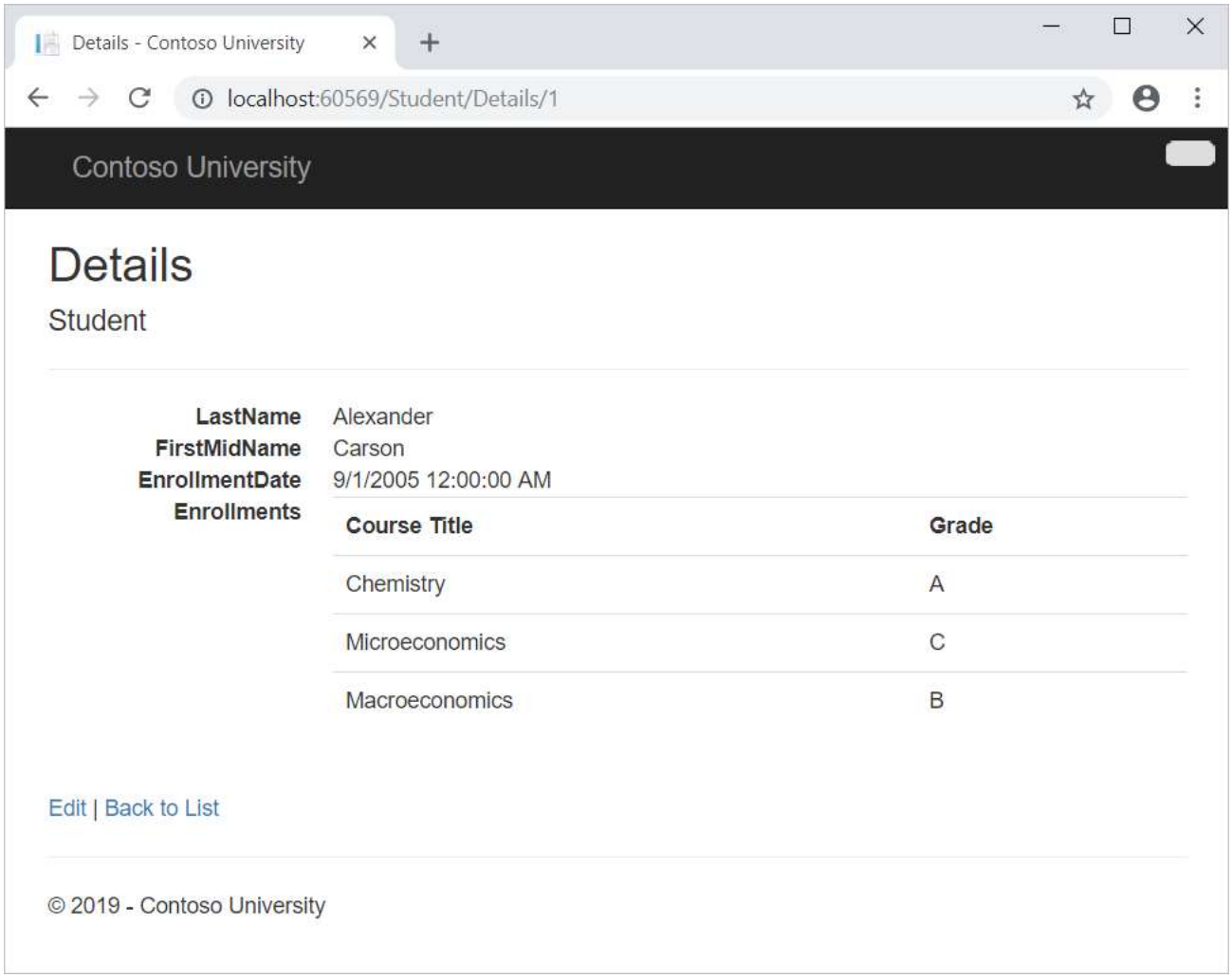
Tutorial: Implement CRUD Functionality with the Entity Framework in ASP.NET MVC

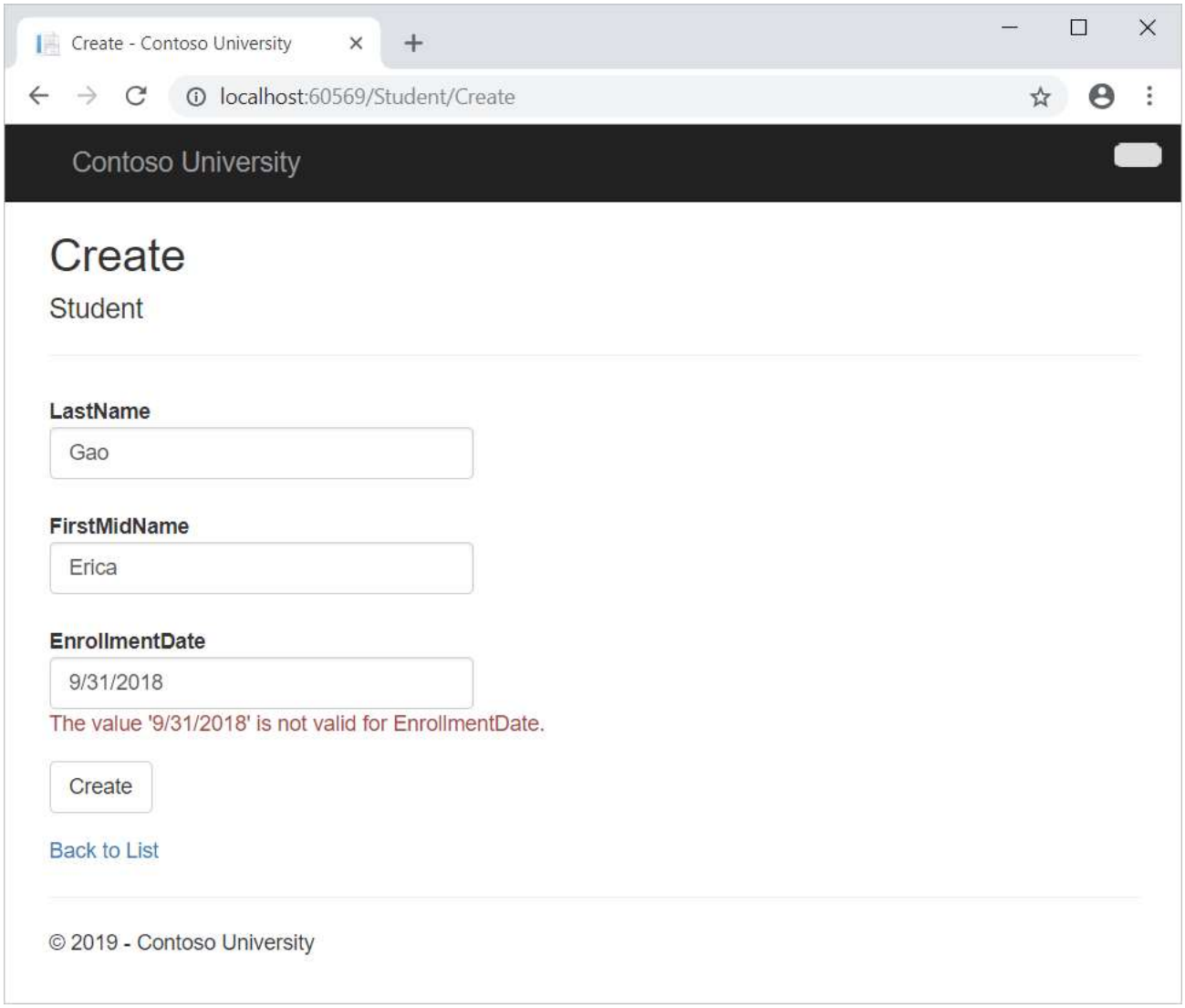
Article • 07/01/2022 • 18 minutes to read

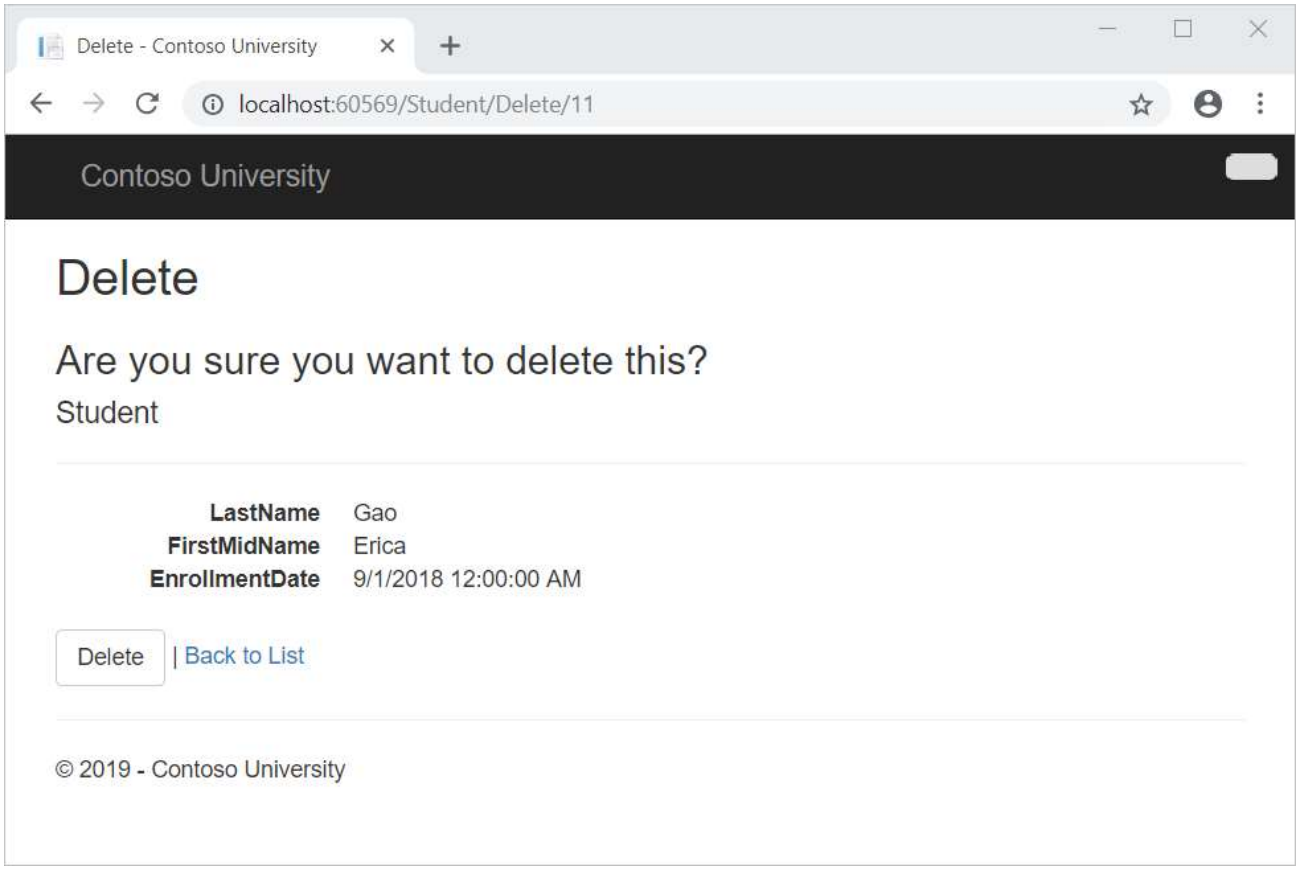
In the previous tutorial, you created an MVC application that stores and displays data using the Entity Framework (EF) 6 and SQL Server LocalDB. In this tutorial, you review and customize the create, read, update, delete (CRUD) code that the MVC scaffolding automatically creates for you in controllers 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 EF 6 itself, they don't use repositories. For info about how to implement repositories, see the **ASP.NET Data Access Content Map**.

Here are examples of the web pages you create:


Details - Contoso University
localhost:60569/Student/Details/1
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
© 2019 - Contoso University


Create - Contoso University
localhost:60569/Student/Create
Contoso University
Create
Student
LastName
Gao
FirstMidName
Erica
EnrollmentDate
9/31/2018
The value '9/31/2018' is not valid for EnrollmentDate.
Create
Back to List
© 2019 - Contoso University



In this tutorial, you:

- ✓ Create a Details page
- ✓ Update the Create page
- ✓ Update the HttpPost Edit method
- ✓ Update the Delete page
- ✓ Close database connections
- ✓ Handle transactions

Prerequisites

- Create the Entity Framework Data Model

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.

```
C#

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.

Tip: 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:

```
C#

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.

CSHTML

```
@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.

CSHTML

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

To create the Details page

1. Open *Views\Student\Details.cshtml*.

   Each field is displayed using a `DisplayFor` helper, as shown in the following example:

   CSHTML

   ```
   <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:

   CSHTML

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

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

4. Close the browser.

Update the Create page

1. In *Controllers\StudentController.cs*, replace the HttpPostAttribute `Create` action method with the following code. This code adds a `try-catch` block and removes `ID` from the BindAttribute attribute for the scaffolded method:

```
C#
```

```
[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);
}
```