# Get Started with Entity Framework 6 Code First using MVC 5

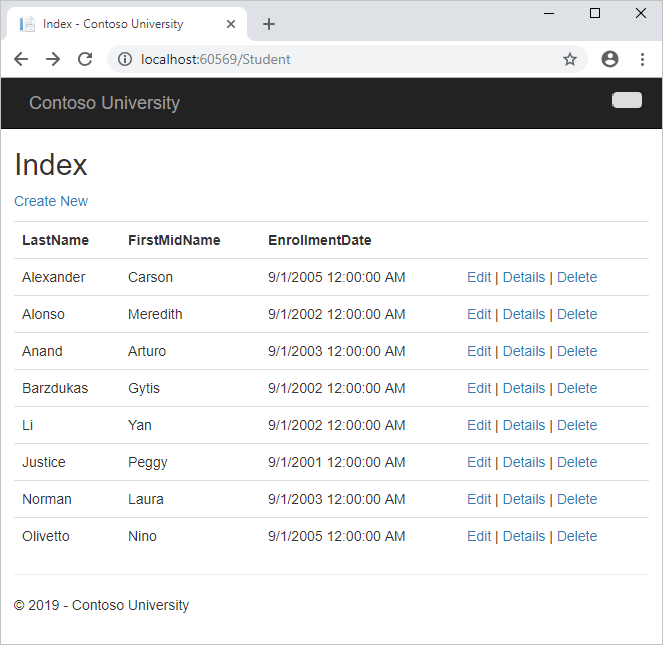
* 19 minutes to read

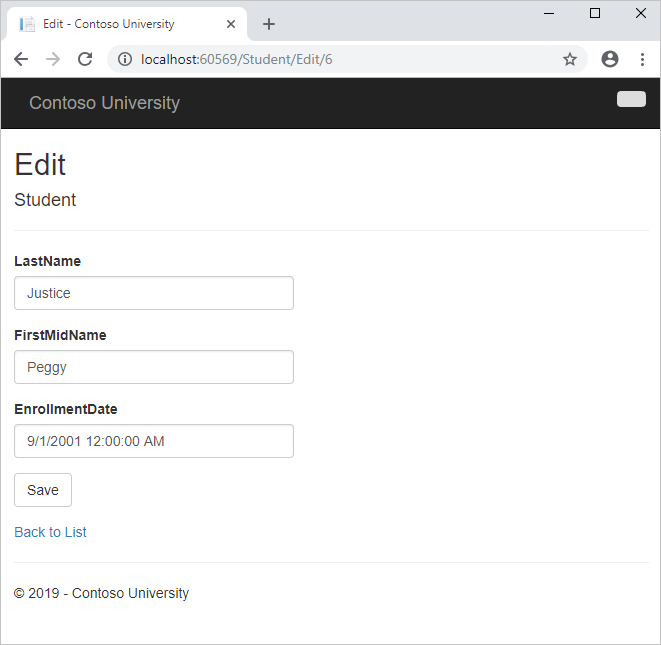
For new development, we recommend [**ASP.NET Core Razor Pages**](https://docs.microsoft.com/en-us/aspnet/core/razor-pages) over ASP.NET MVC controllers and views. For a tutorial series similar to this one using Razor Pages, see [**Tutorial: Get started with Razor Pages in ASP.NET Core**](https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/razor-pages-start). The new tutorial:

* Is easier to follow.
* Provides more EF Core best practices.
* Uses more efficient queries.
* Is more current with the latest API.
* Covers more features.
* Is the preferred approach for new application development.

In this series of tutorials, you learn how to build an ASP.NET MVC 5 application that uses Entity Framework 6 for data access. This tutorial uses the Code First workflow. For information about how to choose between Code First, Database First, and Model First, see [Create a model](https://docs.microsoft.com/en-us/ef/ef6/modeling/).

This tutorial series explains how to build the Contoso University sample application. The sample application is a simple university website. With it, you can view and update student, course, and instructor information. Here are two of the screens you create:





In this tutorial, you:

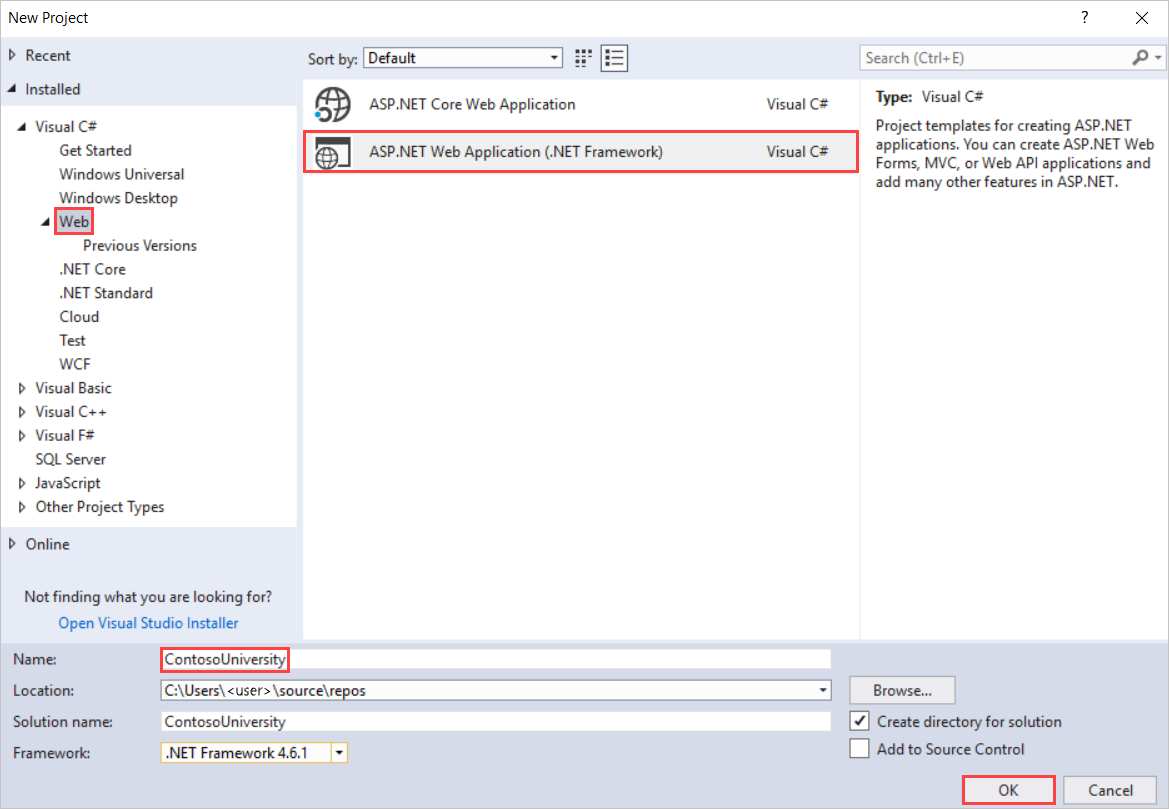
* Create an MVC web app
* Set up the site style
* Install Entity Framework 6
* Create the data model
* Create the database context
* Initialize DB with test data
* Set up EF 6 to use LocalDB
* Create controller and views
* View the database

**Prerequisites**

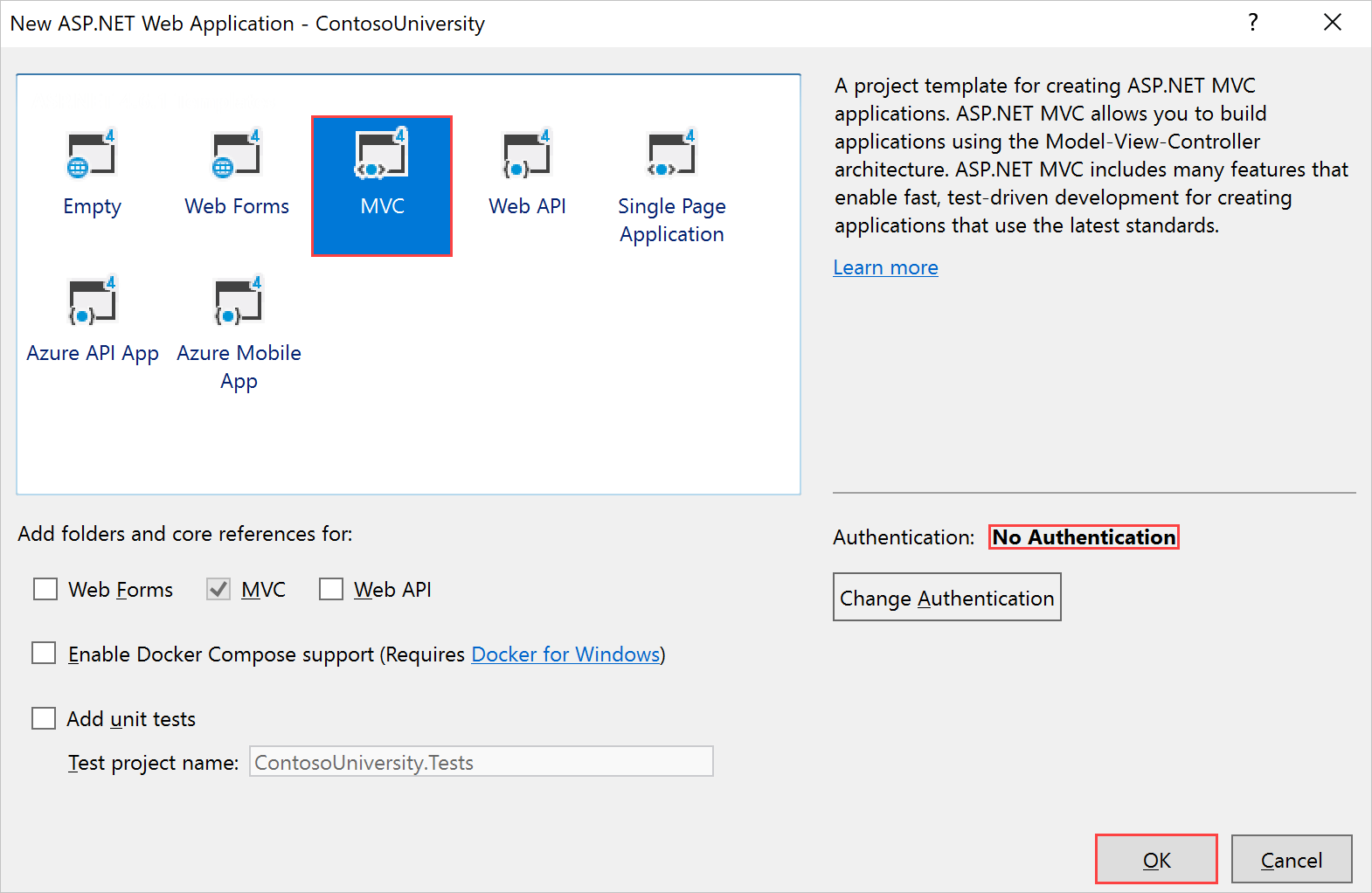
* [Visual Studio 2017](https://visualstudio.microsoft.com/downloads/?utm_medium=microsoft&utm_source=docs.microsoft.com&utm_campaign=button+cta&utm_content=download+vs2017)

**Create an MVC web app**

1. Open Visual Studio and create a C# web project using the **ASP.NET Web Application (.NET Framework)** template. Name the project *ContosoUniversity* and select **OK**.



1. In **New ASP.NET Web Application - ContosoUniversity**, select **MVC**.



**Note**

By default, the **Authentication** option is set to **No Authentication**. For this tutorial, the web app doesn't require users to sign in. Also, it doesn't restrict access based on who's signed in.

1. Select **OK** to create the project.

**Set up the site style**

A few simple changes will set up the site menu, layout, and home page.

1. Open *Views\Shared\\_Layout.cshtml*, and make the following changes:
   * Change each occurrence of "My ASP.NET Application" and "Application name" to "Contoso University".
   * Add menu entries for Students, Courses, Instructors, and Departments, and delete the Contact entry.

The changes are highlighted in the following code snippet:

CSHTMLCopy

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>@ViewBag.Title - Contoso University</title>

@Styles.Render("~/Content/css")

@Scripts.Render("~/bundles/modernizr")

</head>

<body>

<div class="navbar navbar-inverse navbar-fixed-top">

<div class="navbar-inner">

<div class="container">

<button type="button" class="btn btn-navbar" data-toggle="collapse" data-target=".nav-collapse">

<span class="icon-bar"></span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

</button>

@Html.ActionLink("Contoso University", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })

<div class="nav-collapse collapse">

<ul class="nav">

<li>@Html.ActionLink("Home", "Index", "Home")</li>

<li>@Html.ActionLink("About", "About", "Home")</li>

<li>@Html.ActionLink("Students", "Index", "Student")</li>

<li>@Html.ActionLink("Courses", "Index", "Course")</li>

<li>@Html.ActionLink("Instructors", "Index", "Instructor")</li>

<li>@Html.ActionLink("Departments", "Index", "Department")</li>

</ul>

</div>

</div>

</div>

</div>

<div class="container">

@RenderBody()

<hr />

<footer>

<p>&copy; @DateTime.Now.Year - Contoso University</p>

</footer>

</div>

@Scripts.Render("~/bundles/jquery")

@Scripts.Render("~/bundles/bootstrap")

@RenderSection("scripts", required: false)

</body>

</html>

1. In *Views\Home\Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET and MVC with text about this application:

CSHTMLCopy

@{

ViewBag.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 6 in an

ASP.NET MVC 5 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 the tutorial series on the ASP.NET site.</p>

<p><a class="btn btn-default" href="http://www.asp.net/mvc/tutorials/getting-started-with-ef-using-mvc/">See the tutorial &raquo;</a></p>

</div>

<div class="col-md-4">

<h2>Download it</h2>

<p>You can download the completed project.</p>

<p><a class="btn btn-default" href="https://webpifeed.blob.core.windows.net/webpifeed/Partners/ASP.NET%20MVC%20Application%20Using%20Entity%20Framework%20Code%20First.zip">Download &raquo;</a></p>

</div>

</div>

1. Press Ctrl+F5 to run the web site. You see the home page with the main menu.

**Install Entity Framework 6**

1. From the **Tools** menu, choose **NuGet Package Manager**, and then choose **Package Manager Console**.
2. In the **Package Manager Console** window, enter the following command:

textCopy

Install-Package EntityFramework

This step is one of a few steps that this tutorial has you do manually, but that could have been done automatically by the ASP.NET MVC scaffolding feature. You're doing them manually so that you can see the steps required to use Entity Framework (EF). You'll use scaffolding later to create the MVC controller and views. An alternative is to let scaffolding automatically install the EF NuGet package, create the database context class, and create the connection string. When you're ready to do it that way, all you have to do is skip those steps and scaffold your MVC controller after you create your entity classes.

**Create the data model**

Next you'll create entity classes for the Contoso University application. You'll start with the following three entities:

**Course** <-> **Enrollment** <-> **Student**

| **CREATE THE DATA MODEL** | |
| --- | --- |
| **Entities** | **Relationship** |
| Course to Enrollment | One-to-many |
| Student to Enrollment | One-to-many |

There's a one-to-many relationship between Student and Enrollment entities, and there's a one-to-many relationship between Course and Enrollment entities. In other words, a student can be enrolled in any number of courses, and a course can have any number of students enrolled in it.

In the following sections, you'll create a class for each one of these entities.

**Note**

If you try to compile the project before you finish creating all of these entity classes, you'll get compiler errors.

**The Student entity**

* In the *Models* folder, create a class file named *Student.cs* by right-clicking on the folder in **Solution Explorer** and choosing **Add** > **Class**. Replace the template code 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 virtual ICollection<Enrollment> Enrollments { get; set; }

}

}

The ID property will become the primary key column of the database table that corresponds to this class. By default, Entity Framework interprets a property that's named ID or *classname* ID as the primary key.

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 will hold all of the Enrollment entities that are related to that Student entity. In other words, if a given Student row in the database has two related Enrollment rows (rows that contain that student's primary key value in their StudentID foreign key column), that Student entity's Enrollments navigation property will contain those two Enrollment entities.

Navigation properties are typically defined as virtual so that they can take advantage of certain Entity Framework functionality such as *lazy loading*. (Lazy loading will be explained later, in the [Reading Related Data](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/reading-related-data-with-the-entity-framework-in-an-asp-net-mvc-application) tutorial later in this series.)

If a navigation property can hold multiple entities (as in many-to-many or one-to-many relationships), its type must be a list in which entries can be added, deleted, and updated, such as ICollection.

**The Enrollment entity**

* In the *Models* folder, create *Enrollment.cs* and replace the existing code with the following code:

C#Copy

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 virtual Course Course { get; set; }

public virtual Student Student { get; set; }

}

}

The EnrollmentID property will be the primary key; this entity uses the *classname* ID pattern instead of ID by itself as you saw in the Student entity. Ordinarily you would choose one pattern and use it throughout your data model. Here, the variation illustrates that you can use either pattern. In a later tutorial, you'll see how using ID without classname makes it easier to implement inheritance in the data model.

The Grade property is an [enum](https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/data-types/enums). The question mark after the Grade type declaration indicates that the Grade property is [nullable](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/nullable-types/using-nullable-types). 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 can only hold a single Student entity (unlike the Student.Enrollments navigation property you saw earlier, which can hold multiple Enrollment entities).

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

Entity Framework interprets a property as a foreign key property if it's named *<navigation property name><primary key property name>* (for example, StudentID for the Student navigation property since the Student entity's primary key is ID). Foreign key properties can also be named the same simply *<primary key property name>* (for example, CourseID since the Course entity's primary key is CourseID).

**The Course entity**

* In the *Models* folder, create *Course.cs*, replacing the template code with the following code:

C#Copy

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 virtual ICollection<Enrollment> Enrollments { get; set; }

}

}

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

We'll say more about the [DatabaseGeneratedAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations.schema.databasegeneratedattribute) attribute in a later tutorial in this series. Basically, this attribute lets you enter the primary key for the course rather than having the database generate it.

**Create the database context**

The main class that coordinates Entity Framework functionality for a given data model is the *database context* class. You create this class by deriving from the [System.Data.Entity.DbContext](https://msdn.microsoft.com/library/system.data.entity.dbcontext(v=vs.113).aspx) class. In your code, you specify which entities are included in the data model. You can also customize certain Entity Framework behavior. In this project, the class is named SchoolContext.

* To create a folder in the ContosoUniversity project, right-click the project in **Solution Explorer** and click **Add**, and then click **New Folder**. Name the new folder *DAL* (for Data Access Layer). In that folder, create a new class file named *SchoolContext.cs*, and replace the template code with the following code:

C#Copy

using ContosoUniversity.Models;

using System.Data.Entity;

using System.Data.Entity.ModelConfiguration.Conventions;

namespace ContosoUniversity.DAL

{

public class SchoolContext : DbContext

{

public SchoolContext() : base("SchoolContext")

{

}

public DbSet<Student> Students { get; set; }

public DbSet<Enrollment> Enrollments { get; set; }

public DbSet<Course> Courses { get; set; }

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

}

}

}

**Specify entity sets**

This code creates a [DbSet](https://msdn.microsoft.com/library/system.data.entity.dbset(v=vs.113).aspx) property for each entity set. In Entity Framework terminology, an *entity set* typically corresponds to a database table, and an *entity* corresponds to a row in the table.

**Note**

You can omit the DbSet<Enrollment> and DbSet<Course> statements and it would work the same. Entity Framework would include them implicitly because the Student entity references the Enrollment entity and the Enrollment entity references the Course entity.

**Specify the connection string**

The name of the connection string (which you'll add to the Web.config file later) is passed in to the constructor.

C#Copy

public SchoolContext() : base("SchoolContext")

{

}

You could also pass in the connection string itself instead of the name of one that is stored in the Web.config file. For more information about options for specifying the database to use, see [Connection strings and models](https://docs.microsoft.com/en-us/ef/ef6/fundamentals/configuring/connection-strings).

If you don't specify a connection string or the name of one explicitly, Entity Framework assumes that the connection string name is the same as the class name. The default connection string name in this example would then be SchoolContext, the same as what you're specifying explicitly.

**Specify singular table names**

The modelBuilder.Conventions.Remove statement in the [OnModelCreating](https://msdn.microsoft.com/library/system.data.entity.dbcontext.onmodelcreating(v=vs.103).aspx) method prevents table names from being pluralized. If you didn't do this, the generated tables in the database would be named Students, Courses, and Enrollments. Instead, the table names will be Student, Course, and Enrollment. Developers disagree about whether table names should be pluralized or not. This tutorial uses the singular form, but the important point is that you can select whichever form you prefer by including or omitting this line of code.

**Initialize DB with test data**

Entity Framework can automatically create (or drop and re-create) a database for you when the application runs. You can specify that this should be done every time your application runs or only when the model is out of sync with the existing database. You can also write a Seed method that Entity Framework automatically calls after creating the database in order to populate it with test data.

The default behavior is to create a database only if it doesn't exist (and throw an exception if the model has changed and the database already exists). In this section, you'll specify that the database should be dropped and re-created whenever the model changes. Dropping the database causes the loss of all your data. This is generally okay during development, because the Seed method will run when the database is re-created and will re-create your test data. But in production you generally don't want to lose all your data every time you need to change the database schema. Later you'll see how to handle model changes by using Code First Migrations to change the database schema instead of dropping and re-creating the database.

1. In the DAL folder, create a new class file named *SchoolInitializer.cs* and replace the template code with the following code, which causes a database to be created when needed and loads test data into the new database.

C#Copy

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Data.Entity;

using ContosoUniversity.Models;

namespace ContosoUniversity.DAL

{

public class SchoolInitializer : System.Data.Entity. DropCreateDatabaseIfModelChanges<SchoolContext>

{

protected override void Seed(SchoolContext context)

{

var students = new List<Student>

{

new Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2005-09-01")},

new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2002-09-01")},

new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2003-09-01")},

new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2002-09-01")},

new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002-09-01")},

new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2001-09-01")},

new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2003-09-01")},

new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2005-09-01")}

};

students.ForEach(s => context.Students.Add(s));

context.SaveChanges();

var courses = new List<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,}

};

courses.ForEach(s => context.Courses.Add(s));

context.SaveChanges();

var enrollments = new List<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},

};

enrollments.ForEach(s => context.Enrollments.Add(s));

context.SaveChanges();

}

}

}

The Seed method takes the database context object as an input parameter, and the code in the method uses that object to add new entities to the database. For each entity type, the code creates a collection of new entities, adds them to the appropriate DbSet property, and then saves the changes to the database. It isn't necessary to call the SaveChanges method after each group of entities, as is done here, but doing that helps you locate the source of a problem if an exception occurs while the code is writing to the database.

1. To tell Entity Framework to use your initializer class, add an element to the entityFramework element in the application *Web.config* file (the one in the root project folder), as shown in the following example:

XMLCopy

<entityFramework>

<contexts>

<context type="ContosoUniversity.DAL.SchoolContext, ContosoUniversity">

<databaseInitializer type="ContosoUniversity.DAL.SchoolInitializer, ContosoUniversity" />

</context>

</contexts>

<defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory, EntityFramework">

<parameters>

<parameter value="v11.0" />

</parameters>

</defaultConnectionFactory>

<providers>

<provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />

</providers>

</entityFramework>

The context type specifies the fully qualified context class name and the assembly it's in, and the databaseinitializer type specifies the fully qualified name of the initializer class and the assembly it's in. (When you don't want EF to use the initializer, you can set an attribute on the context element: disableDatabaseInitialization="true".) For more information, see [Configuration File Settings](https://docs.microsoft.com/en-us/ef/ef6/fundamentals/configuring/config-file).

An alternative to setting the initializer in the *Web.config* file is to do it in code by adding a Database.SetInitializer statement to the Application\_Start method in the *Global.asax.cs* file. For more information, see [Understanding Database Initializers in Entity Framework Code First](http://www.codeguru.com/csharp/article.php/c19999/Understanding-Database-Initializers-in-Entity-Framework-Code-First.htm).

The application is now set up so that when you access the database for the first time in a given run of the application, Entity Framework compares the database to the model (your SchoolContext and entity classes). If there's a difference, the application drops and re-creates the database.

**Note**

When you deploy an application to a production web server, you must remove or disable code that drops and re-creates the database. You'll do that in a later tutorial in this series.

**Set up EF 6 to use LocalDB**

[LocalDB](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/sql-server-2016-express-localdb?view=sql-server-2017) is a lightweight version of the SQL Server Express database engine. It's easy to install and configure, starts on demand, and runs in user mode. LocalDB runs in a special execution mode of SQL Server Express that enables you to work with databases as *.mdf* files. You can put LocalDB database files in the *App\_Data* folder of a web project if you want to be able to copy the database with the project. The user instance feature in SQL Server Express also enables you to work with *.mdf* files, but the user instance feature is deprecated; therefore, LocalDB is recommended for working with *.mdf* files. LocalDB is installed by default with Visual Studio.

Typically, SQL Server Express is not used for production web applications. LocalDB in particular is not recommended for production use with a web application because it's not designed to work with IIS.

* In this tutorial, you'll work with LocalDB. Open the application *Web.config* file and add a connectionStrings element preceding the appSettings element, as shown in the following example. (Make sure you update the *Web.config* file in the root project folder. There's also a *Web.config* file in the *Views* subfolder that you don't need to update.)

XMLCopy

<connectionStrings>

<add name="SchoolContext" connectionString="Data Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=ContosoUniversity1;Integrated Security=SSPI;" providerName="System.Data.SqlClient"/>

</connectionStrings>

<appSettings>

<add key="webpages:Version" value="3.0.0.0" />

<add key="webpages:Enabled" value="false" />

<add key="ClientValidationEnabled" value="true" />

<add key="UnobtrusiveJavaScriptEnabled" value="true" />

</appSettings>

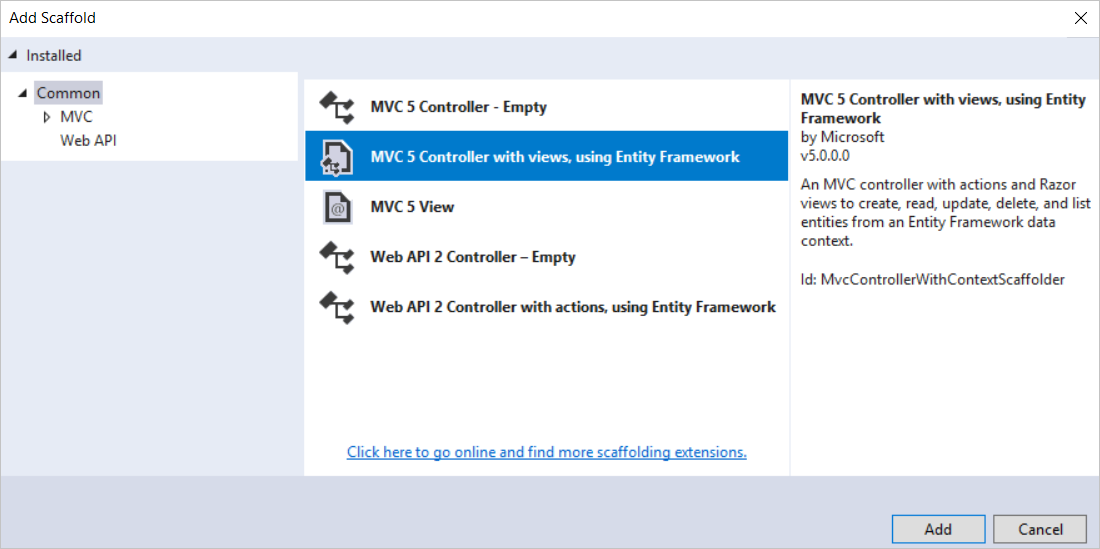
The connection string you've added specifies that Entity Framework will use a LocalDB database named *ContosoUniversity1.mdf*. (The database doesn't exist yet but EF will create it.) If you want to create the database in your *App\_Data* folder, you could add AttachDBFilename=|DataDirectory|\ContosoUniversity1.mdf to the connection string. For more information about connection strings, see [SQL Server Connection Strings for ASP.NET Web Applications](https://docs.microsoft.com/en-us/previous-versions/aspnet/jj653752(v=vs.110)).

You don't actually need a connection string in the *Web.config* file. If you don't supply a connection string, Entity Framework uses a default connection string based on your context class. For more information, see [Code First to a New Database](https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/workflows/new-database).

**Create controller and views**

Now you'll create a web page to display data. The process of requesting the data automatically triggers the creation of the database. You'll begin by creating a new controller. But before you do that, build the project to make the model and context classes available to MVC controller scaffolding.

1. Right-click the **Controllers** folder in **Solution Explorer**, select **Add**, and then click **New Scaffolded Item**.
2. In the **Add Scaffold** dialog box, select **MVC 5 Controller with views, using Entity Framework**, and then choose **Add**.



1. In the **Add Controller** dialog box, make the following selections, and then choose **Add**:
   * Model class: **Student (ContosoUniversity.Models)**. (If you don't see this option in the drop-down list, build the project and try again.)
   * Data context class: **SchoolContext (ContosoUniversity.DAL)**.
   * Controller name: **StudentController** (not StudentsController).
   * Leave the default values for the other fields.

When you click **Add**, the scaffolder creates a *StudentController.cs* file and a set of views (*.cshtml* files) that work with the controller. In the future when you create projects that use Entity Framework, you can also take advantage of some additional functionality of the scaffolder: create your first model class, don't create a connection string, and then in the **Add Controller** box specify **New data context** by selecting the **+** button next to **Data context class**. The scaffolder will create your DbContext class and your connection string as well as the controller and views.

1. Visual Studio opens the *Controllers\StudentController.cs* file. You see that a class variable has been created that instantiates a database context object:

C#Copy

private SchoolContext db = new SchoolContext();

The Index action method gets a list of students from the *Students* entity set by reading the Students property of the database context instance:

C#Copy

public ViewResult Index()

{

return View(db.Students.ToList());

}

The *Student\Index.cshtml* view displays this list in a table:

CSHTMLCopy

<table>

<tr>

<th>

@Html.DisplayNameFor(model => model.LastName)

</th>

<th>

@Html.DisplayNameFor(model => model.FirstMidName)

</th>

<th>

@Html.DisplayNameFor(model => model.EnrollmentDate)

</th>

<th></th>

</tr>

@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>

@Html.ActionLink("Edit", "Edit", new { id=item.ID }) |

@Html.ActionLink("Details", "Details", new { id=item.ID }) |

@Html.ActionLink("Delete", "Delete", new { id=item.ID })

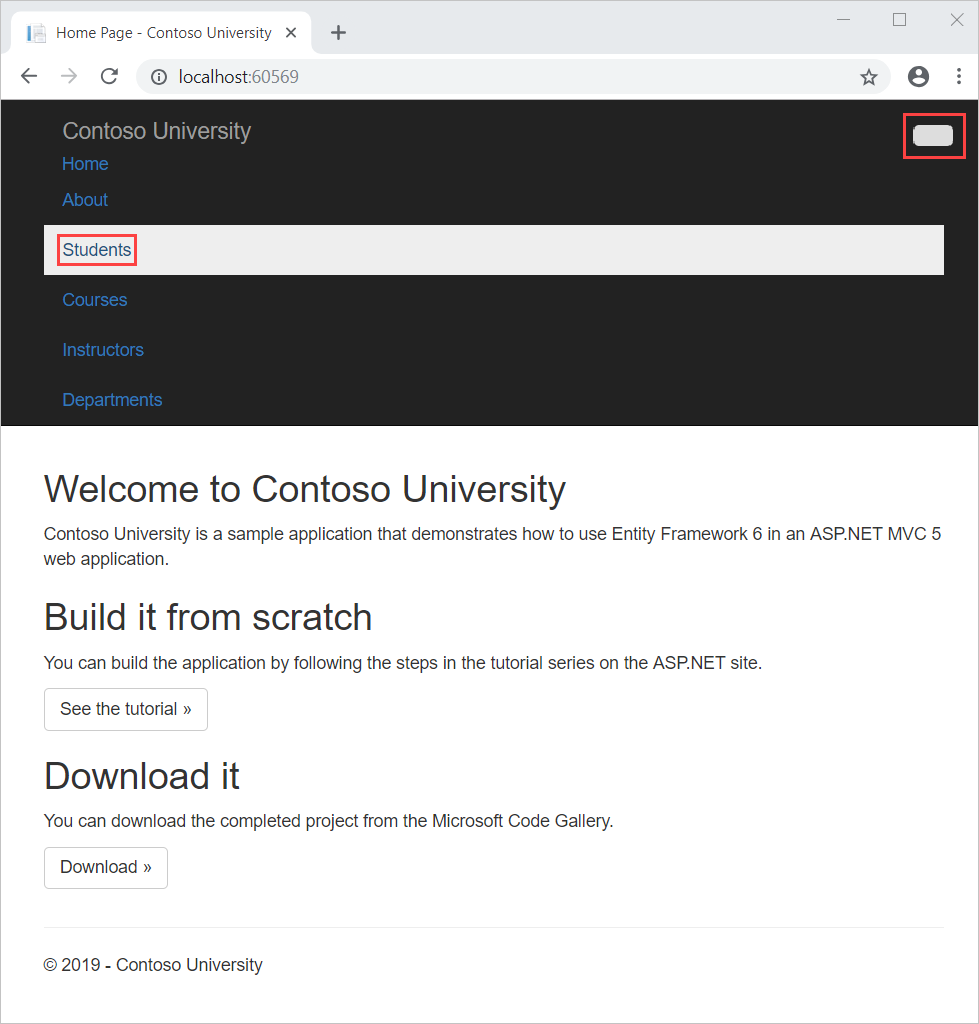
</td>

</tr>

}

1. Press Ctrl+F5 to run the project. (If you get a "Cannot create Shadow Copy" error, close the browser and try again.)

Click the **Students** tab to see the test data that the Seed method inserted. Depending on how narrow your browser window is, you'll see the Student tab link in the top address bar or you'll have to click the upper right corner to see the link.



**View the database**

When you ran the Students page and the application tried to access the database, EF discovered that there was no database and created one. EF then ran the seed method to populate the database with data.

You can use either **Server Explorer** or **SQL Server Object Explorer** (SSOX) to view the database in Visual Studio. For this tutorial, you'll use **Server Explorer**.

1. Close the browser.
2. In **Server Explorer**, expand **Data Connections** (you may need to select the refresh button first), expand **School Context (ContosoUniversity)**, and then expand **Tables** to see the tables in your new database.
3. Right-click the **Student** table and click **Show Table Data** to see the columns that were created and the rows that were inserted into the table.
4. Close the **Server Explorer** connection.

The *ContosoUniversity1.mdf* and *.ldf* database files are in the *%USERPROFILE%* folder.

Because you're using the DropCreateDatabaseIfModelChanges initializer, you could now make a change to the Student class, run the application again, and the database would automatically be re-created to match your change. For example, if you add an EmailAddress property to the Student class, run the Students page again, and then look at the table again, you'll see a new EmailAddress column.

**Conventions**

The amount of code you had to write in order for Entity Framework to be able to create a complete database for you is minimal because of *conventions*, or assumptions that Entity Framework makes. Some of them have already been noted or were used without your being aware of them:

* The pluralized forms of entity class names are used as table names.
* Entity property names are used for column names.
* Entity properties that are named ID or *classname* ID are recognized as primary key properties.
* A property is interpreted as a foreign key property if it's named *<navigation property name><primary key property name>* (for example, StudentID for the Student navigation property since the Student entity's primary key is ID). Foreign key properties can also be named the same simply <primary key property name> (for example, EnrollmentID since the Enrollment entity's primary key is EnrollmentID).

You've seen that conventions can be overridden. For example, you specified that table names shouldn't be pluralized, and you'll see later how to explicitly mark a property as a foreign key property.

**Get the code**

[Download Completed Project](https://webpifeed.blob.core.windows.net/webpifeed/Partners/ASP.NET%20MVC%20Application%20Using%20Entity%20Framework%20Code%20First.zip)

**Additional resources**

For more about EF 6, see these articles:

* [ASP.NET Data Access - Recommended Resources](https://docs.microsoft.com/en-us/aspnet/whitepapers/aspnet-data-access-content-map)
* [Code First Conventions](https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/conventions/built-in)
* [Creating a More Complex Data Model](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-a-more-complex-data-model-for-an-asp-net-mvc-application)

**Next steps**

In this tutorial, you:

* Created an MVC web app
* Set up the site style
* Installed Entity Framework 6
* Created the data model
* Created the database context
* Initialized DB with test data
* Set up EF 6 to use LocalDB
* Created controller and views
* Viewed the database

Advance to the next article to learn how to review and customize the create, read, update, delete (CRUD) code in your controllers and views.

[Implement basic CRUD functionality](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/implementing-basic-crud-functionality-with-the-entity-framework-in-asp-net-mvc-application)

**Recommended content**

**[Tutorial: Implement CRUD Functionality with the Entity Framework in ASP.NET MVC](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/implementing-basic-crud-functionality-with-the-entity-framework-in-asp-net-mvc-application)**

Review and customize the create, read, update, delete (CRUD) code that the MVC scaffolding automatically creates in controllers and views.

**[Tutorial: Read related data with EF in an ASP.NET MVC app](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/reading-related-data-with-the-entity-framework-in-an-asp-net-mvc-application)**

In this tutorial you'll read and display related data — that is, data that the Entity Framework loads into navigation properties.

**[Tutorial: Create a more complex data model for an ASP.NET MVC app](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-a-more-complex-data-model-for-an-asp-net-mvc-application)**

In this tutorial you'll add more entities and relationships and you'll customize the data model by specifying formatting, validation, and database mapping rules.

**[Tutorial: Use EF Migrations in an ASP.NET MVC app and deploy to Azure](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/migrations-and-deployment-with-the-entity-framework-in-an-asp-net-mvc-application)**

In this tutorial, you enable Code First migrations and deploy the application to the cloud in Azure.

**[Tutorial: Update related data with EF in an ASP.NET MVC app](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/updating-related-data-with-the-entity-framework-in-an-asp-net-mvc-application)**

In this tutorial you'll update related data. For most relationships, this can be done by updating either foreign key fields or navigation properties.

**[Creating an Entity Framework Data Model for an ASP.NET MVC Application (1 of 10)](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application)**

A newer version of this tutorial series is available, for Visual Studio 2013, Entity Framework 6, and MVC 5. The Contoso University sample web application de...

**[Tutorial: Learn about advanced EF Scenarios for an MVC 5 Web app](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/advanced-entity-framework-scenarios-for-an-mvc-web-application)**

This tutorial includes introduces several topics that are useful to be aware of when you go beyond the basics of developing ASP.NET web applications that use Entity Framework Code First.

**[Adding a New Field to the Movie Model and Table](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-aspnet-mvc4/adding-a-new-field-to-the-movie-model-and-table)**

Note: An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demo...

Show less

n

using System.Web;

using System.Web.Mvc;

namespace MvcApplication1.Controllers

{

[HandleError]

public class HomeController : Controller

{

public ActionResult Index()

{

return View();

}

public string SayHello()

{

return "Hello!";

}

}

}

In order to be exposed to the universe as an action, a method must meet certain requirements:

* The method must be public.
* The method cannot be a static method.
* The method cannot be an extension method.
* The method cannot be a constructor, getter, or setter.
* The method cannot have open generic types.
* The method is not a method of the controller base class.
* The method cannot contain **ref** or **out** parameters.

Notice that there are no restrictions on the return type of a controller action. A controller action can return a string, a DateTime, an instance of the Random class, or void. The ASP.NET MVC framework will convert any return type that is not an action result into a string and render the string to the browser.

When you add any method that does not violate these requirements to a controller, the method is exposed as a controller action. Be careful here. A controller action can be invoked by anyone connected to the Internet. Do not, for example, create a DeleteMyWebsite() controller action.

**Preventing a Public Method from Being Invoked**

If you need to create a public method in a controller class and you don't want to expose the method as a controller action then you can prevent the method from being invoked by using the [NonAction] attribute. For example, the controller in Listing 2 contains a public method named CompanySecrets() that is decorated with the [NonAction] attribute.

**Listing 2 - Controllers\WorkController.cs**

C#Copy

using System.Web.Mvc;

namespace MvcApplication1.Controllers

{

public class WorkController : Controller

{

[NonAction]

public string CompanySecrets()

{

return "This information is secret.";

}

}

}

If you attempt to invoke the CompanySecrets() controller action by typing /Work/CompanySecrets into the address bar of your browser then you'll get the error message in Figure 1.

**Figure 01**: Invoking a NonAction method([Click to view full-size image](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/creating-an-action-cs/_static/image2.png))

[Previous](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/creating-a-controller-cs) [Next](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/asp-net-mvc-routing-overview-vb)

**Recommended content**

**[Controller.View Method (System.Web.Mvc)](https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.controller.view)**

Creates a ViewResult object using the view name, master-page name, and model that renders a view.

**[Controllers and Routing](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/)**

In this tutorial set, you learn about ASP.NET routing, which maps browser requests to ASP.NET MVC controller actions.

**[Creating a Controller (C#)](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/creating-a-controller-cs)**

In this tutorial, Stephen Walther demonstrates how you can add a controller to an ASP.NET MVC application.

**[ASP.NET MVC Controller Overview (C#)](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/aspnet-mvc-controllers-overview-cs)**

In this tutorial, Stephen Walther introduces you to ASP.NET MVC controllers. You learn how to create new controllers and return different types of action res...

**[ASP.NET MVC Routing Overview (C#)](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/asp-net-mvc-routing-overview-cs)**

In this tutorial, Stephen Walther shows how the ASP.NET MVC framework maps browser requests to controller actions.

**[Creating Custom Routes (C#)](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/creating-custom-routes-cs)**

Learn how to add custom routes to an ASP.NET MVC application. In this tutorial, you learn how to modify the default route table in the Global.asax file.

**[Creating a Route Constraint (C#)](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/creating-a-route-constraint-cs)**

In this tutorial, Stephen Walther demonstrates how you can control how browser requests match routes by creating route constraints with regular expressions.

**[ViewResult Class (System.Web.Mvc)](https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.viewresult)**

Represents a class that is used to render a view by using an IView instance that is returned by an IViewEngine object.

Show less