# EF 6
# DB First

# Introduction

- We will learn how to use Entity Framework with the existing database of the application.
- We'll start by creating an Entity Data Model (EDM) from the existing database and will see how to save and query data using Entity Framework.
- Entity Framework uses EDM for all the database-related operations.
- Entity Data Model is a model that describes entities and the relationships between them.
- Let's create a simple EDM for the School database using Visual Studio and Entity Framework.

# Install Database in SQL Server

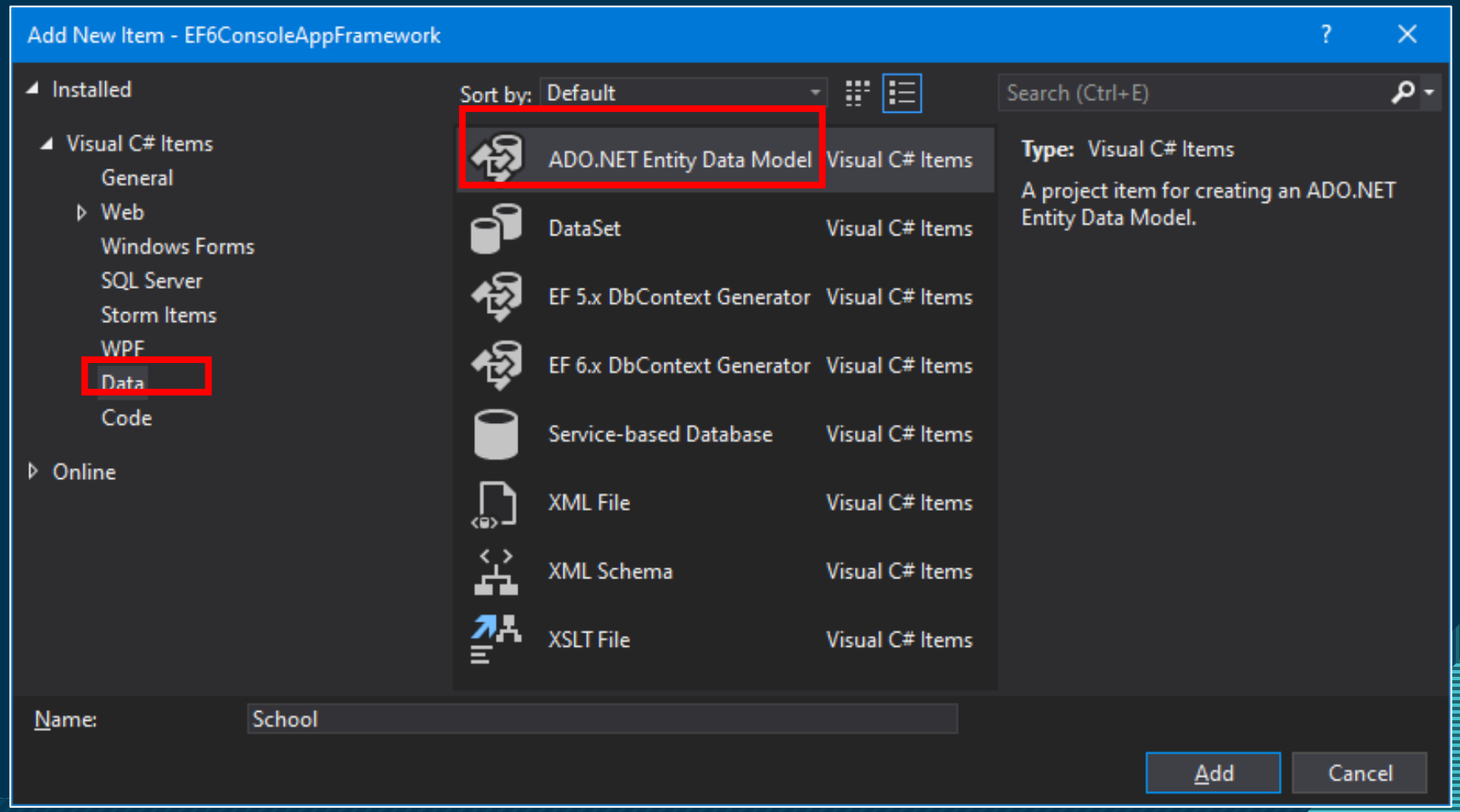
- Download the School database for this example from:

  SLATE → Content → Module 6-Database Programming → Sample Databases.

- Run the query to create the database.
  - Make sure to keep the database name SchoolDB

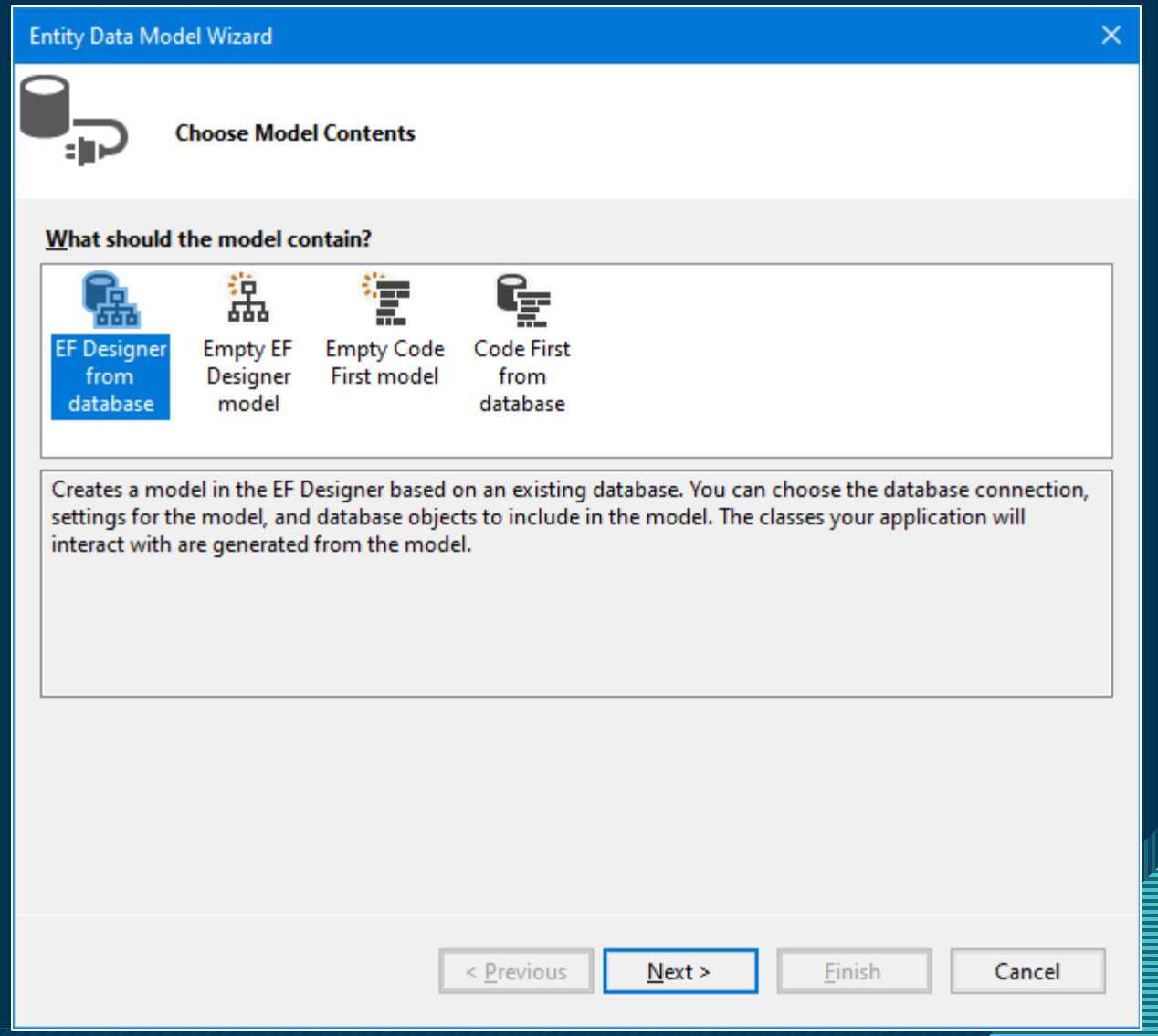
# Creating an Entity Data Model

- Create a new project WPF (.NET Framework) in Visual Studio.
  - Make sure its .NET Framework, not .NET Core.
- Once the project is ready, right-click on the project name in Solution Explorer and select Add → New Item….
- Select Data from the left panel and select ADO.NET Entity Data Model.
- Provide an appropriate name to the EDM (School in this case) and click the Add button.



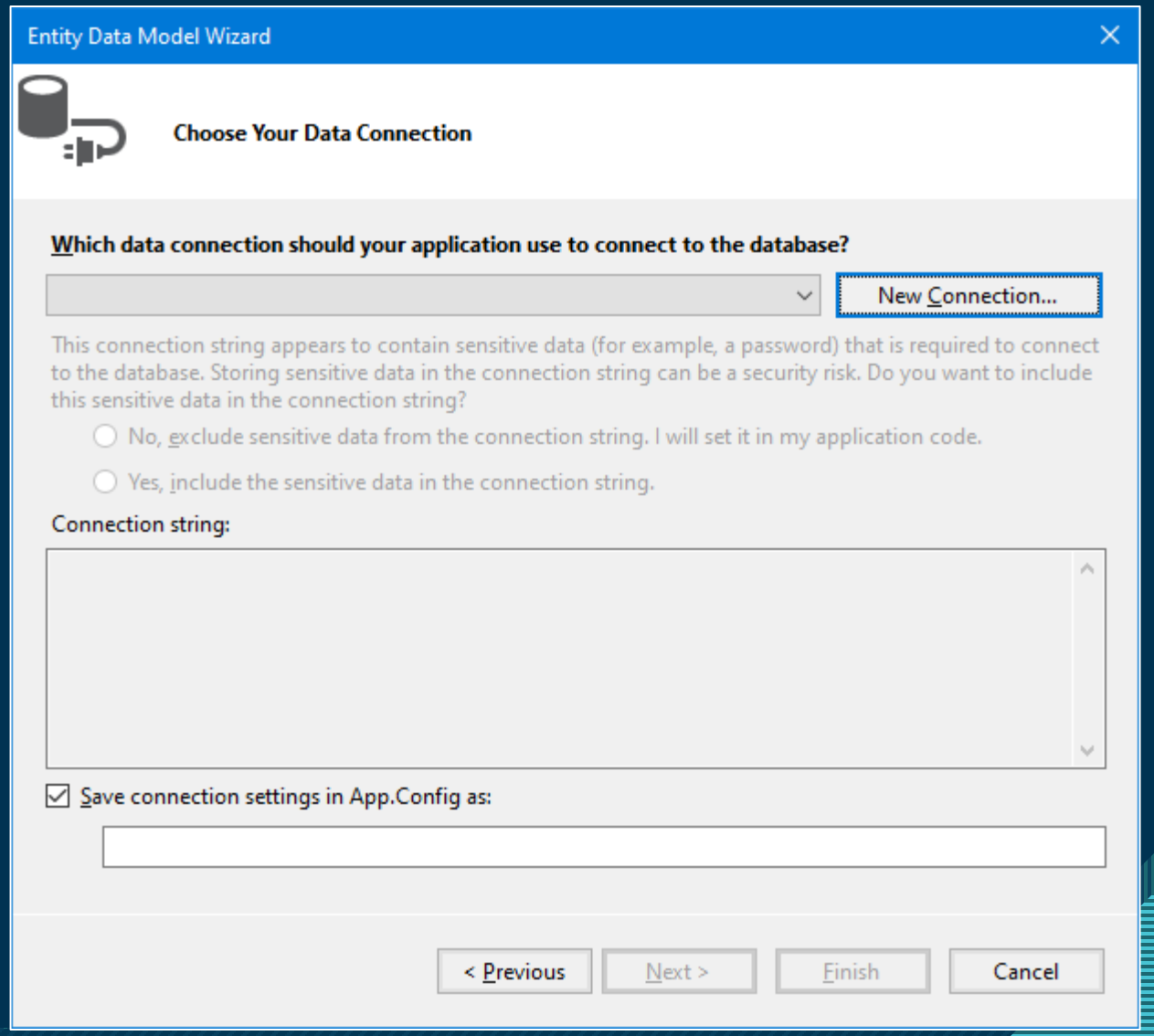
# Creating an Entity Data Model

- Entity Data Model Wizard opens with four options to select from:
  - EF Designer from database for the database-first approach
  - Empty EF Designer model for the model-first approach
  - Empty Code First model for Code-First approach.
  - Code First from database
- We are using database-first approach here, so select the EF Designer from database option and click Next.



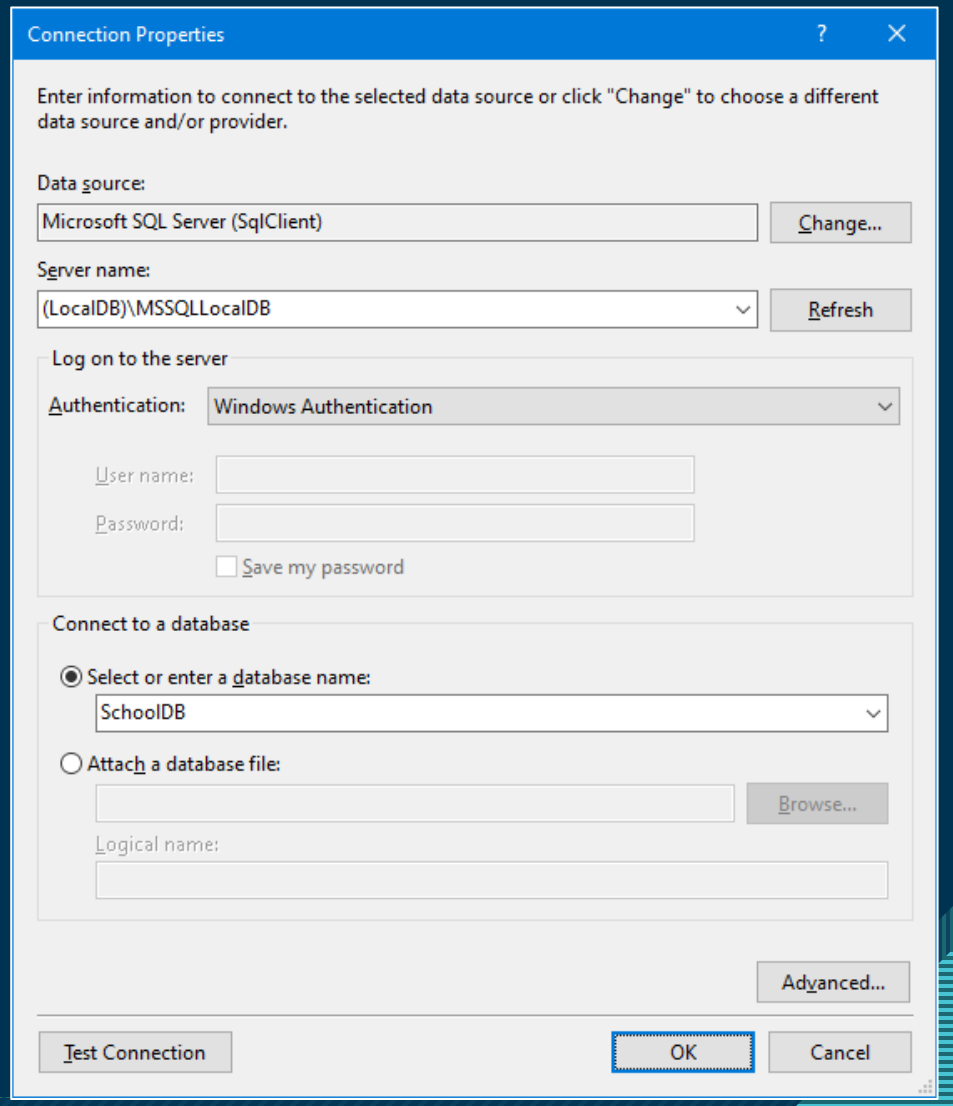
# Creating an Entity Data Model

- You need to create a connection with your existing database.
- If this is the first time you are creating an EDM for your database, then you need to create a new connection by clicking on the New Connection... button.



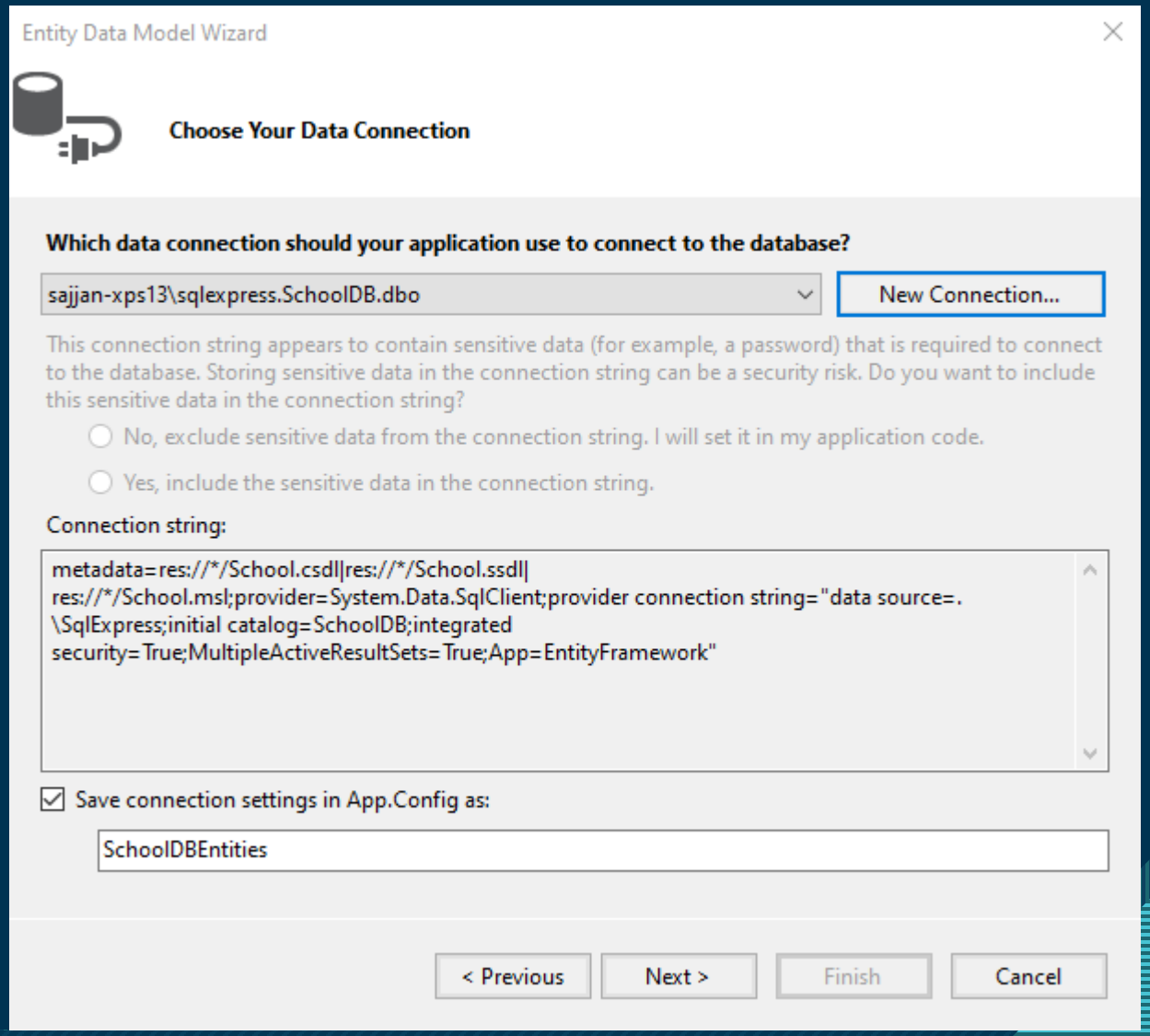
# Creating an Entity Data Model

- In the Connection Properties popup, provide the server name (LocalDB)\MSSQLLocalDB and select your database name (SchoolDB) and click the OK button.



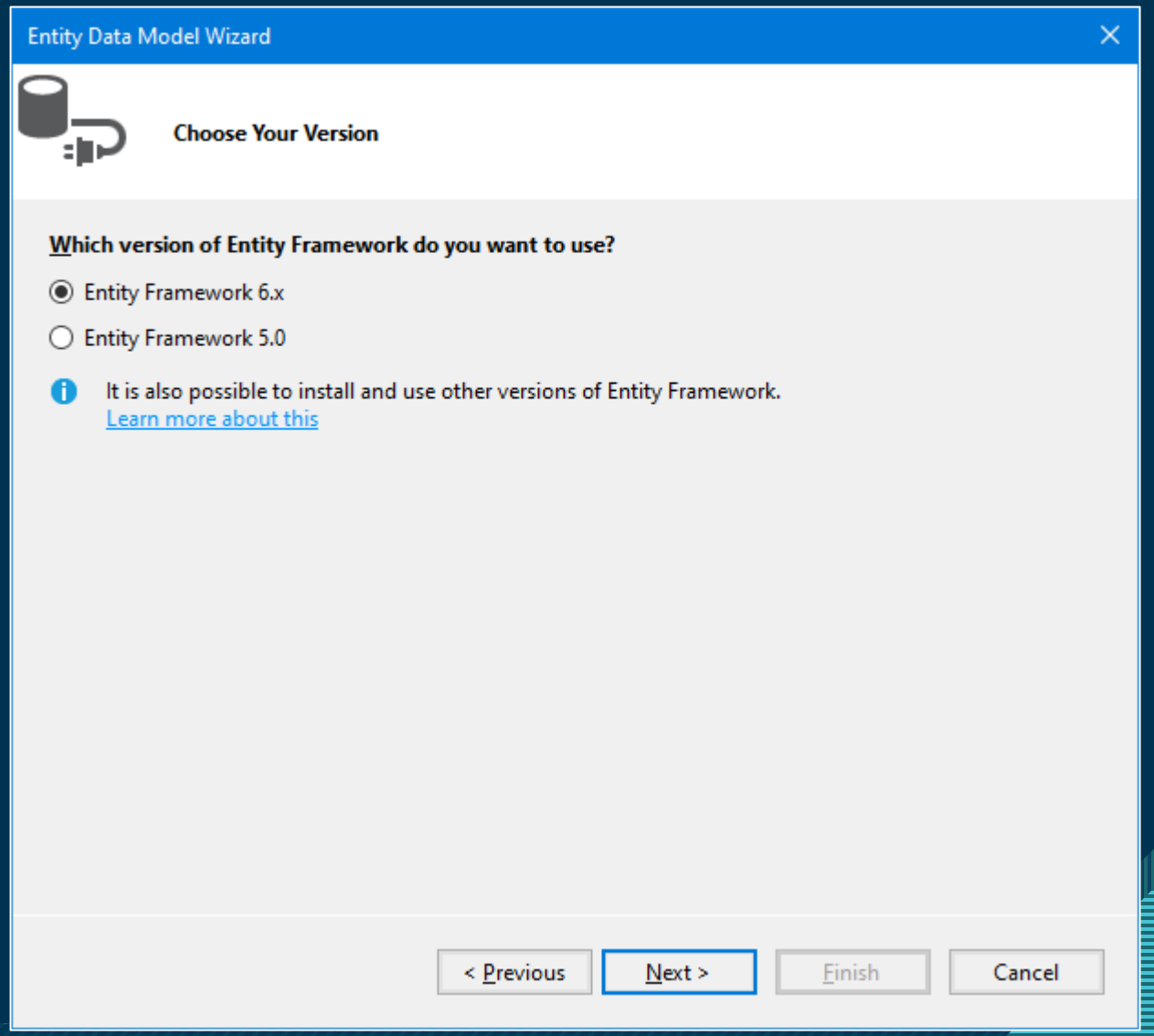
# Creating an Entity Data Model

- This will add a connection string to your app.config file with the `<DB name>Entities` name.
- Click Next after you set up your DB connection.



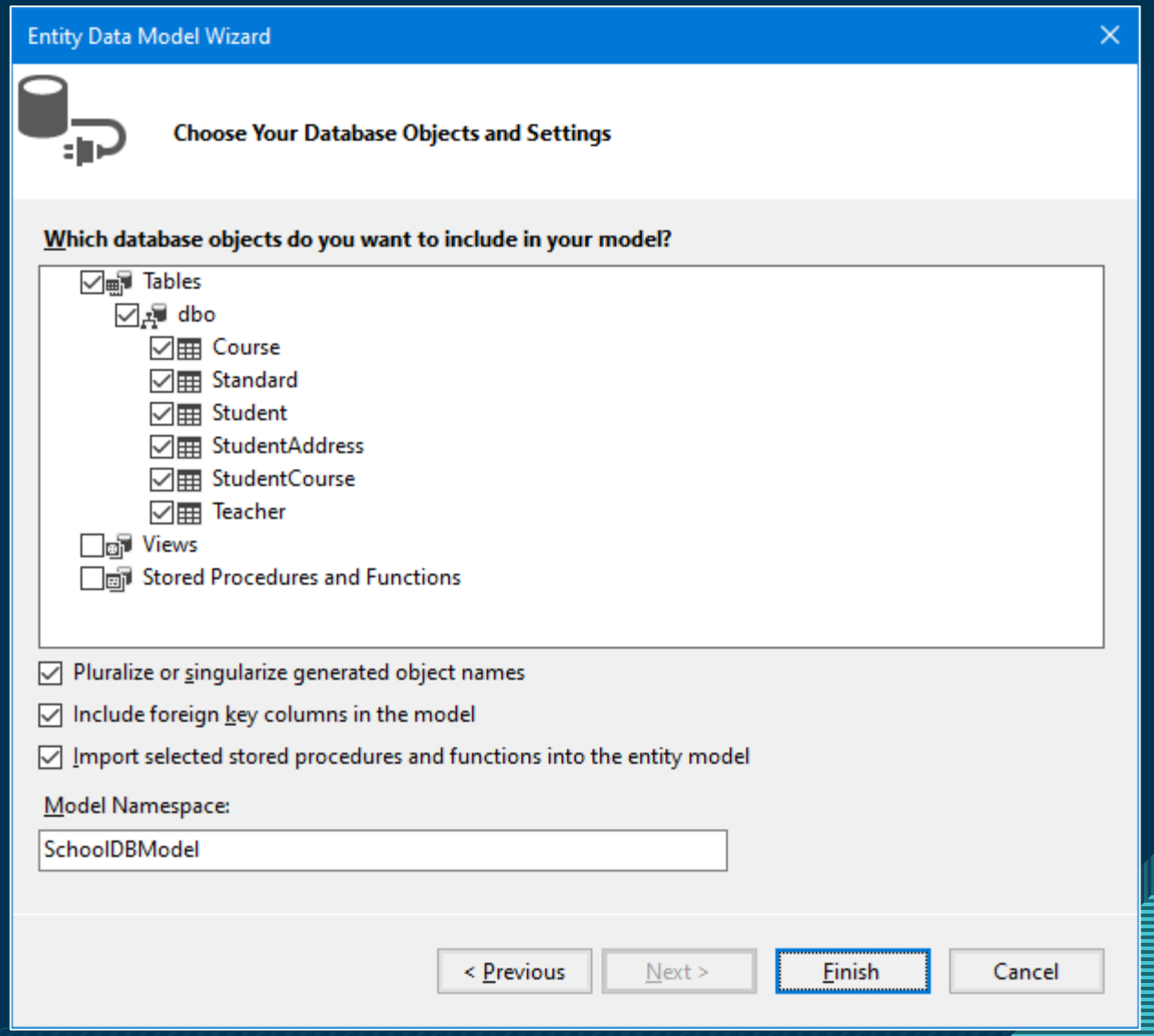
# Creating an Entity Data Model

- In this step, you need to choose the version of Entity Framework.
- We will use Entity Framework 6.0.
- Click Next.



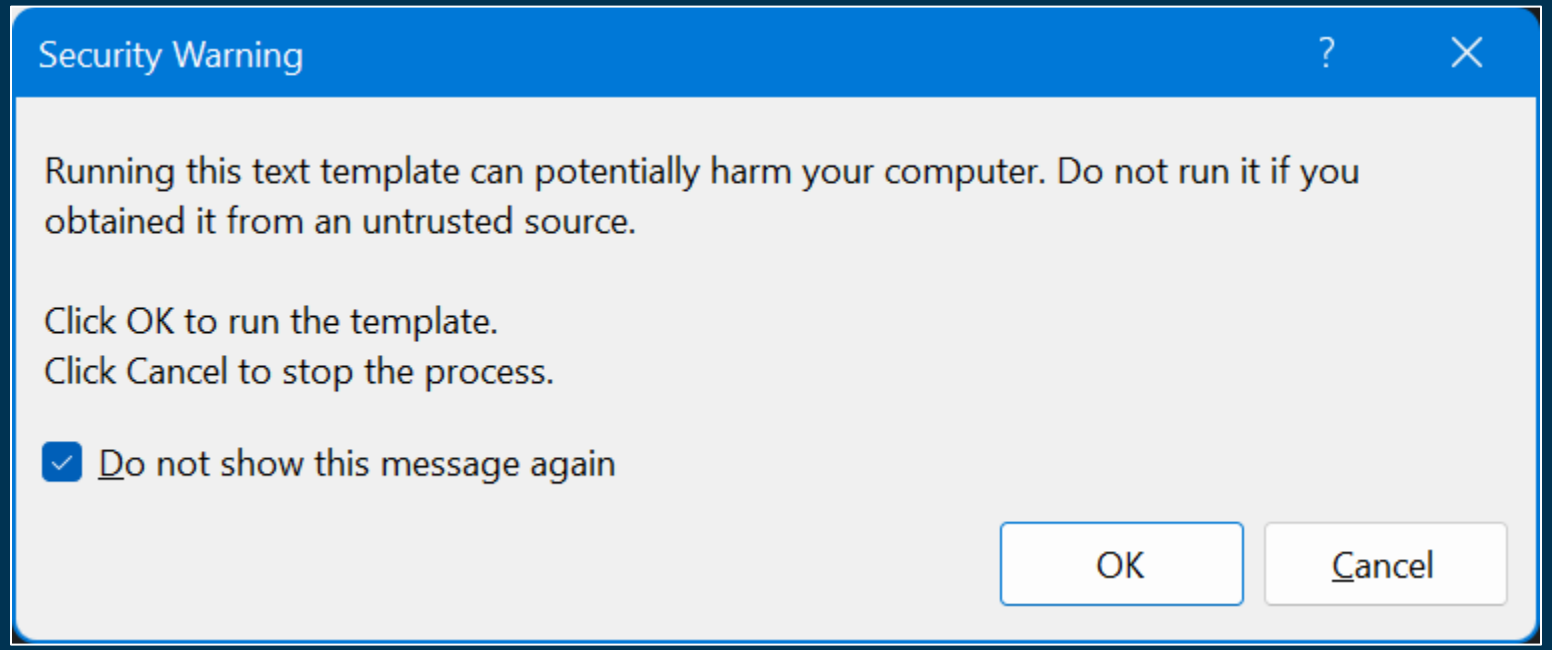
# Creating an Entity Data Model

- This step will display all the Tables, Views and Stored Procedures (SP) in the database.
- Select the Tables, Views and SPs you want, keep the default checkboxes selected and click Finish.
- You can change the Model Namespace if you want.
- After clicking on Finish, a School.edmx file will be added into your project.

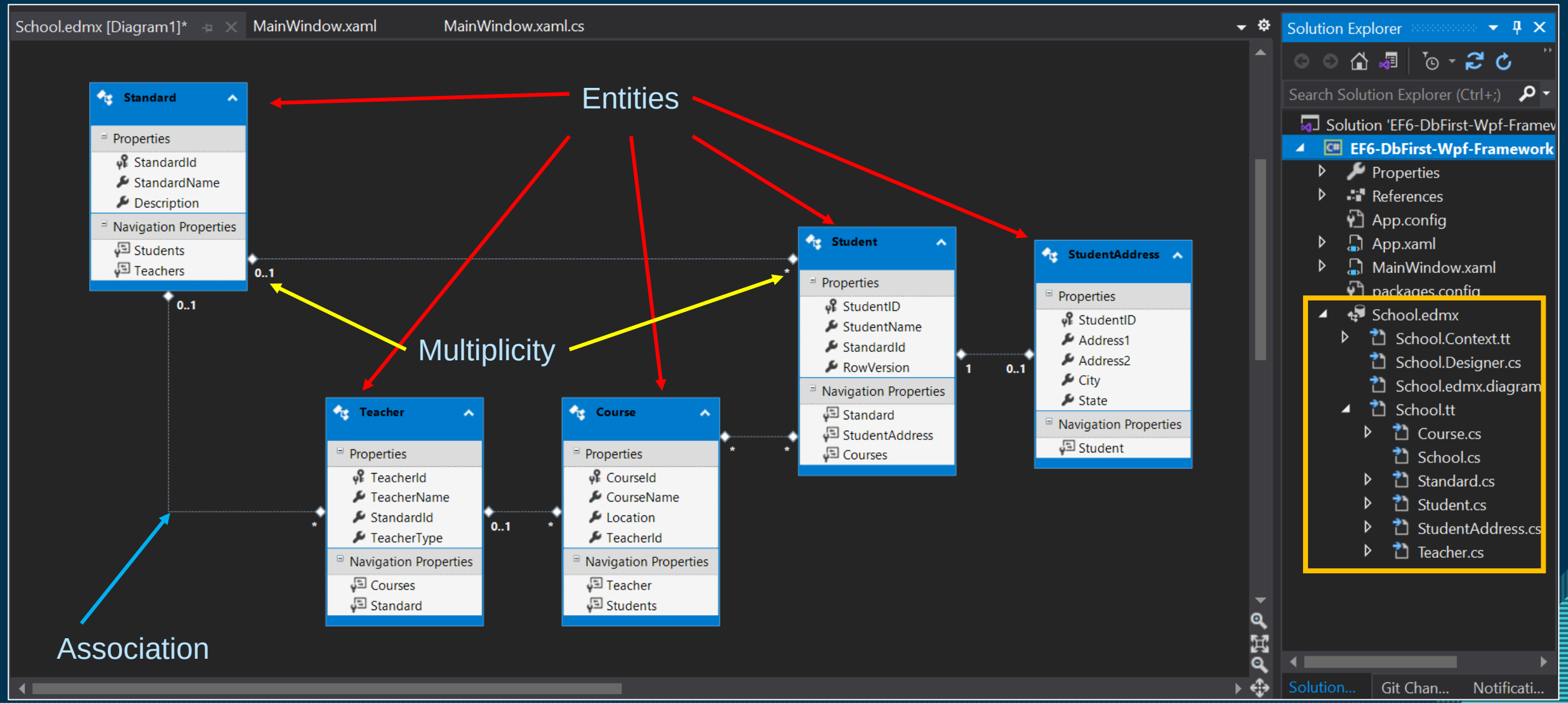


# Creating an Entity Data Model

- If you see this window popping up, you can check Do not show… and click OK.



# Creating an Entity Data Model

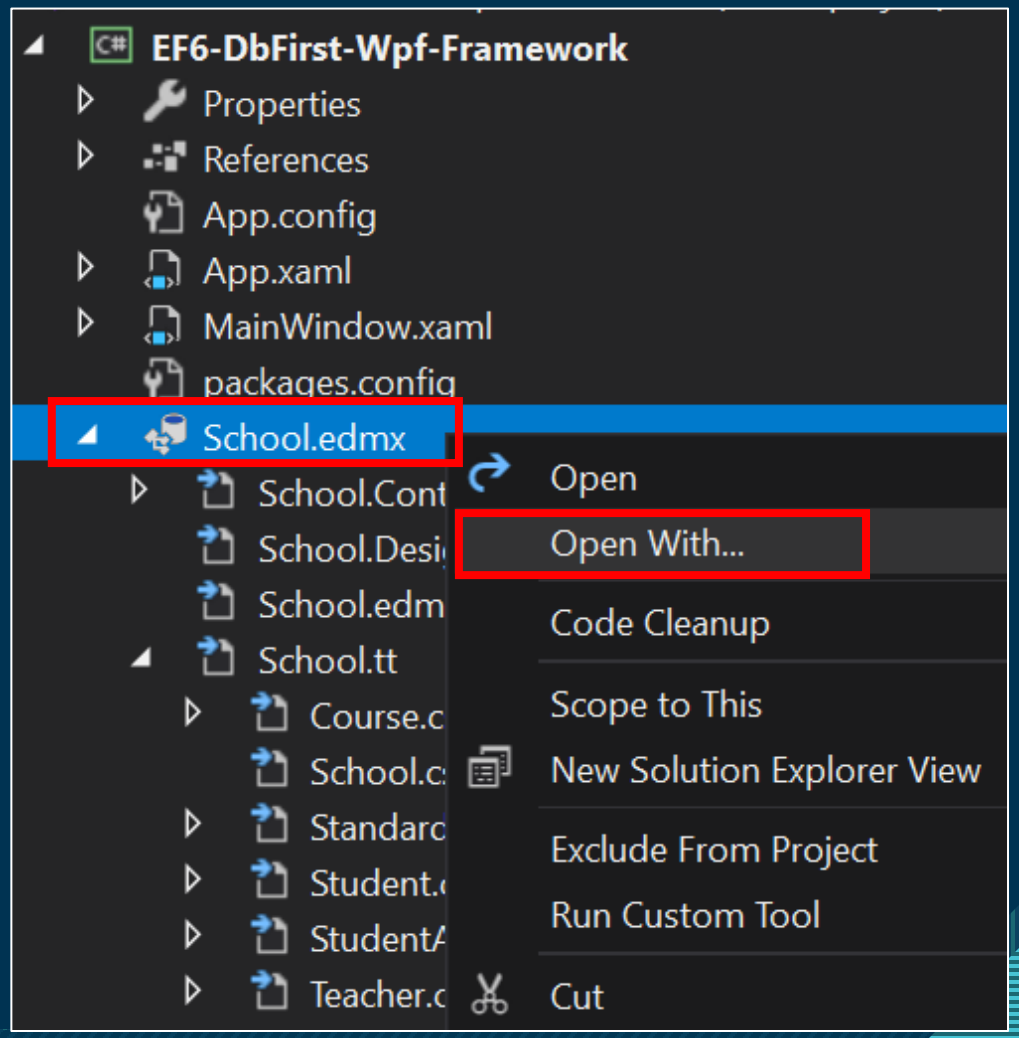


# Creating an Entity Data Model

- You can open an EDM designer in XML view where you can see all the three parts of the EDM:
  - Conceptual schema (CSDL)
  - Storage schema (SSDL)
  - Mapping schema (MSL)

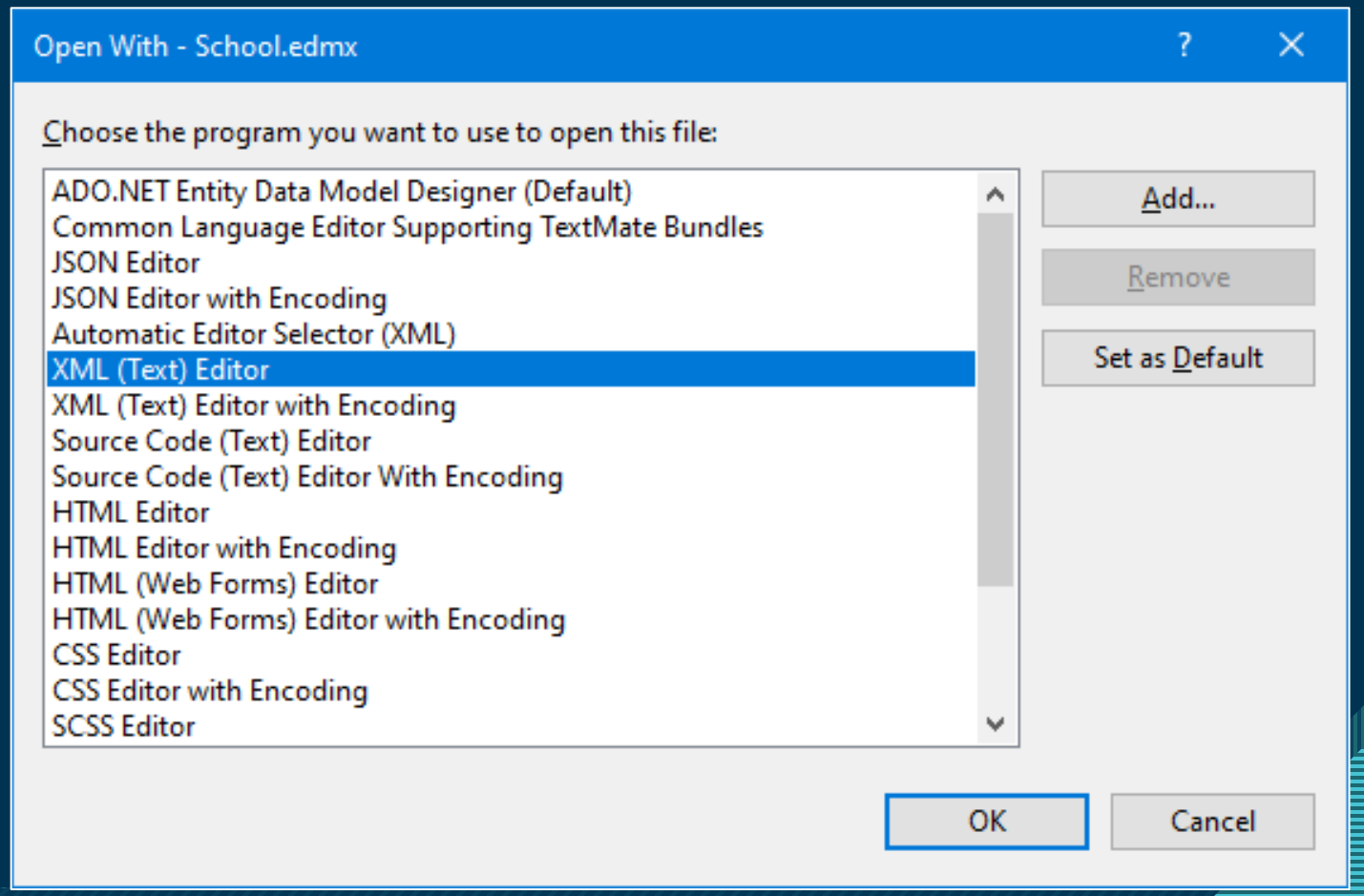
  together in XML view.

- Right click on School.edmx, select Open With….
- This will open a popup window.



# Creating an Entity Data Model

- Select XML (Text) Editor in the popup window.
- Visual Studio cannot display the model in Design View and in XML Format at the same time, so you will see a message asking whether it's OK to close the Design View of the model, click Yes.
- This will open the XML Format view.



# Creating an Entity Data Model

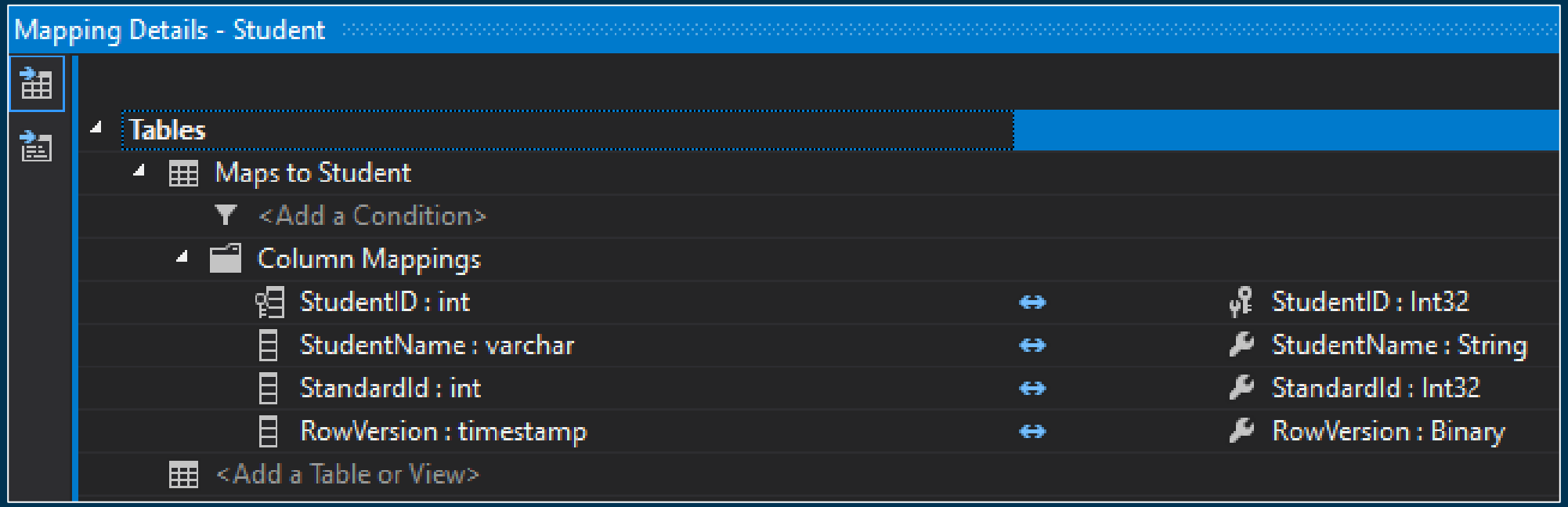
- You can see the following XML view by toggling all outlining as shown below.

School.edmx

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <edmx:Edmx Version="3.0" xmlns:edmx="http://schemas.microsoft.com/ado/2009/11/edmx">
3     <!-- EF Runtime content -->
4     <edmx:Runtime>
5       <!-- SSDL content -->
6       <edmx:StorageModels>...</edmx:StorageModels>
177     <!-- CSDL content -->
178     <edmx:ConceptualModels>...</edmx:ConceptualModels>
324     <!-- C-S mapping content -->
325     <edmx:Mappings>...</edmx:Mappings>
389   </edmx:Runtime>
390   <!-- EF Designer content (DO NOT EDIT MANUALLY BELOW HERE) -->
391   <Designer xmlns="http://schemas.">...</Designer>
409 </edmx:Edmx>
```

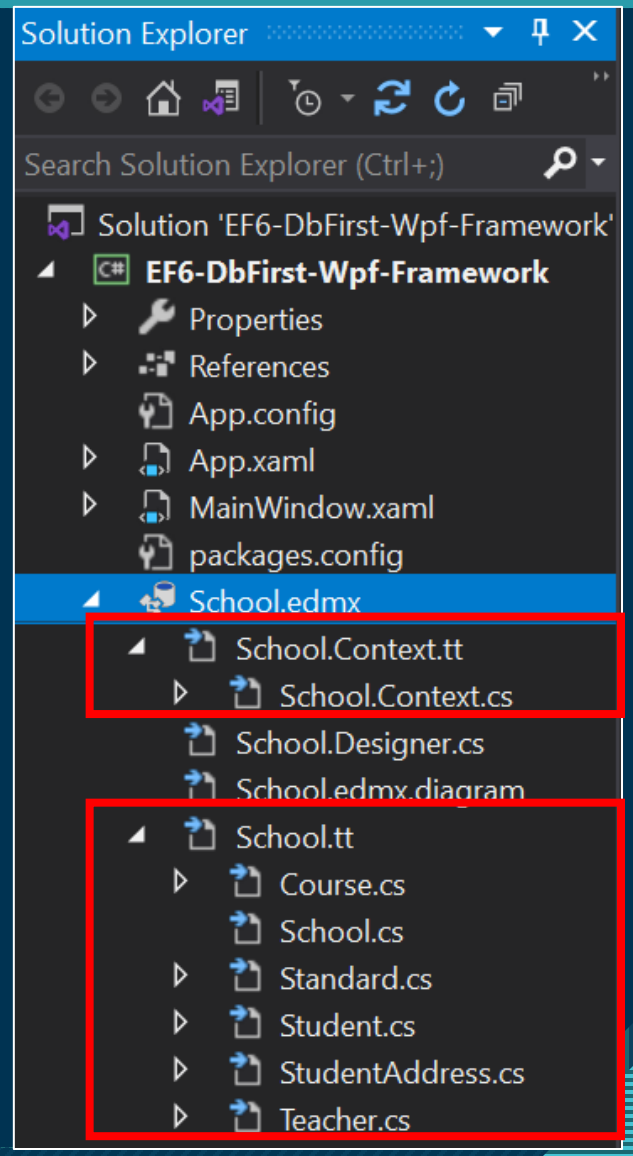
# Entity-Table Mapping

- Each entity in EDM is mapped with the database table.
- You can check the entity-table mapping by right clicking on any entity in the EDM designer and selecting Table Mapping.



# Context and Entity Classes

- Every Entity Data Model generates a context and an entity class for each database table.
- Expand the .edmx file in the solution explorer and open two important files:
  - `<EDM Name>.Context.tt`
  - `<EDM Name>.tt`



# School.Context.tt

- This T4 (Text template transformation toolkit) template file generates a context class whenever you change the Entity Data Model (.edmx file).
- You can see the context class file by expanding School.Context.tt.
- The context class resides in the <EDM Name>.context.cs file.
- The default context class name is `<DB Name>Entities`.
- For example, the context class name for `SchoolDB` is `SchoolDBEntities` and derived from the `DBContext` class.

School.Context.cs

EF6ConsoleAppFramework | EF6ConsoleAppFramework.SchoolDBEntities

```
10  namespace EF6ConsoleAppFramework
11  {
12      using System;
13      using System.Data.Entity;
14      using System.Data.Entity.Infrastructure;
15
16      public partial class SchoolDBEntities : DbContext
17      {
18          public SchoolDBEntities()
19              : base("name=SchoolDBEntities")
20          {
21          }
22
23          protected override void OnModelCreating(DbModelBuilder modelBuilder)
24          {
25              throw new UnintentionalCodeFirstException();
26          }
27
28          public virtual DbSet<Course> Courses { get; set; }
29          public virtual DbSet<Standard> Standards { get; set; }
30          public virtual DbSet<Student> Students { get; set; }
31          public virtual DbSet<StudentAddress> StudentAddresses { get; set; }
32          public virtual DbSet<Teacher> Teachers { get; set; }
33      }
34  }
```

# School.tt

- School.tt is a T4 template file that generates entity classes for each DB table.
- Entity classes are POCO (Plain Old CLR Object) classes.
- This code snippet shows the `Student` entity.

```
7 references
public partial class Student
{
    [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
      "CA2214:DoNotCallOverridableMethodsInConstructors")]
    0 references
    public Student()
    {
        this.Courses = new HashSet<Course>();
    }

    0 references
    public int StudentID { get; set; }
    0 references
    public string StudentName { get; set; }
    0 references
    public Nullable<int> StandardId { get; set; }
    0 references
    public byte[] RowVersion { get; set; }

    0 references
    public virtual Standard Standard { get; set; }
    0 references
    public virtual StudentAddress StudentAddress { get; set; }
    [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
      "CA2227:CollectionPropertiesShouldBeReadOnly")]
    1 reference
    public virtual ICollection<Course> Courses { get; set; }
}
```

# DbContext in Entity Framework

- As you have seen previously, EDM includes the `SchoolDBEntities` class, which is derived from the `System.Data.Entity.DbContext` class, as shown here.
- The class that derives `DbContext` is called context class in entity framework.

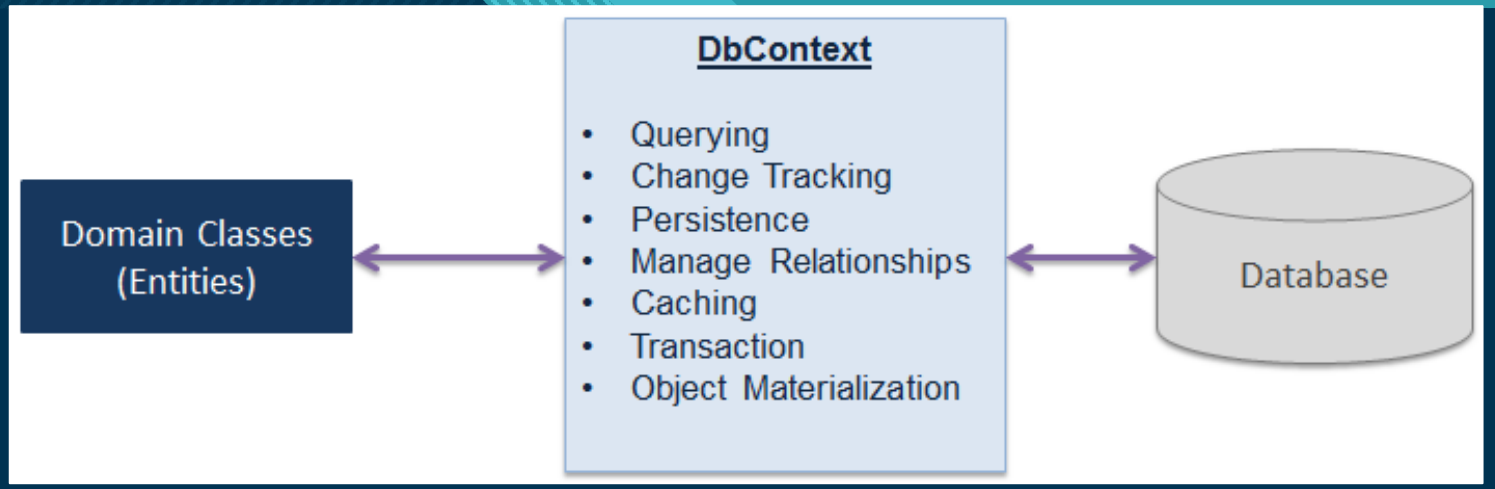
School.Context.cs

EF6ConsoleAppFramework | EF6ConsoleAppFramework.SchoolDBEntities

```
10  namespace EF6ConsoleAppFramework
11  {
12      using System;
13      using System.Data.Entity;
14      using System.Data.Entity.Infrastructure;
15
16      public partial class SchoolDBEntities : DbContext
17      {
18          public SchoolDBEntities()
19              : base("name=SchoolDBEntities")
20          {
21          }
22
23          protected override void OnModelCreating(DbModelBuilder modelBuilder)
24          {
25              throw new UnintentionalCodeFirstException();
26          }
27
28          public virtual DbSet<Course> Courses { get; set; }
29          public virtual DbSet<Standard> Standards { get; set; }
30          public virtual DbSet<Student> Students { get; set; }
31          public virtual DbSet<StudentAddress> StudentAddresses { get; set; }
32          public virtual DbSet<Teacher> Teachers { get; set; }
33      }
34  }
```

# DbContext in Entity Framework

- `DbContext` is an important class in Entity Framework.
- It is a bridge between your domain or entity classes and the database.
- `DbContext` is the primary class that is responsible for interacting with the database.




- It is responsible for the following activities:
  - **Querying:** Converts LINQ-to-Entities queries to SQL query and sends them to the database.
  - **Change Tracking:** Keeps track of changes that occurred on the entities after querying from the database.
  - **Persisting Data:** Performs the Insert, Update and Delete operations to the database, based on entity states.
  - **Caching:** Provides first level caching by default. It stores the entities which have been retrieved during the lifetime of a context class.
  - **Manage Relationship:** Manages relationships using CSDL, MSL and SSDL in Db-First or Model-First approach and using fluent API configurations in Code-First approach.
  - **Object Materialization:** Converts raw data from the database into entity objects.

# DbSet in Entity Framework

- The `DbSet` class represents an entity set that can be used for create, read, update, and delete operations.
- The context class (derived from `DbContext`) must include the `DbSet` type properties for the entities which map to database tables and views.

School.Context.cs

EF6ConsoleAppFramework | EF6ConsoleAppFramework.SchoolDBEntities

```
10  namespace EF6ConsoleAppFramework
11  {
12      using System;
13      using System.Data.Entity;
14      using System.Data.Entity.Infrastructure;
15
            1 reference
16      public partial class SchoolDBEntities : DbContext
17      {
                0 references
18          public SchoolDBEntities()
19              : base("name=SchoolDBEntities")
20          {
21          }
22
                0 references
23          protected override void OnModelCreating(DbModelBuilder modelBuilder)
24          {
25              throw new UnintentionalCodeFirstException();
26          }
27
                0 references
28          public virtual DbSet<Course> Courses { get; set; }
                0 references
29          public virtual DbSet<Standard> Standards { get; set; }
                0 references
30          public virtual DbSet<Student> Students { get; set; }
                0 references
31          public virtual DbSet<StudentAddress> StudentAddresses { get; set; }
                0 references
32          public virtual DbSet<Teacher> Teachers { get; set; }
33      }
34  }
```

# DbSet in Entity Framework

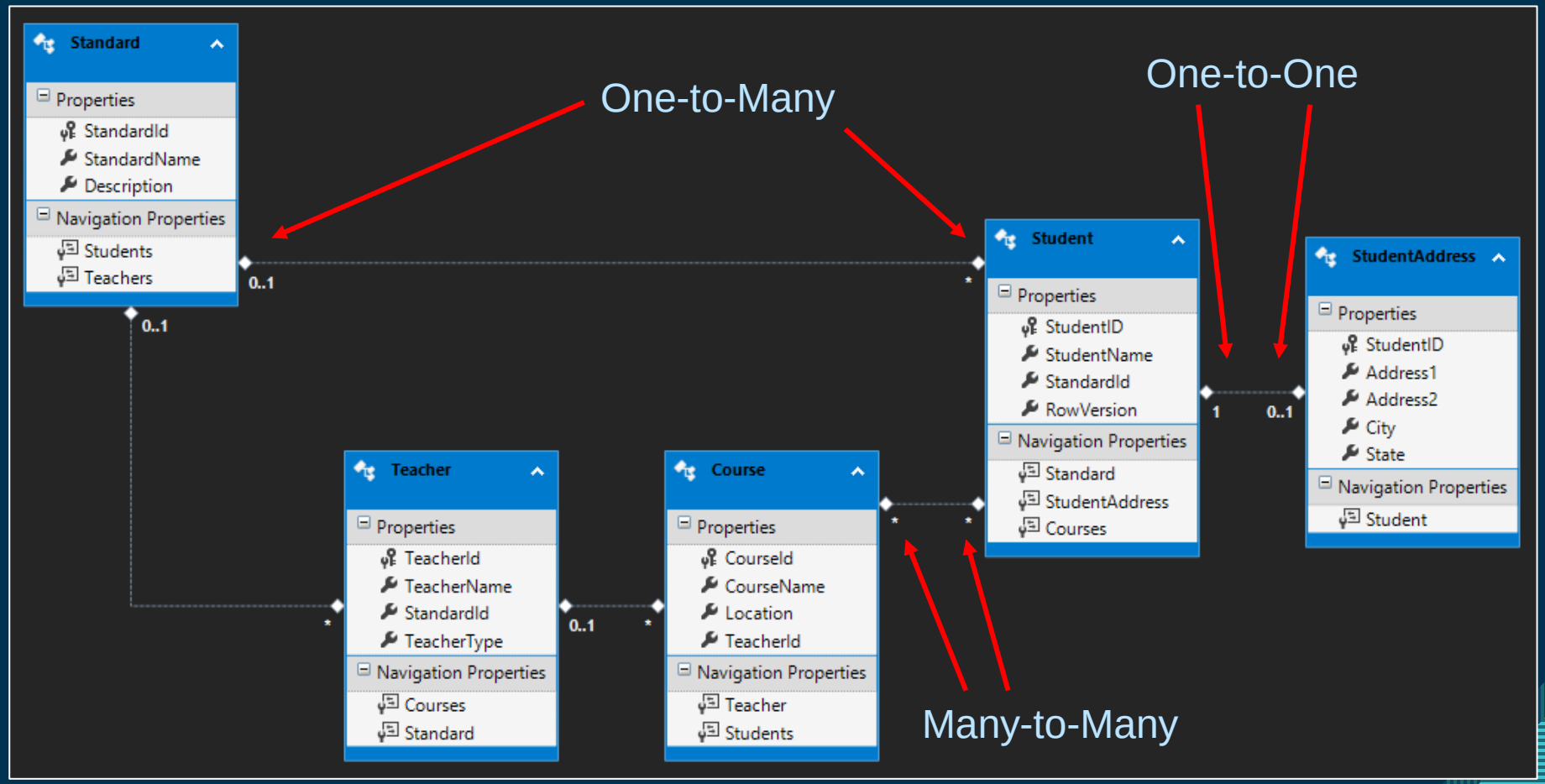
- The following table lists important methods of the `DbSet` class:

| Method Name | Description |
|---|---|
| `Add` | Adds the given entity to the context with the Added state. When the changes are saved, the entities in the Added states are inserted into the database. After the changes are saved, the object state changes to Unchanged.<br>Example: `dbcontext.Students.Add(studentEntity);` |
| `Attach(Entity)` | Attaches the given entity to the context in the Unchanged state<br>Example: `dbcontext.Students.Attach(studentEntity);` |
| `Create` | Creates a new instance of an entity for the type of this set. This instance is not added or attached to the set. The instance returned will be a proxy if the underlying context is configured to create proxies and the entity type meets the requirements for creating a proxy.<br>Example: `var newStudentEntity = dbcontext.Students.Create();` |
| `Find(int)` | Uses the primary key value to find an entity tracked by the context. Note that the Find also returns entities that have been added to the context but have not yet been saved to the database.<br>Example: `Student studEntity = dbcontext.Students.Find(1);` |
| `Include` | Returns the included non-generic LINQ to Entities query against a DbContext. (Inherited from DbQuery)<br>Example:<br>`var studentList = dbcontext.Students.Include("StudentAddress").ToList<Student>();`<br>`var studentList = dbcontext.Students.Include(s => s.StudentAddress).ToList<Student>();` |
| `Remove` | Marks the given entity as Deleted. When the changes are saved, the entity is deleted from the database. The entity must exist in the context in some other state before this method is called.<br>Example: `dbcontext.Students.Remove(studentEntity);` |

# Relationships between Entities in Entity Framework

- Entity framework supports three types of relationships, same as database:

1. One-to-One
2. One-to-Many
3. Many-to-Many



# One-to-One Relationship

- `Student` and `StudentAddress` have a One-to-One relationship (zero or one).
- A student can have only one or zero addresses.
- Entity framework adds the `Student` reference navigation property into the `StudentAddress` entity and the `StudentAddress` navigation entity into the `Student` entity.
- Also, the `StudentAddress` entity has both `StudentId` property as PrimaryKey and ForeignKey, which makes it a one-to-one relationship.

# One-to-One Relationship

```
public partial class Student
{
    public Student()
    {
        this.Courses = new HashSet<Course>();
    }

    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public Nullable<int> StandardId { get; set; }
    public byte[] RowVersion { get; set; }

    public virtual StudentAddress StudentAddress { get; set; }
}
```

```
public partial class StudentAddress
{
    public int StudentID { get; set; }
    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public virtual Student Student { get; set; }
}
```

# One-to-Many Relationship

- The `Standard` and `Student` entities have a One-to-Many relationship marked by multiplicity where 1 is for One and * is for Many.
- This means that Standard can have many Students whereas Student can associate with only one Standard.
- To represent this, the `Standard` entity has the collection navigation property `Students` (please notice that it's plural), which indicates that one Standard can have a collection of Students (many students).
- And the `Student` entity has a `Standard` navigation property (reference property), which indicates that Student is associated with one Standard.
- Also, it contains the `StandardId` foreign key (PK in `Standard` entity).
- This makes it a One-to-Many relationship.

# One-to-Many Relationship

```
public partial class Standard
{
    public Standard()
    {
        this.Students = new HashSet<Student>();
    }

    public int StandardId { get; set; }
    public string StandardName { get; set; }
    public string Description { get; set; }

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

```
public partial class Student
{
    public Student()
    {
        this.Courses = new HashSet<Course>();
    }

    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public Nullable<int> StandardId { get; set; }
    public byte[] RowVersion { get; set; }

    public virtual Standard Standard { get; set; }
}
```

# Many-to-Many Relationship

- The `Student` and `Course` have a Many-to-Many relationship marked by * multiplicity.
- It means one Student can enroll for many Courses and, one Course can be taught to many Students.
- The database includes the StudentCourse junction table which includes the primary key of both the tables (Student and Course tables).
- The `Student` entity includes the collection navigation property `Courses` and `Course` entity includes the collection navigation property `Students` to represent a many-to-many relationship between them.

# Many-to-Many Relationship

```
public partial class Student
{
    public Student()
    {
        this.Courses = new HashSet<Course>();
    }

    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public Nullable<int> StandardId { get; set; }
    public byte[] RowVersion { get; set; }

    public virtual ICollection<Course> Courses { get; set; }
}
```

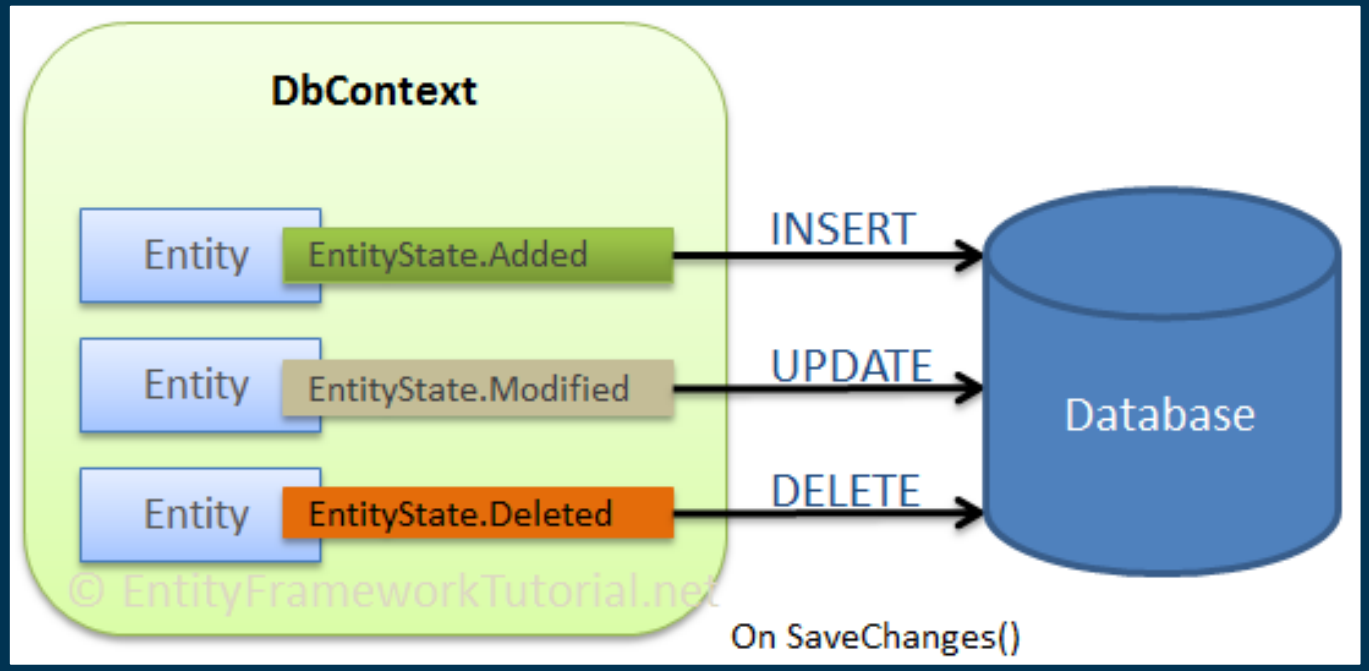
```
public partial class Course
{
    public Course()
    {
        this.Students = new HashSet<Student>();
    }

    public int CourseId { get; set; }
    public string CourseName { get; set; }
    public System.Data.Entity.Spatial.DbGeography Location { get; set; }

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

# Saving Data

- Saving entity data is an easy task because the context automatically tracks the changes that happened on the entity during its lifetime.
- An entity which contains data in its scalar property will be either inserted, updated or deleted, based on its `EntityState`.
- In the Entity Framework, there are two persistence scenarios to save an entity data:
  - Connected
  - Disconnected
- In the connected scenario, the same instance of `DbContext` is used in retrieving and saving entities, whereas this is different in the disconnected scenario.

# Saving Data

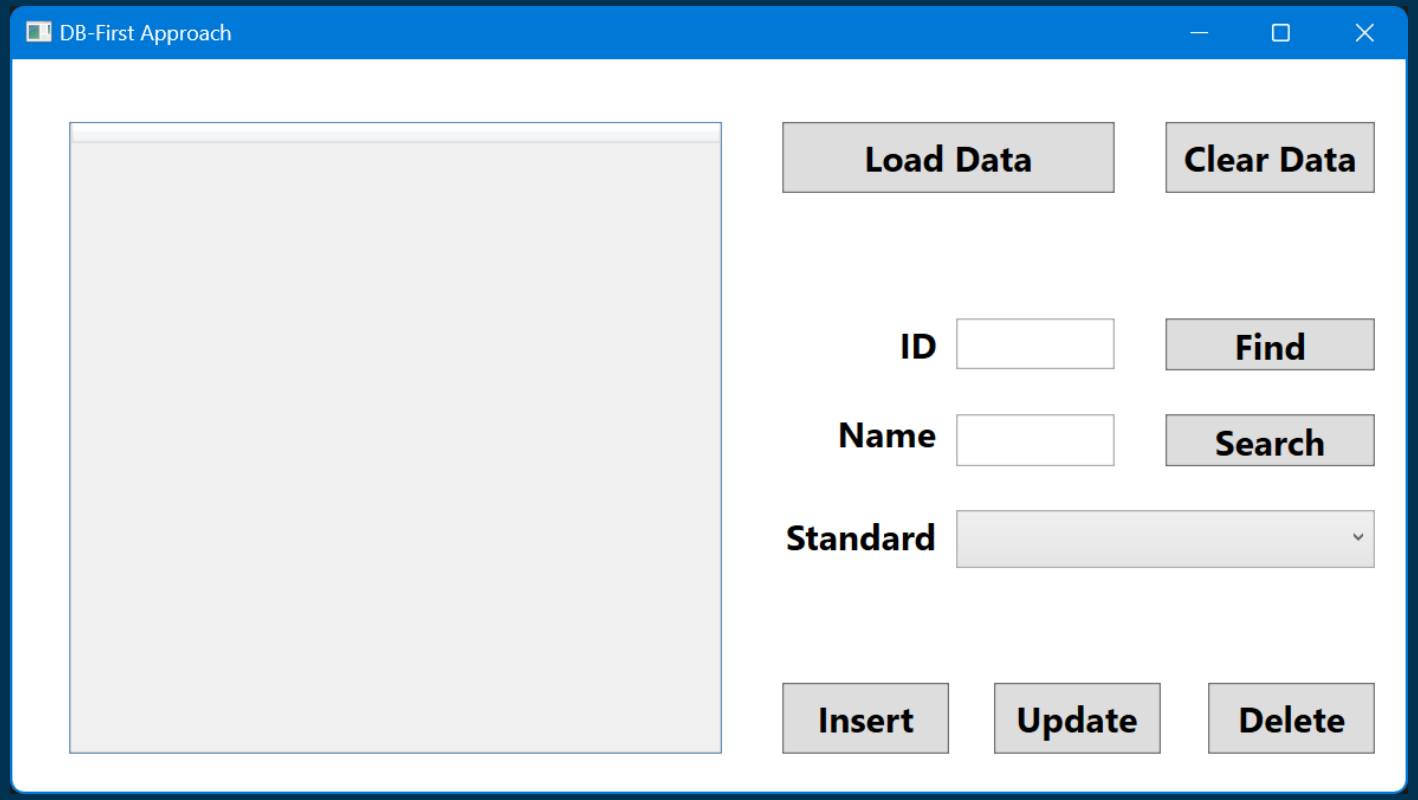
- This image illustrates the CUD (Create, Update, Delete) operations in the connected scenario.
- Entity Framework builds and executes INSERT, UPDATE, and DELETE statements for the entities whose `EntityState` is Added, Modified, or Deleted when the `DbContext.SaveChanges()` method is called.




- An instance of `DbContext` keeps track of all the entities and so, it automatically sets an appropriate `EntityState` to each entity whenever an entity is created, modified, or deleted.

# Design the Window

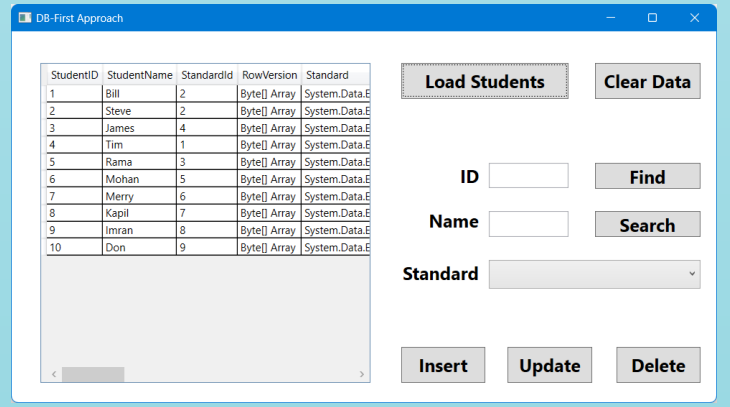
- Design the window something like this.



# Load All Students in DataGrid

- Create an instance of the context class.
- Define a method that fetches all the students.
- And displays them in the DataGrid.
- Call this method on the Load button's click event.

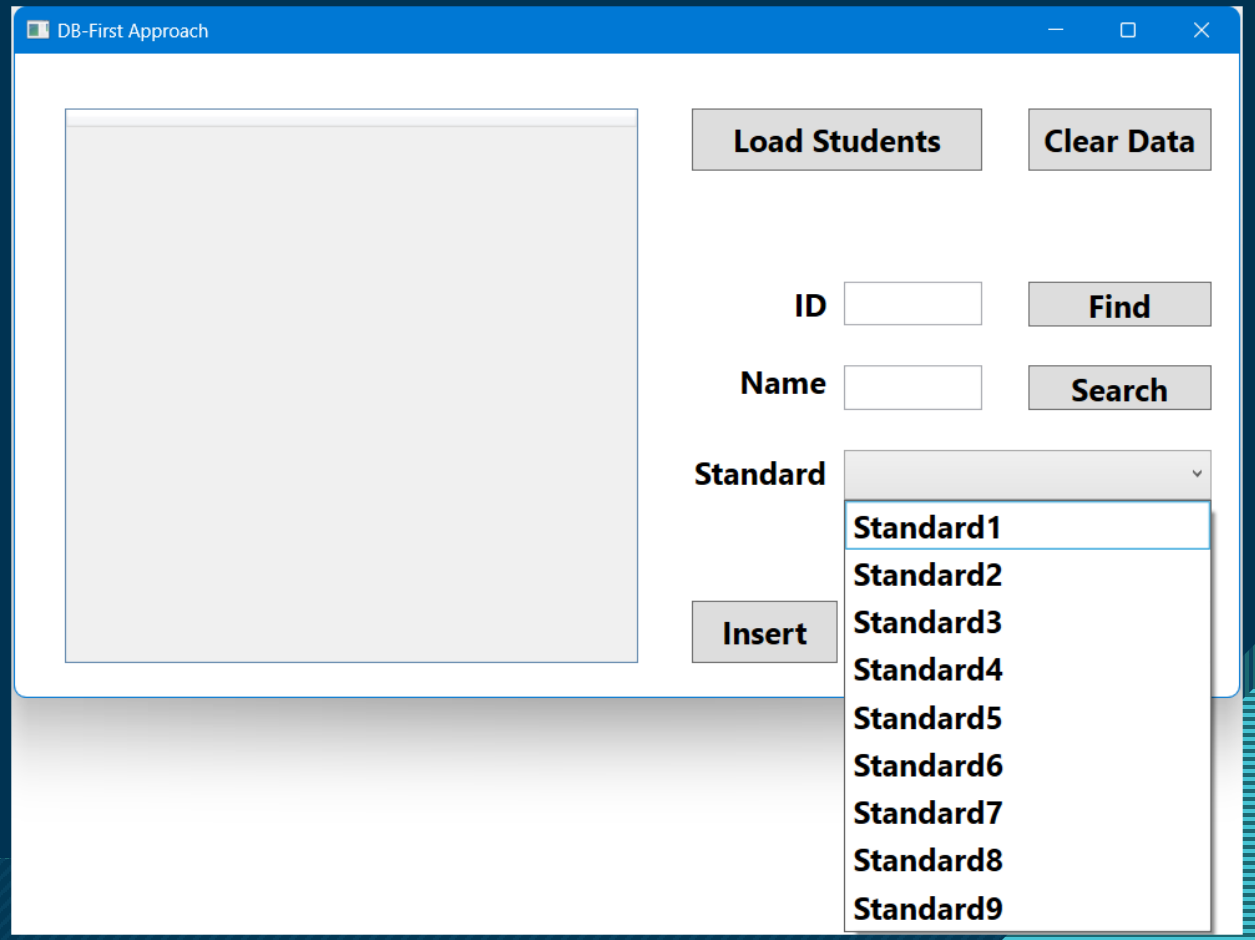
```
public partial class MainWindow : Window
{
    // create an object of context class
    SchoolDBEntities db = new SchoolDBEntities();

    public MainWindow()
    {
        InitializeComponent();
    }

    private void LoadStudents()
    {
        var students = db.Students.ToList();
        grdStudents.ItemsSource = students;
    }

    private void btnLoadStudents_Click(object sender, RoutedEventArgs e)
    {
        LoadStudents();
    }

}
```



# Populate the Standard ComboBox on Window's Loaded Event

- Fetch all the standards and display them in the ComboBox.
- Set the ComboBox's DisplayMemberPath to the StandardName field.
- Set the ComboBox's SelectedValuePath to the StandardId field.

```
private void Window_Loaded(object sender,
RoutedEventArgs e)
{
    var standards = db.Standards.ToList();

    cmbStandard.ItemsSource = standards;
    cmbStandard.DisplayMemberPath = "StandardName";
    cmbStandard.SelectedValuePath = "StandardId";
}
```



# Insert Data

- 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.
- In this code, `context.Students.Add(std)` adds a newly created instance of the `Student` entity to a context with Added `EntityState`.
- Calling the `SaveChanges()` method adds this entity into the database.

```
using(var context = new SchoolDBEntities())
{
    Student std = new Student();
    std.StudentName = txtName.Text;
    std.StandardId = cmbStandard.SelectedValue;

    context.Students.Add(std);

    context.SaveChanges();
}
```

# Updating Data

- 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.

```
using (var context = new SchoolDBEntities())
{
    var std = context.Students.Find(1);
    std.StudentName = txtName.Text;
    std.StandardId = cmbStandard.SelectedValue;

    context.SaveChanges();
}
```

- In this code, we find the student from the database using primary key.
- As soon as we modify the `StudentName`, the context sets its `EntityState` to Modified.
- In an update SQL statement, EF includes the properties with modified values, other properties being ignored.
- In this example, only the `StudentName` property was edited, so an update statement includes only the StudentName column.

# Deleting Data

- Use the `DbSet.Remove()` method to delete a record in the database table.
- In this code, `context.Students.Remove(std)` marks the `std` entity object as Deleted.

```
using (var context = new SchoolDBEntities())
{
    var std = context.Students.Find(1);

    context.Students.Remove(std);

    context.SaveChanges();
}
```

# Querying in Entity Framework

- You can build and execute queries using Entity Framework to fetch the data from the underlying database.
- EF supports different types of queries which in turn convert into SQL queries for the underlying database.

- Entity framework supports three types of queries:
  1. LINQ-to-Entities
  2. Entity SQL
  3. Native SQL

# LINQ-to-Entities

- Language-Integrated Query (LINQ) is a powerful query language introduced in Visual Studio 2008.
- As the name suggests, LINQ-to-Entities queries operate on the entity set (`DbSet` type properties) to access the data from the underlying database.
- You can use the LINQ method syntax or query syntax when querying with EDM.

# LINQ-to-Entities

- The following sample LINQ-to-Entities query fetches the data from the Student table in the database.

- LINQ Method syntax:

```
//Querying with LINQ to Entities
using(var context = new SchoolDBEntities())
{
    var query = context.Students
                .where(s => s.StudentName == "Bill")
                .FirstOrDefault<Student>();
}
```

- LINQ Query syntax:

```
using (var context = new SchoolDBEntities())
{
    var query = from std in context.Students
                where std.StudentName == "Bill"
                select std;

    var student = query.FirstOrDefault<Student>();
}
```

# Entity SQL

- Entity SQL is another way to create a query.
- It is processed by the Entity Framework's Object Services directly.
- It returns `ObjectQuery` instead of `IQueryable`.
- You need an `ObjectContext` to create a query using Entity SQL.

```
//Querying with Object Services and Entity SQL
string sqlString = "SELECT VALUE std FROM SchoolDBEntities.Students " +
                   "AS std WHERE std.StudentName == 'Bill'";

var objctx = (ctx as IObjectContextAdapter).ObjectContext;

ObjectQuery<Student> student = objctx.CreateQuery<Student>(sqlString);
Student newStudent = student.First<Student>();
```

# Native SQL

- You can execute native SQL queries for a relational database, as shown below:

```
using (var ctx = new SchoolDBEntities())
{
    var studentName = ctx.Students.SqlQuery("Select studentid, studentname,
                                             standardId from Student where
                                             studentname='Bill'")
                         .FirstOrDefault<Student>();
}
```

# Linq-to-Entities Query

- The `DbSet` class is derived from `IQueryable`.
- So, we can use LINQ for querying against `DbSet`, which will be converted to an SQL query.
- EF, then 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.
- The following are some of the standard query operators that can be used with LINQ-to-Entities queries.

| LINQ Extension Methods | | |
|---|---|---|
| `First()` | `ToList()` | `Last()` |
| `FirstOrDefault()` | `Count()` | `LastOrDefault()` |
| `Single()` | `Min()` | `Average()` |
| `SingleOrDefault()` | `Max()` | |

# Find()

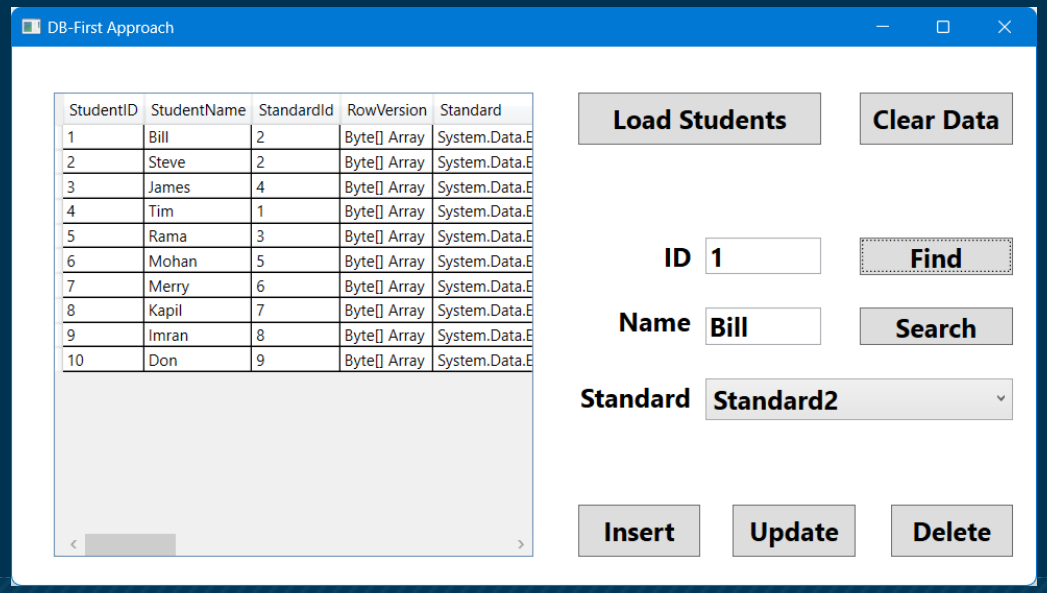
- We can use the `Find()` method of `DbSet` to search the entity based on the primary key value.
- Let's assume that `SchoolDbEntities` is our `DbContext` class and `Students` is the `DbSet` property.

```
int id = int.Parse(txtId.Text);
var student = db.Students.Find(id);
```

- In the above example, `db.Students.Find(id)` returns a student record whose `StudentId` matches the value of the `txtId` textbox.
- If no record is found, then it returns `null`.

# Get a Student by ID

- Use the `Find()` method to get a student by ID.
- If a valid student is found, display its info in the respective fields.
- Otherwise, clear the fields and display an appropriate message.




```
private void btnFind_Click(object sender, RoutedEventArgs e)
{
    int id = int.Parse(txtId.Text);
    var student = db.Students.Find(id);

    if (student != null)
    {
        txtName.Text = student.StudentName;
        cmbStandard.SelectedValue = student.StandardId;
    }
    else
    {
        txtName.Text = "";
        cmbStandard.SelectedIndex = -1;
        MessageBox.Show("Invalid ID. Please try again.");
    }
}
```

# First() / FirstOrDefault()

- If you want to get a single student object, when there are many students, whose name is "Bill" in the database, then use `First` or `FirstOrDefault`, as shown:

- LINQ Query Syntax:

```
using (var ctx = new SchoolDBEntities())
{
    var student = (from s in ctx.Students
                   where s.StudentName == "Bill"
                   select s).FirstOrDefault<Student>();
}
```

- LINQ Method Syntax:

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

# First() / FirstOrDefault()

- The difference between `First` and `FirstOrDefault` is that `First()` will throw an exception if there is no result data for the supplied criteria, whereas `FirstOrDefault()` returns a default value (`null`) if there is no result data.

# SingleOrDefault() vs. FirstOrDefault()

- The difference between `SingleOrDefault()` and `FirstOrDefault()` is that:
- `SingleOrDefault()` returns a single element, if there is exactly one element. An exception is thrown if no element is returned, or more than one elements exist.

- `FirstOrDefault()` returns a single element, if there is at least one element. Or returns `null` when there is no element.

# SingleOrDefault() vs. FirstOrDefault()

- **Summary:**

- If your result set returns 0 records:
  - `SingleOrDefault` returns the default value for the type (e.g. null)
  - `FirstOrDefault` returns the default value for the type

- If you result set returns 1 record:
  - `SingleOrDefault` returns that record
  - `FirstOrDefault` returns that record

- If your result set returns many records:
  - `SingleOrDefault` throws an exception
  - `FirstOrDefault` returns the first record

- Reference:
  - https://stackoverflow.com/questions/1745691/linq-when-to-use-singleordefault-vs-firstordefault-with-filtering-criteria

# ToList()

- The `ToList` method returns the collection result.
- If you want to list all the students with the same name, then use `ToList()`:

```
using (var ctx = new SchoolDBEntities())
{
    var studentList = ctx.Students
                         .Where(s => s.StudentName == "Bill")
                         .ToList();
}
```

- We may also use `ToArray`, `ToDictionary` or `ToLookup`.

# OrderBy()

- Use the `OrderBy` operator with `ascending`/`descending` keywords in LINQ query syntax to get the sorted entity list.

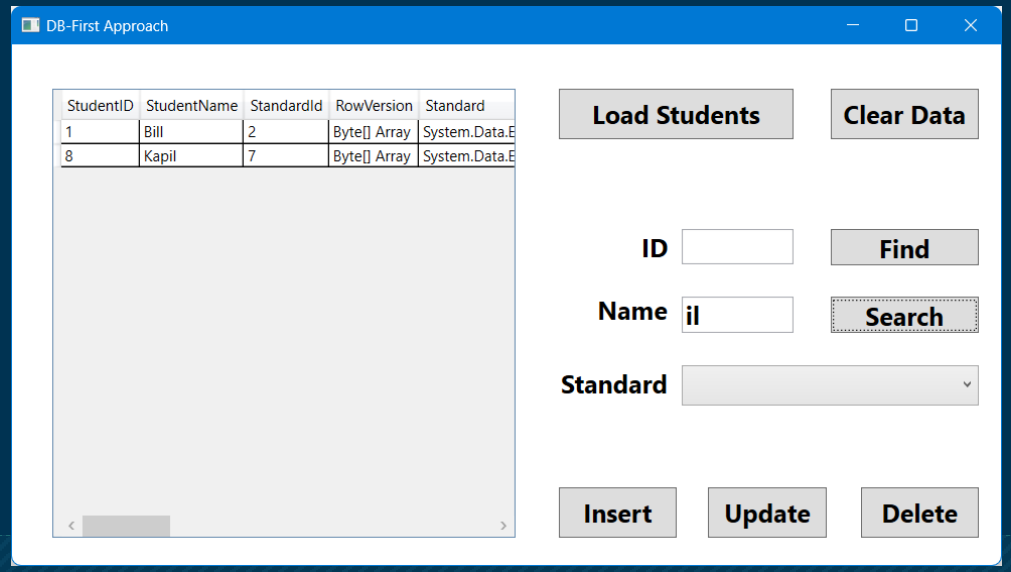
```
using (var ctx = new SchoolDBEntities())
{
    var students = from s in ctx.Students
                   orderby s.StudentName ascending
                   select s;
}
```

- Use the `OrderBy` or `OrderByDescending` method to get the sorted entity list.

```
using (var ctx = new SchoolDBEntities())
{
    var students = ctx.Students.OrderBy(s => s.StudentName).ToList();
    // or descending order
    var  descStudents = ctx.Students.OrderByDescending(s => s.StudentName).ToList();
}
```

# Search a Student by Name

- Write a LINQ query to fetch students by name.
- You can either use query or method syntax.
- Use the `Contains()` method of the `String` to perform a partial match.



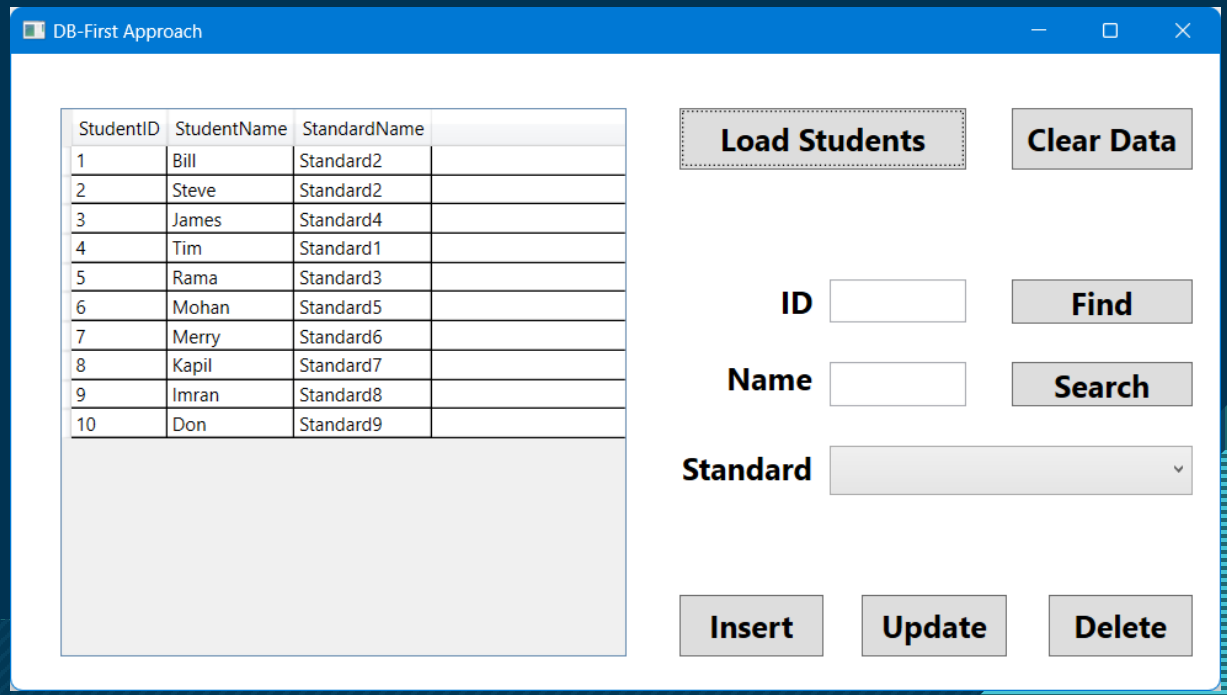

```
private void btnSearch_Click(object sender, RoutedEventArgs e)
{
    string name = txtName.Text;

    // query syntax
    // Contains() method can be used for partial matches
    var students = (from std in db.Students
                    where std.StudentName.Contains(name)
                    select std).ToList();

    // method syntax
    students = db.Students
                 .Where(s => s.StudentName.Contains(name))
                 .ToList();

    grdStudents.ItemsSource = students;
}
```

# Fetching Data from Multiple Entities

- You can easily fetch data from multiple entities using LINQ.
- For e.g., to replace a foreign key ID column with name from another table requires INNER JOIN.
- But this can be easily achieved in EF by creating an anonymous object in LINQ.
- The `select` clause can create a new object of *anonymous type* (a type with no name), which the compiler generates for you based on the properties listed in the curly braces ({}).

```
var students = from std in db.Students
               select new { std.StudentID, std.StudentName, std.Standard.StandardName };
```

- `Standard` is a *reference navigation property* in `Students`.
- Therefore, this property can be used to access properties from the `Standard` entity.
- Implement the above LINQ in the `LoadStudents()` method.



# Thank You

# Copyright

# References

**Most of the material has been taken as is from:**

- Entity Framework 6 Introduction:
  - https://www.entityframeworktutorial.net/entityframework6/introduction.aspx