How to Setup a Project using Entity Framework with PostgreSQL

This tutorial shows how to create a simple console application which uses Entity Framework with a code first model and a PostgreSQL database. Code first means we setup our data model in code and let Entity Framework create the database for us. We are going to use Visual Studio 2015 with Update 3, Entity Framework 6, Npgsql 3.1 and PostgreSQL 9.4.

Create New Project

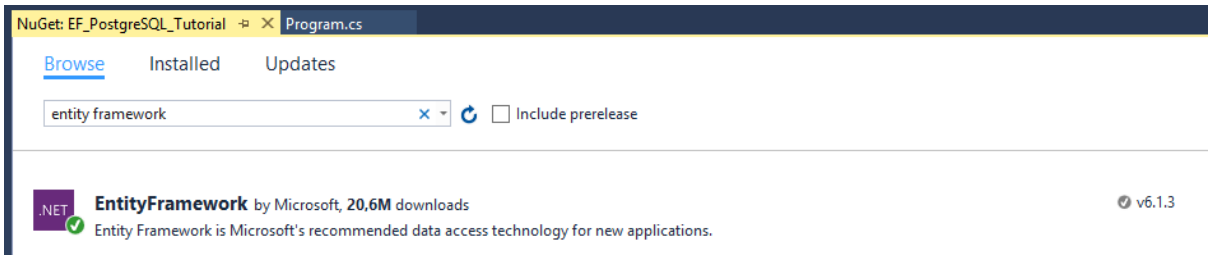
Start Visual Studio and create a new console application (File->New->Project. Select Console Application). Let's name it EF_PostgreSQL_Tutorial.

Download NuGet Packages

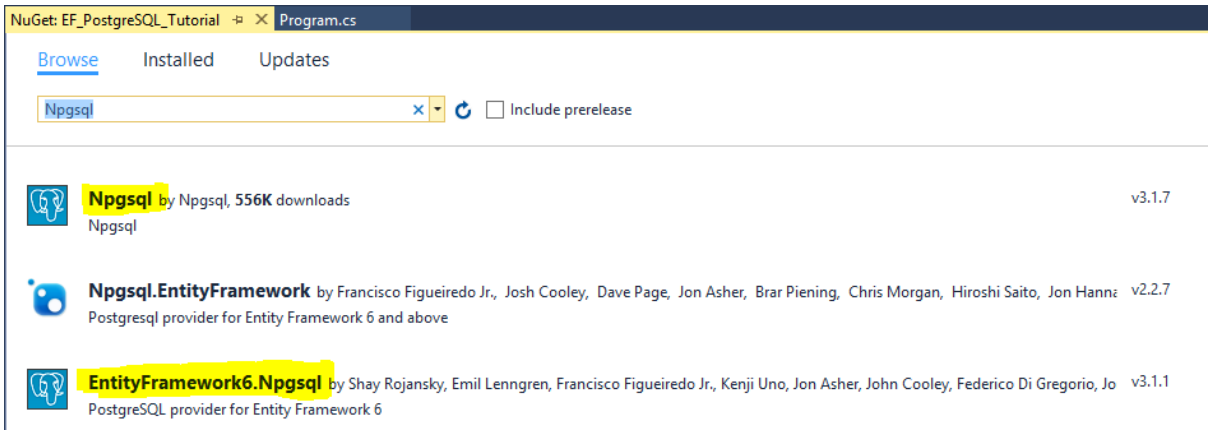
We need to download some packages for Entity Frame work and Npgsql. Npgsql is the provider which adds support for using PostgreSQL with Entity Framework.

In the Solution Explorer, right click on the EF_PostgreSQL_Tutorial and select "Manage NuGet Packages..."

Go to Browse and search for Entity Framework. We're using version 6.1.3 here. Select it and install.



Now search for Npgsql. Install the following **highlighted** packages. Start with Npgsql.



Setup Npgsql in App.config

Now we need to make some modifications to the App.config (web.config if you are building a web application) to hookup Npgsql to work with Entity Framework. Here is the complete App.config with the modifications highlighted.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit
http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
requirePermission="false" />
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.2" />
  </startup>
  <system.data>
    <DbProviderFactories>
      <remove invariant="Npgsql" />
      <add name="Npgsql - .Net Data Provider for PostgreSQL" invariant="Npgsql"
description=".Net Data Provider for PostgreSQL" type="Npgsql.NpgsqlFactory, Npgsql"
/>
    </DbProviderFactories>
  </system.data>
  <entityFramework>
    <!-- The default connection factory is only used when no connection string has
been added to the configuration file for a context. -->
    <defaultConnectionFactory type="Npgsql.NpgsqlConnectionFactory,
EntityFramework6.Npgsql" />
    <providers>
      <provider invariantName="System.Data.SqlClient"
type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer"
/>
      <provider invariantName="Npgsql" type="Npgsql.NpgsqlServices,
EntityFramework6.Npgsql" />
    </providers>
  </entityFramework>
  <connectionStrings>
    <add name="NpgsqlConnectionString"
         providerName="Npgsql"
         connectionString="User
ID=postgres;Password=postgres;Host=localhost;Port=5432;Database=EF_PostgreSQL_Tutor
ial;Pooling=true;"/>
  </connectionStrings>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="Npgsql" publicKeyToken="5d8b90d52f46fda7"
culture="neutral" />
        <bindingRedirect oldVersion="0.0.0.0-3.1.7.0" newVersion="3.1.7.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Here we:

- Added Npgsql to the DBProviderFactories.
- Changed the defaultConnectionFactory to be Npgsql. This is strictly not necessary in this tutorial as we are going to specify a connection string.
- Added a connection string. This assumes we have setup PostgreSQL locally with a super user named postgres and with password postgres. Note that the user needs to have full rights to be able to create databases. We are going to mention a few other details about the PostgreSQL installation and setup later.
- Note that we did not have to add Npgsql to the providers section. This was done automatically when we added the NuGet packages.

Setting up PostgreSQL

Now it is time to install and setup PostgreSQL. We're going to use version 9.4 for Windows and it can be downloaded here:

http://www.enterprisedb.com/products-services-training/pgdownload#windows

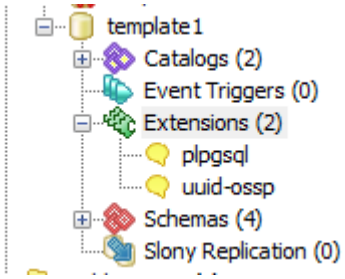
In the installation setup you will be asked to enter a password for the admin user postgres. For simplicity we're going to use postgres as the password too but if you want to change it make sure to also change in the connection string in the App.config.

Once installed you can open the PostgreSQL admin tool pgadmin III. To prepare and make it possible to use EF migrations later on we are going to add an extension to the template1 database.

Open File->Options and select "UI Miscellaneous" and then check "Show System Objects in the treeview". This will enable the Object browser to show the system or built-in databases like in this case the template1 database used as a template when creating a new database. In the Object browser you may need to right-click the Databases node and select Refresh for the template1 database to show up.

Expand the template1 database node in the Object browser, right-click on Extensions and select "New Extension…".

In the Name list, select the uuid_ossp extension and click Ok. It should appear in the list of installed extensions.



Now we can change Options to hide the system objects again if we like.

For a bit more information on why this step was necessary see here:

http://www.npgsql.org/doc/ef6.html

Creating our Data Model

As we are using the code first approach to creating the data model we will add a couple of classes describing the entities we want to store in the database. We will keep it simple and start by adding an Employee class.

```
using System;
using System.ComponentModel.DataAnnotations;

namespace EF_PostgreSQL_Tutorial
{
    public class Employee
    {
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }
        [Required]
        public Company Company { get; set; }
        public DateTime HireDate { get; set; }
    }
}
```

The Employee class has four simple properties and two of them have been explicitly marked as required by the RequiredAttribute because they should not be allowed to be null. In this example we also rely on the EF code first conventions so the first property, Id, will be identified and used as the primary key. You can customize all of these conventions if you like.

More info on the conventions and data annotations via attributes can be found here:

https://msdn.microsoft.com/en-us/data/jj679962

https://msdn.microsoft.com/en-us/data/jj591583.aspx

Now let's add the Company class.

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace EF_PostgreSQL_Tutorial
{
    public class Company
    {
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }
        public int YearFounded { get; set; }

        public virtual ICollection<Employee> Employees { get; set; }
    }
}
```

Nothing too exciting here but you might wonder why the Employees property is virtual. It is to enable "lazy loading" of the Employee data which means when a Company entity is loaded from the database the Employee collection will not be loaded until you actually try to use it. More details can be found here:

https://msdn.microsoft.com/en-us/data/jj574232.aspx

Creating the Database Context Class

It is time to create the class which is the main interface to the database. We will call it CompanyContext and it will inherit from the EF class DbContext.

```
using System.Data.Entity;

namespace EF_PostgreSQL_Tutorial
{
    public class CompanyContext : DbContext
    {
        public virtual DbSet<Company> CompanySet { get; set; }
        public virtual DbSet<Employee> EmployeeSet { get; set; }

        public CompanyContext() : base("NpgsqlConnectionString")
        {

        }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.HasDefaultSchema("public");
            base.OnModelCreating(modelBuilder);
        }
    }
}
```

The class contains two DbSet properties for Company and Employee data. The purpose of having these virtual is not lazy loading like for the Employees property in the Company class. Instead this enables these properties to be mocked by a test framework. More info on that here:

https://msdn.microsoft.com/en-us/library/dn314429.aspx

The constructor passes the name of the connection string (see App.config) to the DbContext base constructor.

We also have to override OnModelCreating and specify the default schema as this is called public in PostgreSQL. If we don't do this EF will use dbo which is default schema for SQLServer.

Putting It Together

Now we have everything we need to actually do something interesting. We are going to add couple of companies and a few employees to the database and then simply read them back and print them to the console. So here's our Program.cs.

```
using System;
using System.Linq;

namespace EF_PostgreSQL_Tutorial
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var context = new CompanyContext())
            {
                // Add a couple of companies and some employees.
                var company = new Company()
                {
                    Name = "MegaCorp",
                    YearFounded = 2000
                };
                context.CompanySet.Add(company);

                context.EmployeeSet.Add(new Employee()
                {
                    Name = "Andrew",
                    Company = company,
                    HireDate = new DateTime(2000, 01, 01)
                });
                context.EmployeeSet.Add(new Employee()
                {
                    Name = "John",
                    Company = company,
                    HireDate = new DateTime(2001, 10, 05)
                });

                company = new Company()
                {
                    Name = "EvilInc",
                    YearFounded = 1996
                };
                context.CompanySet.Add(company);

                context.EmployeeSet.Add(new Employee()
                {
                    Name = "Dr Evil",
                    Company = company,
                    HireDate = new DateTime(1996, 03, 20)
                });
                context.EmployeeSet.Add(new Employee()
                {
                    Name = "Minion",
                    Company = company,
                    HireDate = new DateTime(1996, 04, 07)
                });

                // Save/Commit changes to database.
                context.SaveChanges();

                // Print out the companies and their employees to the console.
                var companies = context.CompanySet.ToList();
                foreach (var c in companies)
                {
                    Console.WriteLine("Company " + c.Name + " founded in " + c.YearFounded);
                    foreach (var employee in c.Employees)
                    {
                        Console.WriteLine("  Name: " + employee.Name + " Hire Date: " +
employee.HireDate);
                    }
                }
                Console.ReadLine();
            }
        }
    }
}
```

On the first run of the program EF will create the database and tables and then populate it with the data. Note that if you run the example multiple times it will add the same companies and employees multiple times as we have no checks to see if they already exist in the database.

Further note that when listing the companies and employees we first get a list of companies instead of directly iterating over context.CompanySet in the first foreach loop. This is to work around a limitation in Npgsql. It is likely this issue (MARS) discussed on the following page:

https://github.com/npgsql/npgsql/issues/826

So now that we have a working example let's look a bit at migrations.

Migrations

Migrations can be used to update the database when our data model changes. We can also use migrations to generate SQL scripts to build up the entire database.

We're going to keep this short and but you can find much more details here:

https://msdn.microsoft.com/en-us/data/jj591621.aspx

First we need to enable migrations. Open View->Other Windows->Package Manager Console.

Run the command "Enable-Migrations". This will add a new folder called Migrations to your project with a couple of files. Configuration.cs can be used to configure how migrations work as well as seed your database with initial data and the other file is the initial migration that creates the database tables.

Now let's add a new property to the Employee class:

```
        public string Department { get; set; }
```

To add a migration which will add this property as a corresponding column in the database we need to create/add a new migration. This is done via the Add-Migration command. So let's run the following command in the Package Manager Console.

```
Add-Migration AddDepartmentToEmployee
```

You will notice that a new file was added to the Migrations folder with a time stamp and the name you specified. The database has not been updated yet so this is the next step. To actually apply the migration we use the Update-Database command. Just running it without any arguments is enough in this case to apply the pending migrations which in our case is the one we just added. So run:

```
Update-Database
```

If you check the Employee table now in pgadmin III you will notice a new column "Department" was added.

Generating SQL Scripts

It is often desirable to generate an SQL script to setup the database so this can be used by a database admin to setup a production database for example. We can use the migrations functionality for this purpose as well.

To generate a script that describes the complete setup needed from first to last migration you can use the following command.

```
Update-Database -Script -SourceMigration: $InitialDatabase
```

However there is a bug in Npgsql which results in an exception:

```
System.NullReferenceException: Object reference not set to an instance of an object.
```

One workaround to this is to first revert back to the initial state of the database, then generate the script and then update back to the latest migration. Please note though that this means your data will be lost so only do it against a local development database. Here are the commands to run.

```
Update-Database -TargetMigration $InitialDatabase

Update-Database -Script

Update-Database
```

Miscellaneous Notes

This is a bit of extra info and lessons learned which might be good to know.

Entity Framework Power Tools

The EF Power Tools have some useful functionality and especially the piece that can generate a visual representation of how EF has interpreted your code first model. Unfortunately it does not seem to work with Npgsql and this may be due to the fact that the Npgsql DDEX Visual Studio integration has not been updated to work with Npgsql version 3.1 yet.

More info on EF Power Tools here:

https://msdn.microsoft.com/en-us/data/jj593170.aspx

Here's a description of how you get them into VS 2015:

http://thedatafarm.com/data-access/installing-ef-power-tools-into-vs2015/

The missing VS integration support is tracked here:

https://github.com/npgsql/npgsql/issues/1140?_pjax=%23js-repo-pjax-container

Model First with the Visual Designer

This tutorial is using the code first approach to describe the data model but there are other ways of doing it as well. One is database first where you manually create the database the way you want it and another is to use model first and create the model visually in the EF designer. Currently this approach does not seem to work with Npgsql and the reason is probably the same as describe above which is the missing VS integrations support. However note that the model first approach does not support migrations which means it might not be what you want to use anyway.