Entity Framework

Windows Programming Course

Agenda

1. Overview Entity Framework
2. Creating a model
3. Querying data
4. Saving data
5. Using Transactions
6. Migration

1.

Overview Entity Framework

ADO.NET

Prior to .NET 3.5, Using to write **ADO.NET** code to save or retrieve application data from the underlying database:

◎ Open a connection to the database

◎ Create a DataSet to fetch

◎ Submit the data to the database

◎ Convert data from the DataSet to .NET objects.

Entity Framework (EF)

Entity Framework is an open-source **ORM** (Object Relationship Mapping) frameworks offering mapping of entities to relationships:

◎ Create types that map to database tables

◎ Create database queries using LINQ

◎ Create and update objects.
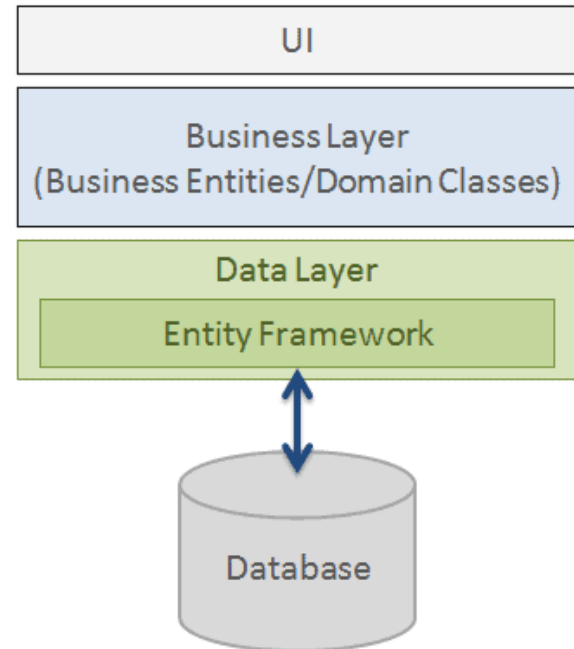
Entity Framework Features

Modelling: EF creates an EDM (Entity Data Model) based on POCO (Plain Old CLR Object) entities with get/set properties of different data types.

Querying: Use LINQ queries to retrieve data from the underlying database or execute raw SQL queries directly to the database.

Change Tracking: Keep track of changes occurred to instances of entities (Property values) which need to be submitted to the database.

Entity Framework Features (cont.)

Saving: Execute INSERT, UPDATE, and DELETE commands to the database based on the changes occurred to entities when calling the SaveChanges() method.

Concurrency: Use Optimistic Concurrency by default to protect overwriting changes made by another user since data was fetched from the database.

Transactions: Performs automatic transaction management while querying or saving data.

Entity Framework Features (cont.)

Caching: Include first level of caching out of the box. So, repeated querying will return data from the cache instead of hitting the database.

Configurations: Allow to configure the EF model by using data annotation attributes or Fluent API to override default conventions.

Migrations: Provide a set of migration commands that can be executed on the NuGet Package Manager Console or the Command Line Interface to create or manage underlying database Schema.
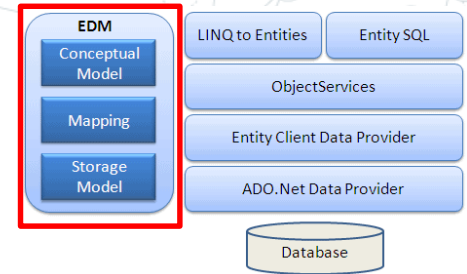
Entity Framework Latest Versions

| EF6	EF Core
First released in 2008 with .NET FW 3.5	First released in June 2016 with .NET Core
Stable and feature rich	New and evolving
Windows only	Windows, Linux, MacOS
Works on .NET FW 3.5+	Works on .NET FW 4.5+ and .NET Core

Entity Framework Architecture
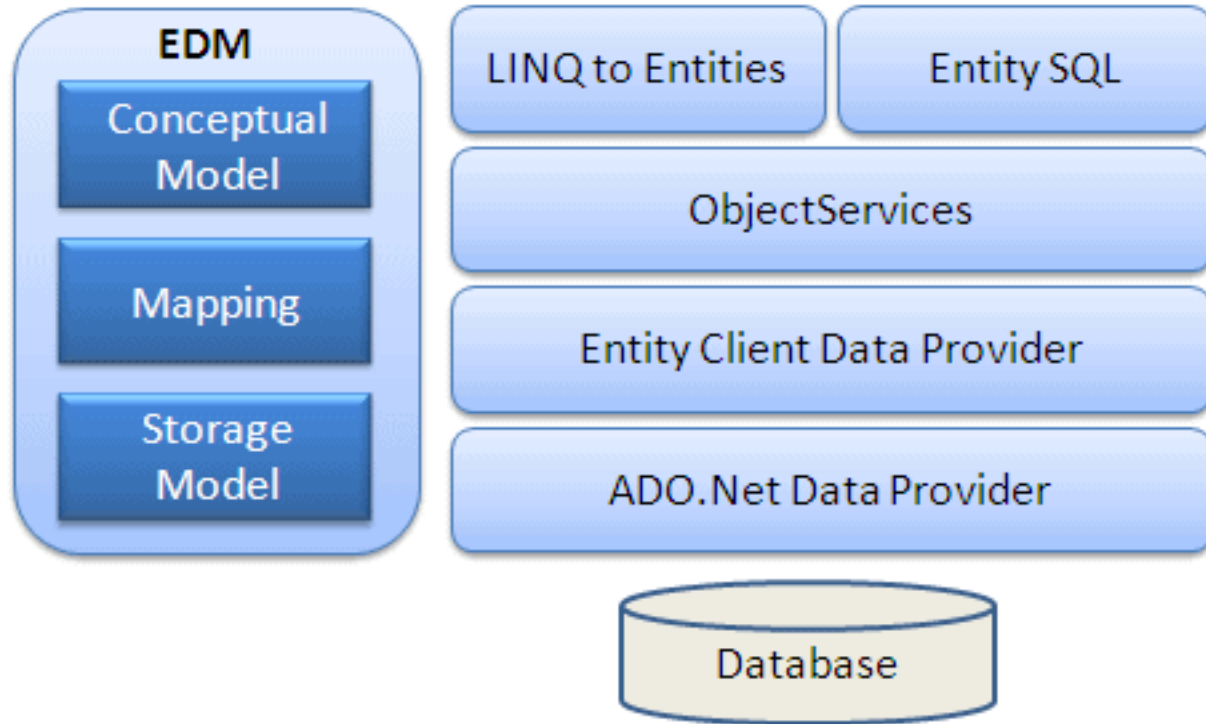
Entity Framework Architecture (cont.)

EDM (Entity Data Model) consists of three main:

◎ **Conceptual Model**: The conceptual model contains the model classes and their relationships. This will be independent from your database table design.

◎ **Storage Model**: The storage model is the database design model which includes tables, views, stored procedures, and their relationships and keys.

Mapping: Mapping consists of information about how the conceptual model is mapped to the storage model.

Entity Framework Architecture (cont.)
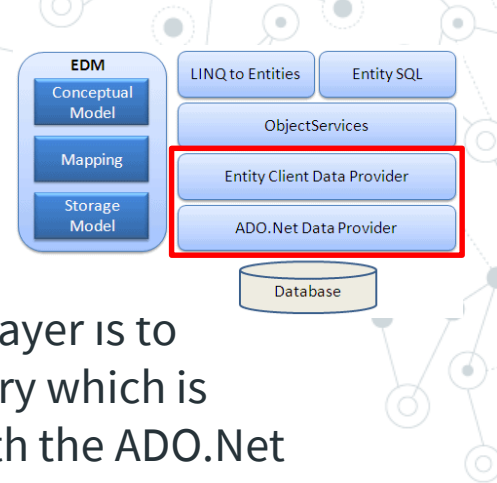
Entity Client Data Provider: The main responsibility of this layer is to convert LINQ-to-Entities or Entity SQL queries into a SQL query which is understood by the underlying database. It communicates with the ADO.Net data provider which in turn sends or retrieves data from the database.

ADO.Net Data Provider: This layer communicates with the database using standard ADO.Net.

DbContext

DbContext represents a session with the underlying database using which you can perform CRUD (Create, Read, Update, Delete) operations.

It is also used to configure domain classes, database related mappings, change tracking settings, caching, transaction etc.

The context class in Entity Framework is a class which derives from `System.Data.Entity.DbContext`.
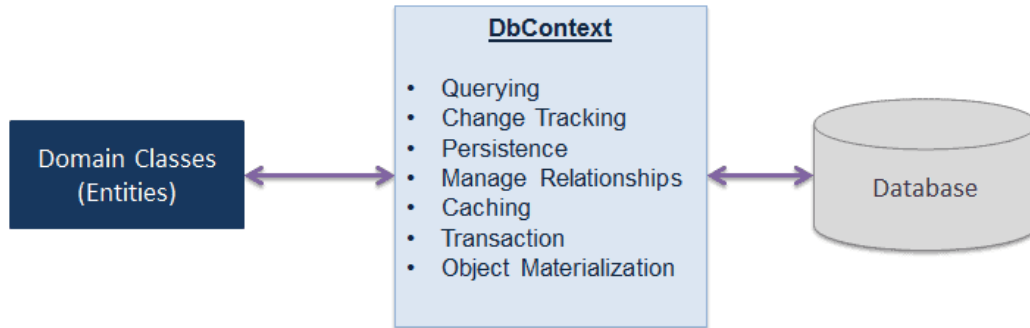
DbContext – Example

```csharp
using System.Data.Entity;

public class SchoolContext : DbContext {
  public SchoolContext() {
  }

  // Entities
  public DbSet<Student> Students { get; set; }
  public DbSet<Examination> Examinations { get; set; }
  public DbSet<Grade> Grades { get; set; }
}
```

Entity

An **Entity** in Entity Framework is a class that maps to a database table.

This class must be included as a **DbSet<TEntity>** type property in the **DbContext** class.

EF maps each entity to a table and each property of an entity to a column in the database.

Entity – Example

```
public class Student {
    public int Id { get; set; }
    public int StudentId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime Birthdate { get; set; }

    public Grade Grade { get; set; }
    public IList<Examination> Exams { get; set; }
}
```

Entity – Include as DbSet<Tentity> property in the context

```
using System.Data.Entity;

public class SchoolContext : DbContext {
  public SchoolContext() : base("name=SchoolDbConnStirng") {
  }

  // Entities
  public DbSet<Student> Students { get; set; }
  public DbSet<Grade> Grades { get; set; }
}
```

Entity – Map to the Students table in the database

Entity – Type of Property

An Entity can include two types of properties:

◎ **Scalar Property**: The primitive type properties are called scalar properties.
- ○ Each scalar property maps to a column in the database table which stores an actual data.

◎ **Navigation Property**: The navigation property represents a relationship to another entity.
- ○ There are two types of navigation properties: Reference Navigation and Collection Navigation

Entity – Type of Entity – Example

```
public class Student {
  // scalar properties
    public int Id { get; set; }
    public int StudentId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime Birthdate { get; set; }


  // reference navigation property
    public Grade Grade { get; set; }
    public IList<Examination> Exams { get; set; }
}
```
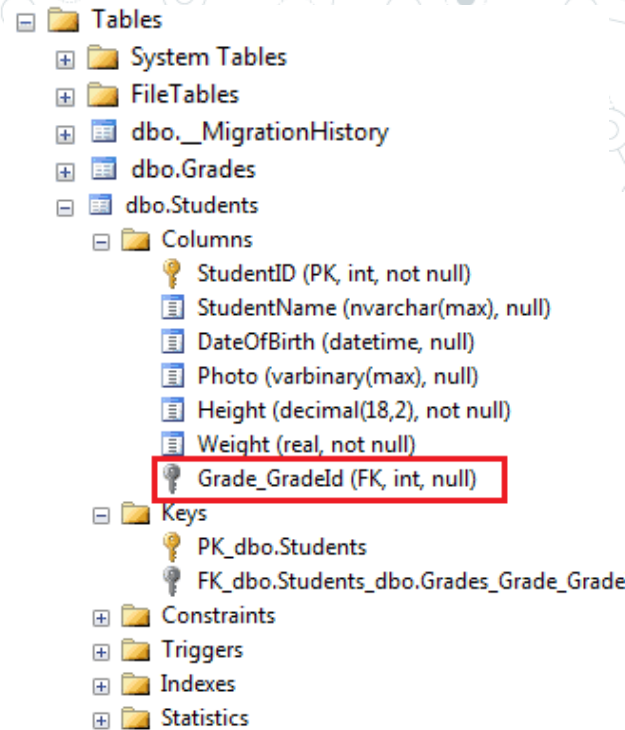
Tables
- System Tables
- FileTables
- dbo.__MigrationHistory
- dbo.Grades
- dbo.Students
  - Columns
    - StudentID (PK, int, not null)
    - StudentName (nvarchar(max), null)
    - DateOfBirth (datetime, null)
    - Photo (varbinary(max), null)
    - Height (decimal(18,2), not null)
    - Weight (real, not null)
    - Grade_GradeId (FK, int, null)
  - Keys
    - PK_dbo.Students
    - FK_dbo.Students_dbo.Grades_Grade_Grade
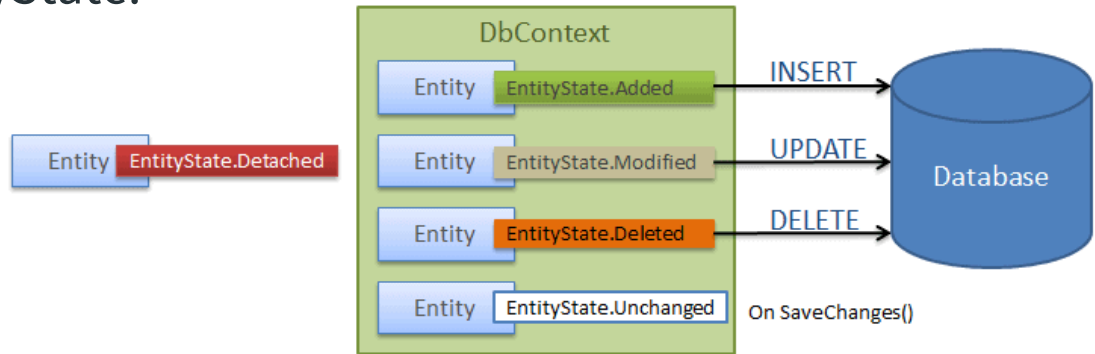 - Constraints
  - Triggers
  - Indexes
  - Statistics

Entity – Entity States

The context keeps trach of entity statues and maintains modifications made to the properties of the entity (**Change tracking**).

The entity state represented by an enum System.Data.Entity.EntityState:

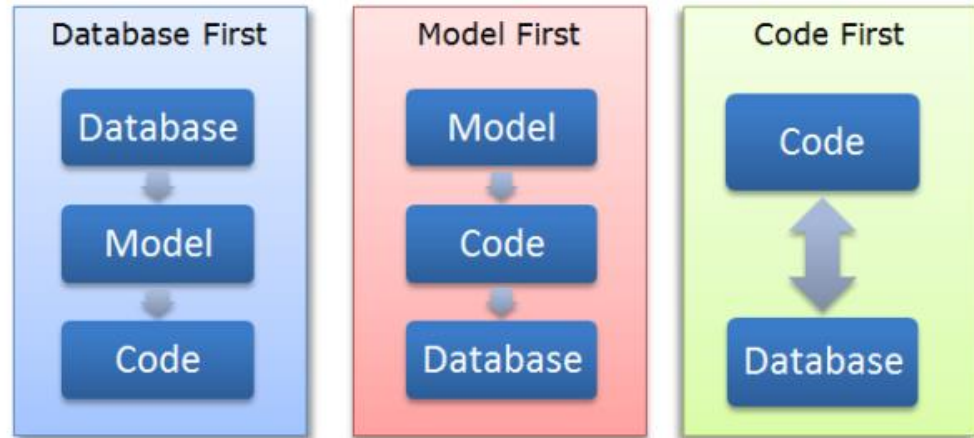◎ Added

◎ Modified

◎ Deleted

◎ Unchanged

◎ Detached

Development Approaches

There are three different approaches you can use while developing your application using Entity Framework:

◎ **Database-First**

◎ **Code-First**

◎ Model-First*

EF Core does not support Model-First.

2.
Creating a model

EF6 – Code-First
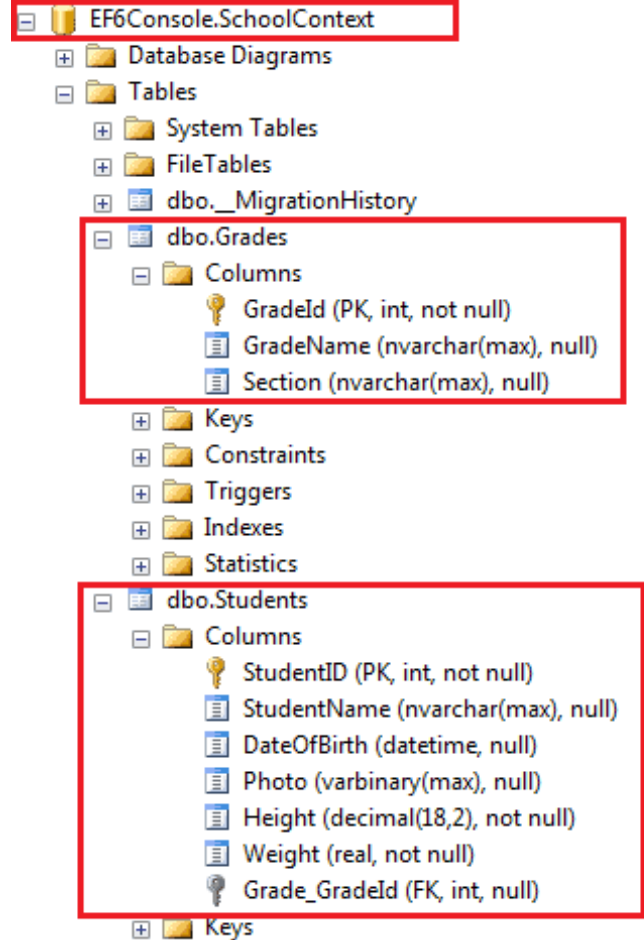
Code-First Example

```
public class Student {
  public int StudentID { get; set; }
  public string StudentName { get; set; }
  public DateTime? DateOfBirth { get; set; }
  public byte[]  Photo { get; set; }
  public decimal Height { get; set; }
  public float Weight { get; set; }

  public Grade Grade { get; set; }
}
```

```
public class Grade {
  public int GradeId { get; set; }
  public string GradeName { get; set; }
  public string Section { get; set; }

  public ICollection<Student> Students
    { get; set; }
}
```

Code-First Example (cont.)

```
public class SchoolContext: DbContext {
    public SchoolContext(): base() { }
    public DbSet<Student> Students { get; set; }
    public DbSet<Grade> Grades { get; set; }
}


using (var ctx = new SchoolContext()) {
    var stud = new Student()
        { StudentName = "Bill" };

    ctx.Students.Add(stud);
    ctx.SaveChanges();
}
```


EF6Console.SchoolContext
- Database Diagrams
- Tables
  - System Tables
  - FileTables
  - dbo.__MigrationHistory
  - dbo.Grades
    - Columns
      - GradeId (PK, int, not null)
      - GradeName (nvarchar(max), null)
      - Section (nvarchar(max), null)
    - Keys
    - Constraints
    - Triggers
    - Indexes
    - Statistics
  - dbo.Students
    - Columns
      - StudentID (PK, int, not null)
      - StudentName (nvarchar(max), null)
      - DateOfBirth (datetime, null)
      - Photo (varbinary(max), null)
      - Height (decimal(18,2), not null)
      - Weight (real, not null)
      - Grade_GradeId (FK, int, null)
    - Keys

Code-First Conventions

Default Convention For	Description
Schema	By default, EF creates all the DB objects into the dbo schema.
Table Name	<Entity Class Name> + 's'
Primary key Name	Id or <Entity Class Name> + "Id" (case insensitive)
Foreign key property Name	<Dependent Navigation Property Name> + "_" + <Principal Entity Primary Key Property Name>
Null column	EF creates a null column for all reference type properties and nullable primitive properties. E.g. string, Nullable<int>, decimal?
Not Null Column	EF creates NotNull columns for Primary Key properties and non-nullable value type properties e.g. int, float, decimal, datetime etc.
DB Columns order	EF will create DB columns in the same order like the properties in an entity class (exclude primary key)
Properties mapping to DB	By default, all properties will map to the database. Use the [NotMapped] attribute to exclude property or class from DB mapping.

C# Data type mapped with SQL Server data type

C# Data Type	SQL Data Type
int	int
string	nvarchar(Max)
decimal	decimal(18,2)
float	real
byte[]	varbinary(Max)
datetime	datetime

C# Data Type	SQL Data Type
bool	bit
byte	tinyint
short	smallint
long	bigint
double	Float
object	No mapping

DB Initialization Stategies in EF6 Code-First

There are four different database initialization strategies:

◎ **CreateDatabaseIfNotExists**: (Default initializer) It will create the database if none exists as per the configuration. It will throw an exception if the model class is different with the DB.

◎ **DropCreateDatabaseIfModelChanges**: This initializer drops an existing database and creates a new database.

◎ **DropCreateDatabaseAlways**: It drops an existing database every time the application starts.

◎ **Custom DB Initializer:** Create your own custom initializer.

DB Initialization Stategies in EF6 Code-First – Example

```
public class SchoolDBContext: DbContext  {

  public SchoolDBContext(): base("SchoolDBConnectionString") {

    Database.SetInitializer<SchoolDBContext>(new
CreateDatabaseIfNotExists<SchoolDBContext>());


    //Database.SetInitializer<SchoolDBContext>(new
DropCreateDatabaseIfModelChanges<SchoolDBContext>());


    //Database.SetInitializer<SchoolDBContext>(new
DropCreateDatabaseAlways<SchoolDBContext>());


    //Database.SetInitializer<SchoolDBContext>(new SchoolDBInitializer());
  }


}
```

DB Initialization Stategies in EF6 Code-First – Example

```
public class SchoolDBInitializer : CreateDatabaseIfNotExists<SchoolDBContext> {

    protected override void Seed(SchoolDBContext context)

    {

        base.Seed(context);

    }

}
```

Configure Domain Classes – The problem

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
}
```

EF_Code_First_Tutorials.SchoolContext
- ⊞ Database Diagrams
- ⊟ Tables
  - ⊞ System Tables
  - ⊞ FileTables
  - ⊞ dbo.__MigrationHistory
  - ⊞ dbo.Standards
  - ⊟ dbo.StudentMaster
    - ⊟ Columns
      - 🔑 StudentId (PK, int, not null)
      - StudentName (nvarchar(max), null)
  - ⊞ Keys
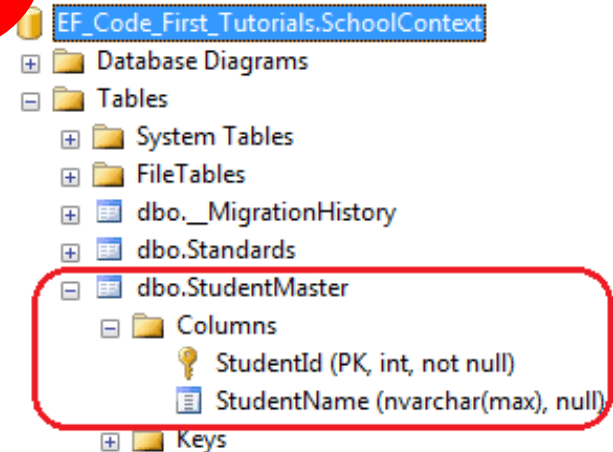
Configure Domain Classes (cont.)

Code-First builds the conceptual model from your domain classes using default conventions.

There are two ways to configure your domain classes:

◎ **Data Annotation Attributes** is a simple attribute based configuration.
  - ○ Apply to domain classes and its properties
  - ○ Read more here

◎ **Fluent API configuration** can be applied when EF builds a model from your domain classes.
  - ○ Inject the Fluent API configurations by overriding the OnModelCreating method of DbContext in EF6
    Read more here

Configure Domain Classes – Data Annotation Attributes

```
[Table("StudentInfo")]
public class Student{
   [Key]
   public int Id { get; set; }


   [Column("Name")]
   [MaxLength(20)]
   public string FirstName { get; set; }


   [NotMapped]
   public int? Age { get; set; }
   public int GradeId { get; set; }
   [ForeignKey("GradeId")]
   public virtual Grade Grade { get; set; }
}
```

Configure Domain Classes – Fluent API

```
public class SchoolContext: DbContext {
  public SchoolDBContext(): base() { }

  public DbSet<Student> Students { get; set; }
  public DbSet<Standard> Standards { get; set; }

  protected override void OnModelCreating(DbModelBuilder modelBuilder) {
    //Configure default schema
    modelBuilder.HasDefaultSchema("Admin");
    //Map entity to table
    modelBuilder.Entity<Student>().ToTable("StudentInfo");
    modelBuilder.Entity<Grade>().ToTable("GradeInfo","dbo");
  }

}
```

Configure Relationship in EF6

◎ Configure One-to-One [here](#)

◎ Configure One-to-Many [here](#)

◎ Configure Many-to-Many [here](#)

3.

Querying data

Executing LINQ-to-Entities

The **DbSet** class is derived from **IQuerayable**. So, we can use LINQ for querying against DbSet, which will be converted to an SQL query.

EF executes this SQL query to the underlying database, gets the flat result set, converts it into appropriate entity objects and returns it as a query result.

Executing LINQ-to-Entities (cont.)

```
var student = ctx.Students
    .Where(s => s.StudentName == "Bill")
    .FirstOrDefault<Student>();
```

```
SELECT TOP (1)
[Extent1].[StudentID] AS [StudentID],
[Extent1].[StudentName] AS [StudentName],
[Extent1].[StandardId] AS [StandardId]
FROM [dbo].[Student] AS [Extent1]
WHERE 'Bill' = [Extent1].[StudentName]
```

Executing LINQ-to-Entities (cont.)

```
var studentList = ctx.Students.Where(s => s.StudentName ==
"Bill").ToList();
```

SELECT

[Extent1].[StudentID] AS [StudentID],

[Extent1].[StudentName] AS [StudentName],

[Extent1].[StandardId] AS [StandardId]

FROM [dbo].[Student] AS [Extent1]

WHERE 'Bill' = [Extent1].[StudentName]

Executing LINQ-to-Entities (cont.)

```
var students = ctx.Students.GroupBy(s => s.StandardId);
```

```sql
SELECT
[Project2].[C1] AS [C1],
[Project2].[StandardId] AS [StandardId],
[Project2].[C2] AS [C2],
[Project2].[StudentID] AS [StudentID],
[Project2].[StudentName] AS [StudentName],
[Project2].[StandardId1] AS [StandardId1]
FROM ( SELECT
    [Distinct1].[StandardId] AS [StandardId],
    1 AS [C1],
    [Extent2].[StudentID] AS [StudentID],
    [Extent2].[StudentName] AS [StudentName],
    [Extent2].[StandardId] AS [StandardId1],
    CASE WHEN ([Extent2].[StudentID] IS NULL) THEN CAST(NULL AS int) ELSE 1 END AS [C2]
    FROM   (SELECT DISTINCT
        [Extent1].[StandardId] AS [StandardId]
        FROM [dbo].[Student] AS [Extent1] ) AS [Distinct1]
    LEFT OUTER JOIN [dbo].[Student] AS [Extent2] ON ([Distinct1].[StandardId] = [Extent2].[StandardId]) OR (([Distinct1].
)  AS [Project2]
ORDER BY [Project2].[StandardId] ASC, [Project2].[C2] ASC
go
```

Executing a Raw SQL Query

The following methods can be used to execute raw SQL queries to the database using EF6:

◎ `DbSet.SqlQuery()`

◎ `DbContext.Database.SqlQuery()`

◎ `DbContext.Database.ExecuteSqlCommand()`

Executing a Raw SQL Query – Example

```
using (var ctx = new SchoolDBEntities()) {

  var studentList = ctx.Students.SqlQuery("Select * from Students")
    .ToList<Student>();


  //Get student name of string type
  string studentName = ctx.Database.SqlQuery<string>("Select studentname from
Student where studentid=1").FirstOrDefault();


  int noOfRowUpdated = ctx.Database.ExecuteSqlCommand("Update student set
studentname ='changed student by command' where studentid=1");

  int noOfRowInserted = ctx.Database.ExecuteSqlCommand("Insert into
student(studentname) values('New Student')");

  int noOfRowDeleted = ctx.Database.ExecuteSqlCommand("Delete from student
where studentid=1");

}
```

4.
Saving data

Saving Data in EF

Disconnected Entity

Connected Entity

Saving Data in the Connected Scenario – Inserting

Use the DbSet.Add method to add a new entity to a context (instance of DbContext), which will insert a new record in the database when you call the SaveChanges() method.

```csharp
using (var context = new SchoolDBEntities())
{
    var std = new Student()
    {
        FirstName = "Bill",
        LastName = "Gates"
    };
    context.Students.Add(std);

    context.SaveChanges();
}
```

```sql
exec sp_executesql N'INSERT [dbo].[Students]([FirstName], [LastName])
VALUES (@0, @1)
SELECT [StudentId]
FROM [dbo].[Students]
WHERE @@ROWCOUNT > 0 AND [StudentId] = scope_identity()',N
''@0 nvarchar(max) ,@1 nvarchar(max) ',@0=N'Bill',@1=N'Gates'
go
```

Saving Data in the Connected Scenario – Updating

EF keeps track of all the entities retrieved using a context.

Therefore, when you edit entity data, EF automatically marks EntityState to Modified, which results in an updated statement in the database when you call the SaveChanges() method.

```csharp
using (var context = new SchoolDBEntities())
{
    var std = context.Students.First<Student>();
    std.FirstName = "Steve";
    context.SaveChanges();
}
```
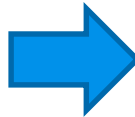
```sql
exec sp_executesql N'UPDATE [dbo].[Students]
SET [FirstName] = @0
WHERE ([StudentId] = @1)',
N'@0 nvarchar(max) ,@1 int',@0=N'Steve',@1=2
Go
```

Saving Data in the Connected Scenario – Deleting

Use the DbSet.Remove() method to delete a record in the database table.

```csharp
using (var context = new SchoolDBEntities())
{
    var std = context.Students.First<Student>();
    context.Students.Remove(std);

    context.SaveChanges();
}
```
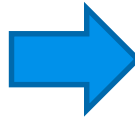
```sql
exec sp_executesql N'DELETE [dbo].[Students]
WHERE ([StudentId] = @0)',N'@0 int',@0=1
Go
```

Attach Disconnected Entities

There are two things we need to do when we get a disconnected entity:

◎ First, attach entities with the new context instance and make the context aware about these entities.

◎ Second, set an appropriate `EntityState` to each entity manually
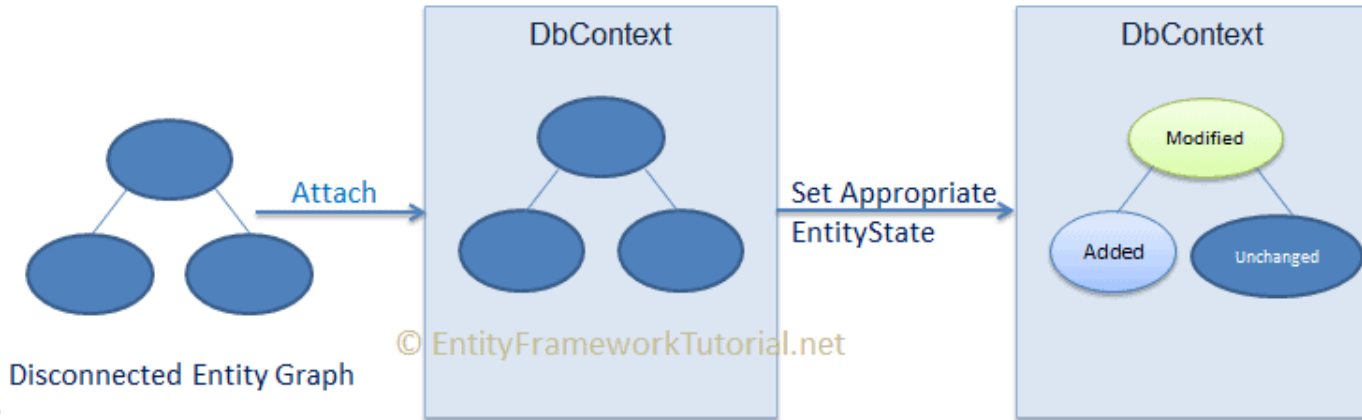
Attach Disconnected Entities (cont.)

Entity Framework provides the following methods that attach disconnected entities to a context and also set the EntityState to each entity in an entity graph.

◎ **DbContext.Entry()**: Use to change the EntityState.

◎ **DbSet.Add()**: Attach the entity to a context and automatically applies the Added state to all entities

◎ **DbSet.Attach()**: Attach an entire entity graph to the new context with the <u>Unchanged</u> entity state

Attach Disconnected Entities (cont.) – DbSet.Attach()

```csharp
// Disconnected entity
Student disconnectedStudent = new Student() { StudentName = "New Student" };

using (var context = new SchoolDBEntities()) {
  context.Students.Attach(disconnectedStudent);
  context.Entry(student).State = EntityState.Modified;
}
```

5.
Using Transactions

The problem

```
using (var context = new SchoolContext())
{
    context.Database.Log = Console.Write;

    var standard = context.Standards.

    context.Students.Add(new St
    {
        FirstName = "Rama
        StandardId = standard.S
    });

    context.SaveChanges();

    context.Courses.Add(new Course() { CourseName
    context.SaveChanges();
}
```

Each SaveChange() method call creates a new transaction and executes database command within it.

New Transaction

Commit

New Transaction

Commit

Multiple SaveChanges in a Single Transaction

Using the following methods to create or use a single transaction with multiple SaveChanges() calls:

1. DbContext.Database.**BeginTransaction()**: Creates a new transaction for the underlying database and allows us to commit or roll back changes made to the database.

2. DbContext.Database.**UseTransaction()**: Allows to pass an existing transaction object created out of the scope of a context object.
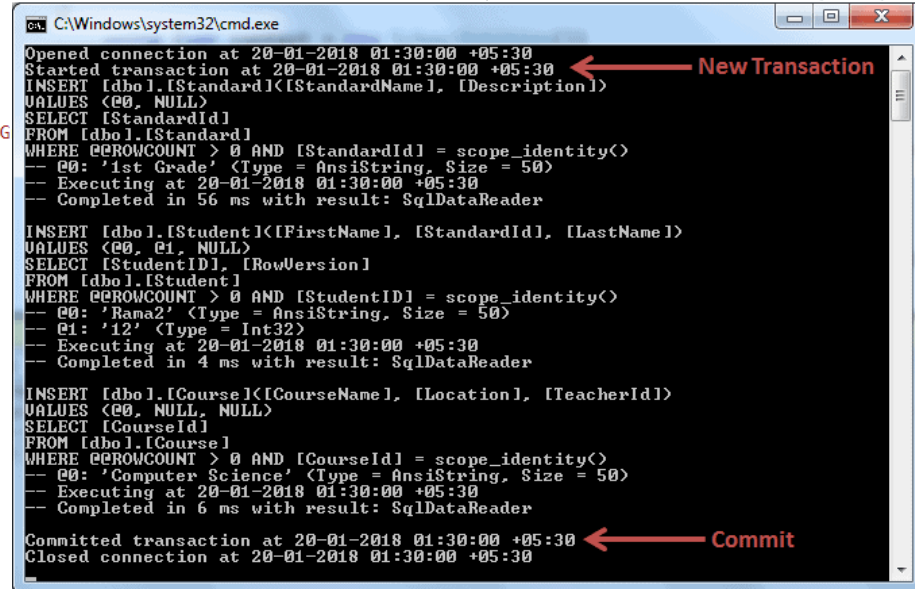
DbContext.Database.BeginTransaction() – Commit

```csharp
using (var context = new SchoolContext())
{
    context.Database.Log = Console.Write;

    using (DbContextTransaction transaction = context.Database.BeginTransaction())
    {
        try
        {
            var standard = context.Standards.Add(new Standard() { StandardName = "1st G

            context.Students.Add(new Student()
            {
                FirstName = "Rama2",
                StandardId = standard.StandardId
            });
            context.SaveChanges();

            context.Courses.Add(new Course() { CourseName = "Computer Science" });
            context.SaveChanges();

            transaction.Commit();
        }
        catch (Exception ex)
        {
            transaction.Rollback();
            Console.WriteLine("Error occurred.");
        }
    }
}
```

DbContext.Database.BeginTransaction() – Rollback

```csharp
using (var context = new SchoolContext())
{
    context.Database.Log = Console.Write;

    using (DbContextTransaction transaction = context.Database.BeginTransaction())
    {
        try
        {
            var standard = context.Standards.Add(new Standard() { StandardName = "

            context.Students.Add(new Student()
            {
                FirstName = "Rama",
                StandardId = standard.StandardId
            });
            context.SaveChanges();
            // throw exectiopn to test roll back transaction
            throw new Exception();

            context.Courses.Add(new Course() { CourseName = "Computer Science" });
            context.SaveChanges();

            transaction.Commit();
        }
        catch (Exception ex)
        {
            transaction.Rollback();
            Console.WriteLine("Error occurred.");
        }
    }
}
```
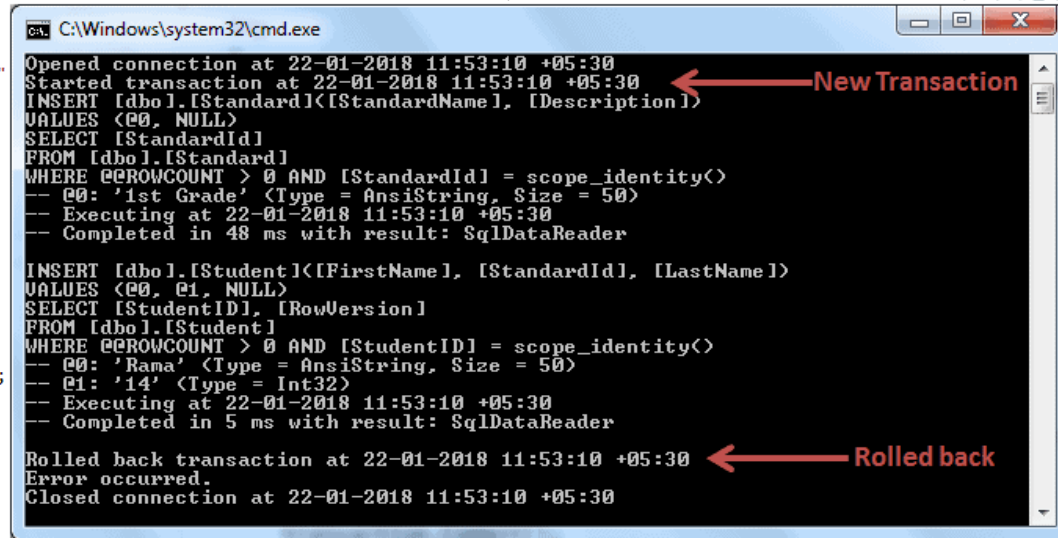
```
C:\Windows\system32\cmd.exe

Opened connection at 22-01-2018 11:53:10 +05:30
Started transaction at 22-01-2018 11:53:10 +05:30          ← New Transaction
INSERT [dbo].[Standard]([StandardName], [Description])
VALUES (@0, NULL)
SELECT [StandardId]
FROM [dbo].[Standard]
WHERE @@ROWCOUNT > 0 AND [StandardId] = scope_identity()
-- @0: '1st Grade' (Type = AnsiString, Size = 50)
-- Executing at 22-01-2018 11:53:10 +05:30
-- Completed in 48 ms with result: SqlDataReader

INSERT [dbo].[Student]([FirstName], [StandardId], [LastName])
VALUES (@0, @1, NULL)
SELECT [StudentID], [RowVersion]
FROM [dbo].[Student]
WHERE @@ROWCOUNT > 0 AND [StudentID] = scope_identity()
-- @0: 'Rama' (Type = AnsiString, Size = 50)
-- @1: '14' (Type = Int32)
-- Executing at 22-01-2018 11:53:10 +05:30
-- Completed in 5 ms with result: SqlDataReader

Rolled back transaction at 22-01-2018 11:53:10 +05:30       ← Rolled back
Error occurred.
Closed connection at 22-01-2018 11:53:10 +05:30
```

6.
Migration

Code-Based Migration

In order to use code-based migration, executing the following commands in the Package Manager Console in Visual Studio:

1. **Enable-Migrations**: Enables the migration in the project.
2. **Add-Migration**: Creates a new migration class as per specified name with the Up() and Down() methods.
3. **Update-Database**: Executes the last migration file created by the Add-Migration command and applies changes to the database schema.

Step 1: Enable-Migrations

The `Enable-Migrations` command will create the `Configuration` class derived from `DbMigrationsConfiguration` with `AutomaticMigrationsEnabled = false.`
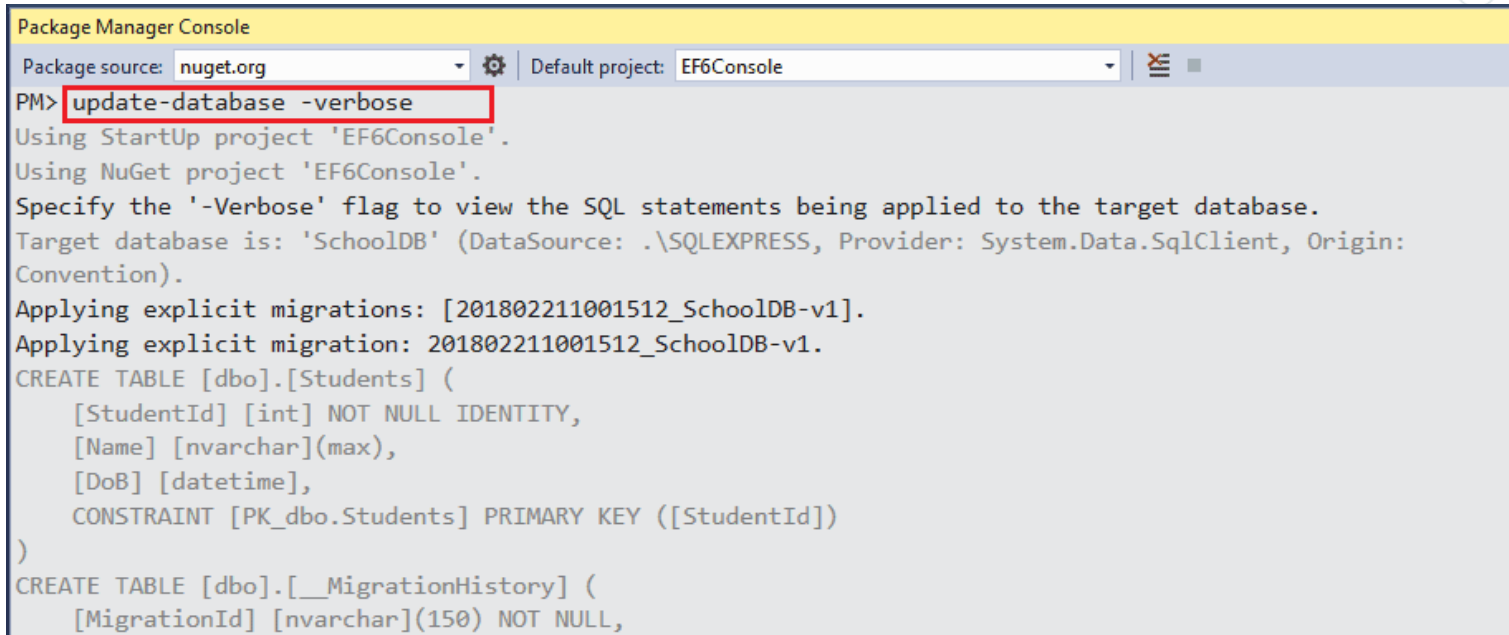
Set the database initializer `MigrateDatabaseToLatestVersion` in your context class:

```csharp
public class SchoolContext: DbContext
{
    public SchoolDBContext(): base("SchoolDB")
    {
        Database.SetInitializer(new MigrateDatabaseToLatestVersion<SchoolDBContext, EF6Console.Migrations.Configuration>());
    }
}
```

Step 2: Add-Migration
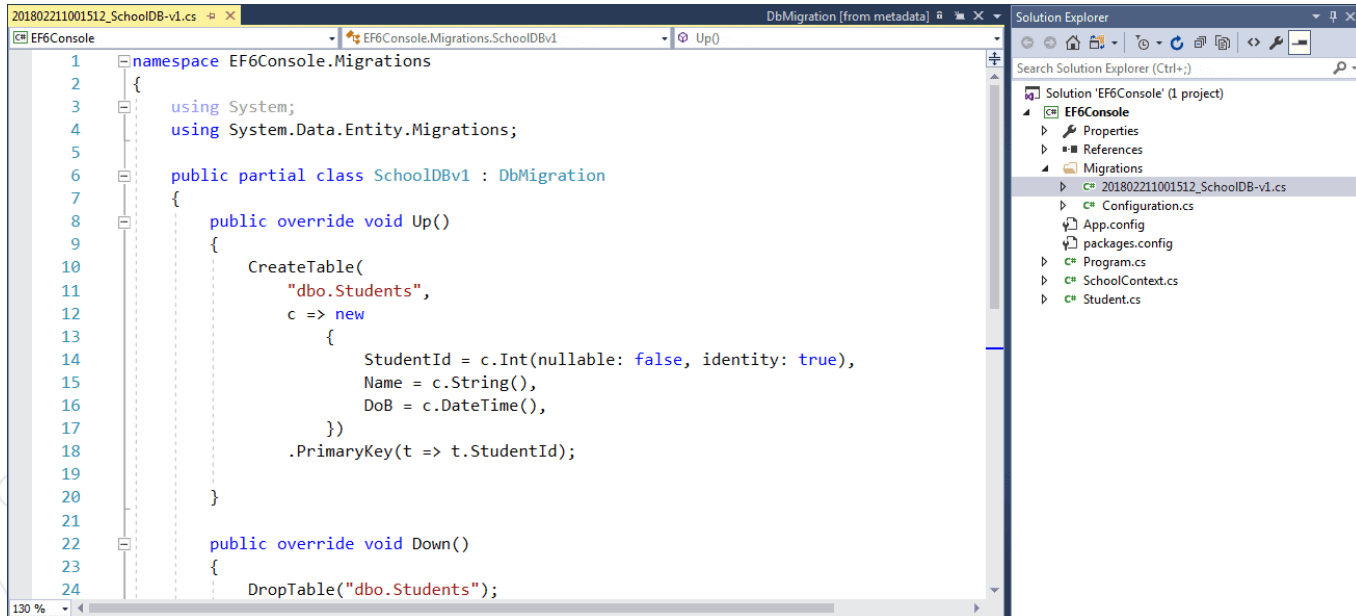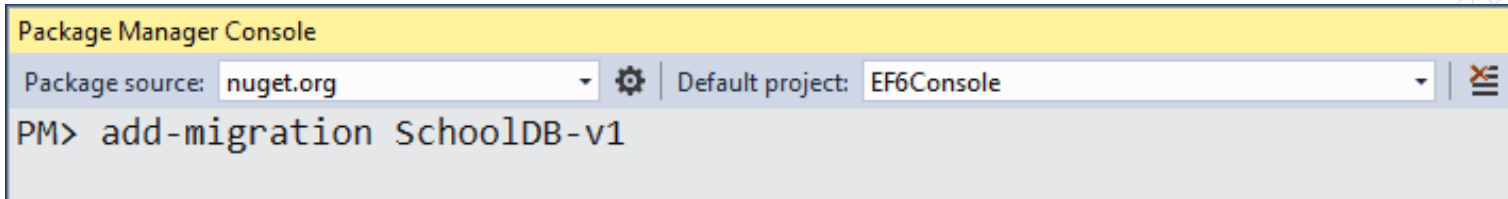
Step 3: Update-Database


```
Package Manager Console
Package source: nuget.org          ⚙   Default project: EF6Console                    ≚  ■
PM> update-database -verbose
Using StartUp project 'EF6Console'.
Using NuGet project 'EF6Console'.
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Target database is: 'SchoolDB' (DataSource: .\SQLEXPRESS, Provider: System.Data.SqlClient, Origin:
Convention).
Applying explicit migrations: [201802211001512_SchoolDB-v1].
Applying explicit migration: 201802211001512_SchoolDB-v1.
CREATE TABLE [dbo].[Students] (
    [StudentId] [int] NOT NULL IDENTITY,
    [Name] [nvarchar](max),
    [DoB] [datetime],
    CONSTRAINT [PK_dbo.Students] PRIMARY KEY ([StudentId])
)
CREATE TABLE [dbo].[__MigrationHistory] (
    [MigrationId] [nvarchar](150) NOT NULL,
```

Use the `-verbose` option to view the SQL statements being applied to the
target database.

Demo

- ◎ Setup EF with Code first approach
- ◎ DB Initialization Strategies
- ◎ Configure Domain Classes
- ◎ Configure Relationship
  - ○ One to one
  - ○ One to many
- ◎ Execute some queries (and log to the console)
- ◎ Save entity: Insert, Update and Delete
- ◎ Update a disconnected entity.

Review Questions

1. What is ORM?

2. What are various approaches to domain modeling in Entity Framework?

3. What are the various Entity States in EF?

4. What C# Datatype is mapped with which Datatype in SQL Server?

Reference

◎ https://www.entityframeworktutorial.net/what-is-entityframework.aspx

Thanks!

Any questions?

You can find me at:

tranminhphuoc@gmail.com