

Начало работы с ASP.NET Core MVC и Entity Framework Core с использованием Visual Studio (1 из 10)

📅 15.03.2017 • ⌚ Время чтения: 35 мин • Соавторы 

В этой статье

[Предварительные требования](#)

[Устранение неполадок](#)

[Веб-приложение университета Contoso](#)

[Создание веб-приложения ASP.NET Core MVC](#)

[Настройка стиля сайта](#)

[Пакеты Entity Framework Core NuGet](#)

[Создание модели данных](#)

[Создание контекста базы данных](#)

[Регистрация контекста с помощью внедрения зависимостей](#)

[Добавление кода для инициализации базы с использованием тестовых данных](#)

[Создание контроллера и представлений](#)

[Просмотр базы данных](#)

[Соглашения](#)

[Асинхронный код](#)

[Сводка](#)

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

Версия учебника по Razor Pages доступна [здесь](#). Версия учебника по Razor Pages проще в изучении, и в ней рассматриваются дополнительные возможности EF. Мы рекомендуем руководствоваться [версией этого учебника для Razor Pages](#).

На примере учебного веб-приложения "Университет Contoso" демонстрируется процесс создания веб-приложений ASP.NET Core 2.0 MVC с помощью Entity Framework (EF) Core 2.0 и Visual Studio 2017.

В этом примере приложения реализуется веб-сайт вымышленного университета Contoso. На нем предусмотрены различные функции, в том числе прием учащихся, создание курсов и назначение преподавателей. Это первый учебник из серии, в котором с самого начала описывается построение примера приложения для университета Contoso.

[Скачайте или ознакомьтесь с готовым приложением.](#)

EF Core 2.0 — это последняя версия платформы EF, однако в ней реализованы еще не все возможности EF 6.x. Дополнительные сведения о выборе между платформами EF 6.x и EF Core см. в [этой статье](#). Если вы используете платформу EF 6.x, ознакомьтесь с [предыдущей версией этой серии учебников](#).

Примечание

- Версия этого учебника для ASP.NET Core 1.1 и VS 2017 с обновлением 2 в формате PDF доступна [здесь](#).
- Версию этого учебника для Visual Studio 2015 см. в статье [документации по ASP.NET Core для VS 2015 в формате PDF](#).

Предварительные требования

Install the following:

- [.NET Core 2.0.0 SDK](#) or later.
- [Visual Studio 2017](#) version 15.3 or later with the **ASP.NET and web development** workload.

Устранение неполадок

Если вы столкнулись с проблемами, для их решения можно попробовать сравнить свой код с кодом [готового проекта](#). Список распространенных ошибок и способы их устранения см. в [разделе "Устранение неполадок" последнего руководства серии](#). Если вам не удалось найти нужную информацию, вы можете задать вопрос на сайте StackOverflow.com в разделах, посвященных [ASP.NET Core](#) или [EF Core](#).

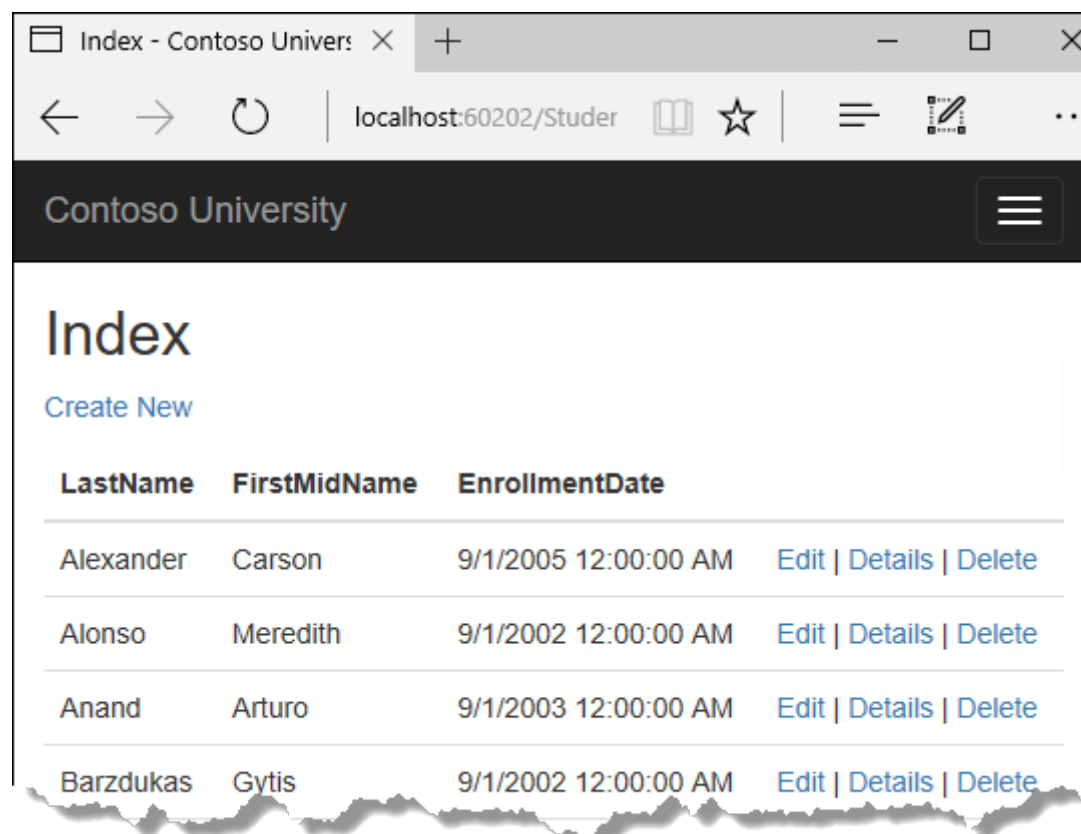
Совет

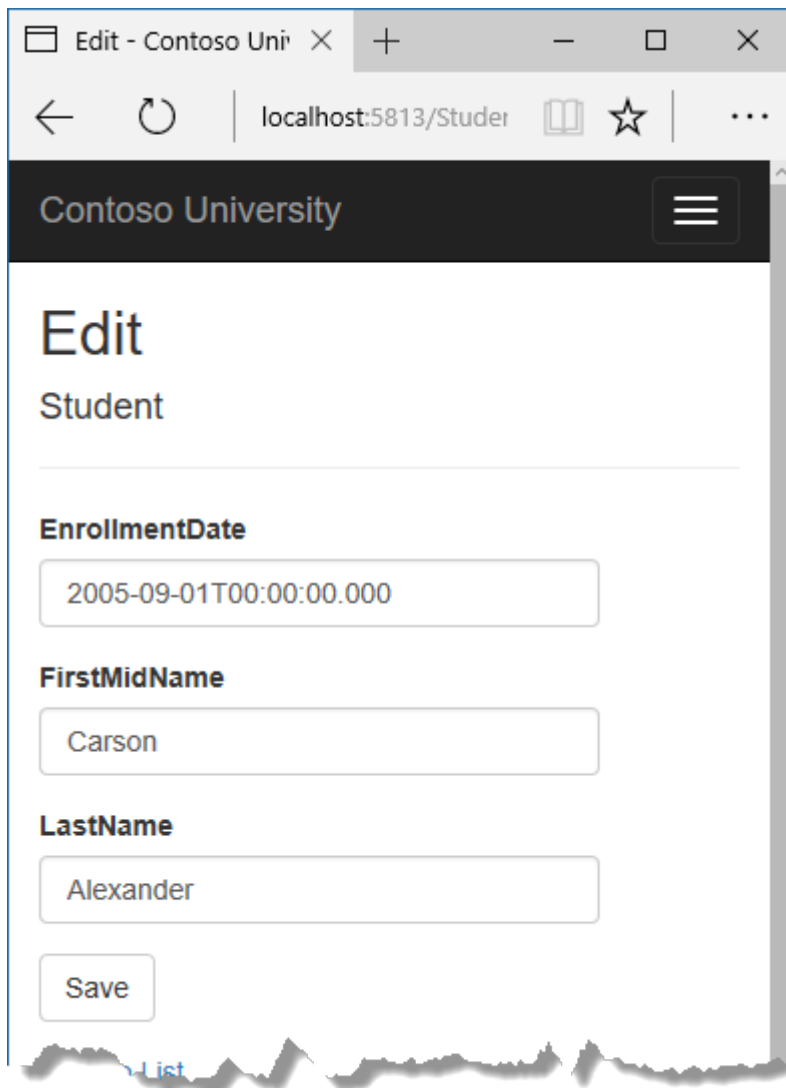
Эта серия включает в себя 10 учебников, содержание каждого из которых базируется на предыдущих учебниках. После успешного завершения каждого руководства рекомендуется сохранять копию проекта. Таким образом, при возникновении проблем вы сможете вернуться к предыдущему учебнику, а не к началу серии.

Веб-приложение университета Contoso

В рамках этих учебников вы будете создавать приложение, которое представляет собой простой веб-сайт университета.

Пользователи приложения могут просматривать и обновлять сведения об учащихся, курсах и преподавателях. Будет создано несколько экранов.



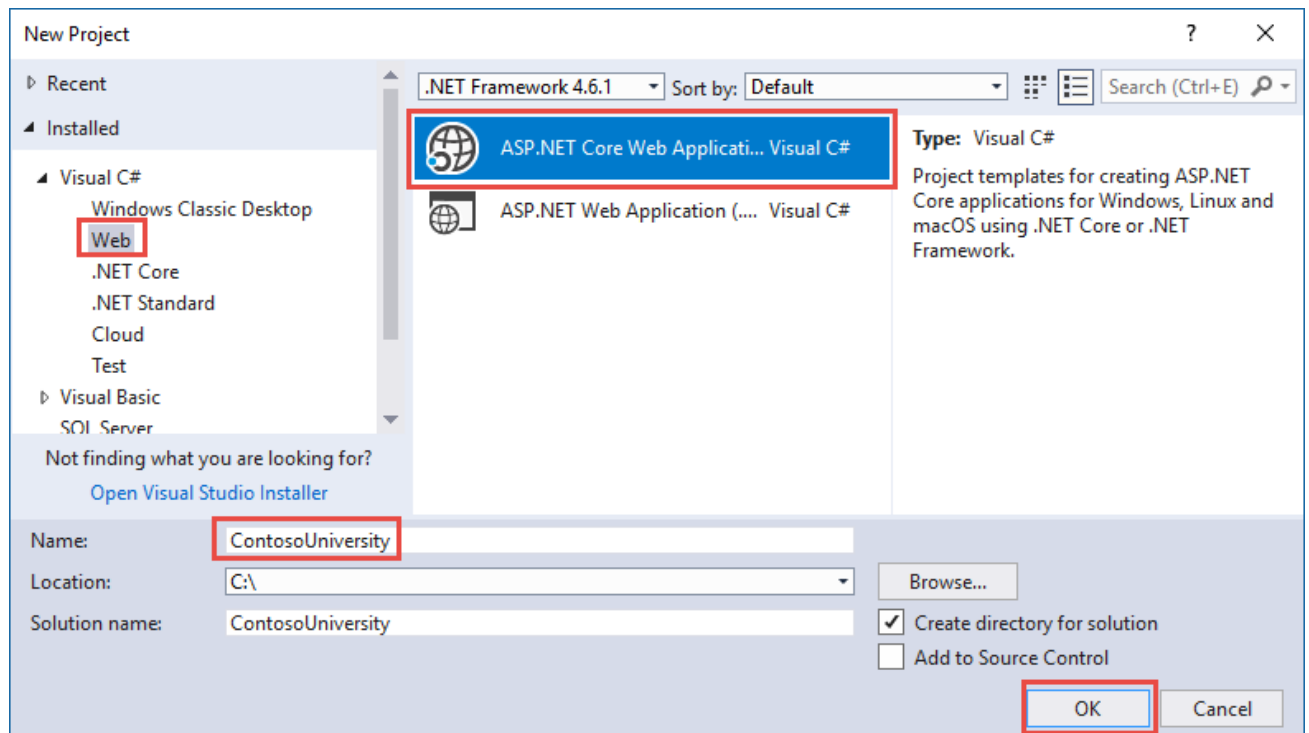


Стиль пользовательского интерфейса этого сайта практически полностью основан на встроенных шаблонах, поскольку это позволяет сосредоточиться на изучении и использовании возможностей платформы Entity Framework.

Создание веб-приложения ASP.NET Core MVC

Откройте Visual Studio и создайте новый веб-проект ASP.NET Core C# с именем "ContosoUniversity".

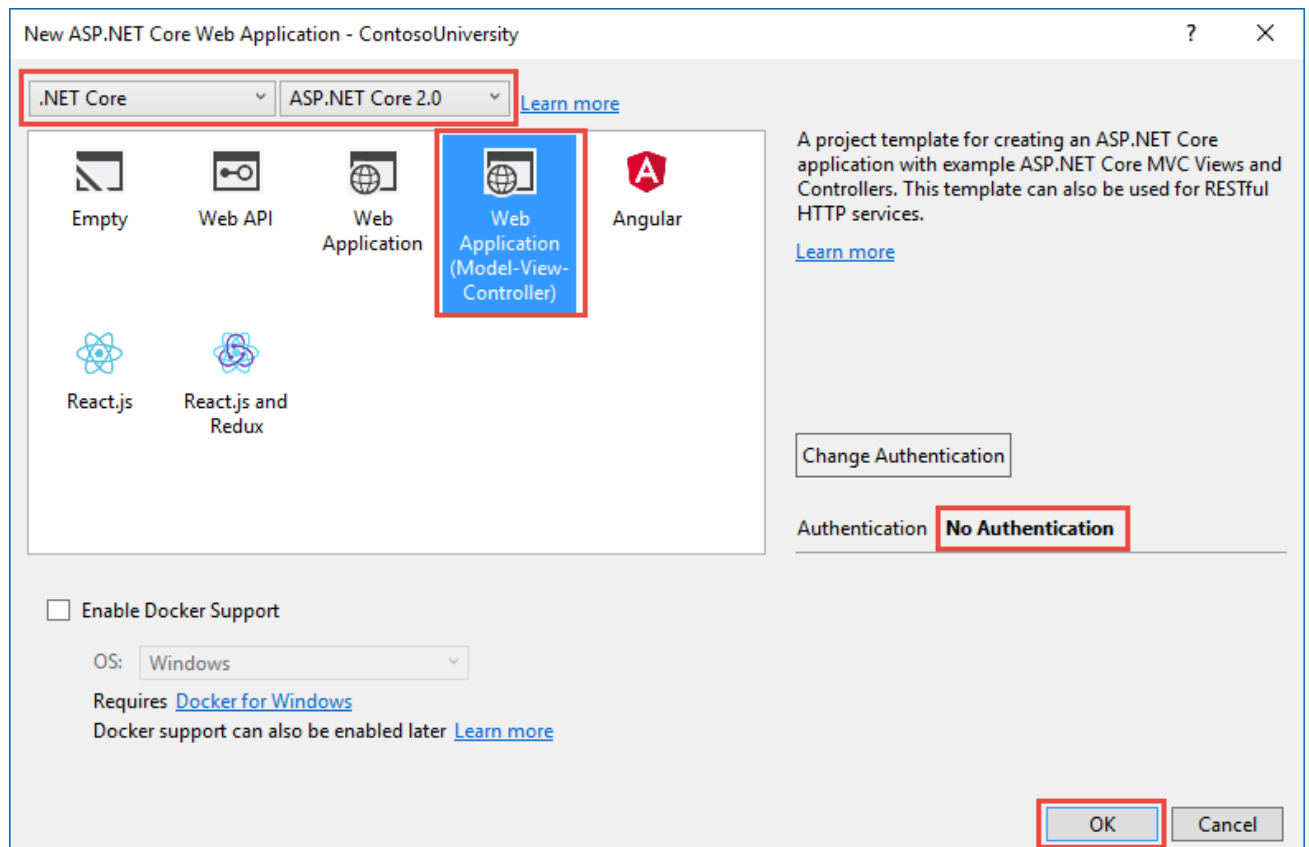
- В меню **Файл** выберите пункт **Создать > Проект**.
- В области слева выберите **Установленные > Visual C# > Интернет**.
- Выберите шаблон проекта **Веб-приложение ASP.NET Core**.
- Введите имя **ContosoUniversity** и нажмите кнопку **OK**.



- Дождитесь появления диалогового окна **Создание веб-приложения ASP.NET Core (.NET Core)**.
- Выберите **ASP.NET Core 2.0** и шаблон **Веб-приложение (модель-представление-контроллер)**.

Примечание. В этом учебнике используются ASP.NET Core 2.0 и EF Core 2.0 или более поздней версии. Убедитесь, что не выбран элемент **ASP.NET Core 1.1**.

- Убедитесь, что для параметра **Проверка подлинности** задано значение **Без проверки подлинности**.
- Нажмите кнопку **OK**.



Настройка стиля сайта

Выполните незначительную настройку меню, макета и домашней страницы сайта.

Откройте файл *Views/Shared/_Layout.cshtml* и внесите следующие изменения:

- Замените все вхождения "ContosoUniversity" на "Contoso University". Таких элементов будет три.
- Добавьте пункты меню **Students** (Учащиеся), **Courses** (Курсы), **Instructors** (Преподаватели) и **Departments** (Кафедры). Удалите пункт меню **Contact** (Контакты).

Изменения выделены.

CSHTML	Копировать
<pre><!DOCTYPE html> <html> <head> <meta charset="utf-8" /> <meta name="viewport" content="width=device-width, initial-scale=1.0" /> <title>@ViewData["Title"] - Contoso University</title> <environment names="Development"> <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" /> <link rel="stylesheet" href="~/css/site.css" /> </environment> <environment names="Staging,Production"></pre>	

```

<link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css
asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
asp-fallback-test-class="sr-only" asp-fallback-test-property="position" a
<link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>

</head>
<body>
  <nav class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-b
      </div>
      <div class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
          <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a>
          <li><a asp-area="" asp-controller="Home" asp-action="About">About</a>
          <li><a asp-area="" asp-controller="Students" asp-action="Index">Stu
          <li><a asp-area="" asp-controller="Courses" asp-action="Index">Cour
          <li><a asp-area="" asp-controller="Instructors" asp-action="Index">
          <li><a asp-area="" asp-controller="Departments" asp-action="Index">
        </ul>
      </div>
    </div>
  </nav>
  <div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
      <p>&copy; 2017 - Contoso University</p>
    </footer>
  </div>

  <environment names="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
  </environment>
  <environment names="Staging,Production">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
      asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
      asp-fallback-test="window.jQuery"
      crossorigin="anonymous"
      integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfHIi9k8z8l9ggpc8X+Ytst
    </script>
    <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
      asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
      asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.f
      crossorigin="anonymous"

```

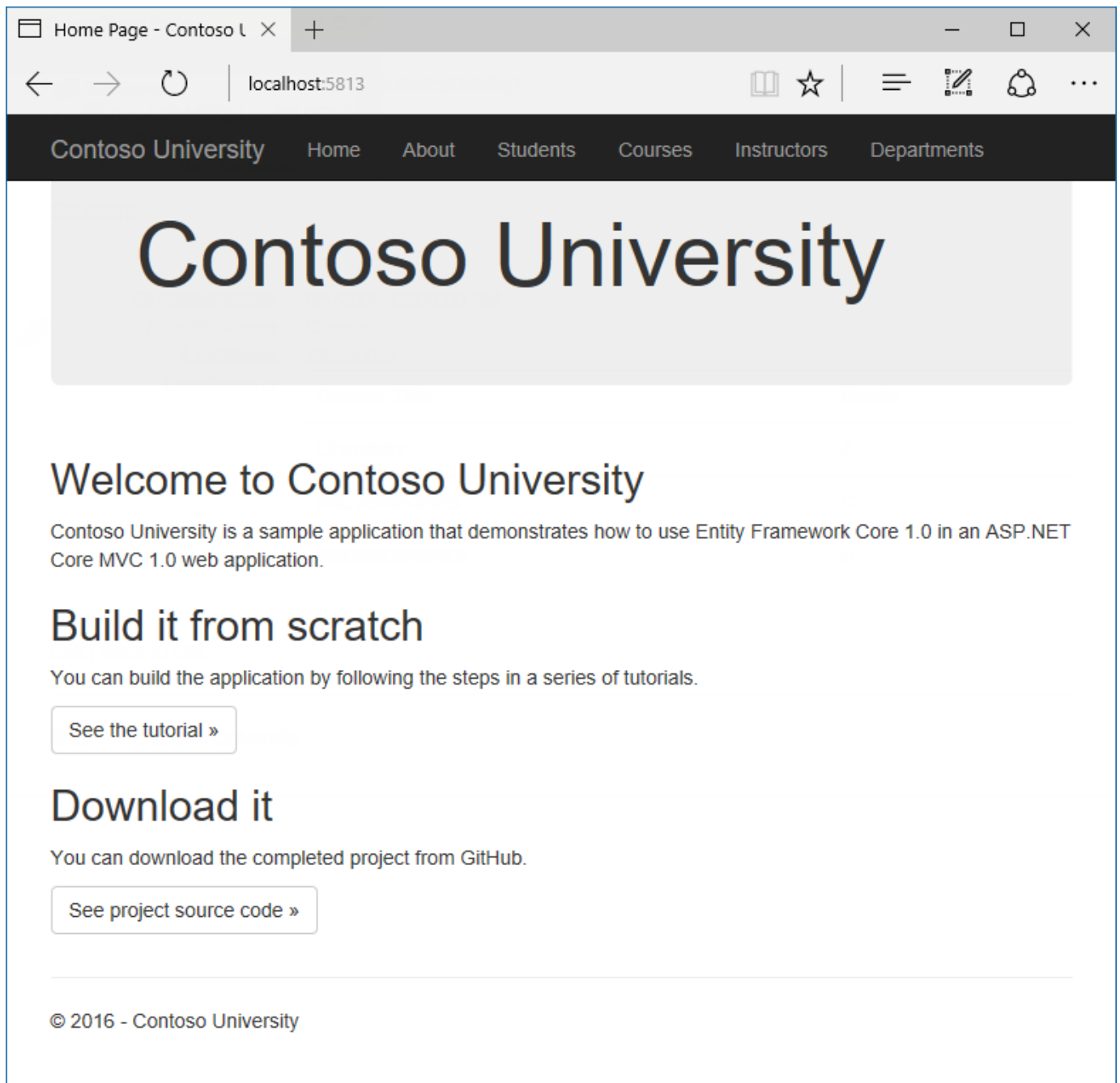
```
integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA712mCWNIpG9mGCD8
</script>
<script src="~/js/site.min.js" asp-append-version="true"></script>
</environment>

@RenderSection("Scripts", required: false)
</body>
</html>
```

Замените содержимое файла *Views/Home/Index.cshtml* следующим кодом, который заменяет текст о ASP.NET и MVC описанием этого приложения:

CSHTML	Копировать
<pre>@{ ViewData["Title"] = "Home Page"; } <div class="jumbotron"> <h1>Contoso University</h1> </div> <div class="row"> <div class="col-md-4"> <h2>Welcome to Contoso University</h2> <p> Contoso University is a sample application that demonstrates how to use Entity Framework Core in an ASP.NET Core MVC web application. </p> </div> <div class="col-md-4"> <h2>Build it from scratch</h2> <p>You can build the application by following the steps in a series of tutorial</p> <p> </div> <div class="col-md-4"> <h2>Download it</h2> <p>You can download the completed project from GitHub.</p> <p><a class="btn btn-default" href="https://github.com/aspnet/Docs/tree/master/ </div> </div></pre>	

Нажмите клавиши CTRL+F5, чтобы запустить проект, и выберите в меню **Отладка > Запуск без отладки**. Откроется домашняя страница, на которой будут представлены созданные в рамках этих учебников вкладки.



Пакеты Entity Framework Core NuGet

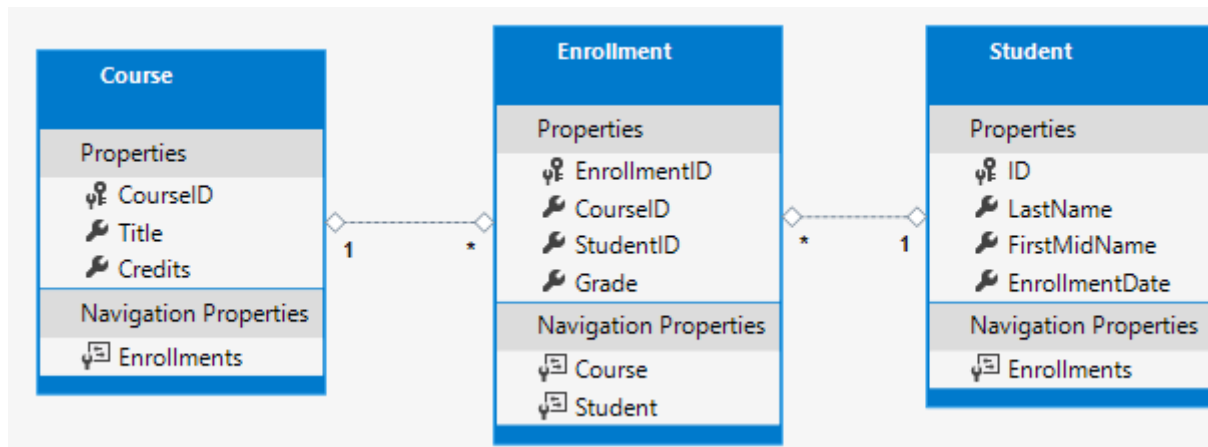
Чтобы реализовать поддержку EF Core в проекте, установите поставщик целевой базы данных. В этом учебнике используется SQL Server, для которого требуется пакет поставщика [Microsoft.EntityFrameworkCore.SqlServer](#). Этот пакет уже входит в состав метапакета [Microsoft.AspNetCore.All](#), поэтому устанавливать его не требуется.

Этот пакет и его зависимости (`Microsoft.EntityFrameworkCore` и `Microsoft.EntityFrameworkCore.Relational`) обеспечивают поддержку среды выполнения для платформы EF. Пакет средств будет добавлен позднее в рамках учебника [Миграции](#).

Дополнительные сведения о других поставщиках баз данных, которые доступны для платформы Entity Framework Core, см. в разделе [Поставщики баз данных](#).

Создание модели данных

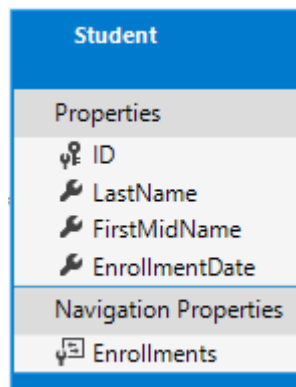
Теперь необходимо создать классы сущностей для приложения университета Contoso. Для начала создаются следующие три сущности.



Между сущностями `Student` и `Enrollment`, а также между сущностями `Course` и `Enrollment` существует отношение "один ко многим". Другими словами, учащийся может быть зарегистрирован в любом количестве курсов, а в отдельном курсе может быть зарегистрировано любое количество учащихся.

В следующих разделах создаются классы для каждой из этих сущностей.

Сущность Student



В папке *Models* создайте файл класса с именем *Student.cs* и замените код шаблона следующим кодом.

C#

Копировать

```

using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
    }
}
  
```

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







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

Свойство `ID` будет использоваться в качестве столбца первичного ключа в таблице базы данных, соответствующей этому классу. По умолчанию платформа Entity Framework интерпретирует в качестве первичного ключа свойство `ID` или `classnameID`.

Свойство `Enrollments` является свойством навигации. Свойства навигации содержат другие сущности, связанные с этой сущностью. В этом случае свойство `Enrollments` сущности `Student` содержит все сущности `Enrollment`, которые связаны с этой сущностью `Student`. Другими словами, если с указанной строкой `Student` в базе данных связаны две строки `Enrollment` (строки, в столбце внешнего ключа `StudentID` которых содержится первичный ключ этого учащегося), в этой сущности `Student` свойство навигации `Enrollments` будет содержать две этих сущности `Enrollment`.

Если свойство навигации может содержать несколько сущностей (как в отношениях "многие ко многим" или "один ко многим"), оно должно иметь тип списка, допускающий добавление, удаление и обновление записей, такой как `ICollection<T>`. Вы можете указать тип `ICollection<T>` либо, например, тип `List<T>` или `HashSet<T>`. Если указан тип `ICollection<T>`, платформа EF по умолчанию создает коллекцию `HashSet<T>`.

Сущность Enrollment

Enrollment	
Properties	
	EnrollmentID
	CourseID
	StudentID
	Grade
Navigation Properties	
	Course
	Student

В папке *Models* создайте файл *Enrollment.cs* и замените существующий код следующим кодом:

C#

 Копировать

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

Свойство `EnrollmentID` будет использоваться в качестве первичного ключа. В этой сущности используется шаблон `classnameID` вместо `ID`, как в сущности `Student`. Как правило, следует выбирать один шаблон, который будет использоваться в рамках всей модели данных. В этом случае демонстрируется возможность использования любого из шаблонов. В [одном из следующих учебников](#) вы узнаете, за счет чего использование идентификатора без имени класса позволяет упростить реализацию наследования в модели данных.

Свойство `Grade` имеет тип `enum`. Знак вопроса после объявления типа `Grade` указывает, что свойство `Grade` допускает значение `NULL`. Оценка со значением `NULL` отличается от нулевой оценки тем, что при таком значении оценка еще не известна или не назначена.

Свойство `StudentID` представляет собой внешний ключ. Ему соответствует свойство навигации `Student`. Сущность `Enrollment` связана с одной сущностью `Student`, поэтому это свойство может содержать одну сущность `Student` (в отличие от представленного ранее свойства навигации `Student.Enrollments`, которое может содержать несколько сущностей `Enrollment`).

Свойство `CourseID` представляет собой внешний ключ. Ему соответствует свойство навигации `Course`. Сущность `Enrollment` связана с одной сущностью `Course`.

Платформа Entity Framework интерпретирует свойство как свойство внешнего ключа, если оно имеет имя `<navigation property name><primary key property name>` (например, `StudentID` для свойства навигации `Student`, поскольку сущность `Student` имеет первичный ключ

ID). Свойства внешнего ключа также могут называться просто `<primary key property name>` (например, `CourseID` поскольку сущность `Course` имеет первичный ключ `CourseID`).

Сущность `Course`

Course	
Properties	
🔑	CourseID
🔑	Title
🔑	Credits
Navigation Properties	
🔗	Enrollments

В папке `Models` создайте файл `Course.cs` и замените существующий код следующим кодом:

C# Копировать

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

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

Свойство `Enrollments` является свойством навигации. Сущность `Course` может быть связана с любым числом сущностей `Enrollment` .

Более подробное описание атрибута `DatabaseGenerated` будет приведено в [одном из следующих учебников](#) этой серии. Фактически, этот атрибут позволяет ввести первичный ключ для курса, а не использовать базу данных, чтобы создать его.


Создание контекста базы данных

Контекст базы данных — это основной класс, который координирует функциональные возможности Entity Framework для заданной модели данных. Этот класс создается путем наследования от класса `Microsoft.EntityFrameworkCore.DbContext` . В коде указываются

сущности, которые включаются в модель данных. Также вы можете настроить реакцию платформы Entity Framework на некоторые события. В этом проекте соответствующий класс называется `SchoolContext`.

В папке проекта создайте папку *Data*.

В папке *Data* создайте новый файл класса с именем *SchoolContext.cs* и замените код шаблона следующим кодом:

C#  Копировать

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}
```

Этот код создает свойство `DbSet` для каждого набора сущностей. В терминологии Entity Framework набор сущностей обычно соответствует таблице базы данных, а сущность — строке в этой таблице.

Вы можете опустить инструкции `DbSet<Enrollment>` и `DbSet<Course>`. Это не нарушит функциональность. Платформа Entity Framework включает эти инструкции неявно, поскольку сущность `Student` ссылается на сущность `Enrollment`, а `Enrollment` ссылается на сущность `Course`.

При создании базы данных платформа EF создает таблицы с именами, соответствующими именам свойств в `DbSet`. Имена свойств для коллекций, как правило, задаются во множественном числе (например, `Students` вместо `Student`), однако единого мнения по поводу присвоения имен во множественном числе таблицам среди разработчиков не существует. В этих учебниках вместо принятого по умолчанию способа таблицам в `DbContext` присваиваются имена в единственном числе. Чтобы сделать это, добавьте выделенный ниже код после последнего свойства `DbSet`.

C#  Копировать

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

Регистрация контекста с помощью внедрения зависимостей

ASP.NET Core по умолчанию реализует технологию [внедрения зависимостей](#). С помощью внедрения зависимостей службы (например, контекст базы данных EF) регистрируются во время запуска приложения. Затем компоненты, которые используют эти службы (например, контроллеры MVC), обращаются к ним через параметры конструктора. Код конструктора контроллера, который получает экземпляр контекста, будет приведен позднее в этом учебнике.

Чтобы зарегистрировать `SchoolContext` как службу, откройте файл `Startup.cs` и добавьте выделенные строки в метод `ConfigureServices`.

C#

 Копировать


```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<SchoolContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddMvc();
}
```

Имя строки подключения передается в контекст путем вызова метода для объекта `DbContextOptionsBuilder`. При локальной разработке [система конфигурации ASP.NET Core](#) считывает строку подключения из файла `appsettings.json`.

Добавьте инструкции `using` для пространств имен `ContosoUniversity.Data` и `Microsoft.EntityFrameworkCore`, после чего выполните построение проекта.


C#

 Копировать

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
```

Откройте файл `appsettings.json` и добавьте строку подключения, как показано в следующем примере.

JSON

 Копировать

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=ContosoUniversity1;Tr
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

SQL Server Express LocalDB

Строка подключения указывает на базу данных SQL Server LocalDB. LocalDB — это упрощенная версия ядра СУБД SQL Server Express, предназначенная для разработки приложений и не ориентированная на использование в производственной среде. LocalDB запускается по запросу в пользовательском режиме, поэтому настройки не слишком сложны. По умолчанию база данных LocalDB создает файлы `.mdf` в каталоге

```
C:/Users/<user> .
```

Добавление кода для инициализации базы с использованием тестовых данных

Платформа Entity Framework создает пустую базу данных. В этом разделе вы напишете метод, который вызывается после создания базы данных и заполняет ее тестовыми

данными.

Здесь будет использоваться метод `EnsureCreated` для автоматического создания базы данных. В [одном из следующих учебников](#) вы узнаете, как обрабатывать изменения модели с использованием Code First Migrations, что позволяет изменять схему базы данных вместо того, чтобы удалять и повторно создавать ее.

В папке *Data* создайте новый файл класса с именем *DbInitializer.cs* и замените код шаблона следующим кодом, который обеспечивает создание базы данных и загрузку в нее тестовых данных.

C#

 Копировать

```
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return; // DB has been seeded
            }

            var students = new Student[]
            {
                new Student{FirstMidName="Carson", LastName="Alexander", EnrollmentDate=DateTime.Now},
                new Student{FirstMidName="Meredith", LastName="Alonso", EnrollmentDate=DateTime.Now},
                new Student{FirstMidName="Arturo", LastName="Anand", EnrollmentDate=DateTime.Now},
                new Student{FirstMidName="Gytis", LastName="Barzdukas", EnrollmentDate=DateTime.Now},
                new Student{FirstMidName="Yan", LastName="Li", EnrollmentDate=DateTime.Parse("2008-09-01")},
                new Student{FirstMidName="Peggy", LastName="Justice", EnrollmentDate=DateTime.Now},
                new Student{FirstMidName="Laura", LastName="Norman", EnrollmentDate=DateTime.Now},
                new Student{FirstMidName="Nino", LastName="Olivetto", EnrollmentDate=DateTime.Now}
            };
            foreach (Student s in students)
            {
                context.Students.Add(s);
            }
            context.SaveChanges();

            var courses = new Course[]
            {
                new Course{CourseID=1050, Title="Chemistry", Credits=3},
                new Course{CourseID=4022, Title="Microeconomics", Credits=3},
            };
            foreach (Course c in courses)
            {
                context.Courses.Add(c);
            }
            context.SaveChanges();
        }
    }
}
```

```
new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
new Course{CourseID=1045,Title="Calculus",Credits=4},
new Course{CourseID=3141,Title="Trigonometry",Credits=4},
new Course{CourseID=2021,Title="Composition",Credits=3},
new Course{CourseID=2042,Title="Literature",Credits=4}
};
foreach (Course c in courses)
{
    context.Courses.Add(c);
}
context.SaveChanges();

var enrollments = new Enrollment[]
{
    new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
    new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
    new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
    new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
    new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
    new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
    new Enrollment{StudentID=3,CourseID=1050},
    new Enrollment{StudentID=4,CourseID=1050},
    new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
    new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
    new Enrollment{StudentID=6,CourseID=1045},
    new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
};
foreach (Enrollment e in enrollments)
{
    context.Enrollments.Add(e);
}
context.SaveChanges();
}
}
```

Этот код проверяет, добавлены ли в базу данных учащиеся. Если нет, база данных считается новой и заполняется тестовыми данными. Для повышения производительности тестовые данные загружаются массивами, а не коллекциями `List<T>`.

В файле *Program.cs* измените метод `Main`, чтобы реализовать следующее поведение при запуске приложения:

- Получение экземпляра контекста базы данных из контейнера внедрения зависимостей.
- Вызов метода инициализации с передачей ему контекста.
- Высвобождение контекста после завершения работы метода заполнения.

```
public static void Main(string[] args)
{
    var host = BuildWebHost(args);

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try
        {
            var context = services.GetRequiredService<SchoolContext>();
            DbInitializer.Initialize(context);
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred while seeding the database.");
        }
    }

    host.Run();
}
```

Добавьте инструкции `using`:

C#

 Копировать

```
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Data;
```

В предыдущих учебниках аналогичный код использовался в методе `Configure` в файле *Startup.cs*. Мы рекомендуем использовать метод `Configure` только для настройки конвейера запросов. Код запуска приложения принадлежит методу `Main`.

Теперь при первом запуске приложения будет создана и заполнена тестовыми данными необходимая для работы база данных. При любом изменении модели данных вы можете удалить базу, обновить метод заполнения и начать работу с новой базой данных аналогичным способом. В следующих учебниках вы узнаете, как изменить базу данных при изменении модели данных, не прибегая к ее удалению и повторному созданию.

Создание контроллера и представлений

Далее вы будете использовать механизм шаблонов Visual Studio для добавления контроллера и представлений MVC, которые будут использовать платформу EF для запроса данных и их сохранения.

Автоматическое создание методов и представлений операций CRUD (создание, чтение, обновление и удаление) называется формированием шаблонов. Формирование шаблонов

отличается от создания кода тем, что шаблонный код является отправной точкой и может изменяться в соответствии с потребностями, тогда как сформированный код обычно не изменяется. В тех случаях, когда требуется настроить созданный код в соответствии с внесенными изменениями, вы можете использовать разделяемые классы или повторно создать код.

- Щелкните правой кнопкой мыши папку **Контроллеры** в **обозревателе решений** и выберите **Добавить > Создать шаблонный элемент**.

Если открывается диалоговое окно **Добавление зависимостей MVC**:

- [Обновите Visual Studio до последней версии](#). Это диалоговое окно отображает версии Visual Studio, предшествующие 15.5.
- Если обновить не удастся, выберите **ДОБАВИТЬ** и снова выполните шаги по добавлению контроллера.
- В диалоговом окне **Добавление шаблона**:
 - Выберите **Контроллер MVC с представлениями, использующий Entity Framework**.
 - Нажмите кнопку **Добавить**.
- В диалоговом окне **Добавление контроллера**:
 - В разделе **Класс модели** выберите **Student**.
 - В разделе **Класс контекста данных** выберите **SchoolContext**.
 - Оставьте предлагаемое по умолчанию имя **StudentsController**.
 - Нажмите кнопку **Добавить**.

The screenshot shows the 'Add Controller' dialog box. The 'Model class' dropdown is set to 'Student (ContosoUniversity.Models)'. The 'Data context class' dropdown is set to 'SchoolContext (ContosoUniversity.Data)'. Under the 'Views' section, the checkboxes for 'Generate views', 'Reference script libraries', and 'Use a layout page:' are all checked. The 'Controller name' text box contains 'StudentsController'. The 'Add' button is highlighted with a red rectangle.

При нажатии кнопки **Добавить** подсистема формирования шаблонов Visual Studio создает файл *StudentsController.cs* и набор представлений (файлы с расширением *.cshtml*), которые будут работать с контроллером.

(Подсистема формирования шаблонов также может создать контекст базы данных, если вы не создали его вручную, как было показано ранее в этом учебнике. Вы можете задать новый класс контекста в диалоговом окне **Добавление контроллера**, нажав на значок плюса справа от раздела **Класс контекста данных**. В этом случае Visual Studio создаст класс `DbContext`, а также контроллеры и представления.)

Обратите внимание, что контроллер принимает `SchoolContext` в качестве параметра конструктора.

```
C# Копировать


namespace ContosoUniversity.Controllers
{
    public class StudentsController : Controller
    {
        private readonly SchoolContext _context;

        public StudentsController(SchoolContext context)
        {
            _context = context;
        }
    }
}
```

Технология внедрения зависимостей ASP.NET обеспечивает передачу экземпляра `SchoolContext` в контроллер. Это поведение было настроено ранее в файле *Startup.cs*.

Контроллер содержит метод действия `Index`, который отображает всех учащихся в базе данных. Этот метод получает список учащихся из набора сущностей `Students`, считывая свойство `Students` экземпляра контекста базы данных:

C#

 Копировать

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

Позднее в этом учебнике будут описаны элементы асинхронного программирования в этом коде.

Представление `Views/Students/Index.cshtml` отображает этот список в таблице:

CSHTML

 Копировать

```
@model IEnumerable<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

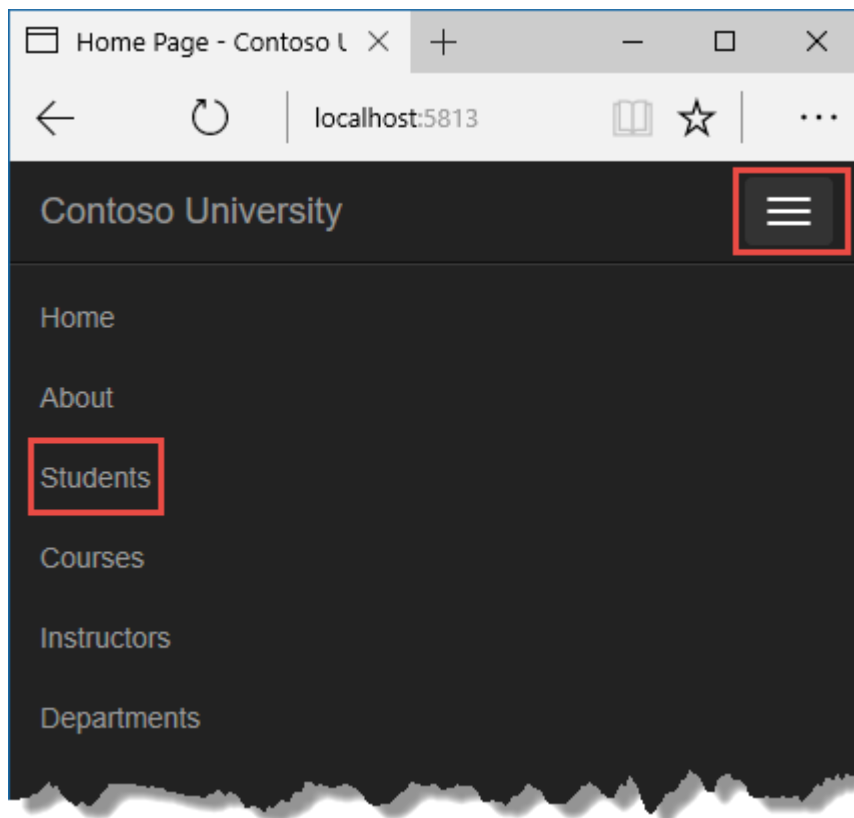
<h2>Index</h2>

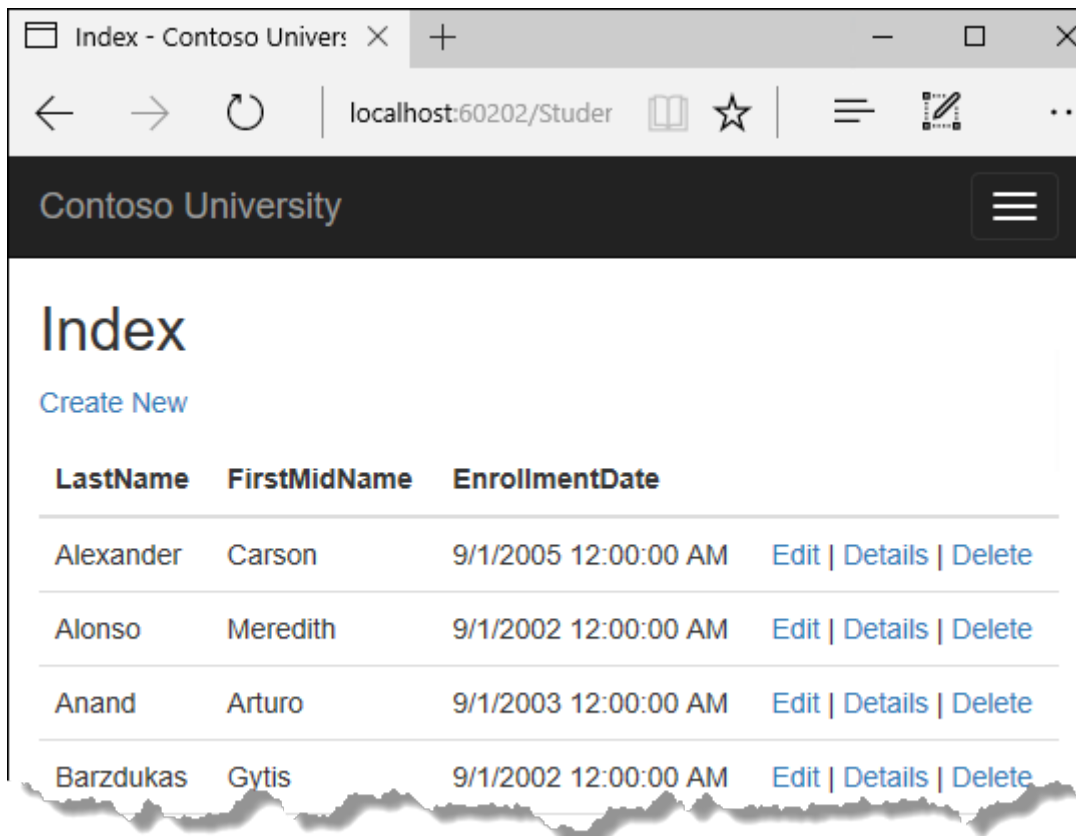
<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.LastName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FirstMidName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.EnrollmentDate)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
```

```
@Html.DisplayFor(modelItem => item.FirstMidName)
</td>
<td>
    @Html.DisplayFor(modelItem => item.EnrollmentDate)
</td>
<td>
    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
</td>
</tr>
}
</tbody>
</table>
```

Нажмите клавиши CTRL+F5, чтобы запустить проект, и выберите в меню **Отладка > Запуск без отладки**.

Перейдите на вкладку Students (Учащиеся), чтобы просмотреть тестовые данные, добавленные методом `DbInitializer.Initialize`. В зависимости от размеров окна браузера ссылку на вкладку `Student` можно найти вверху страницы или щелкнув значок навигации в правом верхнем углу.





Просмотр базы данных

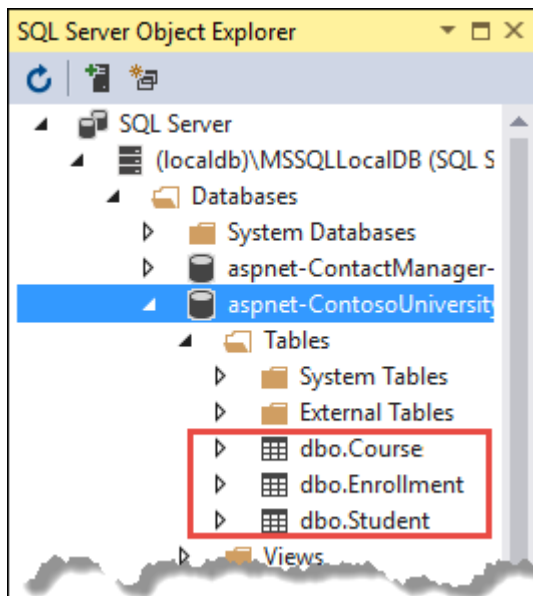
При запуске приложения метод `DbInitializer.Initialize` вызывает метод `EnsureCreated`. Платформа EF определяет, что база данных отсутствует, и создает ее, после чего код в оставшейся части метода `Initialize` заполняет базу данными. Для просмотра базы данных в Visual Studio можно использовать **обозреватель объектов SQL Server (SSOX)**.

Закройте браузер.

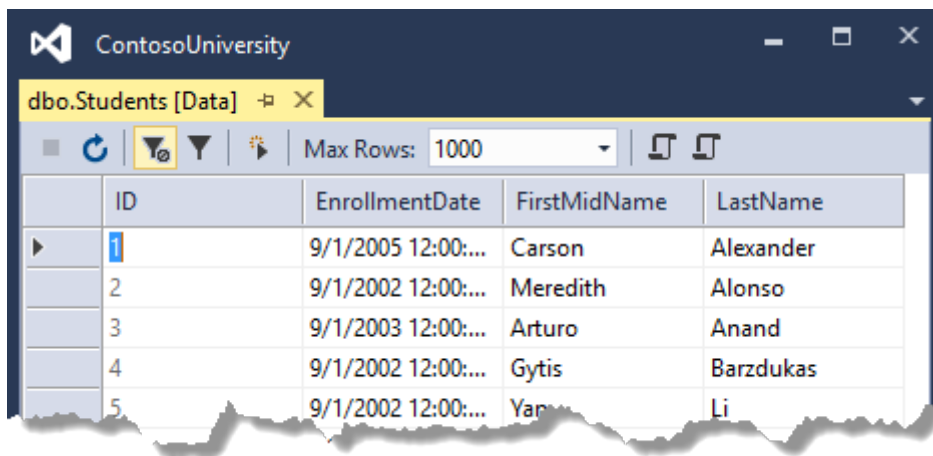
Если окно SSOX не открылось, выберите его в меню **Вид Visual Studio**.

В окне SSOX щелкните **(localdb)\MSSQLLocalDB > Базы данных** и затем щелкните запись базы данных, имя которой указано в строке подключения в вашем файле `appsettings.json`.

Разверните узел **Таблицы**, чтобы просмотреть представленные в базе таблицы.



Щелкните правой кнопкой мыши таблицу **Student** и выберите пункт **Просмотр данных**, чтобы просмотреть созданные столбцы и строки, вставленные в базу данных.



	ID	EnrollmentDate	FirstMidName	LastName
▶	1	9/1/2005 12:00:...	Carson	Alexander
	2	9/1/2002 12:00:...	Meredith	Alonso
	3	9/1/2003 12:00:...	Arturo	Anand
	4	9/1/2002 12:00:...	Gytis	Barzdukas
	5	9/1/2002 12:00:...	Yan	Li

Файлы базы данных с расширением *.mdf* и *.ldf* располагаются в папке *C:\Users<имя_пользователя>*.

Поскольку вы вызываете метод `EnsureCreated` в методе инициализатора, который выполняется при запуске приложения, теперь вы можете внести изменения в класс `Student`, удалить базу данных и снова запустить приложение. После этого база данных будет создана повторно в соответствии с внесенными изменениями. Например, при добавлении свойства `EmailAddress` в класс `Student` во вновь созданной таблице появится столбец `EmailAddress`.

Соглашения

Чтобы платформа Entity Framework автоматически создавала полную базу данных на основе принятых соглашений и допущений, потребуется написать минимальный объем кода.

- В качестве имен таблиц используются имена свойств `DbSet`. Для сущностей, на которые не ссылается свойство `DbSet`, в качестве имен таблиц используются имена классов сущностей.
- В качестве имен столбцов используются имена свойств сущностей.
- Свойства сущностей с именем `ID` или `classNameID` распознаются как свойства первичного ключа.
- Свойство интерпретируется как свойство внешнего ключа, если оно имеет имя (например, `StudentID` для свойства навигации `Student`, так как сущность `Student` имеет первичный ключ `ID`). Свойства внешнего ключа также могут называться просто (например, `EnrollmentID`, поскольку сущность `Enrollment` имеет первичный ключ `EnrollmentID`).

Стандартное поведение можно переопределить. Например, можно явно задать имена таблиц, как было показано ранее в этом учебнике. Также можно указать имена столбцов и задать любое свойство в качестве первичного или внешнего ключа, как будет показано в [одном из следующих учебников](#) этой серии.

Асинхронный код

Асинхронное программирование по умолчанию применяется в ASP.NET Core и EF Core.

Веб-сервер имеет ограниченное число потоков, поэтому при высокой загрузке могут использоваться все доступные потоки. В таких случаях сервер не может обрабатывать новые запросы до тех пор, пока не будут высвобождены потоки. В синхронном коде многие потоки могут быть заняты, не выполняя при этом какие-либо операции и ожидая завершения ввода-вывода. В асинхронном коде в то время, когда процесс ожидает завершения ввода-вывода, его поток высвобождается и может использоваться сервером для обработки других запросов. Таким образом, асинхронный код позволяет более эффективно использовать ресурсы сервера, который может обрабатывать больше трафика без задержек.

Во время выполнения асинхронный код использует немного больше служебных ресурсов, однако при низком объеме трафика этим можно пренебречь. Тем не менее в случае большого объема трафика это дает существенный выигрыш в производительности.

В следующем коде для асинхронного выполнения используются ключевое слово `async`, возвращаемое значение `Task<T>`, ключевое слово `await` и метод `ToListAsync`.

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

- Ключевое слово `async` указывает компилятору создавать обратные вызовы для частей тела метода и автоматически создавать возвращаемый объект `Task<IActionResult>`.
- Тип возвращаемого значения `Task<IActionResult>` представляет текущую операцию с помощью результата типа `IActionResult`.
- Ключевое слово `await` предписывает компилятору разделить метод на две части. Первая часть завершается операцией, которая запускается в асинхронном режиме. Вторая часть помещается в метод обратного вызова, который вызывается при завершении операции.
- `ToListAsync` является асинхронной версией метода расширения `ToList`.

При написании асинхронного кода, который использует Entity Framework, необходимо учитывать некоторые моменты:

- Асинхронно выполняются только те инструкции, в результате которых в базу данных отправляются запросы или команды. К ним относятся, например, `ToListAsync`, `SingleOrDefaultAsync` и `SaveChangesAsync`. В их число не входят, например, инструкции, которые просто изменяют `IQueryable`, такие как

```
var students = context.Students.Where(s => s.LastName == "Davolio");
```
- Контекст EF не является потокобезопасным, поэтому не следует пытаться выполнять несколько операций параллельно. При вызове любого асинхронного метода EF всегда используйте ключевое слово `await`.
- Если вы хотите использовать преимущества в производительности, которые обеспечивает асинхронный код, убедитесь, что все используемые пакеты библиотек (например, для разбиения на страницы) также используют асинхронный код при вызове любых методов Entity Framework, выполняющих запросы к базе данных.

Дополнительные сведения об асинхронных методах программирования в .NET см. в разделе [Обзор асинхронной модели](#).

Сводка

Теперь вы создали простое приложение, которое использует Entity Framework Core и SQL Server Express LocalDB для хранения и отображения данных. В следующем учебнике вы узнаете, как выполнять основные операции CRUD (создание, чтение, обновление и удаление).

[Вперед](#)