Access from ASP.NET app using AAD delegated authentication

Last updated by | Vitor Tomaz | Feb 18, 2021 at 2:30 AM PST

Access from ASP.NET app using AAD delegated authentication

Contents

Isuse

This document describes how to implement Azure AD delegated authentication to access Azure SQL Database from an ASP.NET application that uses Entity Framework Model First.

This walkthrough assumes authentication is performed on an Azure AD domain.

Requirements

Token authentication to Azure SQL Database requires .NET Framework 4.6 or later. This walkthrough requires Visual Studio 2017 configured for the "ASP.NET and web development" workload. The steps were tested using Visual Studio Community 2017.

Limitations

This walkthrough does not try to optimize how tokens are acquired and cached.

For more information about the best practices related to caching the tokens in web applications, please refer to "Cache access tokens in a multitenant application | Microsoft Docs" (https://docs.microsoft.com/en-us/azure/architecture/multitenant-identity/token-cache).

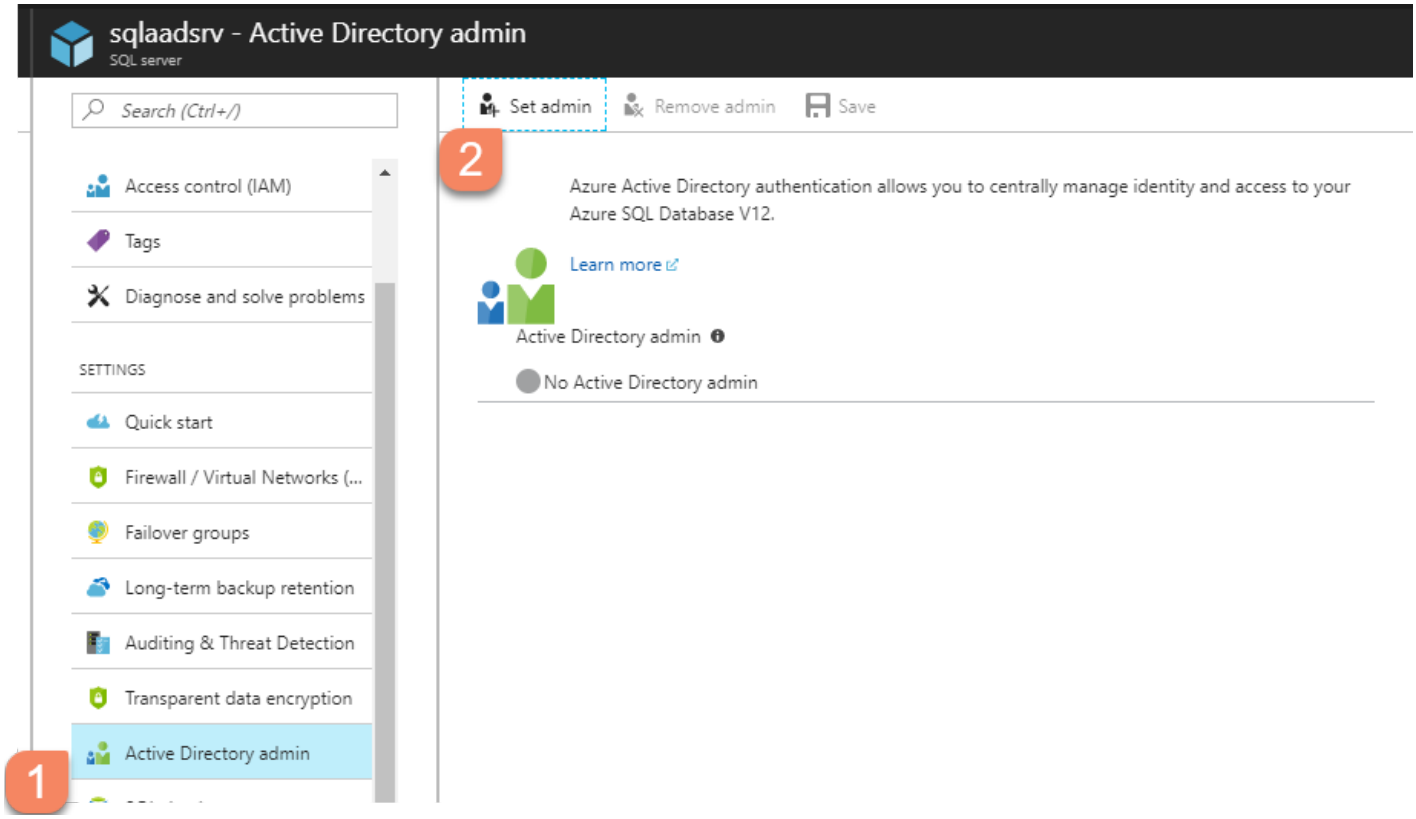
Provision an Azure SQL Database with AAD authentication

Create an Azure SQL Database and configure the firewall rules to allow connectivity from the workstation running Visual Studio.

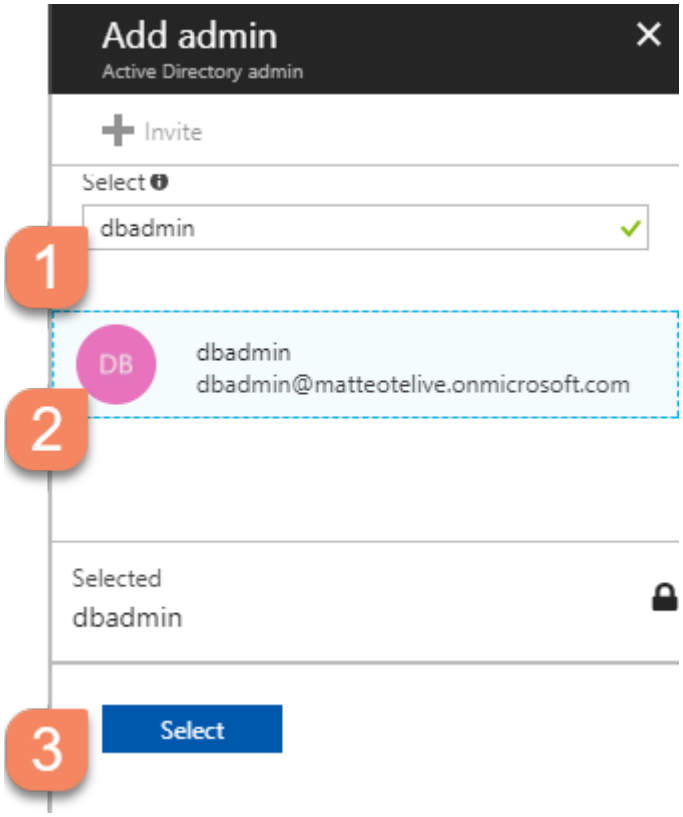
For information about creating an Azure SQL Database, please refer to "Azure portal: Create a SQL database | Microsoft Docs" (https://docs.microsoft.com/en-us/azure/sql-database/sql-database-get-started-portal).

For information about configuring the firewall rules, please refer to "Azure SQL Database firewall rules | Microsoft Docs" (https://docs.microsoft.com/en-us/azure/sql-database/sql-database-firewall-configure). Once the server and database have been provisioned, proceed to configure the Azure AD administrator.

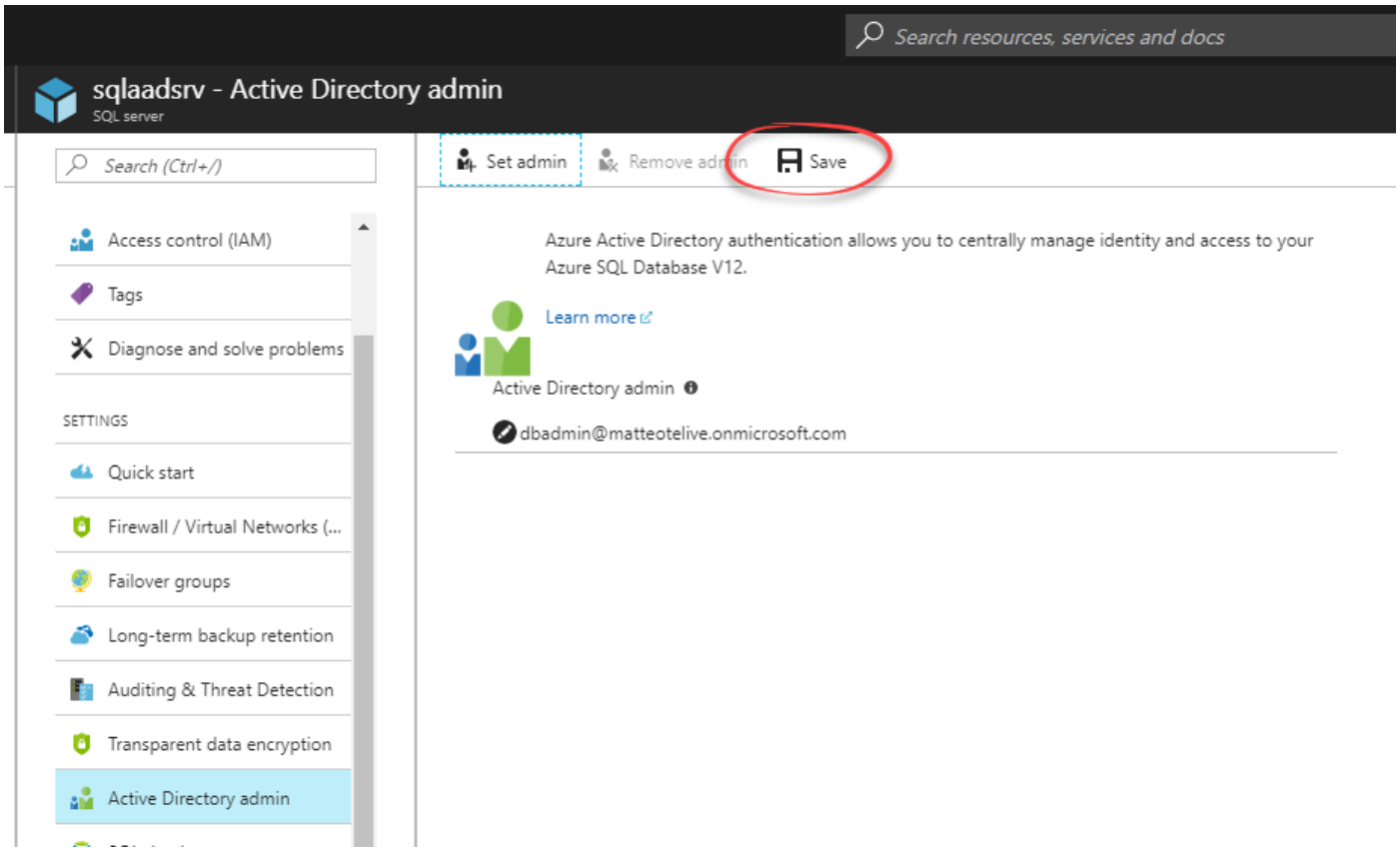
Select the Azure SQL Database server from the Azure Portal, open the Active Directory admin blade and click Set admin.



Type the name of the Azure AD user in the search box, select the user from the search results and click Select.

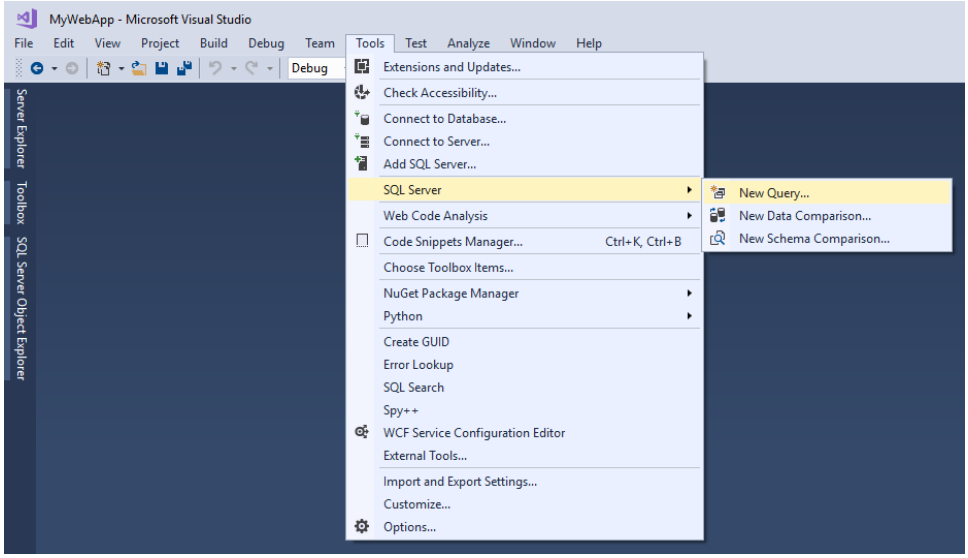


Click Save to apply the changes.

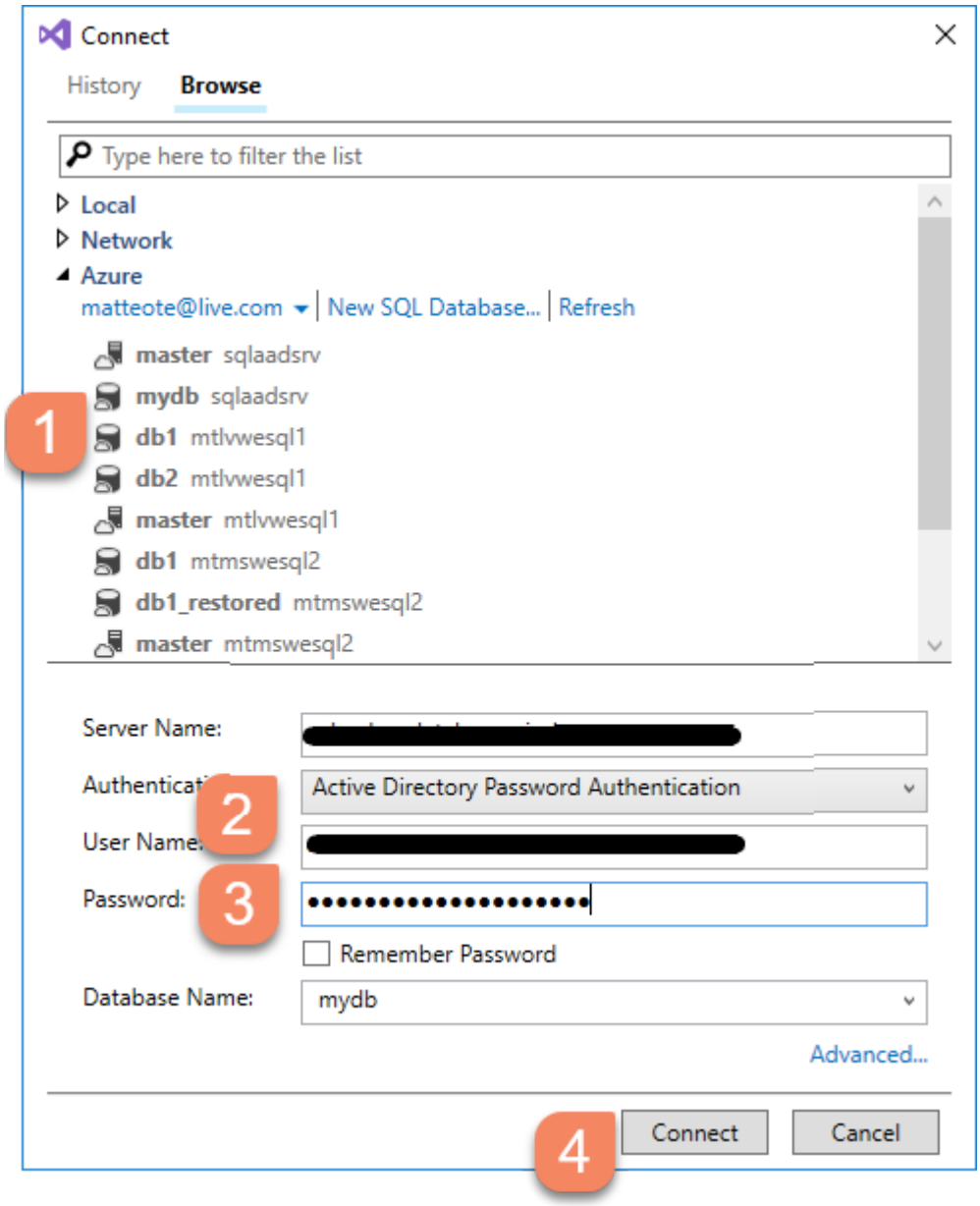


After the Azure AD administrator has been configured, you can enable database access for other Azure AD users.

Open a new SQL Server query in Visual Studio.



In the Connect dialog box, select the database, select Active Directory Password Authentication, enter the credentials of the Azure AD admin and click Connect



Note: your Azure AD domain may not support Azure AD password authentication. Based on your organization's requirements you may need to use Active Directory Integrated Authentication or, if your organization uses Multi-Factor authentication, you need to perform these steps using SQL Server Management Studio connecting to the database with "Active Directory - Universal with MFA support" authentication.

For more information about Universal authentication and obtaining SQL Server Management Studio, please refer to "Multi-Factor authentication - Azure SQL | Microsoft Docs" (https://docs.microsoft.com/en-us/azure/sql-database/sql-database-ssms-mfa-authentication). The following steps assume you are running the query from Visual Studio.

In the query window, enter the CREATE USER command and click the Execute button.

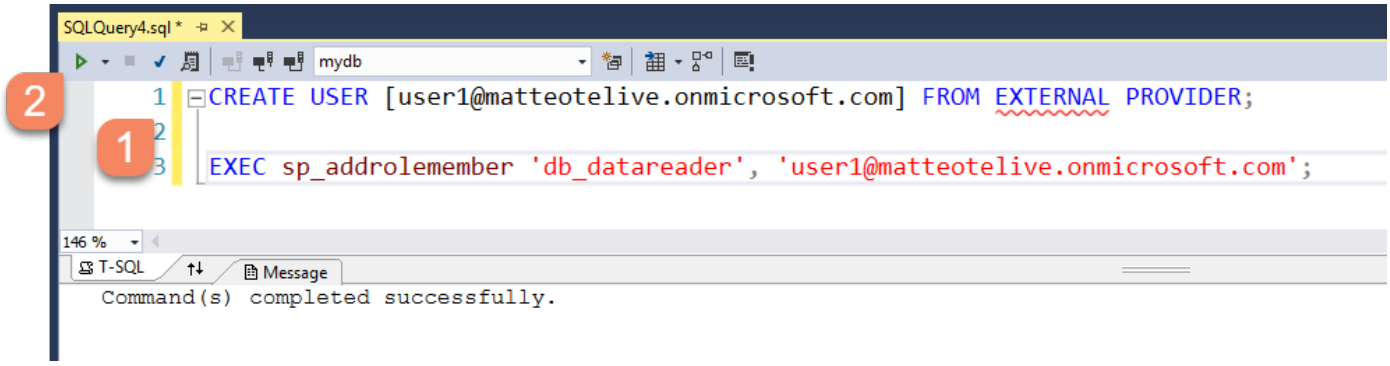
The CREATE USER command is CREATE USER [username@domainname] FROM EXTERNAL PROVIDER, where username@domainname must be replaced with the details of the user you want to grant access to.

Repeat the command for every user that needs to access the database.

The example below includes a command to make the new user a member of the db_datareader role, to provide permissions for reading data from the database.

Make sure the user you create is granted with the right privileges, depending on your requirements.

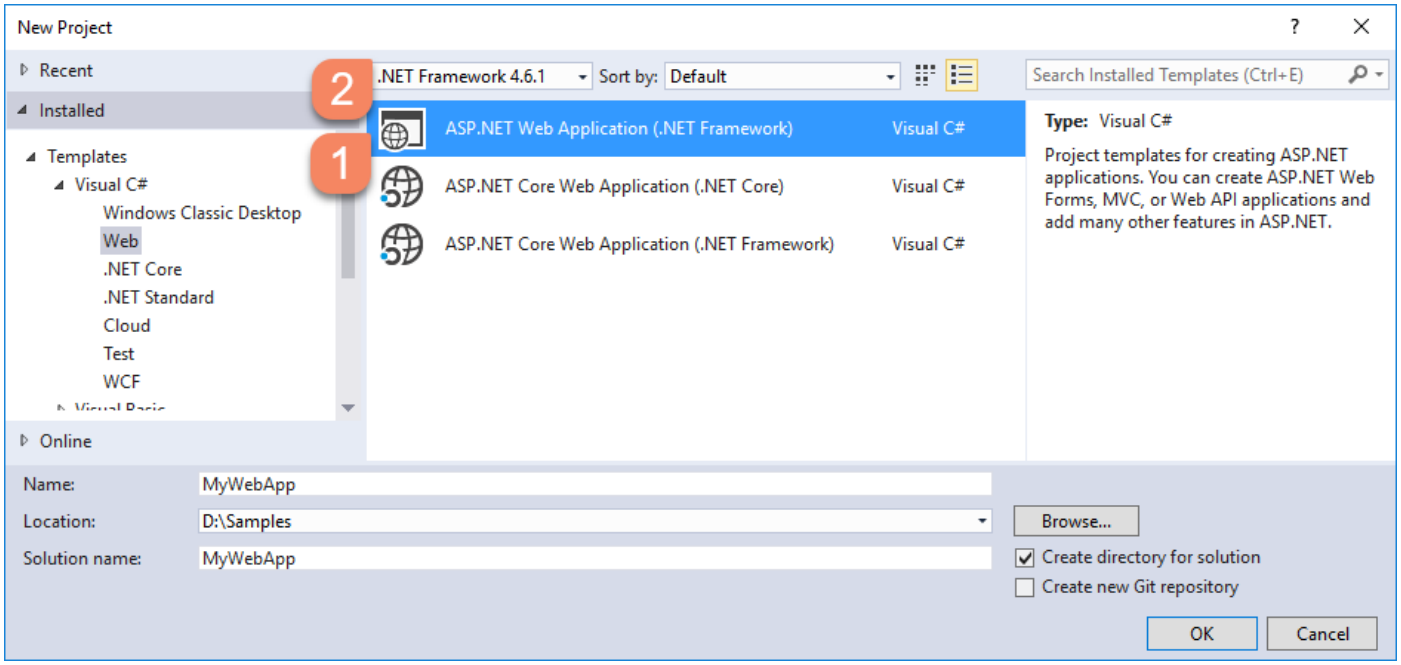
For more information about the authorization concepts in Azure SQL Database, please refer to the "Authorization" section of "Granting access to Azure SQL Database | Microsoft Docs" (https://docs.microsoft.com/en-us/azure/sql-database/sql-database-control-access).



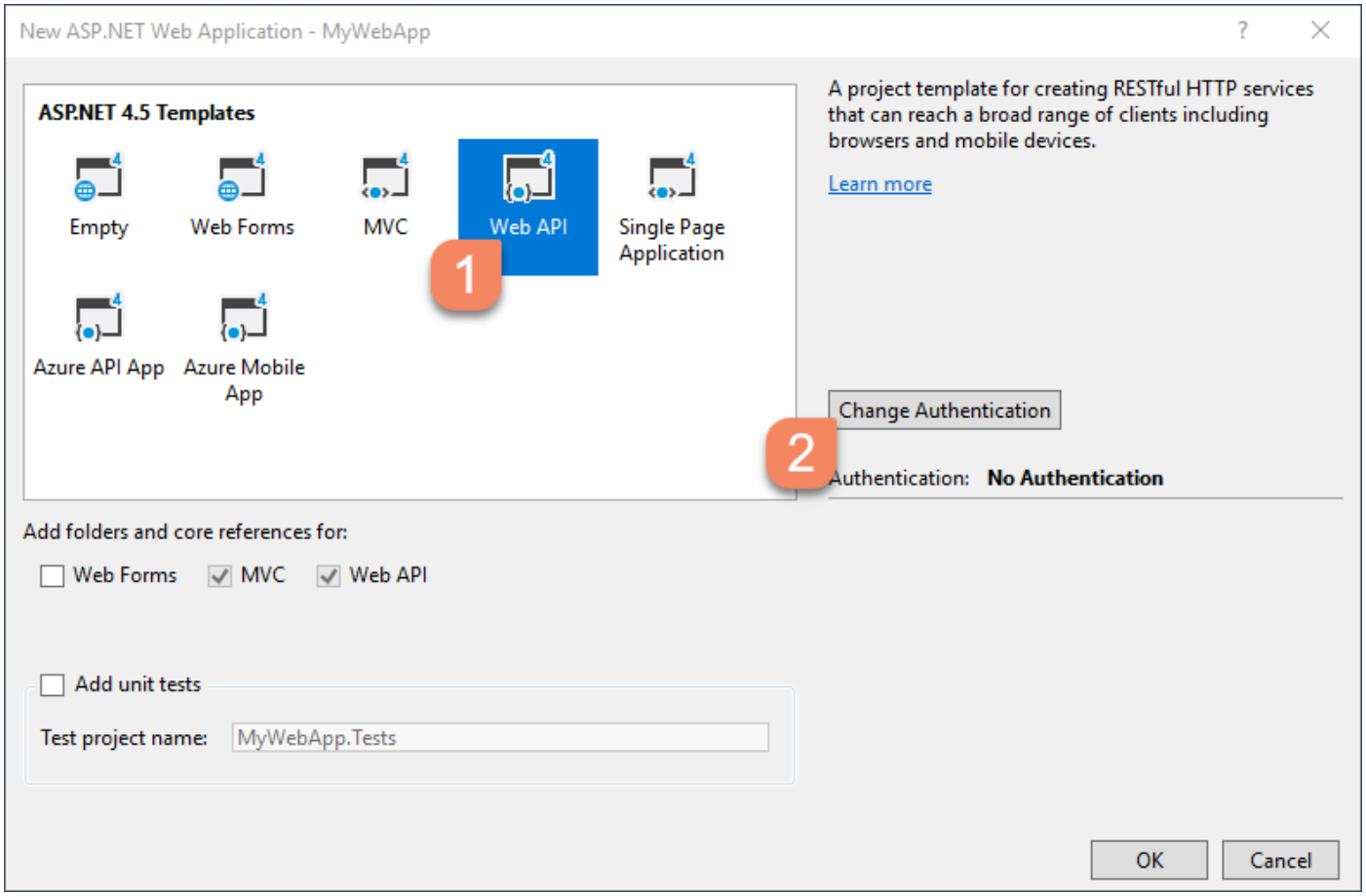
Create the Visual Studio project

Create a new Visual Studio project (menu File->New->Project...).

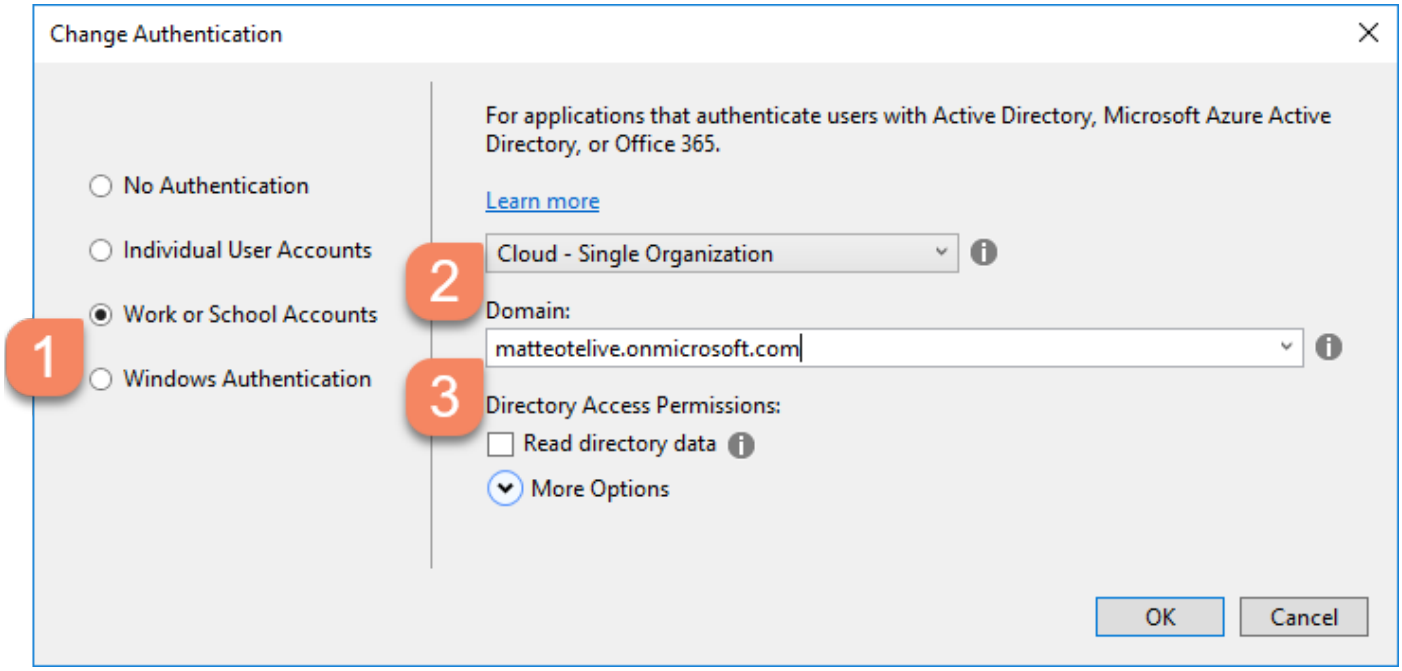
Select "ASP.NET Web Application (.NET Framework)" template, then select .NET Framework 4.6.1.



Select the "Web API" template and click "Change Authentication".



Select "Work or School Accounts", select "Cloud - Single Organization", enter your Azure AD domain and click OK.

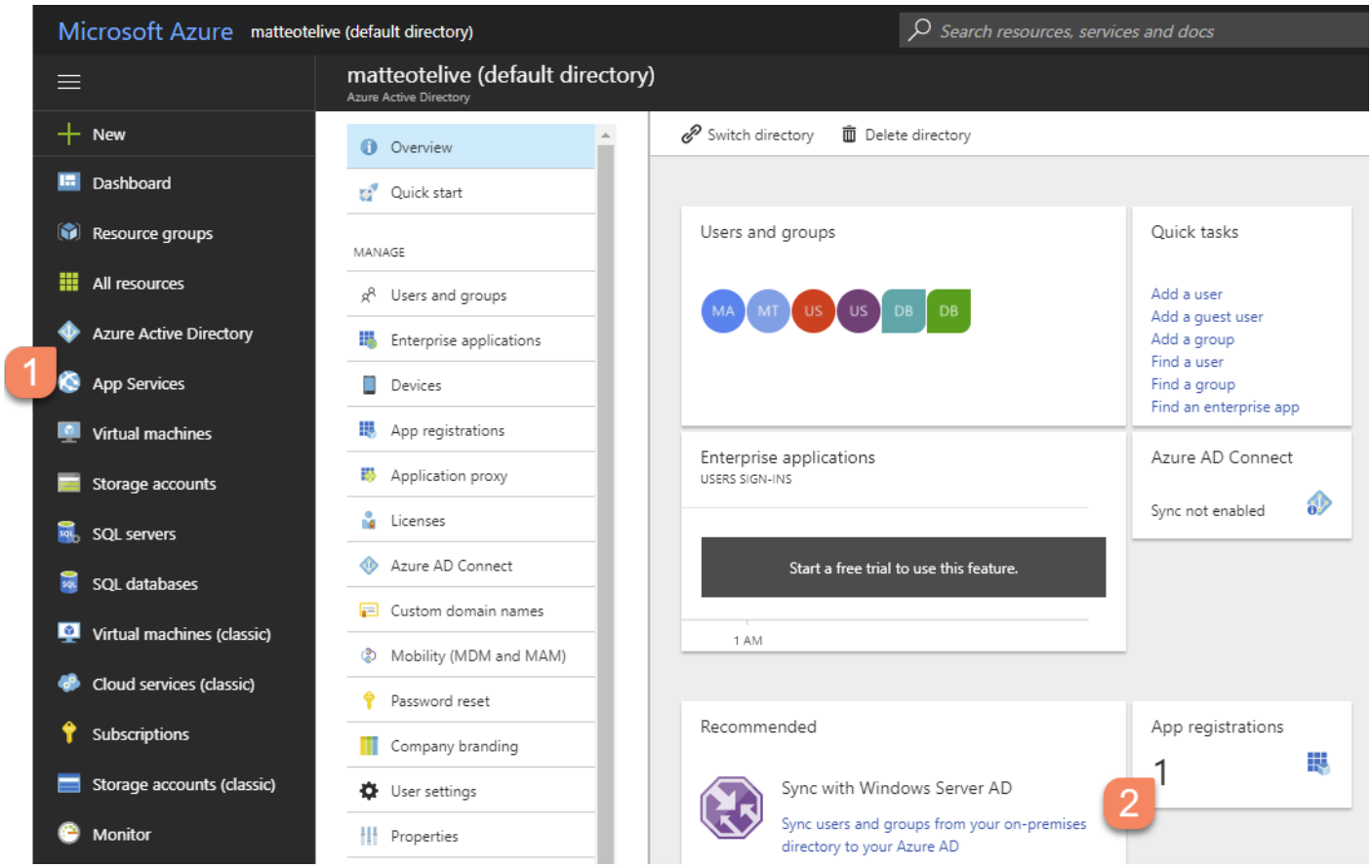


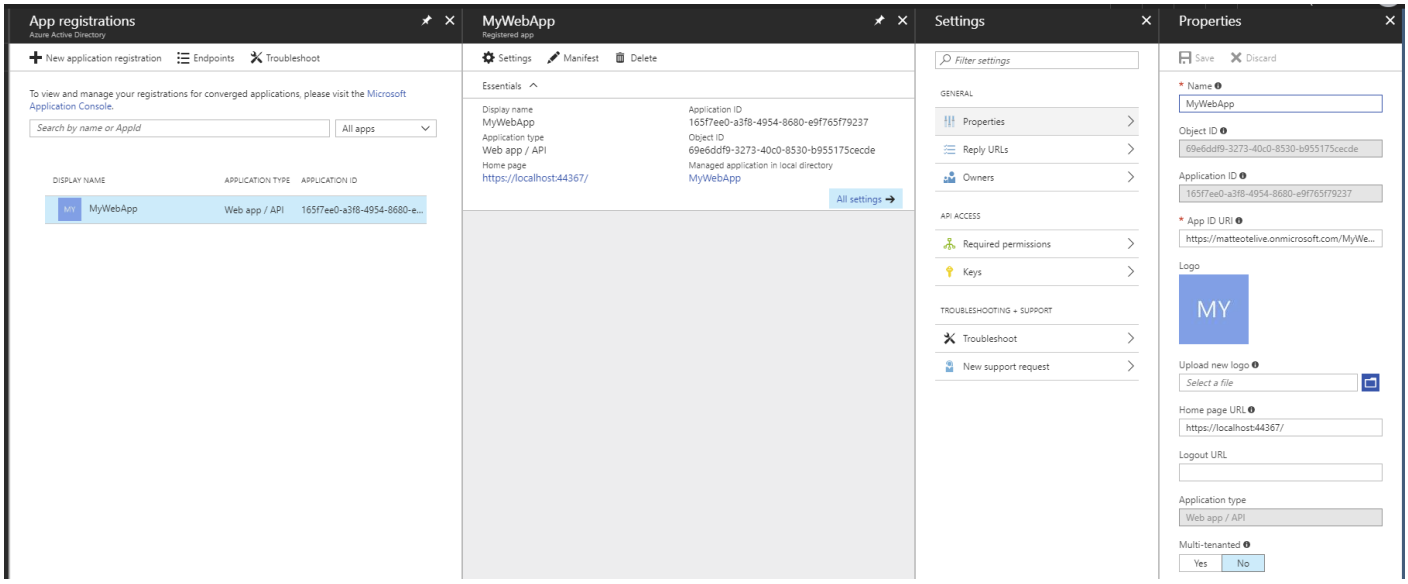
Click OK to complete the project creation.

Examine the app registration in Azure AD

The Visual Studio wizard takes care of creating an app registration in Azure AD for the newly created application.

You can check the app registration from the Azure Portal, selecting Azure Active Directory and clicking App registrations.

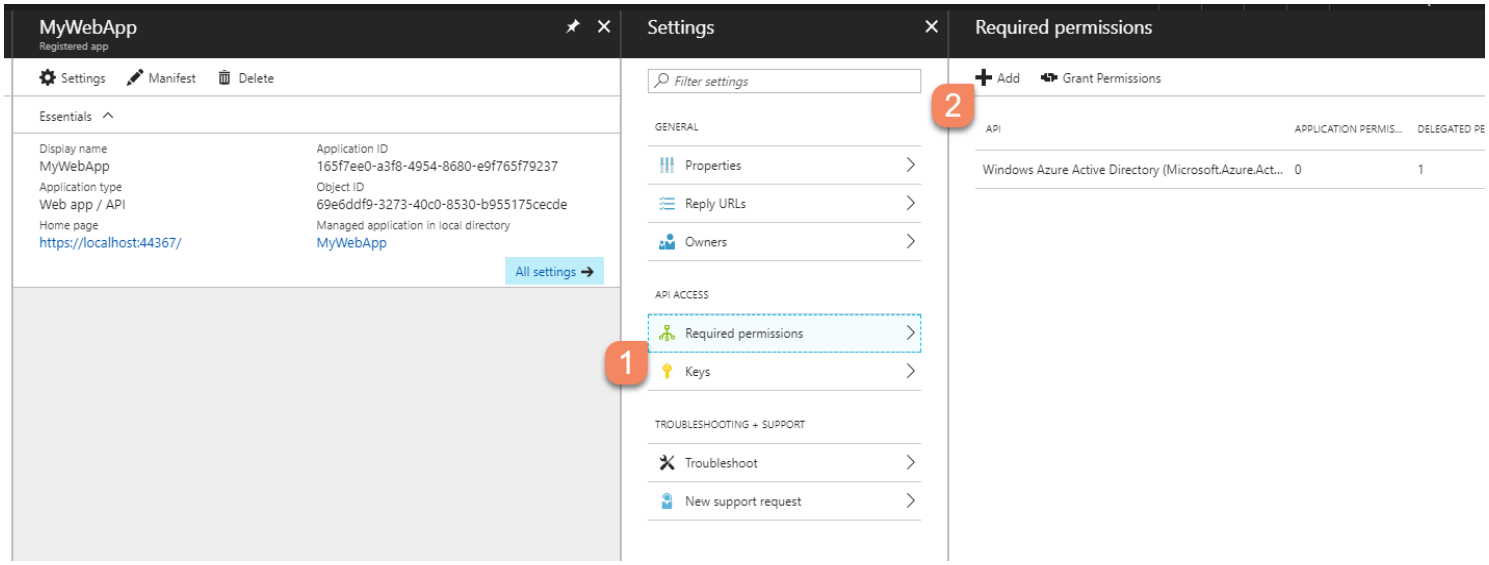




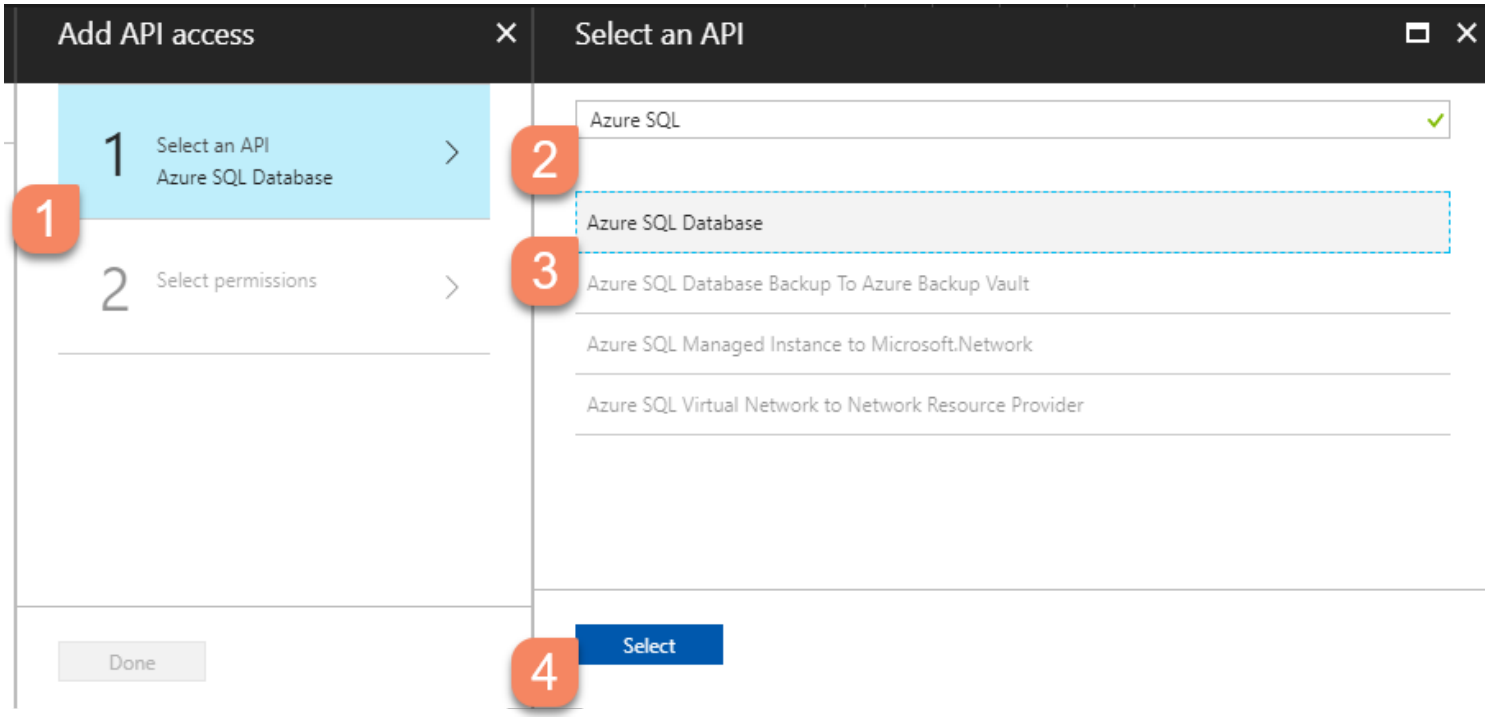
Grant delegated permission to the app registration

The application must be granted delegated permissions to access Azure SQL Database on behalf of users.

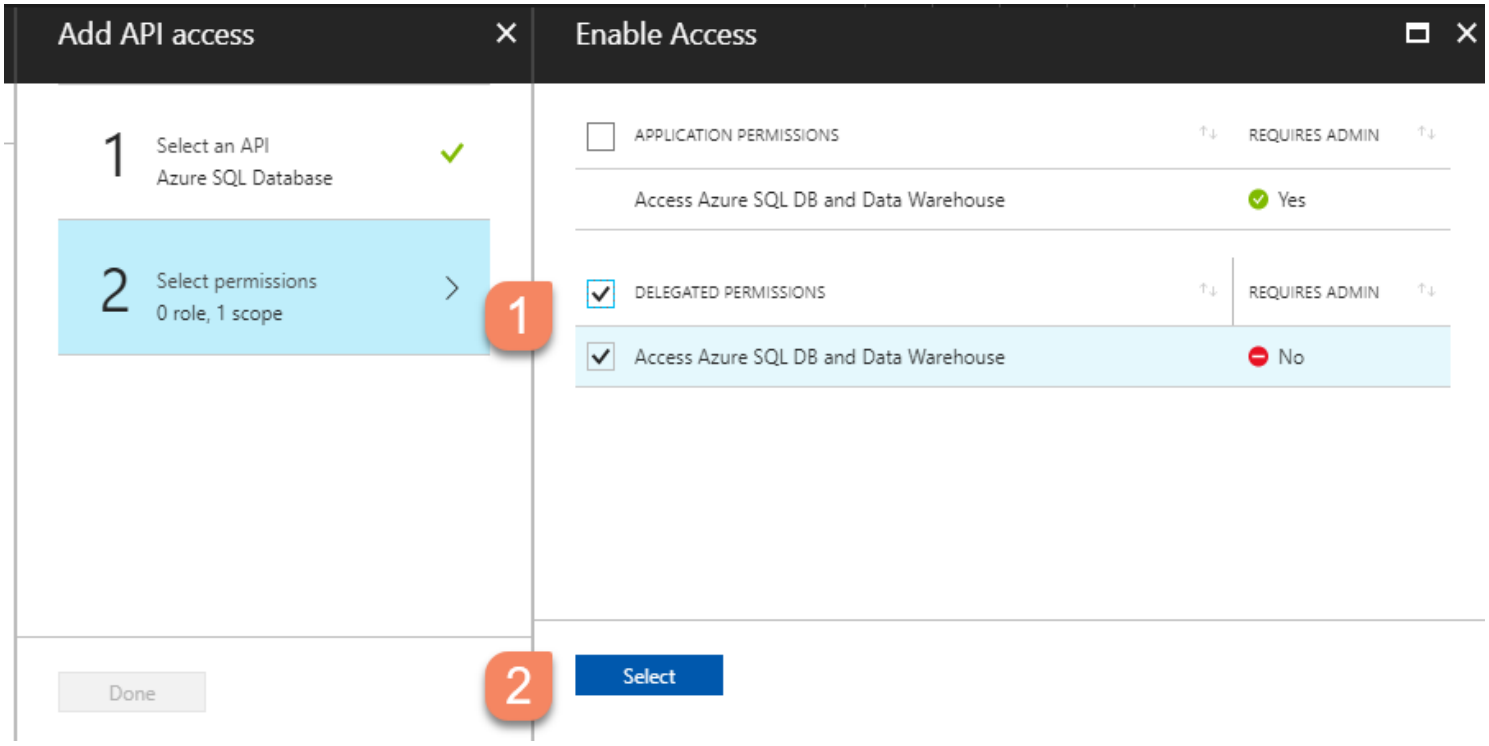
Open the app registration in the Azure Portal, click Required permissions and click Add.



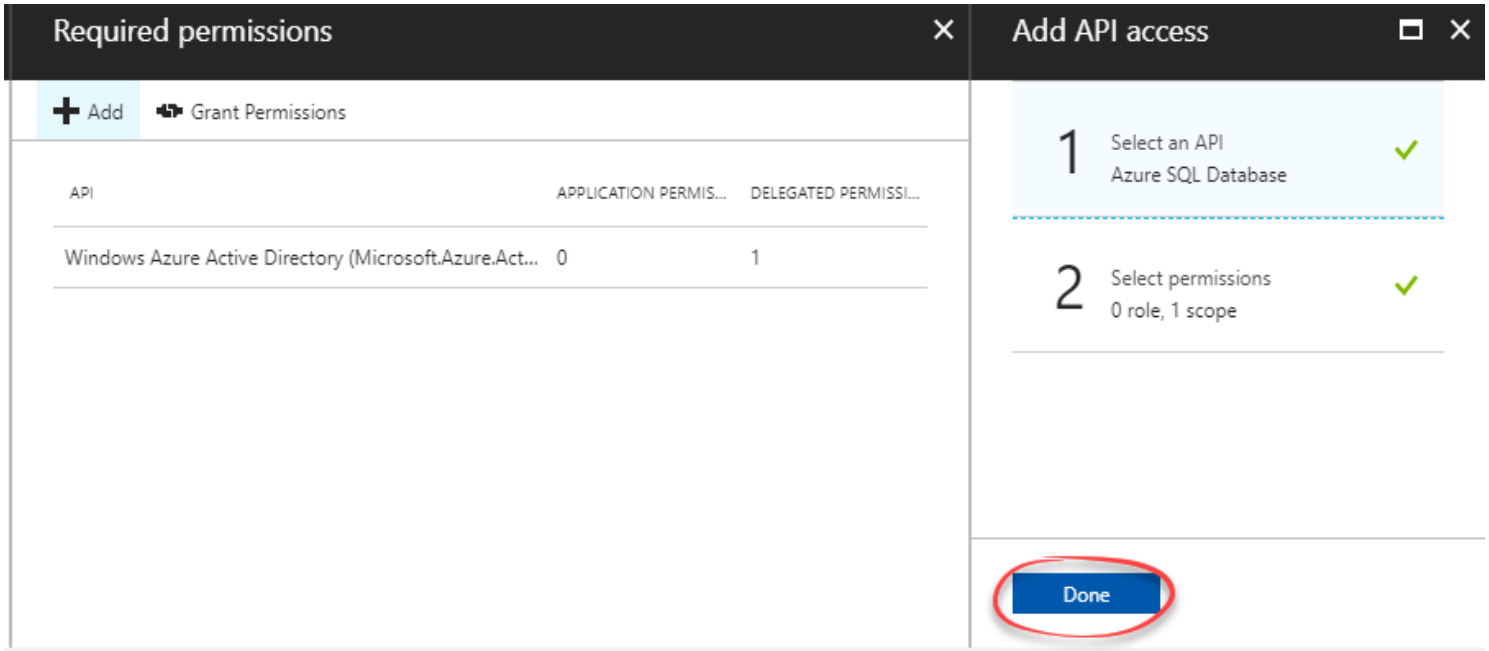
Select "Select an API", type "Azure SQL" in the search textbox, select Azure SQL Database in the search results and click Select.



In "Select permissions", click DELEGATED PERMISSIONS and click Select.



Click Done to apply the changes.



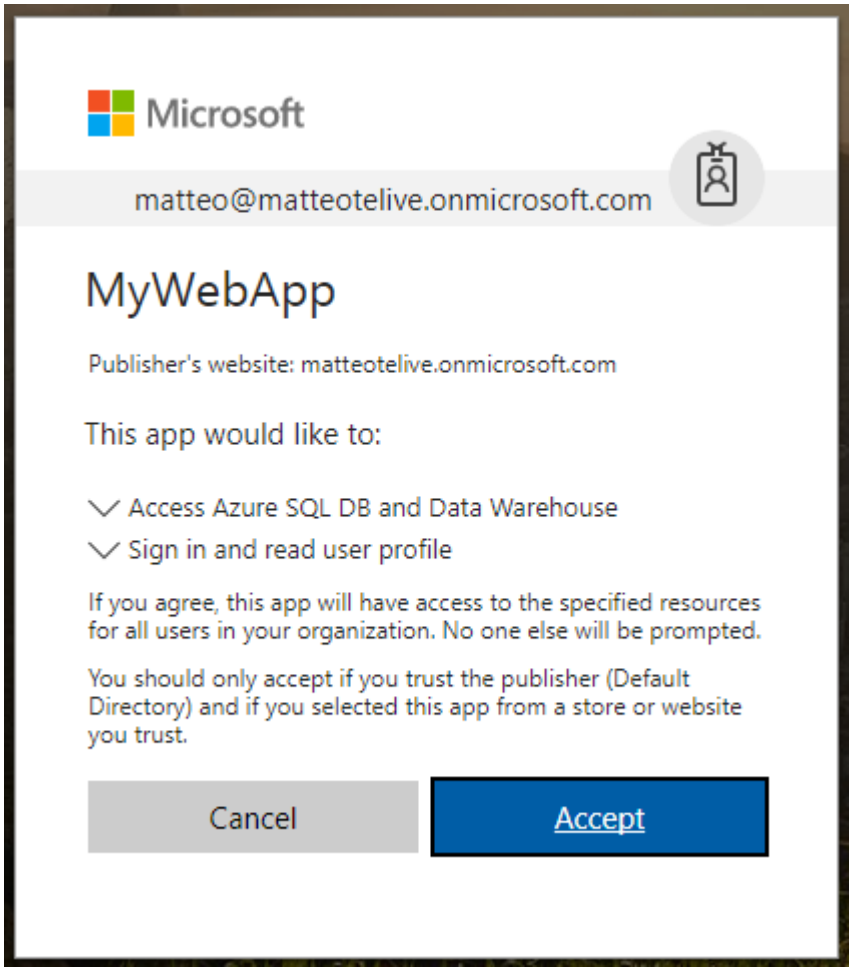
Authorize the application

Access the following URL to authorize the application

https://login.microsoftonline.com/**matteotelive.onmicrosoft.com**/oauth2/authorize?client_id=**165f7ee0-a3f8-4954-8680-e9f765f79237**&response_type=id_token&nonce=1234&scope=openid&prompt=admin_consent

Replace the domain with your Azure AD domain and the client_id with the Application ID of the app registration.

When prompted for the authorization, click Accept.



Examine Web.config

In addition to creating the app registration, Visual Studio populates Web.config with settings to support Azure AD authentication.

ida:Tenant is the Azure AD tenant used for authentication.

ida:Audience is the App ID URI that identifies the application.

ida:ClientId corresponds to the Application ID in the app registration.

ida:Password is a key from the app registration (manageable from the Keys blade).

```
<configuration>
  <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" />
    <add key="ida:Tenant" value="matteotelive.onmicrosoft.com" />
    <add key="ida:Audience" value="https://matteotelive.onmicrosoft.com/MyWebApp" />
    <add key="ida:ClientID" value="165f7ee0-a3f8-4954-8680-e9f765f79237" />
    <add key="ida:Password" value="****************************************" />

  </appSettings>

  […]
```

Add settings to web.config

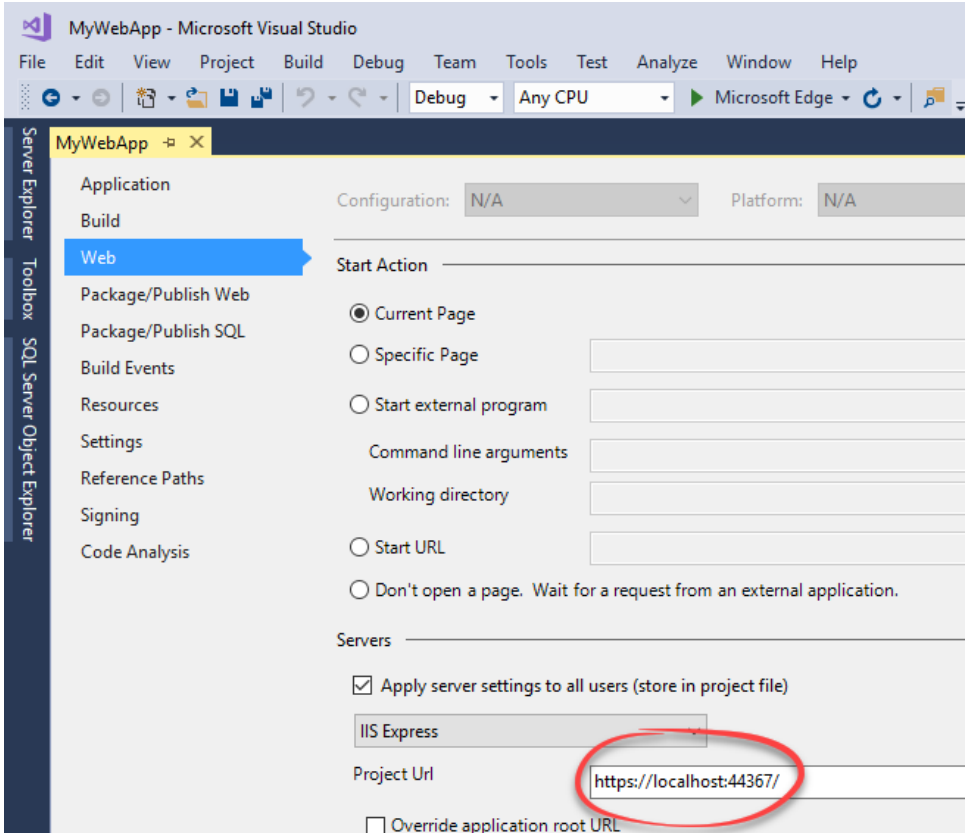
You need to add two settings to web.config.

```
<configuration>
  <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" />
    <add key="ida:Tenant" value="matteotelive.onmicrosoft.com" />
    <add key="ida:Audience" value="https://matteotelive.onmicrosoft.com/MyWebApp" />
    <add key="ida:ClientID" value="165f7ee0-a3f8-4954-8680-e9f765f79237" />
    <add key="ida:Password" value="****************************************" />
    <add key="ida:Authority" value="https://login.microsoftonline.com/"/>
    <add key="ida:PostLogoutRedirectUri" value="https://localhost:44367/"/>

  </appSettings>

  […]
```

ida:Authority must be set to https://login.microsoftonline.com/

ida:PostLogoutRedirectUri must be set to the address of the application. This walkthrough assumes that the application runs locally.

You can find the address of the application in the project Web settings.



Install additional packages

The application must be modified to use Cookie and OpenID authentications, which require additional NuGet packages.

In Visual Studio, open the Package Manager Console and run the following commands.

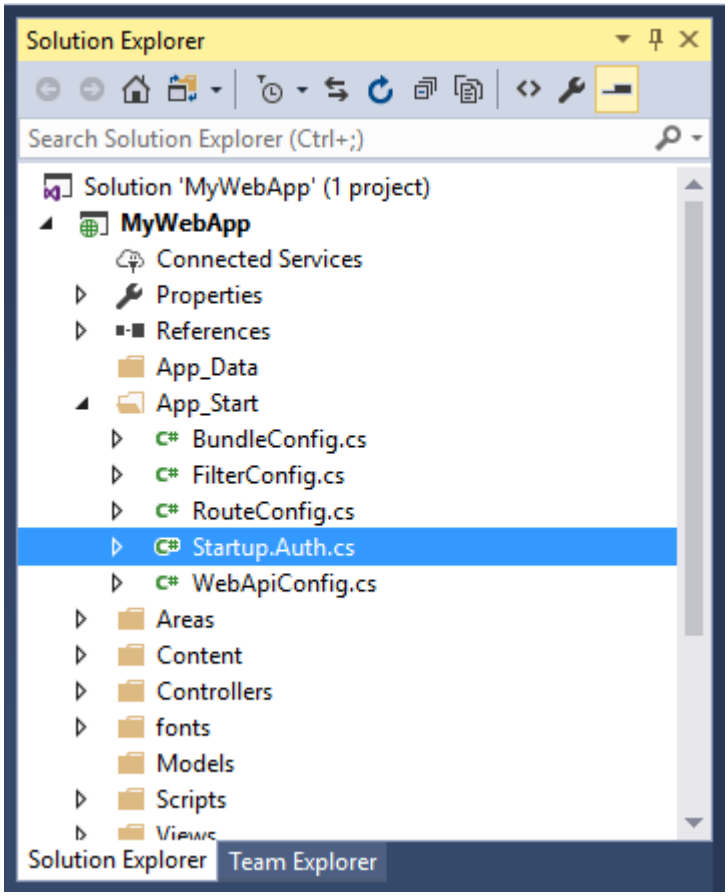
Install-Package Microsoft.Owin.Security.OpenIdConnect -Version 3.1.0

Install-Package Microsoft.Owin.Security.Cookies -Version 3.1.0

Install-Package Microsoft.IdentityModel.Clients.ActiveDirectory -Version 3.17.3

Modify the authentication methods

Open Startup.Auth.cs to modify the authentication middleware.



Replace the file content with the following.

```
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.OpenIdConnect;
using Owin;
using System.Configuration;
using System.IdentityModel.Tokens;
namespace MyWebApp
{
    public partial class Startup
    {
        public void ConfigureAuth(IAppBuilder app)
        {
            app.SetDefaultSignInAsAuthenticationType(CookieAuthenticationDefaults.AuthenticationType);
            app.UseCookieAuthentication(new CookieAuthenticationOptions());
            app.UseOpenIdConnectAuthentication(
                new OpenIdConnectAuthenticationOptions
                {
                    ClientId = ConfigurationManager.AppSettings["ida:ClientID"],
                    Authority = ConfigurationManager.AppSettings["ida:Authority"]
                        + ConfigurationManager.AppSettings["ida:Tenant"],
                    PostLogoutRedirectUri =
                        ConfigurationManager.AppSettings["ida:PostLogoutRedirectUri"],
                    TokenValidationParameters = new TokenValidationParameters
                    {
                        SaveSigninToken = true
                    }
                });
        }
    }
}
```

The code above enables cookie and OpenID authentication methods.

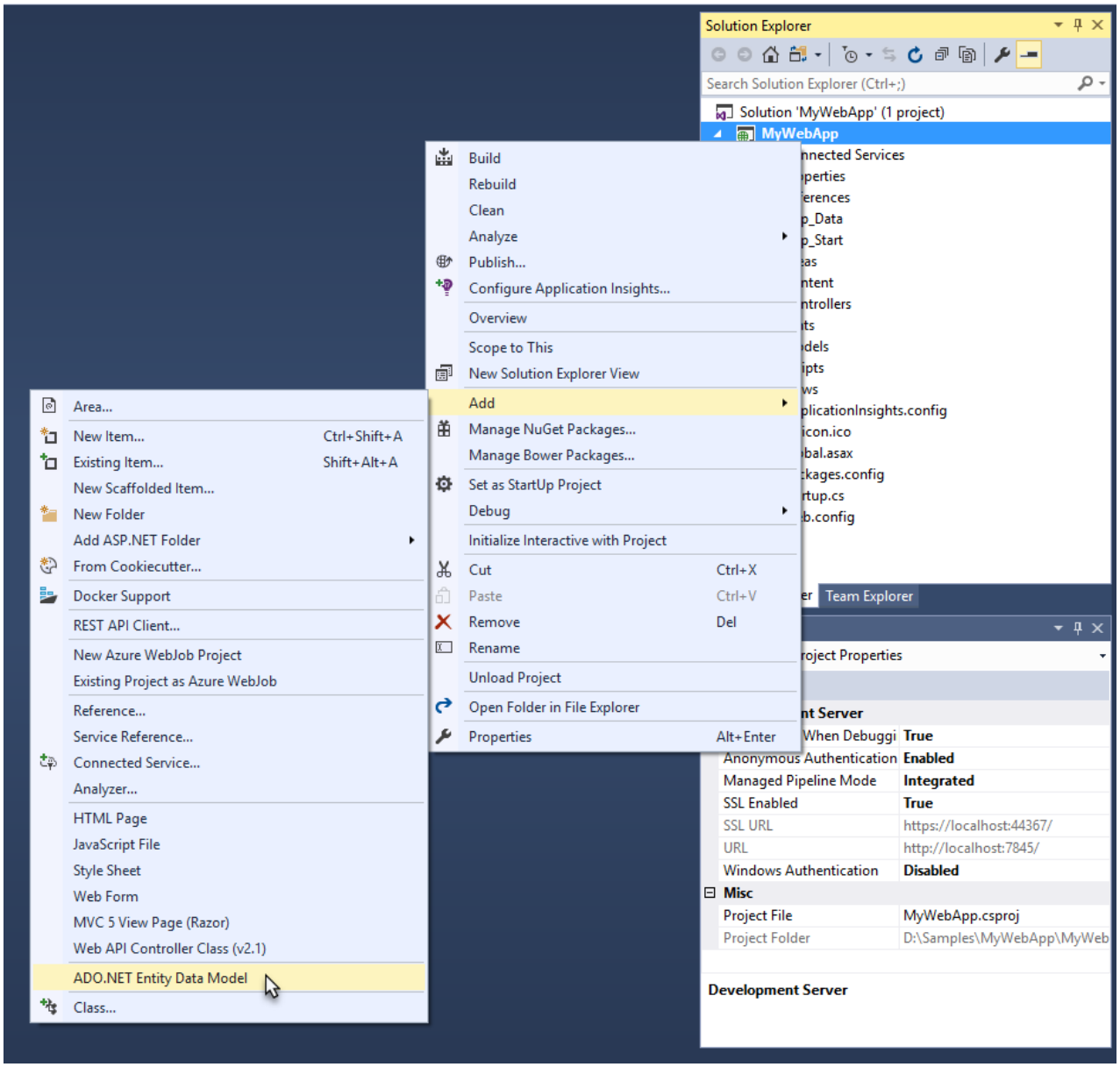
OpenID authentication is initialized with information regarding the app registration, the authentication authority and the Uri for post-authentication redirection.

Additionally TokenValidationParameters specifies that the sign-in token received from the client must be saved (SaveSigninToken = true) so that it can be later retrieved when the application requests a token to access Azure SQL Database on behalf of the user.

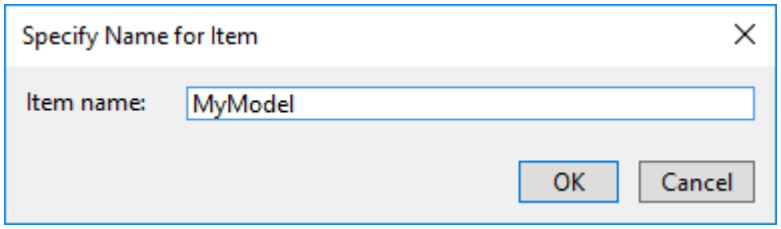
Create a new model and generate the database schema

Create a new Entity Framework model.

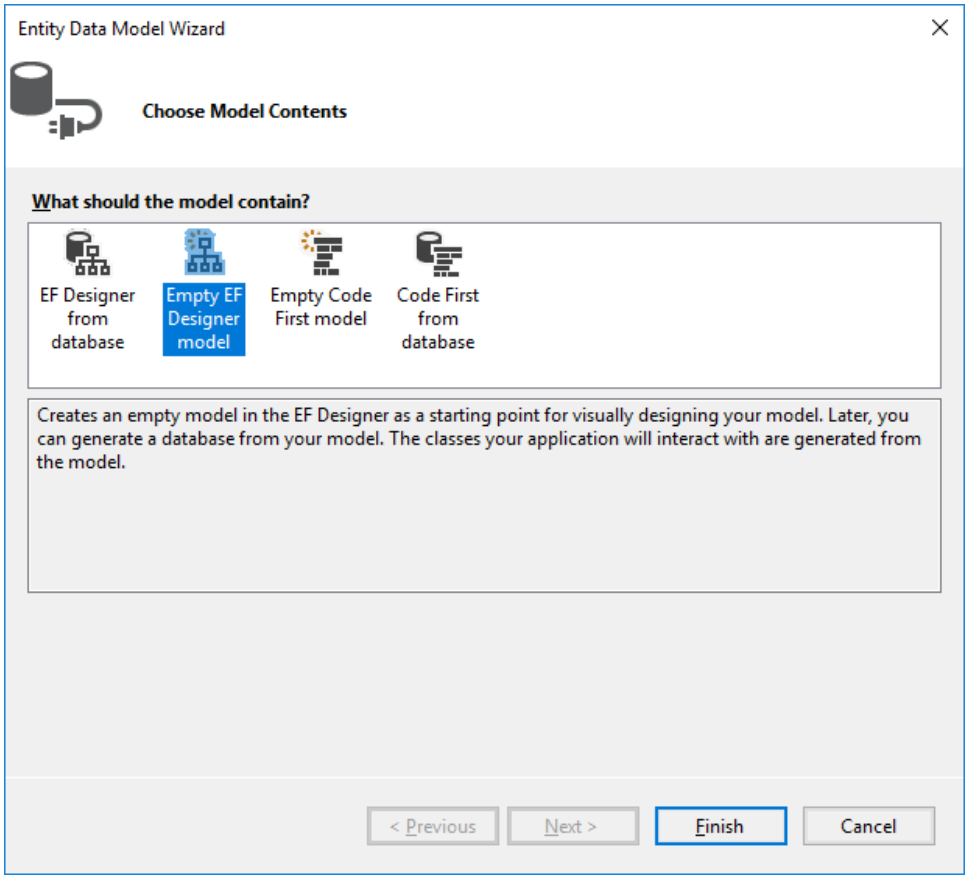
Right-click the project, click Add and select ADO.NET Entity Data Model.



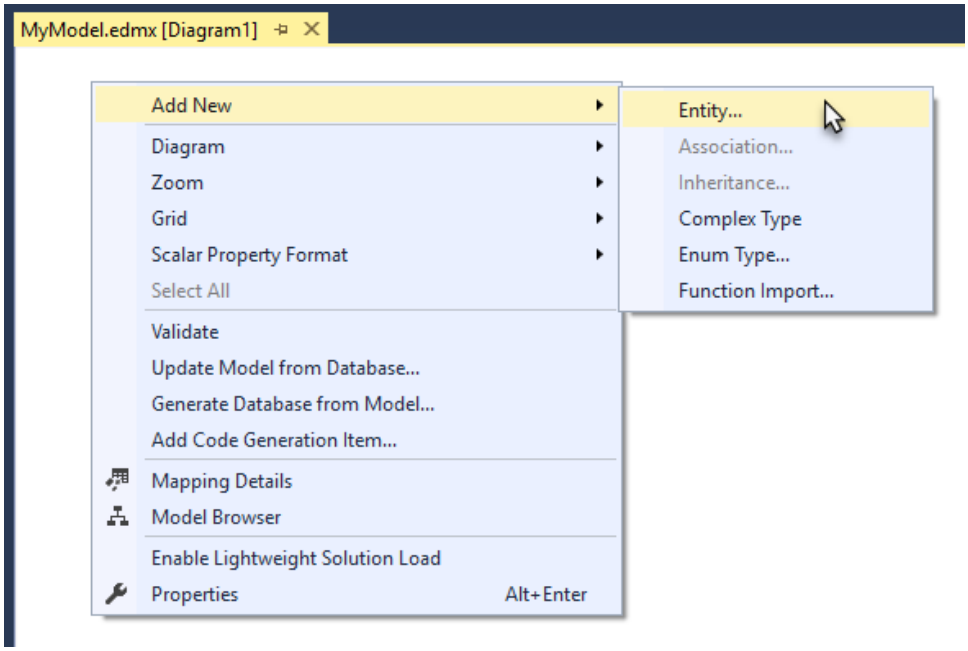
Provide a name for the new model and click OK. We will use MyModel for this example.



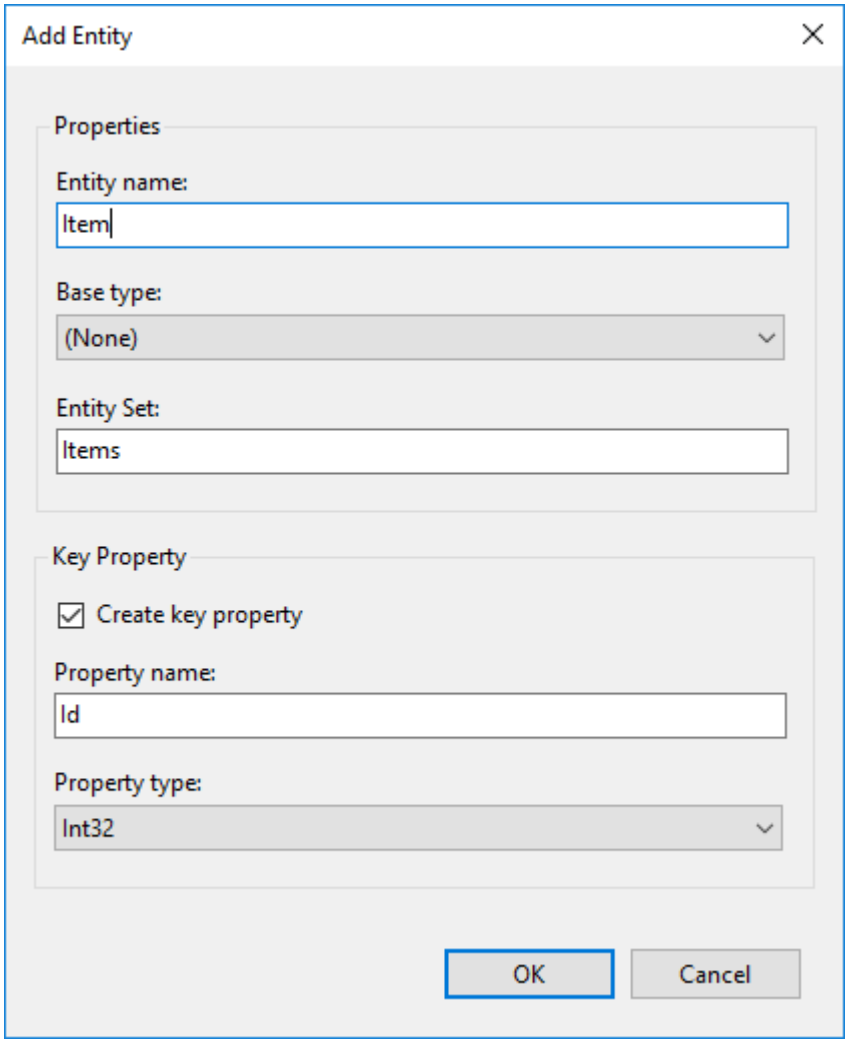
Select "Empty EF Designer model" and click Finish.



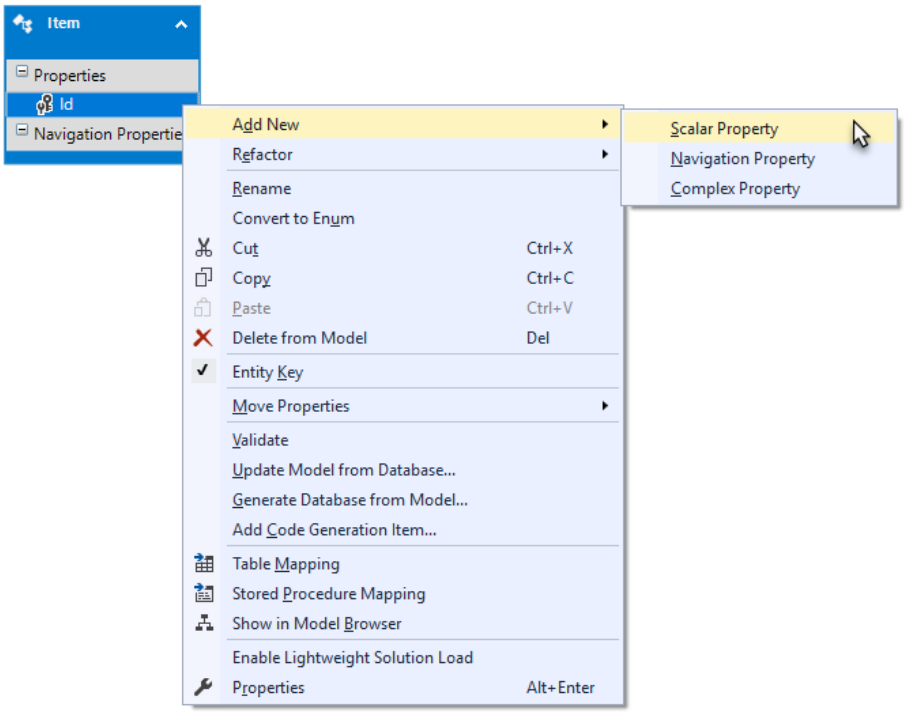
Right-click on the model, select Add New and click Entity...



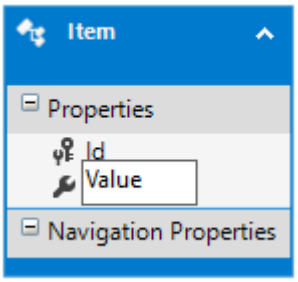
Provide a name for the new entity and click OK.



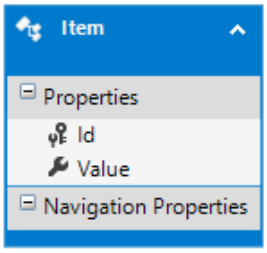
Right-click on the new entity, select Add New and click Scalar Property to add a new field.

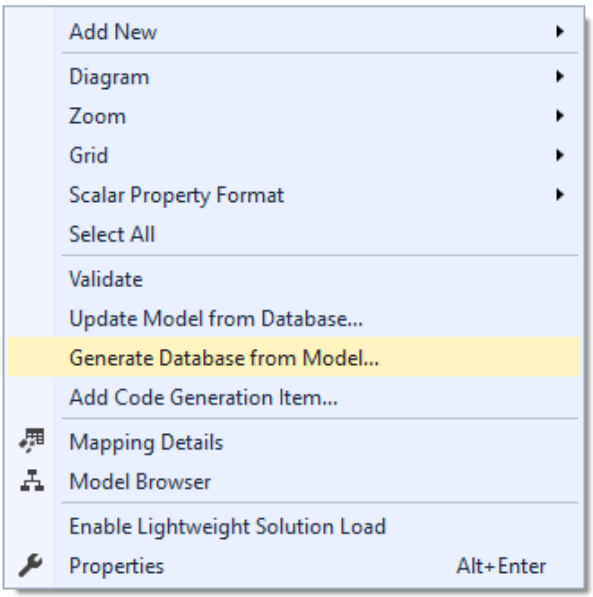


Provide a name for the new field and press Enter.

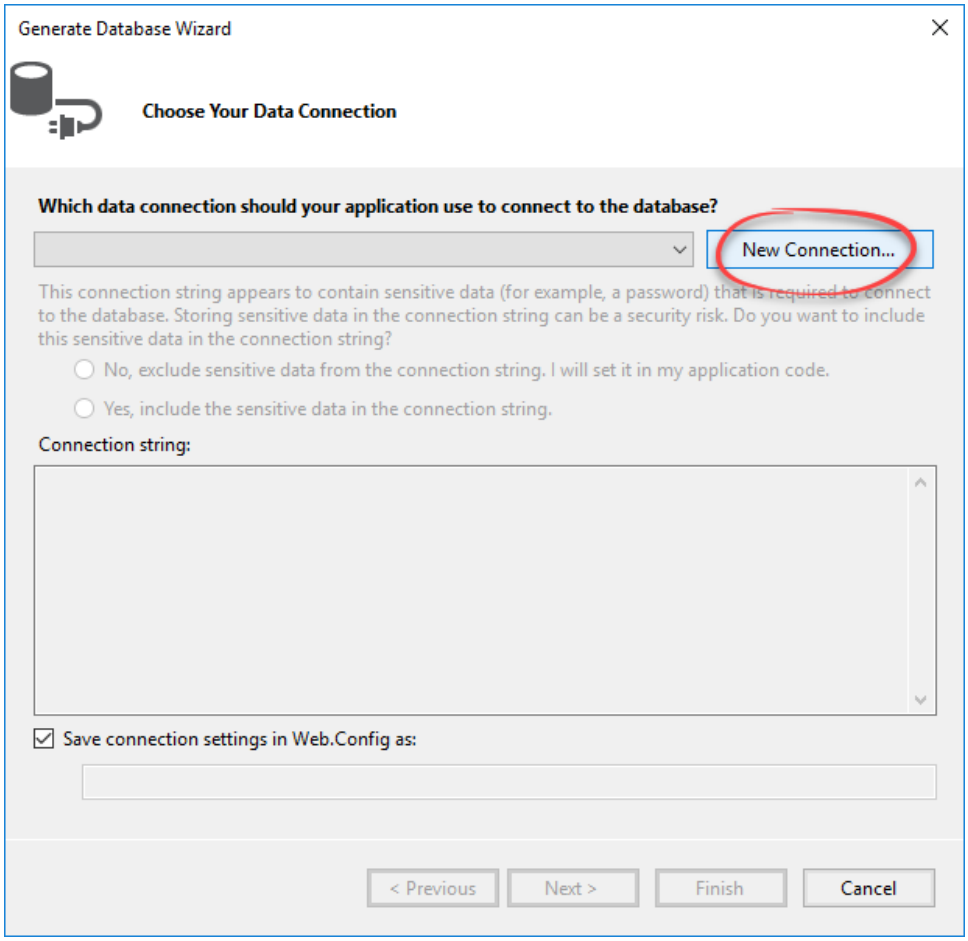


Right-click on the model and select Generate Database from Model...



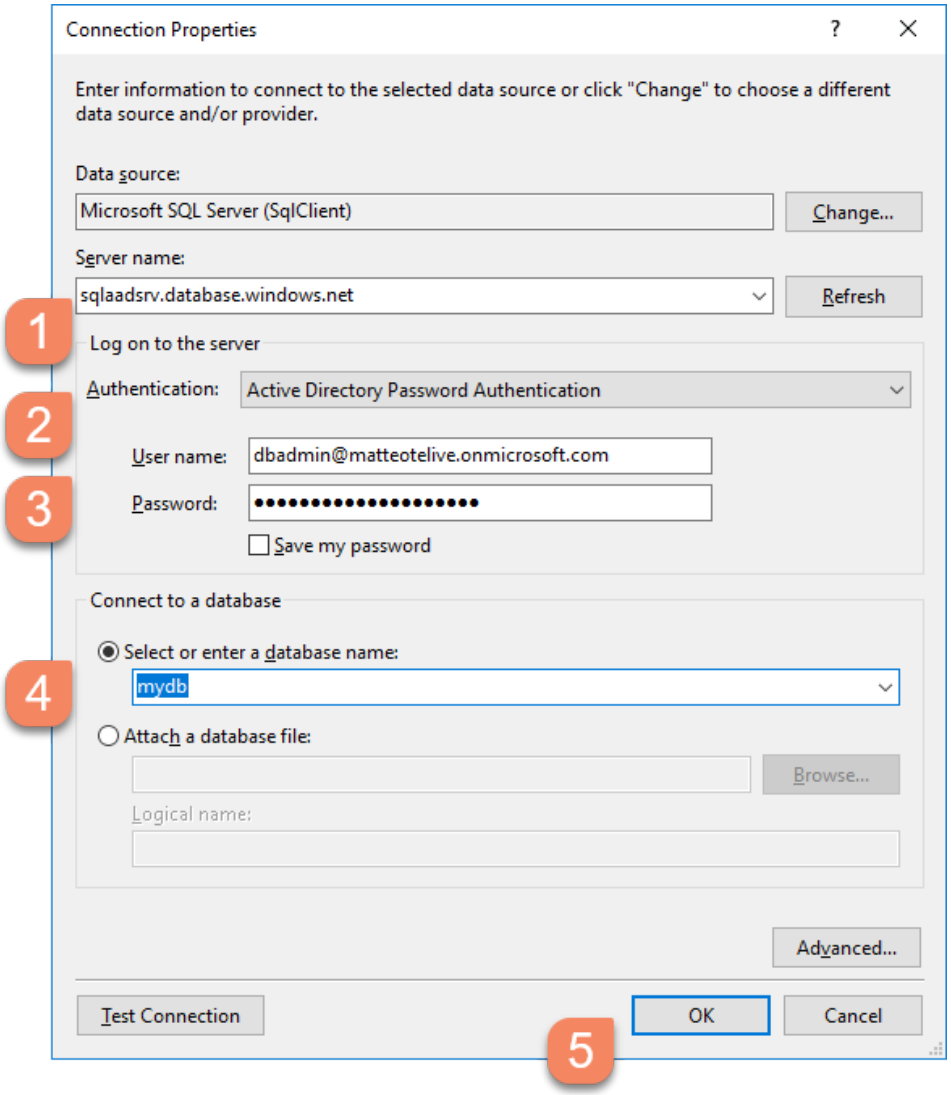


Click New Connection...



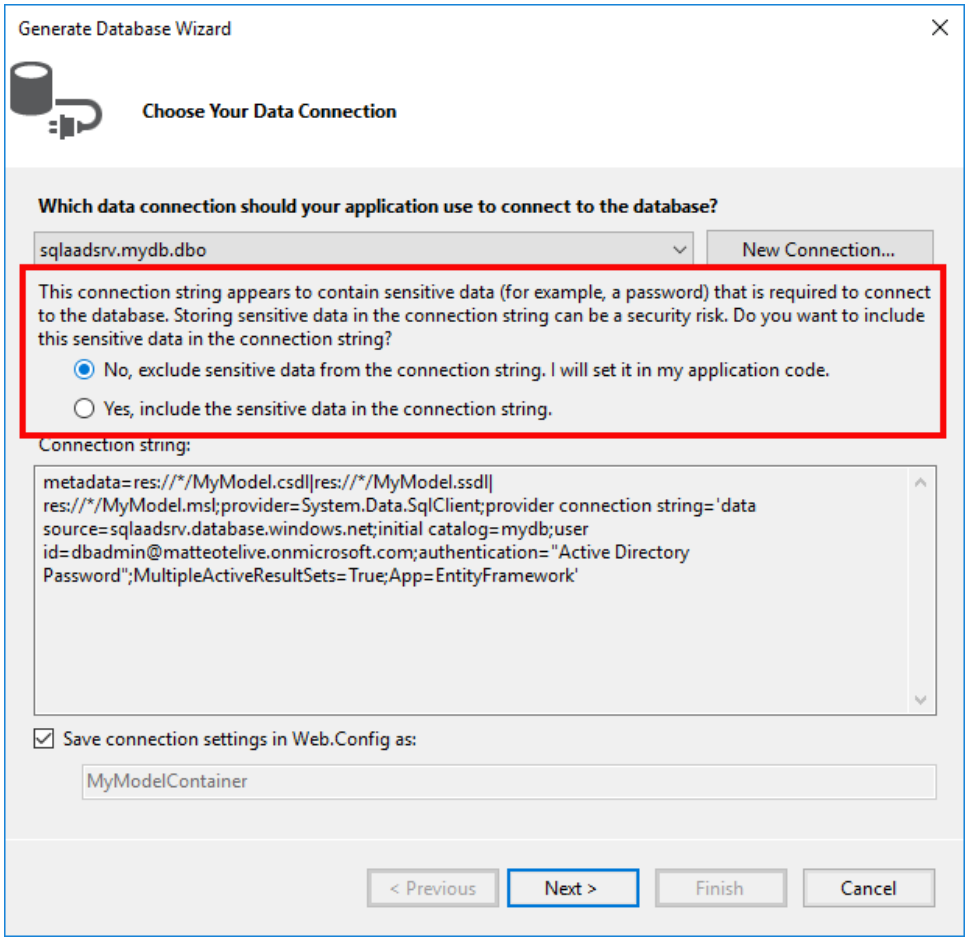
Enter the server name, select an authentication method, enter credentials, specify the database name and click OK.

Note: you need to connect to the database with a user with administrative privileges. This step can be performed either with Azure AD authentication or SQL authentication.

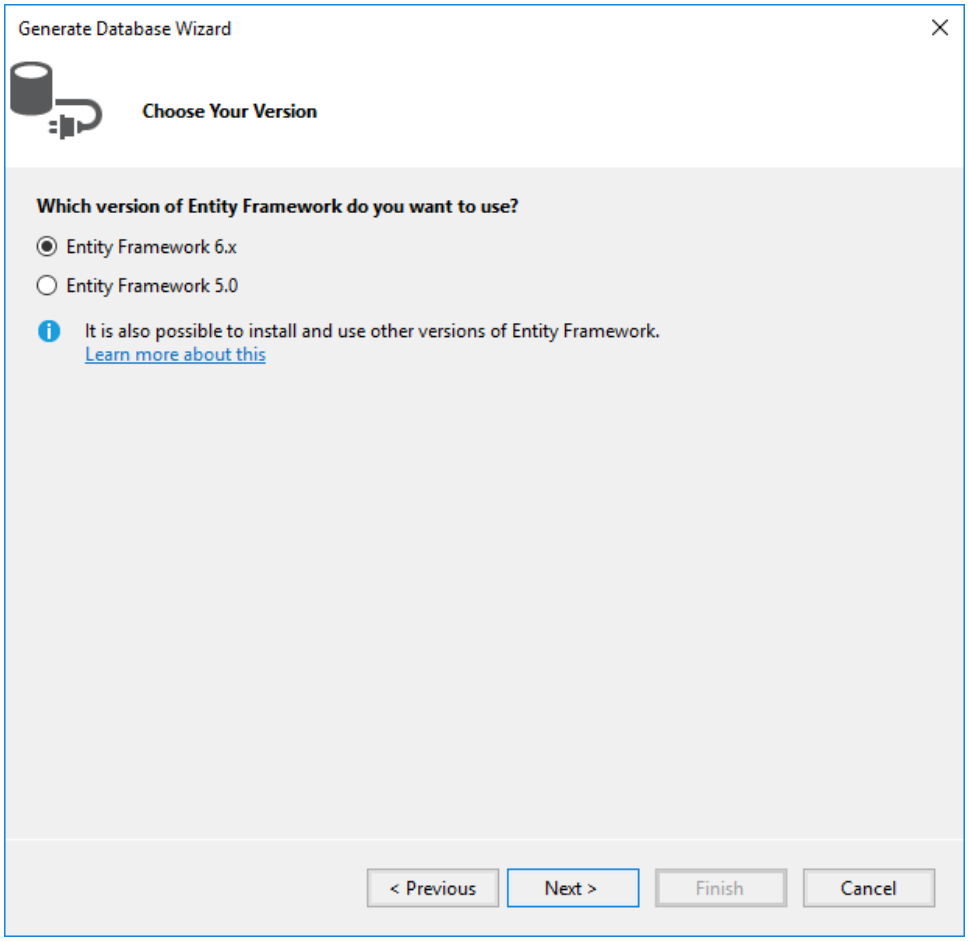


If you are asked to specify whether you want to save sensitive information to the connection string, you can choose any of the settings; the connection string will be changed later anyway to remove any user reference.

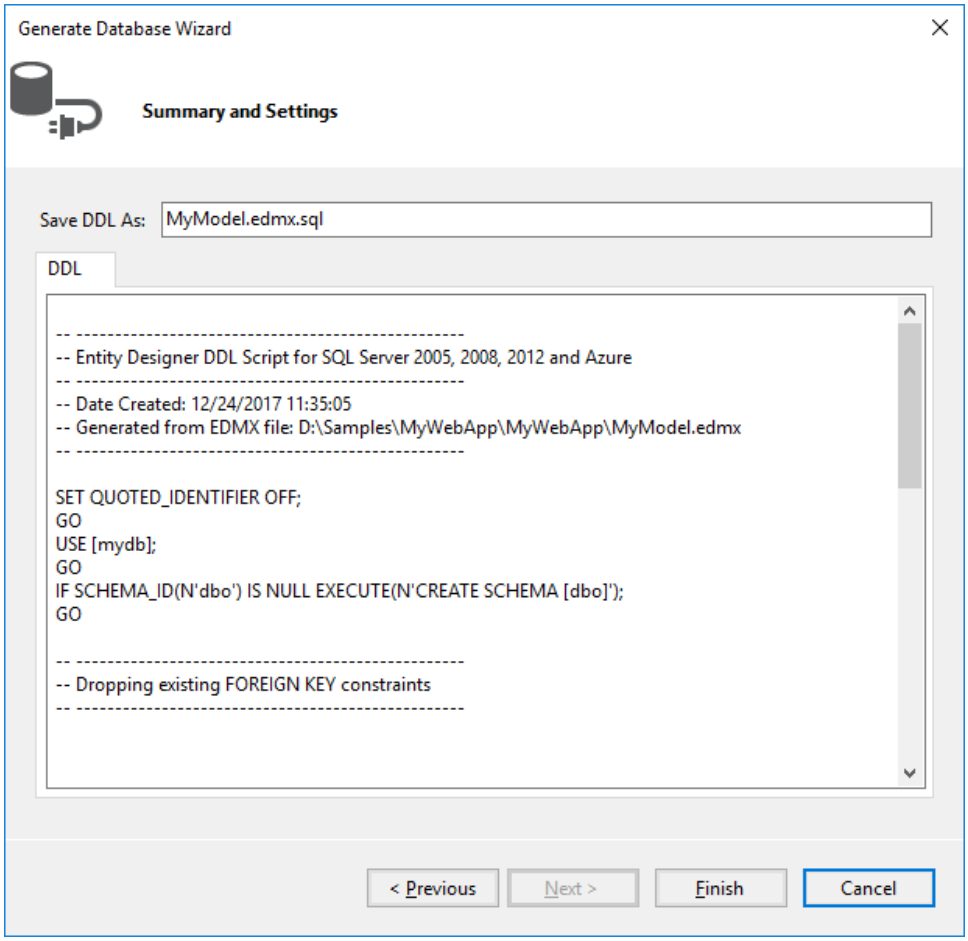
Click Next.



Select Entity Framework 6.x and click Next.

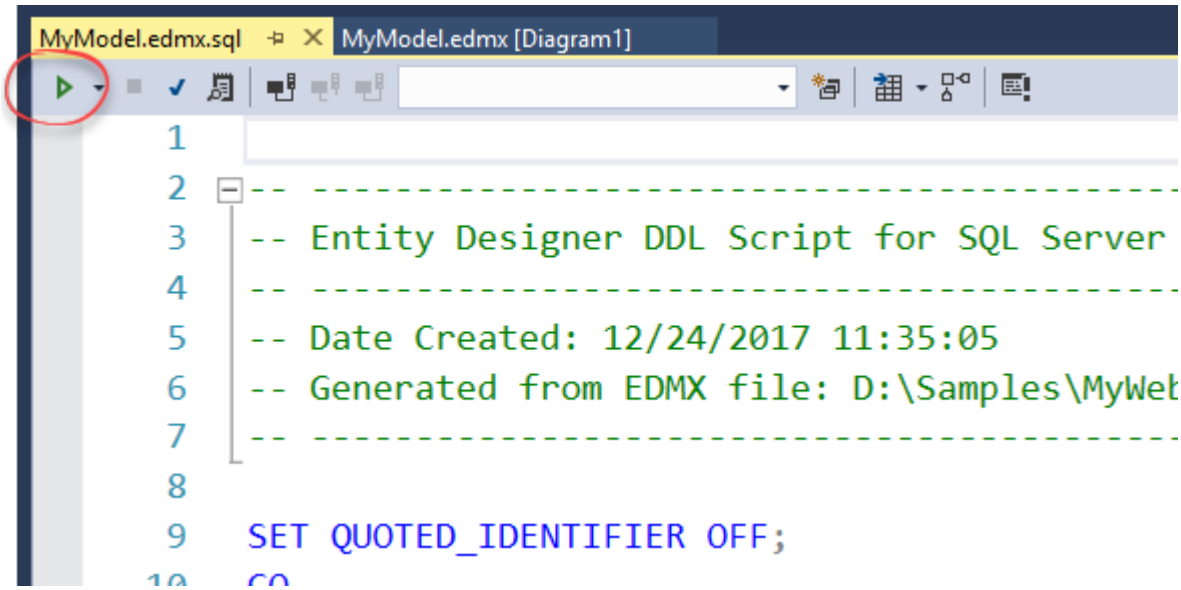


When the DDL is ready, click Finish.



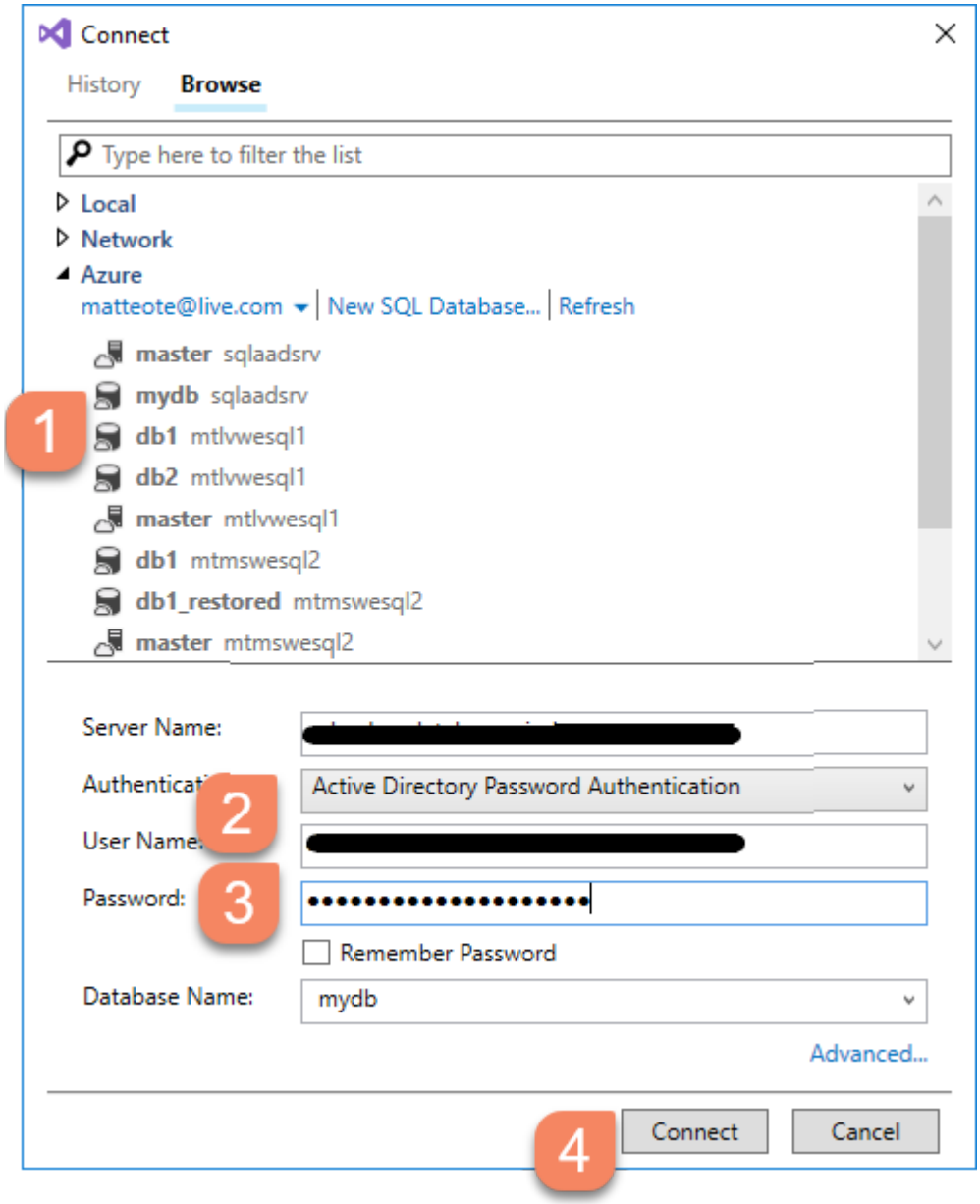
The wizard generates a T-SQL Script to create the database schema. You need to run the script on Azure SQL Database.

Click the Execute button.



In the Connect dialog box, select the database, select Active Directory Password Authentication, enter the credentials of the Azure AD admin and click Connect.

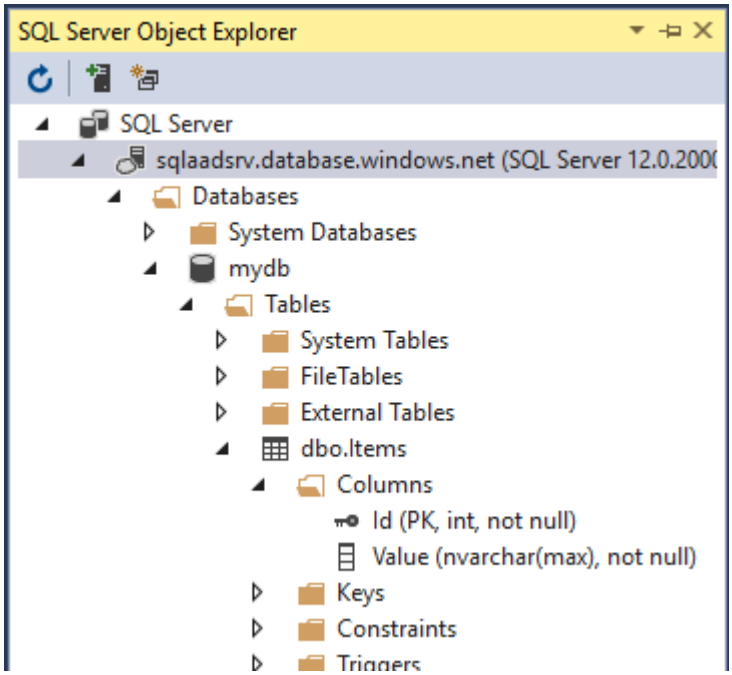
Note: you need to connect to the database with a user with administrative privileges. This step can be performed either with Azure AD authentication or SQL authentication.



The script execution should end with a "Command(s) completed successfully." message.

At this point the database has been populated with new objects to reflect the model.

You can use the SQL Server Object Explorer to navigate the structure of the database.

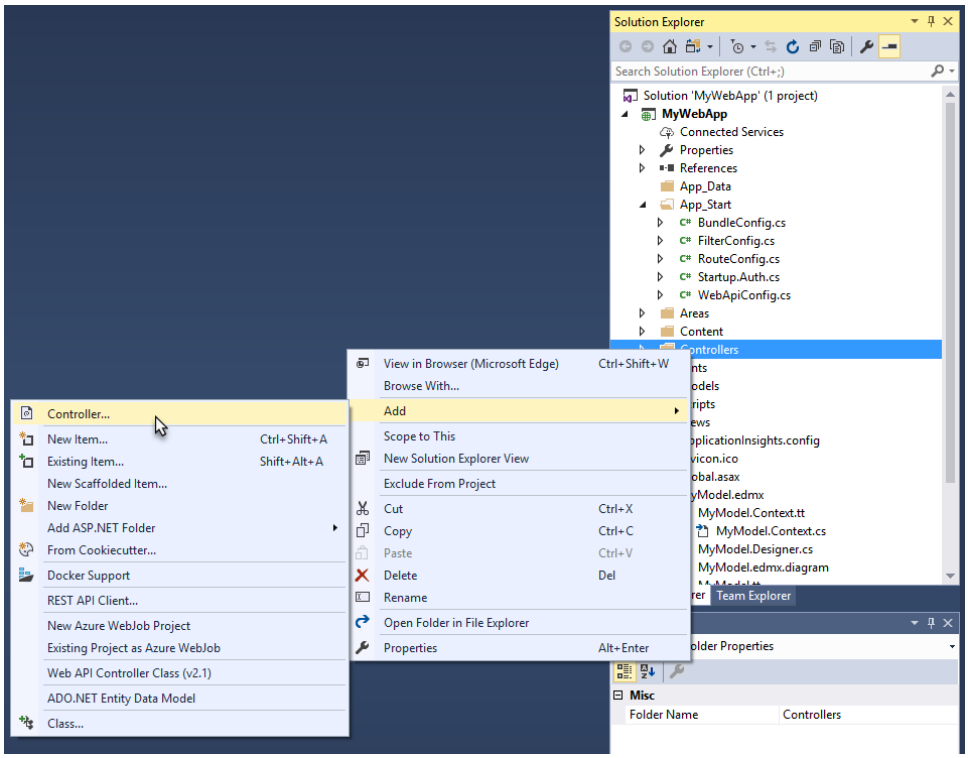


Create a controller for the EF entity

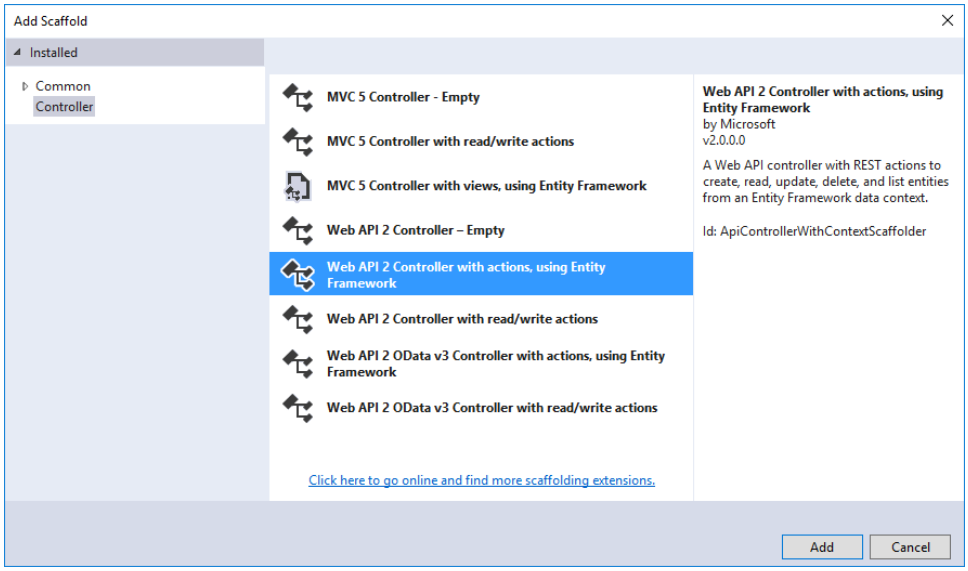
You need to create a new controller to consume the new EF entity.

Rebuild the solution.

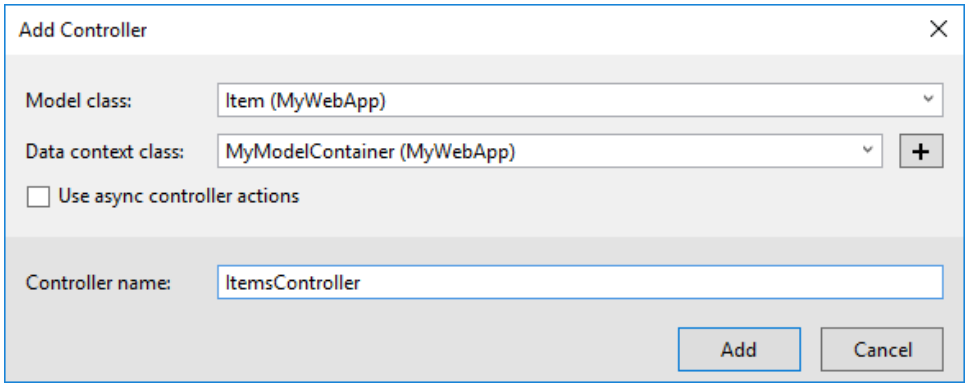
Right-click Controllers, select Add and click Controller...



Select "Web API 2 Controller with actions, using Entity Framework" and click Add.



Select the entity class, select the DbContext class and click Add.



Replace the code with the following.

```
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using System.Net;
using System.Web.Http;
using System.Web.Http.Description;

namespace MyWebApp.Controllers
{
    [Authorize] // changed by the wizard
    public class ItemsController : ApiController
    {
        private MyModelContainer _db = null;        // changed by the wizard
        private MyModelContainer db {               // changed by the wizard
            get                                     // changed by the wizard
            {                                       // changed by the wizard
                if (_db == null)                    // changed by the wizard
                {                                   // changed by the wizard
                    _db = new MyModelContainer();   // changed by the wizard
                }                                   // changed by the wizard
                return _db;                         // changed by the wizard
            }                                       // changed by the wizard
        }

        // GET: api/Items
        public IQueryable<Item> GetItems()
        {
            return db.Items;
        }

        // GET: api/Items/5
        [ResponseType(typeof(Item))]
        public IHttpActionResult GetItem(int id)
        {
            Item item = db.Items.Find(id);
            if (item == null)
            {
                return NotFound();
            }
            return Ok(item);
        }

        // PUT: api/Items/5
        [ResponseType(typeof(void))]
        public IHttpActionResult PutItem(int id, Item item)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            if (id != item.Id)
            {
                return BadRequest();
            }
            db.Entry(item).State = EntityState.Modified;
            try
            {
                db.SaveChanges();
            }
            catch (DbUpdateConcurrencyException)
            {
                if (!ItemExists(id))
                {
                    return NotFound();
                }
                else
                {
                    throw;
```

```
                }
            }
            return StatusCode(HttpStatusCode.NoContent);
        }

        // POST: api/Items
        [ResponseType(typeof(Item))]
        public IHttpActionResult PostItem(Item item)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            db.Items.Add(item);
            db.SaveChanges();
            return CreatedAtRoute("DefaultApi", new { id = item.Id }, item);
        }

        // DELETE: api/Items/5
        [ResponseType(typeof(Item))]
        public IHttpActionResult DeleteItem(int id)
        {
            Item item = db.Items.Find(id);
            if (item == null)
            {
                return NotFound();
            }
            db.Items.Remove(item);
            db.SaveChanges();
            return Ok(item);
        }

        protected override void Dispose(bool disposing)
        {
            if (disposing)
            {
                if (_db != null)        // changed by the wizard
                {                       // changed by the wizard
                    _db.Dispose();      // changed by the wizard
                }                       // changed by the wizard
            }
            base.Dispose(disposing);
        }

        private bool ItemExists(int id)
        {
            return db.Items.Count(e => e.Id == id) > 0;
        }
    }
}
```

The lines with the comment `// changed by the wizard` represent the changes to the code generated by the wizard.

The Authorize attribute is necessary to enable the authentication when accessing the controller.

The DbContext object would be normally created when the controller is instantiated, however the user token is not available at that time. In this example the DbContext object is initialized during the first user request.

The Dispose method was modified accordingly.

Alter the connection string in Web.config

Open Web.config and identify the connection string created by Entity Framework.

[...]

```
<connectionStrings>
  <add name="MyModelContainer" connectionString="metadata=res://*/MyModel.csdl|res://*/MyModel.ssdl|res://*/
</connectionStrings>
```

[...]

The provider connection string property is the following (your version may vary slightly):

```
data source=sqlaadsrv.database.windows.net;initial catalog=mydb;user id=dbadmin@matteotelive.onmicrosoft.com;a
```

The provider connection string must be changed to remove any reference to authentication details.

The result should look like this:

```
data source=sqlaadsrv.database.windows.net;initial catalog=mydb;MultipleActiveResultSets=True;App=EntityFramew
```

The connectionString section of Web.config should then look like this:

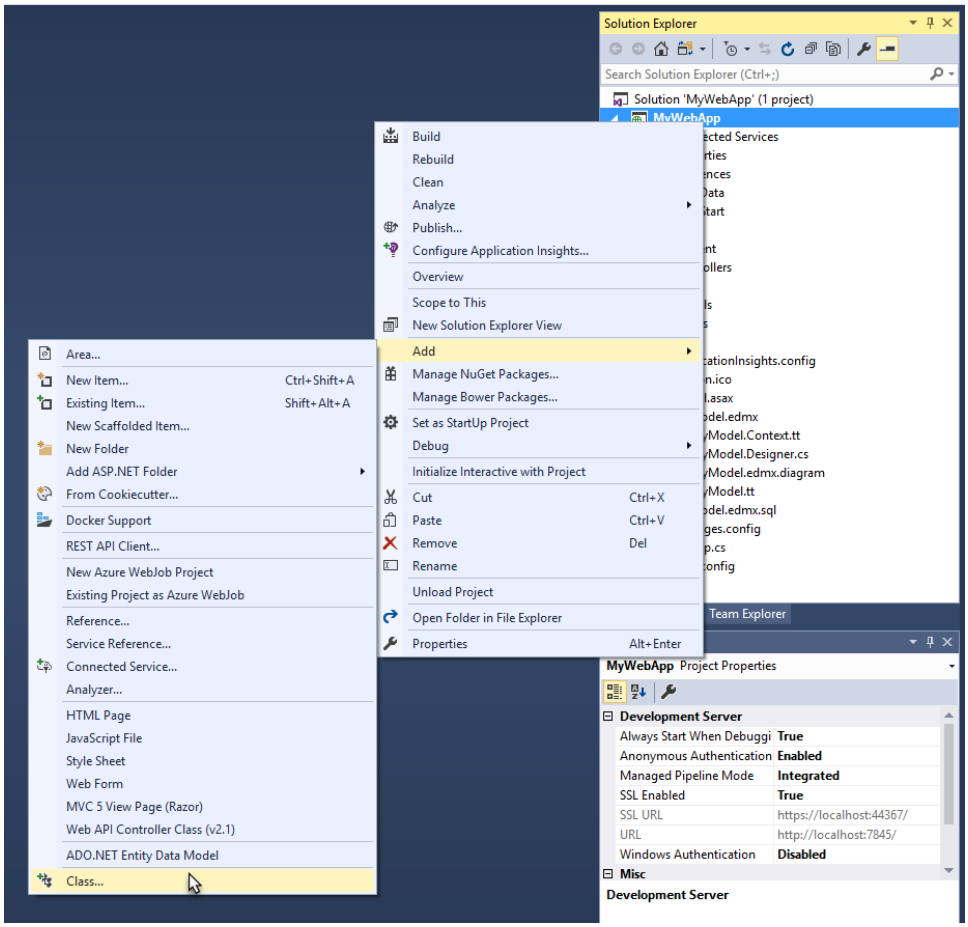
```
[…]
  <connectionStrings>
    <add name="MyModelContainer" connectionString="metadata=res://*/MyModel.csdl|res://*/MyModel.ssdl|res://*/
  </connectionStrings>
[…]
```

Add a TokenFactory class

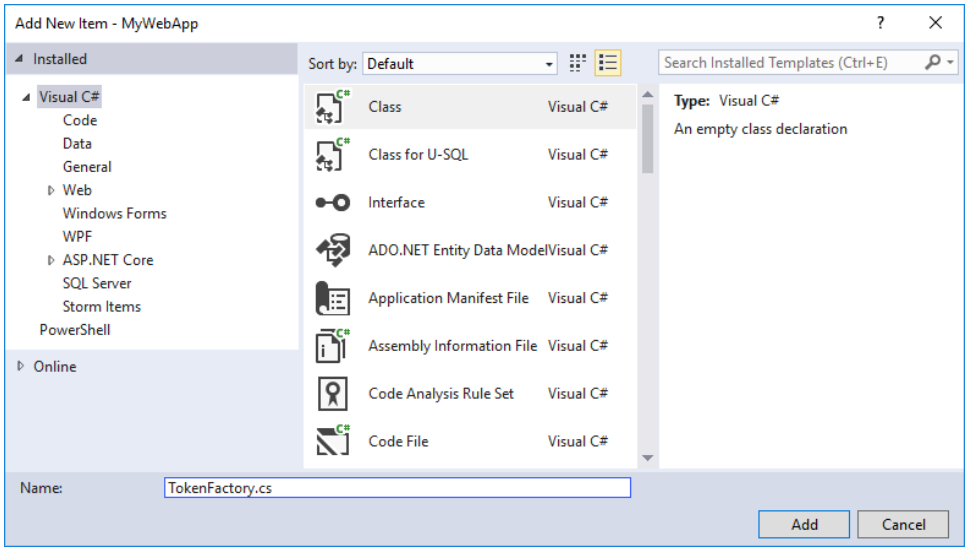
In this example the logic to obtain a token on behalf of the user is encapsulated in a static TokenFactory class.

Add a new class to the project.

Right click on the project, select Add and click Class...



Select "Class Visual C#" and name it TokenFactory.cs.

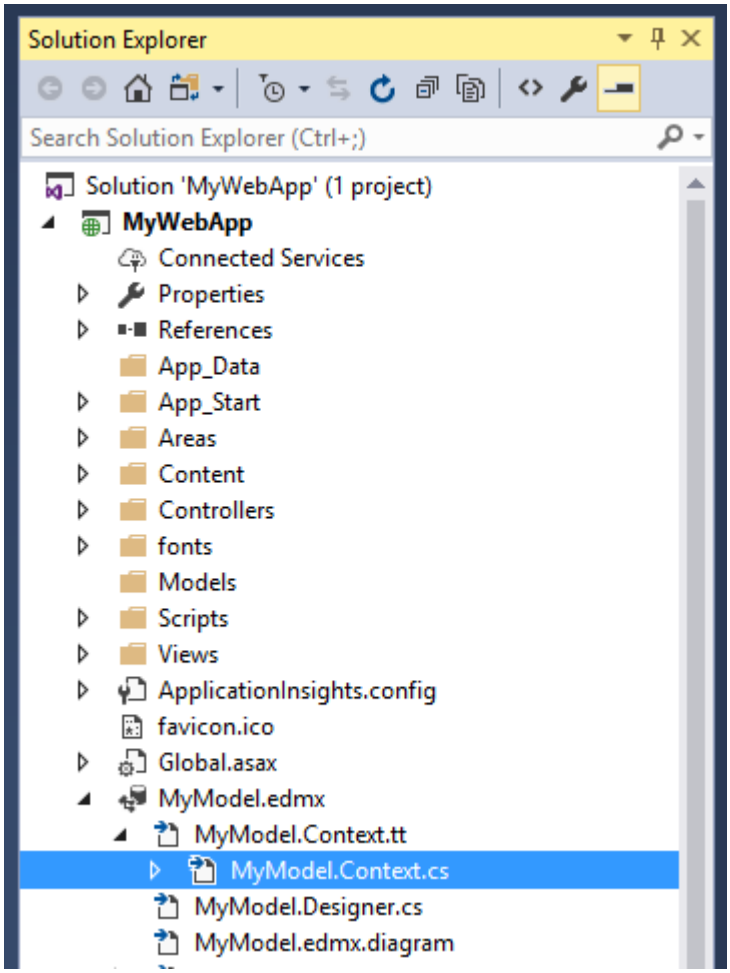


Replace the class code with the following.

```
using Microsoft.IdentityModel.Clients.ActiveDirectory;
using System.Configuration;
using System.IdentityModel.Tokens;
using System.Linq;
using System.Security.Claims;

namespace MyWebApp
{
    public static class TokenFactory
    {
        public static string GetToken()
        {
            ClientCredential clientCred = new ClientCredential(
                ConfigurationManager.AppSettings["ida:ClientId"],
                ConfigurationManager.AppSettings["ida:Password"]
            );

            BootstrapContext bootstrapContext =
                (BootstrapContext)ClaimsPrincipal.Current.Identities.First().BootstrapContext;

            string userAccessToken = bootstrapContext.Token;

            string userName =
                ClaimsPrincipal.Current.FindFirst(ClaimTypes.Upn) != null
                ? ClaimsPrincipal.Current.FindFirst(ClaimTypes.Upn).Value
                : ClaimsPrincipal.Current.FindFirst(ClaimTypes.Email).Value;

            UserAssertion userAssertion = new UserAssertion(
                userAccessToken,
                "urn:ietf:params:oauth:grant-type:jwt-bearer",
                userName
            );

            string authority = ConfigurationManager.AppSettings["ida:Authority"]
                + ConfigurationManager.AppSettings["ida:Tenant"];

            Microsoft.IdentityModel.Clients.ActiveDirectory.AuthenticationContext authContext =
                new Microsoft.IdentityModel.Clients.ActiveDirectory.AuthenticationContext(authority);

            AuthenticationResult result = authContext.AcquireTokenAsync(
                https://database.windows.net/,
                clientCred,
                userAssertion).Result;

            return result.AccessToken;
        }
    }
}
```

Override EntityConnection establishment in DbContext

Open the DbContext class generated by the model.



Replace the code with the following.

```
namespace MyWebApp
{
    using System.Configuration;
    using System.Data.Entity;
    using System.Data.Entity.Core.EntityClient;
    using System.Data.Entity.Infrastructure;
    using System.Data.SqlClient;

    public partial class MyModelContainer : DbContext
    {
        public MyModelContainer()
            : base(GetConnection(), true)
        {
        }

        private static EntityConnection GetConnection()
        {
            EntityConnection entConn = new EntityConnection(
                ConfigurationManager.ConnectionStrings["MyModelContainer"].ConnectionString
            );

            SqlConnection sqlConn = (SqlConnection)entConn.StoreConnection;

            sqlConn.AccessToken = TokenFactory.GetToken();

            entConn.Open();

            return entConn;
        }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            throw new UnintentionalCodeFirstException();
        }

        public virtual DbSet<Item> Items { get; set; }
    }
}
```

Note: make sure to change the connection string name to match Web.config.

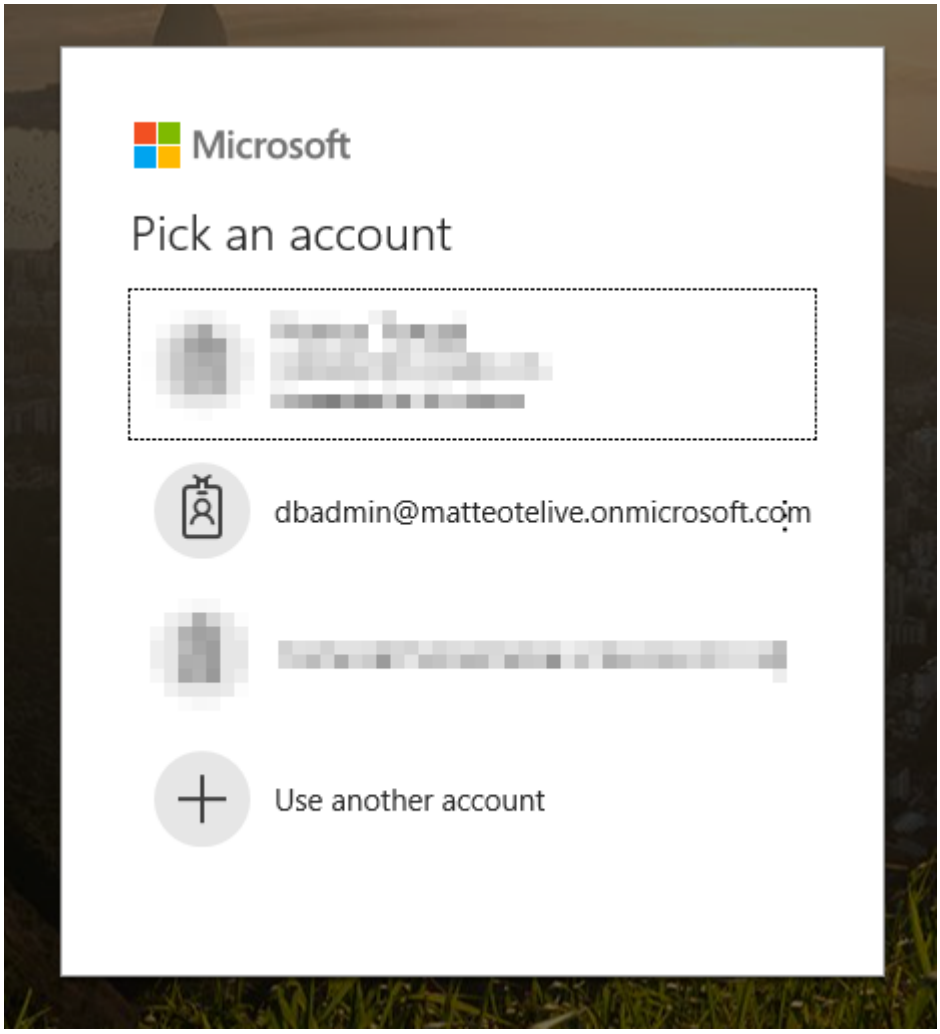
GetConnection method creates an EntityConnection based on the existing connection string and takes care of populating the AccessToken property of the underlying SqlConnection object.

GetConnection is used during the invocation of the base constructor.

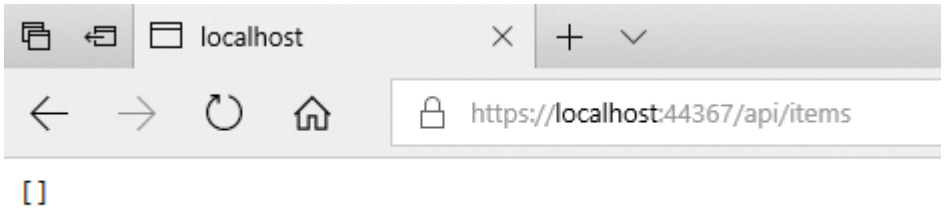
Test the application

Start the application and access the following URL: https://localhost:44367/api/items

Select an account and provide the credentials.



The result should be an empty list:



Classification

Root cause Tree - Connectivity/AAD Issue/Other Client Driver / Client Issue

How good have you found this content?