

Tutorial: Use EF Migrations in an ASP.NET MVC app and deploy to Azure

01/16/2019 • 20 minutes to read •  +6

In this article

[Prerequisites](#)

[Enable Code First migrations](#)

[Deploy to Azure](#)

[Advanced migrations scenarios](#)

[Update specific migration](#)

[Ignore migration changes to database](#)

[Code First initializers](#)

[Get the code](#)

[Additional resources](#)

[Next steps](#)

So far the Contoso University sample web application has been running locally in IIS Express on your development computer. To make a real application available for other people to use over the Internet, you have to deploy it to a web hosting provider. In this tutorial, you enable Code First migrations and deploy the application to the cloud in Azure:

- **Enable Code First Migrations.** The Migrations feature enables you to change the data model and deploy your changes to production by updating the database schema without having to drop and re-create the database.
- **Deploy to Azure.** This step is optional; you can continue with the remaining tutorials without having deployed the project.

We recommend that you use a continuous integration process with source control for deployment, but this tutorial does not cover those topics. For more information, see the [source control](#) and [continuous integration](#) chapters of [Building Real-World Cloud Apps with Azure](#).

In this tutorial, you:

- ✓ Enable Code First migrations
- ✓ Deploy the app in Azure (optional)

Prerequisites

- [Connection Resiliency and Command Interception](#)

Enable Code First migrations

When you develop a new application, your data model changes frequently, and each time the model changes, it gets out of sync with the database. You have configured the Entity Framework to automatically drop and re-create the database each time you change the data model. When you add, remove, or change entity classes or change your `DbContext` class, the next time you run the application it automatically deletes your existing database, creates a new one that matches the model, and seeds it with test data.

This method of keeping the database in sync with the data model works well until you deploy the application to production. When the application is running in production, it is usually storing data that you want to keep, and you don't want to lose everything each time you make a change such as adding a new column. The [Code First Migrations](#) feature solves this problem by enabling Code First to update the database schema instead of dropping and re-creating the database. In this tutorial, you'll deploy the application, and to prepare for that you'll enable Migrations.


1. Disable the initializer that you set up earlier by commenting out or deleting the `contexts` element that you added to the application `Web.config` file.

XML

 Copy


```
<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>
```

2. Also in the application *Web.config* file, change the name of the database in the connection string to ContosoUniversity2.

XML	 Copy
<pre><connectionStrings> <add name="SchoolContext" connectionString="Data Source= (LocalDb)\MSSQLLocalDB;Initial Catalog=ContosoUniversity2;Integrated Security=SSPI;" providerName="System.Data.SqlClient" /> </connectionStrings></pre>	

This change sets up the project so that the first migration creates a new database. This isn't required but you'll see later why it's a good idea.


3. From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**.
4. At the `PM>` prompt enter the following commands:

text	 Copy
<pre>enable-migrations add-migration InitialCreate</pre>	

The `enable-migrations` command creates a *Migrations* folder in the ContosoUniversity project, and it puts in that folder a *Configuration.cs* file that you can edit to configure Migrations.

(If you missed the step above that directs you to change the database name, Migrations will find the existing database and automatically do the `add-migration` command. That's okay, it just means you won't run a test of the migrations code before you deploy the database. Later when you run the `update-database` command nothing will happen because the database already exists.)

Open the *ContosoUniversity\Migrations\Configuration.cs* file. Like the initializer class that you saw earlier, the *Configuration* class includes a `Seed` method.

C#	 Copy
<pre>internal sealed class Configuration : DbMigrationsConfiguration<ContosoUniversity.DAL.SchoolContext> { public Configuration() { AutomaticMigrationsEnabled = false; } }</pre>	

```
protected override void Seed(ContosoUniversity.DAL.SchoolContext
context)
{
    // This method will be called after migrating to the latest
    version.

    // You can use the DbSet<T>.AddOrUpdate() helper extension
    method
    // to avoid creating duplicate seed data. E.g.
    //
    // context.People.AddOrUpdate(
    //     p => p.FullName,
    //     new Person { FullName = "Andrew Peters" },
    //     new Person { FullName = "Brice Lambson" },
    //     new Person { FullName = "Rowan Miller" }
    // );
    //
}
```

The purpose of the [Seed](#) method is to enable you to insert or update test data after Code First creates or updates the database. The method is called when the database is created and every time the database schema is updated after a data model change.

Set up the Seed method

When you drop and re-create the database for every data model change, you use the initializer class's `Seed` method to insert test data, because after every model change the database is dropped and all the test data is lost. With Code First Migrations, test data is retained after database changes, so including test data in the [Seed](#) method is typically not necessary. In fact, you don't want the `Seed` method to insert test data if you'll be using Migrations to deploy the database to production, because the `Seed` method will run in production. In that case, you want the `Seed` method to insert into the database only the data that you need in production. For example, you might want the database to include actual department names in the `Department` table when the application becomes available in production.

For this tutorial, you'll be using Migrations for deployment, but your `Seed` method will insert test data anyway in order to make it easier to see how application functionality works without having to manually insert a lot of data.

1. Replace the contents of the *Configuration.cs* file with the following code, which loads test data into the new database.

C#

 Copy

```

namespace ContosoUniversity.Migrations
{
    using ContosoUniversity.Models;
    using System;
    using System.Collections.Generic;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;

    internal sealed class Configuration :
    DbMigrationsConfiguration<ContosoUniversity.DAL.SchoolContext>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
        }

        protected override void
    Seed(ContosoUniversity.DAL.SchoolContext context)
        {
            var students = new List<Student>
            {
                new Student { FirstMidName = "Carson", LastName =
    "Alexander",
                    EnrollmentDate = DateTime.Parse("2010-09-01") },
                new Student { FirstMidName = "Meredith", LastName =
    "Alonso",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Arturo", LastName =
    "Anand",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Gytis", LastName =
    "Barzdukas",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Yan", LastName =
    "Li",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Peggy", LastName =
    "Justice",
                    EnrollmentDate = DateTime.Parse("2011-09-01") },
                new Student { FirstMidName = "Laura", LastName =
    "Norman",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Nino", LastName =
    "Olivetto",
                    EnrollmentDate = DateTime.Parse("2005-08-11") }
            };
            students.ForEach(s => context.Students.AddOrUpdate(p =>
    p.LastName, 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.AddOrUpdate(p =>
p.Title, s));
    context.SaveChanges();

    var enrollments = new List<Enrollment>
    {
        new Enrollment {
            StudentID = students.Single(s => s.LastName ==
"Alexander").ID,
            CourseID = courses.Single(c => c.Title ==
"Chemistry" ).CourseID,
            Grade = Grade.A
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName ==
"Alexander").ID,
            CourseID = courses.Single(c => c.Title ==
"Microeconomics" ).CourseID,
            Grade = Grade.C
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName ==
"Alexander").ID,
            CourseID = courses.Single(c => c.Title ==
"Macroeconomics" ).CourseID,
            Grade = Grade.B
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName ==
"Alonso").ID,
            CourseID = courses.Single(c => c.Title ==
"Calculus" ).CourseID,
            Grade = Grade.B
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName ==
"Alonso").ID,
            CourseID = courses.Single(c => c.Title ==
"Trigonometry" ).CourseID,
            Grade = Grade.B
        },
    },

```

```

        new Enrollment {
            StudentID = students.Single(s => s.LastName ==
"Alonso").ID,
            CourseID = courses.Single(c => c.Title ==
"Composition" ).CourseID,
            Grade = Grade.B
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName ==
"Anand").ID,
            CourseID = courses.Single(c => c.Title ==
"Chemistry" ).CourseID
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName ==
"Anand").ID,
            CourseID = courses.Single(c => c.Title ==
"Microeconomics").CourseID,
            Grade = Grade.B
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName ==
"Barzdukas").ID,
            CourseID = courses.Single(c => c.Title ==
"Chemistry").CourseID,
            Grade = Grade.B
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName ==
"Li").ID,
            CourseID = courses.Single(c => c.Title ==
"Composition").CourseID,
            Grade = Grade.B
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName ==
"Justice").ID,
            CourseID = courses.Single(c => c.Title ==
"Literature").CourseID,
            Grade = Grade.B
        }
    };

    foreach (Enrollment e in enrollments)
    {
        var enrollmentInDataBase = context.Enrollments.Where(
            s =>
                s.Student.ID == e.StudentID &&
                s.Course.CourseID ==
e.CourseID).SingleOrDefault();
        if (enrollmentInDataBase == null)
        {
            context.Enrollments.Add(e);
        }
    }
}

```

```
        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.

Some of the statements that insert data use the [AddOrUpdate](#) method to perform an "upsert" operation. Because the `Seed` method runs every time you execute the update-database command, typically after each migration, you can't just insert data, because the rows you are trying to add will already be there after the first migration that creates the database. The "upsert" operation prevents errors that would happen if you try to insert a row that already exists, but it **overrides** any changes to data that you may have made while testing the application. With test data in some tables you might not want that to happen: in some cases when you change data while testing you want your changes to remain after database updates. In that case you want to do a conditional insert operation: insert a row only if it doesn't already exist. The `Seed` method uses both approaches.

The first parameter passed to the [AddOrUpdate](#) method specifies the property to use to check if a row already exists. For the test student data that you are providing, the `LastName` property can be used for this purpose since each last name in the list is unique:

C#

 Copy

```
context.Students.AddOrUpdate(p => p.LastName, s)
```

This code assumes that last names are unique. If you manually add a student with a duplicate last name, you'll get the following exception the next time you perform a migration:

Sequence contains more than one element

For information about how to handle redundant data such as two students named "Alexander Carson", see [Seeding and Debugging Entity Framework \(EF\) DBs](#) on

Rick Anderson's blog. For more information about the `AddOrUpdate` method, see [Take care with EF 4.3 AddOrUpdate Method](#) on Julie Lerman's blog.

The code that creates `Enrollment` entities assumes you have the `ID` value in the entities in the `students` collection, although you didn't set that property in the code that creates the collection.

C#

 Copy

```
new Enrollment {  
    StudentID = students.Single(s => s.LastName == "Alexander").ID,  
    CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,  
    Grade = Grade.A  
},
```

You can use the `ID` property here because the `ID` value is set when you call `SaveChanges` for the `students` collection. EF automatically gets the primary key value when it inserts an entity into the database, and it updates the `ID` property of the entity in memory.

The code that adds each `Enrollment` entity to the `Enrollments` entity set doesn't use the `AddOrUpdate` method. It checks if an entity already exists and inserts the entity if it doesn't exist. This approach will preserve changes you make to an enrollment grade by using the application UI. The code loops through each member of the `Enrollment` [List](#) and if the enrollment is not found in the database, it adds the enrollment to the database. The first time you update the database, the database will be empty, so it will add each enrollment.

C#

 Copy

```
foreach (Enrollment e in enrollments)  
{  
    var enrollmentInDataBase = context.Enrollments.Where(  
        s => s.Student.ID == e.Student.ID &&  
            s.Course.CourseID == e.Course.CourseID).SingleOrDefault();  
    if (enrollmentInDataBase == null)  
    {  
        context.Enrollments.Add(e);  
    }  
}
```

2. Build the project.

Execute the first migration

When you executed the `add-migration` command, Migrations generated the code that would create the database from scratch. This code is also in the *Migrations* folder, in the file named `<timestamp>_InitialCreate.cs`. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets, and the `Down` method deletes them.

C#



```
public partial class InitialCreate : DbMigration
{
    public override void Up()
    {
        CreateTable(
            "dbo.Course",
            c => new
            {
                CourseID = c.Int(nullable: false),
                Title = c.String(),
                Credits = c.Int(nullable: false),
            })
            .PrimaryKey(t => t.CourseID);

        CreateTable(
            "dbo.Enrollment",
            c => new
            {
                EnrollmentID = c.Int(nullable: false, identity: true),
                CourseID = c.Int(nullable: false),
                StudentID = c.Int(nullable: false),
                Grade = c.Int(),
            })
            .PrimaryKey(t => t.EnrollmentID)
            .ForeignKey("dbo.Course", t => t.CourseID, cascadeDelete: true)
            .ForeignKey("dbo.Student", t => t.StudentID, cascadeDelete:
true)
            .Index(t => t.CourseID)
            .Index(t => t.StudentID);

        CreateTable(
            "dbo.Student",
            c => new
            {
                ID = c.Int(nullable: false, identity: true),
                LastName = c.String(),
                FirstMidName = c.String(),
                EnrollmentDate = c.DateTime(nullable: false),
            })
            .PrimaryKey(t => t.ID);
    }

    public override void Down()
```

```
{
    DropForeignKey("dbo.Enrollment", "StudentID", "dbo.Student");
    DropForeignKey("dbo.Enrollment", "CourseID", "dbo.Course");
    DropIndex("dbo.Enrollment", new[] { "StudentID" });
    DropIndex("dbo.Enrollment", new[] { "CourseID" });
    DropTable("dbo.Student");
    DropTable("dbo.Enrollment");
    DropTable("dbo.Course");
}
```

Migrations calls the `up` method to implement the data model changes for a migration. When you enter a command to roll back the update, Migrations calls the `Down` method.

This is the initial migration that was created when you entered the `add-migration InitialCreate` command. The parameter (`InitialCreate` in the example) is used for the file name and can be whatever you want; you typically choose a word or phrase that summarizes what is being done in the migration. For example, you might name a later migration `"AddDepartmentTable"`.

If you created the initial migration when the database already exists, the database creation code is generated but it doesn't have to run because the database already matches the data model. When you deploy the app to another environment where the database doesn't exist yet, this code will run to create your database, so it's a good idea to test it first. That's why you changed the name of the database in the connection string earlier—so that migrations can create a new one from scratch.

1. In the **Package Manager Console** window, enter the following command:

```
update-database
```

The `update-database` command runs the `up` method to create the database and then it runs the `Seed` method to populate the database. The same process will run automatically in production after you deploy the application, as you'll see in the following section.

2. Use **Server Explorer** to inspect the database as you did in the first tutorial, and run the application to verify that everything still works the same as before.

Deploy to Azure

So far the application has been running locally in IIS Express on your development computer. To make it available for other people to use over the Internet, you have to deploy it to a web hosting provider. In this section of the tutorial, you'll deploy it to Azure. This section is optional; you can skip this and continue with the following tutorial,

or you can adapt the instructions in this section for a different hosting provider of your choice.

Use Code First migrations to deploy the database

To deploy the database, you'll use Code First Migrations. When you create the publish profile that you use to configure settings for deploying from Visual Studio, you'll select a check box labeled **Update Database**. This setting causes the deployment process to automatically configure the application *Web.config* file on the destination server so that Code First uses the `MigrateDatabaseToLatestVersion` initializer class.

Visual Studio doesn't do anything with the database during the deployment process while it is copying your project to the destination server. When you run the deployed application and it accesses the database for the first time after deployment, Code First checks if the database matches the data model. If there's a mismatch, Code First automatically creates the database (if it doesn't exist yet) or updates the database schema to the latest version (if a database exists but doesn't match the model). If the application implements a `Migrations Seed` method, the method runs after the database is created or the schema is updated.

Your `Migrations Seed` method inserts test data. If you were deploying to a production environment, you would have to change the `Seed` method so that it only inserts data that you want to be inserted into your production database. For example, in your current data model you might want to have real courses but fictional students in the development database. You can write a `Seed` method to load both in development, and then comment out the fictional students before you deploy to production. Or you can write a `Seed` method to load only courses, and enter the fictional students in the test database manually by using the application's UI.

Get an Azure account

You'll need an Azure account. If you don't already have one, but you do have a Visual Studio subscription, you can [activate your subscription benefits](#). Otherwise, you can create a free trial account in just a couple of minutes. For details, see [Azure Free Trial](#).

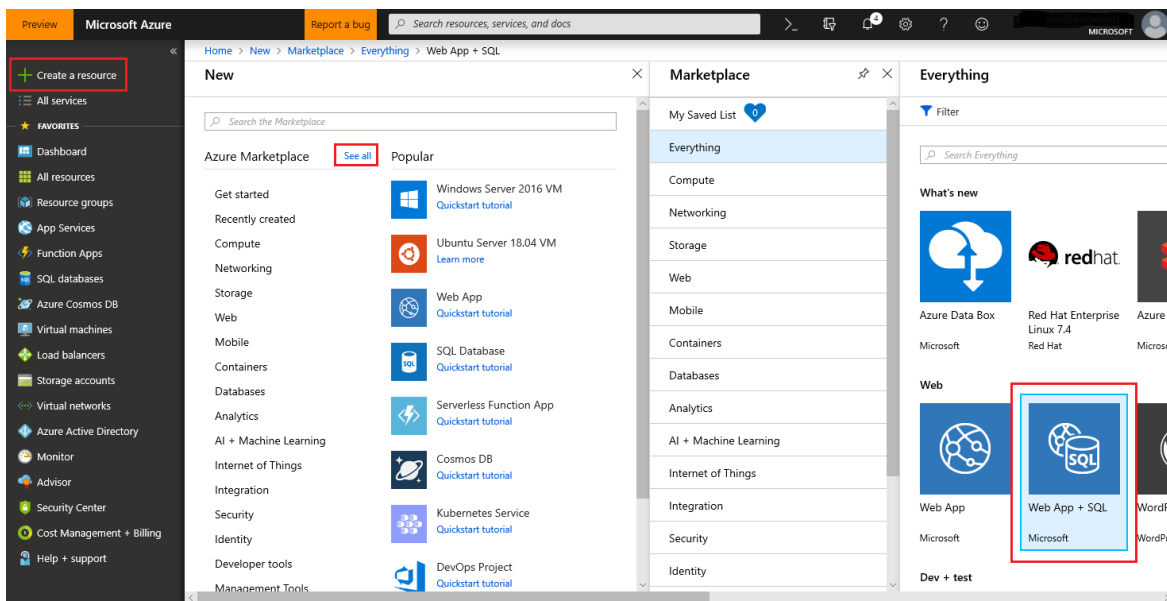
Create a web site and a SQL database in Azure

Your web app in Azure will run in a shared hosting environment, which means it runs on virtual machines (VMs) that are shared with other Azure clients. A shared hosting environment is a low-cost way to get started in the cloud. Later, if your web traffic

increases, the application can scale to meet the need by running on dedicated VMs. To learn more about Pricing Options for Azure App Service, read [App Service pricing](#).

You'll deploy the database to Azure SQL database. SQL database is a cloud-based relational database service that is built on SQL Server technologies. Tools and applications that work with SQL Server also work with SQL database.

1. In the [Azure Management Portal](#), choose **Create a resource** in the left tab and then choose **See all** on the **New** pane (or *blade*) to see all available resources. Choose **Web App + SQL** in the **Web** section of the **Everything** blade. Finally, choose **Create**.



The form to create a new **New Web App + SQL** resource opens.

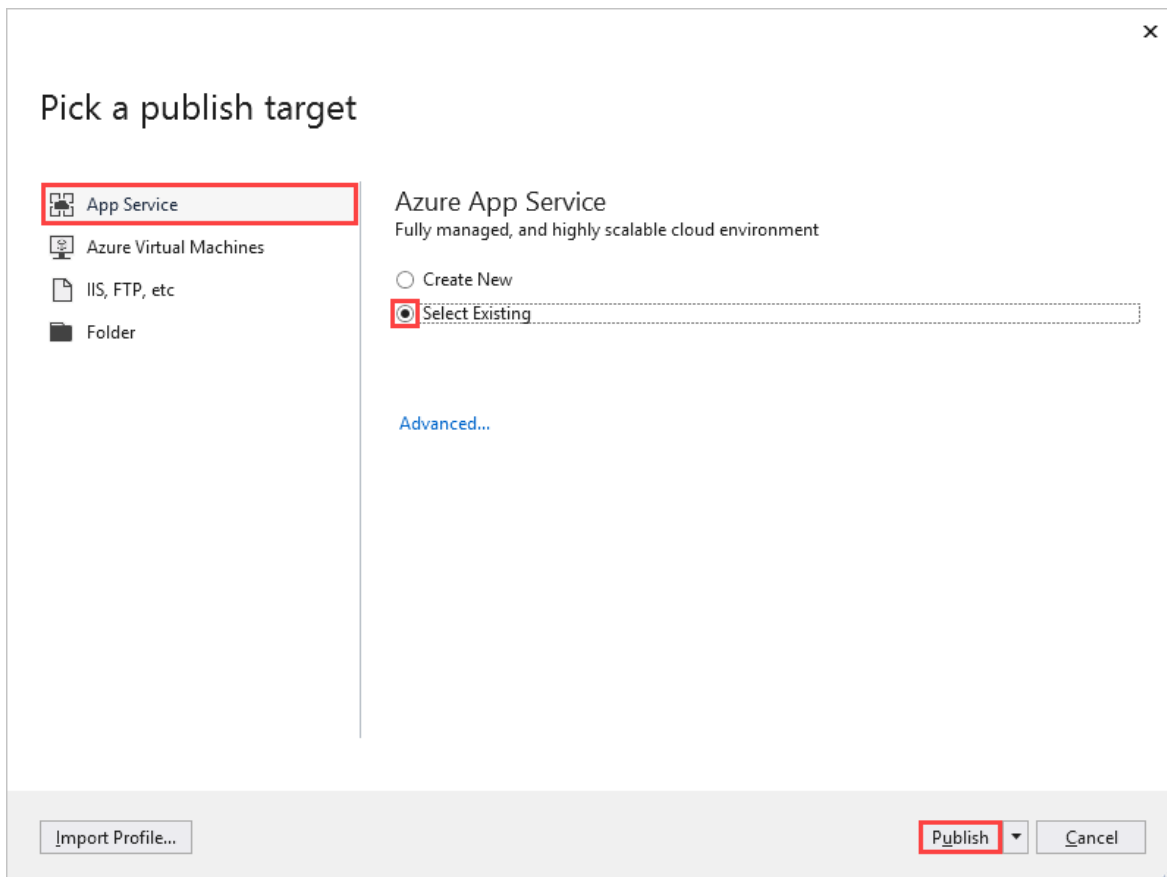
2. Enter a string in the **App name** box to use as the unique URL for your application. The complete URL will consist of what you enter here plus the default domain of Azure App Services (.azurewebsites.net). If the **App name** is already taken, the Wizard notifies you with a red *The app name is not available* message. If the **App name** is available, you see a green checkmark.
3. In the **Subscription** box, choose the Azure Subscription in which you want the **App Service** to reside.
4. In the **Resource Group** text box, choose a Resource Group or create a new one. This setting specifies which data center your web site will run in. For more information about Resource Groups, see [Resource groups](#).
5. Create a new **App Service Plan** by clicking the *App Service section*, **Create New**, and fill in **App Service plan** (can be same name as App Service), **Location**, and **Pricing tier** (there is a free option).

6. Click **SQL Database**, and then choose **Create a new database** or select an existing database.
7. In the **Name** box, enter a name for your database.
8. Click the **Target Server** box, and then select **Create a new server**. Alternatively, if you previously created a server, you can select that server from list of available servers.
9. Choose **Pricing tier** section, choose *Free*. If additional resources are needed, the database can be scaled up at any time. To learn more on Azure SQL Pricing, see [Azure SQL Database pricing](#).
10. Modify [collation](#) as needed.
11. Enter an administrator **SQL Admin Username** and **SQL Admin Password**.
 - If you selected **New SQL Database server**, define a new name and password that you'll use later when you access the database.
 - If you selected a server that you created previously, enter credentials for that server.
12. Telemetry collection can be enabled for App Service using Application Insights. With little configuration, Application Insights collects valuable event, exception, dependency, request, and trace information. To learn more about Application Insights, see [Azure Monitor](#).
13. Click **Create** at the bottom to indicate that you're finished.

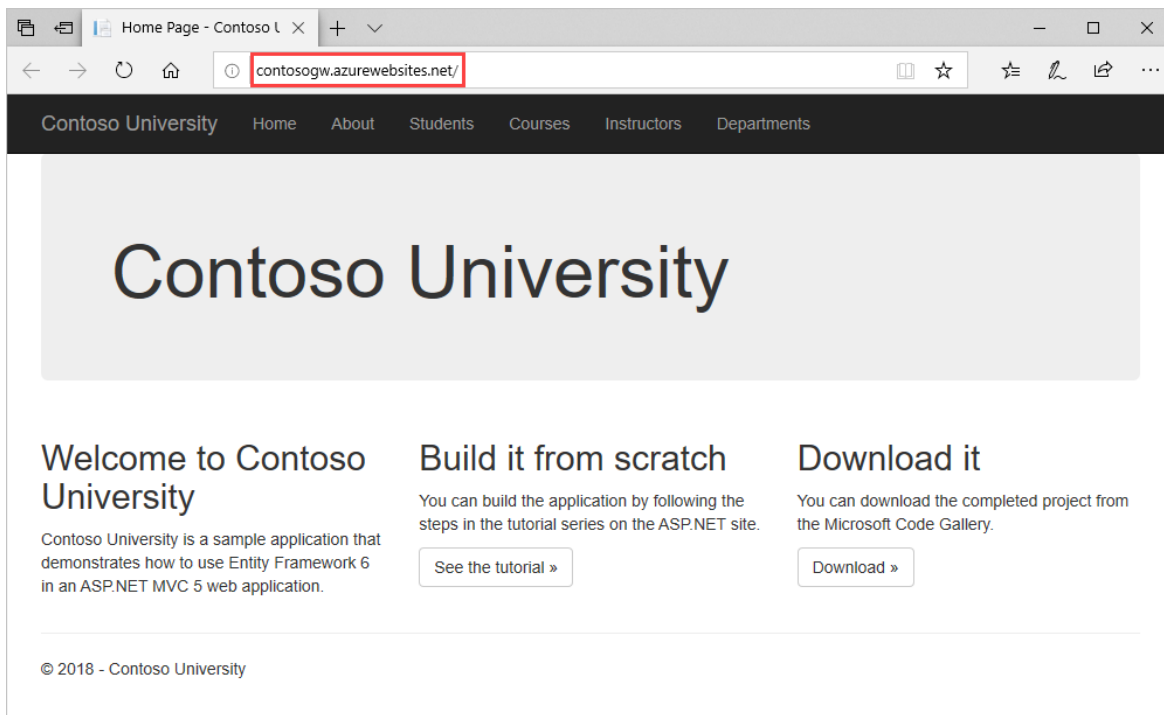
The Management Portal returns to the Dashboard page, and the **Notifications** area at the top of the page shows that the site is being created. After a while (typically less than a minute), there's a notification that the Deployment succeeded. In the navigation bar at the left, the new App Service appears in the **App Services** section and the new SQL database appears in the **SQL databases** section.

Deploy the app to Azure

1. In Visual Studio, right-click the project in **Solution Explorer** and select **Publish** from the context menu.
2. On the **Pick a publish target** page, choose **App Service** and then **Select Existing**, and then choose **Publish**.



3. If you haven't previously added your Azure subscription in Visual Studio, perform the steps on the screen. These steps enable Visual Studio to connect to your Azure subscription so that the list of **App Services** will include your web site.
4. On the **App Service** page, select the **Subscription** you added the App Service to. Under **View**, select **Resource Group**. Expand the resource group you added the App Service to, and then select the App Service. Choose **OK** to publish the app.
5. The **Output** window shows what deployment actions were taken and reports successful completion of the deployment.
6. Upon successful deployment, the default browser automatically opens to the URL of the deployed web site.



Your app is now running in the cloud.

At this point, the *SchoolContext* database has been created in the Azure SQL database because you selected **Execute Code First Migrations (runs on app start)**. The *Web.config* file in the deployed web site has been changed so that the [MigrateDatabaseToLatestVersion](#) initializer runs the first time your code reads or writes data in the database (which happened when you selected the **Students** tab):

```
</runtime>
<entityFramework>
  <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory"
    <parameters>
      <parameter value="v11.0" />
    </parameters>
  </defaultConnectionFactory>
  <contexts>
    <context type="ContosoUniversity.Models.SchoolContext, ContosoUniversity">
      <databaseInitializer type="System.Data.Entity.MigrateDatabaseToLatestVersion`2[
        [ContosoUniversity.Models.SchoolContext, ContosoUniversity],
        [ContosoUniversity.Migrations.Configuration, ContosoUniversi
          EntityFramework, PublicKeyToken=b77a5c561934e089]">
        <parameters>
          <parameter value="SchoolContext_DatabasePublish"/>
        </parameters>
      </databaseInitializer>
    </context>
  </contexts>
</entityFramework>
</configuration>
```

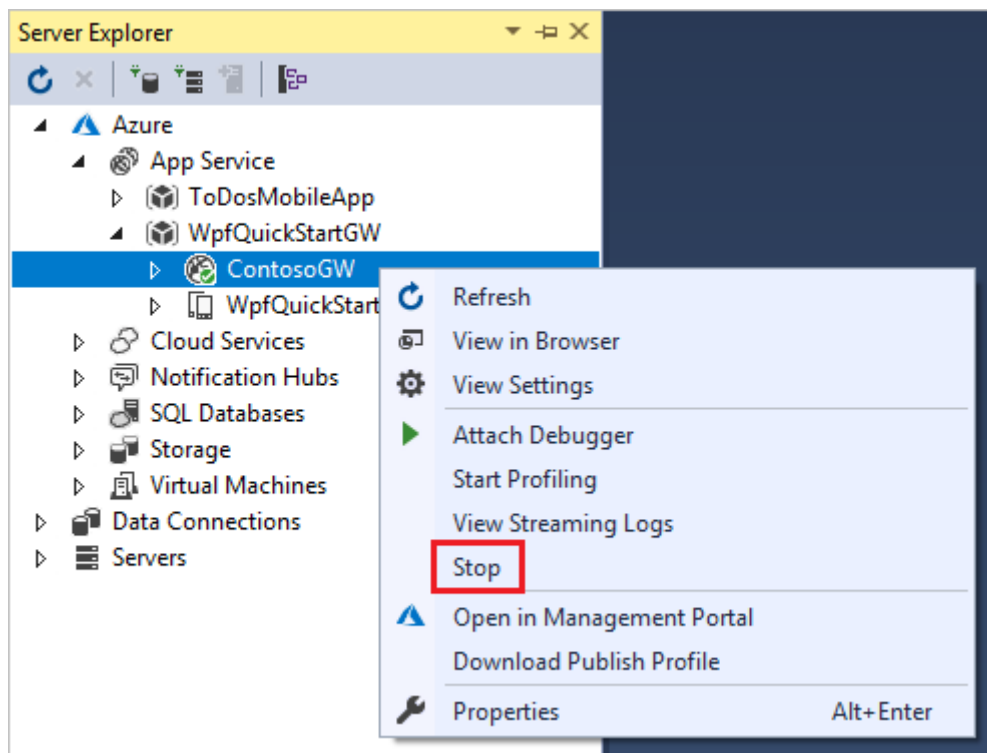
The deployment process also created a new connection string (*SchoolContext_DatabasePublish*) for Code First Migrations to use for updating the database schema and seeding the database.

```
<connectionStrings>
  <add name="SchoolContext" connectionString="Data Source=tcp:d015leqjdx.database.winc
  <add name="SchoolContext_DatabasePublish" connectionString="Data Source=tcp:d015leqj
</connectionStrings>
```


You can find the deployed version of the Web.config file on your own computer in *ContosoUniversity\obj\Release\Package\PackageTmp\Web.config*. You can access the deployed *Web.config* file itself by using FTP. For instructions, see [ASP.NET Web Deployment using Visual Studio: Deploying a Code Update](#). Follow the instructions that start with "To use an FTP tool, you need three things: the FTP URL, the user name, and the password."

ⓘ Note

The web app doesn't implement security, so anyone who finds the URL can change the data. For instructions on how to secure the web site, see [Deploy a Secure ASP.NET MVC app with Membership, OAuth, and SQL database to Azure](#). You can prevent other people from using the site by stopping the service using the Azure Management Portal or **Server Explorer** in Visual Studio.



Advanced migrations scenarios

If you deploy a database by running migrations automatically as shown in this tutorial, and you are deploying to a web site that runs on multiple servers, you could get multiple servers trying to run migrations at the same time. Migrations are atomic, so if two servers try to run the same migration, one will succeed and the other will fail (assuming the operations can't be done twice). In that scenario if you want to avoid those issues, you can call migrations manually and set up your own code so that it only happens once. For more information, see [Running and Scripting Migrations from Code](#)

on Rowan Miller's blog and [Migrate.exe](#) (for executing migrations from the command line).

For information about other migrations scenarios, see [Migrations Screencast Series](#).

Update specific migration

```
update-database -target MigrationName
```

The `update-database -target MigrationName` command runs the targeted migration.

Ignore migration changes to database

```
Add-migration MigrationName -ignoreChanges
```

`ignoreChanges` creates an empty migration with the current model as a snapshot.

Code First initializers

In the deployment section, you saw the [MigrateDatabaseToLatestVersion](#) initializer being used. Code First also provides other initializers, including [CreateDatabaseIfNotExists](#) (the default), [DropCreateDatabaseIfModelChanges](#) (which you used earlier) and [DropCreateDatabaseAlways](#). The `DropCreateAlways` initializer can be useful for setting up conditions for unit tests. You can also write your own initializers, and you can call an initializer explicitly if you don't want to wait until the application reads from or writes to the database.

For more information about initializers, see [Understanding Database Initializers in Entity Framework Code First](#) and chapter 6 of the book [Programming Entity Framework: Code First](#) by Julie Lerman and Rowan Miller.

Get the code

[Download the Completed Project](#)

Additional resources

Links to other Entity Framework resources can be found in [ASP.NET Data Access - Recommended Resources](#).

Next steps

In this tutorial, you:

- ✓ Enabled Code First migrations
- ✓ Deployed the app in Azure (optional)

Advance to the next article to learn how to create a more complex data model for an ASP.NET MVC Application.

Create a more complex data model

Is this page helpful?

 Yes  No
