Getting Started with Entity Framework 6 Code First using MVC 5

Tom Dykstra, Rick Anderson

Step By Step, Guide

Microsoft

Code First Migrations and Deployment with the Entity Framework in an ASP.NET MVC Application

So far the 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'll deploy the Contoso University application to the cloud in a Windows Azure Web Site.

The tutorial contains the following sections:

- 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 Windows Azure. This step is optional; you can continue with the remaining tutorials without having deployed the project.

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.

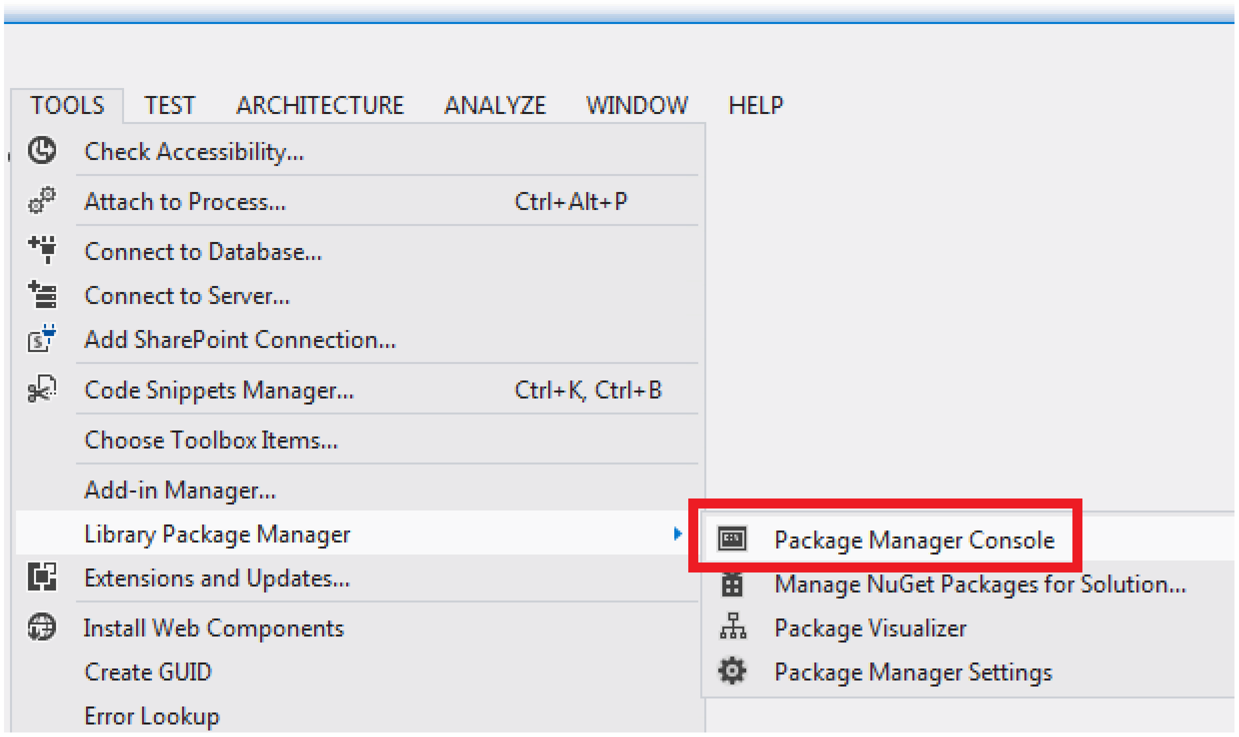
```
<entityFramework>
  <!--<contexts>
    <context type="ContosoUniversity.DAL.SchoolContext, 
ContosoUniversity">
      <databaseInitializer 
type="ContosoUniversity.DAL.SchoolInitializer, ContosoUniversity" />
    </context>
  </contexts>-->
  <defaultConnectionFactory
type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory,
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.

```
<connectionStrings>
  <add name="SchoolContext" connectionString="Data
Source=(LocalDb)\v11.0;Initial Catalog=ContosoUniversity2;Integrated
Security=SSPI;" providerName="System.Data.SqlClient" />
</connectionStrings>
```

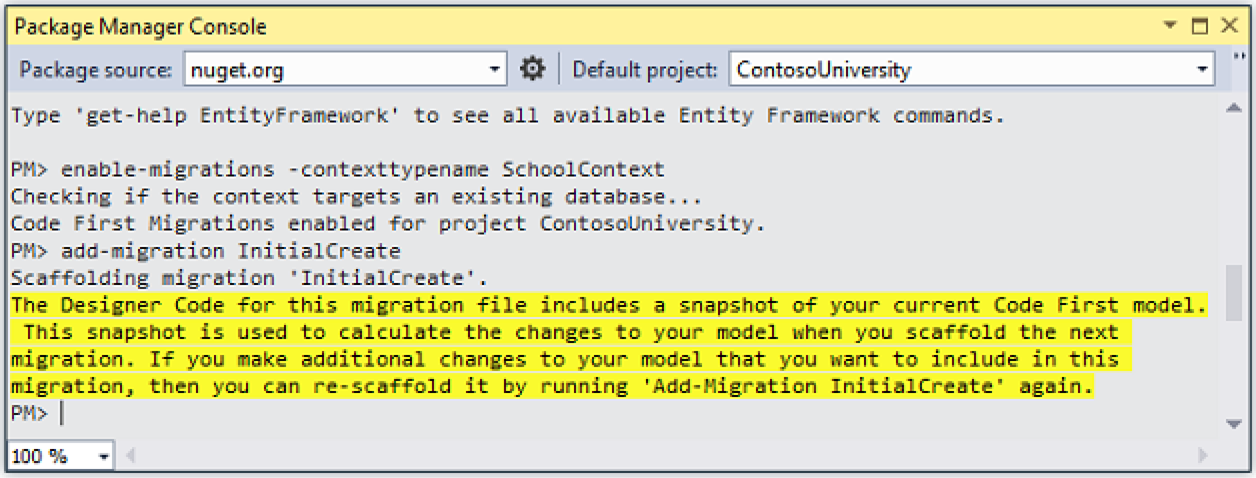
   This change sets up the project so that the first migration will create a new database. This isn't required but you'll see later why it's a good idea.

3. From the **Tools** menu, click **Library Package Manager** and then **Package Manager Console**.



4. At the `PM>` prompt enter the following commands:

```
enable-migrations
add-migration InitialCreate
```

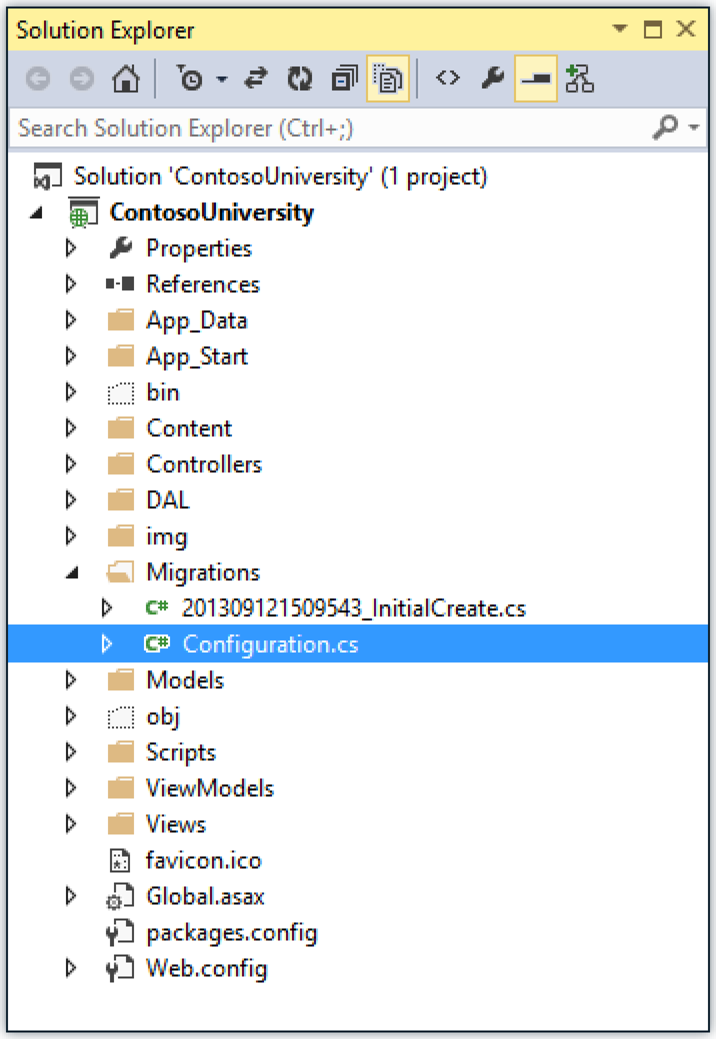


```
Type 'get-help EntityFramework' to see all available Entity Framework commands.

PM> enable-migrations -contexttypename SchoolContext
Checking if the context targets an existing database...
Code First Migrations enabled for project ContosoUniversity.
PM> add-migration InitialCreate
Scaffolding migration 'InitialCreate'.
The Designer Code for this migration file includes a snapshot of your current Code First model.
 This snapshot is used to calculate the changes to your model when you scaffold the next
migration. If you make additional changes to your model that you want to include in this
migration, then you can re-scaffold it by running 'Add-Migration InitialCreate' again.
PM>
```

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 OK, 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 will already exist.)



Like the initializer class that you saw earlier, the `Configuration` class includes a `Seed` method.

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

    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 are dropping and re-creating 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 will load test data into the new database.

```
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:

```
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.

```
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.

```
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.

```
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");
    }
}
```

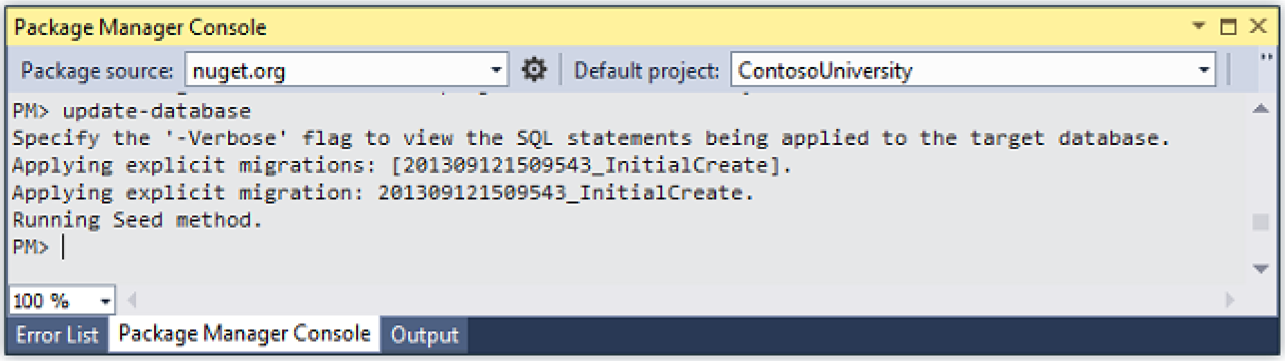
Migrations calls the `Up` method to implement the data model changes for a migration. When you enter a command to roll back the update, Migrations calls the `Down` method.

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
   ```

   ```
   PM> update-database
   Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
   Applying explicit migrations: [201309121509543_InitialCreate].
   Applying explicit migration: 201309121509543_InitialCreate.
   Running Seed method.
   PM> 
   ```

   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 Windows 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 a Windows Azure Web Site. 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.

Using 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 **Execute Code First Migrations (runs on application start)**. 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 a Windows Azure account

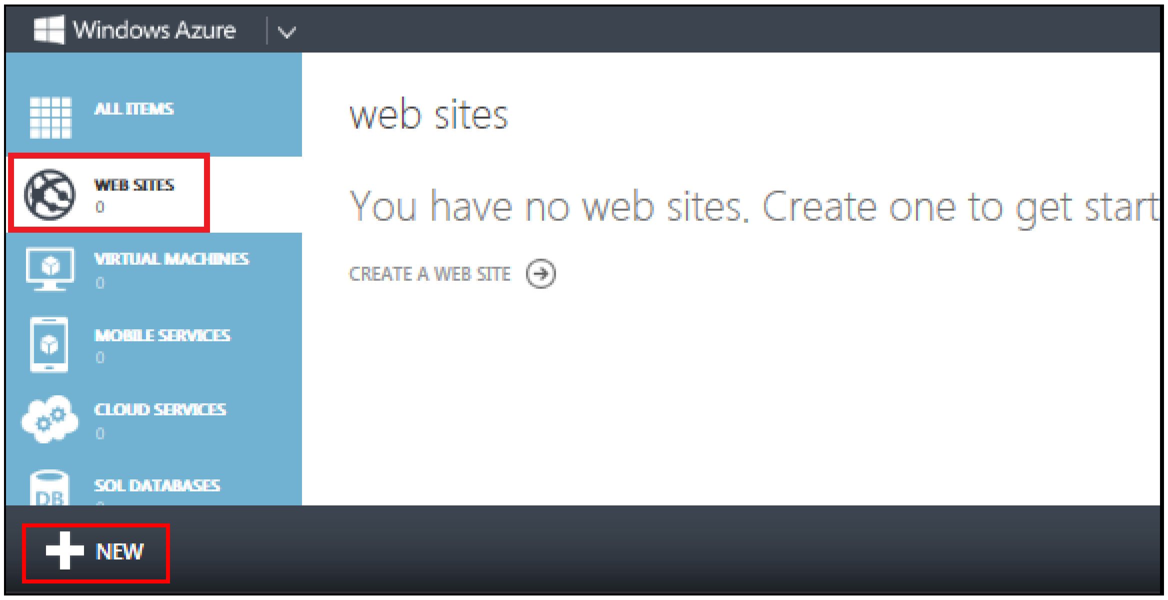
You'll need a Windows Azure account. If you don't already have one, but you do have an MSDN subscription, you can activate your MSDN subscription benefits. Otherwise, you can create a free trial account in just a couple of minutes. For details, see Windows Azure Free Trial.

Create a web site and a SQL database in Windows Azure

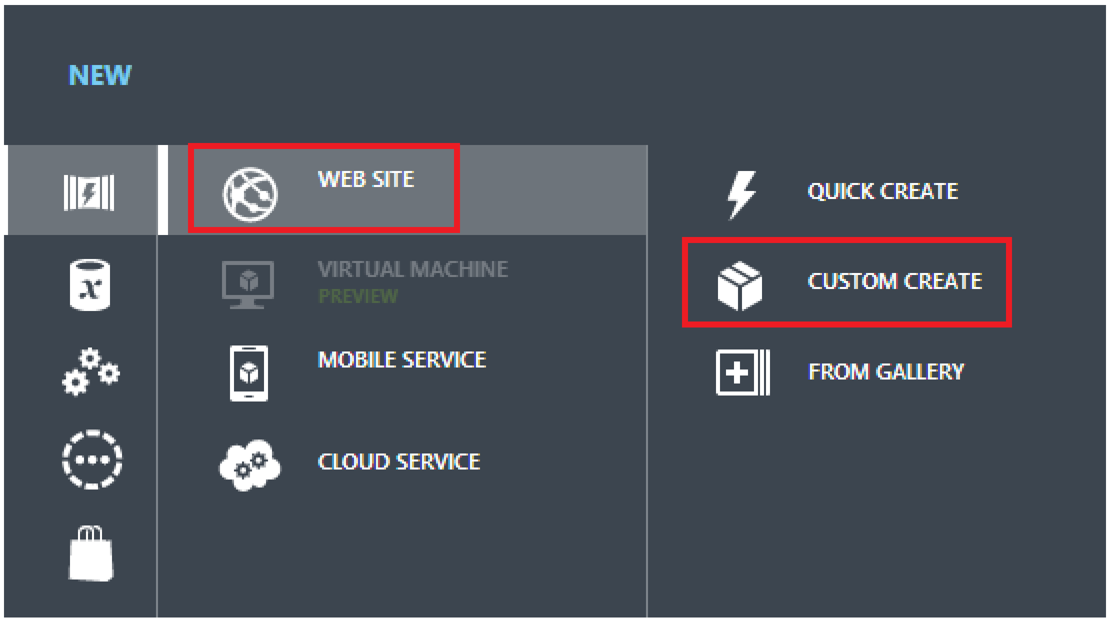
Your Windows Azure Web Site will run in a shared hosting environment, which means it runs on virtual machines (VMs) that are shared with other Windows Azure clients. A shared hosting environment is a low-cost way to get started in the cloud. Later, if your web traffic increases, the application can scale to meet the need by running on dedicated VMs.

You'll deploy the database to Windows 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 Windows Azure Management Portal, click **Web Sites** in the left tab, and then click **New**.

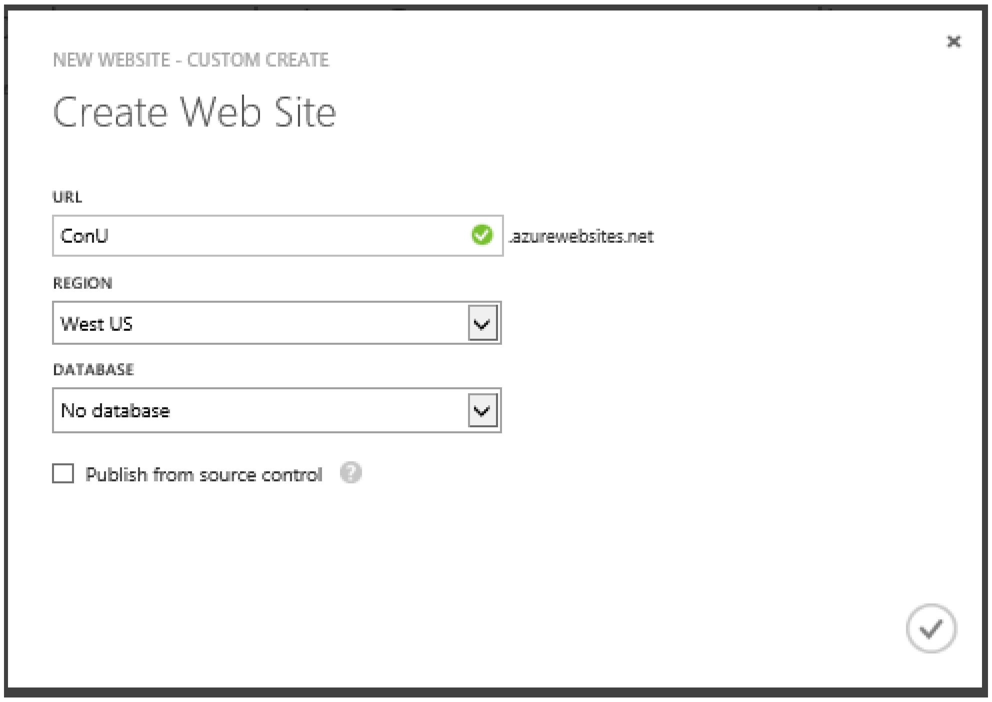


2. Click **CUSTOM CREATE**.



The **New Web Site - Custom Create** wizard opens.

3. In the **New Web Site** step of the wizard, enter a string in the **URL** box to use as the unique URL for your application. The complete URL will consist of what you enter here plus the suffix that you see next to the text box. The illustration shows "ConU", but that URL is probably taken so you will have to choose a different one.


NEW WEBSITE - CUSTOM CREATE

Create Web Site

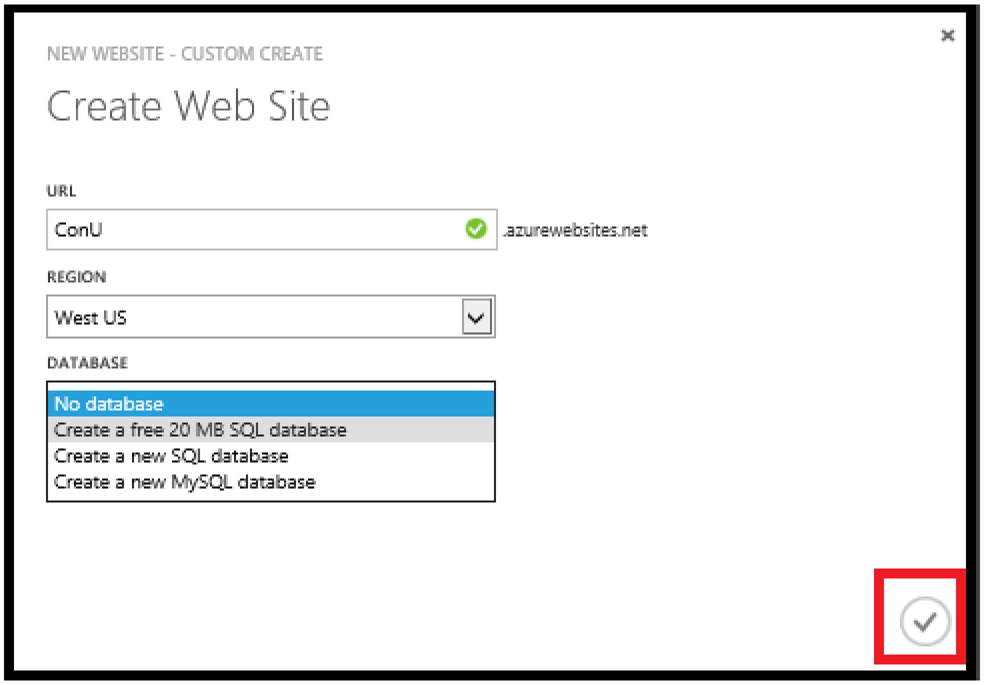
URL: ConU .azurewebsites.net

REGION: West US

DATABASE: No database

☐ Publish from source control

4. In the **Region** drop-down list, choose a region close to you. This setting specifies which data center your web site will run in.
5. In the **Database** drop-down list, choose **Create a free 20 MB SQL database**.


NEW WEBSITE - CUSTOM CREATE

Create Web Site

URL

ConU .azurewebsites.net

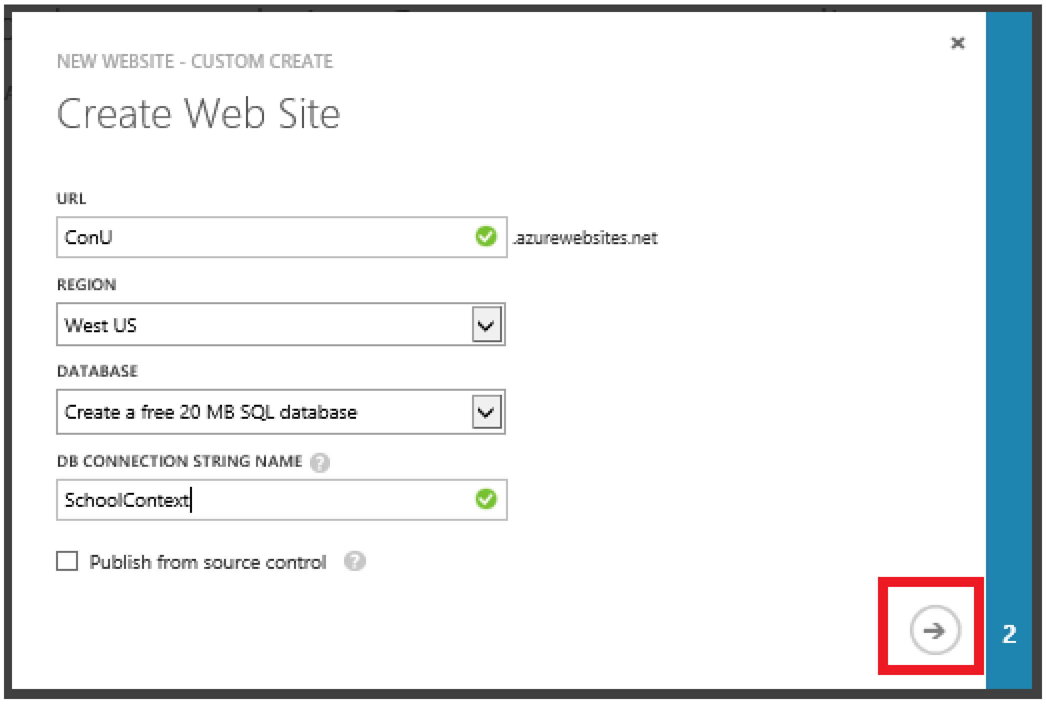
REGION

West US

DATABASE

No database
Create a free 20 MB SQL database
Create a new SQL database
Create a new MySQL database

6. In the **DB CONNECTION STRING NAME**, enter *SchoolContext*.


NEW WEBSITE - CUSTOM CREATE

Create Web Site

URL

ConU .azurewebsites.net

REGION

West US

DATABASE

Create a free 20 MB SQL database

DB CONNECTION STRING NAME

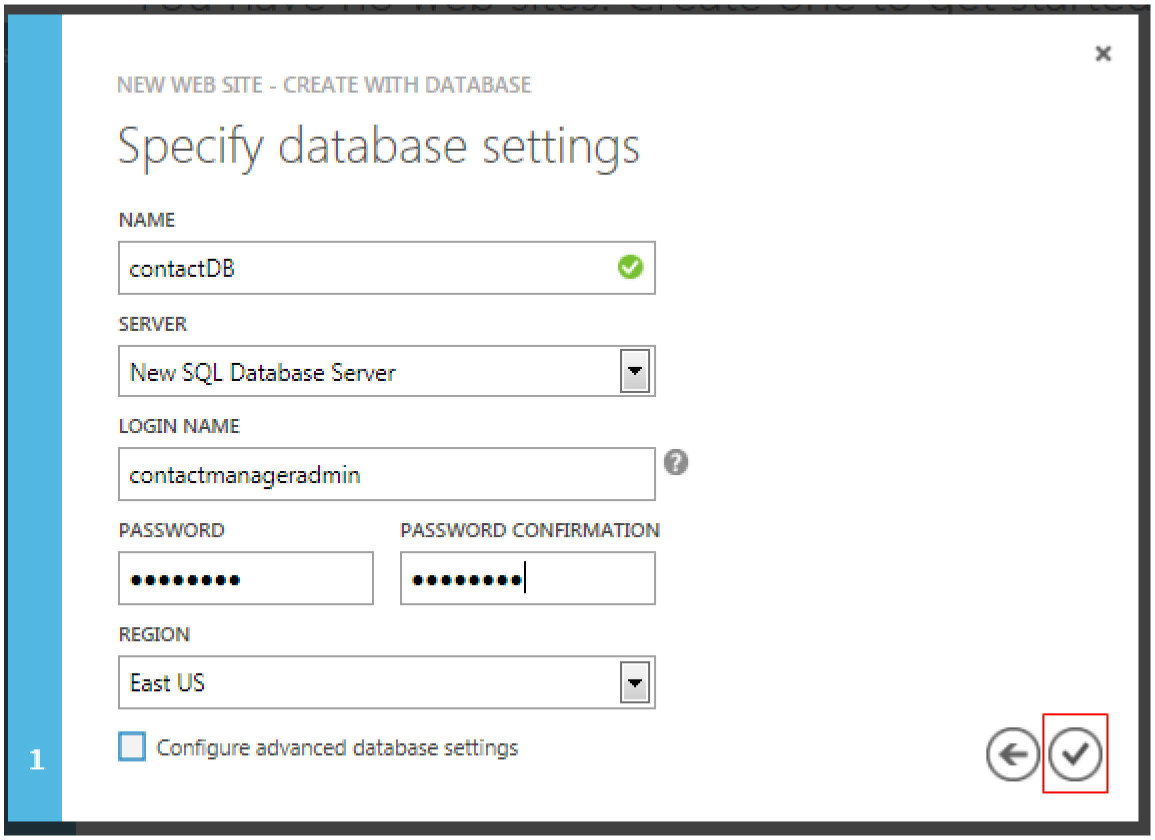
SchoolContext

Publish from source control

7. Click the arrow that points to the right at the bottom of the box. The wizard advances to the **Database Settings** step.
8. In the **Name** box, enter *ContosoUniversityDB*.

9. In the **Server** box, select **New SQL Database server**. Alternatively, if you previously created a server, you can select that server from the drop-down list.
10. Enter an administrator **LOGIN NAME** and **PASSWORD**. If you selected **New SQL Database server** you aren't entering an existing name and password here, you're entering a new name and password that you're defining now to use later when you access the database. If you selected a server that you created previously, you'll enter credentials for that server. For this tutorial, you won't select the ***Advanced*** check box. The ***Advanced*** options enable you to set the database collation.
11. Choose the same **Region** that you chose for the web site.

12. Click the check mark at the bottom right of the box to indicate that you're finished.


NEW WEB SITE - CREATE WITH DATABASE

Specify database settings

NAME: contactDB

SERVER: New SQL Database Server

LOGIN NAME: contactmanageradmin

PASSWORD: ••••••••

PASSWORD CONFIRMATION: ••••••••

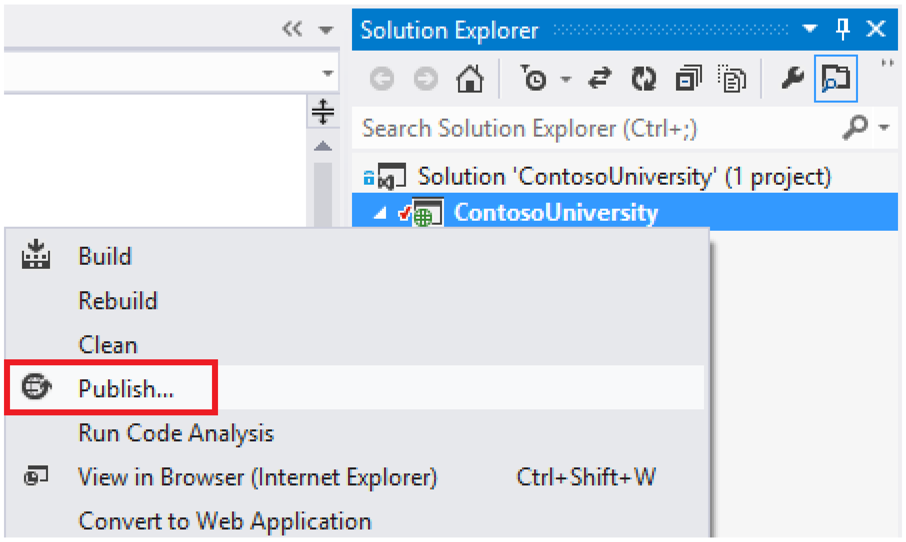
REGION: East US

☐ Configure advanced database settings

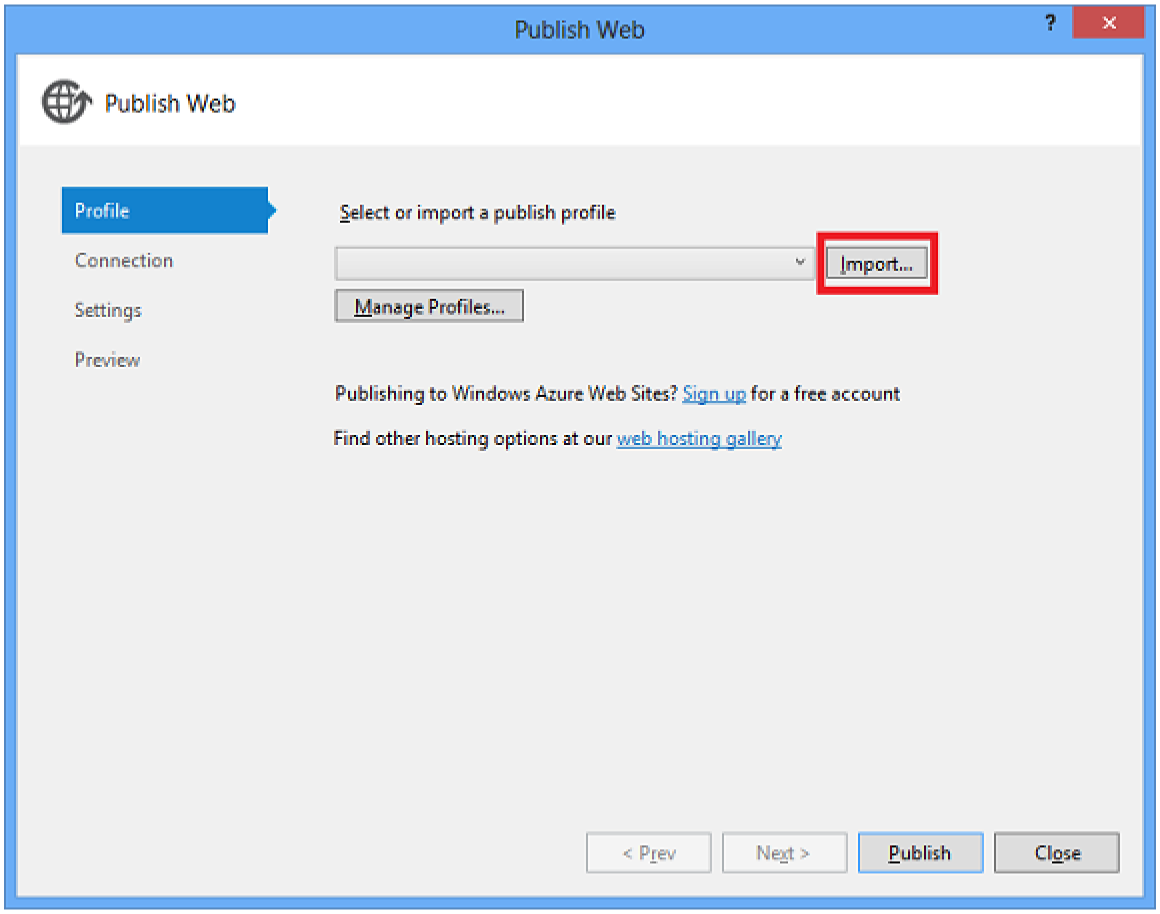
The Management Portal returns to the Web Sites page, and the **Status** column shows that the site is being created. After a while (typically less than a minute), the **Status** column shows that the site was successfully created. In the navigation bar at the left, the number of sites you have in your account appears next to the **Web Sites** icon, and the number of databases appears next to the **SQL Databases** icon.

Deploy the application to Windows Azure

1. In Visual Studio, right-click the project in **Solution Explorer** and select **Publish** from the context menu.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ContosoUniversity' (1 project)
ContosoUniversity
Build
Rebuild
Clean
Publish...
Run Code Analysis
View in Browser (Internet Explorer) Ctrl+Shift+W
Convert to Web Application

2. In the **Profile** tab of the **Publish Web** wizard, click **Import**.

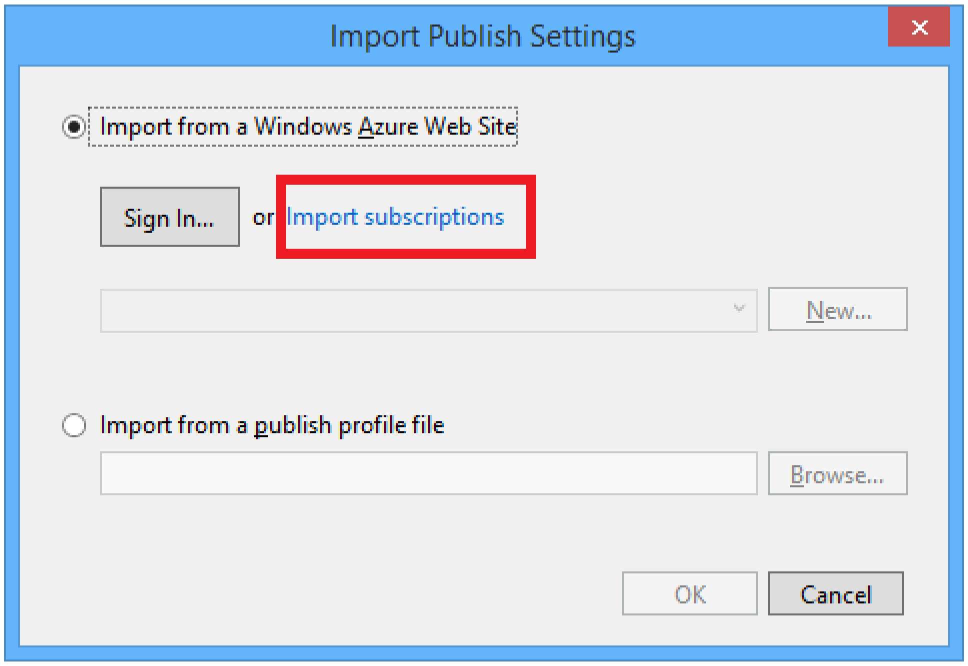

Publish Web
Publish Web
Profile
Connection
Settings
Preview
Select or import a publish profile
Import...
Manage Profiles...
Publishing to Windows Azure Web Sites? Sign up for a free account
Find other hosting options at our web hosting gallery
< Prev
Next >
Publish
Close

3. If you have not previously added your Windows Azure subscription in Visual Studio, perform the following steps. These steps enable Visual Studio to connect to your

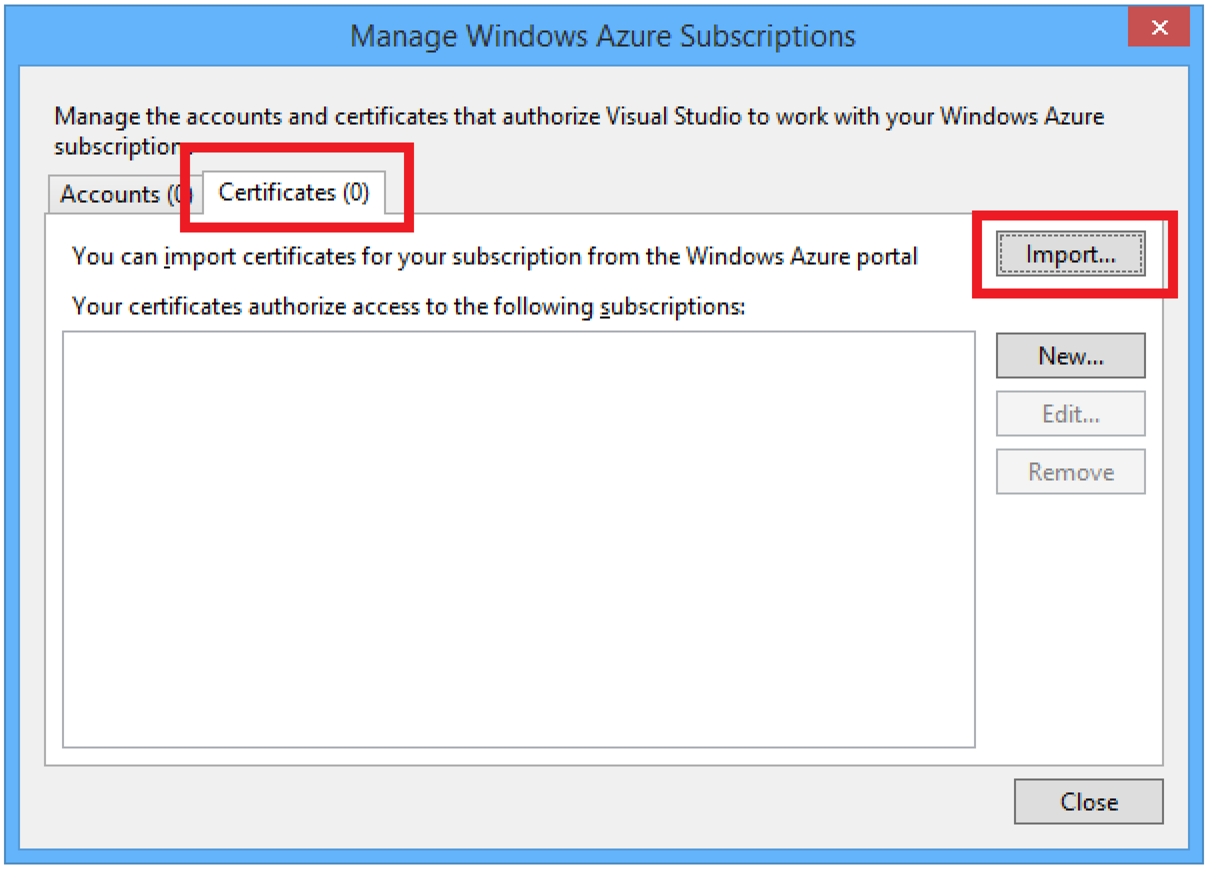
Windows Azure subscription so that the drop-down list under **Import from a Windows Azure web site** will include your web site.

As an alternative, you can sign in directly to your Windows Azure account without downloading a subscription file. To use this method, click **Sign In** instead of **Manage subscriptions** in the next step. This alternative is simpler, but as this tutorial is being written in November, 2013, only the subscription file download enables **Server Explorer** to connect to Windows Azure SQL Database.

a. In the **Import Publish Profile** dialog box, click **Manage subscriptions**.



b. In the **Manage Windows Azure Subscriptions** dialog box, click the **Certificates** tab, and then click **Import**.

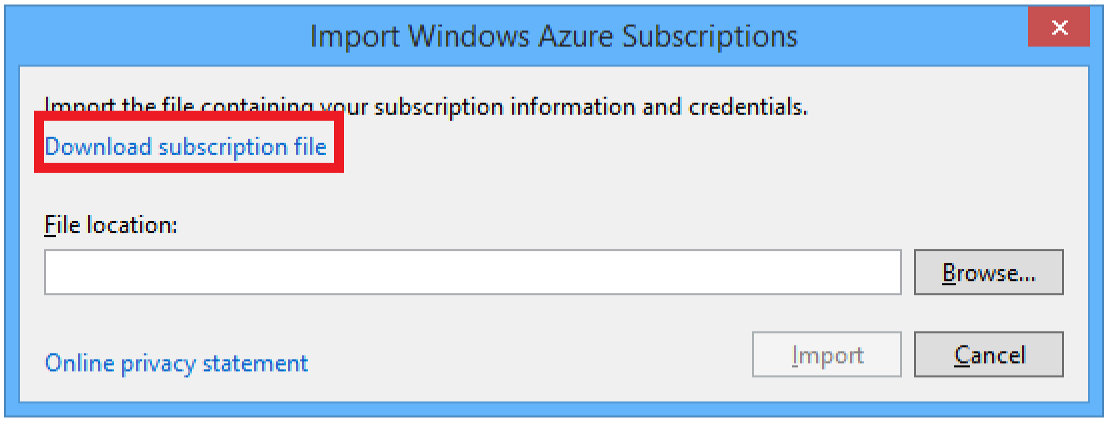

Manage Windows Azure Subscriptions

Manage the accounts and certificates that authorize Visual Studio to work with your Windows Azure subscriptions.

You can import certificates for your subscription from the Windows Azure portal

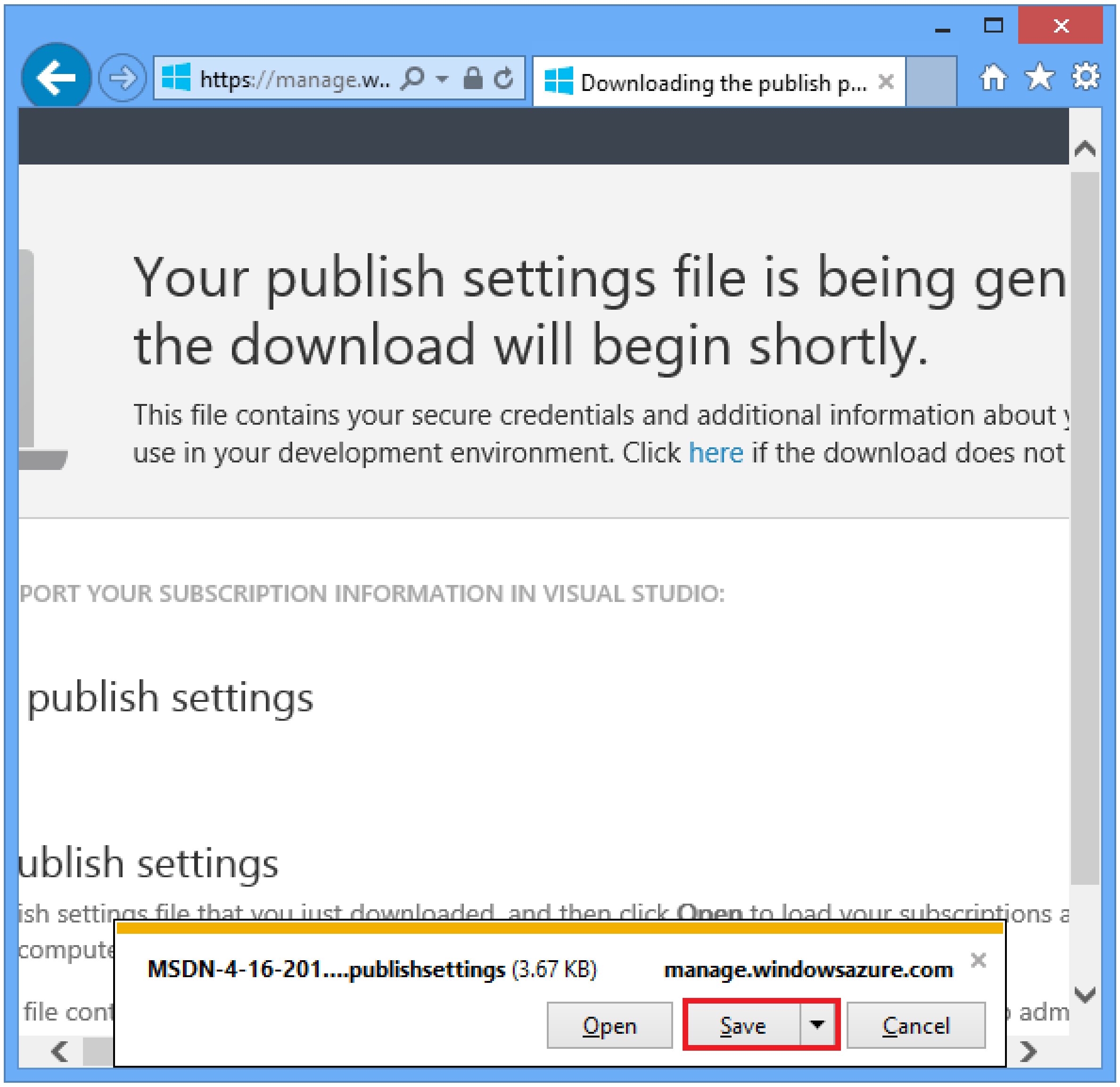
Your certificates authorize access to the following subscriptions:

c. In the **Import Windows Azure Subscriptions** dialog box, click **Download subscription file** .


Import Windows Azure Subscriptions

Import the file containing your subscription information and credentials.

d. In your browser window, save the *.publishsettings* file.



Your publish settings file is being gen
the download will begin shortly.

This file contains your secure credentials and additional information about y
use in your development environment. Click here if the download does not

PORT YOUR SUBSCRIPTION INFORMATION IN VISUAL STUDIO:

publish settings

ublish settings

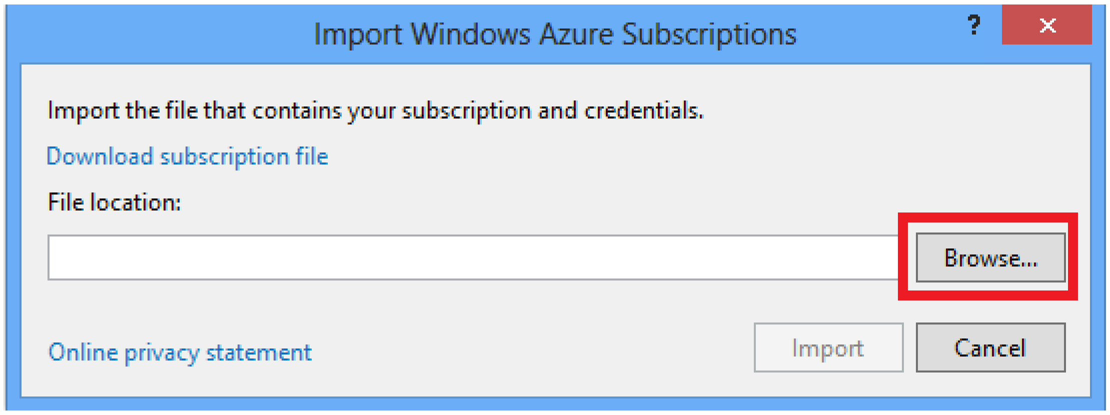
ish settings file that you just downloaded, and then click Open to load your subscriptions a
compute

file cont

MSDN-4-16-201....publishsettings (3.67 KB) manage.windowsazure.com

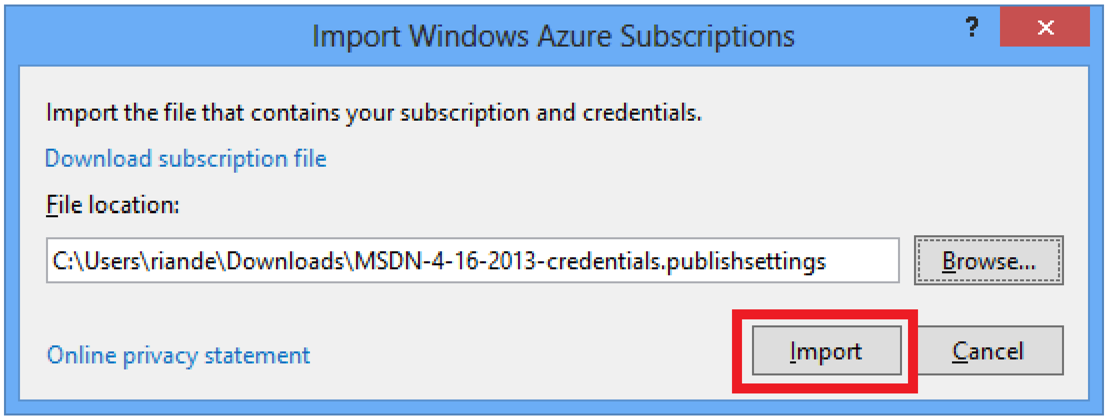
Security Note: The *publishsettings* file contains your credentials (unencoded) that are used to administer your Windows Azure subscriptions and services. The security best practice for this file is to store it temporarily outside your source directories (for example in the *Downloads* folder), and then delete it once the import has completed. A malicious user who gains access to the `.publishsettings` file can edit, create, and delete your Windows Azure services.

e. In the **Import Windows Azure Subscriptions** dialog box, click **Browse** and navigate to the *.publishsettings* file.



Import the file that contains your subscription and credentials.

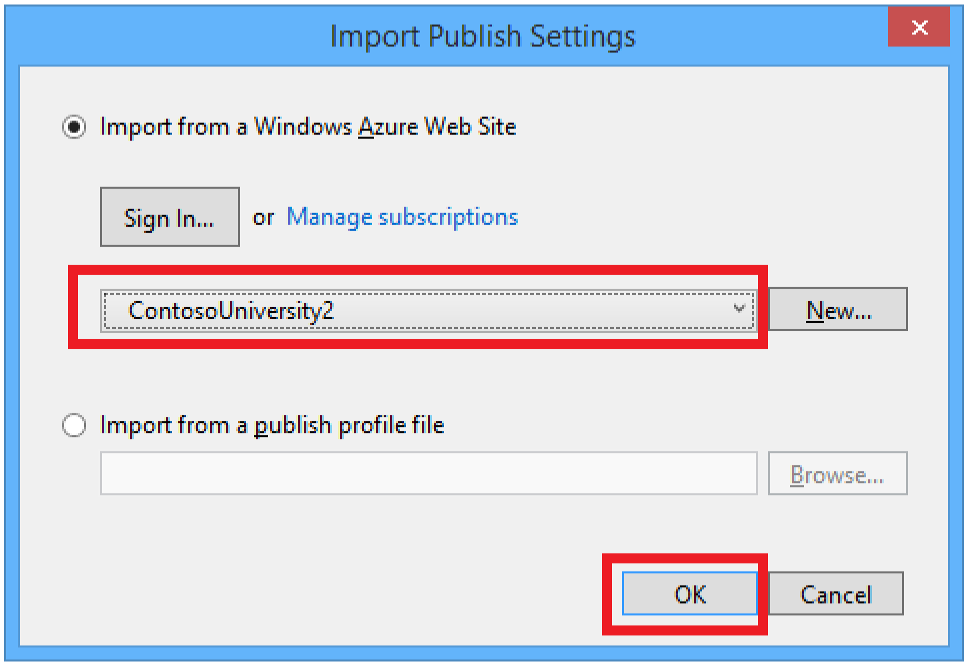
e. Click **Import**.



Import the file that contains your subscription and credentials.

C:\Users\riande\Downloads\MSDN-4-16-2013-credentials.publishsettings

4. Close the **Manage Windows Azure Subscriptions** box.
5. In the **Import Publish Profile** dialog box, select **Import from a Windows Azure web site**, select your web site from the drop-down list, and then click **OK**.

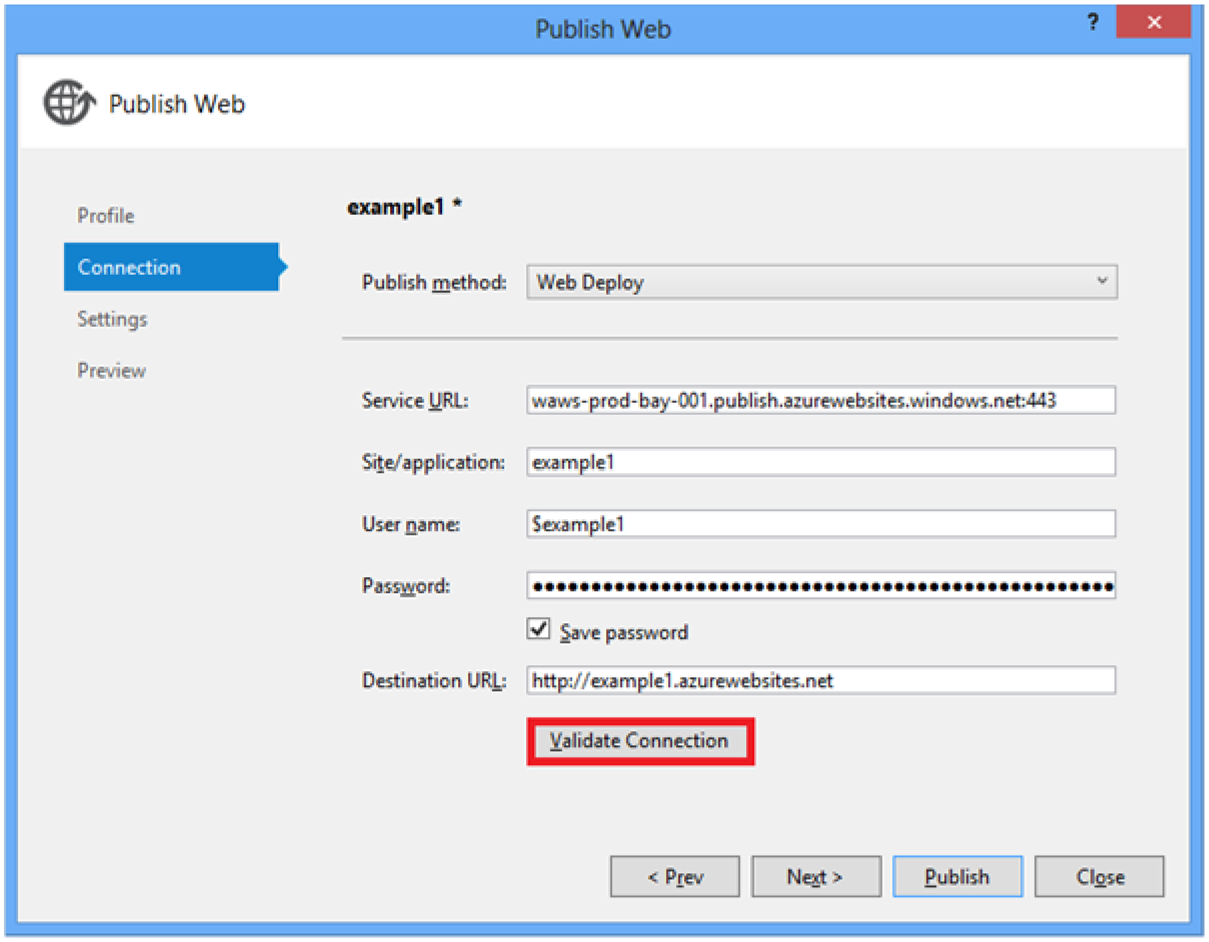

Import Publish Settings

(●) Import from a Windows Azure Web Site

ContosoUniversity2

( ) Import from a publish profile file

6. In the **Connection** tab, click **Validate Connection** to make sure that the settings are correct.


Publish Web

Publish Web

Profile

Connection

Settings

Preview

example1 *

Publish method: Web Deploy

Service URL: waws-prod-bay-001.publish.azurewebsites.windows.net:443

Site/application: example1

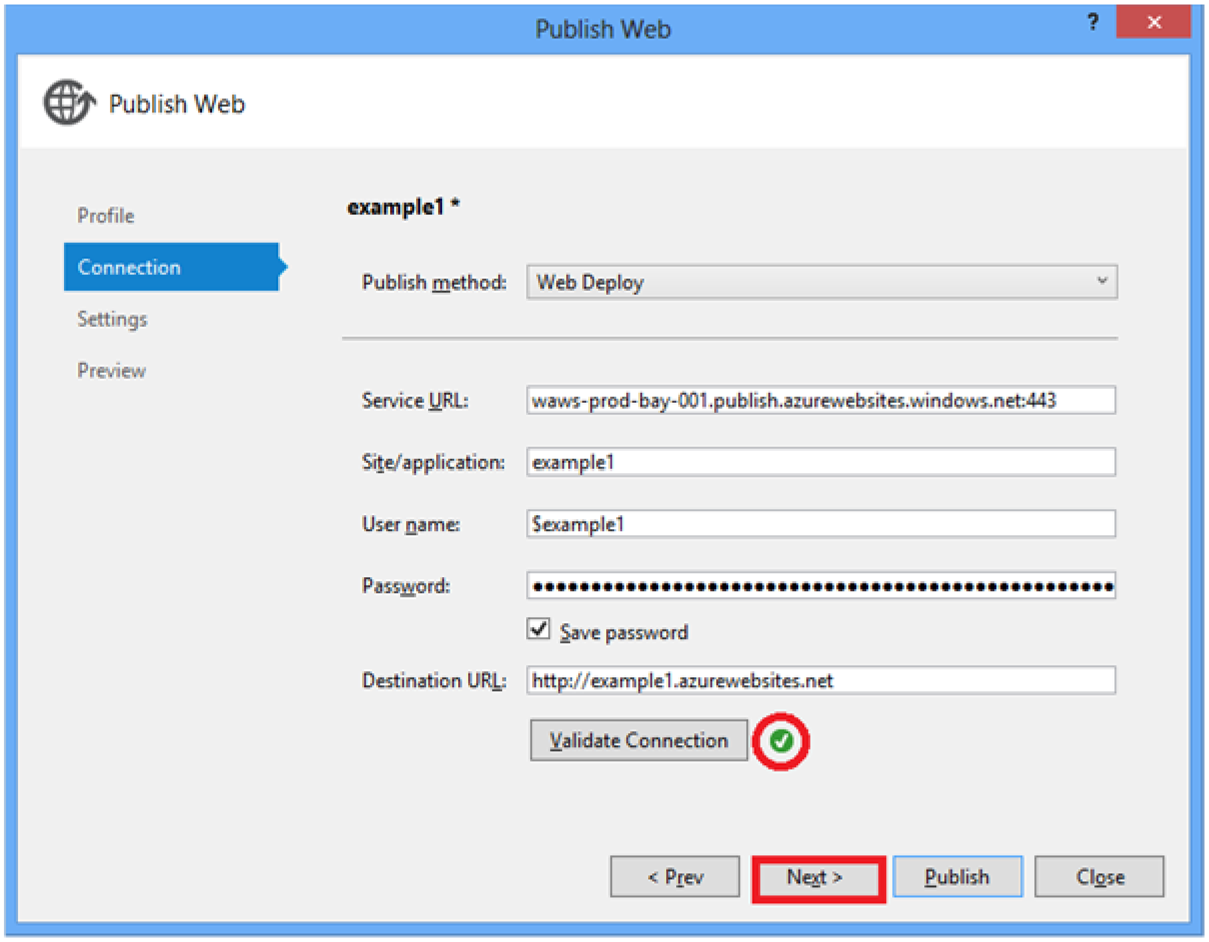
User name: $example1

Password: ••

☑ Save password

Destination URL: http://example1.azurewebsites.net

7. When the connection has been validated, a green check mark is shown next to the **Validate Connection** button. Click **Next**.


Publish Web

Publish Web

example1 *

Publish method: Web Deploy

Service URL: waws-prod-bay-001.publish.azurewebsites.windows.net:443

Site/application: example1

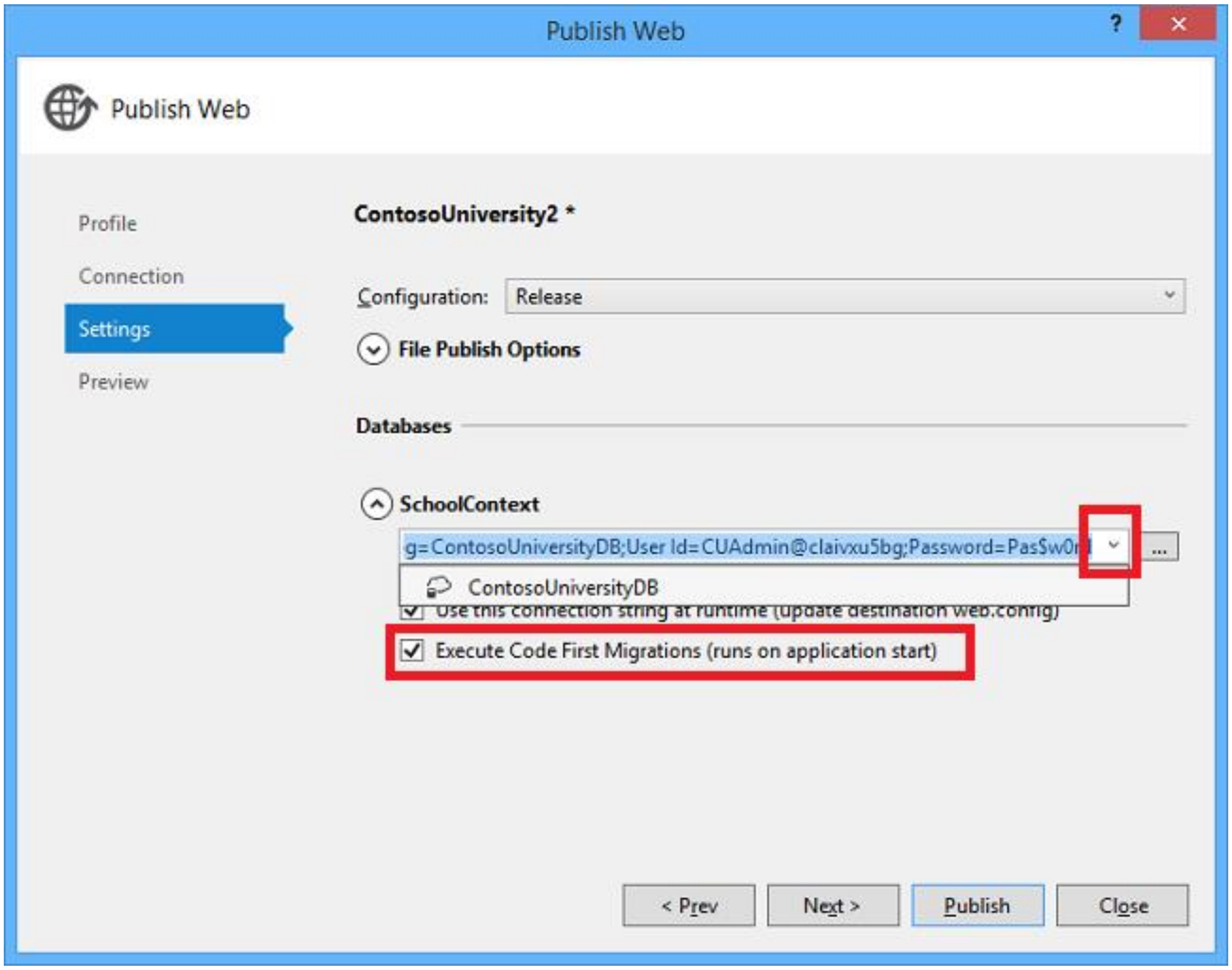
User name: $example1

Password: ••

✓ Save password

Destination URL: http://example1.azurewebsites.net

8. Open the **Remote connection string** drop-down list under **SchoolContext** and select the connection string for the database you created.
9. Select **Execute Code First Migrations (runs on application start)**.



ContosoUniversity2 *

Configuration: Release

File Publish Options

Databases

SchoolContext

g=ContosoUniversityDB;User Id=CUAdmin@claivxu5bg;Password=Pas$w0rd

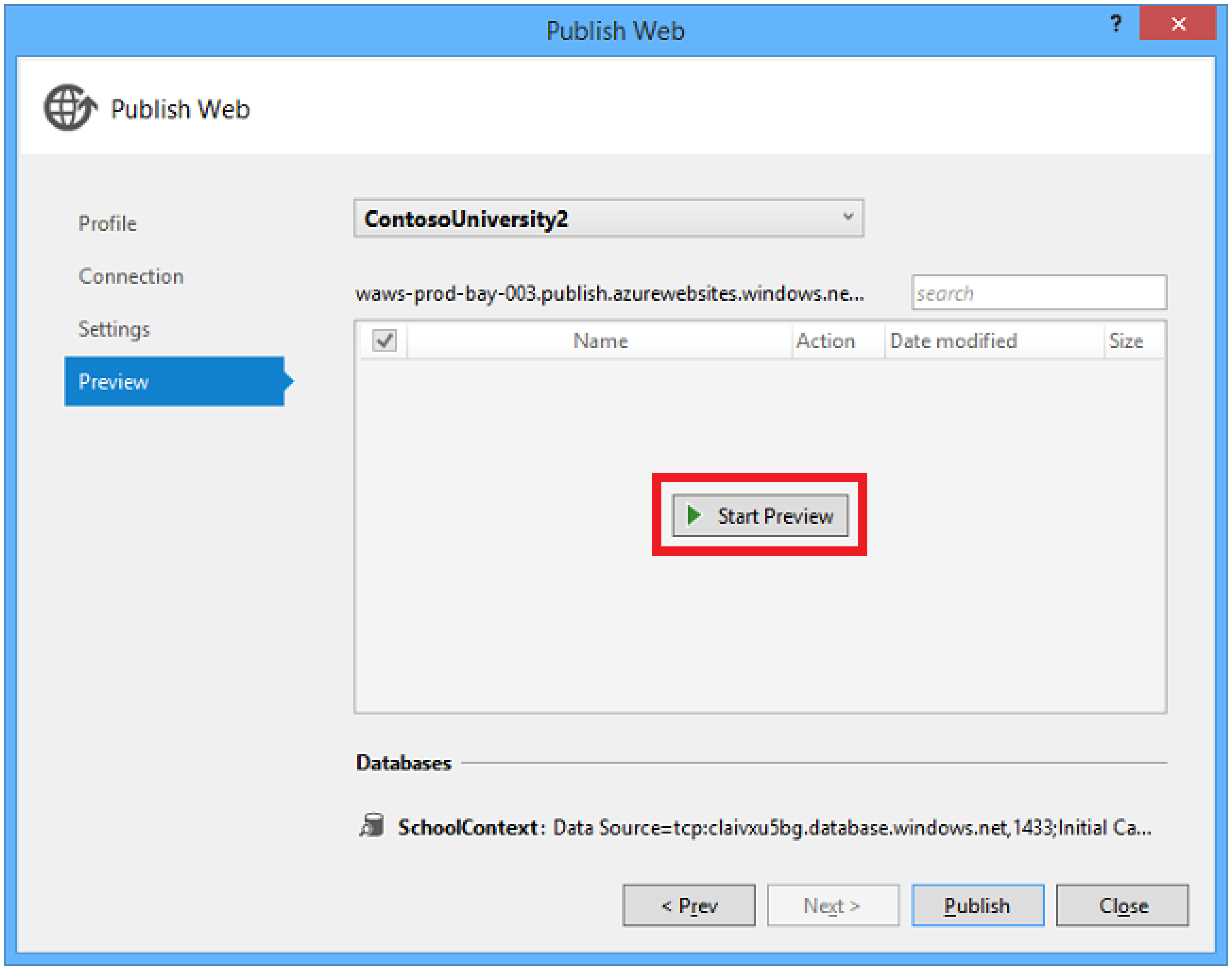
ContosoUniversityDB

Use this connection string at runtime (update destination web.config)

Execute Code First Migrations (runs on application start)

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.

10. Click **Next**.
11. In the **Preview** tab, click **Start Preview**.



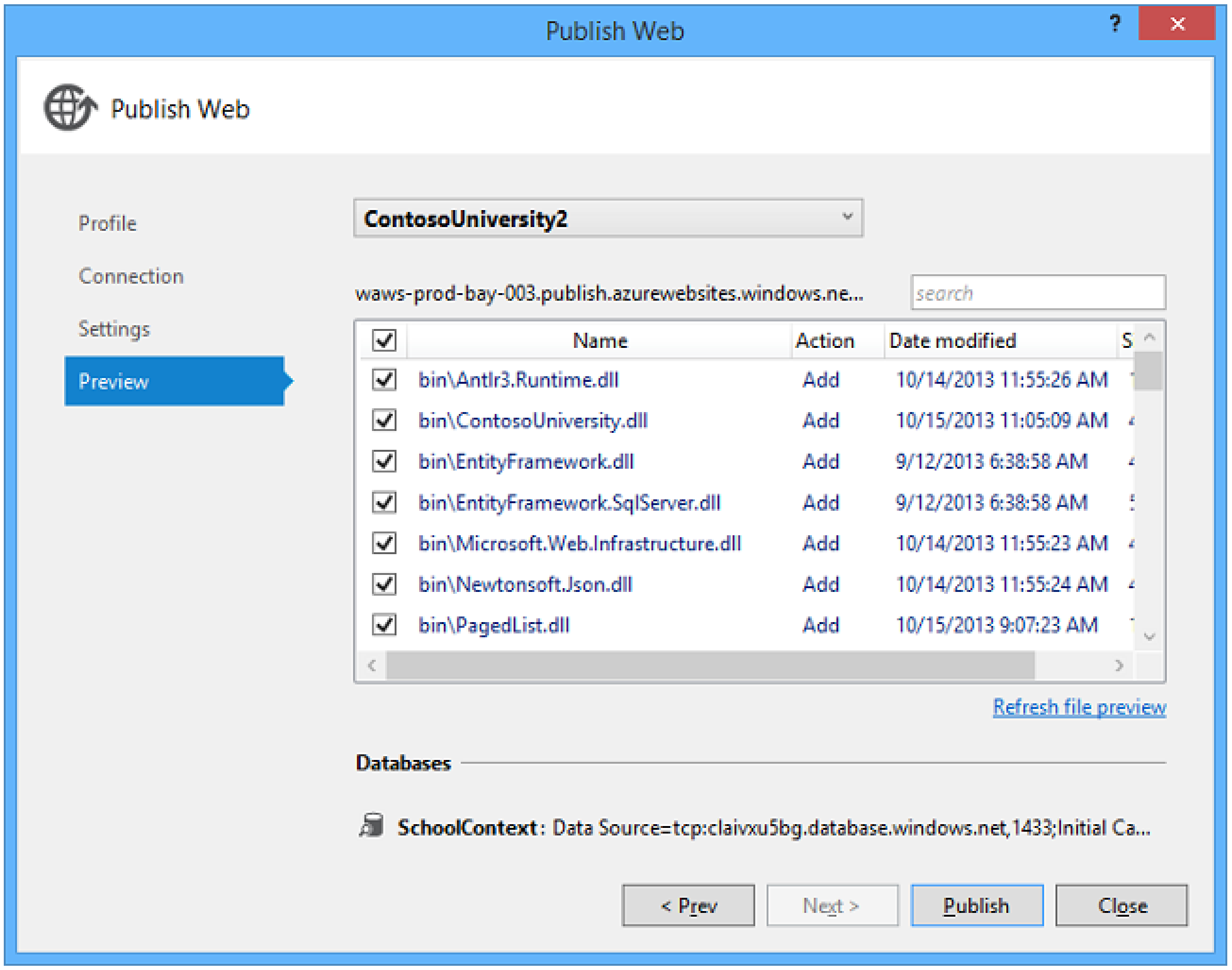
ContosoUniversity2

waws-prod-bay-003.publish.azurewebsites.windows.ne...

Databases

SchoolContext: Data Source=tcp:claivxu5bg.database.windows.net,1433;Initial Ca...

The tab displays a list of the files that will be copied to the server. Displaying the preview isn't required to publish the application but is a useful function to be aware of. In this case, you don't need to do anything with the list of files that is displayed. The next time you deploy this application, only the files that have changed will be in this list.

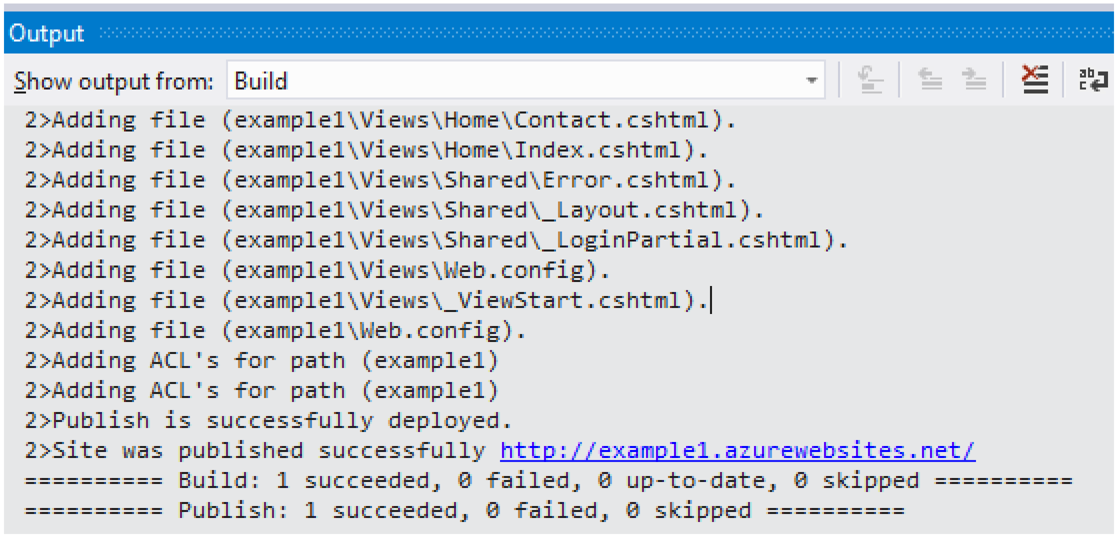


☑	Name	Action	Date modified
☑	bin\Antlr3.Runtime.dll	Add	10/14/2013 11:55:26 AM
☑	bin\ContosoUniversity.dll	Add	10/15/2013 11:05:09 AM
☑	bin\EntityFramework.dll	Add	9/12/2013 6:38:58 AM
☑	bin\EntityFramework.SqlServer.dll	Add	9/12/2013 6:38:58 AM
☑	bin\Microsoft.Web.Infrastructure.dll	Add	10/14/2013 11:55:23 AM
☑	bin\Newtonsoft.Json.dll	Add	10/14/2013 11:55:24 AM
☑	bin\PagedList.dll	Add	10/15/2013 9:07:23 AM

Databases

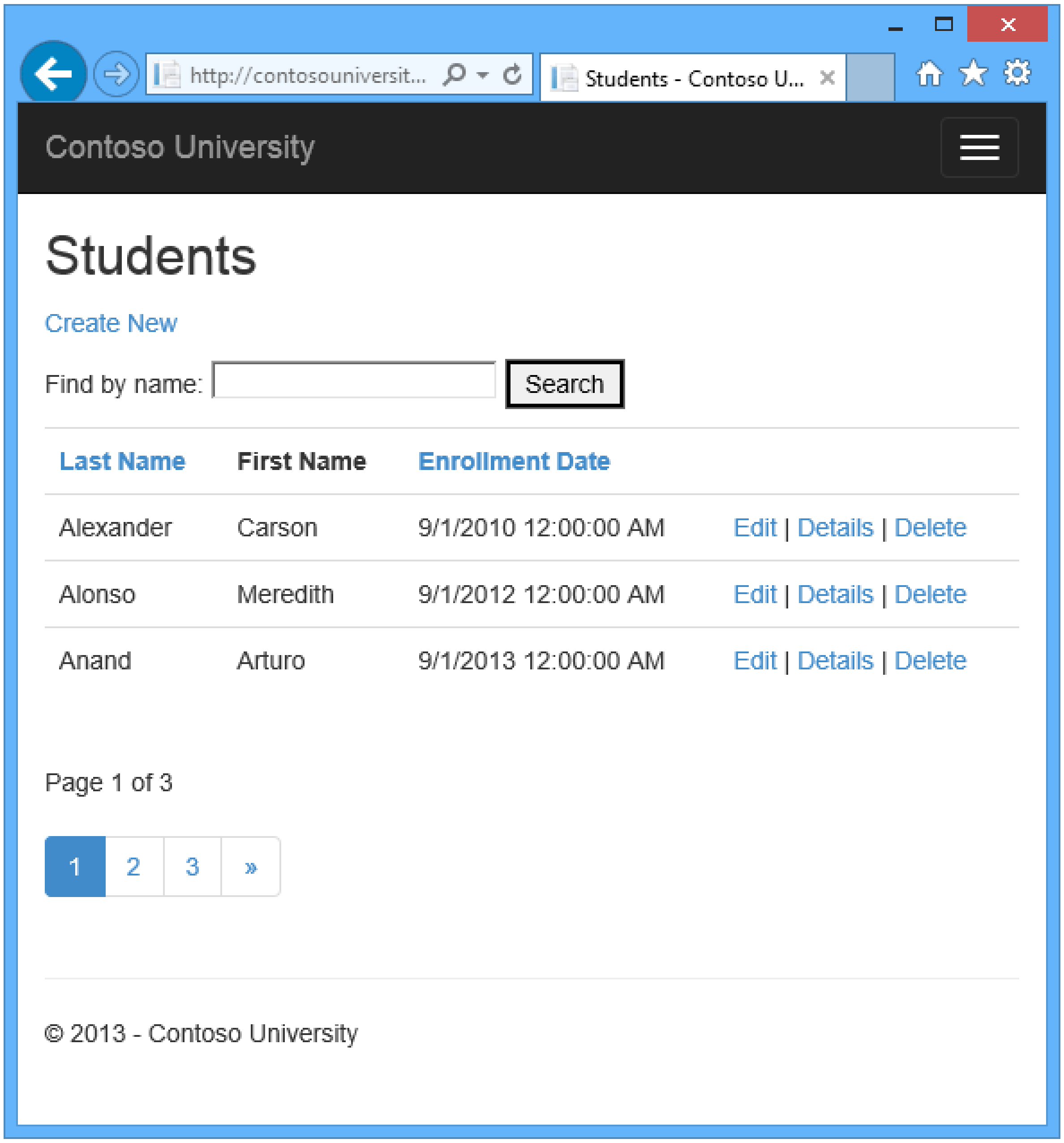
SchoolContext: Data Source=tcp:claivxu5bg.database.windows.net,1433;Initial Ca...

12. Click **Publish**.
    Visual Studio begins the process of copying the files to the Windows Azure server.
13. The **Output** window shows what deployment actions were taken and reports successful completion of the deployment.



```
2>Adding file (example1\Views\Home\Contact.cshtml).
2>Adding file (example1\Views\Home\Index.cshtml).
2>Adding file (example1\Views\Shared\Error.cshtml).
2>Adding file (example1\Views\Shared\_Layout.cshtml).
2>Adding file (example1\Views\Shared\_LoginPartial.cshtml).
2>Adding file (example1\Views\Web.config).
2>Adding file (example1\Views\_ViewStart.cshtml).
2>Adding file (example1\Web.config).
2>Adding ACL's for path (example1)
2>Adding ACL's for path (example1)
2>Publish is successfully deployed.
2>Site was published successfully http://example1.azurewebsites.net/
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
========== Publish: 1 succeeded, 0 failed, 0 skipped ==========
```

14. Upon successful deployment, the default browser automatically opens to the URL of the deployed web site.
    The application you created is now running in the cloud. Click the Students tab.



Students

Create New

Find by name: Search

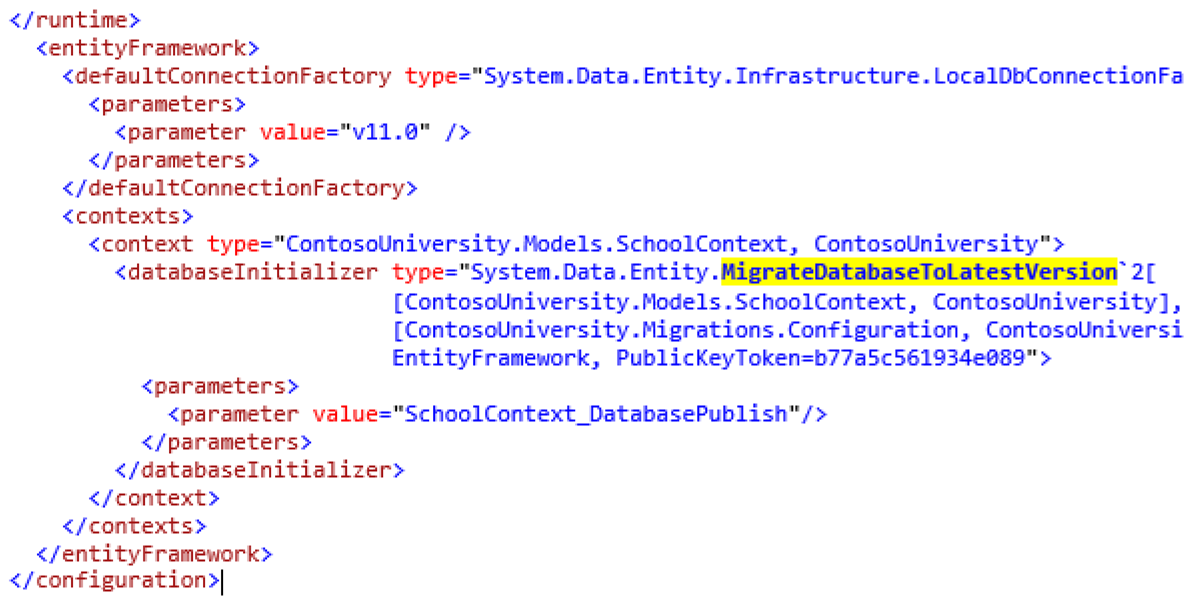
Last Name	First Name	Enrollment Date	
Alexander	Carson	9/1/2010 12:00:00 AM	Edit | Details | Delete
Alonso	Meredith	9/1/2012 12:00:00 AM	Edit | Details | Delete
Anand	Arturo	9/1/2013 12:00:00 AM	Edit | Details | Delete

Page 1 of 3

1 2 3 »

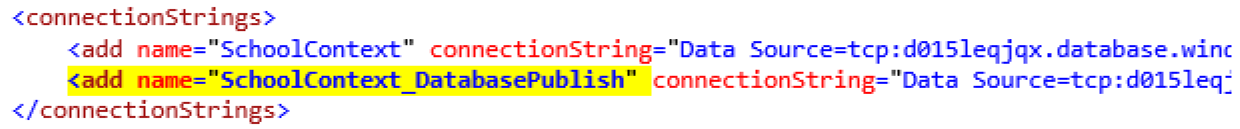
© 2013 - Contoso University

At this point your *SchoolContext* database has been created in the Windows 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.LocalDbConnectionFa
      <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>
```

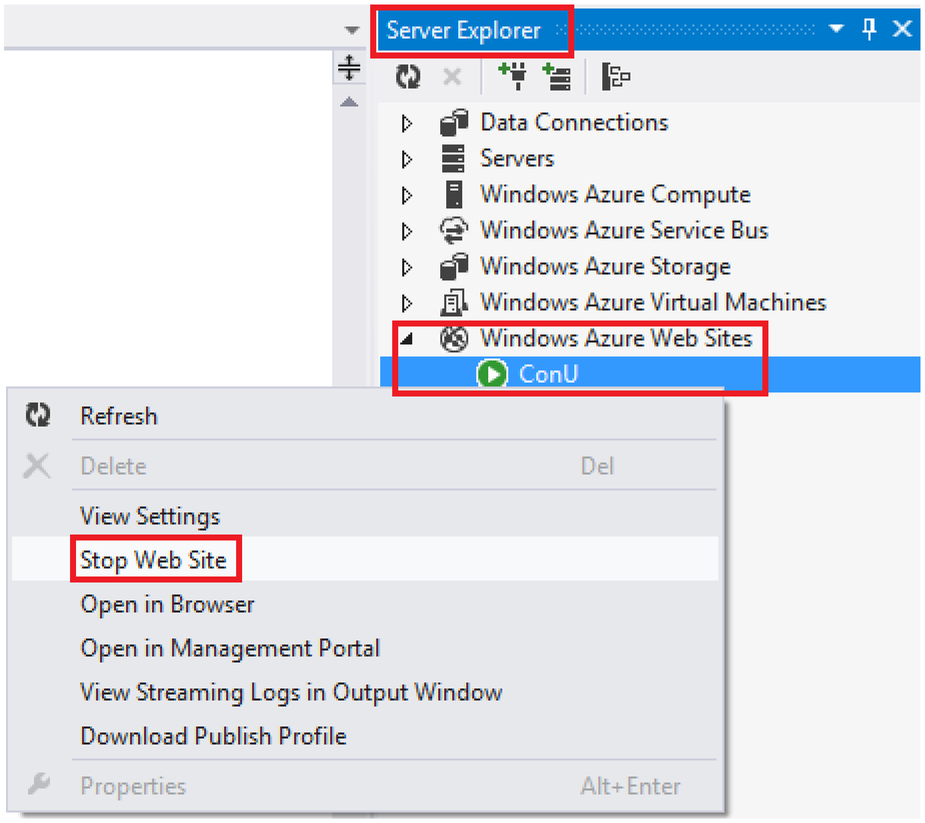
The deployment process also created a new connection string *(SchoolContext_DatabasePublish*) for Code First Migrations to use for updating the database schema and seeding the database.



```
<connectionStrings>
    <add name="SchoolContext" connectionString="Data Source=tcp:d015leqjqx.database.winc
    <add name="SchoolContext_DatabasePublish" connectionString="Data Source=tcp:d015leq
</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 a Windows Azure Web Site. You can prevent other people from using the site by using the Windows Azure Management Portal or **Server Explorer** in Visual Studio to stop the site.



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 mutiple 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) on MSDN.

For information about other migrations scenarios, see Migrations Screencast Series.

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. At the time this tutorial is being written in November, 2013, you can only use the Create and

DropCreate initializers before you enable migrations. The Entity Framework team is working on making these initializers usable with migrations as well.

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.

Summary

In this tutorial you've seen how to enable migrations and deploy the application. In the next tutorial you'll begin looking at more advanced topics by expanding the data model.