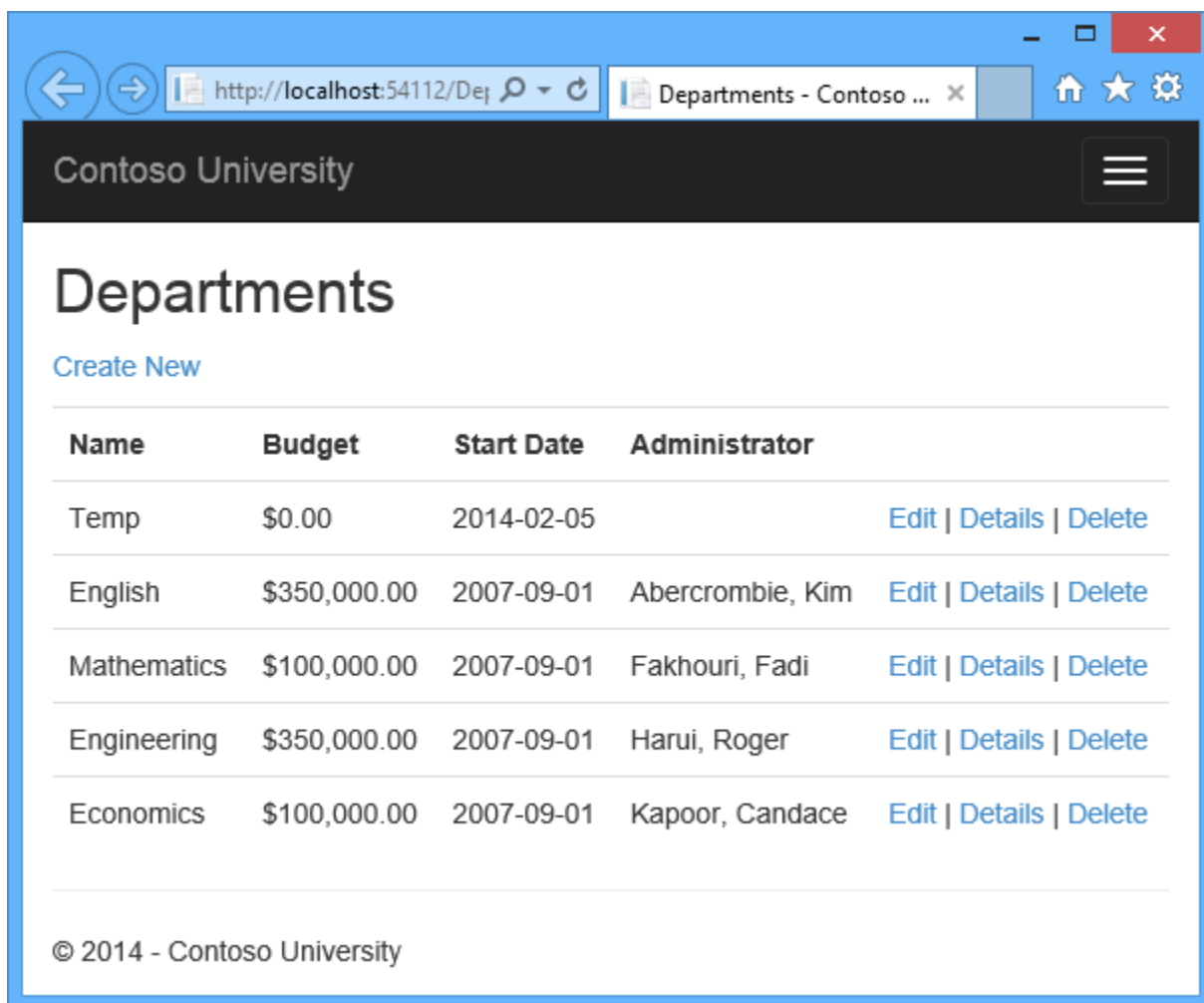
Step By Step, Guide



Handling Concurrency with the Entity Framework 6 in an ASP.NET MVC 5 Application (10 of 12)

In earlier tutorials you learned how to update data. This tutorial shows how to handle conflicts when multiple users update the same entity at the same time.

You'll change the web pages that work with the `Department` entity so that they handle concurrency errors. The following illustrations show the Index and Delete pages, including some messages that are displayed if a concurrency conflict occurs.



The screenshot shows a web browser window displaying the 'Departments' page of the Contoso University application. The browser's address bar shows the URL `http://localhost:54112/De`. The page has a dark header with the 'Contoso University' logo and a hamburger menu icon. Below the header, the page title 'Departments' is displayed, followed by a 'Create New' link. A table lists the departments with columns for Name, Budget, Start Date, and Administrator. Each row includes links for 'Edit', 'Details', and 'Delete'. The footer of the page shows the copyright notice '© 2014 - Contoso University'.

Name	Budget	Start Date	Administrator	
Temp	\$0.00	2014-02-05		Edit Details Delete
English	\$350,000.00	2007-09-01	Abercrombie, Kim	Edit Details Delete
Mathematics	\$100,000.00	2007-09-01	Fakhouri, Fadi	Edit Details Delete
Engineering	\$350,000.00	2007-09-01	Harui, Roger	Edit Details Delete
Economics	\$100,000.00	2007-09-01	Kapoor, Candace	Edit Details Delete

© 2014 - Contoso University

Contoso University

Edit

Department

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

Name

Budget
 Current value: \$0.00

Start Date
 Current value: 9/1/2007

Administrator

[Back to List](#)

Concurrency Conflicts

A concurrency conflict occurs when one user displays an entity's data in order to edit it, and then another user updates the same entity's data before the first user's change is written to the database. If you don't enable the detection of such conflicts, whoever updates the database last overwrites the other user's changes. In many applications, this risk is acceptable: if there are few users, or few updates, or if it isn't really critical if some changes are overwritten, the cost of programming for concurrency might outweigh the benefit. In that case, you don't have to configure the application to handle concurrency conflicts.

Pessimistic Concurrency (Locking)

If your application does need to prevent accidental data loss in concurrency scenarios, one way to do that is to use database locks. This is called *pessimistic concurrency*. For example, before you read a row from a database, you request a lock for read-only or for update access. If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access, because they would get a copy of data that's in the process of being changed. If you lock a row for read-only access, others can also lock it for read-only access but not for update.

Managing locks has disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases. For these reasons, not all database management systems support pessimistic concurrency. The Entity Framework provides no built-in support for it, and this tutorial doesn't show you how to implement it.

Optimistic Concurrency

The alternative to pessimistic concurrency is *optimistic concurrency*. Optimistic concurrency means allowing concurrency conflicts to happen, and then reacting appropriately if they do. For example, John runs the Departments Edit page, changes the **Budget** amount for the English department from \$350,000.00 to \$0.00.

The screenshot shows a web browser window with the URL `http://localhost:54112/Dej`. The page is titled "Contoso University" and "Departments". Below the title is a "Create New" link. A table lists five departments: Temp, English, Mathematics, Engineering, and Economics. The "English" department's budget of "\$0.00" is highlighted with a red rectangle. Each row has links for "Edit", "Details", and "Delete".

Name	Budget	Start Date	Administrator	
Temp	\$0.00	2014-02-05		Edit Details Delete
English	\$0.00	2007-09-01	Abercrombie, Kim	Edit Details Delete
Mathematics	\$100,000.00	2007-09-01	Fakhouri, Fadi	Edit Details Delete
Engineering	\$350,000.00	2007-09-01	Harui, Roger	Edit Details Delete
Economics	\$100,000.00	2007-09-01	Kapoor, Candace	Edit Details Delete

© 2014 - Contoso University

Before John clicks **Save**, Jane runs the same page and changes the **Start Date** field from 9/1/2007 to 8/8/2013.

Contoso University

Edit

Department

Name

Budget

Start Date

Administrator

[Back to List](#)

© 2014 - Contoso University

John clicks **Save** first and sees his change when the browser returns to the Index page, then Jane clicks **Save**. What happens next is determined by how you handle concurrency conflicts. Some of the options include the following:

- You can keep track of which property a user has modified and update only the corresponding columns in the database. In the example scenario, no data would be lost, because different properties were updated by the two users. The next time someone browses the English department, they'll see both John's and Jane's changes — a start date of 8/8/2013 and a budget of Zero dollars.

This method of updating can reduce the number of conflicts that could result in data loss, but it can't avoid data loss if competing changes are made to the same property of an entity. Whether the Entity Framework works this way depends on how you implement your update code. It's often not practical in a web application, because it can require that you maintain large amounts of state in order to keep track of all original property values for an entity as well as new values. Maintaining large amounts of state can affect application performance because it either requires server resources or must be included in the web page itself (for example, in hidden fields) or in a cookie.

- You can let Jane's change overwrite John's change. The next time someone browses the English department, they'll see 8/8/2013 and the restored \$350,000.00 value. This is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) As noted in the introduction to this section, if you don't do any coding for concurrency handling, this will happen automatically.
- You can prevent Jane's change from being updated in the database. Typically, you would display an error message, show her the current state of the data, and allow her to reapply her changes if she still wants to make them. This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You'll implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted to what's happening.

Detecting Concurrency Conflicts

You can resolve conflicts by handling [OptimisticConcurrencyException](#) exceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. Therefore, you must configure the database and the data model appropriately. Some options for enabling conflict detection include the following:

- In the database table, include a tracking column that can be used to determine when a row has been changed. You can then configure the Entity Framework to include that column in the `Where` clause of `SQL Update` or `Delete` commands.

The data type of the tracking column is typically [rowversion](#). The [rowversion](#) value is a sequential number that's incremented each time the row is updated. In an `Update` or `Delete` command, the `Where` clause includes the original value of the tracking column (the original row version). If the row being updated has been changed by another user, the value in the [rowversion](#) column is different than the original value, so the `Update` or `Delete` statement can't find the row to update because of the `Where` clause. When the Entity Framework finds that no rows have been updated by the `Update` or `Delete` command (that is, when the number of affected rows is zero), it interprets that as a concurrency conflict.

- Configure the Entity Framework to include the original values of every column in the table in the `Where` clause of `Update` and `Delete` commands.

As in the first option, if anything in the row has changed since the row was first read, the `Where` clause won't return a row to update, which the Entity Framework interprets as a concurrency conflict. For database tables that have many columns, this approach can result in very large `Where` clauses, and can require that you maintain large amounts of state. As noted earlier, maintaining large amounts of state can affect application performance. Therefore this approach is generally not recommended, and it isn't the method used in this tutorial.

If you do want to implement this approach to concurrency, you have to mark all non-primary-key properties in the entity you want to track concurrency for by adding the [ConcurrencyCheck](#) attribute to them. That change enables the Entity Framework to include all columns in the SQL `WHERE` clause of `UPDATE` statements.

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

Add an Optimistic Concurrency Property to the Department Entity

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

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

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

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

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

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

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

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

The [Timestamp](#) attribute specifies that this column will be included in the `Where` clause of `Update` and `Delete` commands sent to the database. The attribute is called [Timestamp](#) because

previous versions of SQL Server used a SQL [timestamp](#) data type before the SQL [rowversion](#) replaced it. The .Net type for [rowversion](#) is a byte array.

If you prefer to use the fluent API, you can use the [IsConcurrencyToken](#) method to specify the tracking property, as shown in the following example:

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

By adding a property you changed the database model, so you need to do another migration. In the Package Manager Console (PMC), enter the following commands:

```
Add-Migration RowVersion  
Update-Database
```

Modify the Department Controller

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

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

In the *DepartmentController.cs* file, change all four occurrences of "LastName" to "FullName" so that the department administrator drop-down lists will contain the full name of the instructor rather than just the last name.

```
ViewBag.InstructorID = new SelectList(db.Instructors, "InstructorID",  
"FullName");
```

Replace the existing code for the `HttpPost Edit` method with the following code:

```
[HttpPost]  
[ValidateAntiForgeryToken]  
public async Task<ActionResult> Edit(  
    [Bind(Include = "DepartmentID, Name, Budget, StartDate, RowVersion,  
InstructorID")]  
    Department department)  
{  
    try  
    {  
        if (ModelState.IsValid)  
        {  
            db.Entry(department).State = EntityState.Modified;  
            await db.SaveChangesAsync();  
            return RedirectToAction("Index");  
        }  
    }  
    catch (DbUpdateConcurrencyException ex)  
    {  
        var entry = ex.Entries.Single();  
        var clientValues = (Department)entry.Entity;  
        var databaseEntry = entry.GetDatabaseValues();
```

```

        if (databaseEntry == null)
        {
            ModelState.AddModelError(string.Empty,
                "Unable to save changes. The department was deleted by
another user.");
        }
        else
        {
            var databaseValues = (Department)databaseEntry.ToObject();

            if (databaseValues.Name != clientValues.Name)
                ModelState.AddModelError("Name", "Current value: "
                    + databaseValues.Name);
            if (databaseValues.Budget != clientValues.Budget)
                ModelState.AddModelError("Budget", "Current value: "
                    + String.Format("{0:c}", databaseValues.Budget));
            if (databaseValues.StartDate != clientValues.StartDate)
                ModelState.AddModelError("StartDate", "Current value: "
                    + String.Format("{0:d}", databaseValues.StartDate));
            if (databaseValues.InstructorID != clientValues.InstructorID)
                ModelState.AddModelError("InstructorID", "Current value: "
                    +
db.Instructors.Find(databaseValues.InstructorID).FullName);
            ModelState.AddModelError(string.Empty, "The record you attempted
to edit "
                + "was modified by another user after you got the original
value. The "
                + "edit operation was canceled and the current values in the
database "
                + "have been displayed. If you still want to edit this
record, click "
                + "the Save button again. Otherwise click the Back to List
hyperlink.");
            department.RowVersion = databaseValues.RowVersion;
        }
    }
    catch (RetryLimitExceededException /* dex */)
    {
        //Log the error (uncomment dex variable name and add a line here to
write a log.
        ModelState.AddModelError(string.Empty, "Unable to save changes. Try
again, and if the problem persists contact your system administrator.");
    }

    ViewBag.InstructorID = new SelectList(db.Instructors, "ID", "FullName",
department.InstructorID);
    return View(department);
}

```

The view will store the original `RowVersion` value in a hidden field. When the model binder creates the `department` instance, that object will have the original `RowVersion` property value and the new values for the other properties, as entered by the user on the Edit page. Then when the Entity Framework creates a SQL `UPDATE` command, that command will include a `WHERE` clause that looks for a row that has the original `RowVersion` value.

If no rows are affected by the `UPDATE` command (no rows have the original `RowVersion` value), the Entity Framework throws a [DbUpdateConcurrencyException](#) exception, and the code in the `catch` block gets the affected `Department` entity from the exception object.

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

This object has the new values entered by the user in its `Entity` property, and you can get the values read from the database by calling the `GetDatabaseValues` method.

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

The `GetDatabaseValues` method returns `null` if someone has deleted the row from the database; otherwise, you have to cast the object returned to the `Department` class in order to access the `Department` properties.

```
if (databaseEntry == null)
{
    ModelState.AddModelError(string.Empty,
        "Unable to save changes. The department was deleted by another user.");
}
else
{
    var databaseValues = (Department)databaseEntry.ToObject();
```

Next, the code adds a custom error message for each column that has database values different from what the user entered on the Edit page:

```
if (databaseValues.Name != currentValues.Name)
    ModelState.AddModelError("Name", "Current value: " +
        databaseValues.Name);
// ...
```

A longer error message explains what happened and what to do about it:

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

Finally, the code sets the `RowVersion` value of the `Department` object to the new value retrieved from the database. This new `RowVersion` value will be stored in the hidden field when the Edit page is redisplayed, and the next time the user clicks **Save**, only concurrency errors that happen since the redisplay of the Edit page will be caught.

In `Views\Department\Edit.cshtml`, add a hidden field to save the `RowVersion` property value, immediately following the hidden field for the `DepartmentID` property:

```

@model ContosoUniversity.Models.Department

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Department</h4>
        <hr />
        @Html.ValidationSummary(true)
        @Html.HiddenFor(model => model.DepartmentID)
        @Html.HiddenFor(model => model.RowVersion)
    </div>
}

```

Testing Optimistic Concurrency Handling

Run the site and click **Departments**:

Contoso University

Departments

[Create New](#)

Name	Budget	Start Date	Administrator	
Temp	\$0.00	2014-02-05		Edit Details Delete
English	\$350,000.00	2007-09-01	Abercrombie, Kim	Edit Details Delete
Mathematics	\$100,000.00	2007-09-01	Fakhouri, Fadi	Edit Details Delete
Engineering	\$350,000.00	2007-09-01	Harui, Roger	Edit Details Delete
Economics	\$100,000.00	2007-09-01	Kapoor, Candace	Edit Details Delete

© 2014 - Contoso University

Right click the **Edit** hyperlink for the English department and select **Open in new tab**, then click the **Edit** hyperlink for the English department. The two tabs display the same information.

Contoso University

Edit

Department

Name

Budget

Start Date

Administrator

[Back to List](#)

© 2014 - Contoso University

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

Contoso University

Edit

Department

Name
English

Budget
0

Start Date
2007-09-01

Administrator
Abercrombie, Kim ▼

Save

[Back to List](#)

© 2014 - Contoso University

The browser shows the Index page with the changed value.

The screenshot shows a web browser window with the URL `http://localhost:54112/Departments`. The page title is "Contoso University" and the main heading is "Departments". There is a "Create New" link. Below is a table with columns: Name, Budget, Start Date, and Administrator. The "English" department row has its "Budget" value of "\$0.00" highlighted with a red box. Each row also has "Edit", "Details", and "Delete" links. The footer shows "© 2014 - Contoso University".

Name	Budget	Start Date	Administrator	
Temp	\$0.00	2014-02-05		Edit Details Delete
English	\$0.00	2007-09-01	Abercrombie, Kim	Edit Details Delete
Mathematics	\$100,000.00	2007-09-01	Fakhouri, Fadi	Edit Details Delete
Engineering	\$350,000.00	2007-09-01	Harui, Roger	Edit Details Delete
Economics	\$100,000.00	2007-09-01	Kapoor, Candace	Edit Details Delete

Change a field in the second browser tab and click **Save**.

Contoso University

Edit

Department

Name

Budget

Start Date

Administrator

[Back to List](#)

© 2014 - Contoso University

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

Contoso University

Edit

Department

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

Name

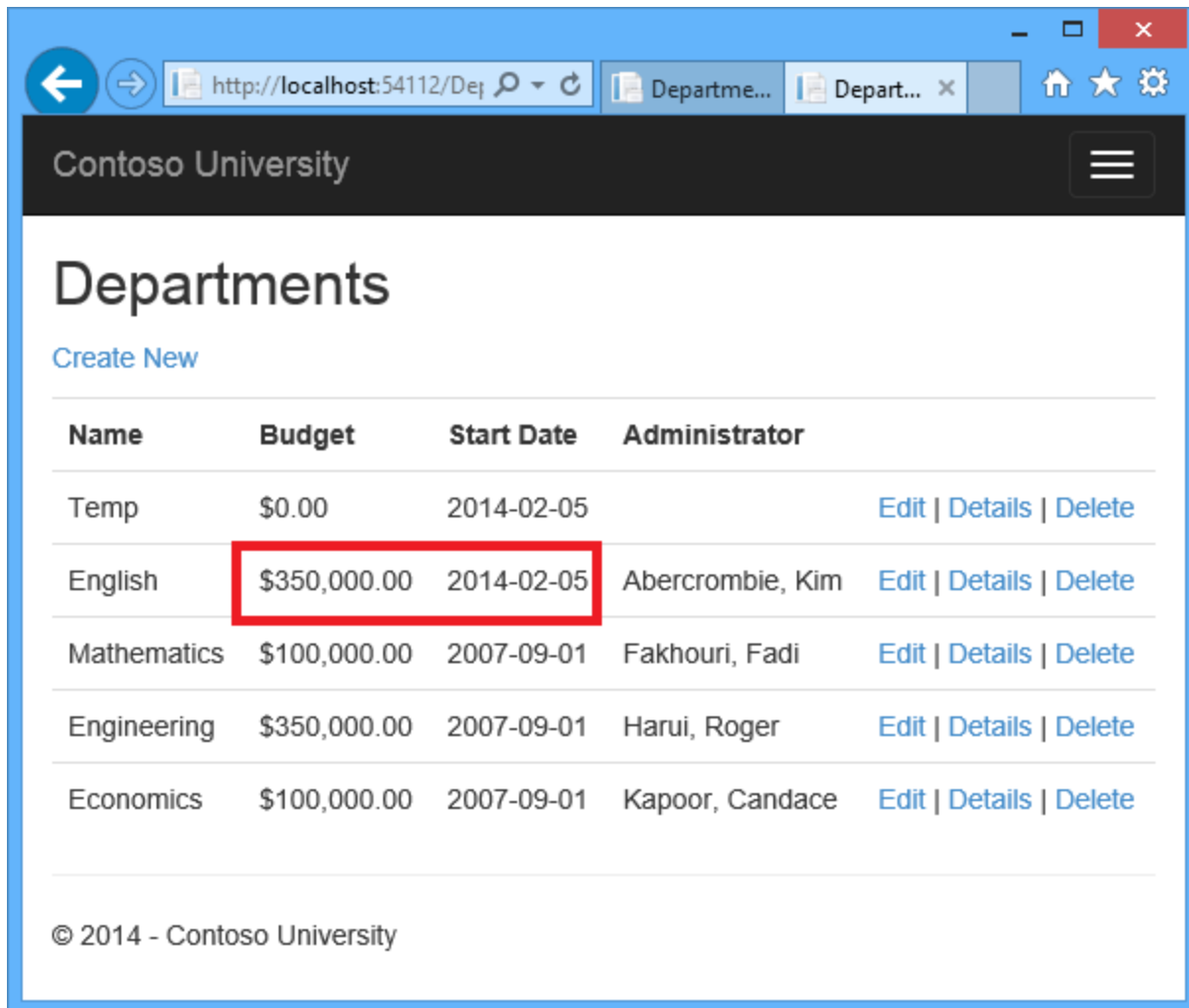
Budget
 Current value: \$0.00

Start Date
 Current value: 9/1/2007

Administrator

[Back to List](#)

Click **Save** again. The value you entered in the second browser tab is saved along with the original value of the data you changed in the first browser. You see the saved values when the Index page appears.



Updating the Delete Page

For the Delete page, the Entity Framework detects concurrency conflicts caused by someone else editing the department in a similar manner. When the `HttpGet Delete` method displays the confirmation view, the view includes the original `RowVersion` value in a hidden field. That value is then available to the `HttpPost Delete` method that's called when the user confirms the deletion. When the Entity Framework creates the SQL `DELETE` command, it includes a `WHERE` clause with the original `RowVersion` value. If the command results in zero rows affected (meaning the row was changed after the Delete confirmation page was displayed), a concurrency exception is thrown, and the `HttpGet Delete` method is called with an error flag set to `true` in order to redisplay the confirmation page with an error message. It's also possible that zero rows were affected because the row was deleted by another user, so in that case a different error message is displayed.

In `DepartmentController.cs`, replace the `HttpGet Delete` method with the following code:

```
public ActionResult Delete(int? id, bool? concurrencyError)
```

```

{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Department department = db.Departments.Find(id);
    if (department == null)
    {
        return HttpNotFound();
    }

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

    return View(department);
}

```

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

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

```

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Delete(Department department)
{
    try
    {
        db.Entry(department).State = EntityState.Deleted;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
}

```

```

    }
    catch (DbUpdateConcurrencyException)
    {
        return RedirectToAction("Delete", new { concurrencyError=true } );
    }
    catch (DataException /* dex */)
    {
        //Log the error (uncomment dex variable name after DataException and
        add a line here to write a log.
        ModelState.AddModelError(string.Empty, "Unable to delete. Try again,
        and if the problem persists contact your system administrator.");
        return View(department);
    }
}

```

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

```
public ActionResult DeleteConfirmed(int id)
```

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

```
public ActionResult Delete(Department department)
```

You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code named the `HttpPost Delete` method `DeleteConfirmed` to give the `HttpPost` method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the `HttpPost` and `HttpGet` delete methods.

If a concurrency error is caught, the code redisplay the Delete confirmation page and provides a flag that indicates it should display a concurrency error message.

In *Views\Department\Delete.cshtml*, replace the scaffolded code with the following code that adds an error message field and hidden fields for the `DepartmentID` and `RowVersion` properties. The changes are highlighted.

```

@model ContosoUniversity.Models.Department

@{
    ViewBag.Title = "Delete";
}

<h2>Delete</h2>

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

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">

```

```

<dt>
    Administrator
</dt>

<dd>
    @Html.DisplayFor(model => model.Administrator.FullName)
</dd>

<dt>
    @Html.DisplayNameFor(model => model.Name)
</dt>

<dd>
    @Html.DisplayFor(model => model.Name)
</dd>

<dt>
    @Html.DisplayNameFor(model => model.Budget)
</dt>

<dd>
    @Html.DisplayFor(model => model.Budget)
</dd>

<dt>
    @Html.DisplayNameFor(model => model.StartDate)
</dt>

<dd>
    @Html.DisplayFor(model => model.StartDate)
</dd>

</dl>

@using (Html.BeginForm()) {
    @Html.AntiForgeryToken()
    @Html.HiddenFor(model => model.DepartmentID)
    @Html.HiddenFor(model => model.RowVersion)

    <div class="form-actions no-color">
        <input type="submit" value="Delete" class="btn btn-default" /> |
        @Html.ActionLink("Back to List", "Index")
    </div>
}
</div>

```

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

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

It replaces LastName with FullName in the Administrator field:

```

<dt>
    Administrator
</dt>

```

```
<dd>
    @Html.DisplayFor(model => model.Administrator.FullName)
</dd>
```

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

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

Run the Departments Index page. Right click the **Delete** hyperlink for the English department and select **Open in new tab**, then in the first tab click the **Edit** hyperlink for the English department.

In the first window, change one of the values, and click **Save** :

Contoso University

Edit

Department

Name

Budget

Start Date

Administrator

[Back to List](#)

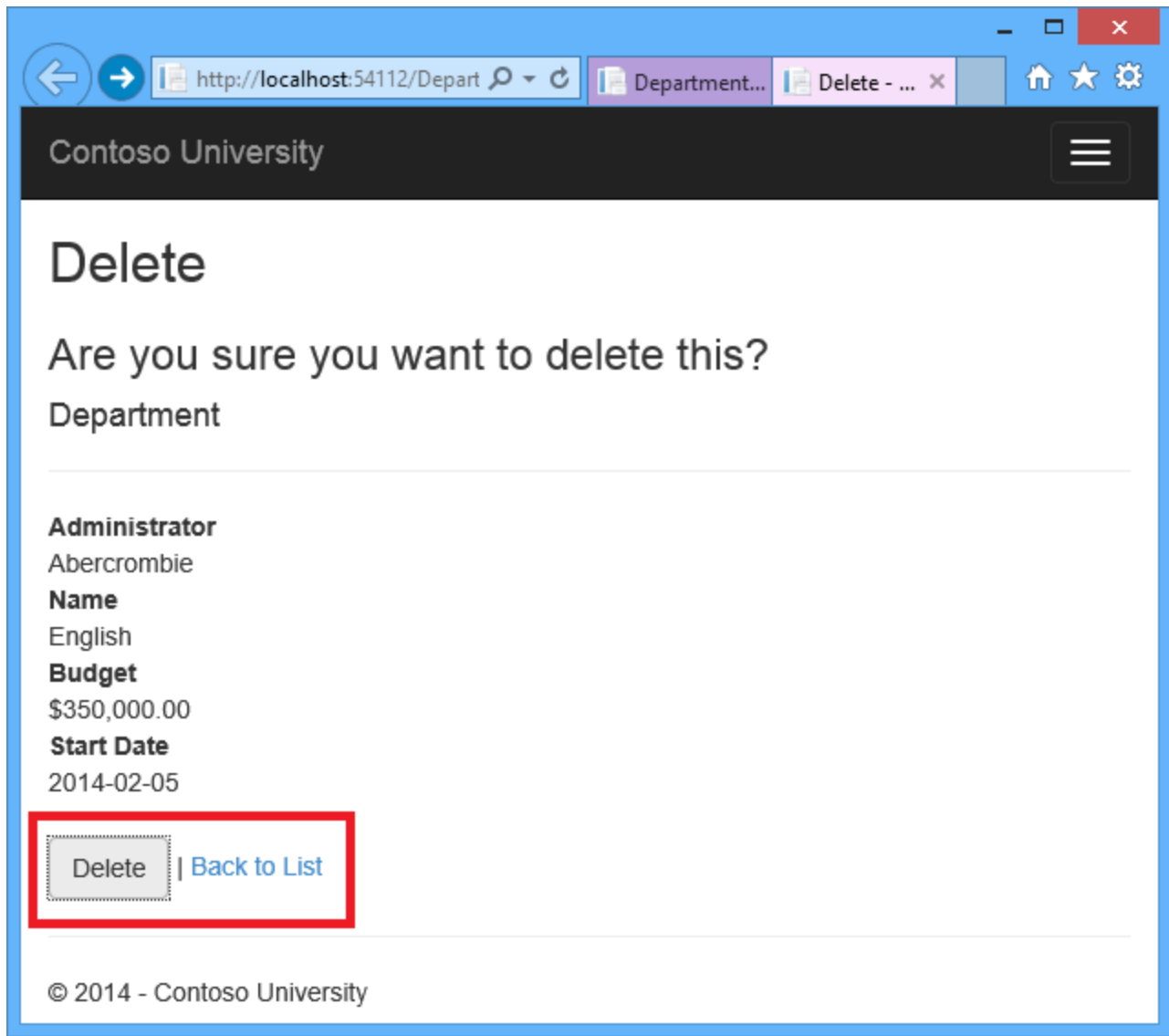
© 2014 - Contoso University

The Index page confirms the change.

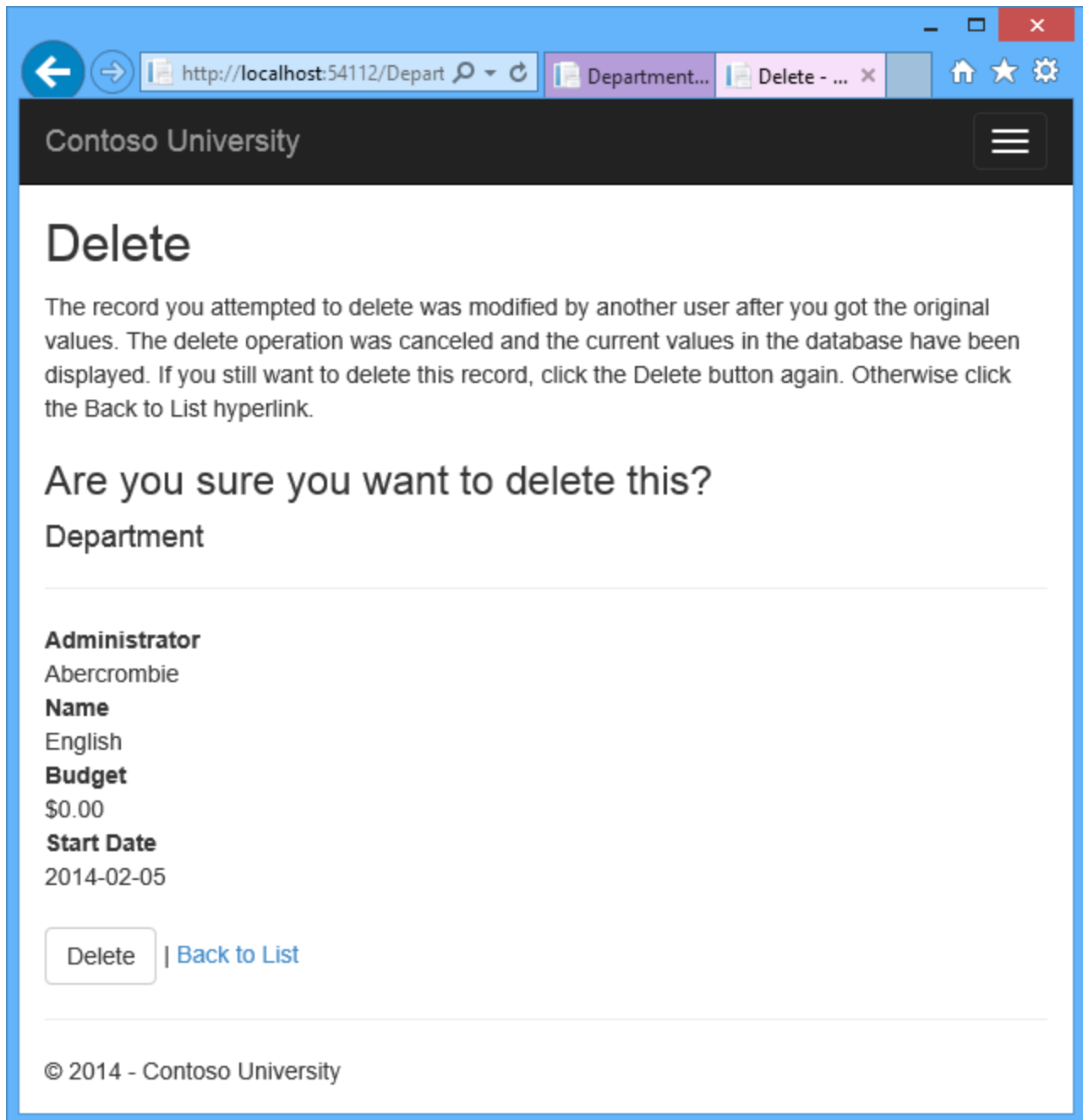
The screenshot shows a web browser window with the URL `http://localhost:54112/Depart`. The page title is "Contoso University". The main heading is "Departments". Below the heading is a link "Create New". A table lists departments with columns: Name, Budget, Start Date, and Administrator. The "English" department's budget of "\$0.00" is highlighted with a red box. Each row has links for "Edit", "Details", and "Delete". The footer shows "© 2014 - Contoso University".

Name	Budget	Start Date	Administrator	
Temp	\$0.00	2014-02-05		Edit Details Delete
English	\$0.00	2014-02-05	Abercrombie, Kim	Edit Details Delete
Mathematics	\$100,000.00	2007-09-01	Fakhouri, Fadi	Edit Details Delete
Engineering	\$350,000.00	2007-09-01	Harui, Roger	Edit Details Delete
Economics	\$100,000.00	2007-09-01	Kapoor, Candace	Edit Details Delete

In the second tab, click **Delete**.



You see the concurrency error message, and the Department values are refreshed with what's currently in the database.



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

Summary

This completes the introduction to handling concurrency conflicts. For information about other ways to handle various concurrency scenarios, see [Optimistic Concurrency Patterns](#) and [Working with Property Values](#) on MSDN. The next tutorial shows how to implement table-per-hierarchy inheritance for the `Instructor` and `Student` entities.