بسم الله الرحمن الرحیم

آموزشگاه تحلیل داده

تخصصی ترین مرکز برنامه نویسی و دیتابیس در ایران

پیاده سازی قابلیت های ساده ی CRUD با تکنولوژی EF در برنامه ی تحت وب ASP.NET MVC

مدرس : مهندس افشین رفوآ

دوره آموزش MVC

پیاده سازی قابلیت های ساده ی CRUD با تکنولوژی EF در برنامه ی تحت وب ASP.NET MVC

در فصل قبلی یک برنامه ی MVC نوشتیم که داده ها را با بهره گیری از تکنولوژی EF و SQL Server LocalDB ذخیره کرده و نمایش می داد. در مبحث حاضر کدهای CRUD (ایجاد کردن، خواندن، بروز رسانی و حذف) را که به صورت خودکار توسط امکان MVC scaffolding ایجاد می شود ویژه ی controller ها و view ها مورد بازبینی قرار داده و سفارشی تنظیم می کنیم.

نکته: رسم بر این است که برای ایجاد یک لایه ی انتزاعی (abstraction layer) بین controller و لایه ی دسترسی داده (data access layer)، الگوی طراحی Repository را پیاده سازی کرد. در مقاله های آموزشی به منظور قرار دادن تمرکز بر روی آموزش نحوه ی استفاده از خود EF، از الگوی طراحی Repository استفاده نمی شود.

در درس حاضر، صفحات وب زیر را ایجاد خوهیم کرد:

www.tahlildadeh.com

http://localhost:40436/Stu

Details - Contoso University

Contoso University

Details

Student

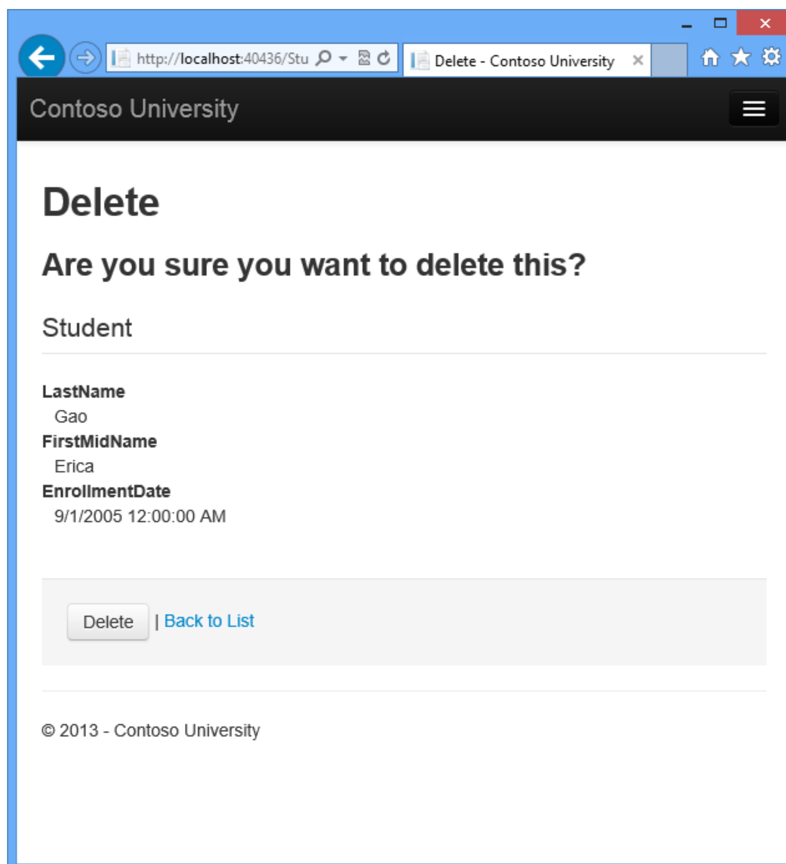
LastName
Alexander
FirstMidName
Carson
EnrollmentDate
9/1/2005 12:00:00 AM
Enrollments

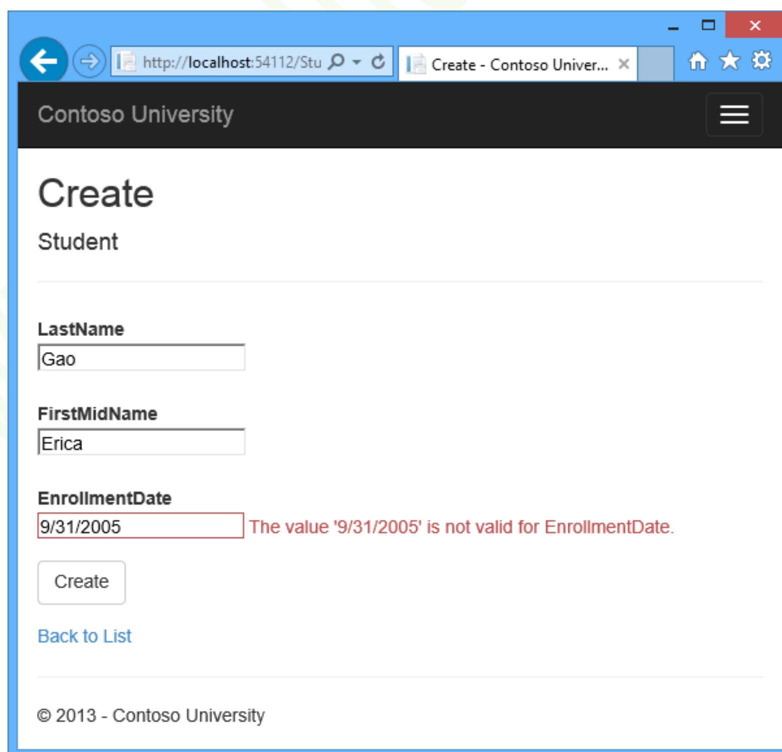
Course Title	Grade
Chemistry	A
Microeconomics	C
Macroeconomics	B

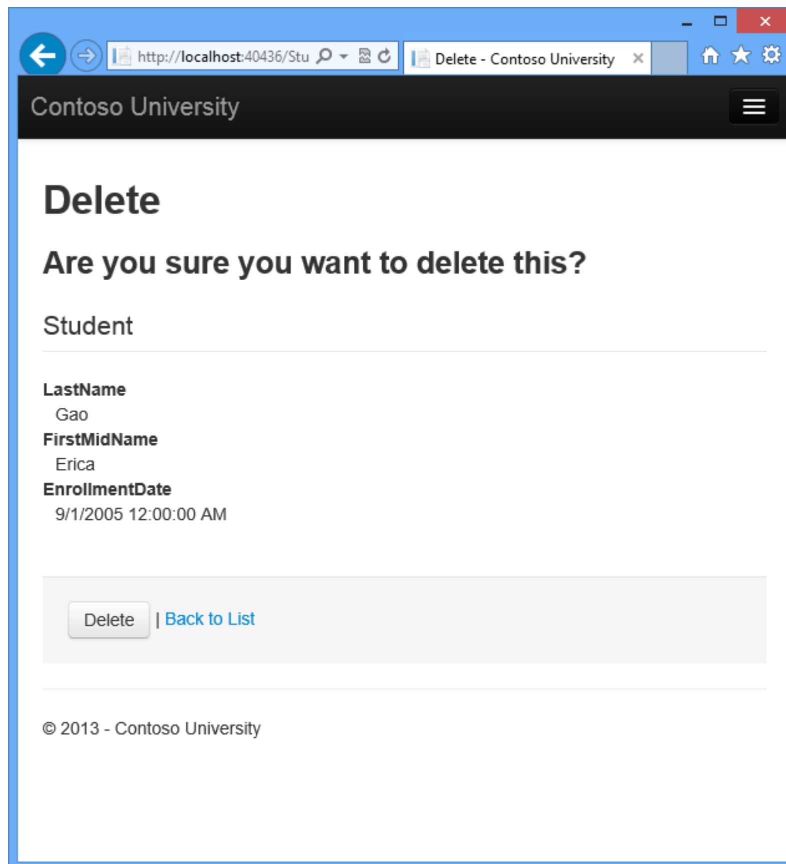
Edit | Back to List

© 2013 - Contoso University

آدرس آموزشگاه : تهران - خیابان شریعتی - بالا تر از خیابان ملک - جنب بانک صادرات - پلاک 561 - واحد 7
88146323 - 88446780 - 88146330


http://localhost:40436/Stu
Delete - Contoso University
Contoso University
Delete
Are you sure you want to delete this?
Student
LastName
Gao
FirstMidName
Erica
EnrollmentDate
9/1/2005 12:00:00 AM
Delete
| Back to List
© 2013 - Contoso University


http://localhost:54112/Stu
Create - Contoso Univer...
Contoso University
Create
Student
LastName
Gao
FirstMidName
Erica
EnrollmentDate
9/31/2005
The value '9/31/2005' is not valid for EnrollmentDate.
Create
Back to List
© 2013 - Contoso University



ایجاد یک صفحه به نام Details

کدی که توسط قابلیت scaffolding برای صفحه ی Index ایجاد شده، خاصیت Enrollments را به کدهای اضافه نکرده است، زیرا خاصیت ذکرشده دارای یک مجموعه (collection) می باشد. در صفحه ی Details، محتوای مجموعه ی مورد نظر را در یک جدول HTML نمایش خواهیم داد.

در فایل Controllers\StudentController.cs، یک action method برای Details view وجود دارد که با استفاده از متد Find، یک موجودیت Student را بازیابی می کند.

```
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);
    }
```

مقدار کلید به عنوان پارامتر id به متد ارسال می شود که از route data ی لینک Details موجود در صفحه ی Index گرفته شده.

داده های مسیریابی (Route data)

Route data، درواقع اطلاعاتی هستند که model binder در یک بخش از URL، تعریف شده در جدول مسیریابی، آن ها را می یابد. به عنوان مثال، مسیر پیش فرض (default route) قسمت های controller، action و id را مشخص می کند:

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

در URL نمونه ی زیر، مسیر پیش فرض Instructor را به عنوان controller، Index را به عنوان action و 1 را به عنوان id نگاشت می کند؛ اقلام ذکر شده همگی مقادیر route data محسوب می شوند.

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

"?courseID=2021" درواقع یک query string است. اگر id را به عنوان مقدار query string ارسال کنید، model binder بازهم کار خواهد کرد:

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

URL ها توسط دستورهای ActionLink در Razor view ایجاد می شوند. در کد زیر، پارامتر id با مسیر پیش فرض مطابقت دارد، بنابراین id به route data اضافه می شود.

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

اما در کد نمونه ی زیر، courseID با پارامتر تعریف شده در مسیر پیش فرض مطابقت ندارد، بنابراین به عنوان یک query string اضافه شده است.

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


www.tahlildadeh.com

1. فایل Views\Student\Details.cshtml را باز کنید. همان طور که در مثال زیر مشاهده می کنید، هریک از فیلدها به کمک یک helper DisplayFor نمایش داده می شود:

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

2. پس از فیلد EnrollmentDate و درست قبل از تگ پایانی </dl>، کد رنگی شده ی زیر را برای نمایش فهرستی از enrollment ها، درج کنید:

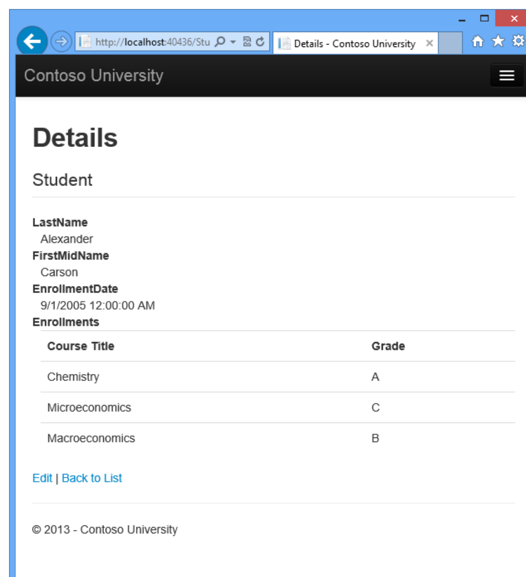
```
    <dt>
        @Html.DisplayNameFor(model => model.EnrollmentDate)
     </dt>

     <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>
```

چنانچه تورفتگی کدها پس از جایگذاری (paste)، دچار بهم ریختگی شد، کلید های CTRL-K-D را همزمان فشار داده تا مرتب شود.

این کد داخل entity های موجود در Enrollments navigation property، حلقه می زند. به ازای هر یک از موجودیت های Enrollment، فیلدهای course title و grade را نمایش می دهد. course title از موجودیت Course بازیابی شده که در این موجودیت در Course navigation property موجودیت Enrollments ذخیره گردیده است. تمامی این داده ها به صورت خودکار از پایگاه داده و در صورت نیاز بازیابی می شوند. (به عبارتی دیگر، در این اینجا از قابلیت بارگذاری با تاخیر یا lazy loading استفاده می شود. به این خاطر که eager loading را برای property Courses navigation تعریف نکرده اید، enrollment ها در query یکسان که student ها را بازیابی کرد، برگردانده نشده اند. در عوض، اولین باری که سعی می کنید به Enrollments navigation property دسترسی پیدا کنید، یک query جدید برای بازگردانی داده های مورد نظر به پایگاه داده ارسال می شود.)

3. با انتخاب تب Students و کلیک بر روی یک لینک Details برای Alexander Carson، صفحه را اجرا کنید. (اگر کلیدهای CTRL+F5 را به هنگام باز بودن فایل Details.cshtml فشار دهید، یک پیغام خطای HTTP 400 دریافت می کنید، زیرا Visual Studio تلاش می کند صفحه ی Details را اجرا کند ولی دسترسی از طریق لینکی صورت نگرفته که در آن نمایش دادن student تعریف شده باشد. جهت حل این مشکل، کافی است "Student/Details" را URL حذف کرده و مجددا امتحان کنید و یا مرورگر را بسته، بر روی پروژه راست کلیک کرده و View را انتخاب کنید، سپس View in Browser را کلیک نمایید.)

در زیر فهرست دوره های آموزشی (course) و نمره های (grade) دانشجوی مورد نظر را مشاهده می کنید:



بروز رسانی صفحه ی Create

1. در فایل Controllers\StudentController.cs، متد اکشن HttpPost Create را با کد زیر جایگزین کنید تا یک قطعه کد try-catch به آن اضافه نموده و ID را از متد ساخته شده توسط scaffolding حذف کنید:

```
[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);
}
```

2. این کد موجودیت Student ایجاد شده توسط ASP.NET MVC model binder را به Students entity set اضافه کرده، سپس تغییرات را در پایگاه داده ذخیره می کند. (Model binder درواقع به یک قابلیت

ASP.NET MVC اشاره دارد که به شما کمک می کند با داده های ارسال شده توسط یک فرم آسان تر کار کنید؛ model binder همچنین نوع مقادیر ارسالی فرم را به نوع های CLR تبدیل کرده و آن را در قالب پارامترهایی به action method مورد نظر پاس می دهد. در این نمونه، model binder به وسیله ی مقادیر property، از مجموعه ی Form یک موجودیت Student نمونه سازی کرده است.)

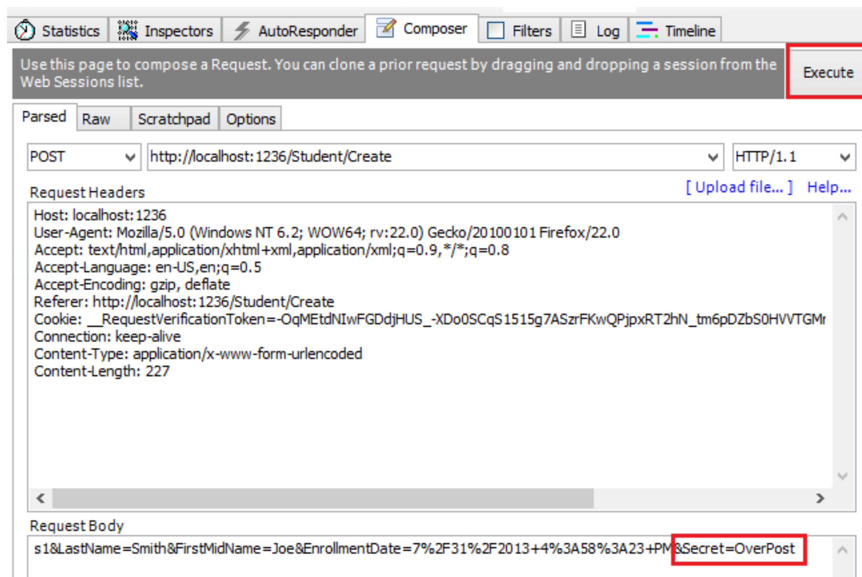
به این خاطر ID را از خصیصه ی Bind حذف می کنیم که ID یک مقدار primary key هست که SQL Server به صورت خودکار هنگامی که سطر درج می شود، تنظیم می کند. ورودی دریافتی از کاربر مقدار ID را تنظیم نمی کند.

نکته ی امنیتی: خصیصه ی ValidateAntiForgeryToken از حملات جعل درخواست سایت متقابل (cross-site request forgery) جلوگیری می کند. به ازای خصیصه ی ذکر شده بایستی یک دستور Html.AntiForgeryToken() متناظر در view بکار برد، که بعدا به آن خواهیم پرداخت.

خصیصه ی Bind یکی از روش های مقابله با حمله ی over posting (پست های پست سرهم) در سناریوها یا عملیات create می باشد. برای مثال، فرض کنید موجودیت Student دارای یک خاصیت Secret است که شما به هردلیلی نمی خواهید این صفحه آن را مقداردهی کند.

```
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; }

 }
```

حتی اگر هم فیلد Secret در صحفه ی وب خود ندارید، یک هکر می تواند به کمک ابزاری همچون fiddler و یا نوشتن مقداری کد جاوا اسکریپت، یک مقدار فرم Secret ارسال کند. خصیصه ی Bind فیلدهایی را که model binder حین ساختن یک نمونه ی Student استفاده می کند، محدود می سازد. بدون این خصیصه، model binder مقدار فرم Secret را برای ساختن نمونه ی موجودیت Student بکار می برد، در پی آن هر مقداری که هکر برای فیلد فرم Secret تعیین کرده، در پایگاه داده ی شما نفوذ کرده و بروز رسانی می شود. تصویر زیر ابزار fiddler را نشان می دهد که یک فیلد Secret (با مقدار "OverPost") به مقادیر ارسالی فرم اضافه می کند:



بنابراین مقدار "OverPost" به صورت موفقیت آمیز به خاصیت Secret سطر درج شده اضافه می گردد، اگرچه شما به هیچ وجه نمی خواستید که صفحه ی وب مورد نظر آن خاصیت را مقداردهی کند.

از این رو توصیه می کنیم پارامتر Include را به همراه خصیصه ی Bind برای فیلدهای whitelist بکار ببرید. همچنین می توان پارامتر Exclude را برای فیلدهای blacklist ای که نمی خواهید استفاده شوند، بکار ببرید. Include به این خاطر امن تر است که هنگامی که شما یک خصیصه ی جدید به entity مورد نظر اضافه می کنید، فیلد جدید به صورت خودکار توسط لیست Exclude محافظت نمی شود.

با خواندن موجودیت هایی اول از پایگاه داده و سپس فراخوانی TryUpdateModel به همراه لیستی از خاصیت های مجاز، از Overposting در زمان ویرایش جلوگیری می کنیم. این روشی است که در این آموزش مورد استفاده قرار می دهیم.

روش دیگری که بسیاری از برنامه نویسان آن را ترجیح می دهند، این است که بجای کلاس های entity از view model همراه با model binding استفاده کنید. در این روش، تنها خاصیت هایی را که می خواهید در view model بروز رسانی شود، اضافه (include) کنید. به مجرد اینکه MVC model binder کار خود را به اتمام رساند، خاصیت های view model را در نمونه ی entity جای گذاری کنید.برای این منظور ترجیحا از ابزاری همچون AutoMapper استفاده می کنیم. با استفاده از db.Entry برای نمونه ی entity، حالت (state) نمونه ی entity را روی unchanged تنظیم کنید، سپس Property("PropertyName").IsModified را در هر خاصیت entity که در model جدید اضافه شده، بر روی true تنظیم کنید. این روش در هر دو سناریوی edit و create قابل استفاده می باشد.

به استثنای خصیصه ی Bind، قطعه کد try-catch تنها تغییری است که به کد ایجاد شده توسط scaffolding، وارد شد. اگر خطایی که از DataException مشتق می شود در حین ذخیره شدن ضبط (catch) شود، در آن صورت یک پیغام خطای عمومی (generic error message) صادر می شود. خطاهای DataException

بیشتر از عوامل خارجی نشات می گیرند تا خطاهای مربوط به برنامه نویسی، بنابراین معمولا به کاربر توصیه می شود مجددا تلاش کند. یک برنامه ی کارآمد و باکیفیت باید قابلیت ثبت خطا (exception logging) را داشته باشد اما از آنجایی که برنامه یک نمونه ی آزمایشی بود، قابلیت مذکوردر آن پیاده سازی نشد.

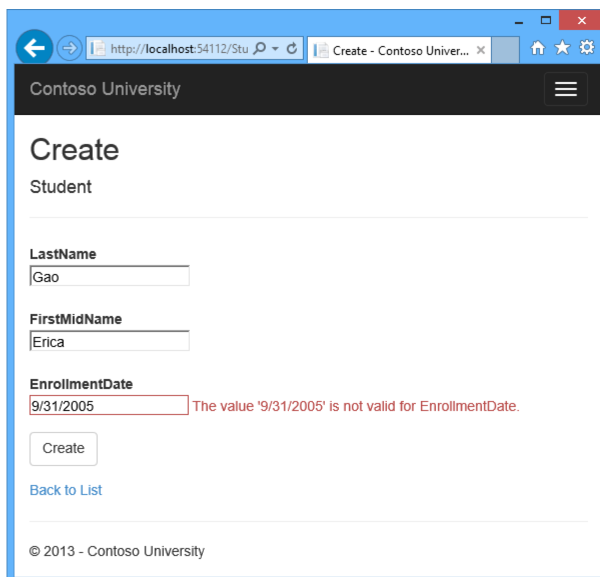
کد موجود در فایل Views\Student\Create.cshtml شبیه به کدی است که در فایل Details.cshtml مشاهده کردید، با این تفاوت که در آن بجای DisplayFor، helper های EditorFor و ValidationMessageFor برای هریک از فیلدها بکار برده شده است.

```
<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>
```

فایل Create.chstml، تابع ()@Html.AntiForgeryToken را نیز شامل می شود. این تابع به همراه خصیصه ی ValidateAntiForgeryToken در controller و به منظور مقابله با حملات جعل درخواست سایت متقابل (cross-site request forgery) مورد استفاده قرار می گیرد.

لازم نیست هیچ تغییری در فایل Create.chstml ایجاد کنید.

2. با انتخاب تب Students و کلیک بر روی Create New، صفحه ی مورد نظر را اجرا نمایید.
3. چندین اسم به ضمیمه ی یک تاریخ نامعتبر وارد کرده و با کلیک بر روی Create، پیام خطا را مشاهده نمایید.



این همان اعتبارسنجی در سمت سرویس دهنده است که به صورت پیش فرض دراختیار شما قرار می گیرد؛ در مباحث بعدی نحوه ی ایجاد اعتبارسنجی در سمت سرویس گیرنده را با افزودن خصیصه های جدید آموزش خواهیم داد. کدهای رنگی شده ی زیر model validation check در متد Create را نشان می دهد:

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

4. تاریخ فیلد EnrolllmentDate را به یک تاریخ معتبر تغییر داده و Create را کلیک نمایید. خواهید دید که دانشجوی جدید در صفحه ی Index ظاهر خواهد شد.

Contoso University

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

بروز رسانی متد Edit HttpPost

در فایل Controllers\StudentController.cs، متد HttpGet Edit (بدون خصیصه ی HttpPost) با استفاده از متد Find موجودیت Student انتخابی را بازیابی می کند، همان طور که در متد Details شاهد آن بودیم. لزومی ندارد این متد را اصلاح کنید.

اما، اکشن متد HttpPost Edit را با کد زیر جایگزین می کنیم:

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public ActionResult EditPost(int? id)
{
  if (id == null)
  {
    return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
  }
  var studentToUpdate = db.Students.Find(id);
  if (TryUpdateModel(studentToUpdate, "",
   new string[] { "LastName", "FirstMidName", "EnrollmentDate" }))
  {
    try
    {
      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(studentToUpdate);

}
```

تغییراتی که اعمال شد، یک روش بهینه ی امنیتی را پیاده سازی می کند که از حملات پست مکرر (overposting) جلوگیری می کند، scaffolder یک خصیصه ی Bind ایجاد کرده و نیز موجودیتی که توسط model binder ساخته شده را با یک Modified flag به entity set اضافه می کند. استفاده از چنین کدی دیگر توصیه نمی شود زیرا خصیصه ی Bind تمامی داده های از پیش موجود در فیلدهایی که داخل پارامتر Include آورده نشده را پاک می کند. در آینده، MVC controller scaffolder گونه ای بروز رسانی می شود که برای متدهای Edit سرخود خصیصه ی Bind ایجاد نکند.

کد جدید موجودیت جاری را خوانده و فیلدهای فرم را با ورودی های کاربر از داده های فرم ارسالی به وسیله ی فراخوانی TryUpdateModel بروز رسانی می کند. قابلیت ردیابی خودکار تغییر (automatic change tracking) EF، Modified flag را به موجودیت (entity) مورد نظر متصل (set) می کند. به محض فراخوانی متد SaveChanges، Modified flag سبب می شود EF با ساخت دستورات SQL، سطر مورد نظر در پایگاه داده را بروز رسانی کند. تداخلات همزمانی (Concurrency conflicts) نادیده گرفته می شوند، همچنین تمامی ستون های سطر پایگاه داده (database row) از جمله آن ستون هایی که کاربر تغییر نداده، بروز رسانی می شوند.

یکی از بهترین روش های مقابله با پست مکرر (overposting)، این است که فیلدهایی که می خواهید توسط صفحه ی Edit قابل بروز رسانی باشد، در پارامترهای TryUpdateModel مجاز اعلام (whitelist) کنید. در حال حاضر هیچ فیلد اضافه بر سازمانی وجود ندارد که از آن محافظت کنید، اما لیست کردن آن فیلدهایی که می خواهید model binder پیوند دهد (bind)، اطمینان حاصل می کند که اگر شما در آینده فیلدهایی را به

data model اضافه کردید، آن فیلدها به صورت خودکار محافظت شوند و تا زمانی که آن ها را به صورت صریح به اینجا (در پارامترهای TryUpdateModel) اضافه نکرده اید، این روند ادامه یابد.

در نتیجه ی این تغییرات، ورودی ها و خروجی های متد HttpPost Edit (HttpPost Edit method signature) با امضای متد HttpGet edit یکسان می باشد؛ بنابراین متد را به EditPost تغییر نام می دهیم.

وضعیت های موجودیت (Entity States) و متدهای SaveChanges

Database context بررسی می کند آیا موجودیت های داخل حافظه با سطرهای متناظر در پایگاه داده همگام است یا خیر، سپس آن اطلاعات را ضبط می کند. این اطلاعات تعیین می کند هنگامی که شما متد SaveChanges را صدا می زنید، چه اتفاقی رخ دهد. به عنوان مثال، زمانی که یک موجودیت جدید را به متد Add پاس می دهید، وضعیت آن موجودیت بر روی Added تنظیم می شود. سپس هنگامی که متد SaveChanges را صدا می زنید، database context یک دستور SQL INSERT صادر می کند.

یک موجودیت می تواند در وضعیت های زیر قرار داشته باشد:

- وضعیت Added. در این حالت موجودیت مورد نظر هنوز در پایگاه داده موجود نمی باشد. تابع SaveChanges بایستی یک دستور INSERT صادر کند.
- Unchanged. نیازی نیست متد SaveChanges بر روی این موجودیت تغییر اعمال کند. هنگامی که شما یک entity را از پایگاه داده می خوانید، موجودیت ابتدا در این وضعیت قرار دارد.
- Modified. در این وضعیت برخی و یا تمام مقادیر property موجودیت، اصلاح شده است. در این حالت متد SaveChanges بایستی یک دستور UPDATE صادر کند.
- Deleted. موجودیت مورد نظر برای حذف شدن علامت گذاری شده است. SaveChanges باید یک دستور DELETE صادر کند.
- Detached. Database context هیچ اطلاعی از موجودیت ندارد.

در یک برنامه ی تحت ویندوز، تغییرات وضعیت معمولا به صورت خودکار تنظیم می شوند. در برنامه های تحت ویندوز، یک موجودیت را می خوانیم و تغییراتی را به مقادیر خاصیت (property) آن اعمال می کنیم. این امر سبب می شود وضعیت برنامه به صورت اتوماتیک بر روی Modified تنظیم شود. سپس به هنگام فراخوانی

www.tahlildadeh.com

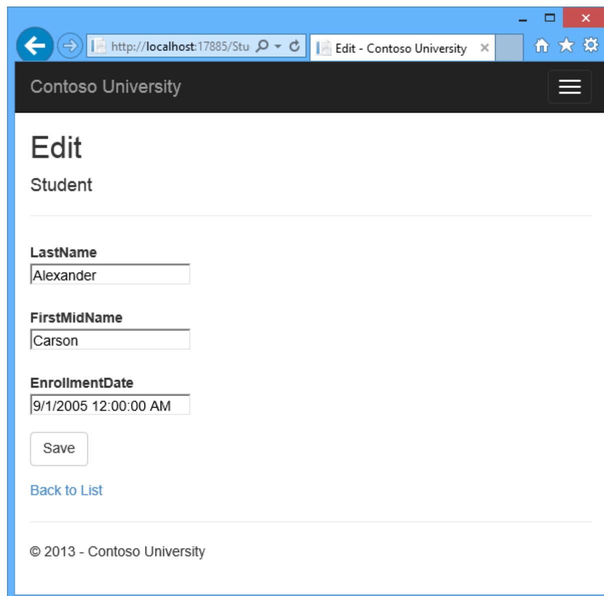
متد SaveChanges، EF یک دستور SQL UPDATE اجرا می کند که تنها property هایی را بروز رسانی می کند که شما تغییر دادید.

نظر به اینکه ذات برنامه های تحت وب مجزا می باشد، امکان این پیوستگی و توالی منظم وجود ندارد. DbContext ای که یک موجودیت را می خواند، بلافاصله پس از اجرا و نمایش (render) صفحه، دور انداخته می شود. هنگامی که اکشن متد HttpPost Edit صدا زده می شود، یک درخواست جدید شکل گرفته و در پی آن شما یک نمونه از DbContext خواهید داشت، از این رو شما مجبور می شوید به صورت دستی وضعیت entity را بر روی Modified تنظیم نمایید. سپس هنگامی که شما متد SaveChanges فراخوانی می کنید، EF تمامی ستون های سطر پایگاه داده را بروز رسانی می کند، چون که context هیچ راهی برای آگاهی یافتن از اینکه کدام property ها اصلاح شده اند ندارد.

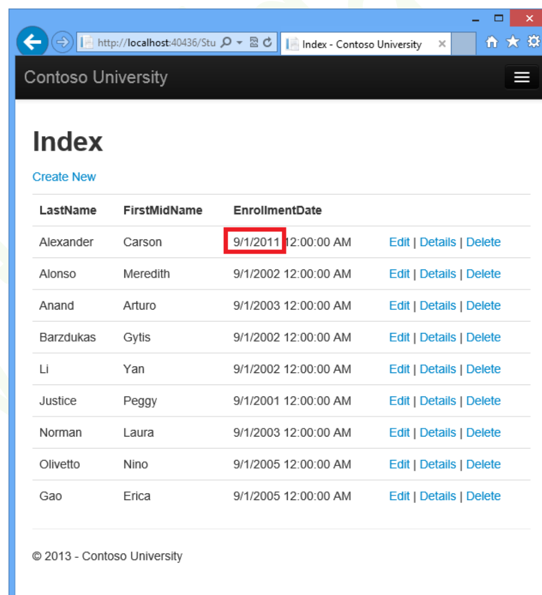
اگر می خواهید دستور SQL Update فقط آن فیلدهایی را که کاربر تغییر داده بروز رسانی کند، می توانید مقادیر اصلی را با استفاده از روشی همچون فیلدهای مخفی ذخیره نگه دارید تا زمان فراخوانی HttpPost Edit، مقادیر اولیه دردسترس باشند. پس از آن شما می توانید با استفاده از مقادیر اصلی یک موجودیت Student ایجاد کرده، متد Attach را همراه با نسخه ی اصلی موجودیت مورد نظر فراخوانی کنید، مقادیر موجودیت را بروز رسانی کرده، سپس متد SaveChanges را صدا بزنید.

HTML و کد Razor موجود در فایل Views\Student\Edit.cshtml مشابه آن کدی است که در فایل Create.cshtml مشاهده کردید، بنابراین لزومی به ایجاد تغییر نیست.

با انتخاب تب Students و کلیک بر روی لینک Edit، صفحه را اجرا کنید.



برخی از اطلاعات را تغییر داده و دکمه ی Save را کلیک کنید. داده های اصلاح شده را در صفحه ی Index مشاهده می کنید:



بروز رسانی صفحه ی Delete

داخل فایل Controllers\StudentController.cs ، کد template متعلق به متد HttpGet Delete با استفاده از متد Find موجودیت Student انتخابی را بازیابی می کند (همانگونه که در متدهای Details و Edit

نظاره گر آن بودید). اما به منظور پیاده سازی یک پیام خطای سفارشی به هنگام ناموفق بودن فراخوانی متد SaveChanges، لازم است یک قابلیت جدید به این متد و view متناظر آن اضافه کنید.

همان طور که در عملیات update و create مشاهده کردید، عملیات delete به دو action method نیاز دارد. متدی که در پاسخ درخواست GET فراخوانده می‌شود، یک view نمایش می دهد که به کاربر این فرصت را می دهد که عملیات مربوطه را پذیرفته یا لغو کند. اگر کاربر با عملیات موافقت کرده و آن را پذیرفت، یک درخواست POST شکل می گیرد (ایجاد می‌شود). پس از این اتفاق، متد HttpPost Delete صدا زده می‌شود، سپس متد مزبور عملیات delete را به طور واقعی پیاده می کند.

یک قطعه کد try-catch به متد HttpPost Delete اضافه می کنیم تا در صورت رویداد خطا حین بروز رسانی پایگاه داده، بتوان آن ها را مدیریت کرد. اگر خطایی رخ داد، HttpPost Delete متد HttpGet Delete را فرا خوانده و پارامتری به آن ارسال می کند که نشانگر رخداد خطا می باشد. متد HttpGet Delete صفحه ی تایید را بار دیگر همراه با پیام خطا به نمایش می گذارد که به کاربر فرصت می دهد عملیات را لغو کرده یا مجددا امتحان کند.

1. کد زیر را جایگزین اکشن متد HttpGet Delete کنید، این کد گزارش خطا را مدیریت می کند:

```
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);
}
```

1.

این کد یک پارامتر اختیاری می پذیرد که مشخص می کند آیا متد پس از ناموفق بودن ذخیره سازی تغییرات صدا زده شده یا خیر. هنگامی که متد HttpGet Delete بدون شکست قبلی فراخواند می شود، این پارامتر false خواهد بود. حال اگر توسط متد HttpPost Delete در پاسخ به یک خطای پایگاه داده صدا زده شود، در آن صورت پارامتر true خواهد بود و یک پیام خطا به view ارسال می شود.

2. اکشن متد HttpPost Delete (نام گذاری شده DeleteConfirmed) را با کد زیر جایگزین کنید، کار آن اجرای عملیات حذف و ضبط خطاهای مربوط به بروز رسانی پایگاه داده می باشد.

```
[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");
}
```

این کد موجودیت انتخابی را بازیابی کرده، سپس با فراخوانی متد Remove، وضعیت موجودیت را به Deleted تغییر می دهد. هنگامی که متد SaveChanges صدا زده می شود، یک دستور SQL DELETE صادر می شود. اسم اکشن متد را از DeleteConfirmed به Delete تغییر دادیم. کد ایجاد شده توسط امکان scaffolding، متد HttpPost Delete را DeleteConfirmed نام گذاری کرده تا بتواند به متد HttpPost امضای (signature) منحصربفرد اختصاص دهد. (CLR ایجاب می کند که متدهای overload شده دارای پارامترهای متفاوتی باشند.) حال که امضای متدها منحصر بفرد می باشد، می توانید با روال عادی MVC پیش رفته و اسم یکسان را برای متدهای HttpPost و HttpGet بکار ببرید.

اگر افزایش کارایی در برنامه ی بزرگ و پرحجم یک اولویت محسوب می شود، می توانید با جایگزین کردن کد زیر با خط کدهایی که توابع Find و Remove را صدا می زند، از استفاده از یک دستور غیرالزامی SQL query که سطر مورد نظر را بازیابی می کند، خودداری نمایید و درنهایت سرعت اجرا را بالا ببرید:

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


www.tahlildadeh.com

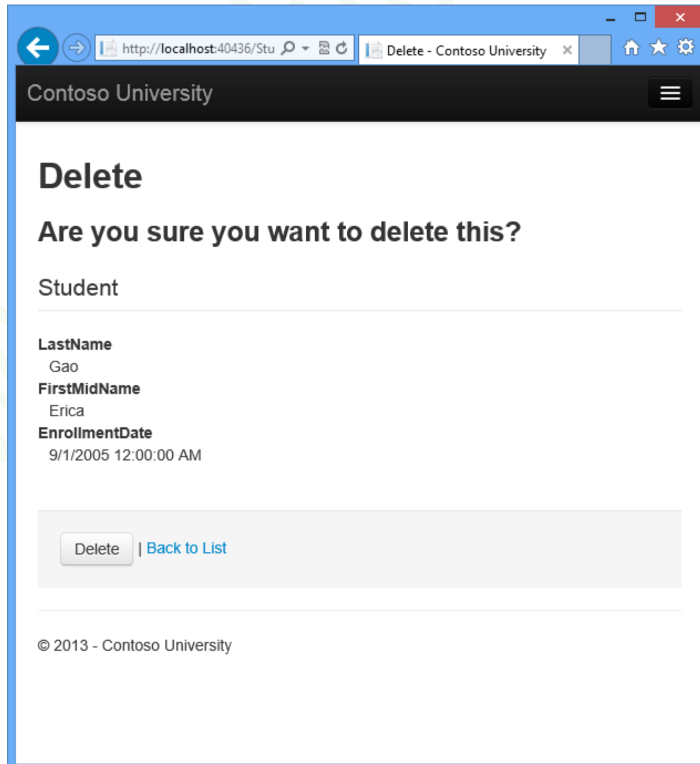
این کد تنها با استفاده از مقدار primary key، یک موجودیت Student نمونه سازی می کند، سپس وضعیت entity را به Deleted تغییر دهید. این تمام آن چیزی است که EF نیاز دارد تا موجودیت مورد نظر را به وسیله ی آن حذف کند.

همان طور که پیش تر گفته شد، متد HttpGet Delete، داده ها را حذف نمی کند. اجرای عملیات delete در پاسخ به یک درخواست GET (و اجرای عملیات edit، create و یا هر گونه عملیاتی که طی آن اطلاعات تغییر می کنند) باعث ایجاد یک خطر امنیتی می شود.

3. در فایل *Views\Student\Delete.cshtml* ، *یک پیام خطا بین سرتیترهای* h2 و h3 درج نمایید، همان گونه که در نمونه ی زیر نمایش داده شده است:

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

با انتخاب تب Students و کلیک بر روی لینک Delete، صفحه را اجرا کنید:



4. بر روی Delete کلیک کنید. صفحه ی Index بدون دانشجوی (student) حذف شده، نمایش داده می شود.

بستن اتصال به پایگاه داده

به منظور بستن اتصال به پایگاه داده و آزاد ساختن منابعی که توسط آن بکار گرفته شده، بایستی نمونه ی context را بلافاصله بعد از اینکه کارتان با آن تمام شد، آزاد (dispose) کنید. به این خاطر هم است که کد تولید شده توسط امکان scaffolding یک متد Dispose در انتهای کلاس StudentController داخل فایل *StudentController.cs فراهم می کند:*

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

کلاس پایه ی Controller از پیش و به صورت پیش فرض رابط (interface) Idisposable را پیاده سازی می کند، این کد صرفا یک override به متد Dispose(bool) اضافه می کند تا نمونه ی context را به طور صریح آزاد کند.

مدیریت تراکنش ها (Handling Transactions)

به صورت پیش فرض، EF خود تراکنش ها را پیاده سازی می کند. در شرایطی که به طور همزمان چندین سطر یا جدول را تغییر می دهید سپس SaveChanges را فراخوانی می کنید، EF به صورت خودکار کاری می کند که یا تمام تغییرات با موفقیت انجام شوند و یا هیچ یک از آن ها اعمال نشوند. چنانچه اول برخی از تغییرات انجام شوند ولی بلافاصله با خطا مواجه گردند، کلیه ی آن تغییرات اعمال شده به صورت اتوماتیک به حالت اول باز خواهد گشت.

چکیده

هم اکنون شما دارای یک برنامه ی چند صفحه ای هستید که عملیات ساده ی ایجاد، خواندن، بروز رسانی و حذف را برای موجودیت های Student انجام می دهد. در فصل حاضر با استفاده از helper های MVC، المان

های ویژه ی رابط کاربری و UI را برای فیلدهای داده ایجاد کردیم. در مبحث آموزشی بعدی، قابلیت های صفحه ی Index را با افزودن sorting (مرتب سازی) و paging (صفحه بندی) گسترش خواهیم داد.

www.tahlildadeh.com