

Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8

09/26/2019 • 33 minutes to read •  +4

In this article

[Prerequisites](#)

[Database engines](#)

[Troubleshooting](#)

[The sample app](#)

[Create the web app project](#)

[Set up the site style](#)

[The data model](#)

[The Student entity](#)

[The Enrollment entity](#)

[The Course entity](#)

[Scaffold Student pages](#)
[Database connection string](#)
[Update the database context class](#)
[Startup.cs](#)
[Create the database](#)
[Seed the database](#)
[View the database](#)
[Asynchronous code](#)
[Next steps](#)

By [Tom Dykstra](#) and [Rick Anderson](#)

This is the first in a series of tutorials that show how to use Entity Framework (EF) Core in an [ASP.NET Core Razor Pages](#) app. The tutorials build a web site for a fictional Contoso University. The site includes functionality such as student admission, course creation, and instructor assignments. The tutorial uses the code first approach. For information on following this tutorial using the database first approach, see [this Github issue](#).

[Download or view the completed app.](#) [Download instructions.](#)

Prerequisites

- If you're new to Razor Pages, go through the [Get started with Razor Pages](#) tutorial series before starting this one.

Visual Studio

Visual Studio Code

- [Visual Studio 2019](#) with the **ASP.NET and web development** workload
- [.NET Core 3.0 SDK or later](#)

Database engines

The Visual Studio instructions use [SQL Server LocalDB](#), a version of SQL Server Express that runs only on Windows.

The Visual Studio Code instructions use [SQLite](#), a cross-platform database engine.


If you choose to use SQLite, download and install a third-party tool for managing and viewing a SQLite database, such as [DB Browser for SQLite](#).

Troubleshooting

If you run into a problem you can't resolve, compare your code to the [completed project](#). A good way to get help is by posting a question to StackOverflow.com, using the [ASP.NET Core tag](#) or the [EF Core tag](#).

The sample app

The app built in these tutorials is a basic university web site. Users can view and update student, course, and instructor information. Here are a few of the screens created in the tutorial.

Contoso University 

Index


[Create New](#)

Find by name: [Search](#) | [Back to full List](#)

Last Name	First Name	Enrollment Date	
Alexander	Carson	2016-09-01	Edit Details Delete
Alonso	Meredith	2018-09-01	Edit Details Delete
Anand	Arturo	2019-09-01	Edit Details Delete

[Previous](#) [Next](#)

Contoso University



Edit

Student

Last Name

Alexander

First Name

Carson

Enrollment Date

09/01/2016

Save

The UI style of this site is based on the built-in project templates. The tutorial's focus is on how to use EF Core, not how to customize the UI.

Follow the link at the top of the page to get the source code for the completed project. The *cu30* folder has the code for the ASP.NET Core 3.0 version of the tutorial. Files that reflect the state of the code for tutorials 1-7 can be found in the *cu30snapshots* folder.

Visual Studio

Visual Studio Code

To run the app after downloading the completed project:

- Delete three files and one folder that have *SQLite* in the name.
- Build the project.
- In Package Manager Console (PMC) run the following command:

PowerShell

 Copy**Update-Database**

- Run the project to seed the database.

Create the web app project

Visual Studio

Visual Studio Code

From the Visual Studio **File** menu, select **New > Project**

- From the Visual Studio **File** menu, select **New > Project**.
- Select **ASP.NET Core Web Application**.
- Name the project *ContosoUniversity*. It's important to use this exact name including capitalization, so the namespaces match when code is copied and pasted.
- Select **.NET Core** and **ASP.NET Core 3.0** in the dropdowns, and then select **Web Application**.

Set up the site style

Set up the site header, footer, and menu by updating *Pages/Shared/_Layout.cshtml*:

- Change each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- Delete the **Home** and **Privacy** menu entries, and add entries for **About**, **Students**, **Courses**, **Instructors**, and **Departments**.

The changes are highlighted.

CSHTML	 Copy
<pre><!DOCTYPE html> <html lang="en"> <head> <meta charset="utf-8" /> <meta name="viewport" content="width=device-width, initial-scale=1.0" /> <title>@ViewData["Title"] - Contoso University</title> <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" /> <link rel="stylesheet" href="~/css/site.css" /> </head> <body> <header> <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3"></pre>	

```

<div class="container">
  <a class="navbar-brand" asp-area="" asp-page="/Index">Contoso University</a>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-
collapse" aria-controls="navbarSupportedContent"
    aria-expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
    <ul class="navbar-nav flex-grow-1">

      <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-page="/About">About</a>
      </li>
      <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-page="/Students/Index">Students</a>
      </li>
      <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-page="/Courses/Index">Courses</a>
      </li>
      <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-
page="/Instructors/Index">Instructors</a>
      </li>
      <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-
page="/Departments/Index">Departments</a>
      </li>
    </ul>
  </div>
</div>
</nav>
</header>
<div class="container">
  <main role="main" class="pb-3">
    @RenderBody()
  </main>

```



```
</div>

<footer class="border-top footer text-muted">
  <div class="container">
    &copy; 2019 - Contoso University - <a asp-area="" asp-page="/Privacy">Privacy</a>
  </div>
</footer>

<script src="~/lib/jquery/dist/jquery.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
<script src="~/js/site.js" asp-append-version="true"></script>

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

In *Pages/Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET Core with text about this app:

CSHTML

 Copy

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

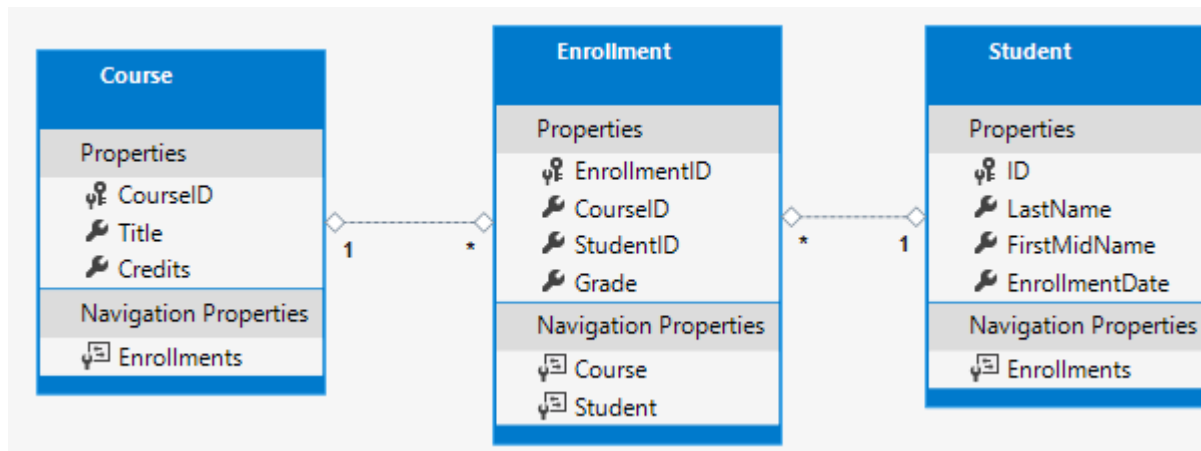
<div class="row mb-auto">
  <div class="col-md-4">
    <div class="row no-gutters border mb-4">
      <div class="col p-4 mb-4">
        <p class="card-text">
          Contoso University is a sample application that
          demonstrates how to use Entity Framework Core in an
          ASP.NET Core Razor Pages web app.
        </p>
      </div>
    </div>
  </div>
</div>
```

```
        </div>
    </div>
</div>
<div class="col-md-4">
    <div class="row no-gutters border mb-4">
        <div class="col p-4 d-flex flex-column position-static">
            <p class="card-text mb-auto">
                You can build the application by following the steps in a series of tutorials.
            </p>
            <p>
                <a href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro" class="stretched-
link">See the tutorial</a>
            </p>
        </div>
    </div>
</div>
<div class="col-md-4">
    <div class="row no-gutters border mb-4">
        <div class="col p-4 d-flex flex-column">
            <p class="card-text mb-auto">
                You can download the completed project from GitHub.
            </p>
            <p>
                <a href="https://github.com/dotnet/AspNetCore.Docs/tree/master/aspnetcore/data/ef-
rp/intro/samples" class="stretched-link">See project source code</a>
            </p>
        </div>
    </div>
</div>
</div>
```

Run the app to verify that the home page appears.

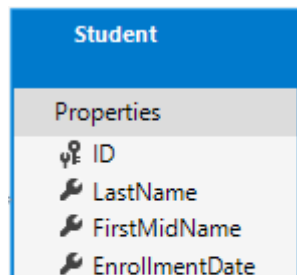
The data model

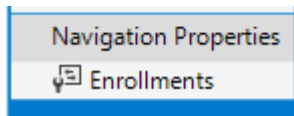
The following sections create a data model:



A student can enroll in any number of courses, and a course can have any number of students enrolled in it.

The Student entity





- Create a *Models* folder in the project folder.
- Create *Models/Student.cs* with the following code:

```
C# Copy  
  
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; }  
    }  
}
```

The `ID` property becomes the primary key column of the database table that corresponds to this class. By default, EF Core interprets a property that's named `ID` or `classnameID` as the primary key. So the alternative automatically recognized name for the `Student` class primary key is `StudentID`.

The `Enrollments` property is a [navigation property](#). Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student` entity holds all of the `Enrollment` entities that are related to that `Student`. For example, if a `Student` row in the database has two related `Enrollment` rows, the `Enrollments` navigation property contains those two `Enrollment` entities.


In the database, an Enrollment row is related to a Student row if its StudentID column contains the student's ID value. For example, suppose a Student row has ID=1. Related Enrollment rows will have StudentID = 1. StudentID is a *foreign key* in the Enrollment table.

The `Enrollments` property is defined as `ICollection<Enrollment>` because there may be multiple related Enrollment entities. You can use other collection types, such as `List<Enrollment>` or `HashSet<Enrollment>`. When `ICollection<Enrollment>` is used, EF Core creates a `HashSet<Enrollment>` collection by default.

The Enrollment entity

Enrollment	
Properties	
PK	EnrollmentID
FK	CourseID
FK	StudentID
FK	Grade
Navigation Properties	
1	Course
1	Student

Create `Models/Enrollment.cs` with the following code:

C#	 Copy
<pre>namespace ContosoUniversity.Models { public enum Grade { A, B, C, D, F } public class Enrollment {</pre>	

```
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; }  
}  
}
```

The `EnrollmentID` property is the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself. For a production data model, choose one pattern and use it consistently. This tutorial uses both just to illustrate that both work. Using `ID` without `classname` makes it easier to implement some kinds of data model changes.

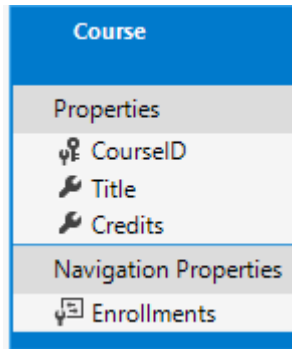
The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is [nullable](#). A grade that's null is different from a zero grade—null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property contains a single `Student` entity.

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

EF Core interprets a property as a foreign key if it's named `<navigation property name><primary key property name>`. For example, `StudentID` is the foreign key for the `Student` navigation property, since the `Student` entity's primary key is `ID`. Foreign key properties can also be named `<primary key property name>`. For example, `CourseID` since the `Course` entity's primary key is `CourseID`.

The Course entity



Create *Models/Course.cs* with the following code:

C#	 Copy
<pre>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; } } }</pre>	

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

The `DatabaseGenerated` attribute allows the app to specify the primary key rather than having the database generate it.

Build the project to validate that there are no compiler errors.

Scaffold Student pages

In this section, you use the ASP.NET Core scaffolding tool to generate:

- An EF Core *context* class. The context is the main class that coordinates Entity Framework functionality for a given data model. It derives from the `Microsoft.EntityFrameworkCore.DbContext` class.
- Razor pages that handle Create, Read, Update, and Delete (CRUD) operations for the `Student` entity.

Visual Studio

Visual Studio Code

- Create a *Students* folder in the *Pages* folder.
- In **Solution Explorer**, right-click the *Pages/Students* folder and select **Add > New Scaffolded Item**.
- In the **Add Scaffold** dialog, select **Razor Pages using Entity Framework (CRUD) > ADD**.
- In the **Add Razor Pages using Entity Framework (CRUD)** dialog:
 - In the **Model class** drop-down, select **Student (ContosoUniversity.Models)**.
 - In the **Data context class** row, select the + (plus) sign.
 - Change the data context name from *ContosoUniversity.Models.ContosoUniversityContext* to *ContosoUniversity.Data.SchoolContext*.
 - Select **Add**.

The following packages are automatically installed:

- `Microsoft.VisualStudio.Web.CodeGeneration.Design`
- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.Extensions.Logging.Debug`
- `Microsoft.EntityFrameworkCore.Tools`

If you have a problem with the preceding step, build the project and retry the scaffold step.

The scaffolding process:

- Creates Razor pages in the *Pages/Students* folder:
 - *Create.cshtml* and *Create.cshtml.cs*
 - *Delete.cshtml* and *Delete.cshtml.cs*
 - *Details.cshtml* and *Details.cshtml.cs*
 - *Edit.cshtml* and *Edit.cshtml.cs*
 - *Index.cshtml* and *Index.cshtml.cs*
- Creates *Data/SchoolContext.cs*.
- Adds the context to dependency injection in *Startup.cs*.
- Adds a database connection string to *appsettings.json*.

Database connection string

Visual Studio Visual Studio Code

The connection string specifies [SQL Server LocalDB](#).

JSON  Copy

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SchoolContext": "Server=
(localdb)\\mssqllocaldb;Database=SchoolContext;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

```
}  
}
```

LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. By default, LocalDB creates *.mdf* files in the `C:/Users/<user>` directory.

Update the database context class

The main class that coordinates EF Core functionality for a given data model is the database context class. The context is derived from [Microsoft.EntityFrameworkCore.DbContext](https://docs.microsoft.com/en-us/aspnet/core/data/ef-rp/intro?toc=%2Faspnet%2Fcore%2Ftoc.json&bc=%2Faspnet%2Fcore%2Fbreadcrumb%2Ftoc.json&view=aspnetcore-3.1&tabs=visual-studio). The context specifies which entities are included in the data model. In this project, the class is named `SchoolContext`.

Update *SchoolContext.cs* with the following code:

C#

 Copy

```
using Microsoft.EntityFrameworkCore;  
using ContosoUniversity.Models;  
  
namespace ContosoUniversity.Data  
{  
    public class SchoolContext : DbContext  
    {  
        public SchoolContext (DbContextOptions<SchoolContext> options)  
            : base(options)  
        {  
        }  
  
        public DbSet<Student> Students { get; set; }  
        public DbSet<Enrollment> Enrollments { get; set; }  
        public DbSet<Course> Courses { get; set; }  
    }  
}
```

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

The highlighted code creates a [DbSet<TEntity>](#) property for each entity set. In EF Core terminology:

- An entity set typically corresponds to a database table.
- An entity corresponds to a row in the table.

Since an entity set contains multiple entities, the DbSet properties should be plural names. Since the scaffolding tool created a `student` DbSet, this step changes it to plural `Students`.

To make the Razor Pages code match the new DbSet name, make a global change across the whole project of `_context.Student` to `_context.Students`. There are 8 occurrences.


Build the project to verify there are no compiler errors.

Startup.cs

ASP.NET Core is built with [dependency injection](#). Services (such as the EF Core database context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a database context instance is shown later in the tutorial.

The scaffolding tool automatically registered the context class with the dependency injection container.

- In `ConfigureServices`, the highlighted lines were added by the scaffolder:

C#	 Copy
<pre>public void ConfigureServices(IServiceCollection services) { services.AddRazorPages(); services.AddDbContext<SchoolContext>(options => options.UseSqlServer(Configuration.GetConnectionString("SchoolContext"))); }</pre>	

The name of the connection string is passed in to the context by calling a method on a [DbContextOptions](#) object. For local development, the [ASP.NET Core configuration system](#) reads the connection string from the `appsettings.json` file.

Create the database

Update `Program.cs` to create the database if it doesn't exist:

C#	 Copy
<pre>using ContosoUniversity.Data; using Microsoft.Extensions.DependencyInjection; using Microsoft.AspNetCore.Hosting; using Microsoft.Extensions.Hosting; using Microsoft.Extensions.Logging; using System; namespace ContosoUniversity { public class Program {</pre>	

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

    CreateDbIfNotExists(host);

    host.Run();
}

private static void CreateDbIfNotExists(IHost host)
{
    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;

        try
        {
            var context = services.GetRequiredService<SchoolContext>();
            context.Database.EnsureCreated();
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred creating the DB.");
        }
    }
}

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
}
```

```
}
```

The [EnsureCreated](#) method takes no action if a database for the context exists. If no database exists, it creates the database and schema. `EnsureCreated` enables the following workflow for handling data model changes:

- Delete the database. Any existing data is lost.
- Change the data model. For example, add an `EmailAddress` field.
- Run the app.
- `EnsureCreated` creates a database with the new schema.

This workflow works well early in development when the schema is rapidly evolving, as long as you don't need to preserve data. The situation is different when data that has been entered into the database needs to be preserved. When that is the case, use migrations.

Later in the tutorial series, you delete the database that was created by `EnsureCreated` and use migrations instead. A database that is created by `EnsureCreated` can't be updated by using migrations.

Test the app

- Run the app.
- Select the **Students** link and then **Create New**.
- Test the Edit, Details, and Delete links.

Seed the database

The `EnsureCreated` method creates an empty database. This section adds code that populates the database with test data.

Create *Data/DbInitializer.cs* with the following code:

```
C#
```



```
using ContosoUniversity.Data;
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.Parse("2019-09-01")},
                new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2017-09-01")},
                new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2018-09-01")},
                new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2017-09-01")},
                new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2017-09-01")},
                new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2016-09-01")},
                new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2018-09-01")},
                new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2019-09-01")}
            };

            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},
    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();  
}  
}
```

The code checks if there are any students in the database. If there are no students, it adds test data to the database. It creates the test data in arrays rather than `List<T>` collections to optimize performance.

- In *Program.cs*, replace the `EnsureCreated` call with a `DbInitializer.Initialize` call:

C#

 Copy

```
// context.Database.EnsureCreated();  
DbInitializer.Initialize(context);
```

Visual Studio

Visual Studio Code

Stop the app if it's running, and run the following command in the **Package Manager Console (PMC)**:

PowerShell

 Copy**Drop-Database**

- Restart the app.
- Select the Students page to see the seeded data.

View the database

Visual Studio

Visual Studio Code

- Open **SQL Server Object Explorer** (SSOX) from the **View** menu in Visual Studio.
- In SSOX, select **(localdb)\MSSQLLocalDB > Databases > SchoolContext-{GUID}**. The database name is generated from the context name you provided earlier plus a dash and a GUID.
- Expand the **Tables** node.
- Right-click the **Student** table and click **View Data** to see the columns created and the rows inserted into the table.
- Right-click the **Student** table and click **View Code** to see how the `Student` model maps to the `Student` table schema.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server can handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time. For low traffic situations, the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the [async](#) keyword, `Task<T>` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

C#

 Copy

```
public async Task OnGetAsync()
```

```
public async Task OnGetAsync()  
{  
    Students = await _context.Students.ToListAsync();  
}
```

- The `async` keyword tells the compiler to:
 - Generate callbacks for parts of the method body.
 - Create the `Task` object that's returned.
- The `Task<T>` return type represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF Core context isn't thread safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the database.

For more information about asynchronous programming in .NET, see [Async Overview](#) and [Asynchronous programming with async and await](#).

Next steps

Next tutorial