Getting Started with Entity Framework 6 Code First using MVC 5

Tom Dykstra, Rick Anderson

Step By Step, Guide

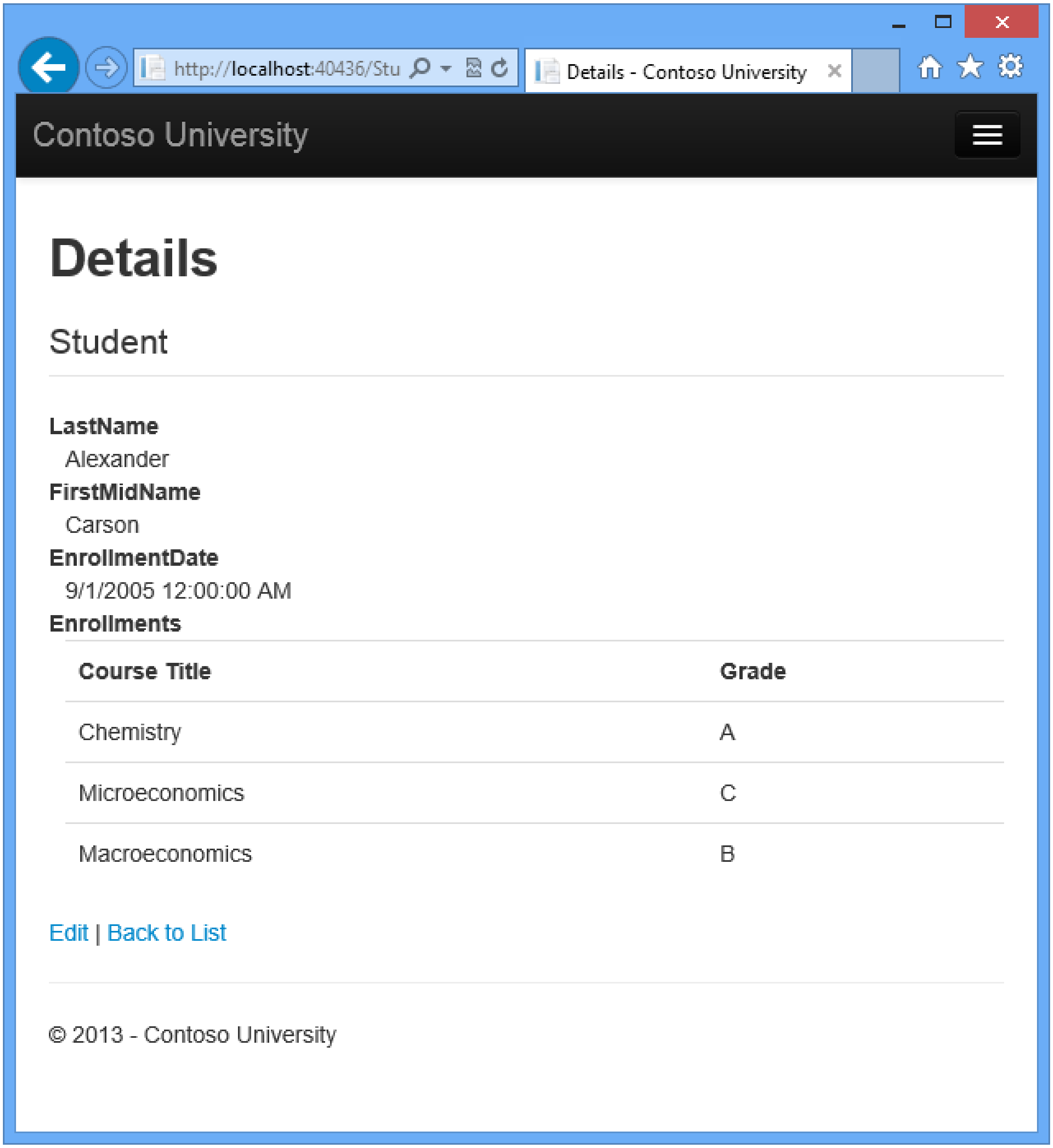
Microsoft

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

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

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

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



Details

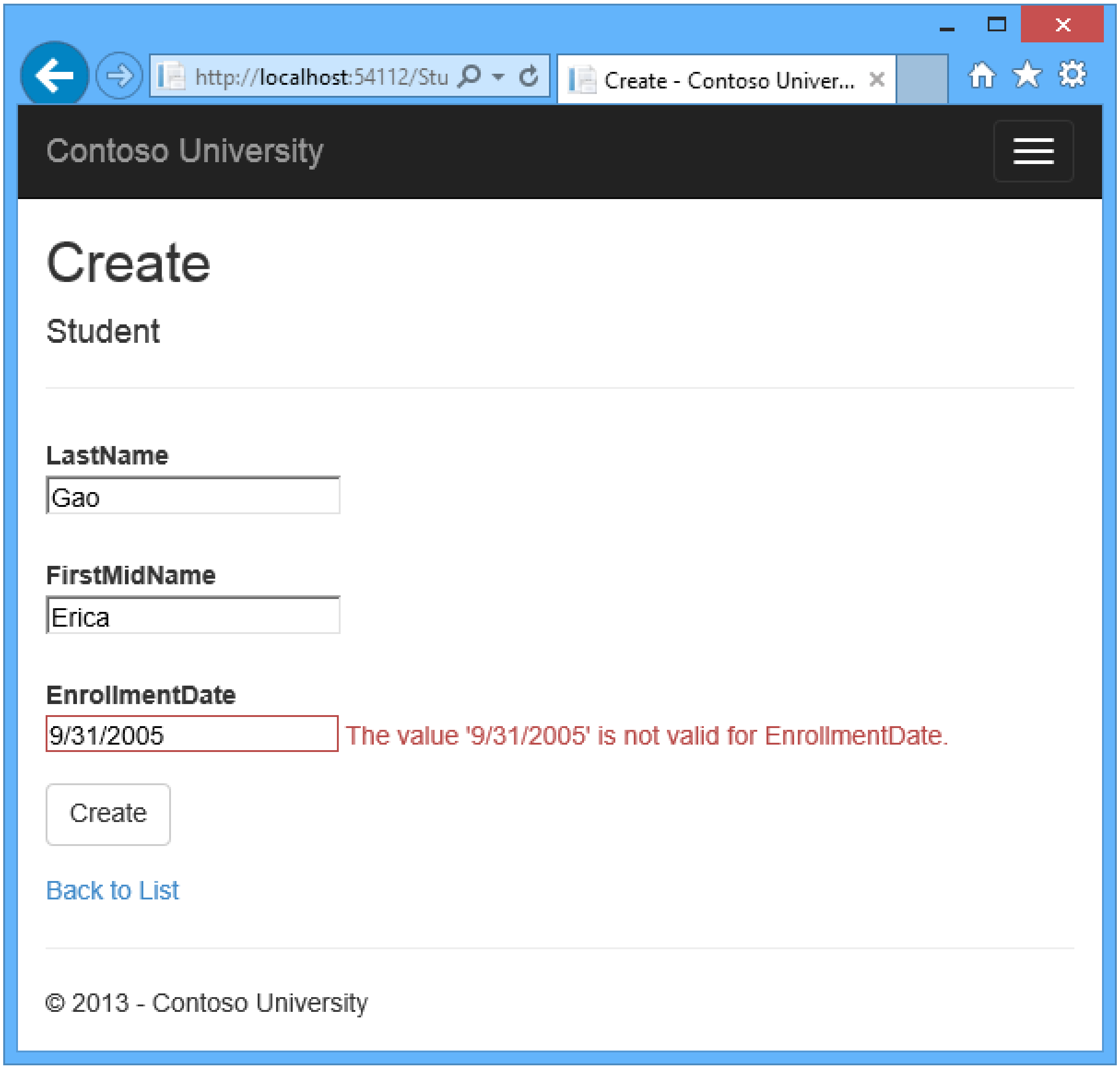
Student

LastName
Alexander
FirstMidName
Carson
EnrollmentDate
9/1/2005 12:00:00 AM
Enrollments

Course Title	Grade
Chemistry	A
Microeconomics	C
Macroeconomics	B

Edit | Back to List

© 2013 - Contoso University



Create

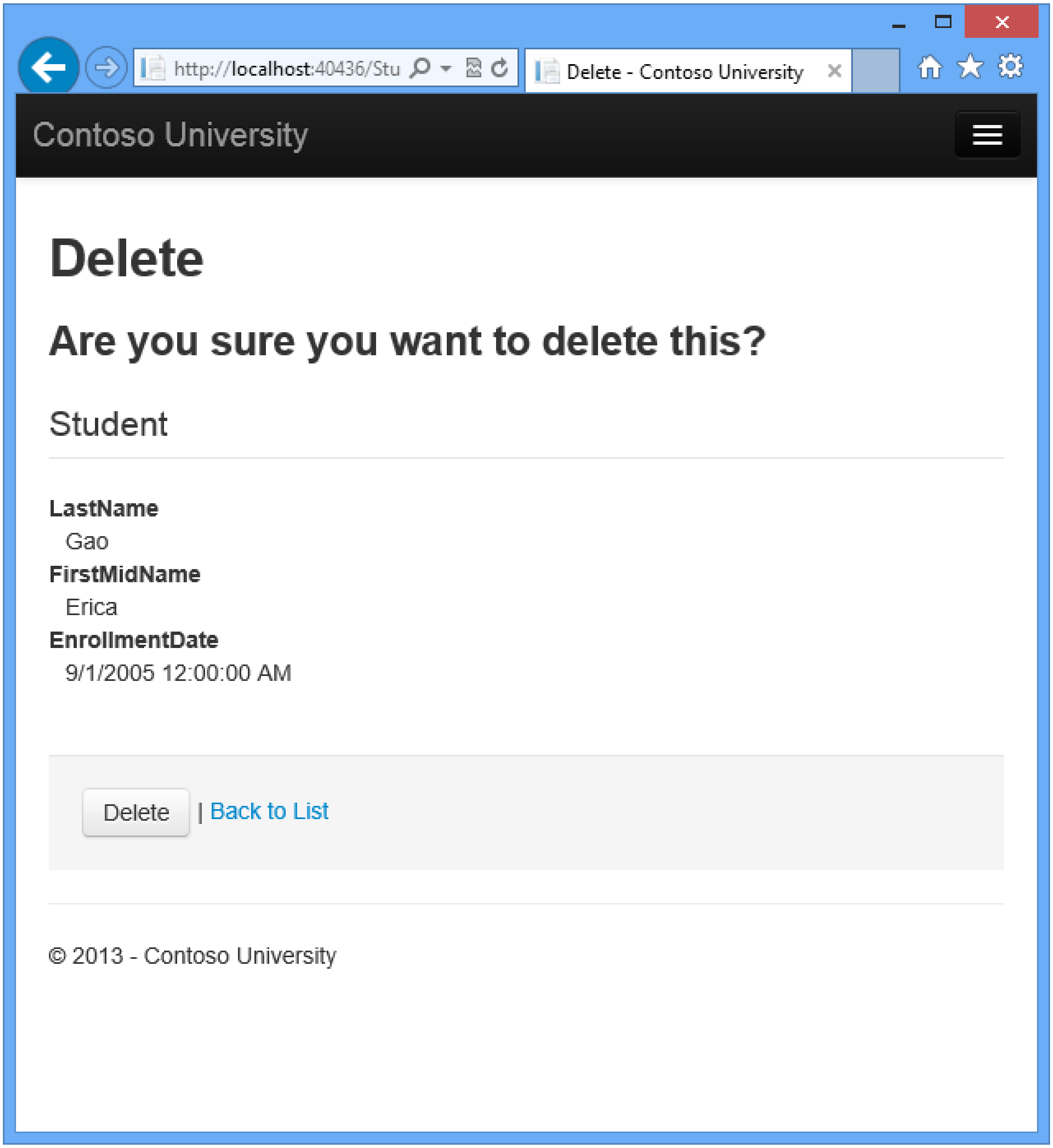
Student

LastName
Gao

FirstMidName
Erica

EnrollmentDate
9/31/2005
The value '9/31/2005' is not valid for EnrollmentDate.

© 2013 - Contoso University



Delete

Are you sure you want to delete this?

Student

LastName
Gao
FirstMidName
Erica
EnrollmentDate
9/1/2005 12:00:00 AM

© 2013 - Contoso University

Create a Details Page

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

In *Controllers\StudentController.cs*, the action method for the `Details` view uses the Find method to retrieve a single `Student` entity.

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

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

Route data

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

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

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

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

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

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

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

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

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

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

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

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

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

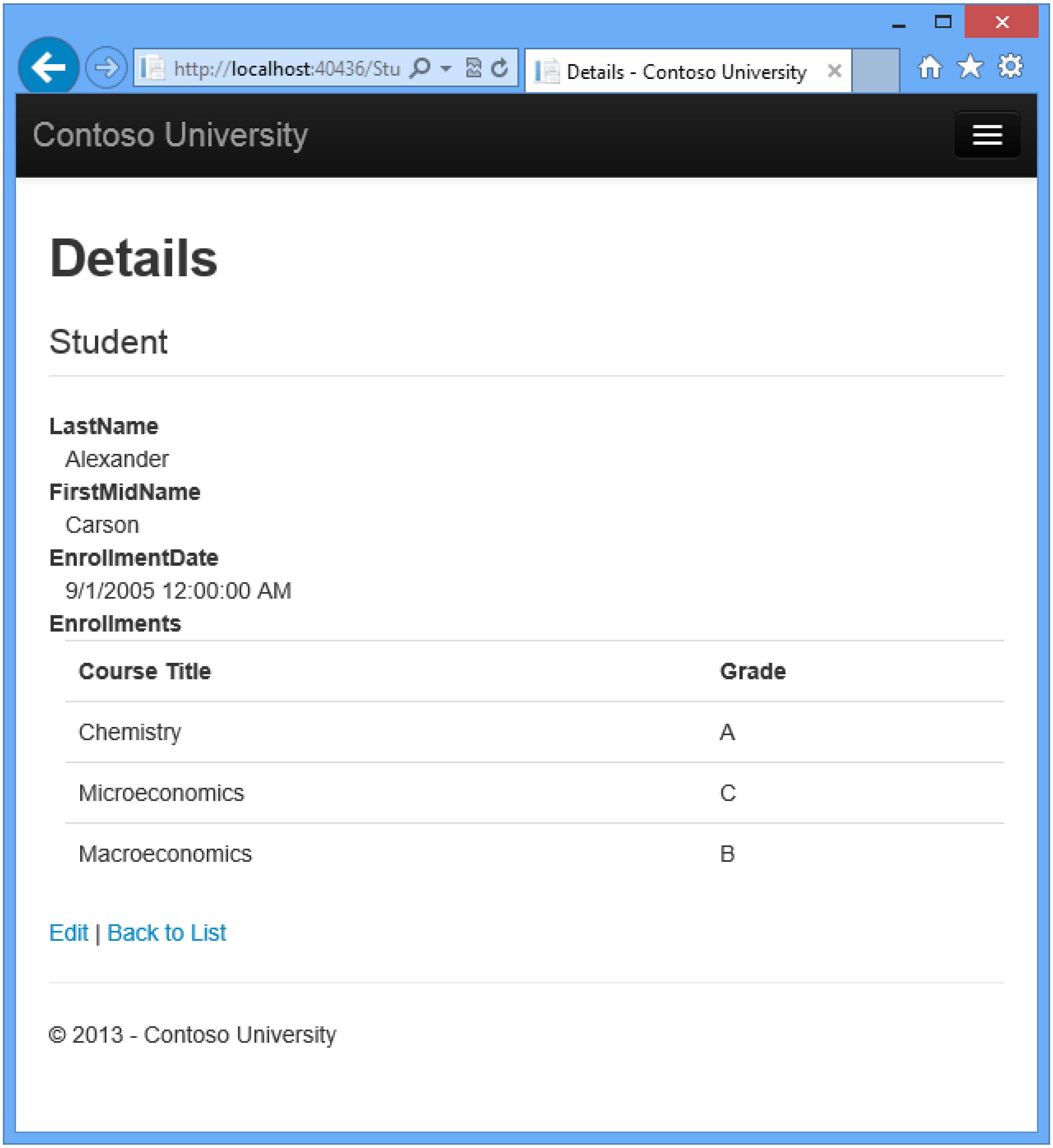
```
        <dt>
            @Html.DisplayNameFor(model => model.EnrollmentDate)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.EnrollmentDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Enrollments)
        </dt>
        <dd>
            <table class="table">
                <tr>
                    <th>Course Title</th>
                    <th>Grade</th>
                </tr>
                @foreach (var item in Model.Enrollments)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem =>
item.Course.Title)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Grade)
                        </td>
                    </tr>
                }
            </table>
        </dd>
    </dl>
</div>
<p>
    @Html.ActionLink("Edit", "Edit", new { id = Model.ID }) |
    @Html.ActionLink("Back to List", "Index")
</p>
```

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

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

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

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



Contoso University

Details

Student

LastName
Alexander
FirstMidName
Carson
EnrollmentDate
9/1/2005 12:00:00 AM
Enrollments

Course Title	Grade
Chemistry	A
Microeconomics	C
Macroeconomics	B

Edit | Back to List

© 2013 - Contoso University

Update the Create Page

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

```
[HttpPost]
```

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

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

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

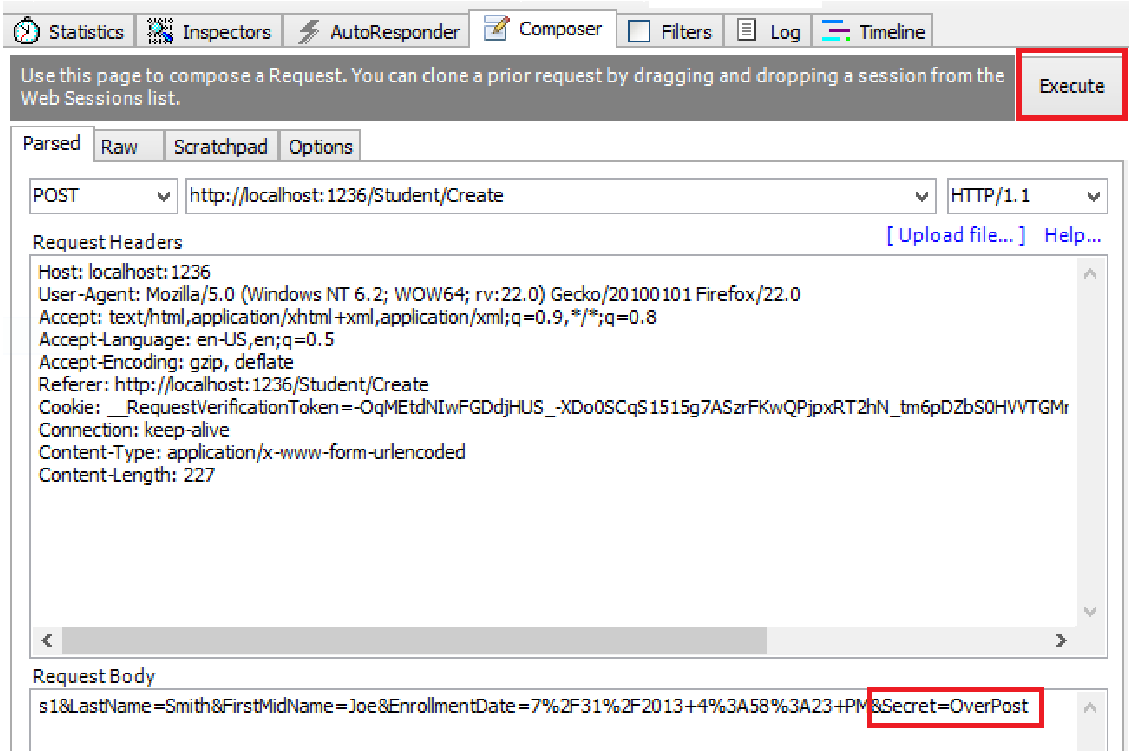
Security Note: The `ValidateAntiForgeryToken` attribute helps prevent cross-site request forgery attacks. It requires a corresponding `Html.AntiForgeryToken()` statement in the view, which you'll see later.

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

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

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

```
    }
```

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


Statistics
Inspectors
AutoResponder
Composer
Filters
Log
Timeline
Use this page to compose a Request. You can clone a prior request by dragging and dropping a session from the Web Sessions list.
Execute
Parsed
Raw
Scratchpad
Options
POST
http://localhost:1236/Student/Create
HTTP/1.1
Request Headers
[Upload file...]
Help...
Request Body

```
Host: localhost:1236
User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64; rv:22.0) Gecko/20100101 Firefox/22.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost:1236/Student/Create
Cookie: __RequestVerificationToken=-OqMEtdNIwFGDdjHUS_-XDo0SCqS1515g7ASzrFKwQPjpxRT2hN_tm6pDZbS0HVVTGMr
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 227
```

```
s1&LastName=Smith&FirstMidName=Joe&EnrollmentDate=7%2F31%2F2013+4%3A58%3A23+PM&Secret=OverPost
```

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

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

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

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

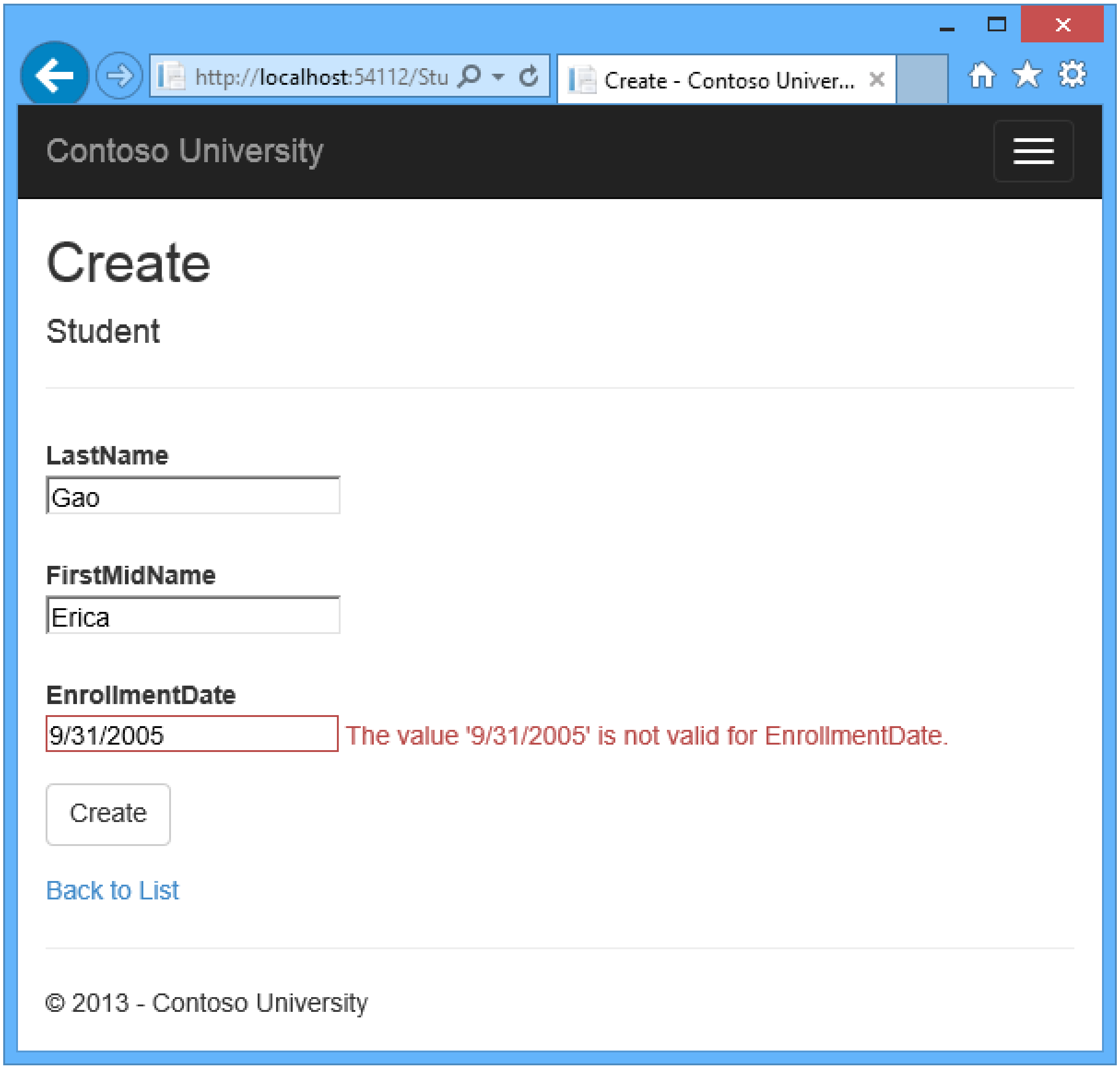
The code in *Views\Student\Create.cshtml* is similar to what you saw in *Details.cshtml*, except that `EditorFor` and `ValidationMessageFor` helpers are used for each field instead of `DisplayFor`. Here is the relevant code:

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

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

No changes are required in *Create.cshtml*.

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



Create

Student

LastName

Gao

FirstMidName

Erica

EnrollmentDate

9/31/2005

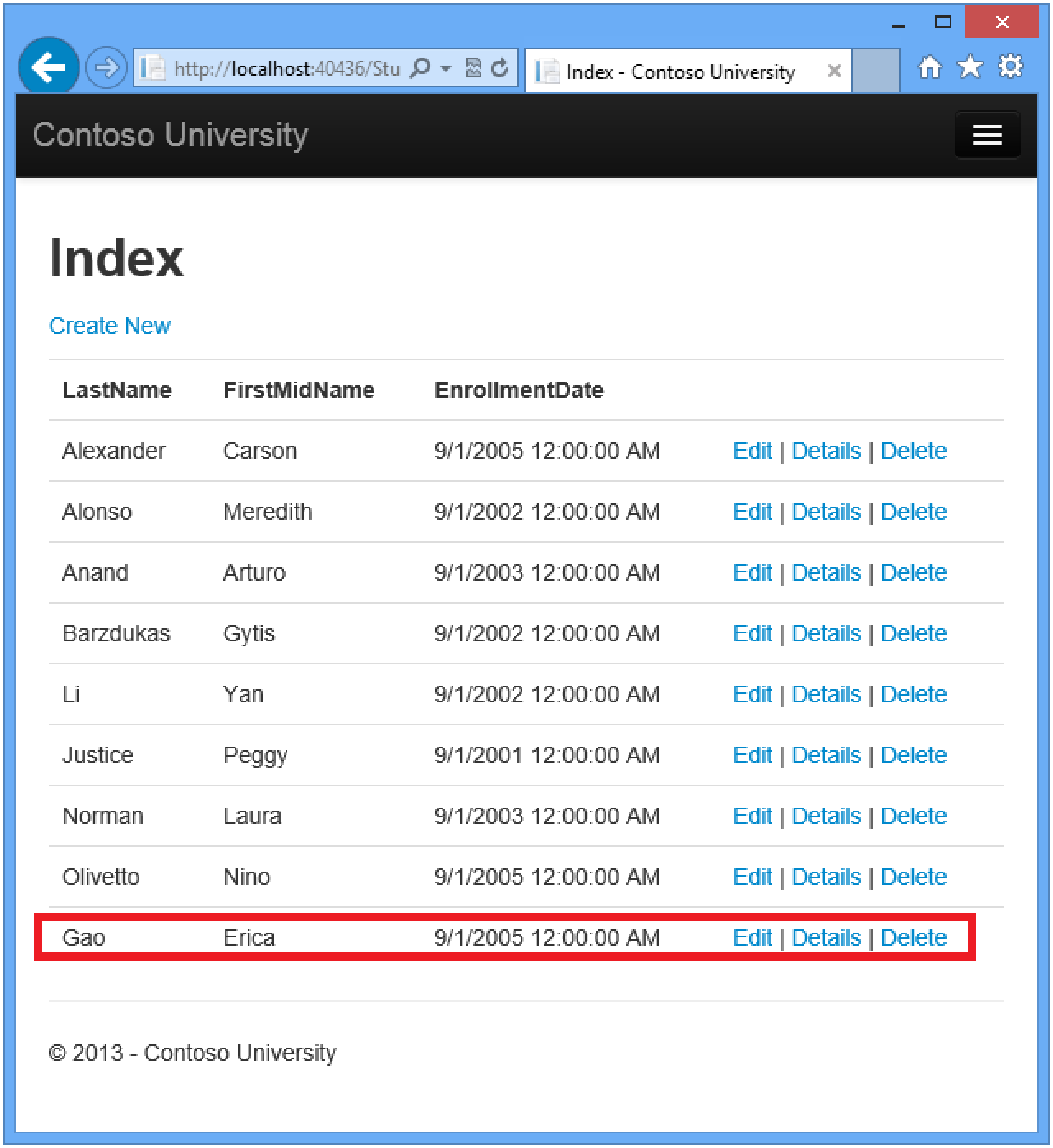
The value '9/31/2005' is not valid for EnrollmentDate.

© 2013 - Contoso University

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

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

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



Index

Create New

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

© 2013 - Contoso University

Update the Edit HttpPost Page

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

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

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

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

Entity States and the Attach and SaveChanges Methods

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

An entity may be in one of the following states:

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

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

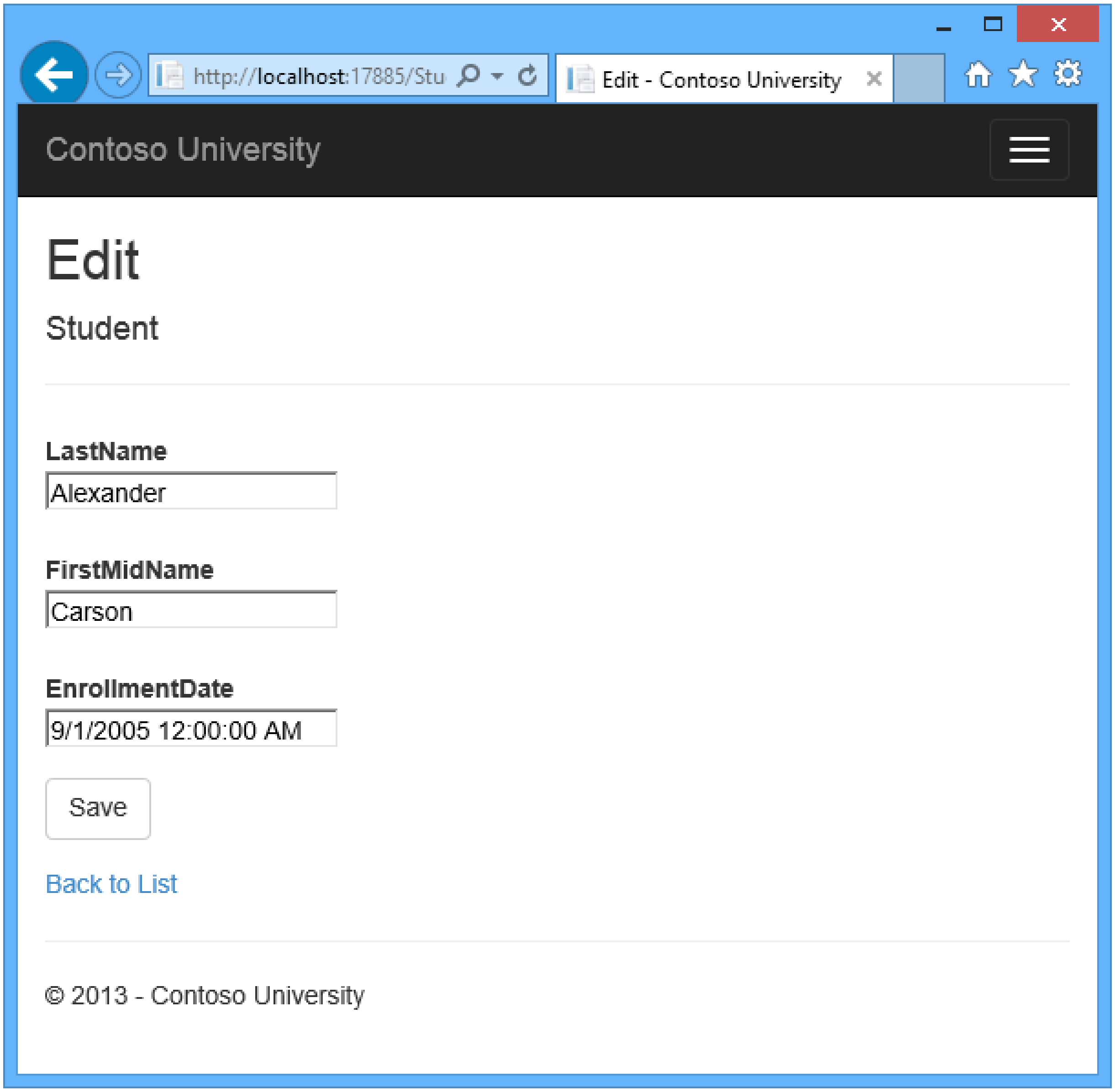
In a desktop application, state changes are typically set automatically. In a desktop type of application, you read an entity and make changes to some of its property values. This causes its entity state to automatically be changed to `Modified`. Then when you call `SaveChanges`, the Entity Framework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

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

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

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

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


http://localhost:17885/Stu
Edit - Contoso University
Contoso University

Edit

Student

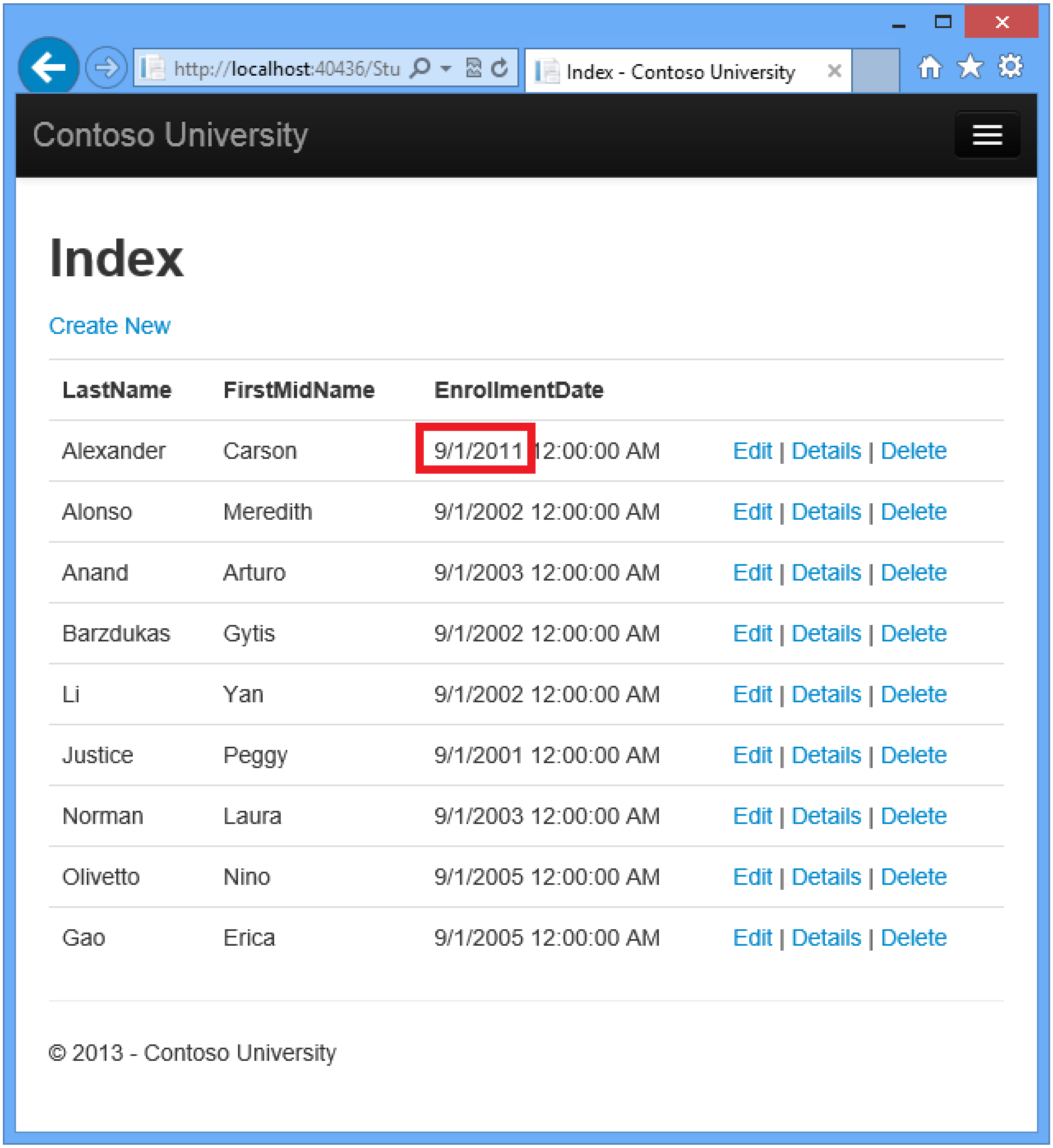
LastName
Alexander

FirstMidName
Carson

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

Save
Back to List
© 2013 - Contoso University

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



Index

Create New

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

© 2013 - Contoso University

Updating the Delete Page

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

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

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

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

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

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

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

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Delete(int id)
{
    try
    {
        Student student = db.Students.Find(id);
        db.Students.Remove(student);
        db.SaveChanges();
    }
    catch (DataException/* dex */)
    {
        //Log the error (uncomment dex variable name and add a line
here to write a log.
        return RedirectToAction("Delete", new { id = id,
saveChangesError = true });
    }
    return RedirectToAction("Index");
}
```

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

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

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

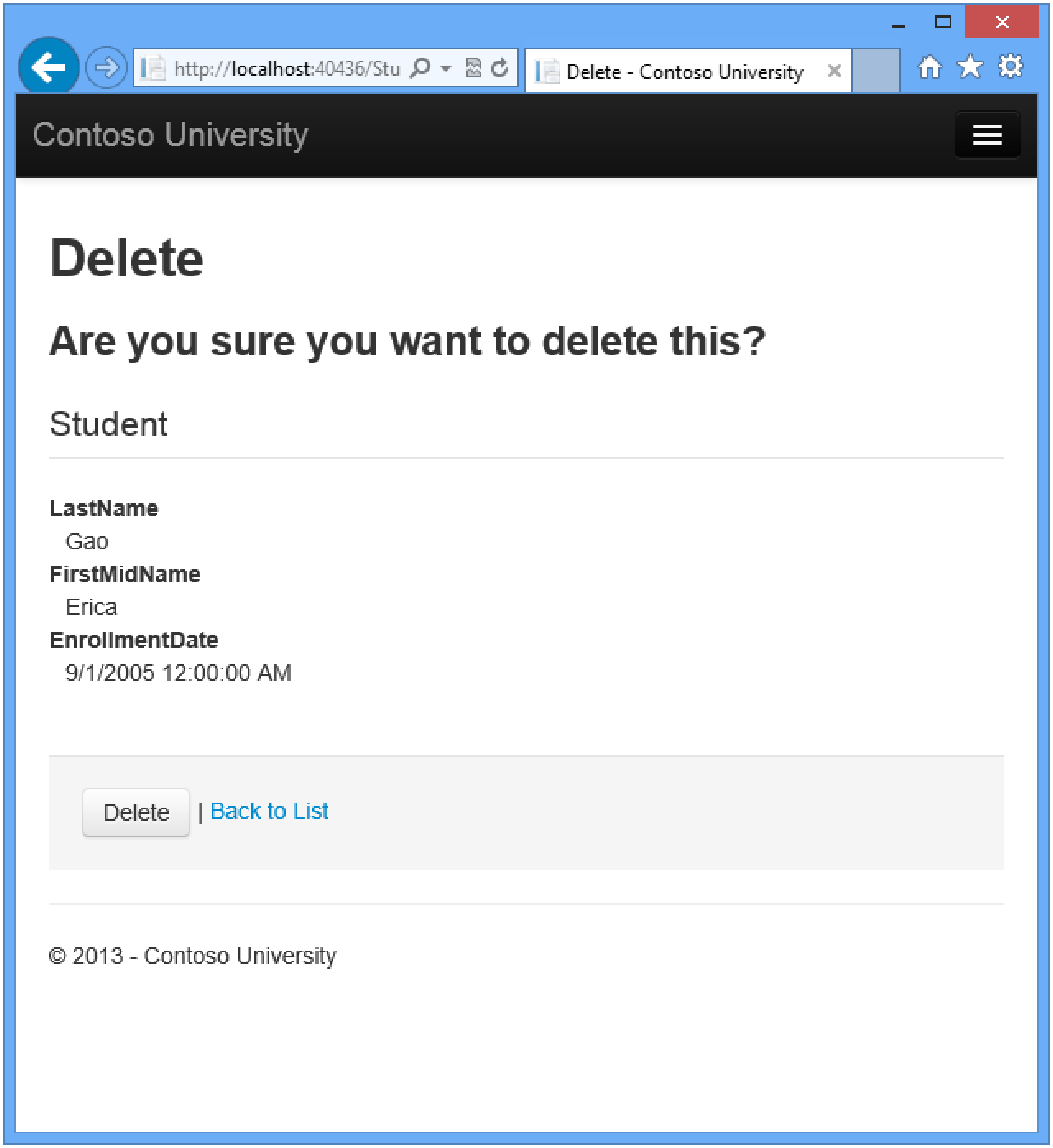
This code instantiates a `Student` entity using only the primary key value and then sets the entity state to `Deleted`. That's all that the Entity Framework needs in order to delete the entity.

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

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

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

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



Delete

Are you sure you want to delete this?

Student

LastName
Gao
FirstMidName
Erica
EnrollmentDate
9/1/2005 12:00:00 AM

© 2013 - Contoso University

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

Ensuring that Database Connections Are Not Left Open

To make sure that database connections are properly closed and the resources they hold freed up, you have to dispose the context instance when you are done with it. That is why the scaffolded code provides a Dispose method at the end of the `StudentController` class in *StudentController.cs*, as shown in the following example:

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

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

Handling Transactions

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

Summary

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

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