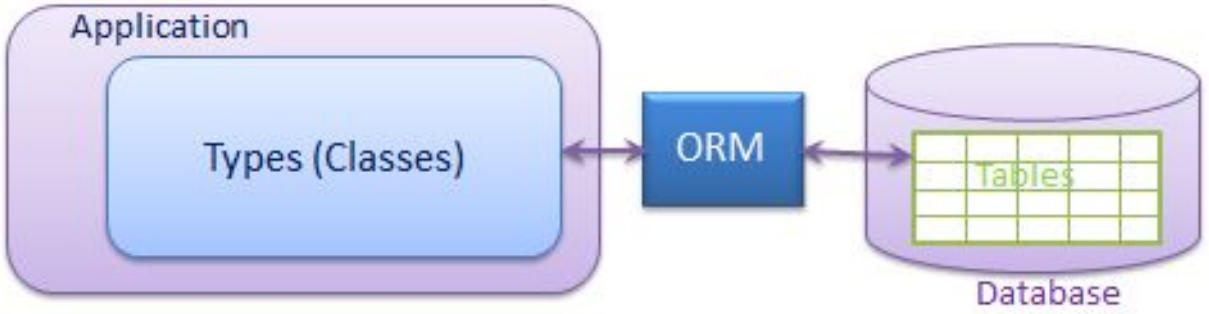
DAPPER VS EF

Agenda

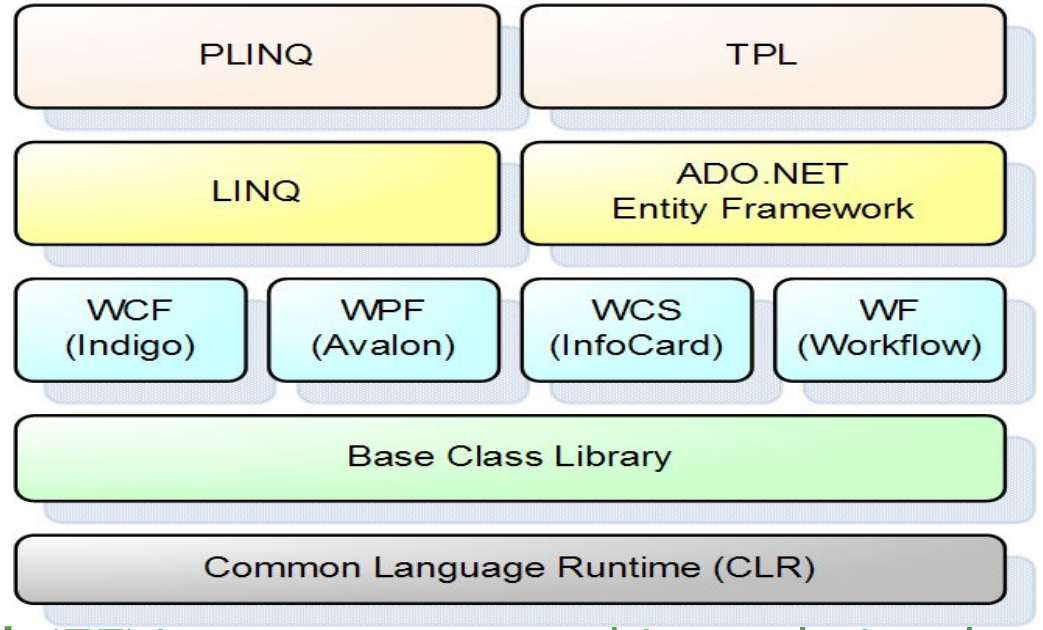
- ORM
- Entity Framework
- DB Working approaches
- Database initialization
- Data Annotations
- Fluent API
- Migration
- Query Examples
- Lazy Loading
- Dapper
- How Dapper Works?
- Fluent Map

ORM



- Object-relational mapping (ORM) is a programming technique in which a metadata descriptor is used to connect object code to a relational database.
- ORM allows us to keep our database design separate from our domain class design.

ADO.NET Entity Framework



- **Entity Framework** (**EF**) is an open source object-relational mapping (ORM) framework for ADO.NET.

Advantages and Disadvantages

Advantages:

- One common syntax (LINQ) for all object queries
- Auto generated code
- Reduce development time/cost

Disadvantages:

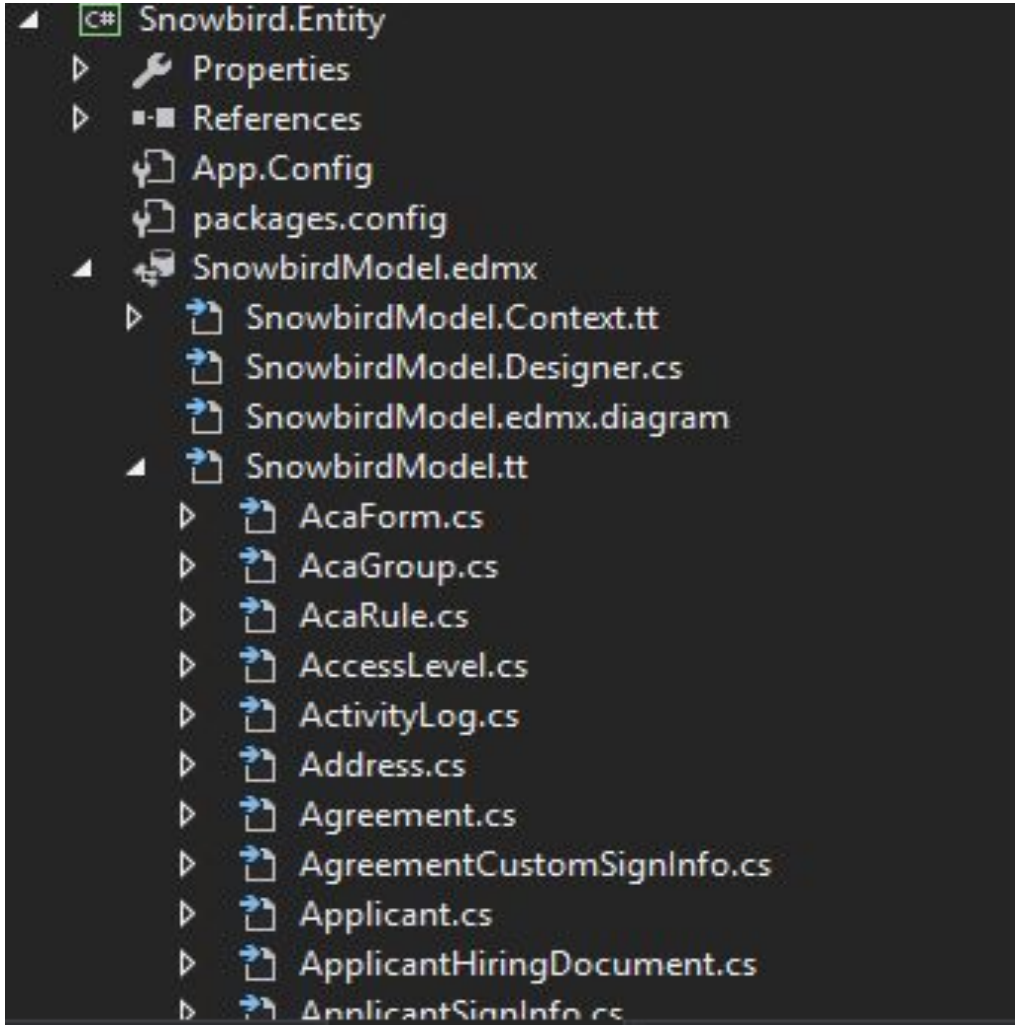
- Performance
- DB Schema Dependency
- Scalability (not good for huge domain models)

DB Working approaches

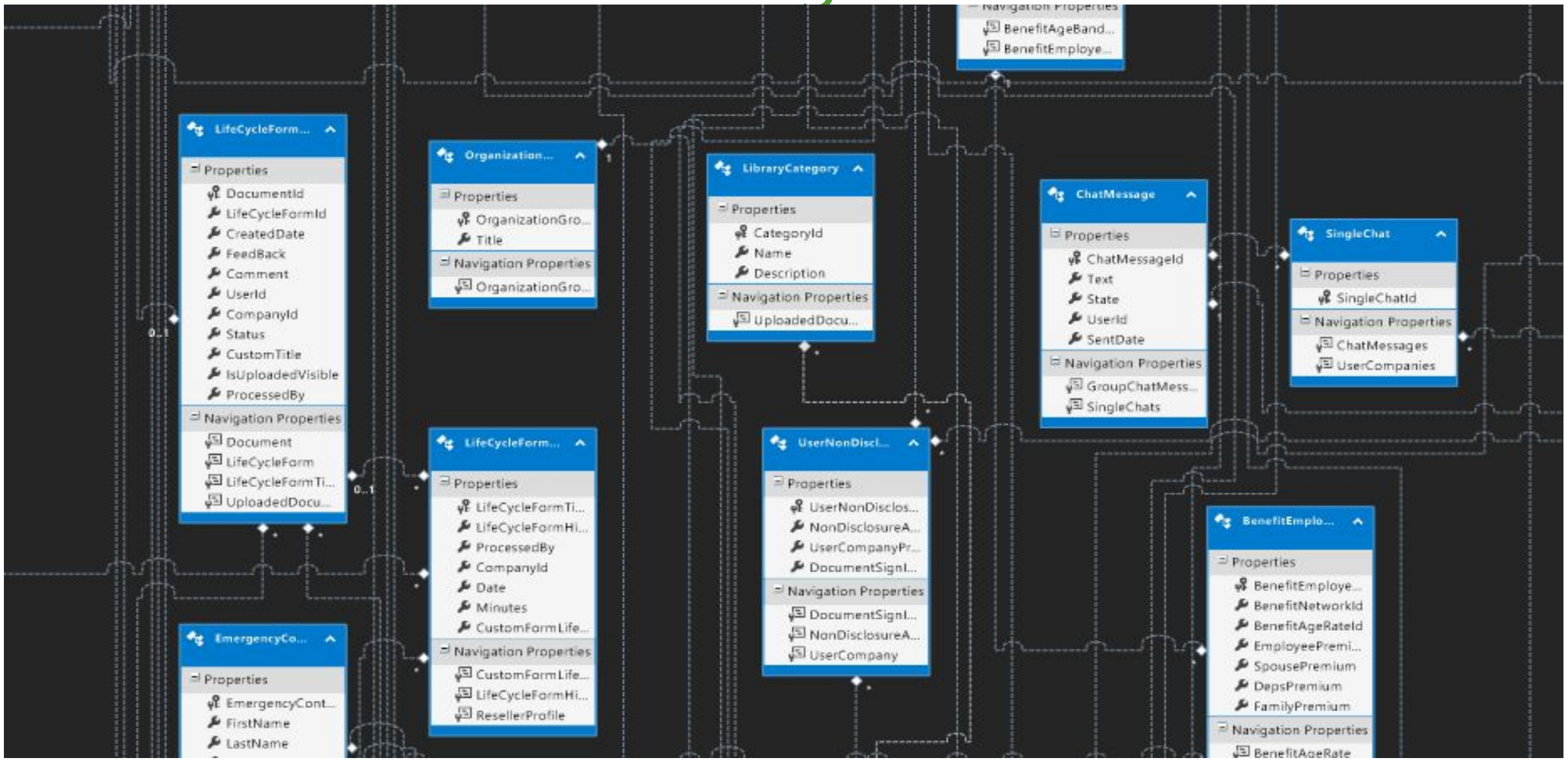
- Code first
- DB first
- Schema first

DB First

- Allows to use an existing DB
- Generates EDMX based on DB schema



DB Diagram


BenefitAgeBand...
BenefitEmploye...
LifeCycleForm...
Properties
DocumentId
LifeCycleFormId
CreatedDate
FeedBack
Comment
UserId
CompanyId
Status
CustomTitle
IsUploadedVisible
ProcessedBy
Navigation Properties
Document
LifeCycleForm
LifeCycleFormTi...
UploadedDocu...
Organization...
Properties
OrganizationGro...
Title
Navigation Properties
OrganizationGro...
LibraryCategory
Properties
CategoryId
Name
Description
Navigation Properties
UploadedDocu...
ChatMessage
Properties
ChatMessageId
Text
State
UserId
SentDate
Navigation Properties
GroupChatMess...
SingleChats
SingleChat
Properties
SingleChatId
Navigation Properties
ChatMessages
UserCompanies
LifeCycleForm...
Properties
LifeCycleFormTi...
LifeCycleFormHi...
ProcessedBy
CompanyId
Date
Minutes
CustomFormLife...
Navigation Properties
CustomFormLife...
LifeCycleFormHi...
ResellerProfile
UserNonDiscl...
Properties
UserNonDisclos...
NonDisclosureA...
UserCompanyPr...
DocumentSignI...
Navigation Properties
DocumentSignI...
NonDisclosureA...
UserCompany
BenefitEmplo...
Properties
BenefitEmploye...
BenefitNetworkId
BenefitAgeRateId
EmployeePremi...
SpousePremium
DepsPremium
FamilyPremium
Navigation Properties
BenefitAgeRate
EmergencyCo...
Properties
EmergencyCont...
FirstName
LastName
0..1
0..1
1

Entity Models: One-to-Many

```
86 references
public partial class Right
{
    26 references | 0/1 passing
    public Right()
    {
        this.RoleRights = new HashSet<RoleRight>();
        this.TodoCategories = new HashSet<TodoCategory>();
        this.RoleAssignedCcompanies = new HashSet<RoleAssignedCompany>();
    }

    39 references | 0/1 passing
    public int RightId { get; set; }
    27 references
    public string Name { get; set; }
    26 references | 0/1 passing
    public int RigthCategoryId { get; set; }
    1 reference
    public virtual ICollection<RoleRight> RoleRights { get; set; }
```

```
3 references
public partial class RightCategory
{
    0 references
    public RightCategory()
    {
        this.Rights = new HashSet<Right>();
    }

    0 references
    public int RightCategoryId { get; set; }
    0 references
    public string Title { get; set; }

    1 reference
    public virtual ICollection<Right> Rights { get; set; }
}
```

Entity Models: Many-to-Many

```
public partial class RoleRight
{
    11 references | 0/1 passing
    public int RoleRightId { get; set; }
    10 references
    public string RoleId { get; set; }
    45 references
    public int RightId { get; set; }

    0 references
    public virtual AspNetRole AspNetRole { get; set; }
    13 references | 0/1 passing
    public virtual Right Right { get; set; }
}
```

```
86 references
public partial class Right
{
    26 references | 0/1 passing
    public Right()
    {
        this.RoleRights = new HashSet<RoleRight>();
        this.TodoCategories = new HashSet<TodoCategory>();
        this.RoleAssignedCompanies = new HashSet<RoleAssignedCompany>();
    }

    39 references | 0/1 passing
    public int RightId { get; set; }
    27 references
    public string Name { get; set; }
    26 references | 0/1 passing
    public int RigthCategoryId { get; set; }
    1 reference
    public virtual ICollection<RoleRight> RoleRights { get; set; }
```

```
public partial class AspNetRole
{
    11 references | 0/1 passing
    public AspNetRole()...

    35 references | 0/2 passing
    public string Id { get; set; }
    25 references | 0/3 passing
    public string Name { get; set; }

    51 references | 0/1 passing
    public virtual ICollection<RoleRight> RoleRights { get; set; }
```

Code First

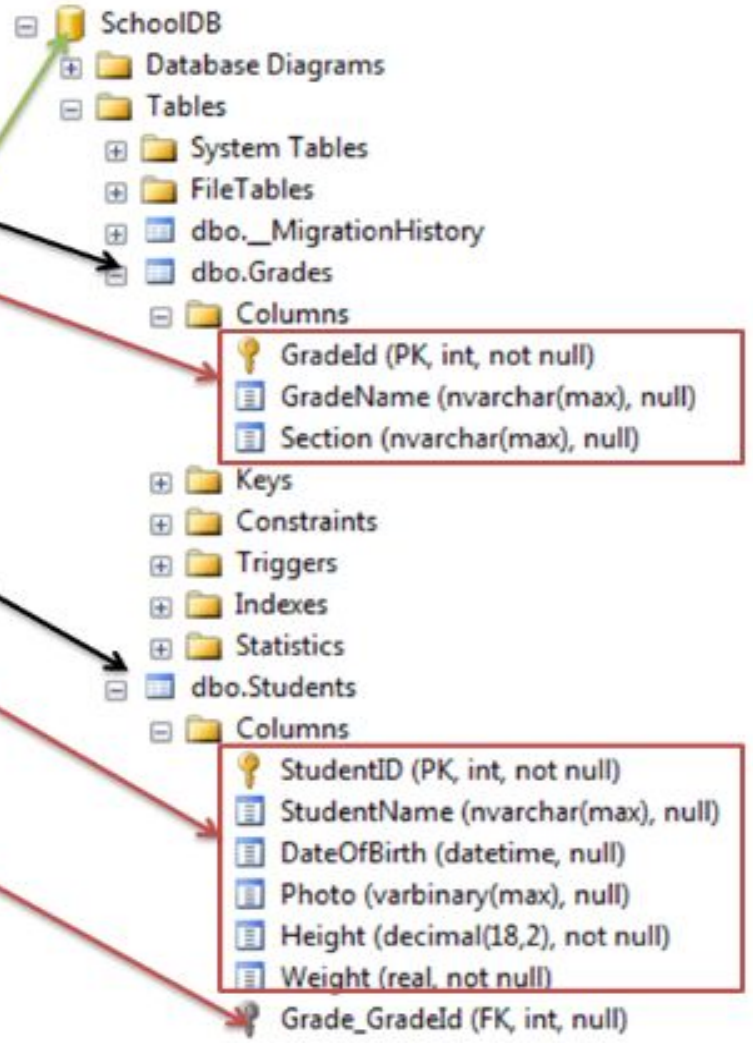
- **Development Speed** - You do not have to worry about creating a DB you just start coding. Good for developers coming from a programming background without much DBA experience.
- Automated DB update according to your models.

DB Types

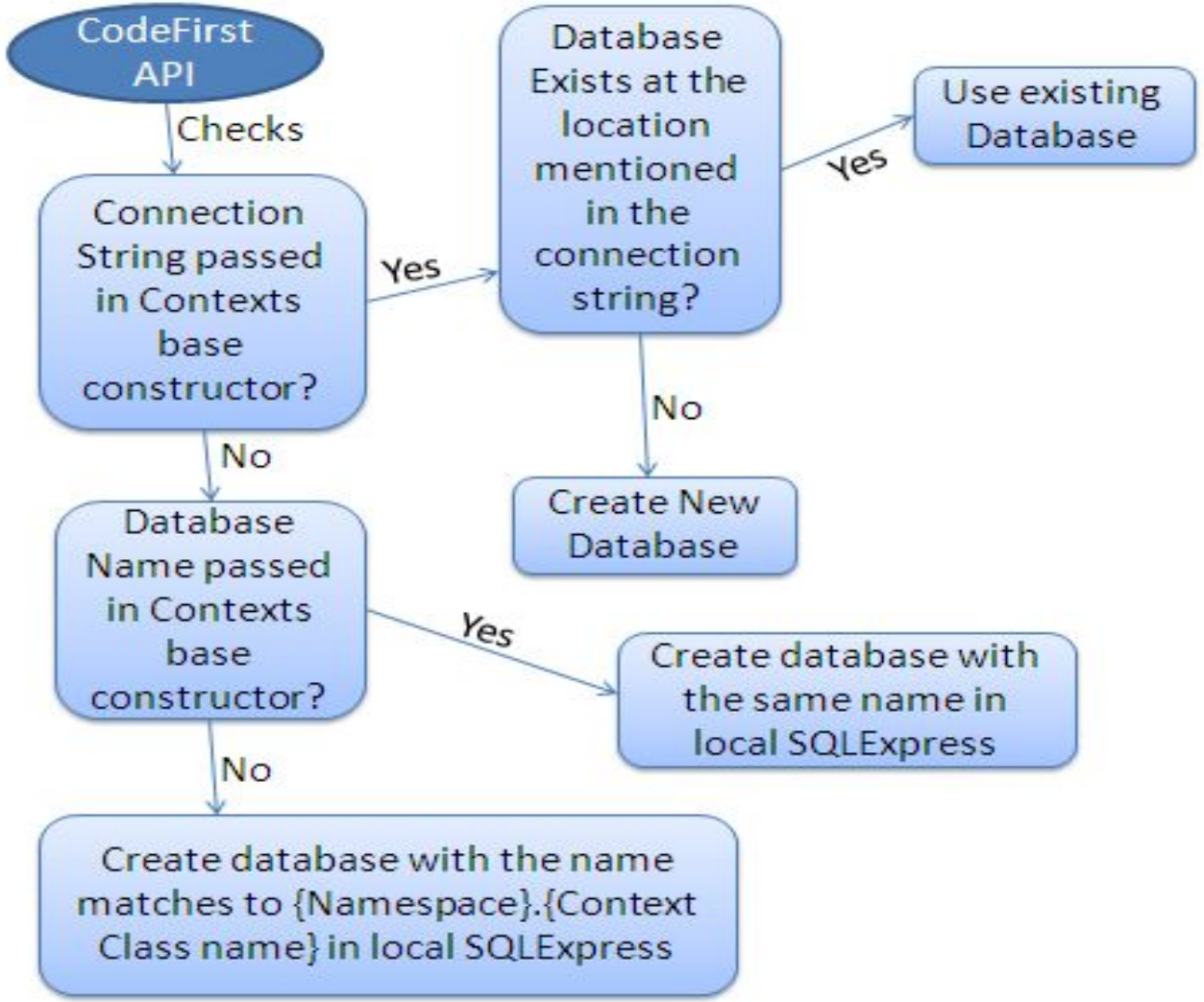
C# Data Type	Mapping to SQL Server Data Type
int	int
string	nvarchar(Max)
decimal	decimal(18,2)
float	real
byte[]	varbinary(Max)
datetime	datetime
bool	bit
byte	tinyint
short	smallint
long	bigint
double	float
char	No mapping
sbyte	No mapping (throws exception)
object	No mapping

Mapping with database

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

    public ICollection<Student> Students { get; set; }
}
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 SchoolContext : DbContext
{
    public SchoolContext() : base("SchoolDB")
    {

    }

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



Database Initialization



Database Initialization

- **No parameter**. Database name = {Namespace}.{Context class name}

```
namespace CustomNamespace
{
     public class Context: DbContext
     {
          public Context(): base()
          {

          }
     }
}
```

- **Database name.**

```
namespace CustomNamespace
{
    public class Context: DbContext
    {
        public Context(): base("MyDBName")
        {

        }
    }
}
```

- **Connection String**

```
namespace CustomNamespace
{
    public class Context: DbContext
    {
        public Context() : base("name=MyDBNameConnectionString")
        {
        }
    }
}
```

Database Initialization Strategies

- CreateDatabaseIfNotExists
- DropCreateDatabaseIfModelChanges
- DropCreateDatabaseAlways
- Custom DB Initializer

```
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());
    }

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

Custom DB Initializer

```
public class SchoolDBInitializer : DropCreateDatabaseAlways<SchoolDBContext>
{
    protected override void Seed(SchoolDBContext context)
    {
        IList<Standard> defaultStandards = new List<Standard>();

        defaultStandards.Add(new Standard() { StandardName = "Standard 1", Description = "First Standard" });
        defaultStandards.Add(new Standard() { StandardName = "Standard 2", Description = "Second Standard" });
        defaultStandards.Add(new Standard() { StandardName = "Standard 3", Description = "Third Standard" });

        context.Standards.AddRange(defaultStandards);

        base.Seed(context);
    }
}
```

Turn off the DB Initializer

```
public class SchoolDBContext: DbContext
{
    public SchoolDBContext() : base("SchoolDBConnectionString")
    {
        //Disable initializer
        Database.SetInitializer<SchoolDBContext>(null);
    }
    public DbSet<Student> Students { get; set; }
    public DbSet<Standard> Standards { get; set; }
}
```

Data Annotations: System.ComponentModel.DataAnnotations

Attribute	Description
Key	Can be applied to a property to specify a key property in an entity and make the corresponding column a PrimaryKey column in the database.
Timestamp	Can be applied to a property to specify the data type of a corresponding column in the database as `rowversion`.
ConcurrencyCheck	Can be applied to a property to specify that the corresponding column should be included in the optimistic concurrency check.
Required	Can be applied to a property to specify that the corresponding column is a NotNull column in the database.
MinLength	Can be applied to a property to specify the minimum string length allowed in the corresponding column in the database.
MaxLength	Can be applied to a property to specify the maximum string length allowed in the corresponding column in the database.
StringLength	Can be applied to a property to specify the maximum string length allowed in the corresponding column in the database.

Data Annotations: System.ComponentModel.DataAnnotations.Schema

Attribute	Description
Table	Can be applied to an entity class to configure the corresponding table name and schema in the database.
Column	Can be applied to a property to configure the corresponding column name, order and data type in the database.
Index	Can be applied to a property to configure that the corresponding column should have an Index in the database. (EF 6.1 onwards only)
ForeignKey	Can be applied to a property to mark it as a foreign key property.
NotMapped	Can be applied to a property or entity class which should be excluded from the model and should not generate a corresponding column or table in the database.
DatabaseGenerated	Can be applied to a property to configure how the underlying database should generate the value for the corresponding column e.g. identity, computed or none.
InverseProperty	Can be applied to a property to specify the inverse of a navigation property that represents the other end of the same relationship.
ComplexType	Marks the class as complex type in EF 6. EF Core 2.0 does not support this attribute.

Fluent API

- Entity Framework Fluent API is used to configure classes to override conventions.
- To write Fluent API configurations, override the OnModelCreating() method of DbContext in a context class, as shown below.

```
public class SchoolContext: DbContext
{

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

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        //Write Fluent API configurations here

    }
}
```

Fluent API: Configure Default Schema

```
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");
    }
}
```

Fluent API: Map Entity to Table

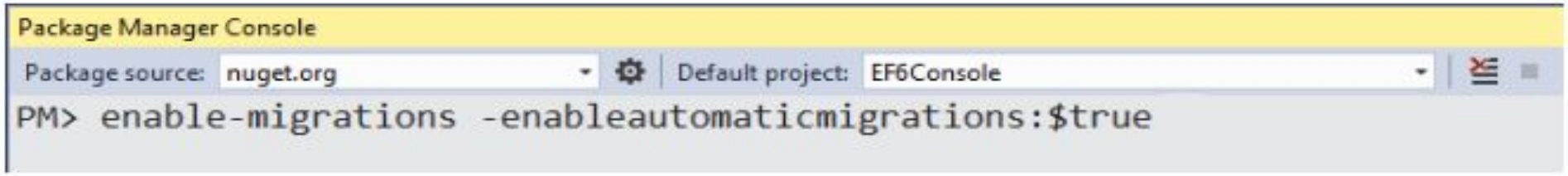
```
public class SchoolContext: DbContext
{
    public SchoolDBContext(): base()
    {
    }

    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<Standard>().ToTable("StandardInfo","dbo");
    }
}
```

Migration

- Automated Migration
- Code-based Migration

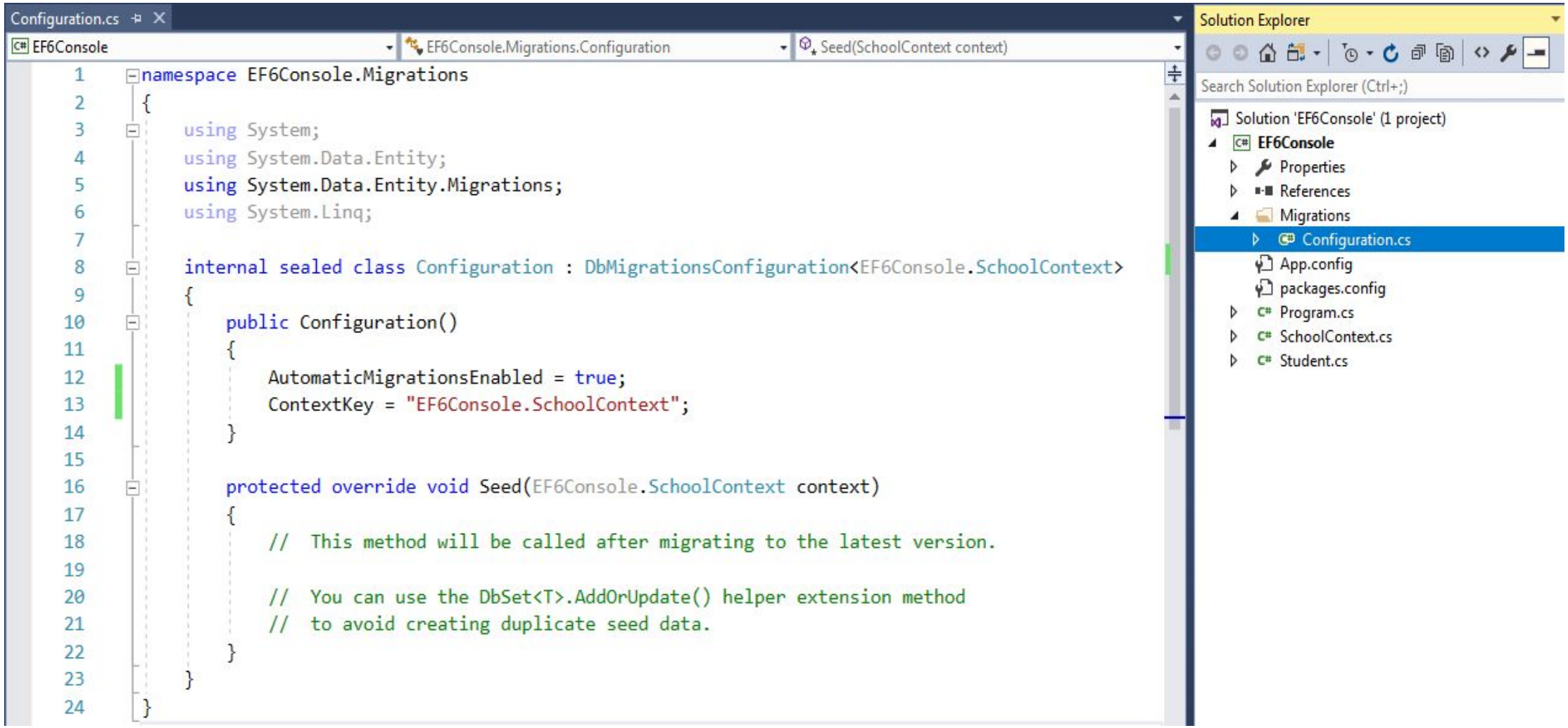
Automated Migration

- Tools → Library Package Manager → Package Manager Console
- Make sure that the default project is the project where your context class is
- Run the ***enable-migrations –EnableAutomaticMigration:$true*** command
- Set the database initializer in the context class to MigrateDatabaseToLatestVersion



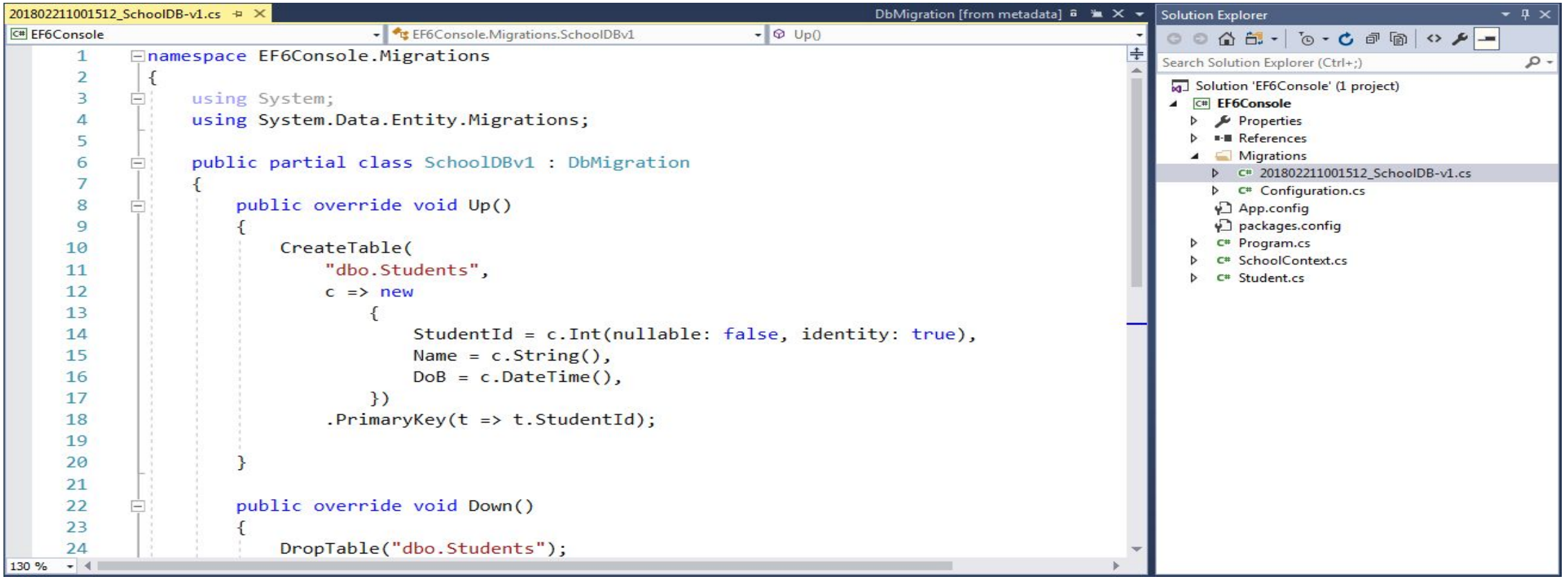
This works only if you add new classes or remove classes, but it won't work when you add, modify or remove properties.

Automated Migration Result



Code-based Migration

- **Enable-Migrations:** Enables the migration in your project by creating a Configuration class.
- **Add-Migration:** Creates a new migration class as per specified name with the Up() and Down() methods. Example: ***add-migration <MIGRATION_NAME>***
- **Update-Database:** Executes the last migration file created by the Add-Migration command and applies changes to the database schema.



Query Examples

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

Parameterized Query

```
using (var ctx = new SchoolDBEntities())
{
    string name = "Bill";
    var student = ctx.Students
                  .Where(s => s.StudentName == name)
                  .FirstOrDefault<Student>();
}
```

```
SELECT TOP (1)
[Extent1].[StudentId] AS [StudentId],
[Extent1].[Name] AS [Name]
FROM [dbo].[Student] AS [Extent1]
WHERE ([Extent1].[Name] = @p__linq__0) OR (([Extent1].[Name] IS NULL)
        AND (@p__linq__0 IS NULL))',N'@p__linq__0 nvarchar(4000)',@p__linq__0=N'Bill'
```

Lazy loading

- Lazy loading is delaying the loading of related data, until you specifically request for it.

```
using (var ctx = new SchoolDBEntities())
{
    //Loading students only
    IList<Student> studList = ctx.Students.ToList<Student>();

    Student std = studList[0];

    //Loads Student address for particular Student only (seperate SQL query)
    StudentAddress add = std.StudentAddress;
}
```

```
SELECT
[Extent1].[StudentID] AS [StudentID],
[Extent1].[StudentName] AS [StudentName],
[Extent1].[StandardId] AS [StandardId]
FROM [dbo].[Student] AS [Extent1]
```

```
exec sp_executesql N'SELECT
[Extent1].[StudentID] AS [StudentID],
[Extent1].[Address1] AS [Address1],
[Extent1].[Address2] AS [Address2],
[Extent1].[City] AS [City],
[Extent1].[State] AS [State]
FROM [dbo].[StudentAddress] AS [Extent1]
WHERE [Extent1].[StudentID] = @EntityKeyValue1',N'@EntityKeyValue1 int',@EntityKeyValue1=1
```

Disable Lazy loading

- We can disable lazy loading for a particular entity or a context. To turn off lazy loading for a particular property, do not make it virtual. To turn off lazy loading for all entities in the context, set its configuration property to false.

```
public partial class SchoolDBEntities : DbContext
{
    public SchoolDBEntities(): base("name=SchoolDBEntities")
    {
        this.Configuration.LazyLoadingEnabled = false;
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
    }
}
```

Lazy loading Rules

- *context.Configuration.ProxyCreationEnabled* should be true.
- *context.Configuration.LazyLoadingEnabled* should be true.
- Navigation property should be defined as public, virtual. Context will **NOT** do lazy loading if the property is not defined as virtual.

IEnumerable<T> vs IQueryable<T>

```
using (var ctx = new SchoolDBEntities())
{
    var id = 3;

    var phones = ctx.Phones().Where(p => p.Id > id).ToList();

}
```

IEnumerable<T>

```
SELECT
    [Extent1].[Id] AS [Id],
    [Extent1].[Name] AS [Name],
    [Extent1].[Company] AS [Company]
    FROM [dbo].[Phones] AS [Extent1]
```

IQueryable<T>

```
SELECT
    [Extent1].[Id] AS [Id],
    [Extent1].[Name] AS [Name],
    [Extent1].[Company] AS [Company]
    FROM [dbo].[Phones] AS [Extent1]
    WHERE [Extent1].[Id] >3
```

Dapper

- Dapper is a simple object mapper for .NET and own the title of **King of Micro ORM** in terms of speed and is virtually as fast as using a raw ADO.NET data reader.

Advantages and Disadvantages

Advantages:

- Performance
- Easy integration

Disadvantages:

- Attention to Data Types
- Support
- A lot of SQL in the code

DB Working approaches

- DB First

How Dapper Works?

- Create an IDbConnection object.
- Write a query to perform CRUD operations.
- Pass query as a parameter in Execute method.

```
public override IEnumerable<TEntity> ExecuteQuery(string queryString, object parameters)
{
    using (IDbConnection db = new SqlConnection(base.CurrentContext.DbConnection))
    {
        var queryArgs = ParameterBuilder.BuildParametersFromModel(parameters);
        return db.Query<TEntity>(queryString, queryArgs);
    }
}
```

Dapper Parameters

- Anonymous
- Dynamic
- List
- String

```
public IEnumerable<AccessPointMenuEM> GetMenuItems(int roleId, int permissionId)
{
    var query = @"SELECT apm.* FROM tAccessPointMenu AS apm
                  JOIN tAccessPoint AS ap ON apm.AccessPointId = ap.AccessID
                  JOIN tUserRoleAccess AS ura ON ap.AccessID = ura.AccessID
                  WHERE ura.RoleID = @RoleID
                    AND ura.PermissionID = @PermissionID
                    AND apm.IsActive = 1";
    var queryParams = new DynamicParameters();
    queryParams.Add("@RoleID", roleId, DbType.Int32, ParameterDirection.Input);
    queryParams.Add("@PermissionID", permissionId, DbType.Int32, ParameterDirection.Input);

    using (IDbConnection db = new SqlConnection(base.CurrentContext.DbConnection))
    {
        return db.Query<AccessPointMenuEM>(query, queryParams);
    }
}
```

Dapper: Entity Models

```cs
using Dapper.Contrib.Extensions;
using System.Collections.Generic;

[Table("tAccessPoint")]
public class AccessPointEM
{
    public AccessPointEM()
    {
        Permissions = new List<PermissionEM>();
    }

    [Key]
    public int AccessID { get; set; }
    public string Code { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public bool IsActive { get; set; }

    #region Navigation Properties
    public ICollection<PermissionEM> Permissions { get; set; }
    #endregion
```

```cs
public override long Insert(TEntity entity)
{
    using (IDbConnection db = new SqlConnection(base.CurrentContext.DbConnection))
    {
        var result = db.Insert<TEntity>(entity);
        return result != null ? long.Parse(result.ToString()) : default(long);
    }
}
```

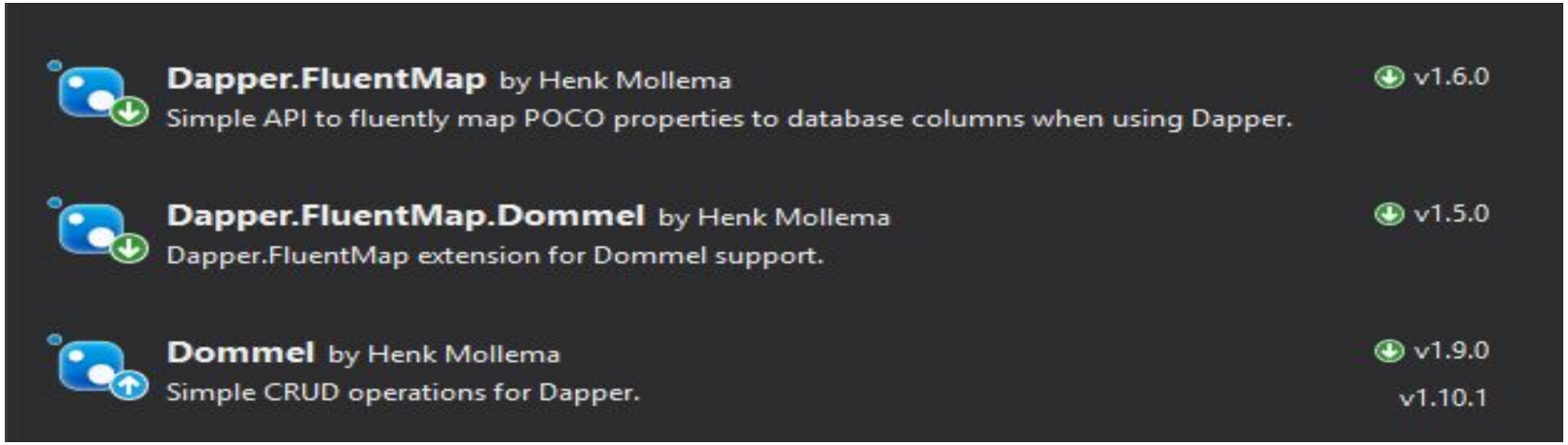
Dapper: Entity Models

```
public class UserRoleAccessEM
{
    public int RoleAccessID { get; set; }
    public int RoleID { get; set; }
    public int AccessID { get; set; }
    public int PermissionID { get; set; }
}
```

```
public void AssignAccessPoint(int roleId, IEnumerable<UserRoleAccessEM> userRoles)
{
    const string sqlInsertQuery =
        @"INSERT INTO [dbo].[tUserRoleAccess]([RoleID], [AccessID], [PermissionID])
                        VALUES(@RoleID, @AccessID, @PermissionID)";

    ExecuteQuery(sqlInsertQuery, userRoles);
}
```

Dapper: Fluent Map

- Fluent Map allows to associate your models with specific tables in DB.
- To use Mapping you need to install the following packages:



Dapper: Fluent Map usage

```
private void InitializeEntityMappings()
{
    FluentMapper.Initialize(config =>
    {
        //// TODO: add entity mapping here
        AddMapping(config, new ContactMap());
        AddMapping(config, new PersonDiagnosisMap());
        AddMapping(config, new ProviderDetailMap());

        config.ForDommel();
        DommelMapper.SetKeyPropertyResolver(new PrimaryKeyResolver());
        DommelMapper.SetTableNameResolver(new TableNameResolver());
    });
}
```

Custom Mapping

```
using Dapper.FluentMap.Dommel.Mapping;
using NMN.Data.Entity;

public class ContactMap : DommelEntityMap<ContactEM>
{
    public ContactMap()
    {
        Map(p => p.Primary).ToColumn("[Primary]");
    }
}
```

Useful links

- http://www.entityframeworktutorial.net/ - Entity Framework(EF) and EF Core tutorials
- https://metanit.com/sharp/entityframework/ - EF tutorial (in Russian)
- https://dapper-tutorial.net/ - Dapper ORM tutorial

Q&A session