NANGARHAR UNIVERSITY

VISUAL PROGRAMMING

ENTITY FRAMEWORK

SHAKIRULLAH WASEEB
COMPUTER SCIENCE FACULTY
Software Engineering Department

Contents

Database Connectivity

The ADO.NET (Active Data Object) has enabled .NET programmers to work with relational data (in a relatively straightforward manner) since the initial release of the .NET platform. However, Microsoft introduced a new component of the ADO.NET API called the *Entity Framework* (or simply, *EF*) in .NET 3.5 Service Pack 1. We will be studying 6.4.0 version of EF for the full .NET Framework, which is packed full of features and performance enhancements over earlier versions.

Entity Framework

The overarching goal of EF is to allow us to interact with data from relational databases using an object model that maps directly to the business objects (or domain objects) in our application. For example, rather than treating a batch of data as a collection of rows and columns, we can operate on a collection of strongly typed objects termed *entities*. These entities are also natively LINQ aware, and you can query against them using the same LINQ grammar we learned in our previous lectures. The EF runtime engine translates the LINQ queries into proper SQL queries on our behalf.

In this tutorial we will introduce data access using the Entity Framework. We will learn about creating a domain model, mapping model classes to the database, and the role of the DbContext class.

Using EF, you can interact with a relational database without ever seeing a line of SQL code (if you so choose). Rather, when you apply LINQ queries to your strongly typed classes, the EF runtime generates proper SQL statements on our behalf.

Entities

Entities are a conceptual model of a physical database that maps to your business domain. Formally speaking, this model is termed an *entity data model* (EDM). The EDM is a client-side set of classes that are mapped to a physical database by Entity Framework convention and configuration.

For example, take the simple ***Item*** table in the Point of Sale (POS) database and the ***Product*** model class. Through model configuration, we inform EF that the Item model represents the Product table. This loose coupling means you can shape the entities so they closely model your business domain.

We will build a full example with EF in just a bit. However, for the time being, consider the following Program class, which uses the ***Product*** model class and a related context class (covered soon) named PosDbContext to add a new row to the Item table of PosDb.

```
class Program
{
   static void Main(string[] args)
     {
       // Connection string automatically read from config file.
       using (PosDbContext  context = new PosDbContext  ())
         {
           // Add a new record to Items table, using our model.
          context. Products.Add(new Product() { Name = "Nestle Nectar", Type = "Juice", Unit = "ml" , UnitValue=”1.5” });
          context.SaveChanges();
        }
    }
}
```

EF examines the configuration of the models and the context class to take the client-side representation of the Item table (here, a class named Product) and map it back to the correct columns of the Item table. Notice that we see no trace of any sort of SQL INSERT statement. We simply add a new Product object to the collection maintained by the aptly named Item property of the context object and then save your changes. There is no magic in the preceding example. Under the covers, a connection to the database is made and opened, a proper SQL statement is generated and executed, and the connection is released and closed. These details are handled on your behalf by the framework. Now let’s look at the core services of EF that make this possible.

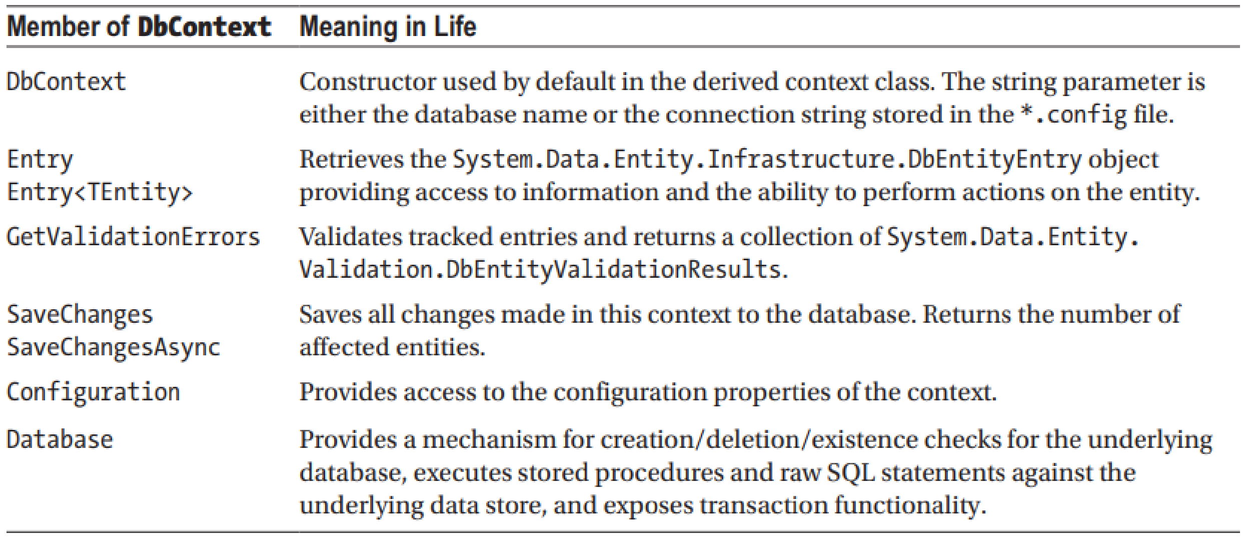
The Building Blocks of the Entity Framework

EF (at its core) the ADO.NET infrastructure and therefore like any ADO.NET interaction, EF uses an ADO.NET data provider to communicate with the data store. However, the data provider must be updated so it supports a new set of services before it can interact with the EF API. As we might expect, the Microsoft SQL Server data provider has been updated with the necessary infrastructure, which is accounted for when using the System.Data.Entity.dll assembly.

In addition to adding the necessary bits to the Microsoft SQL Server data provider, the System.Data.Entity.dll assembly contains various namespaces that account for the EF services themselves. The first key piece of the EF API to concentrate on is the DbContext class. You will create a derived, model-specific context when you use EF for your data access libraries.

The DbContext Class

The DbContext class represents a combination of the Unit of Work and Repository arrangements that can be used to query from a database and group together changes that will be written back as a single unit of work. DbContext provides a number of core services to child classes, including the ability to save all changes (which results in a database update), get the connection string, delete objects, call stored procedures, and handle other fundamental details. Following table shows some of the more commonly used members of the DbContext.



Member of **DbContext**	Meaning in Life
`DbContext`	Constructor used by default in the derived context class. The string parameter is either the database name or the connection string stored in the `*.config` file.

`Entry`
`Entry<TEntity>`
	Retrieves the `System.Data.Entity.Infrastructure.DbEntityEntry` object providing access to information and the ability to perform actions on the entity.

`GetValidationErrors`	Validates tracked entries and returns a collection of `System.Data.Entity.Validation.DbEntityValidationResults`.
`SaveChanges` `SaveChangesAsync`	Saves all changes made in this context to the database. Returns the number of affected entities.
`Configuration`	Provides access to the configuration properties of the context.
`Database`	Provides a mechanism for creation/deletion/existence checks for the underlying database, executes stored procedures and raw SQL statements against the underlying data store, and exposes transaction functionality.

The Derived Context Class

As mentioned, the DbContext class provides the core functionality when working with EF Code First. In our POS project, we will create a class that derives from DbContext for our Point of Sale domain. In the constructor, we need to pass the name of the connection string for this context class to the base class, as shown here:

```
public class PosDbContext  : DbContext
{
      public PosDbContext  () : base("name= PosDbConnection"){}
      protected override void Dispose(bool disposing){}
}
```

The DbSet<T>

To add tables into our context, we add a DbSet<T> for each table in our object model. To enable lazy loading, the properties in the context need to be virtual, like this:

```
public virtual DbSet<Product> Products { get; set; }
public virtual DbSet<Category> Categories { get; set; }
public virtual DbSet<Stock> Stock { get; set; }
public virtual DbSet<Supplier> Suppliers { get; set; }
public virtual DbSet<Purchase> Purchases { get; set; }
public virtual DbSet<PurchaseDetail> PurchaseDetails { get; set; }
public virtual DbSet<Customer> Customers { get; set; }
public virtual DbSet<Order> Orders { get; set; }
public virtual DbSet<OrderDetail> OrderDetails { get; set; }
```

Each DbSet<T> provides a number of core services to each collection, such as creating, deleting, and finding records in the represented table. The following table describes some of the core members of the DbSet<T> class.

Member of **DbSet<T>**	Meaning in Life
Add AddRange	Allows you to insert a new object (or range of objects) into the collection. They will be marked with the Added state and will be inserted into the database when SaveChanges (or SaveChangesAsync) is called on the DbContext.
Attach	Associates an object with the DbContext. This is commonly used in disconnected applications like ASP.NET/MVC.
Create Create<T>	Creates a new instance of the specified entity type.
Find FindAsync	Finds a data row by the primary key and returns an object representing that row.
Remove RemoveRange	Marks an object (or range of objects) for deletion.
SqlQuery	Creates a raw SQL query that will return entities in this set.

Once we drill into the correct property of the object context, we can call any member of DbSet<T>. Consider again the sample code shown in the first few pages of this tutorial:

```
using (PosDbContext  context = new PosDbContext  ())
 {
   // Add a new record to Inventory table, using our model.
  context.Products.Add(new Product() { Name = "Nestle Nectar", Type = "Juice", Unit = "ml" , UnitValue="1.5" });
  context.SaveChanges();
 }
```

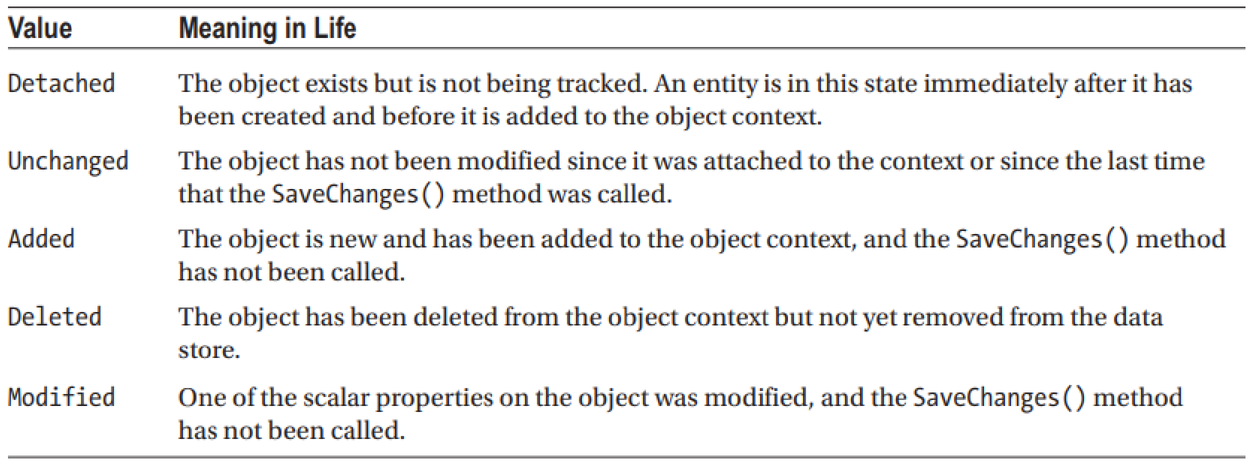
Here, PosDbContext *is-a* derived DbContext. The Products property gives us access to the DbSet<Product> variable. We use this reference to insert a new Product entity object and tell the DbContext to save all changes to the database.
DbSet<T> is typically the target of LINQ to Entity queries; as such, DbSet<T> supports the same extension methods we learned in our last lectures (collections, Lambdas etc.), such as ForEach(), Select(), and All(). Moreover, DbSet<T> gains a good deal of functionality from its direct parent class, DbQuery<T>, which is a class that represents a strongly typed LINQ (or Entity SQL) query.

Code First Approach

As mentioned in a previous note, Code First doesn't mean you can't use EF with an existing database. It really just means no EDMX model. We can use Code First from an existing database or create a new database from the entities using EF migrations.

Entity State and Change Tracking

The DbChangeTracker automatically tracks the state for any object loaded into a DbSet<T> within a DbContext. In the previous examples, while inside the using statement, any changes to the data will be tracked and saved when SaveChanges is called on the PosDbContext class. The following table lists the possible values for the state of an object.



Value	Meaning in Life
`Detached`	The object exists but is not being tracked. An entity is in this state immediately after it has been created and before it is added to the object context.
`Unchanged`	The object has not been modified since it was attached to the context or since the last time that the `SaveChanges()` method was called.
`Added`	The object is new and has been added to the object context, and the `SaveChanges()` method has not been called.
`Deleted`	The object has been deleted from the object context but not yet removed from the data store.
`Modified`	One of the scalar properties on the object was modified, and the `SaveChanges()` method has not been called.

If we need to check the state of an object, use the following code:

```
EntityState state = context.Entry(entity).State;
```

We usually don't need to worry about the state of your objects. However, in the case of deleting an object, we can set the state of an object to EntityState.Deleted and save a round-trip to the database. We will do this in coming sections.

Entity Framework Data Annotations

Data annotations are C# attributes that are used to shape your entities. The following table lists some of the most commonly used data annotations for defining how your entity classes and properties map to database tables and fields. There are many more annotations that can be used to refine the model and add validations.

Data Annotation	Meaning in Life
`Key`	Defines the primary key for the model. This is not necessary if the key property is named `Id` or combines the class name with `Id`, such as `OrderId`. If the key is a composite, you must add the `Column` attribute with an `Order`, such as `Column[Order=1]` and `Column[Order=2]`. Key fields are implicitly also `[Required]`.
`Required`	Declares the property as not nullable.
`ForeignKey`	Declares a property that is used as the foreign key for a navigation property.
`StringLength`	Specifies the min and max lengths for a string property.
`NotMapped`	Declares a property that is not mapped to a database field.
`ConcurrencyCheck`	Flags a field to be used in concurrency checking when the database server does updates, inserts, or deletes.
`TimeStamp`	Declares a type as a row version or timestamp (depending on the database provider).
`Table` `Column`	Allows you to name your model classes and fields differently than how they are declared in the database. The `Table` attribute allows specification of the schema as well (as long as the data store supports schemas).
`DatabaseGenerated`	Specifies if the field is database generated. This takes one of `Computed`, `Identity`, or `None`.
`NotMapped`	Specifies that EF needs to ignore this property in regard to database fields.
`Index`	Specifies that a column should have an index created for it. You can specify clustered, unique, name, and order.

A Driven Example

Create a console application (e.g. EFDemo) using visual studio. Once you created the console project follow these steps:

Adding EntityFramework Package

Download the Package from the link in the Google Class Room

Browse to the path: C:\Program Files (x86)\Microsoft SDKs\NuGetPackages

Paste the downloaded package to given folder NuGetPackage.

Right Click (1) on the Project (**EFDemo**) from the visual studio solution folder, click (2) on **Manage NuGet Packages.**

Figure 2 will be appeared in the visual studio working place. Now click on the settings icon (1). Figure 3 will be appeared.

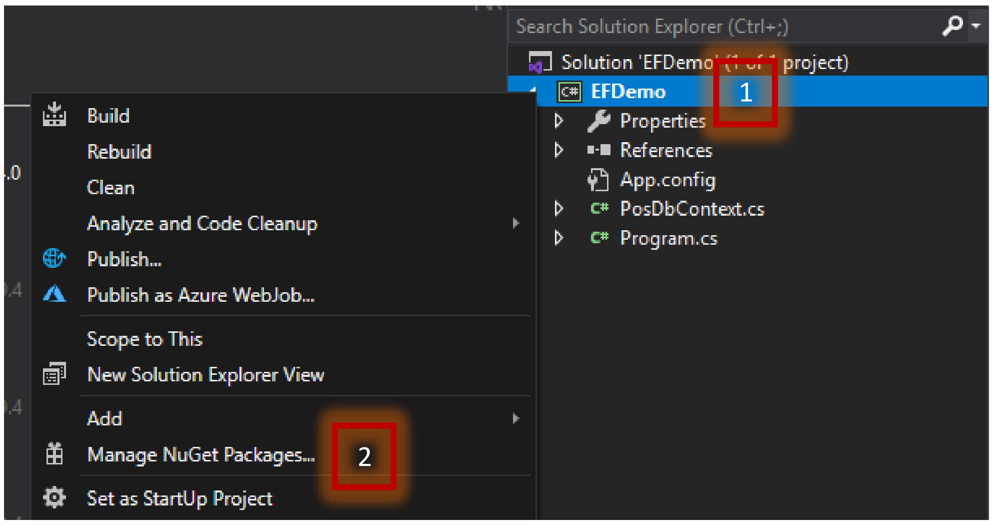


Figure 1 NuGet Package Manager

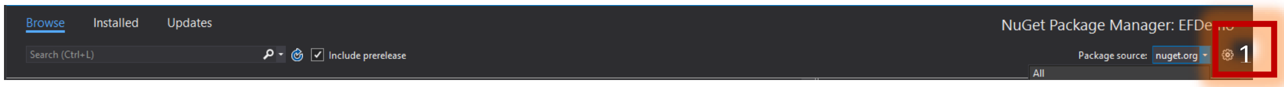
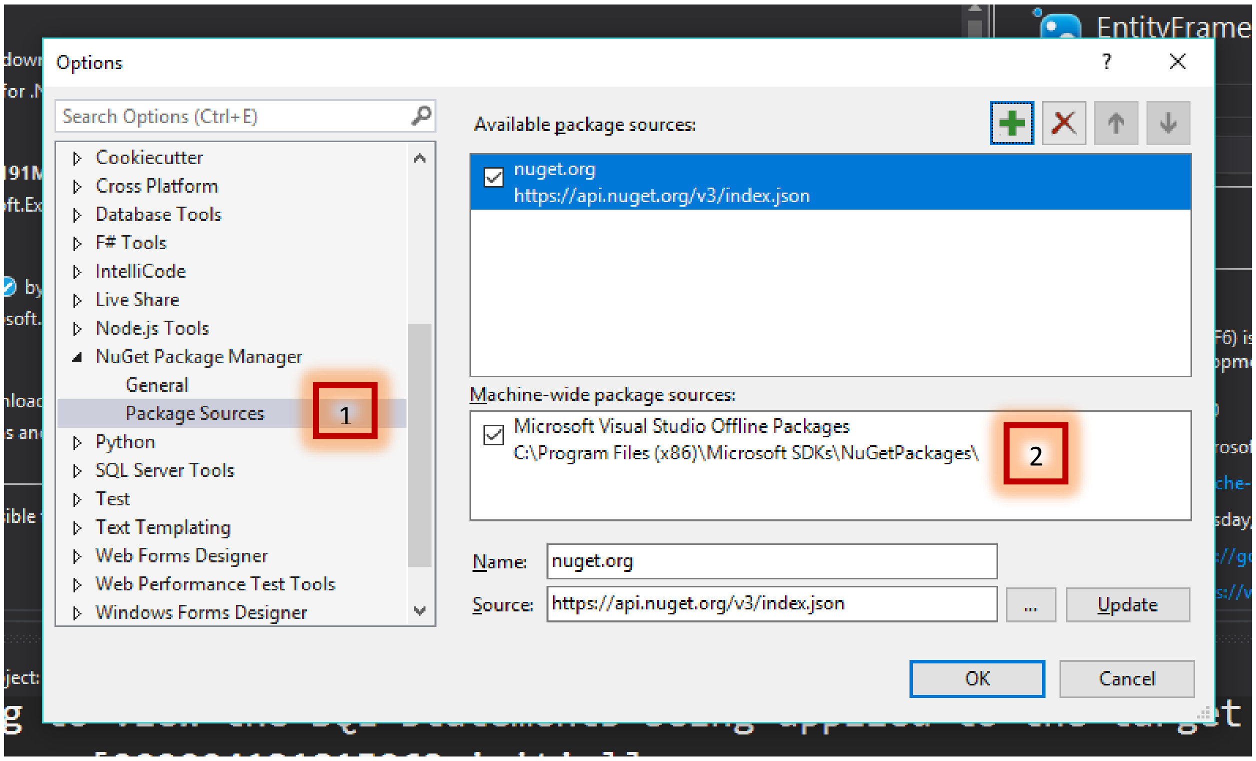


Figure 2 NuGet Package Manager: EFDemo

Figure 3 provides the view to manage the NuGet packages. As we don't have access to internet always so we will prefer the offline package source. For this reason, click on NuGet Package Manager to expand and then select the Package Sources (1), and on the right side of the figure "Machine-wide package sources:" check the Microsoft Visual Studio Offline Package if unchecked. Finally click on **OK** button to finish.



Available package sources:

☑ nuget.org
https://api.nuget.org/v3/index.json

Machine-wide package sources:

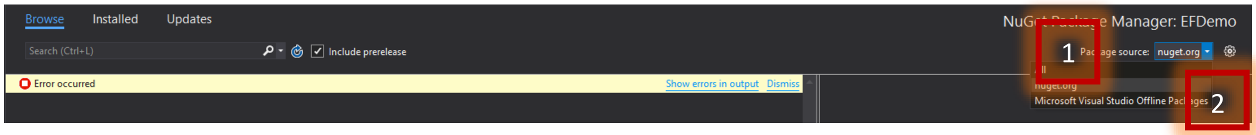
☑ Microsoft Visual Studio Offline Packages
C:\Program Files (x86)\Microsoft SDKs\NuGetPackages\

Name: nuget.org

Source: https://api.nuget.org/v3/index.json

Figure 3 Microsoft Visual Studio Offline Packages

In Figure 4 Click (1) on Package source drop down and then select (2) Microsoft Visual Studio Offline Packages or select the All option. You will see the changes as in



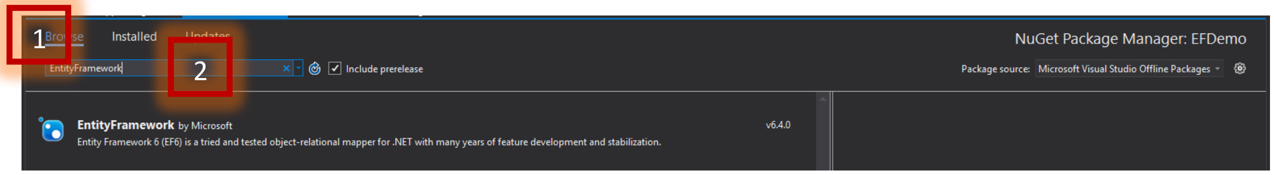
Package source: nuget.org

Error occurred

All
nuget.org
Microsoft Visual Studio Offline Packages

Figure 4 Offline Packages Setting

Be sure that Browse (1) tab is selected, all the offline packages will be listed here, if the list goes longer used the search text box (2) type the EntityFramework to filter the required package. See Figure 5.



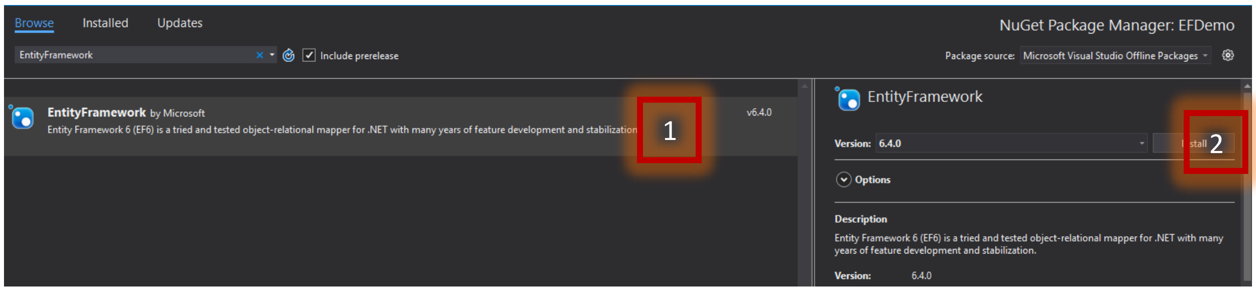
EntityFramework

Package source: Microsoft Visual Studio Offline Packages

EntityFramework by Microsoft v6.4.0
Entity Framework 6 (EF6) is a tried and tested object-relational mapper for .NET with many years of feature development and stabilization.

Figure 5 Selecting Entity Framework Package

Click/Select (1) on the target package “EntityFramework” and then on the right side of the figure click (2) on the Install button to add this package to the project. See Figure 6, subsequently click (1) on OK for accepting the changes to the project, and click (2) on I Accept button to accept the license agreement as show in Figure 7.

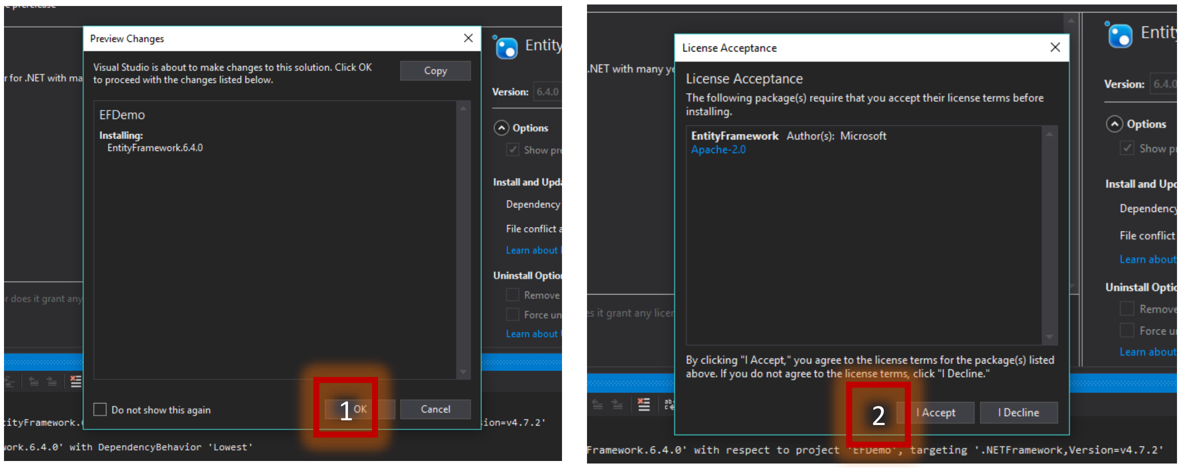

Browse
Installed
Updates
NuGet Package Manager: EFDemo
EntityFramework
Include prerelease
Package source: Microsoft Visual Studio Offline Packages
1
v6.4.0
2
Options

EntityFramework by Microsoft
Entity Framework 6 (EF6) is a tried and tested object-relational mapper for .NET with many years of feature development and stabilization

EntityFramework
Version: 6.4.0

Description
Entity Framework 6 (EF6) is a tried and tested object-relational mapper for .NET with many years of feature development and stabilization.
Version: 6.4.0

Figure 6 Installing Entity Framework Package



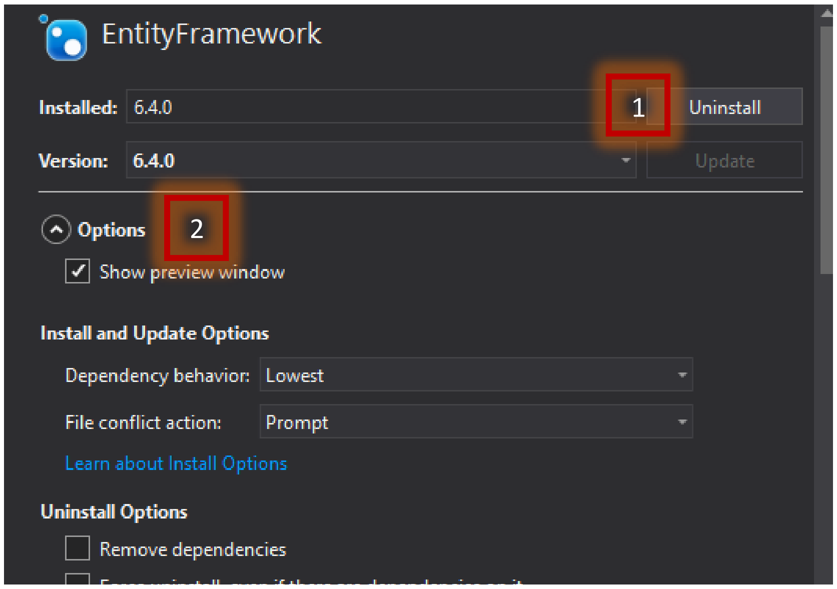
Preview Changes
Visual Studio is about to make changes to this solution. Click OK to proceed with the changes listed below.
EFDemo
Installing:
EntityFramework.6.4.0
Do not show this again

License Acceptance
License Acceptance
The following package(s) require that you accept their license terms before installing.
EntityFramework Author(s): Microsoft
Apache-2.0
By clicking "I Accept," you agree to the license terms for the package(s) listed above. If you do not agree to the license terms, click "I Decline."

```
Framework.6.4.0' with respect to project 'EFDemo', targeting '.NETFramework,Version=v4.7.2'
```

Figure 7 Changes, and License Agreement

Once the package is installed you will see Figure 8, the installed button will be labeled (1) with Uninstall, and if you expand (2) Options you will see the Install, Update, and Uninstall Options.



EntityFramework
Installed: 6.4.0
Version: 6.4.0
Show preview window ✓

Install and Update Options
Dependency behavior: Lowest
File conflict action: Prompt

Uninstall Options
Remove dependencies

Figure 8 Installed Packages View

That's It we are ready to use Entity Framework in our project. Now we will create a class derived from DbContext, then we will define a model as entity and will add this as the DbSet generic properties of the context class, and final perform the Migrations to map our code core database schema in the relational database server or DBMS. We will be using Microsoft SQL server as our DBMS.

Creating Database Context Class:

Before creating the context class first add the following connection string to configuration section of the App.config file.

Note: copy only the connectionStrings section the configuration tag and the dots lines are given just for explanation of where to put it.

```
<configuration>
………………………………………
………………………………………
<connectionStrings>
    <add name="EfDbConnection"
      connectionString="data source=.;initial catalog=EfDb;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
      providerName="System.Data.SqlClient"/>
  </connectionStrings>
………………………………………
………………………………………
</configuration>
```

Create a class as EfDbContext having the following code.

```
using System.Data.Entity;

namespace EFDemo
{
    class EfDbContext : DbContext
    {
        public EfDbContext(): base("name:EfDbConnection") { }
    }
}
```

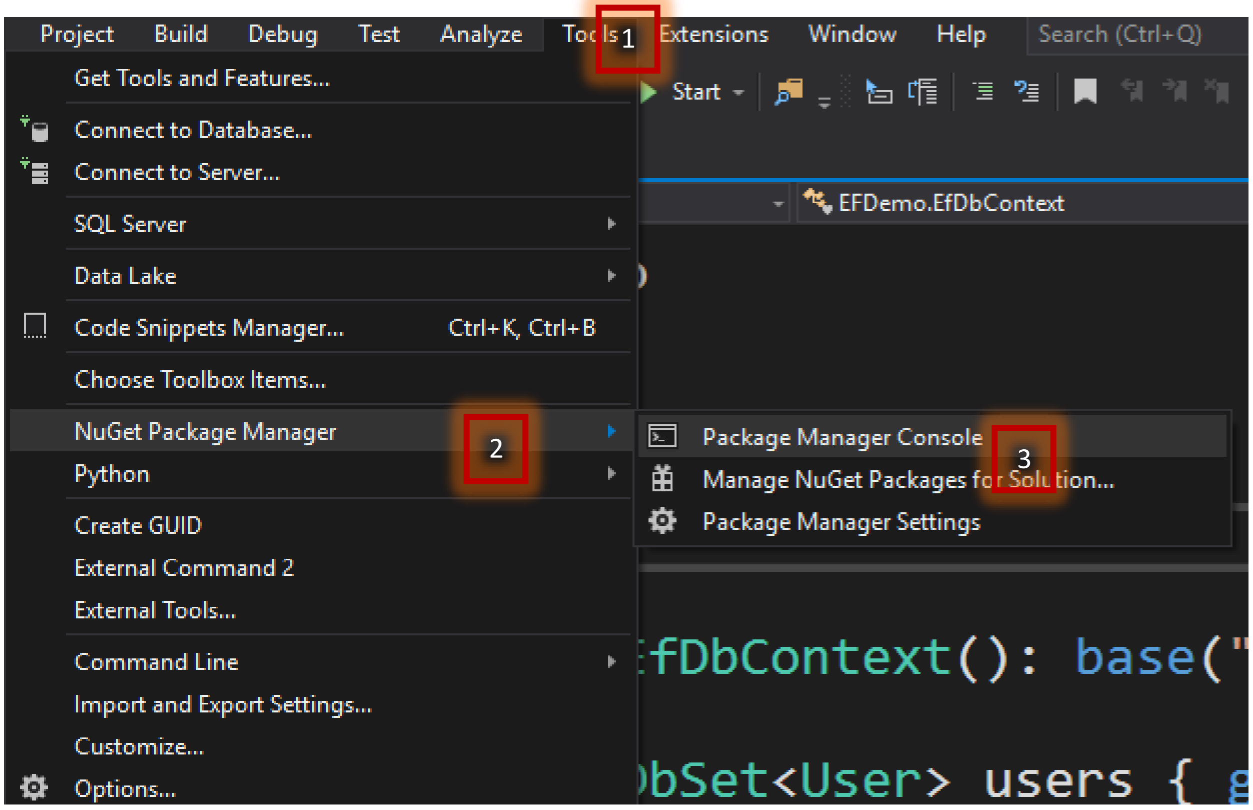
Once we created the EfDbContext class, it's the time to create a model/entity and attach it to the EfDbContext class using the DbSet<> generic property of the EfDbContext class. Let us create a User class as:

```
using System.ComponentModel.DataAnnotations;
namespace EFDemo
{
    public class User
    {
        [Key]
        public int Id { get; set; }
        public string Name { get; set; }
    }
}
```

Update the EfDbContext as:

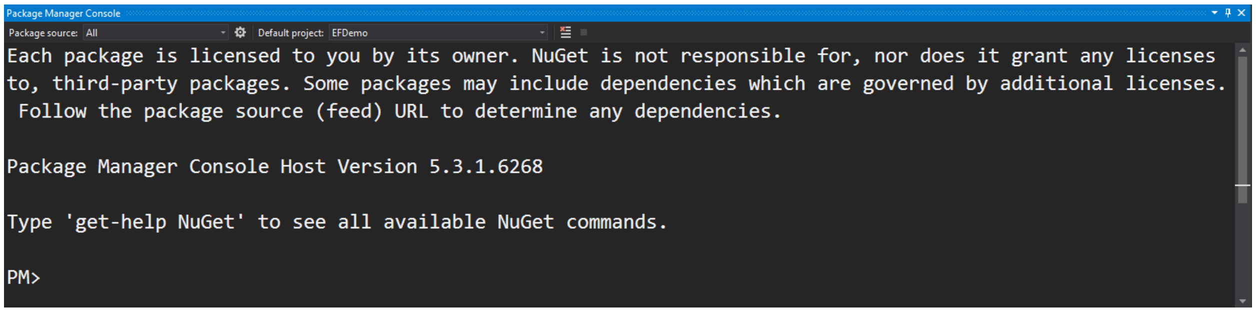
```
    class EfDbContext : DbContext
    {
        public EfDbContext(): base("name:EfDbConnection") { }
        public DbSet<User> users { get; set; }
    }
```

Now it's time for migrating the code schema to DMBS relational schema. For this follow the following steps:
Click on Tools Menu then Point to NuGet Package Manager, then click on Package Manager Console to open the console. You will see the view in the bottom of the visual studio workplace area as shown in Figure 10.
Menue > NuGet Package Manager > Package Manager Console



```
EfDbContext(): base("
DbSet<User> users { g
```

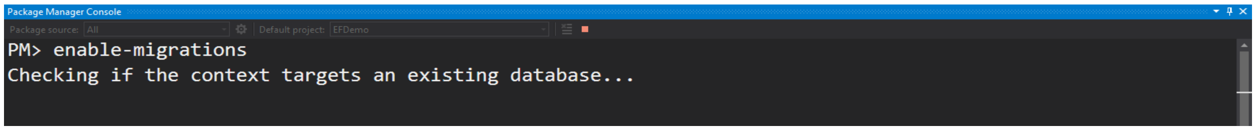
Figure 9 Using Package Manager Console for Database Migrations



```
Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses
to, third-party packages. Some packages may include dependencies which are governed by additional licenses.
 Follow the package source (feed) URL to determine any dependencies.

Package Manager Console Host Version 5.3.1.6268

Type 'get-help NuGet' to see all available NuGet commands.

PM>
```

Figure 10 Package Manager Console

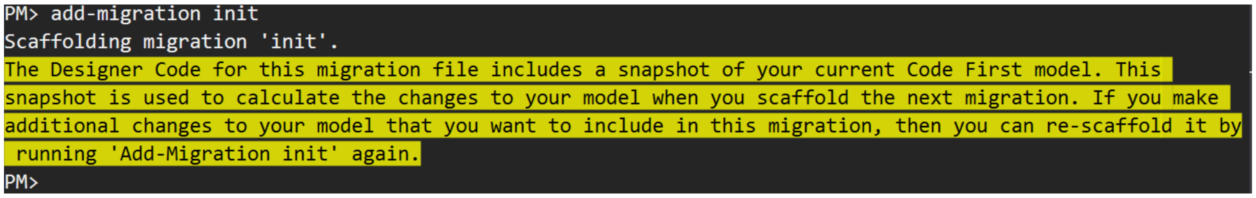
The first step is to enable the migration and add the migrations configuration to the project.



```
PM> enable-migrations
Checking if the context targets an existing database...
```

Figure 11 Enabling Database Migrations

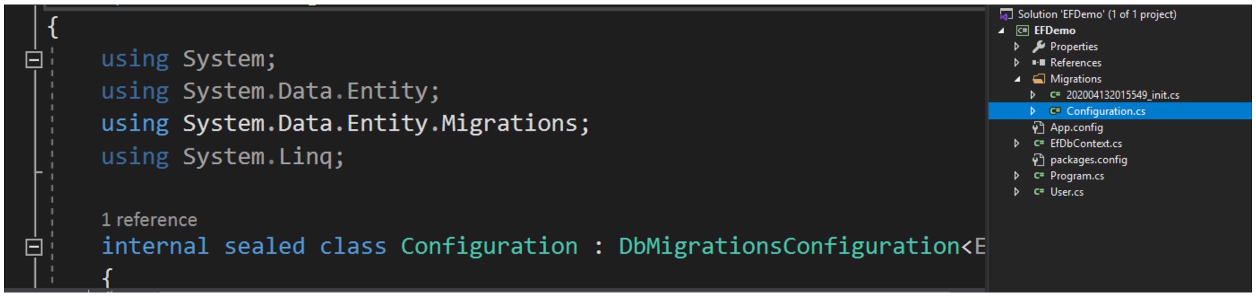
Every time we bring any changes to the entities/schemas on the code side we need to add a new migration as we are migrating for the first time so named this migration to init as show in Figure 12.



```
PM> add-migration init
Scaffolding migration 'init'.
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 init' again.
PM>
```

Figure 12 Adding Migrations

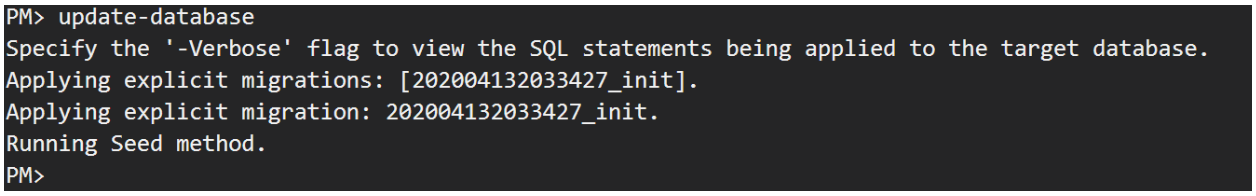
After we enabled migrations and added our very first migration these commands will add a new folder named Migrations and the project directory will look like Figure 13Figure 12.



```
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;

    1 reference
    internal sealed class Configuration : DbMigrationsConfiguration<E
    {
```

Figure 13 Project Directory Structure after Migrations

Once we have migrations in hand it’s the time to reflect the changes in the DBMS relational schema, we do this by using update-database command as show in Figure 14.



```
PM> update-database
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Applying explicit migrations: [202004132033427_init].
Applying explicit migration: 202004132033427_init.
Running Seed method.
PM>
```

Figure 14 Updating the DBMS schema with changes in the Code