

# Учебник по операциям создания, чтения, обновления и удаления в ASP.NET Core MVC с EF Core (2 из 10)

📅 15.03.2017 • ⌚ Время чтения: 33 мин • Соавторы 

## В этой статье

[Настройка страницы сведений](#)

[Обновление страницы Create](#)

[Обновление страницы редактирования](#)

[Обновление страницы удаления](#)

[Закрытие подключений к базе данных](#)

[Обработка транзакций](#)

[Отключение отслеживания запросов](#)

[Сводка](#)

Авторы: [Том Дайкстра](#) (Tom Dykstra) и [Рик Андерсон](#) (Rick Anderson)

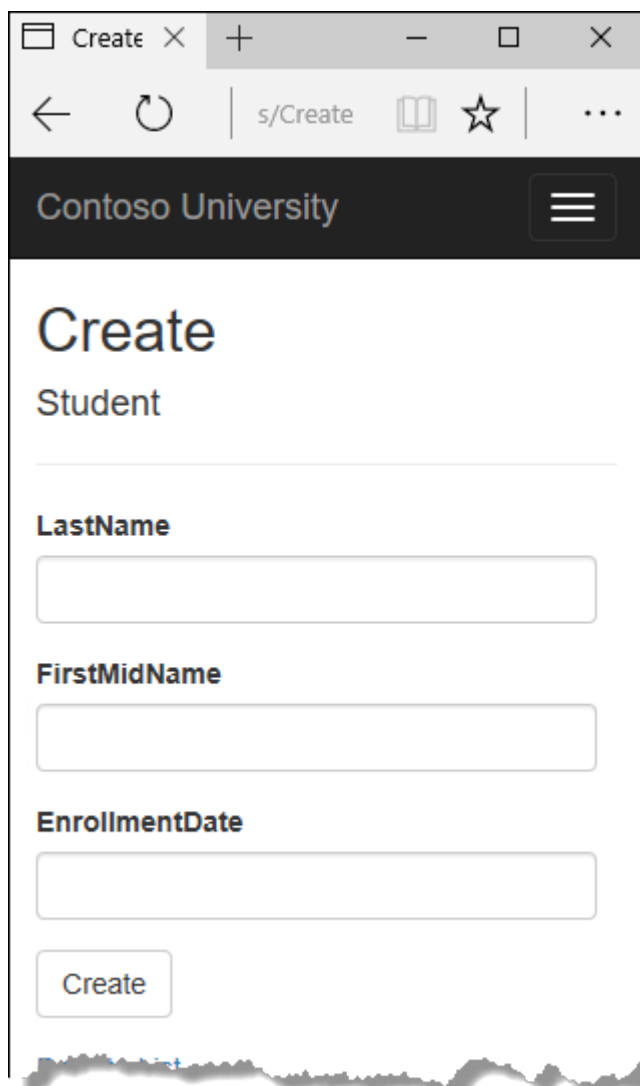
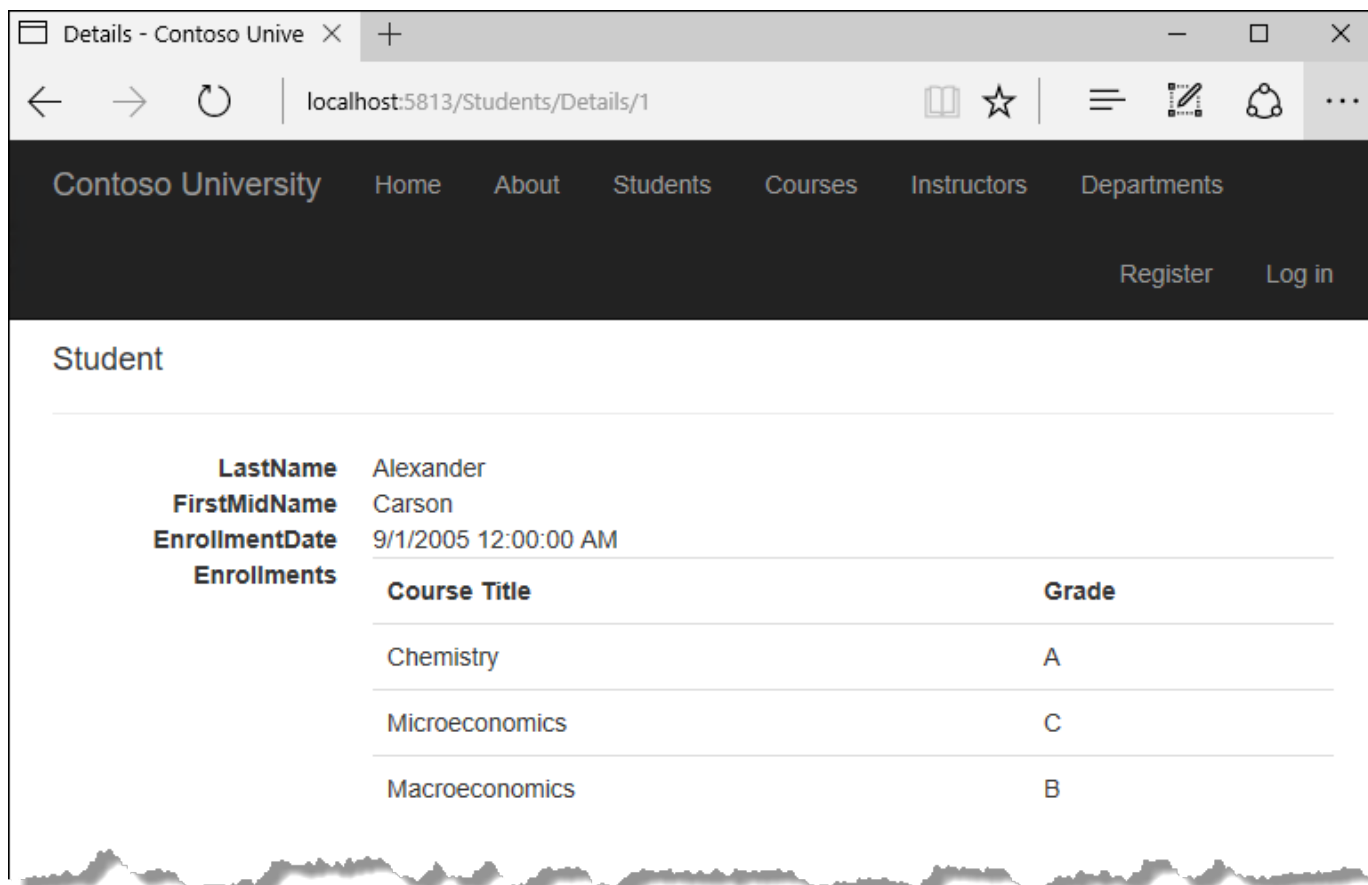
На примере учебного веб-приложения "Университет Contoso" демонстрируется процесс создания веб-приложений ASP.NET Core MVC с помощью Entity Framework Core и Visual Studio. Сведения о серии руководств см. в [первом руководстве серии](#).

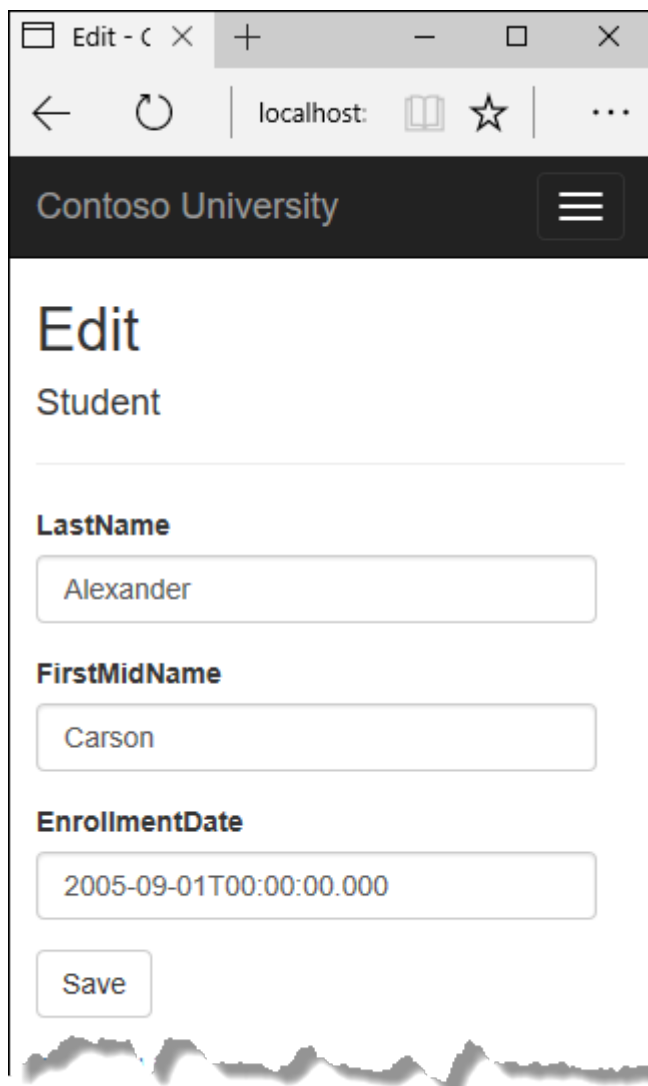
В предыдущем учебнике было создано приложение MVC, которое сохраняет и отображает данные, используя платформу Entity Framework и SQL Server LocalDB. В рамках этого учебника вы сможете ознакомиться с кодом операций CRUD (создание, чтение, обновление, удаление), который автоматически создается технологией формирования шаблонов MVC в контроллерах и представлениях, а также настроить этот код.

## 📘 Примечание

Широко распространена практика реализации шаблона репозитория, позволяющего создать уровень абстракции между контроллером и уровнем доступа к данным. Чтобы максимально упростить эти учебники и сконцентрироваться на работе с самой платформой Entity Framework, мы не используем в них репозитории. Дополнительные сведения о репозиториях на платформе EF см. в [последнем учебнике серии](#).

В рамках этого учебника вы будете работать со следующими страницами:





Edit - C X + - □ X

← ↻ | localhost: | ☆ | ...

Contoso University ☰

## Edit

### Student

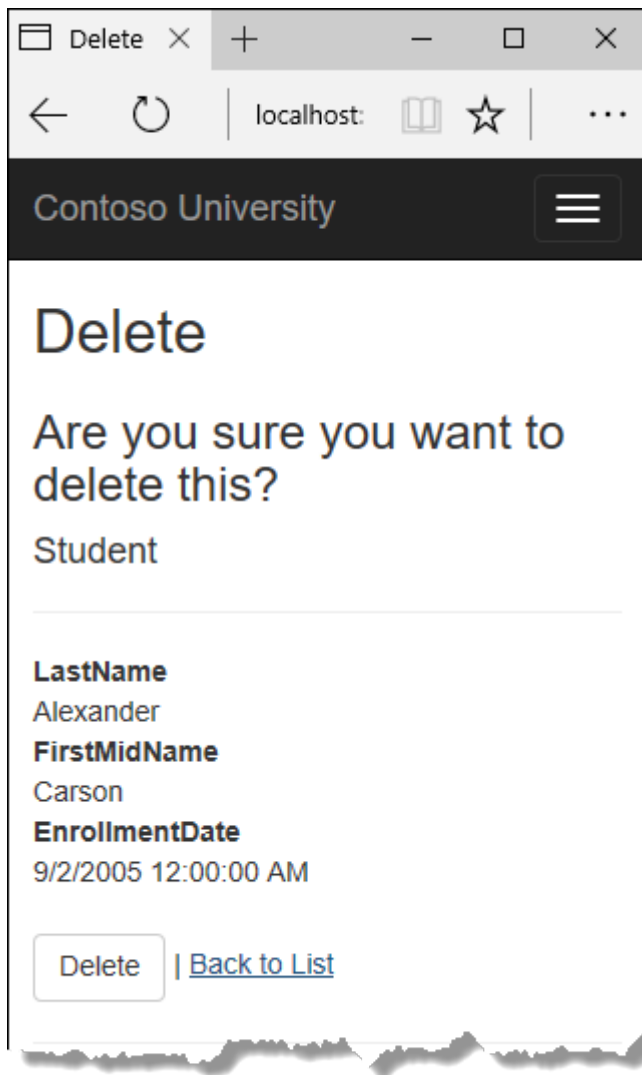
---

**LastName**

**FirstMidName**

**EnrollmentDate**

Save



## Настройка страницы сведений

В шаблонном коде страницы указателя учащихся опущено свойство `Enrollments`, поскольку оно содержит коллекцию. На странице **Details** (Сведения) содержимое коллекции отображается в таблице HTML.

В методе действия `Controllers/StudentsController.cs` для представления Details используется метод `SingleOrDefaultAsync` для извлечения одной сущности `Student`. Добавьте код, который вызывает методы `Include`, `ThenInclude` и `AsNoTracking`, как показано ниже в выделенном коде.

C#	Копировать
<pre>public async Task&lt;IActionResult&gt; Details(int? id) {     if (id == null)     {         return NotFound();     }      var student = await _context.Students</pre>	

```
.Include(s => s.Enrollments)
    .ThenInclude(e => e.Course)
.AsNoTracking()
.SingleOrDefaultAsync(m => m.ID == id);

if (student == null)
{
    return NotFound();
}

return View(student);
}
```

Методы `Include` и `ThenInclude` инструктируют контекст для загрузки свойства навигации `Student.Enrollments`, а также свойства навигации `Enrollment.Course` в пределах каждой регистрации. Дополнительные сведения об этих методах см. в учебнике [Чтение связанных данных](#).

Метод `AsNoTracking` повышает производительность в тех сценариях, где возвращаемые сущности не будут обновляться во время существования текущего контекста. Дополнительные сведения о методе `AsNoTracking` приводятся в конце этого учебника.

## Данные маршрута

Значение ключа, которое передается в метод `Details`, поступает из *данных маршрута*. Данные маршрута обнаруживаются связывателем модели в сегменте URL-адреса. Например, маршрут по умолчанию задает сегменты контроллера, действия и идентификатора:

C#

 Копировать


```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

В следующем URL-адресе маршрут по умолчанию сопоставляет контроллер `Instructor`, действие `Index` и идентификатор `1`, которые принимаются в качестве значений данных маршрута.


 Копировать

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


Последняя часть URL-адреса ("?courseID=2021") представляет значение строки запроса. Связыватель модели также будет передавать значение идентификатора в метод `Details` в параметр `id`, если вы передаете его в качестве значения строки запроса:

	 Копировать
<pre>http://localhost:1230/Instructor/Index?id=1&amp;CourseID=2021</pre>	


На странице `Index` URL-адреса гиперссылок создаются с помощью инструкций вспомогательной функции тегов в представлении Razor. В следующем коде Razor параметр `id` соответствует маршруту по умолчанию, поэтому к данным маршрута добавляется `id`.

HTML	 Копировать
<pre>&lt;a asp-action="Edit" asp-route-id="@item.ID"&gt;Edit&lt;/a&gt;</pre>	


Если `item.ID` равен 6, создается следующий код HTML:

HTML	 Копировать
<pre>&lt;a href="/Students/Edit/6"&gt;Edit&lt;/a&gt;</pre>	

В следующем коде Razor `studentID` не соответствует параметру в маршруте по умолчанию и добавляется в качестве строки запроса.

HTML	 Копировать
<pre>&lt;a asp-action="Edit" asp-route-studentID="@item.ID"&gt;Edit&lt;/a&gt;</pre>	

Если `item.ID` равен 6, создается следующий код HTML:


HTML	 Копировать
<pre>&lt;a href="/Students/Edit?studentID=6"&gt;Edit&lt;/a&gt;</pre>	

Дополнительные сведения о вспомогательных функциях тегов см. в разделе [Вспомогательные функции тегов в ASP.NET Core](#).


## Добавление регистраций в представление сведений

Откройте `Views/Students/Details.cshtml`. Каждое поле отображается с помощью вспомогательных функций `DisplayNameFor` и `DisplayFor`, как показано в следующем

примере:

HTML	 Копировать
<pre>&lt;dt&gt;     @Html.DisplayNameFor(model =&gt; model.LastName) &lt;/dt&gt; &lt;dd&gt;     @Html.DisplayFor(model =&gt; model.LastName) &lt;/dd&gt;</pre>	

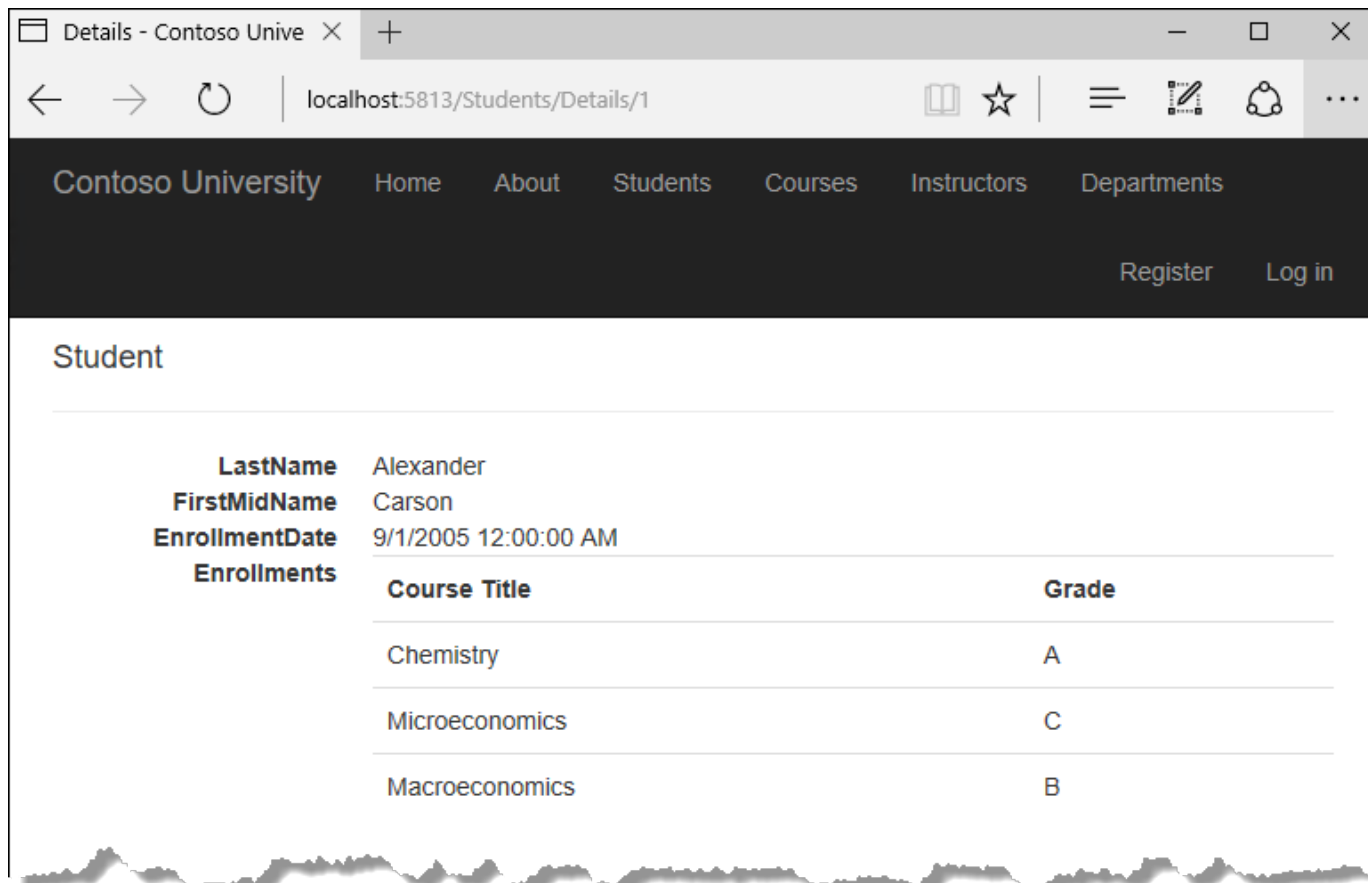
После последнего поля и непосредственно перед закрывающим тегом `</dl>` добавьте следующий код, чтобы отобразить список регистраций:

HTML	 Копировать
<pre>&lt;dt&gt;     @Html.DisplayNameFor(model =&gt; model.Enrollments) &lt;/dt&gt; &lt;dd&gt;     &lt;table class="table"&gt;         &lt;tr&gt;             &lt;th&gt;Course Title&lt;/th&gt;             &lt;th&gt;Grade&lt;/th&gt;         &lt;/tr&gt;         @foreach (var item in Model.Enrollments)         {             &lt;tr&gt;                 &lt;td&gt;                     @Html.DisplayFor(modelItem =&gt; item.Course.Title)                 &lt;/td&gt;                 &lt;td&gt;                     @Html.DisplayFor(modelItem =&gt; item.Grade)                 &lt;/td&gt;             &lt;/tr&gt;         }     &lt;/table&gt; &lt;/dd&gt;</pre>	

Если после вставки кода нарушаются отступы в нем, нажмите клавиши CTRL-K-D, чтобы исправить это.

Этот код циклически обрабатывает сущности в свойстве навигации `Enrollments`. Для каждой регистрации он отображает название курса и оценку. Название курса извлекается из сущности `Course`, которая хранится в свойстве навигации `Course` сущности `Enrollments`.

Запустите приложение, выберите вкладку **Students** (Учащиеся) и щелкните ссылку **Details** (Сведения) для учащегося. Откроется список курсов и оценок для выбранного учащегося:



## Обновление страницы Create

В файле *StudentsController.cs* измените метод `HttpPost Create`, добавив в него блок `try-catch` и удалив идентификатор из атрибута `Bind`.

```
C# Копировать  
  
[HttpPost]  
[ValidateAntiForgeryToken]  
public async Task<IActionResult> Create(  
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)  
{  
    try  
    {  
        if (ModelState.IsValid)  
        {  
            _context.Add(student);  
            await _context.SaveChangesAsync();  
            return RedirectToAction(nameof(Index));  
        }  
    }  
    catch (DbUpdateException /* ex */)   
    {  
        //Log the error (uncomment ex variable name and write a log.  
        ModelState.AddModelError("", "Unable to save changes. " +  
            "Try again, and if the problem persists " +  
            "see your system administrator.");  
    }  
}
```



```
    }  
    return View(student);  
}
```

Этот код добавляет сущность `Student`, созданную связывателем модели ASP.NET MVC, в набор сущностей `Students`, после чего сохраняет изменения в базе данных. (Связыватель модели использует функциональные возможности ASP.NET MVC, упрощая работу с данными, которые вы предоставляете в форме. Связыватель модели преобразует значения из отправленной формы в типы CLR и передает их в виде параметров в метод действия. В этом случае связыватель модели создает сущность `Student`, используя значения свойств из коллекции `Form`.)

Атрибут `ID` удаляется из атрибута `Bind` в связи с тем, что он содержит значение первичного ключа, которое будет автоматически устанавливаться SQL Server при вставке строки. Значение `ID` не задается на основе введенных пользователем данных.


Помимо атрибута `Bind`, в шаблонном коде изменяется только блок `try-catch`. Если во время сохранения изменений перехватывается исключение, производное от `DbUpdateException`, отображается сообщение об общей ошибке. Исключения `DbUpdateException` иногда связаны с внешними факторами, а не с ошибкой при программировании приложения, поэтому рекомендуется попробовать повторить выполненные действия снова. В этом примере такое поведение не реализовано, однако в рабочем приложении, как правило, исключения заносятся в журнал. Дополнительные сведения см. в разделе **Ведение журналов для анализа** статьи [Мониторинг и телеметрия \(построение реальных облачных приложений для Azure\)](#).

Атрибут `ValidateAntiForgeryToken` позволяет предотвратить атаки с подделкой межсайтовых запросов. Токен автоматически вставляется в представление с помощью [FormTagHelper](#) и включается при отправке формы пользователем. Токен проверяется по атрибуту `ValidateAntiForgeryToken`. Дополнительные сведения об атаках с подделкой межсайтовых запросов см. в разделе [Защита от подделки запросов](#).

## Примечание о безопасности в связи с атаками чрезмерной передачи данных

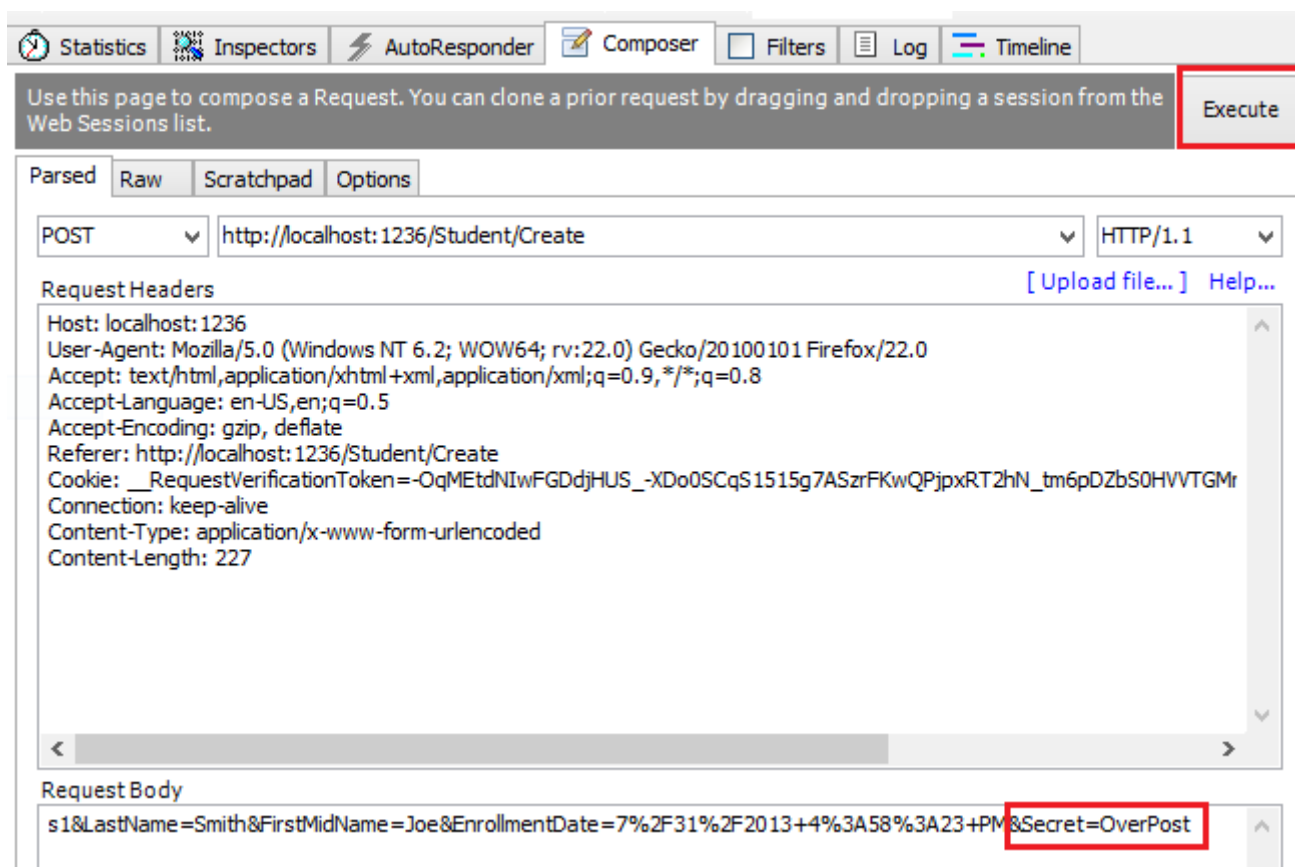
Атрибут `Bind`, который включается шаблонным кодом в метод `Create`, является одним из способов защиты от чрезмерной передачи данных в сценариях создания. Допустим, сущность `Student` включает свойство `Secret`, которое не требуется устанавливать на этой веб-странице.

C#

 Копировать

```
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; }
}
```

Даже если на веб-странице отсутствует поле `Secret`, злоумышленник может использовать такие средства, как Fiddler, или собственный код JavaScript, для отправки значения формы `Secret`. Если отсутствует атрибут `bind`, ограничивающий поля, которые связыватель модели использует при создании экземпляра `Student`, связыватель модели выберет это значение формы `Secret` и использует его для создания экземпляра сущности `Student`. Таким образом, какое бы значение ни задал злоумышленник для поля `Secret`, оно будет обновлено в базе данных. На следующем рисунке показано средство Fiddler, с помощью которого в отправленные значения формы добавляется поле `Secret` (со значением "OverPost").



После этого значение "OverPost" будет успешно добавлено в свойство `Secret` вставленной строки, хотя вы не разрешали установку этого свойства на веб-странице.

Чтобы предотвратить чрезмерную передачу данных в сценариях редактирования, можно сначала считать сущность из базы данных и затем вызвать метод `TryUpdateModel`, передав в

него список явно разрешенных свойств. Именно этот метод используется в этих учебниках.

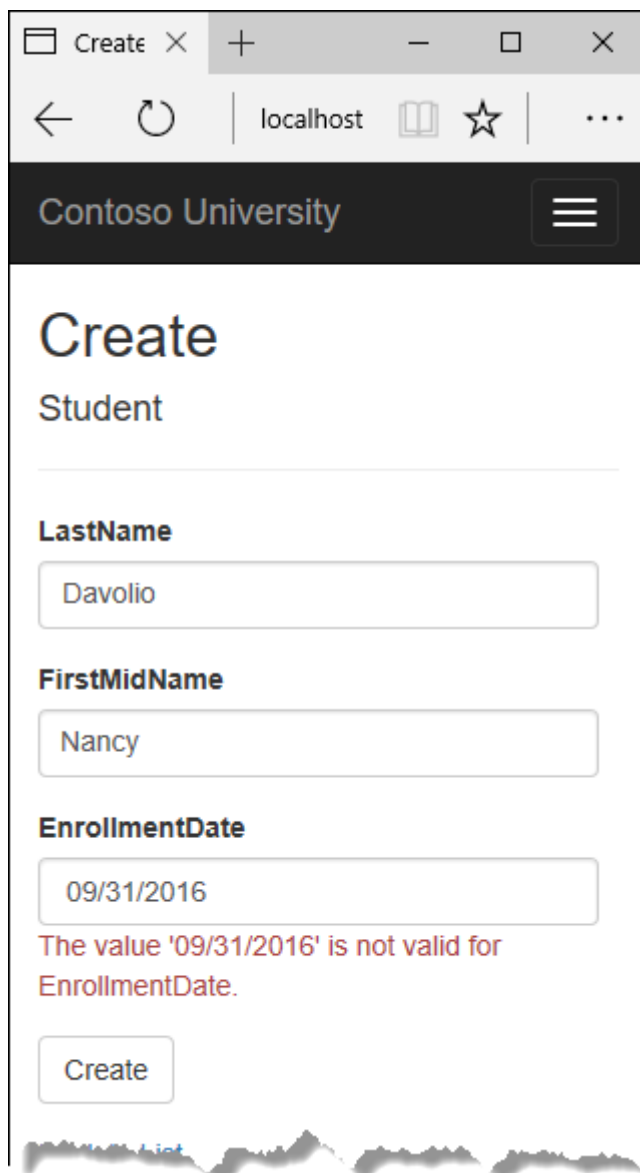
Кроме того, многие разработчики предпочитают для защиты от чрезмерной передачи данных использовать модели представлений вместо классов сущностей с привязкой моделей. Включайте только те свойства, которые требуется обновлять в модели представления. После завершения работы связывателя модели MVC скопируйте свойства модели представления в экземпляр сущности, например с помощью такого средства, как AutoMapper. С помощью `_context.Entry` в экземпляре сущности задайте для него состояние `Unchanged`, после чего присвойте значение `true` атрибуту `Property("PropertyName").IsModified` для каждого свойства, которое включается в модель представления. Этот метод подходит для сценариев редактирования и создания.

## Проверка страницы создания

Для каждого поля в коде в *Views/Students/Create.cshtml* используются вспомогательные функции тегов `label`, `input` и `span` (для сообщений о проверке).

Запустите приложение, выберите вкладку **Students** (Учащиеся) и щелкните **Create New** (Создать).

Введите имена и даты. Если браузер допускает это, попробуйте ввести недопустимую дату. (В некоторых браузерах принудительно используется управляющий элемент выбора даты.) Щелкните **Create** (Создать), чтобы просмотреть сообщение об ошибке.



Create x + - □ x

← ↻ | localhost | ☆ | ...

Contoso University ☰

# Create

## Student

**LastName**

**FirstMidName**

**EnrollmentDate**

The value '09/31/2016' is not valid for EnrollmentDate.

Эта проверка по умолчанию выполняется на стороне сервера. Позднее в учебнике вы узнаете, как добавлять атрибуты, которые будут создавать код для проверки на стороне клиента. В выделенном ниже коде демонстрируется проверка модели в методе `Create`.

```
C# Копировать
```

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException /* ex */)
    {
    }
}
```

```
{  
    //Log the error (uncomment ex variable name and write a log.  
    ModelState.AddModelError("", "Unable to save changes. " +  
        "Try again, and if the problem persists " +  
        "see your system administrator.");  
}  
return View(student);  
}
```

Измените дату на допустимую и щелкните **Create** (Создать), чтобы добавить нового учащегося на страницу **Index** (Указатель).

## Обновление страницы редактирования

В файле *StudentController.cs* метод `HttpGet Edit` (метод без атрибута `HttpPost`) использует метод `SingleOrDefaultAsync` для извлечения выбранной сущности *Student*, как показано в методе `Details`. Изменять этот метод не нужно.

### Рекомендуемый код метода `HttpPost Edit`: чтение и обновление

Замените метод действия `HttpPost Edit` следующим кодом.

C#

 Копировать

```
[HttpPost, ActionName("Edit")]  
[ValidateAntiForgeryToken]  
public async Task<IActionResult> EditPost(int? id)  
{  
    if (id == null)  
    {  
        return NotFound();  
    }  
    var studentToUpdate = await _context.Students.SingleOrDefault(s => s.ID == id)  
    if (await TryUpdateModelAsync<Student>(  
        studentToUpdate,  
        "",  
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))  
    {  
        try  
        {  
            await _context.SaveChangesAsync();  
            return RedirectToAction(nameof(Index));  
        }  
        catch (DbUpdateException /* ex */)   
        {  
            //Log the error (uncomment ex variable name and write a log.)  
            ModelState.AddModelError("", "Unable to save changes. " +  
                "Try again, and if the problem persists, " +  
                "see your system administrator.");  
        }  
    }  
}
```

```
    }  
    }  
    return View(studentToUpdate);  
}
```

Благодаря этому изменению реализуются рекомендации по безопасности, позволяющие предотвратить чрезмерную отправку данных. Шаблон создал атрибут `Bind` и добавил сущность, созданную связывателем модели, в набор сущностей с флагом `Modified`. В большинстве сценариев не рекомендуется использовать этот код, поскольку атрибут `Bind` очищает любые ранее существовавшие данные в полях, которые не перечислены в параметре `Include`.

Новый код считывает существующую сущность и вызывает метод `TryUpdateModel` для обновления полей в извлеченной сущности [на основании данных, введенных пользователем в отправленной форме](#). Технология автоматического отслеживания изменений платформы Entity Framework устанавливает флаг `Modified` для полей, которые были изменены на основе введенных в форму данных. При вызове метода `SaveChanges` платформа Entity Framework создает инструкции SQL для обновления строки базы данных. Конфликты параллелизма игнорируются, а в базе данных обновляются только те столбцы таблицы, которые были обновлены пользователем. (Порядок обработки конфликтов параллелизма будет показан позднее в учебнике.)

Чтобы предотвратить чрезмерную передачу данных, рекомендуется добавить поля, которые требуется обновлять на странице **Edit**, в список разрешенных в параметрах `TryUpdateModel`. (Пустая строка перед списком полей в списке параметров предназначена для префикса, который используется с именами полей формы.) На данный момент другие поля не защищаются. Если включить в список поля, которые должен привязывать связыватель модели, это позволяет гарантировать, что при добавлении полей в модель данных в будущем они будут автоматически защищаться до тех пор, пока вы явно не добавите их сюда.

В результате этих изменений сигнатура метода `HttpPost Edit` будет совпадать с методом `HttpGet Edit`. Таким образом, вы просто переименовали метод `EditPost`.

## Альтернативный код метода `HttpPost Edit`: создание и присоединение

Рекомендуемый код метода `HttpPost` гарантирует обновление только измененных столбцов и сохраняет данные в свойствах, которые не требуется включать в привязку моделей. Тем не менее при подходе с предварительным считыванием дополнительно выполняется чтение из базы данных, в результате чего код может усложняться для обработки конфликтов параллелизма. В качестве альтернативы можно присоединить сущность,

созданную связывателем модели, к контексту EF и пометить ее как измененную. (Не добавляйте этот код в проект, поскольку он показан исключительно как пример альтернативного подхода.)

C# Копировать

```
public async Task<IActionResult> Edit(int id, [Bind("ID,EnrollmentDate,FirstMidName,LastMidName")] Student student)
{
    if (id != student.ID)
    {
        return NotFound();
    }
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(student);
}
```

Этот подход можно использовать в тех случаях, когда пользовательский интерфейс веб-страницы включает все поля сущности и может обновлять любые из них.

Шаблонный код использует подход с созданием и присоединением, однако лишь перехватывает исключения `DbUpdateConcurrencyException` и возвращает коды ошибок 404. В показанном примере перехватываются любые исключения обновления базы данных и отображается сообщение об ошибке.

## Состояния сущностей

Контекст базы данных отслеживает состояние синхронизации сущностей в памяти с соответствующими им строками в базе данных. Данные отслеживания определяют, что происходит при вызове метода `SaveChanges`. Например, при передаче новой сущности в метод `Add` ей присваивается состояние `Added`. При последующем вызове метода `SaveChanges` контекст базы данных выполняет команду SQL INSERT.

Возможны следующие состояния сущности:

- `Added` . Сущность еще не существует в базе данных. Метод `SaveChanges` выполняет инструкцию INSERT.
- `Unchanged` . С этой сущностью не нужно выполнять никакие действия с помощью метода `SaveChanges` . Это начальный статус сущности, который она имеет при чтении из базы данных.
- `Modified` . Были изменены значения некоторых или всех свойств сущности. Метод `SaveChanges` выполняет инструкцию UPDATE.
- `Deleted` . Сущность отмечена для удаления. Метод `SaveChanges` выполняет инструкцию DELETE.
- `Detached` . Сущность не отслеживается контекстом базы данных.

В классическом приложении изменения состояния обычно осуществляются автоматически. Например, вы можете считать сущность и изменить значения некоторых ее свойств. В этом случае состояние сущности автоматически изменится на `Modified` . Если затем вызвать метод `SaveChanges` , платформа Entity Framework выполнит инструкцию SQL UPDATE, которая обновит только фактически измененные свойства.

В веб-приложении объект `DbContext` , который изначально считывает сущность и отображает ее данные для редактирования, ликвидируется после отрисовки страницы. При вызове метода действия `HttpPost Edit` выполняется новый веб-запрос и создается новый экземпляр `DbContext` . Если повторно считать сущность в этот новый контекст, таким образом будет смоделирована обработка в классическом приложении.

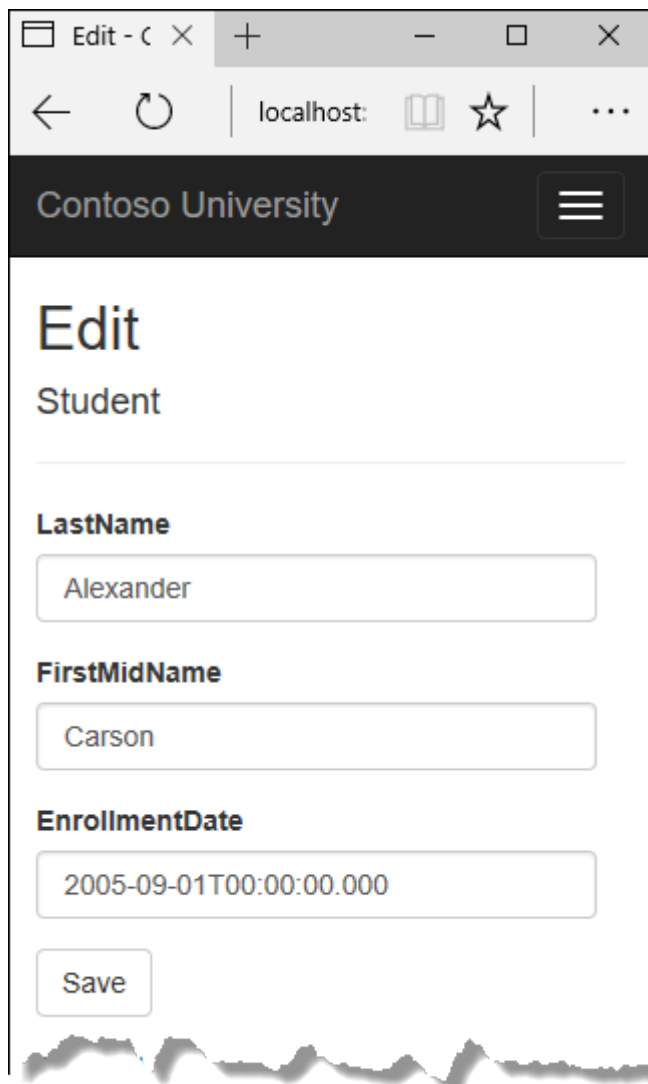
Однако если выполнять дополнительную операцию чтения не требуется, необходимо использовать объект сущности, созданный связывателем модели. Для этого проще всего присвоить сущности состояние `Modified`, как это сделано в показанном ранее альтернативном методе `HttpPost Edit`. При последующем вызове метода `SaveChanges` платформа Entity Framework обновляет все столбцы в строке базы данных, поскольку у контекста нет возможности определить, какие свойства были изменены.

Если вы не хотите сначала выполнять чтение, но вам нужно, чтобы инструкция SQL UPDATE обновляла только те поля, которые пользователь фактически изменяет, код будет выглядеть более сложным. Вам необходимо каким-либо образом сохранить исходные значения (например, используя скрытые поля), чтобы они были доступны при вызове метода `HttpPost Edit` . Затем вы можете создать сущность `Student`, используя исходные значения, вызвать метод `Attach` с исходной версией этой сущности, обновить значения сущности и вызвать метод `SaveChanges` .



## Проверка страницы редактирования

Запустите приложение, выберите вкладку **Students** (Учащиеся) и щелкните гиперссылку **Edit** (Изменить).



Измените определенные данные и нажмите кнопку **Save** (Сохранить). Откроется страница **Index** (Указатель), на которой будут представлены измененные данные.

## Обновление страницы удаления

В файле *StudentController.cs* в коде шаблона для метода `HttpGet` `Delete` используется метод `SingleOrDefaultAsync` для извлечения выбранной сущности `Student`, как показано в методах `Details` и `Edit`. Тем не менее, чтобы реализовать настраиваемое сообщение об ошибке при сбое вызова метода `SaveChanges`, необходимо добавить некоторые функции в этот метод и соответствующее ему представление.

Как и в случае с операциями обновления и создания, операции удаления требуют двух методов действия. Метод, вызываемый в ответ на запрос `GET`, отображает представление, в котором пользователь может подтвердить или отменить операцию удаления. Если

пользователь подтверждает ее, создается запрос POST. В этом случае вызывается метод `HttpPost Delete`, который фактически выполняет операцию удаления.

Для обработки ошибок, которые могут произойти при обновлении базы данных, следует добавить в метод `HttpPost Delete` блок try-catch. В случае ошибки метод `HttpPost Delete` вызывает метод `HttpGet Delete`, передавая в него параметр, указывающий на состояние ошибки. Метод `HttpGet Delete` повторно отображает страницу подтверждения и сообщение об ошибке, предлагая пользователю отменить операцию или повторить ее еще раз.

Замените метод действия `HttpGet Delete` следующим кодом, в котором реализуется управление сообщениями об ошибках.

C#

 Копировать

```
public async Task<IActionResult> Delete(int? id, bool? saveChangesError = false)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return NotFound();
    }

    if (saveChangesError.GetValueOrDefault())
    {
        ViewData["ErrorMessage"] =
            "Delete failed. Try again, and if the problem persists " +
            "see your system administrator.";
    }

    return View(student);
}
```

Этот код принимает необязательный параметр, который указывает, был ли метод вызван после сбоя при сохранении изменений. Если перед вызовом метода `HttpGet Delete` не произошел сбой, этот параметр будет иметь значение `false`. Если он вызывается методом `HttpPost Delete` в ответ на ошибку при обновлении базы данных, этот параметр будет иметь значение `true`, а в представление передается сообщение об ошибке.

## Подход с предварительным чтением для метода `HttpPost Delete`

Замените метод действия `HttpPost Delete` (имеет имя `DeleteConfirmed`) следующим кодом, в котором выполняется фактическая операция удаления и перехватываются любые ошибки при обновлении базы данных.

C# Копировать

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var student = await _context.Students
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return RedirectToAction(nameof(Index));
    }

    try
    {
        _context.Students.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof>Delete), new { id = id, saveChangesError = true
    }
}
```

Этот код извлекает выбранную сущность и вызывает метод `Remove`, чтобы присвоить ей состояние `Deleted`. При вызове метода `SaveChanges` создается инструкция SQL DELETE.

## Подход с созданием и присоединением для метода `HttpPost Delete`

Если требуется обеспечить максимальную производительность крупного приложения, можно избежать создания ненужных запросов SQL. Для этого можно создать экземпляр сущности `Student`, используя только значение первичного ключа, и затем присвоить этой сущности состояние `Deleted`. Это все, что платформе Entity Framework необходимо для удаления сущности. (Не используйте этот код в проекте. Он показан здесь исключительно в качестве примера.)

C# Копировать


```
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public async Task<IActionResult> DeleteConfirmed(int id)
{
    try
    {
        Student studentToDelete = new Student() { ID = id };
        _context.Entry(studentToDelete).State = EntityState.Deleted;
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id, saveChangesError = true });
    }
}
```

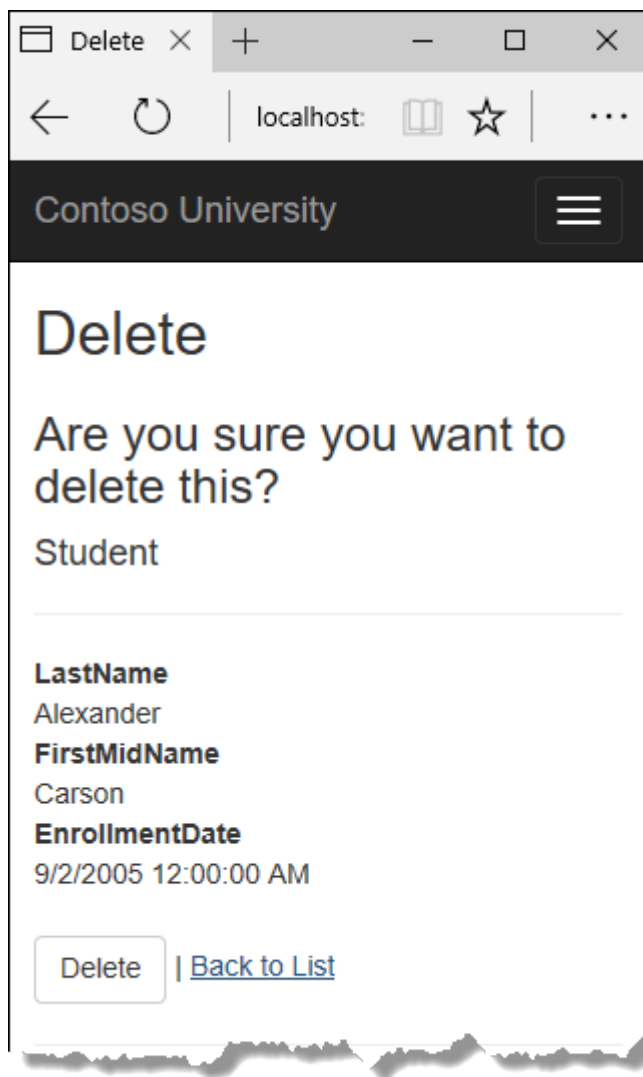
Если также требуется удалить связанные с сущностью данные, убедитесь, что в базе данных настроено каскадное удаление. При таком подходе к удалению сущности платформе EF может быть неизвестно о наличии связанных сущностей, которые требуется удалить.

## Обновление представления удаления

В файле *Views/Student/Delete.cshtml* добавьте сообщение об ошибке между заголовками h2 и h3, как показано в следующем примере:

HTML	 Копировать
<pre>&lt;h2&gt;Delete&lt;/h2&gt; &lt;p class="text-danger"&gt;@ViewData["ErrorMessage"]&lt;/p&gt; &lt;h3&gt;Are you sure you want to delete this?&lt;/h3&gt;</pre>	

Запустите приложение, выберите вкладку **Students** (Учащиеся) и щелкните гиперссылку **Delete** (Удалить):



Щелкните **Delete** (Удалить). Отображается страница Index (Указатель), на которой удаленный учащийся будет отсутствовать. (В учебнике, посвященном параллелизму, приводится пример кода обработки ошибок.)

## Закрытие подключений к базе данных

Чтобы высвободить ресурсы, используемые подключением к базе данных, необходимо как можно скорее ликвидировать экземпляр контекста после завершения работы с ним. Эта задача реализуется с помощью встроенной в ASP.NET Core технологии [внедрения зависимостей](#).

В файле *Startup.cs* вызывается [метод расширения AddDbContext](#), чтобы подготовить класс `DbContext` в контейнере ASP.NET DI. Этот метод по умолчанию устанавливает время существования службы `Scoped`. Значение `Scoped` указывает, что срок существования объекта контекста соответствует сроку существования веб-запроса. Таким образом, по завершении веб-запроса автоматически будет вызываться метод `Dispose`.

## Обработка транзакций

По умолчанию платформа Entity Framework реализует транзакции неявно. В сценариях, когда вы вносите изменения в несколько строк или таблиц и затем вызываете метод `SaveChanges`, платформа Entity Framework автоматически гарантирует, что одновременно все изменения либо выполняются успешно, либо завершаются неудачно. Если ошибка происходит после того, как были выполнены некоторые изменения, эти изменения автоматически откатываются. Если вам требуется дополнительный контроль, например в сценариях с операциями, выполняемыми в транзакции вне платформы Entity Framework, ознакомьтесь с разделом [Транзакции](#).

## Отключение отслеживания запросов

Когда контекст базы данных извлекает строки таблицы и создает представляющие их объекты сущностей, по умолчанию отслеживается состояние синхронизации сущностей в памяти с содержимым базы данных. При обновлении сущности данные в памяти выступают в роли кэша. В веб-приложении такое кэширование часто не нужно, поскольку экземпляры контекста, как правило, существуют недолго (для каждого запроса создается и ликвидируется собственный экземпляр), и контекст, считывающий сущность, как правило, ликвидируется до того, как сущность будет использована снова.

Чтобы отключить отслеживание объектов сущностей в памяти, вызовите метод

`AsNoTracking`. Как правило, это требуется в следующих сценариях:

- В течение срока существования контекста не требуется обновлять сущности, и не нужно, чтобы платформа EF [автоматически загружала свойства навигации с сущностями, извлекаемыми с помощью отдельных запросов](#). Эти условия часто выполняются в методах действия контроллера `HttpGet`.
- Выполняется запрос, который извлекает большой объем данных, и при этом обновляется только небольшая часть возвращаемых данных. Для повышения эффективности можно отключить отслеживание для больших запросов и выполнить запрос позднее для нескольких обновляемых сущностей.
- Необходимо присоединить запрос для его обновления, однако ранее та же сущность уже была извлечена для других целей. Поскольку сущность уже отслеживается контекстом базы данных, присоединить сущность, которую требуется изменить, нельзя. Одним из решений в такой ситуации является вызов метода `AsNoTracking` для предшествующего запроса.

Дополнительные сведения см. в разделе [Работа с отслеживанием и и без него](#).

## Сводка

Теперь у вас есть полный набор страниц, которые реализуют простые операции CRUD (создание, чтение, обновление и удаление) для сущностей Student. В следующем учебнике мы добавим на страницу **Index** (Указатель) функции добавления, сортировки, фильтрации и разбиения на страницы.

---

[Назад](#) [Вперед](#)