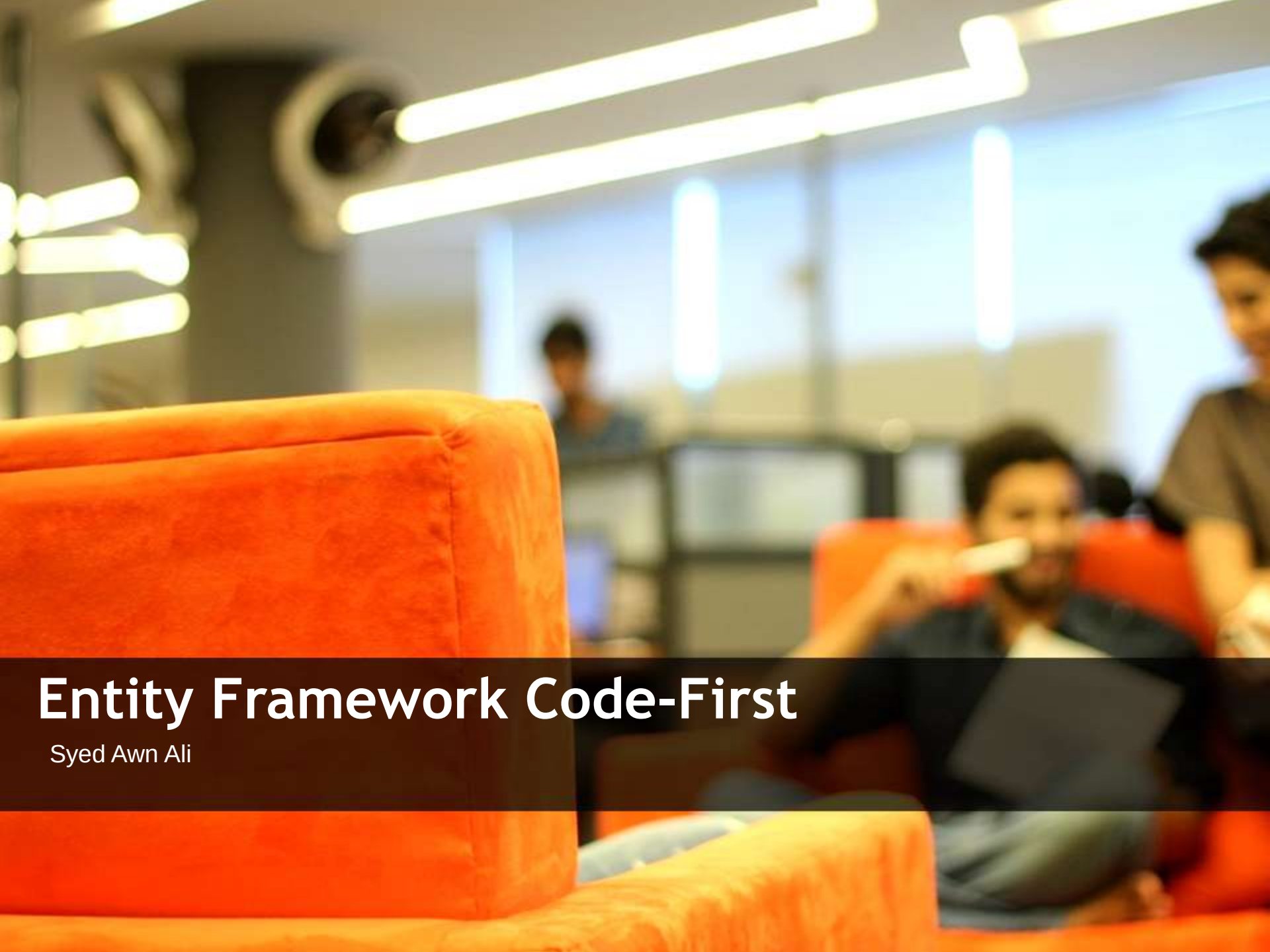


Entity Framework Code-First

Syed Awn Ali

Entity Framework

- Object/Relational Mapping (O/RM)
- Enhancement to ADO.NET
- Why Entity Framework?
  - Domain objects to relational database without much programming
  - Maintainable and extendable
  - Automates CRUD
- Other ORMs
  - DataObjects.Net, NHibernate, OpenAccess, SubSonic

Types Of EF

- Code First
  - Write your classes first
- Model First
  - Create Entities, relationships, and inheritance hierarchies
- Database First
  - Generate EDMX from existing database

Code First EF

- Starts from Entity Framework 4.1
- Features
  - Useful in Domain Driven Design
  - One-to-one, one-to-many and many-to-many relationship
  - DataAnnotation
  - Fluent API
- Prerequisites:
  - .Net Framework3.5, C#, Visual Studio 2010 and MS SQL Server

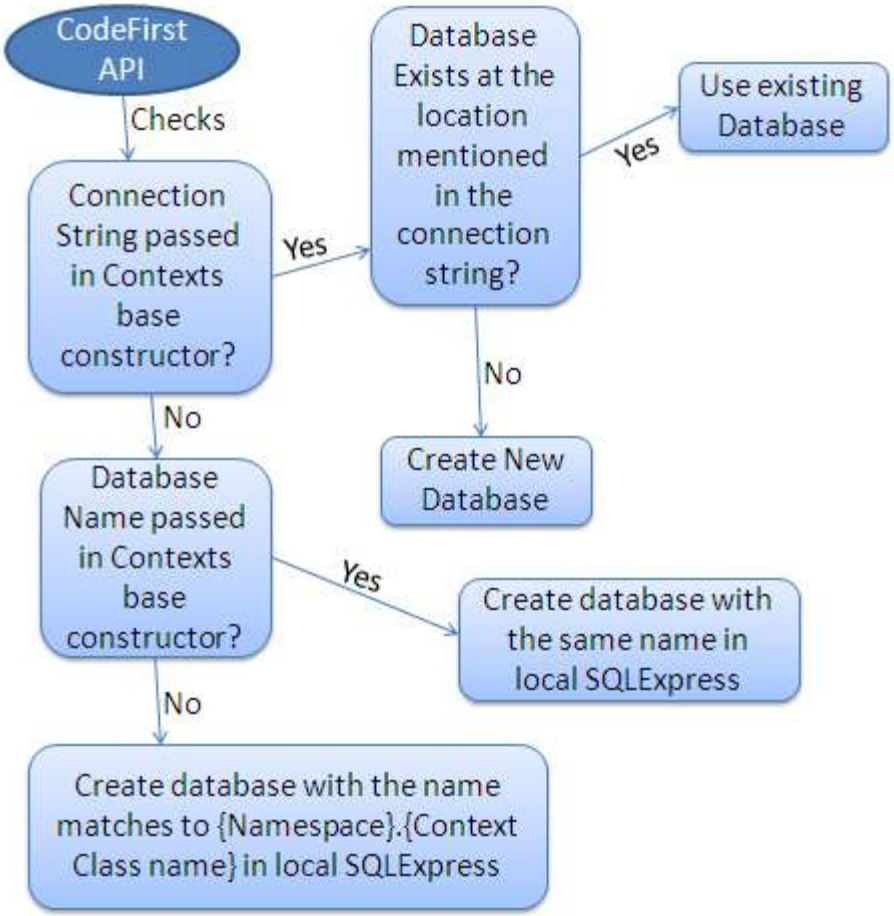
Basic Workflow

- Create classes
- Create new database
- Map your classes
- Basic workflow:
  - Write code-first application classes
  - Hit F5 to run the application
  - Code First API creates new database or map with existing database from application classes
  - Inserts default/test data into the database
  - Finally launch the application

Simple Code First Example

- Quick Start Example Demonstration

Database Initialization



Database Initialization

- No Parameter:
  - Creates the database in your local SQLEXPRESS with name matches your {Namespace}.{Context class name}

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

    }
}
```

Database Initialization

- Name:
  - Creates database in the local SQLEXPRESS db server using that name

```
public class Context: DbContext
{
    public Context(): base("SchoolDB-CodeFirst")
    {

    }
}
```

Database Initialization

- ConnectionStringName:
  - Create the database as per connection string

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="SchoolDBConnectionString"
    connectionString="Data Source=.;Initial Catalog=SchoolDB-ByConnectionString;Integrated Security=
    providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>
```

Database Initialization Strategies

- CreateDatabaseIfNotExists:
  - Default initializer
  - Create the database if not exists
- DropCreateDatabaseIfModelChanges:
  - Creates new database if your model classes have been changed
- DropCreateDatabaseAlways:
  - Drops and Creates an existing database every time
- Custom DB Initializer
  - Create your own custom initializer

Turn off DB Initializer

- Production environment you don't want to lose existing data

```
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; }
}
```

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="DatabaseInitializerForType SchoolDataLayer.SchoolDBContext, SchoolDataLayer"
         value="Disabled" />
  </appSettings>
</configuration>
```

Seed Database

- Provide some test data for your application

```
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 Star
        defaultStandards.Add(new Standard() { StandardName = "Standard 2", Description = "Second Sta
        defaultStandards.Add(new Standard() { StandardName = "Standard 3", Description = "Third Star

        foreach (Standard std in defaultStandards)
            context.Standards.Add(std);

        //All standards will
        base.Seed(context);
    }
}
```

Configure Domain Classes

- There are two ways by which you can configure your domain classes:
  - DataAnnotation
  - Fluent API

DataAnnotation

- Attribute based configuration
- System.ComponentModel.DataAnnotations namespace
- Provides only subset of Fluent API
- Don’t find some attributes in DataAnnotation then you have to use Fluent API

DataAnnotation

- Example Code:

```
[Table("StudentInfo")]
public class Student
{
    public Student() { }

    [Key]
    public int SID { get; set; }

    [Column("Name", TypeName="ntext")]
    [MaxLength(20)]
    public string StudentName { get; set; }

    [NotMapped]
    public int? Age { get; set; }


    public int StdId { get; set; }

    [ForeignKey("StdId")]
    public virtual Standard Standard { get; set; }
}
```

DataAnnotation

- **Validation Attributes:**

Annotation Attribute	Description
Required	The Required annotation will force EF (and MVC) to ensure that property has data in it.
MinLength	MinLength annotation validates property whether it has minimum length of array or string.
MaxLength	MaxLength annotation maximum length of property which in-tern set maximum length of column in the database
StringLength	Specifies the minimum and maximum length of characters that are allowed in a data field.

DataAnnotation

- **Database Schema related Attributes:**

Annotation Attribute	Description
Table	Specify name of the DB table which will be mapped with the class
Column	Specify column name and datatype which will be mapped with the property
Key	Mark property as EntityKey which will be mapped to PK of related table.
ComplexType	Mark the class as complex type in EF.
Timestamp	Mark the property as a non-nullable timestamp column in the database.
ForeignKey	Specify Foreign key property for Navigation property
NotMapped	Specify that property will not be mapped with database
ConcurrencyCheck	ConcurrencyCheck annotation allows you to flag one or more properties to be used for concurrency checking in the database when a user edits or deletes an entity.
DatabaseGenerated	DatabaseGenerated attribute specifies that property will be mapped to Computed column of the database table. So the property will be read-only property. It can also be used to map the property to identity column (auto incremental column).
InverseProperty	InverseProperty is useful when you have multiple relationship between two classes.

DataAnnotation

```
[Table("StudentInfo")]
public class Student
{
    public Student(){ }

    [Key]
    public int SID { get; set; }

    [Required(ErrorMessage="Student Name is Required" )]
    [Column("Name", TypeName="ntext")]
    [MaxLength(20), MinLength(2, ErrorMessage="Student name can not be 2 character or less")]
    public string StudentName { get; set; }

    [NotMapped]
    public int? Age { get; set; }

    [ConcurrencyCheck()]
    [Timestamp]
    public Byte[] LastModifiedTimestamp { get; set; }

    public int? MathScore { get; set; }

    public int? ScienceScore { get; set; }

    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    public int? TotalScore
    {
        get;
        set;

    }

    public int StdId { get; set; }

    [ForeignKey("StdId")]
    public virtual Standard Standard { get; set; }
}
```

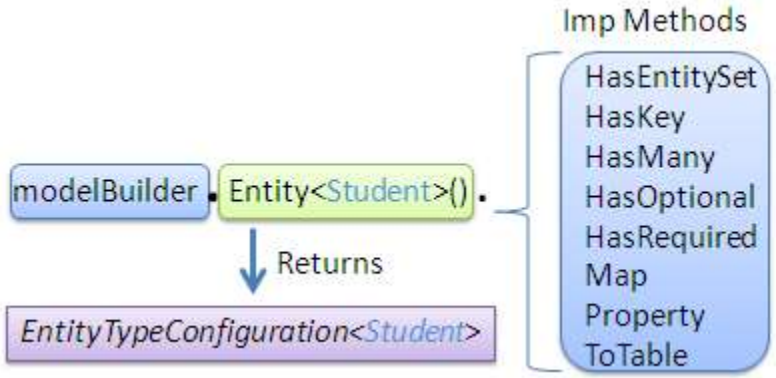
Fluent API

- Don’t find some attributes in DataAnnotation then you have to use Fluent API
- Example:

```
public class SchoolDBContext: DbContext
{
    public SchoolDBContext(): base("SchoolDBConnectionString")
    {
    }

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

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        //Configure domain classes using Fluent API here

        base.OnModelCreating(modelBuilder);
    }
}
```

Fluent API

- EntityTypeConfiguration Class:
- Class that allows configuration to be performed for an entity type in a model
- Obtained by calling Entity method of DbModelBuilder class



EntityTypeConfiguration Class

- EntityTypeConfiguration has following important methods

Method Name	Return Type	Description
HasKey<TKey>	EntityTypeConfiguration	Configures the primary key property(s) for this entity type.
HasMany<TTargetEntity>	ManyNavigationPropertyConfiguration	Configures a many relationship from this entity type.
HasOptional<TTargetEntity>	OptionalNavigationPropertyConfiguration	Configures an optional relationship from this entity type. Instances of the entity type will be able to be saved to the database without this relationship being specified. The foreign key in the database will be nullable.
HasRequired<TTargetEntity>	RequiredNavigationPropertyConfiguration	Configures a required relationship from this entity type. Instances of the entity type will not be able to be saved to the database unless this relationship is specified. The foreign key in the database will be non-nullable.
Ignore<TProperty>	Void	Excludes a property from the model so that it will not be mapped to the database.
Map	EntityTypeConfiguration	Allows advanced configuration related to how this entity type is mapped to the database schema.
Property<T>	StructuralTypeConfiguration	Configures a struct property that is defined on this type.
ToTable	Void	Configures the table name that this entity type is mapped to.

Fluent API classes

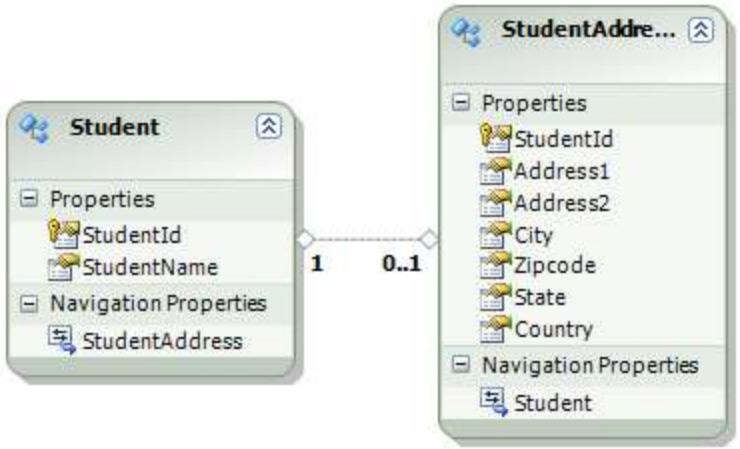
- Level 1:
  - DbModelBuilder Class
    - Main class
- Level 2:
  - EntityTypeConfiguration
    - Set relationship between entities
- Level 3:
  - ManyNavigarionConfiguration
  - OptionalNavigationPropertyConfiguration
  - RequiredNavigationPropertyConfiguration

- Level 1 and Level 2 classes can be used to configure relationship between the entities that will be mapped to database tables

- Level 3 & 4 can be used to configure additional mapping between the entities.

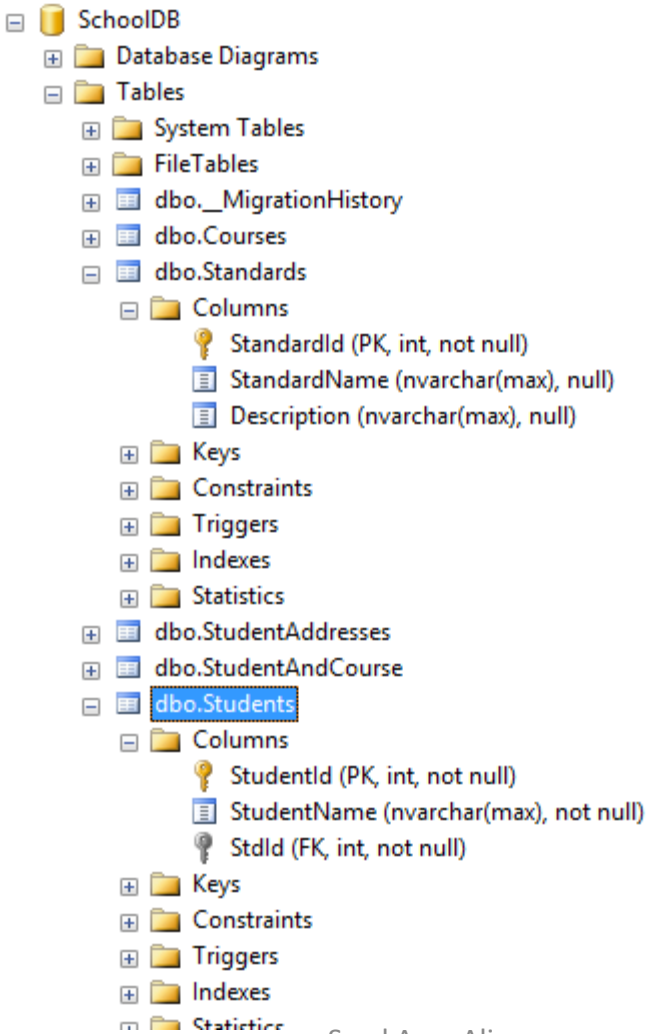
One-to-One Relationship

- When primary key of one table becomes PK & FK in another table

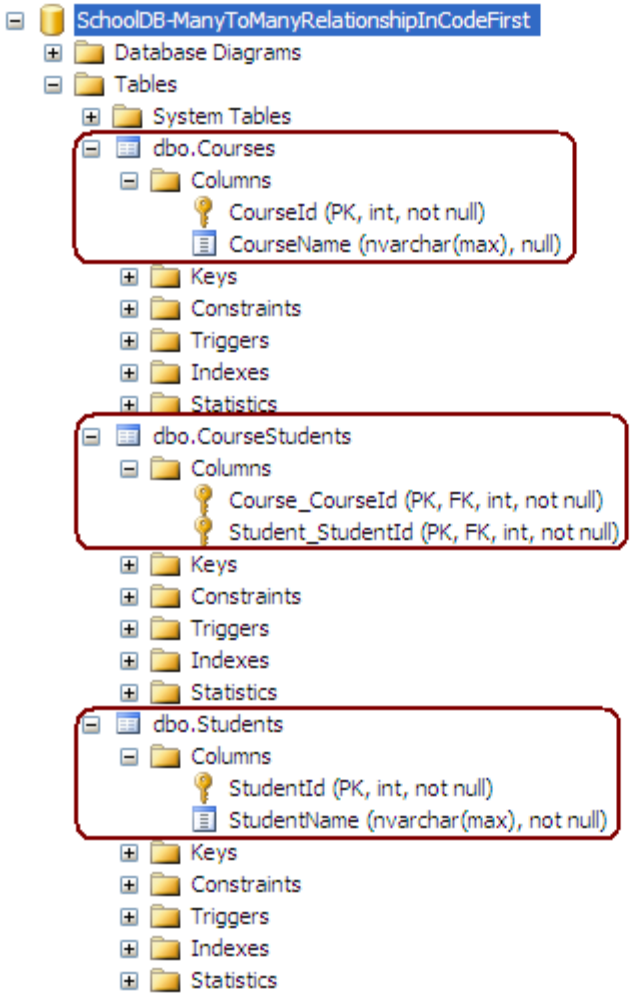


- Quick Start Example:

One-to-Many Relationship



Many-to-Many Relationship

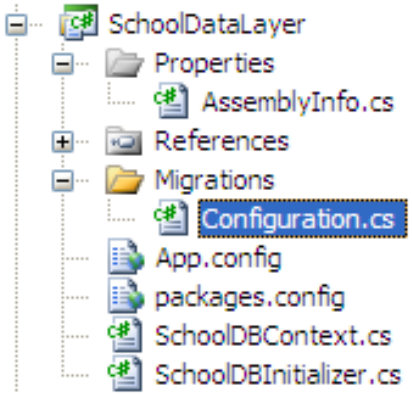


Migration in Code-First

- Database initialization strategies:
  - CreateDatabaseIfNotExists
  - DropCreateDatabaseIfModelChanges
  - DropCreateDatabaseAlways
- Problem:
  - Already have data
  - Existing Stored Procedures
  - Triggers
- Kinds of Migration:
  - Automated Migration
  - Code based Migration

Automated Migration

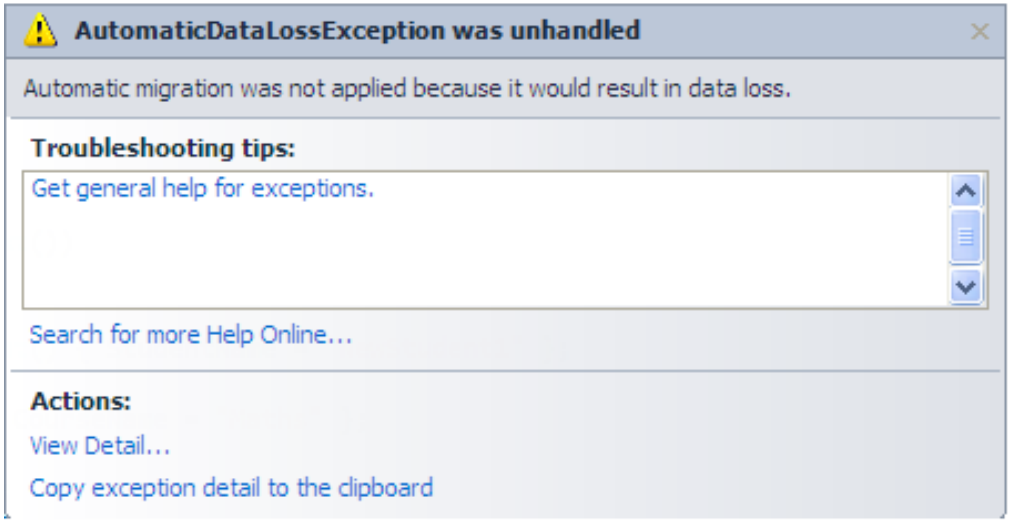
- Don’t have to maintain database migration manually in code file
- Process:
  - Comman: enable-migrations –EnableAutomaticMigration:$true



  - Need to set the database initializer
  - Run the application and see the created database
  - System table __MigrationHistory

Automated Migration

- Oops … If your Database tables already have data?



- AutomaticMigrationDataLossAllowed = true
- get-help enable-migrations

Code-based Migration

- Commands:
  - **Add-migration**
    - Scaffold the next migration for the changes you have made
  - Update-database
    - Apply pending changes to the database
    - –verbose to see what's going on in the database
- Rollback Database change
  - Update-database -TargetMigration:"First School DB schema"
- Get All Migrations
  - Use "get-migration" command to see what migration have been applied.

References

- http://www.entityframeworktutorial.net
- http://weblogs.asp.net/scottgu/archive/2010/07/16/code-first-development-with-entity-framework-4.aspx
- http://msdn.microsoft.com/en-us/data/jj591620.aspx
- http://blogs.msdn.com/b/efdesign/archive/2010/06/01/conventions-for-code-first.aspx

Thank you ☺