Chapter 5
Entity Framework

Mrs. Swati Satpute
Fergusson College

Objective

- Entity framework Basics
- Modelling approaches

ORM

- An object-relational mapper provides an object-oriented **layer between** relational databases and object-oriented programming languages **without having to write SQL queries**.

- It standardizes interfaces and speeds development time.

Entity Framework

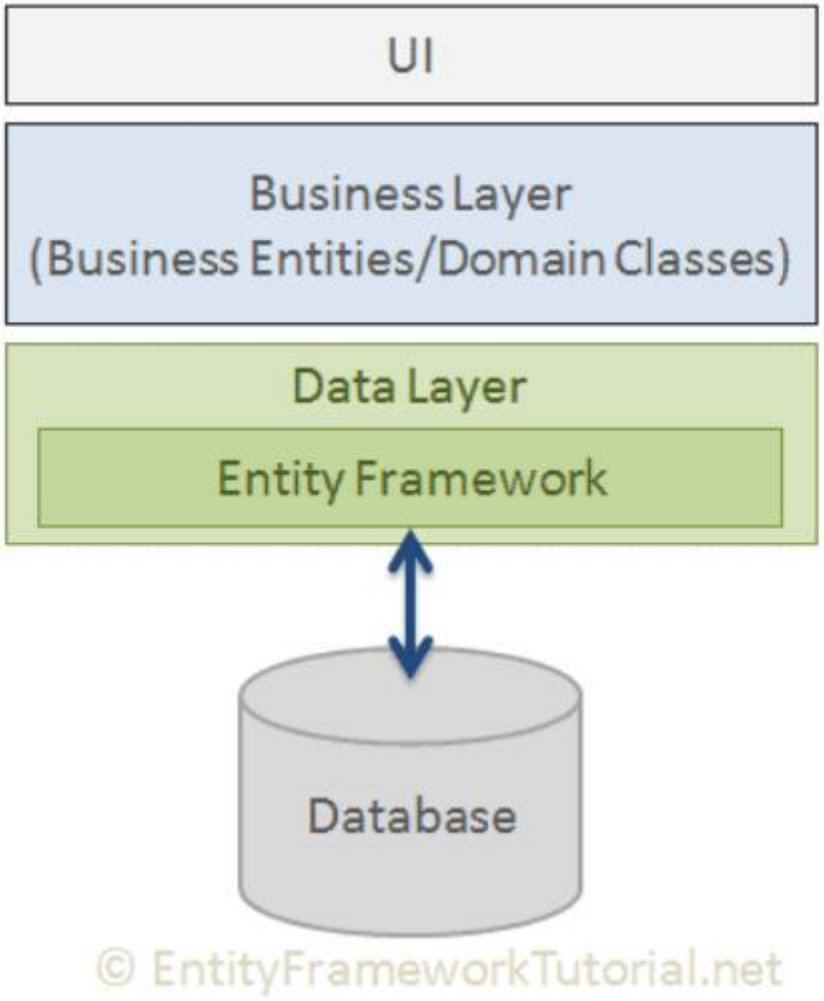
- **Entity Framework** is an **Object Relational Mapper** between application objects and table and columns of Database.

- Entity Framework is an **ORM for ADO.NET**

- EF takes care of
  - **creating database connections** and **executing command**
  - **Fetching query results** and **converting them into data objects**
  - It also **tracks updates made** in memory to those objects and **applying changes into database on commit**

- Examples of ORM: NHibernate and LLBLGen Pro

Use of EF in application



Entity Framework Features

- **Cross-platform:** can run on Windows, Linux and Mac.

- **Modelling:** EF creates an EDM (**Entity Data Model**) based on POCO (Plain Old CLR Object) entities with get/set properties of different data types. It **uses this model** when **querying** or **saving entity data to the underlying database**.

- **Querying:** uses LINQ queries for data retrieval (from DB), execution of raw SQL queries is possible.

Entity Framework Features

- **Change Tracking:** EF keeps track of changes at instances entities, submit to DB

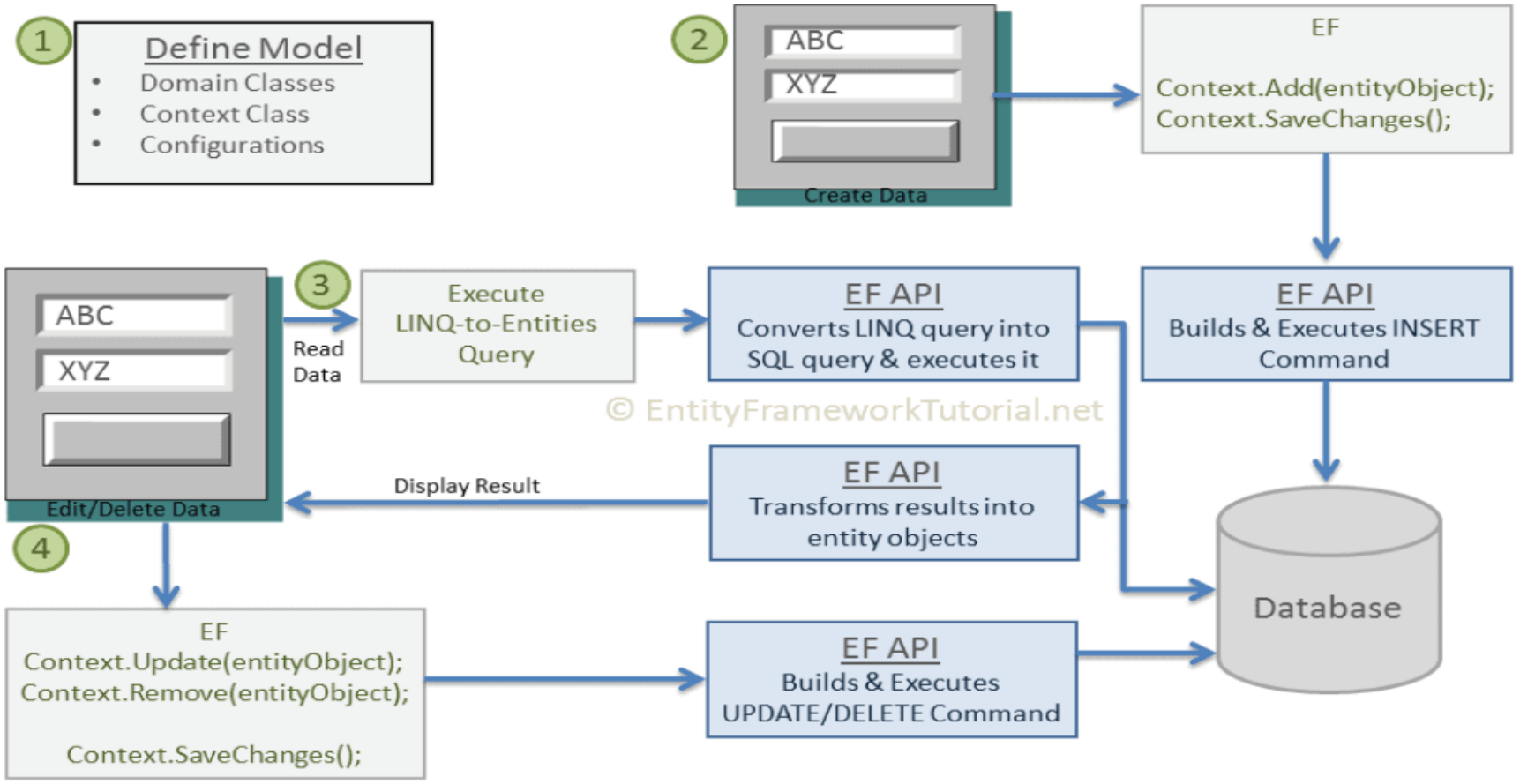
- **Saving:** EF executes **INSERT**, **UPDATE**, and **DELETE** commands to the database based on the changes occurred to your entities when you call the **SaveChanges**().

- **Concurrency**: Data Protection possible for multiusers.

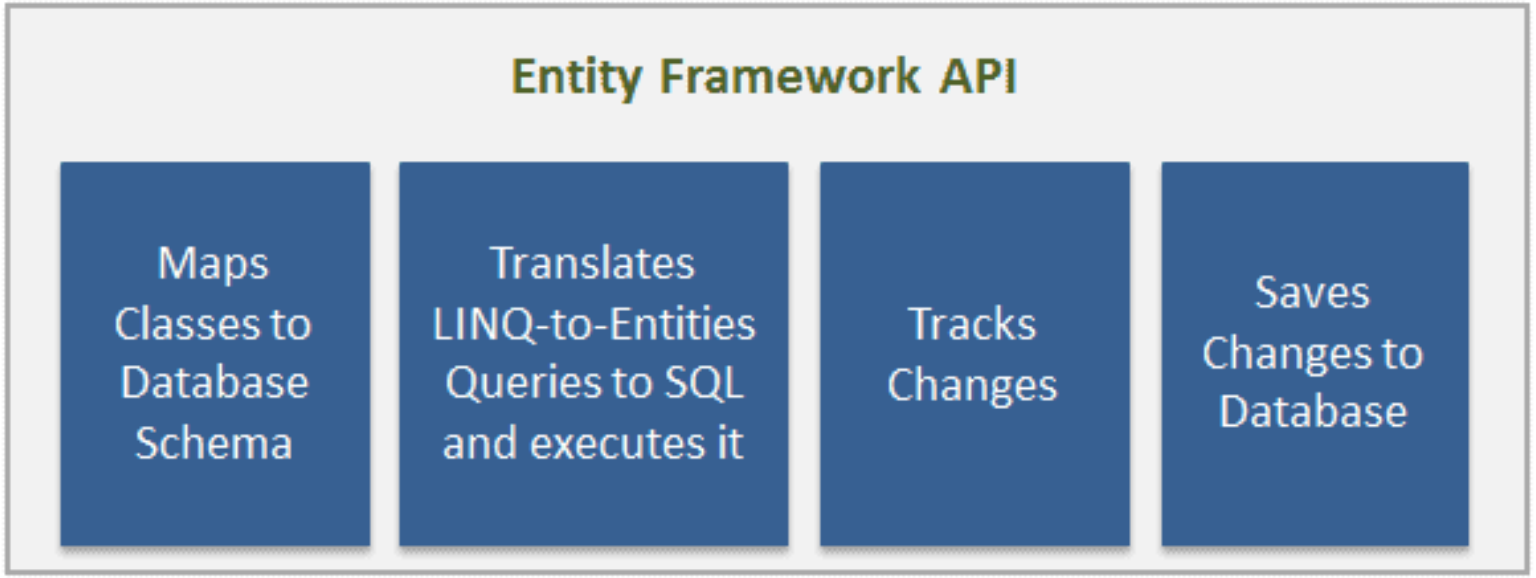
Entity Framework Features

- **Transactions:**
  - Performs **automatic transaction management** while querying or saving data.
  - provides options to **customize transaction management.**
- **Caching:** repeated querying will **return data from the cache** instead of hitting the DB.
- **Configurations:** EF allows use of **data annotation** to override defaults configure Data
- **Migration:** EF **provides 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

Basic WorkFlow in Entity Framework



Working of Entity Framework (EF)



- **EF API** maps domain (entity) classes to the database schema,
- Translate & execute LINQ queries to SQL,
- Track changes occurred on entities during their lifetime,
- Save changes to the database.

Entity Data Model

- It is an **in-memory representation** of the entire metadata includes
  - **Conceptual model**: includes Model classes & relations**, independent from DB**. Defines LINQ-to-Entities (L2E)
  - **Storage model**: Its DB design model, includes tables, views, stored procedures, and their relationships and keys.
    - In the code-first approach, this will be **inferred from the conceptual model**
    - In the database-first approach, this will be **inferred from the targeted database.**
  - **Mapping**: consists of information about how the conceptual model is mapped to the storage model.

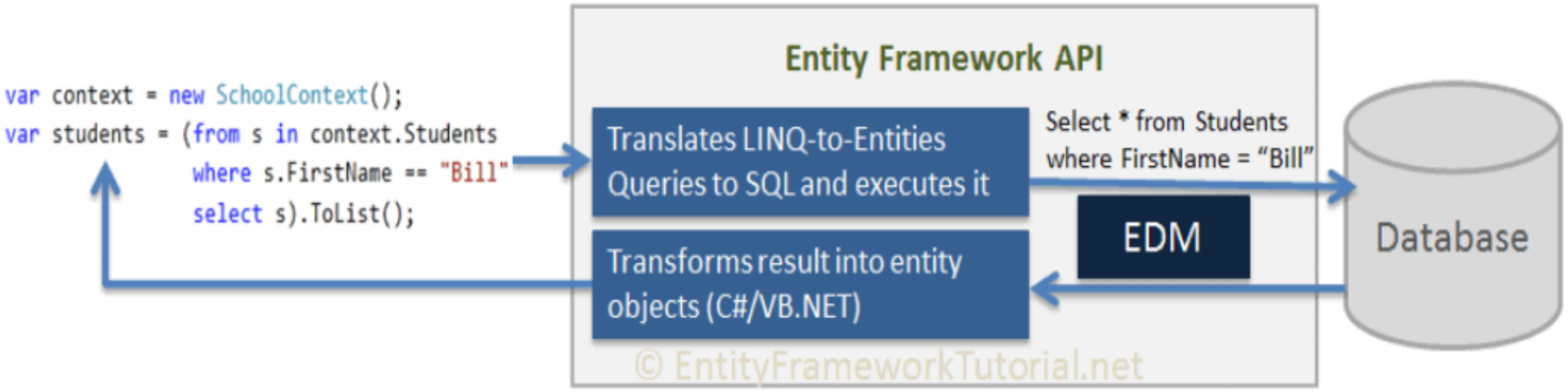
Entity Data Model

- EF performs **CRUD operations** using this EDM.
-
- It uses EDM in **building SQL queries** from LINQ queries, building INSERT, UPDATE, and DELETE commands, and

- **Transform database result into entity objects**

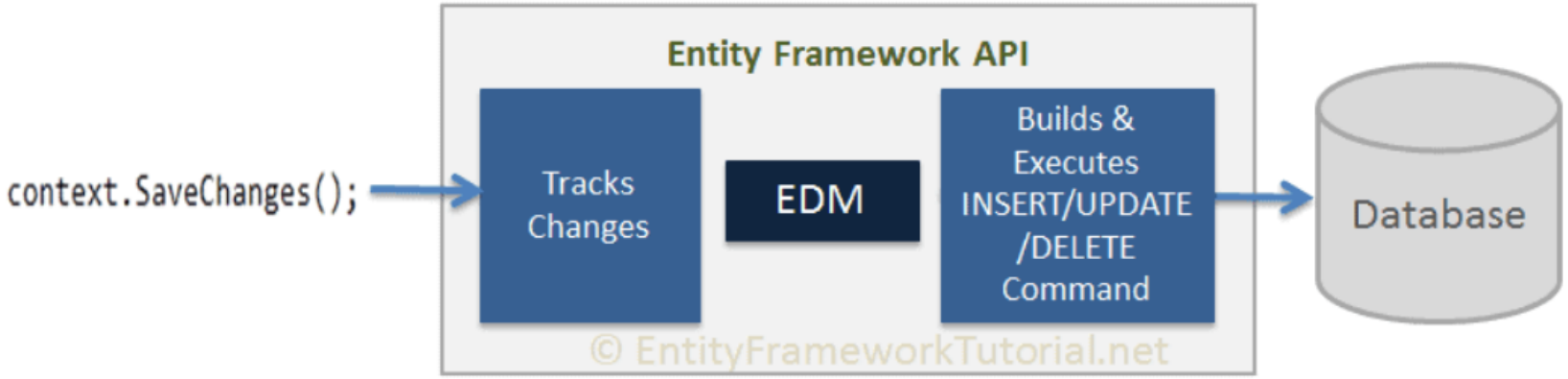
Querying in EF

EF API translates LINQ-to-Entities queries to SQL queries for relational databases using EDM and also converts results back to entity objects.

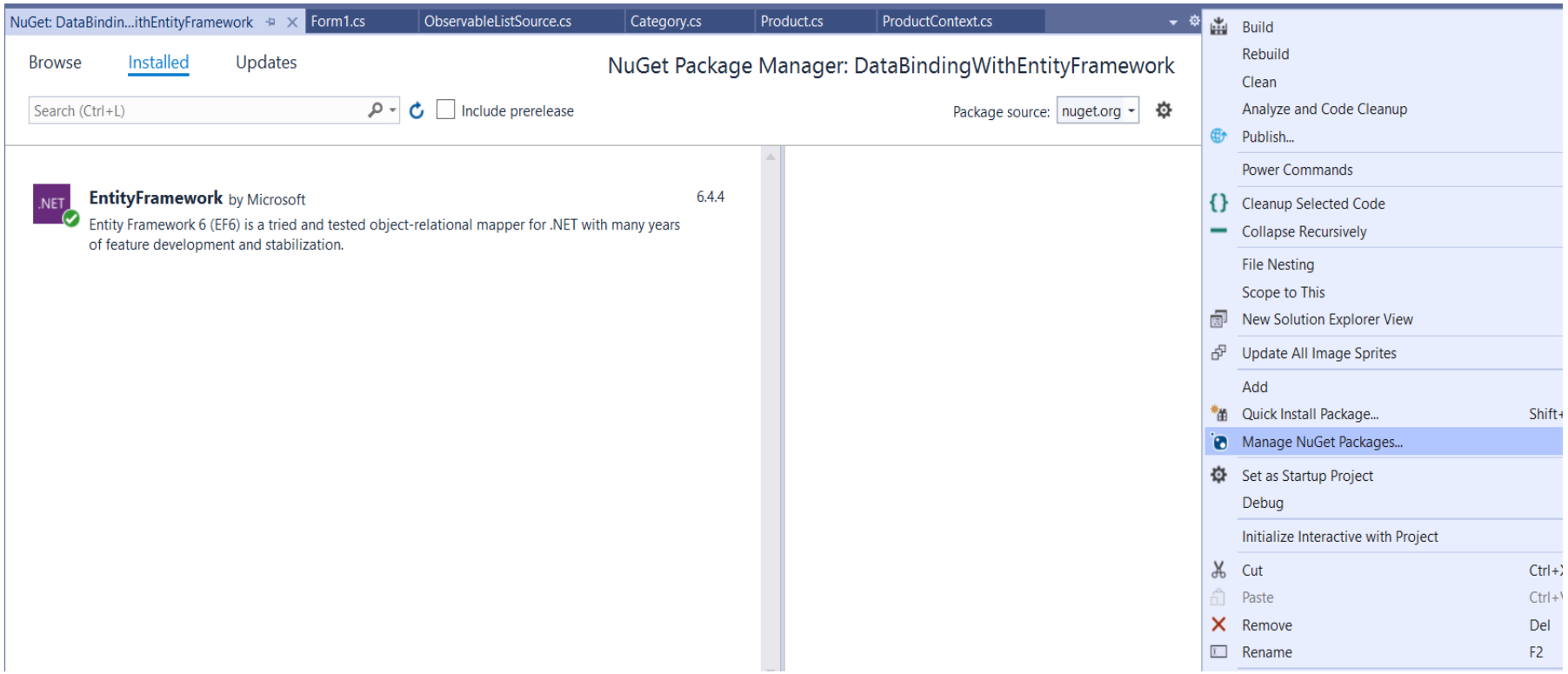


Saving data in EF

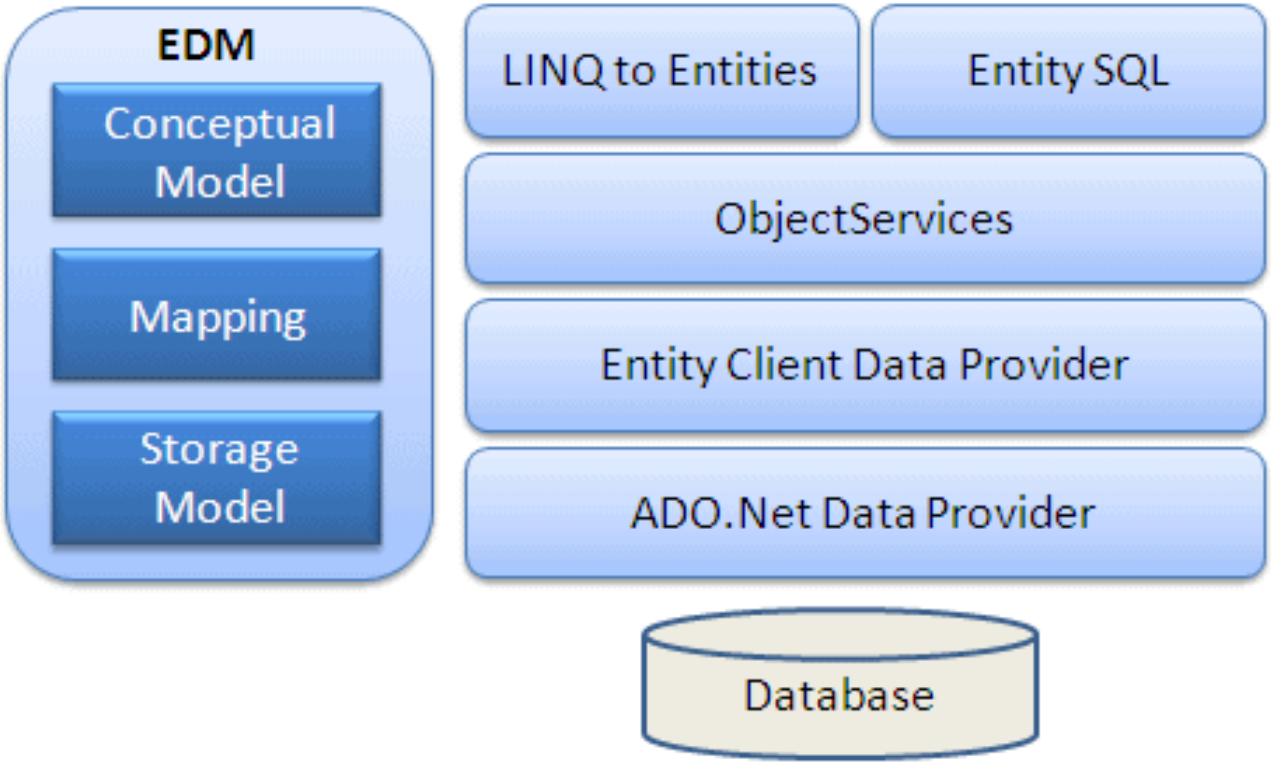
EF API infers INSERT, UPDATE, and DELETE commands based on the state of entities when the `SaveChanges()` method is called. The ChangeTrack keeps track of the states of each entity as and when an action is performed.



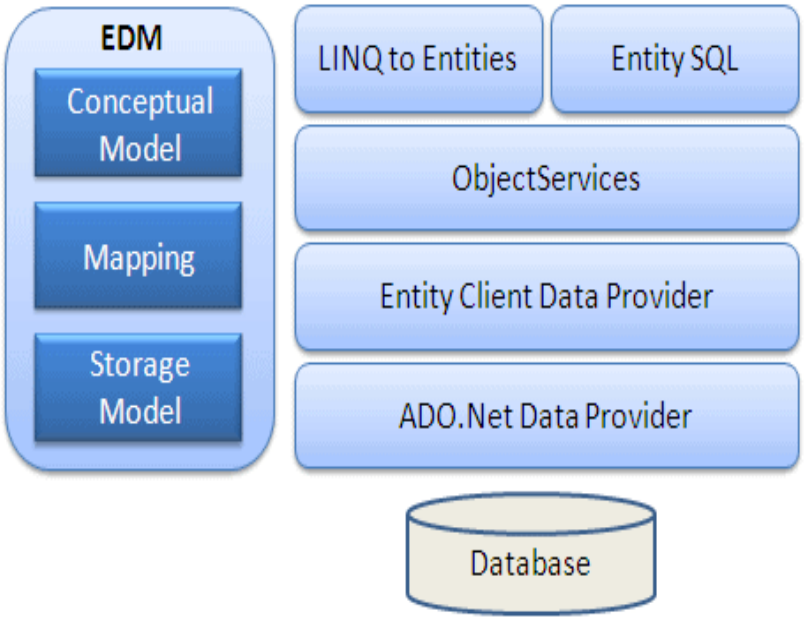
Installing Entity Framework from NuGet



Entity Framework Architecture

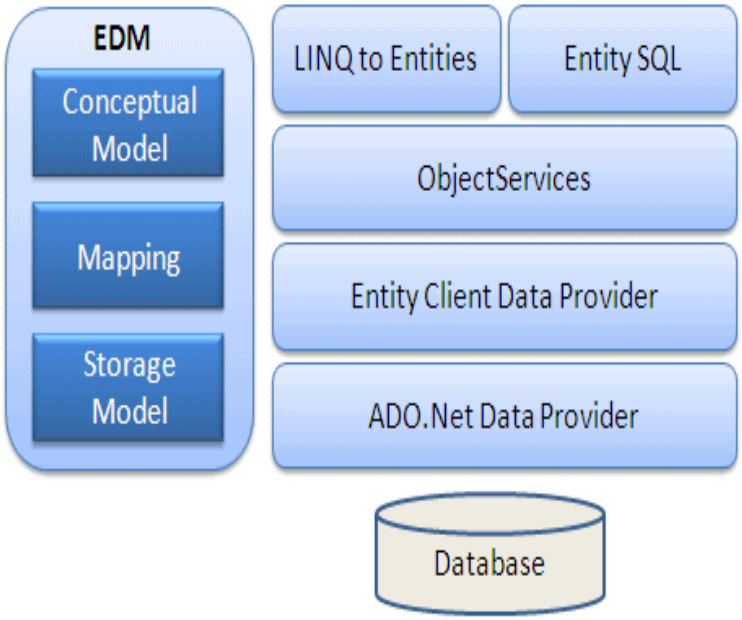


Entity Framework Architecture



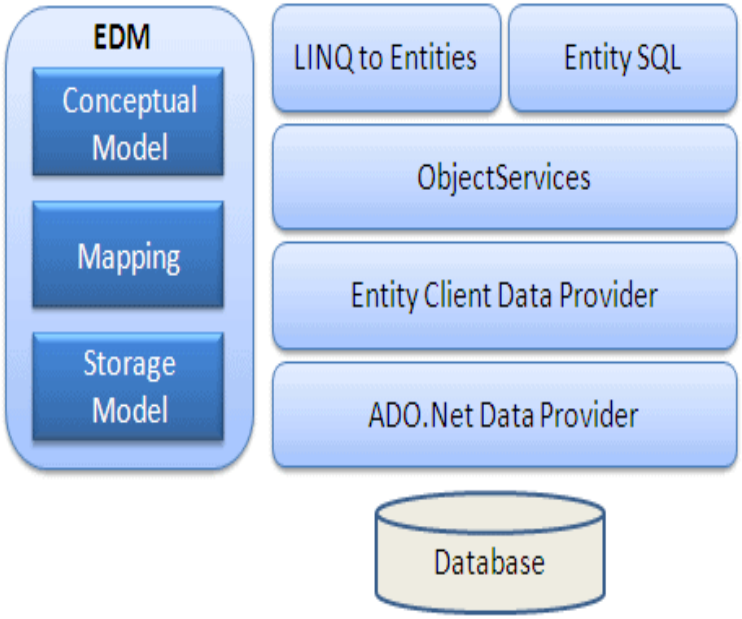
- **Entity Data Model**
  - Conceptual model
  - Mapping
  - Storage Model

Entity Framework Architecture



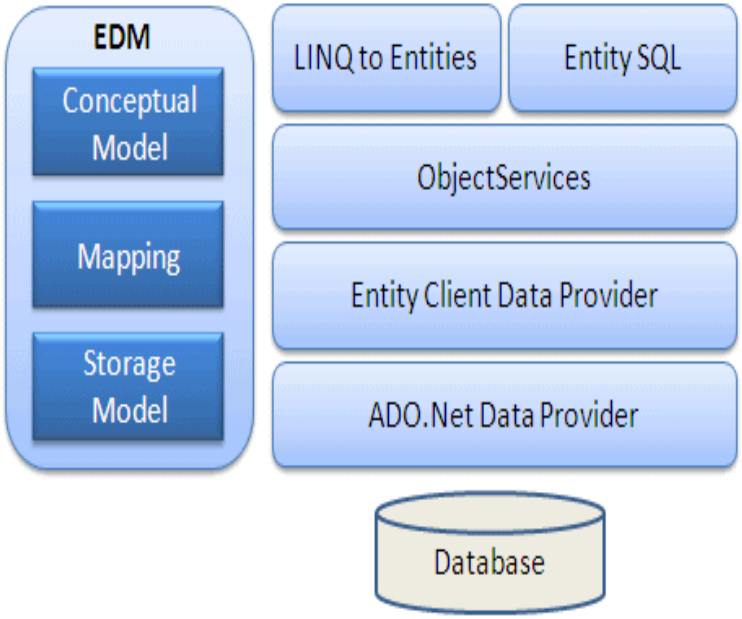
- **LINQ-to-Entities (L2E)**
  - is a query language **used to write queries against the object model**. It **returns entities**, which are defined in the conceptual model.
- **Entity SQL**
  - Another query language (For EF 6 only) just like LINQ to Entities

Entity Framework Architecture



- **Object Service**
  - is a main **entry point** for **accessing data from the database** and **returning it back**.
  - responsible for **materialization**, which is the **process of converting data returned from an Entity client data provider to an Entity object structure**

Entity Framework Architecture



- **Entity Client Data Provider**
  - main responsibility of this layer is to **convert LINQ-to-Entities or Entity SQL queries** into a **SQL query**

- 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

Entity Data Model

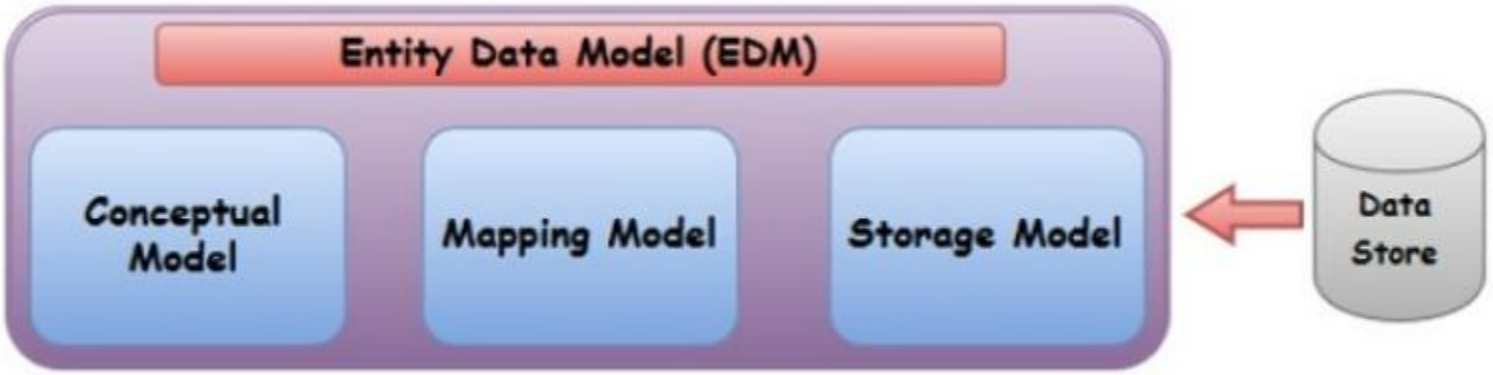
Entity Data Model

- The Entity Data Model (EDM) describes **conceptual model of data** irrespective of their storage.

- Core parts of EDM
  - Storage Schema Model
  - Conceptual Model
  - Mapping Model

Storage Schema Model

- Its also called as Storage Schema Definition Layer (SSDL).
- It represents **schematic representation of backend data source**.



The Conceptual Mode

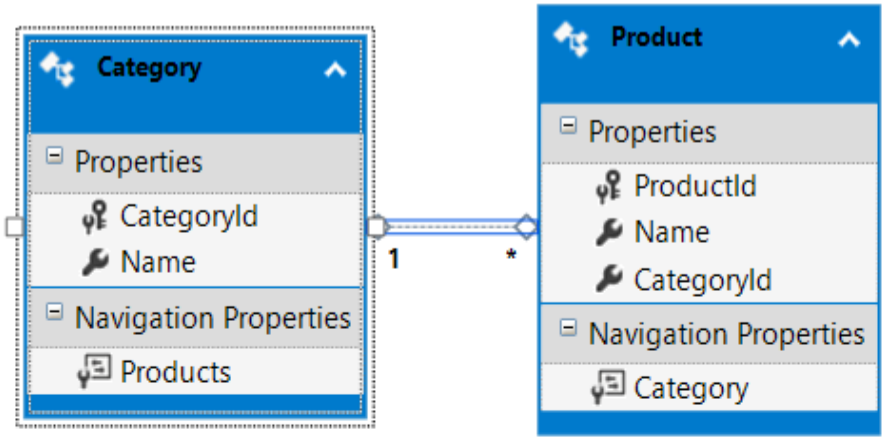
- Also called as **Conceptual Schema Definition Layer (CSDL).**

- These are **classes represent Entity.**
- Queries are written for these entities.

The Mapping Model - EDM

- The **logical schema** and its **mapping** with the **physical schema** is **represented** as an EDM.
- **Entity Designer** in Visual Studio depicts this mapping.
- **.Edmx** file contains **Entity Framework metadata**

Edmx design view



Physical Schema Attributes

Conceptual Entity Attributes

EDM Schema

The EDM describe data structure in following ways

- Entity Type
- Association Type
- Property

Entity Type

- In a conceptual model, **entity types are constructed from properties** and **describe the structure of top-level concepts** like Category or Product in from real world.
- **Each entity must have a unique entity key** within an entity set.
- An **entity set** is a **collection of instances** of a specific entity type.
- **Entity sets** (and association sets) are **logically grouped in an entity container**.
- Inheritance is supported with entity types

Association Type

- An association represents a relationship between two entity types.
  - E.g. 1 Category contains multiple products.
- Every association has two association ends that specify the entity types involved in the association.
- **Each association end specifies an association end multiplicity** that indicates the **number of entities** that can be at that **end of the association.**
- An association end multiplicity can have a value of one (1), zero or one (0..1), or many (*).
- Entities at one end of an association can be accessed through **navigation properties**, or **through foreign keys** if they are exposed on an entity type.

Property

- Entity types contain properties that define their structure and characteristics.
  - E.g. Category has Category Name

- **A property can contain primitive data** (such as a string, an integer, or a Boolean value), or **structured data** (such as a **complex type**).

Entity Framework - DbContext

DbContext

- Entity Framework enable CRUD operations on entities. It also maps conceptual entity attributes to database.

- It also provides facilities
  - Materialize data returned from the database as entity objects
  - **Track changes** that were made to the objects
  - **Handle concurrency**
  - Propagate object changes back to the database
  - Bind objects to controls

- The primary class that is responsible for interacting with data as objects is **System.Data.Entity.DbContext**.

DbContext Class in Entity Framework

- In Code First Approach, one has to **derive** a class **from DbContext** and **exposes DbSet properties** that **represent collections of the specified entities** in the context.

```
class ProductContext : DbContext
{
    2 references
    public DbSet<Category> Categories { get; set; }
    2 references
    public DbSet<Product> Products { get; set; }

}
```

DbContext

- DbCobtext class instance **represents a session** with the underlying database.

- It can **combine multiple changes under a single database transaction.**

- Adding new Entity
  - Create new entity instance
  - register it using the Add method on DbContext
  - Save changes

```
var category = new Category
{
    Name = "Mobile",
};

context.Categories.Add(category);
context.SaveChanges();
```

- Update new Entity
  - Change the properties of entity instance
  - Call SaveChanges() on DbSet

```
var category = (from d in context.Categories
                where d.Name == "Mobile"
                select d).Single();
category.Name = "SmartPhone";
context.SaveChanges();
```

- Remove Entity
  - Remove() on collection will remove entity from change tracker
  - Call SaveChanges to make permanent changes

```
private static void RemoveCategory()
{
    using (var context = new ProductContext())
    {
        var category = (from d in context.Categories
                        where d.Name == "Mobile"
                        select d).Single();
        context.Categories.Remove(category);
        context.SaveChanges();
    }
}
```

Types of Entities

Two types of writing entity class for data model

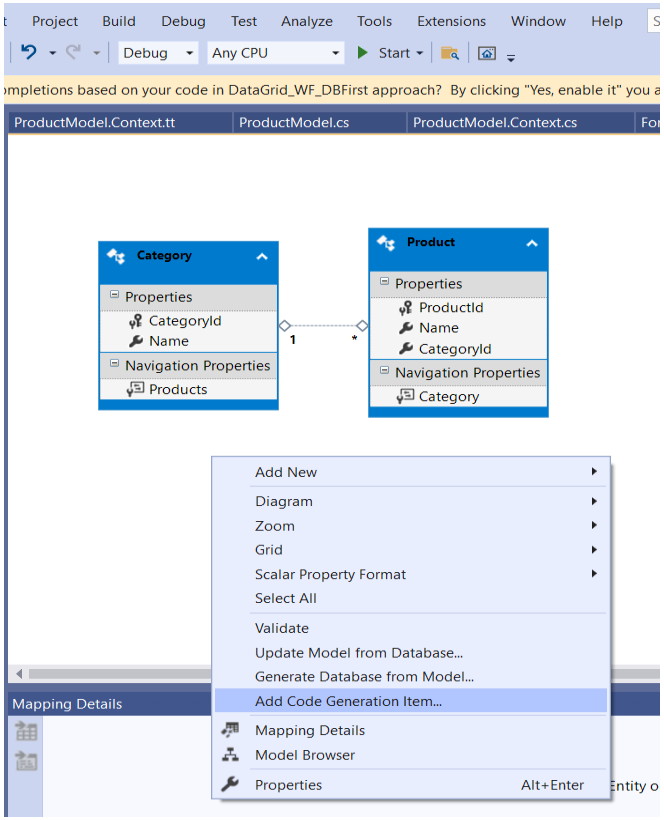
- **POCO entities**
- **Dynamic Proxy**

- POCO Entities
  - POCO stands for **"plain-old" CLR objects** which can be used as existing domain objects with your data model.
  - POCO data classes which are mapped to entities are **defined in a data model.**

  - It also supports most of the same query, insert, update, and delete behaviors as entity types that are generated by the Entity Data Model tools.
  - You can **use the POCO template to generate persistence-ignorant entity types** from a conceptual model.

- They are **autogenerated from Entity Model**

POCO entity – autogenerated Code




```
using System;
using System.Collections.Generic;

4 references
public partial class Category
{
    [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage", "CA2214:DoNotCallOverridableMethodsInConstruc
    0 references
    public Category()
    {
        this.Products = new HashSet<Product>();
    }

    0 references
    public int CategoryId { get; set; }
    0 references
    public string Name { get; set; }

    [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage", "CA2227:CollectionPropertiesShouldBeReadOnly"
    1 reference
    public virtual ICollection<Product> Products { get; set; }
```

POCO entity – autogenerated DbContext Code

```
3 references
public partial class ProductContext : DbContext
{
    1 reference
    public ProductContext()
        : base("name=ProductContext")
    {
    }

    0 references
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        throw new UnintentionalCodeFirstException();
    }

    2 references
    public virtual DbSet<Category> Categories { get; set; }
    2 references
    public virtual DbSet<Product> Products { get; set; }
}
```

Dynamic Proxy Entity Type

- When **creating instances of POCO entity types**, the Entity Framework often creates instances of a dynamically generated derived type that acts as a **proxy for the entity**

- Can be **useful for automatic updates in properties.**

- This mechanism is **used to support lazy loading of relationships and automatic change tracking**.

- This **technique also applies to those models** which are **created with Code First and EF Designer**.

Rules for dynamic Proxy type

- If you want the Entity Framework to support lazy loading of the related objects and to track changes in POCO classes, then the POCO classes must meet the following requirements

  - Custom data class must be declared with public access.
  - Custom data class must not be sealed.
  - Custom data class must not be abstract.
  - Custom data class must have a public or protected constructor that does not have parameters.

Dynamic Proxy Entity Type

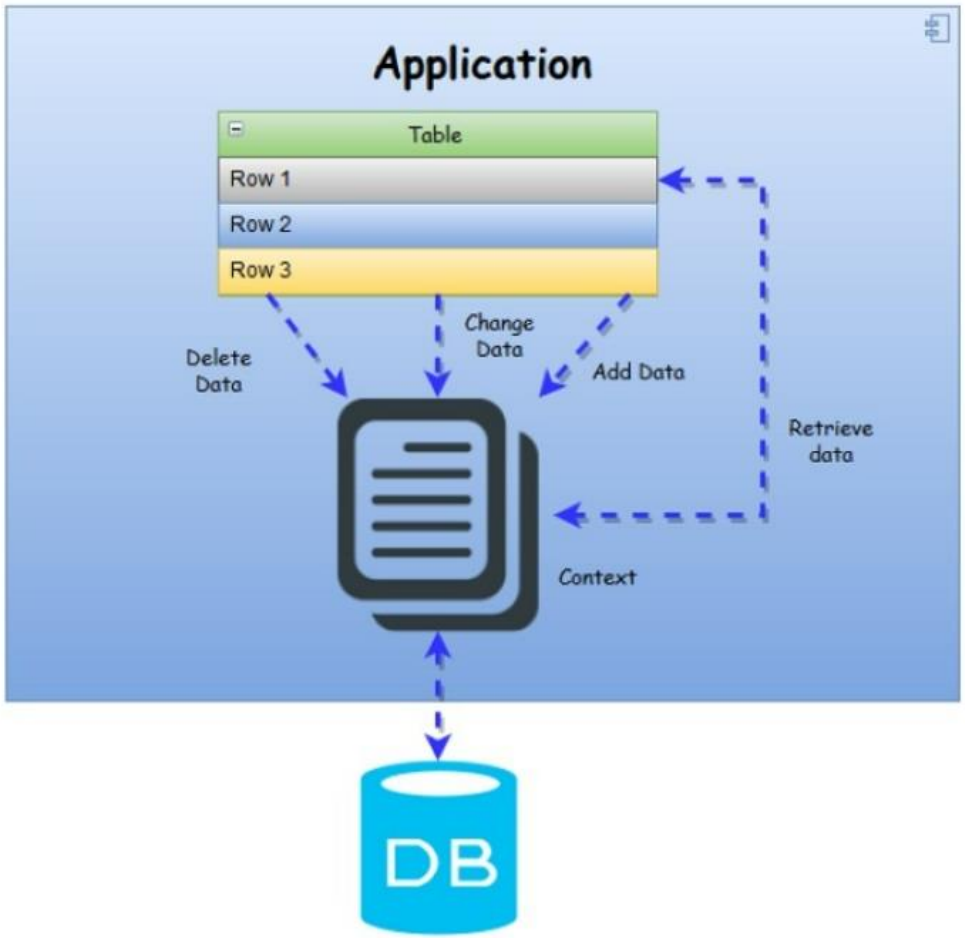
```
public class Category
{
    private readonly ObservableListSource<Product> _products =
                new ObservableListSource<Product>();

    0 references
    public int CategoryId { get; set; }
    4 references
    public string Name { get; set; }
    0 references
    public virtual ObservableListSource<Product> Products { get { return _products; } }

}
```

Entity Framework - Lifecycle

- Lifetime of DbContext instance is from its create till it is collected by GC



Context object hold all the entity instances in memory, resulting in huge memory consumption.

Entity Lifecycle

Enum of type **System.Data.EntityState** represents entity states:

- **Added:** The entity is marked as added.
- **Deleted:** The entity is marked as deleted.
- **Modified:** The entity has been modified.
- **Unchanged:** The entity hasn’t been modified.
- **Detached:** The entity isn’t tracked.

Using Statement

- To free up all the resources held by DbContext, use the Using statement.
- Compiler automatically creates try/catch and finally block to dispose all resources.

```
using (var context = new ProductContext())
{
    var category = (from d in context.Categories
                    where d.Name == "Mobile"
                    select d).Single();
    context.Categories.Remove(category);
    context.SaveChanges();
}
```

Thumb rule for DbContext

- In case of **Web Applications**, the **DbContext** should be **used per request.**
- In case of **Desktop application** like WinForms/ WPF use **DbContext per form / dialog / page.**

- This way, we will gain a lot of the context's abilities and won't suffer from the implications of long running contexts.

Data Annotations

- They help to define extra information needed for configuration.
- **System.ComponentModel.DataAnnotations** includes the following attributes that impacts the **nullability or size of the column**.
  - Key - set the property as Primary key in Database
  - Timestamp – sets column as non-nullable.
  - ConcurrencyCheck
  - Required – Mandatory field
  - MinLength – specifying column length
  - MaxLength
  - StringLength

Data Annotations

- **System.ComponentModel.DataAnnotations.Schema** namespace includes the following attributes that impacts the **schema** of the database.
  - Table
  - Column
  - Index
  - ForeignKey
  - NotMapped
  - InverseProperty

Data Annotations

[Key] – Primary Key

```
public class Student {

   [Key]
   public int StdntID { get; set; }
   public string LastName { get; set; }
   public string FirstMidName { get; set; }
   public DateTime EnrollmentDate { get; set; }

   public virtual ICollection<Enrollment> Enrollments { get; set; }
}
```

Composite Key

```
public class DrivingLicense {

   [Key, Column(Order = 1)]
   public int LicenseNumber { get; set; }
   [Key, Column(Order = 2)]
   public string IssuingCountry { get; set; }
   public DateTime Issued { get; set; }
   public DateTime Expires { get; set; }
}
```

Data Annotations

Max length and Min Length

```
public class Course {

   public int CourseID { get; set; }
   [ConcurrencyCheck]
   [MaxLength(24) , MinLength(5)]
   public string Title { get; set; }
   public int Credits { get; set; }

   public virtual ICollection<Enrollment> Enrollments { get; set; }
}
```

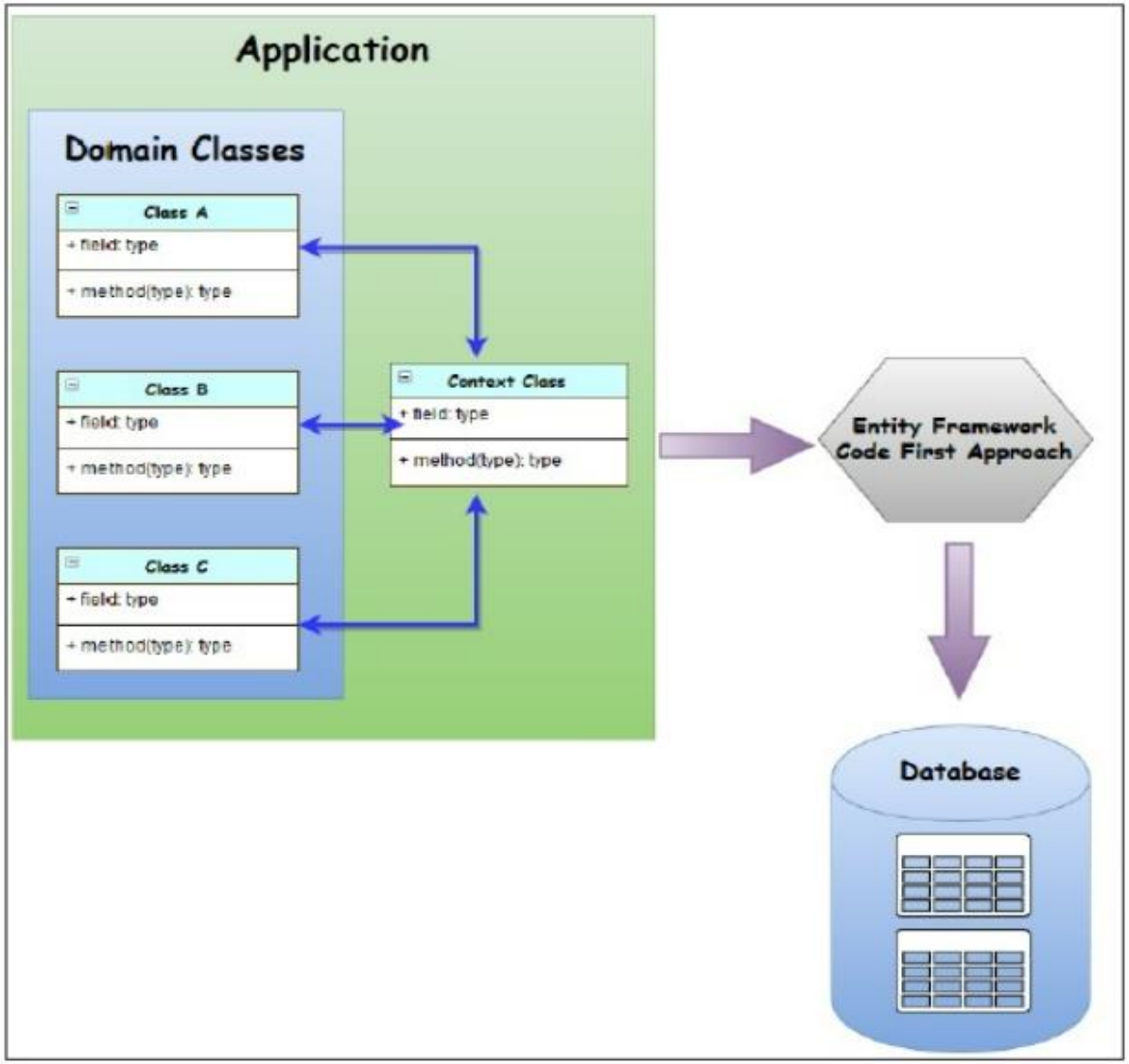
EF – Development Approach

- Code First
- Database First
- Model First

Code First

- In this, Developer defines conceptual entities
- Implement DbContext class
- Target Database **may not exist or may be empty**
- Database get created automatically by this approach

Code First


Application
Domain Classes
Class A
+ field: type
+ method(type): type
Class B
+ field: type
+ method(type): type
Class C
+ field: type
+ method(type): type
Context Class
+ field: type
+ method(type): type
Entity Framework
Code First Approach
Database

Why Code First?

- Developer can focus on defining domain classes and their relation, business logic

- These class have nothing to do with Entity Framework.

- The **context class** (derived from DbContext), **manages interaction between** the **domain classes and database.** Context class is a feature of EF.

Why Code First?

- **Code First adds a model builder** that inspects your classes that the context is managing, and then uses a set of rules or conventions to determine how those classes and the relationships describe a model, and how that model should map to your database.

- All of this happens at runtime. **One can not see the model working behind it.**

- Code First has the ability to use that model to create a database if required.

- In can also update database if model gets changed.

Lets Try

Demo

- Write an Data Binding application displaying Category and Product relationship using Code First approach

- Demo1 - ConsoleApp_CodeFirst
- Demo2 – DataBindingWithEntityFramework
  - Form1
  - Form2 CRUD operation - LINQ

- Refer –
  - Code First_Demo1_Consol App Steps
  - Code First_Demo2_DataBindingWithEntityFrameworkt.txt

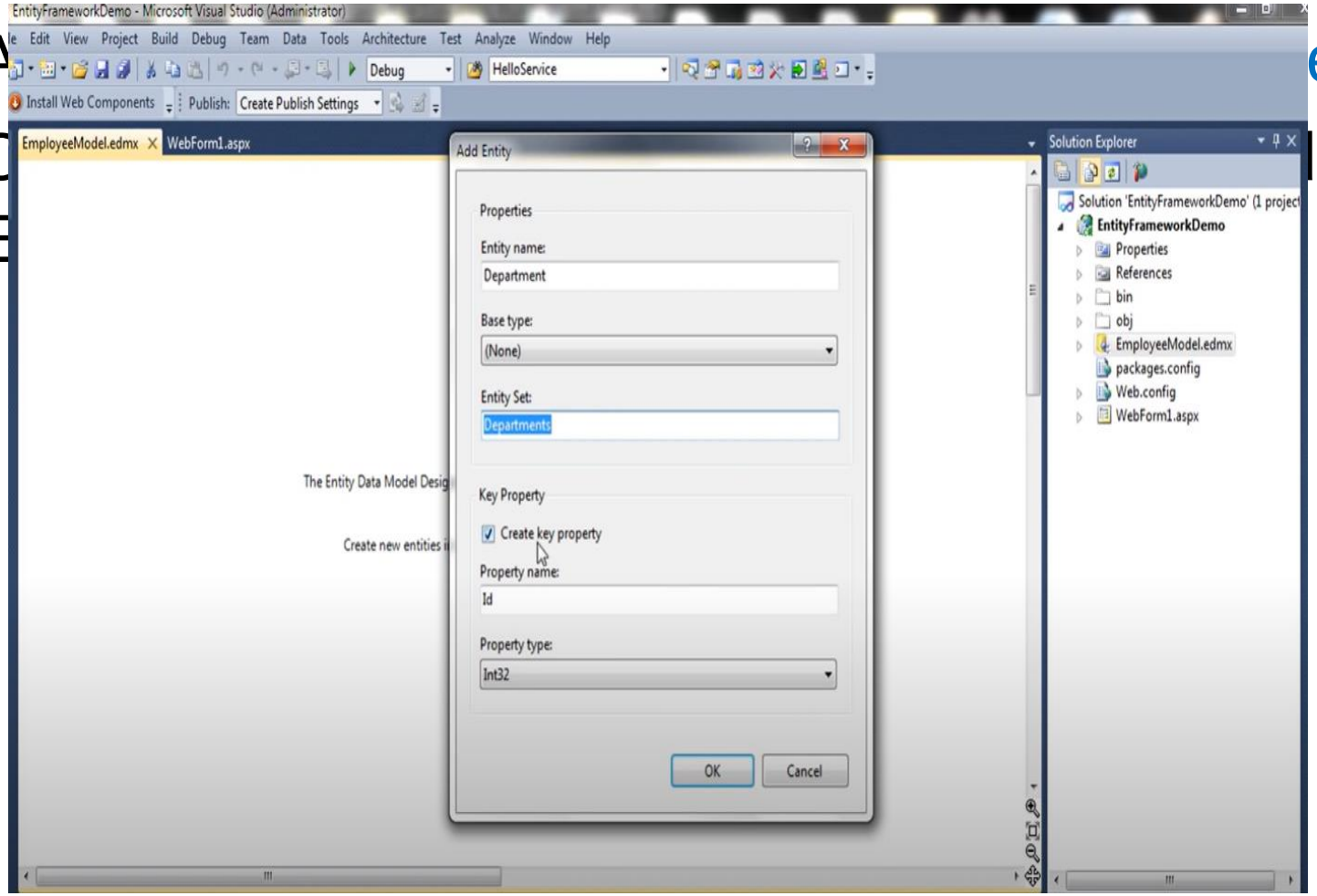
Model First Approach

Model First Approach

- Model First is great for when you're starting a new project **where the database doesn't even exist** yet.
- The **model is stored in an EDMX file** and can be viewed and edited in the Entity Framework Designer.
- In Model First,
  - you **define your model in an Entity Framework designer**
  - then **generate SQL**, which will **create database schema** to match your model and
  - then you **execute the SQL to create the schema** in your database.
- The **classes** that you interact with in your application are **automatically generated from the EDMX file.**

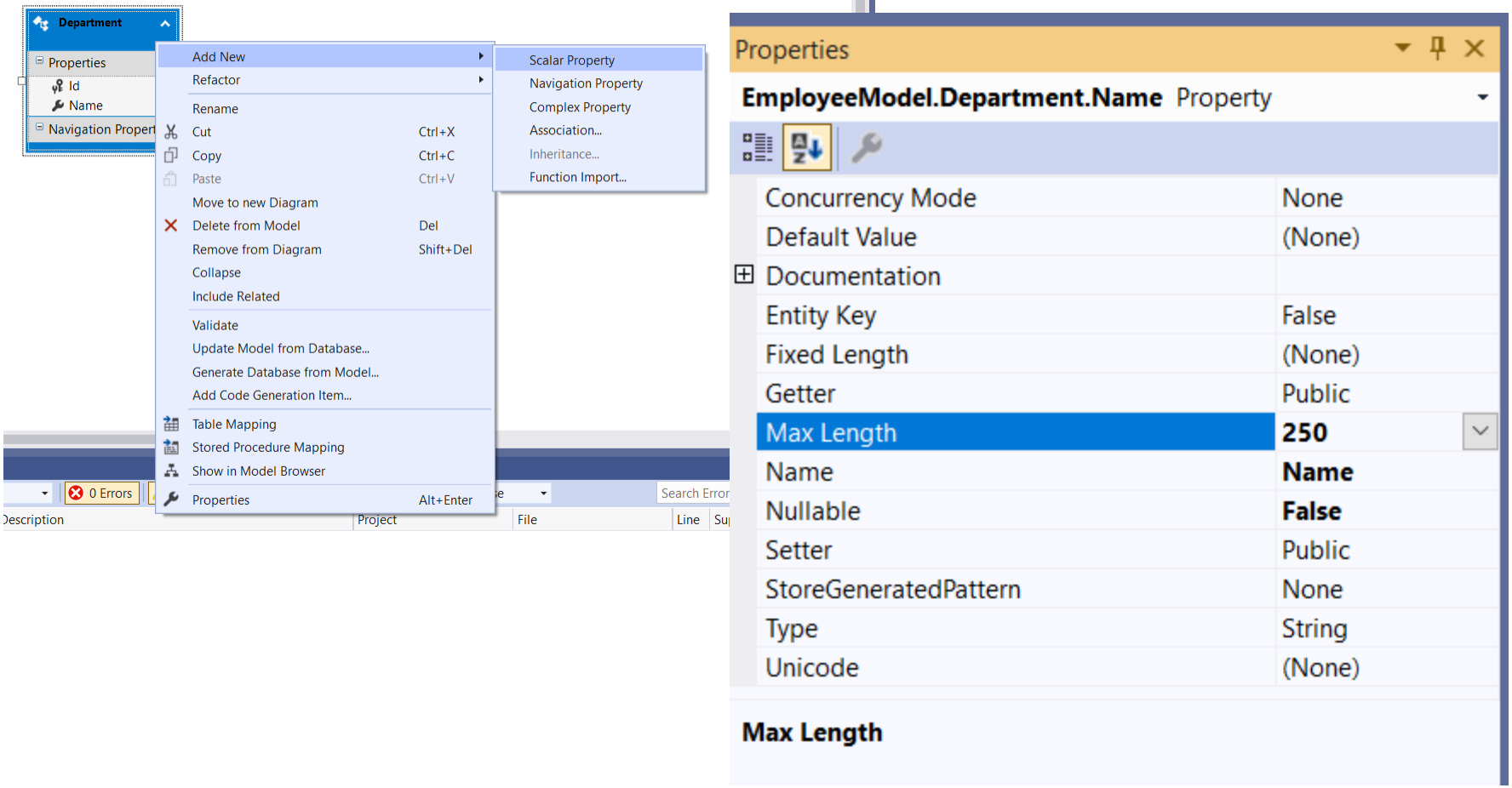
Model First

- Create Form base app
- A
- C

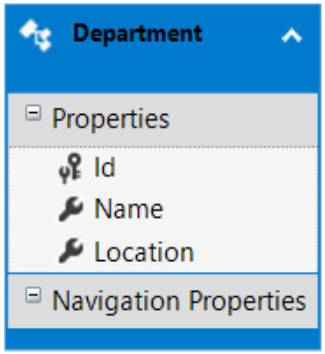


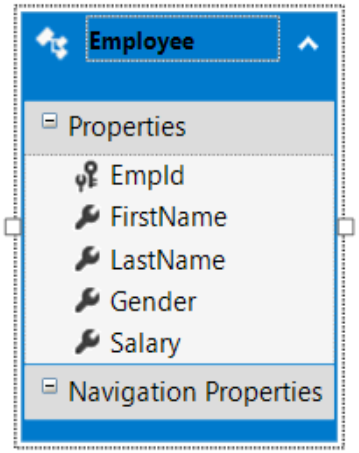
Configure Model

- Create Entities: Department and Employee
- Add additional columns like DeptID, Name, Budget
- Configure properties like max length, check getter and setter are public

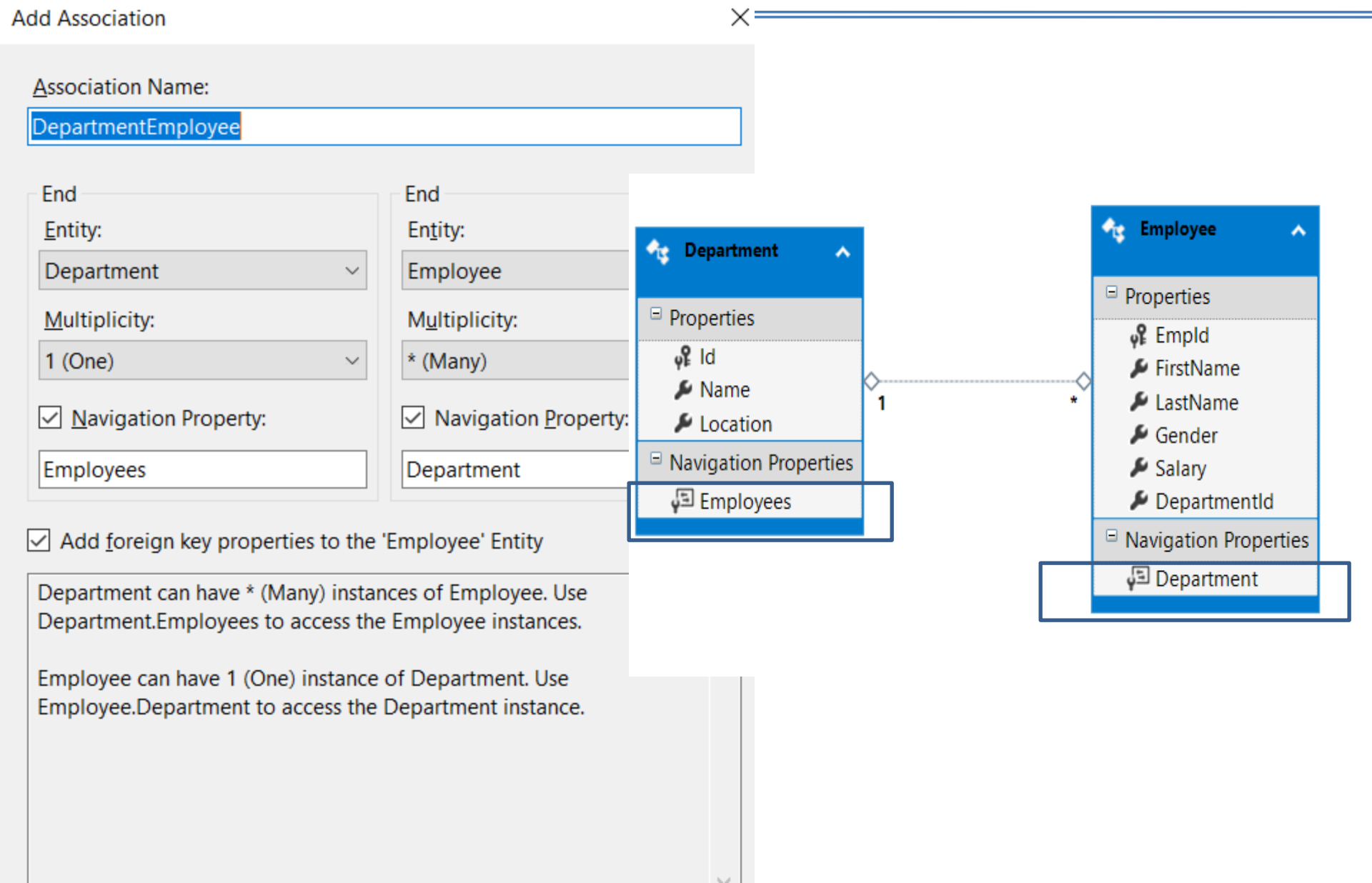


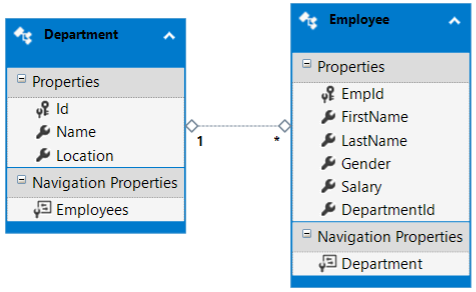
Configure Department and Employee entities





Add Association





Ensure Database is already created in system

Generate Database Wizard

Choose Your Data Connection

Which data connection should your application use to connect to the database?

laptop-fu5jm60v\mssqlserver2.ModelFirstEmployeeDB.dbo

New Connection...

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

- No, exclude sensitive data from the connection string. I will set it in my application code.
- Yes, include the sensitive data in the connection string.

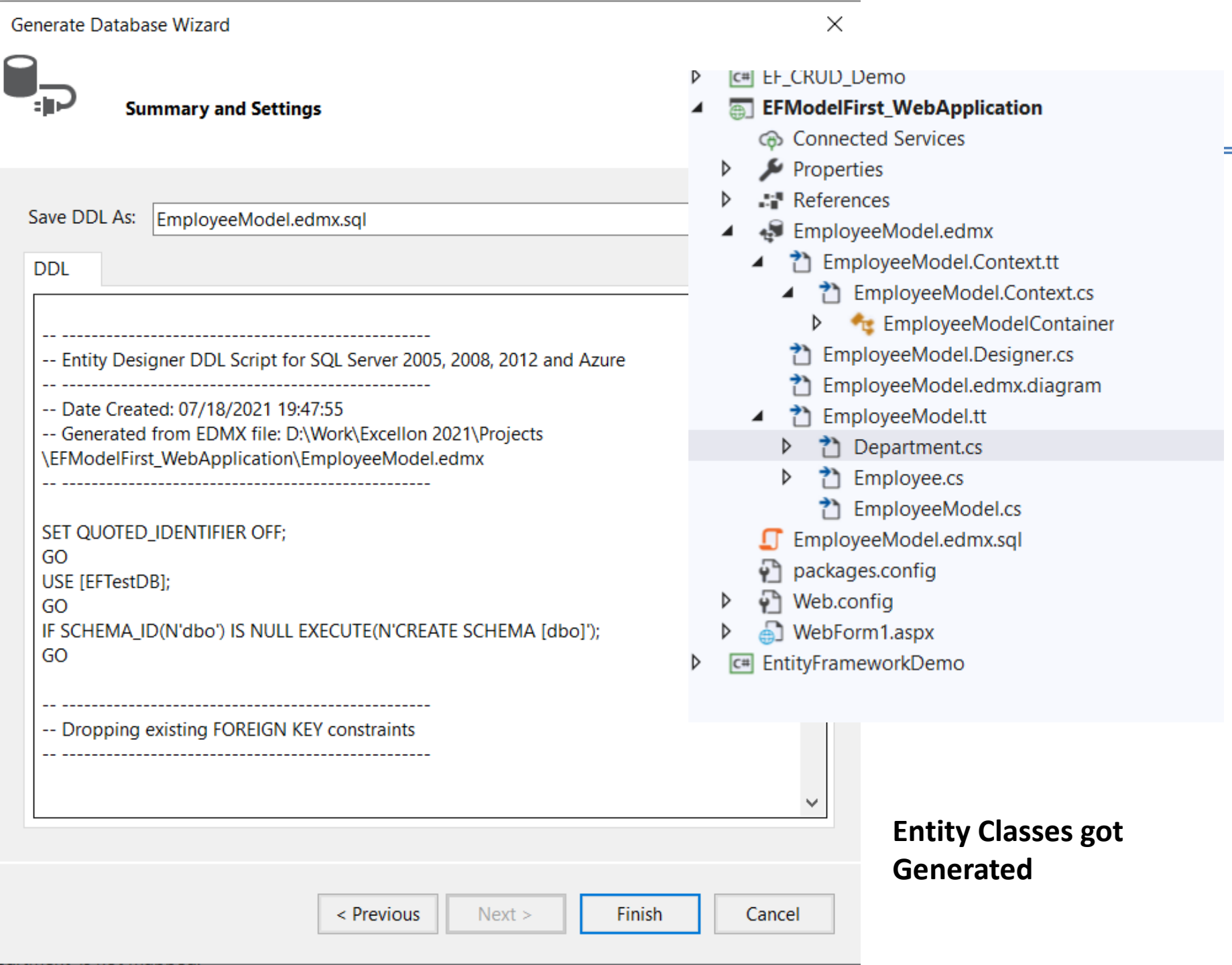
Connection string:

```
metadata=res://*/EmployeeModel.csdl|res://*/EmployeeModel.ssdl|
res://*/EmployeeModel.msl;provider=System.Data.SqlClient;provider connection string="data
source=LAPTOP-FU5JM60V\MSSQLSERVER2;initial catalog=ModelFirstEmployeeDB;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
```

☑ Save connection settings in Web.Config as:

EmployeeModelContainer

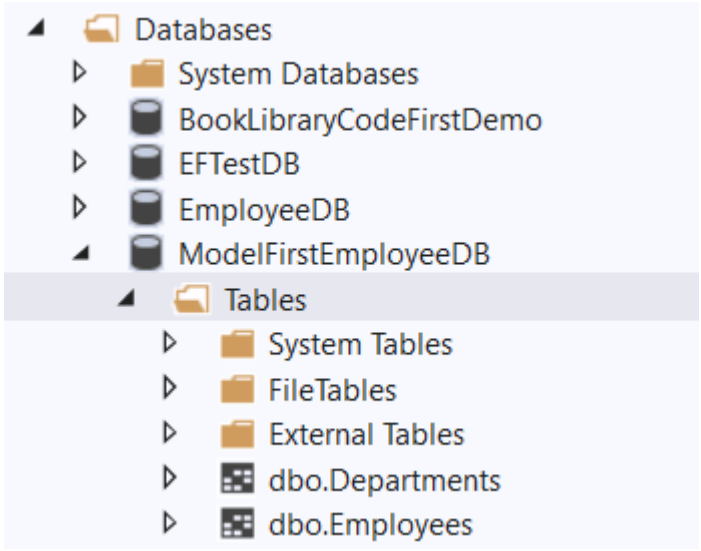
< Previous | Next > | Finish | Cancel



Entity Classes got Generated

CreateDB

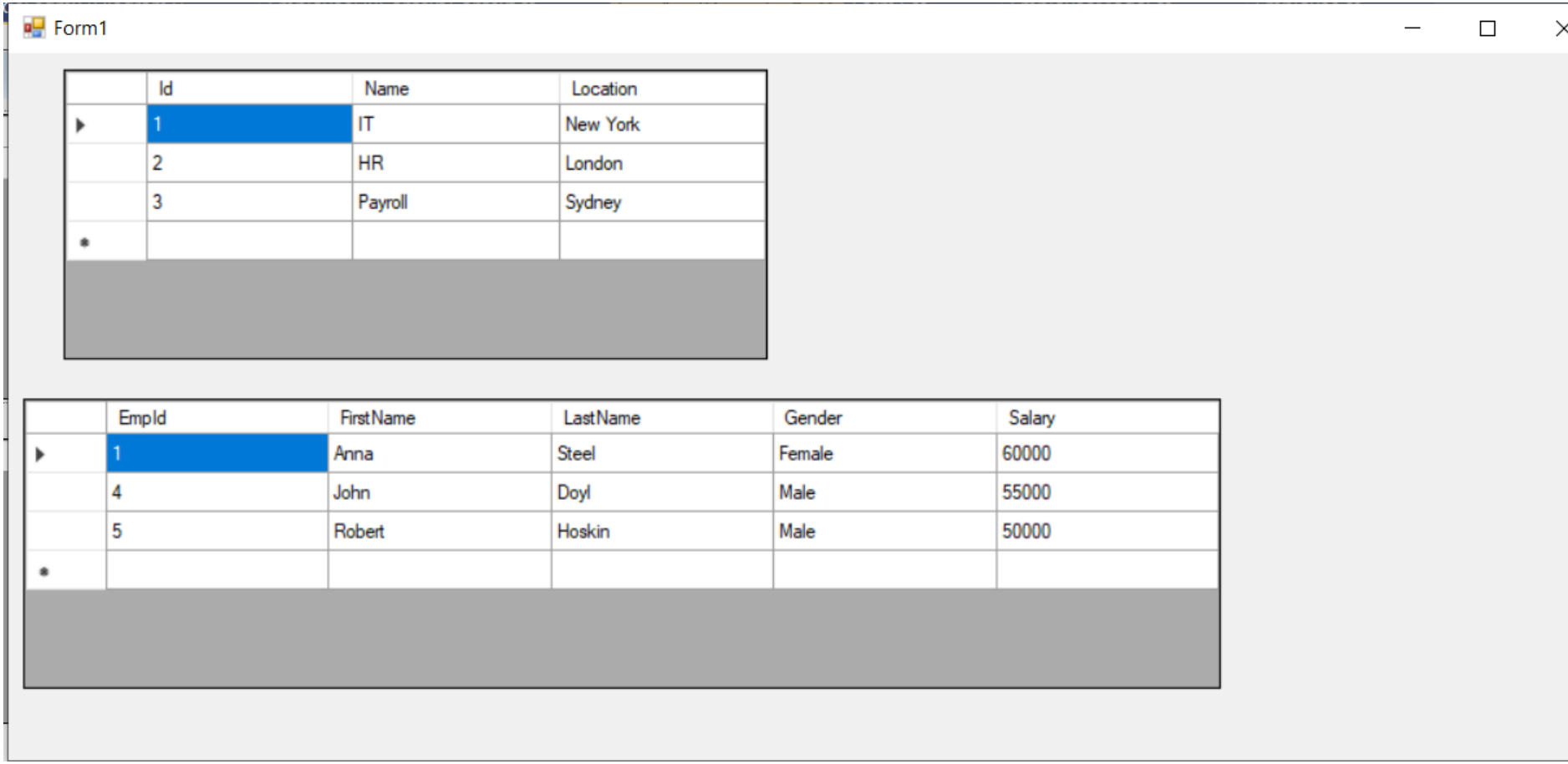
- Run <EmployeeModel>.**edmx.sql** to create tables in Database



Run DataPopulate.sql to populate tables

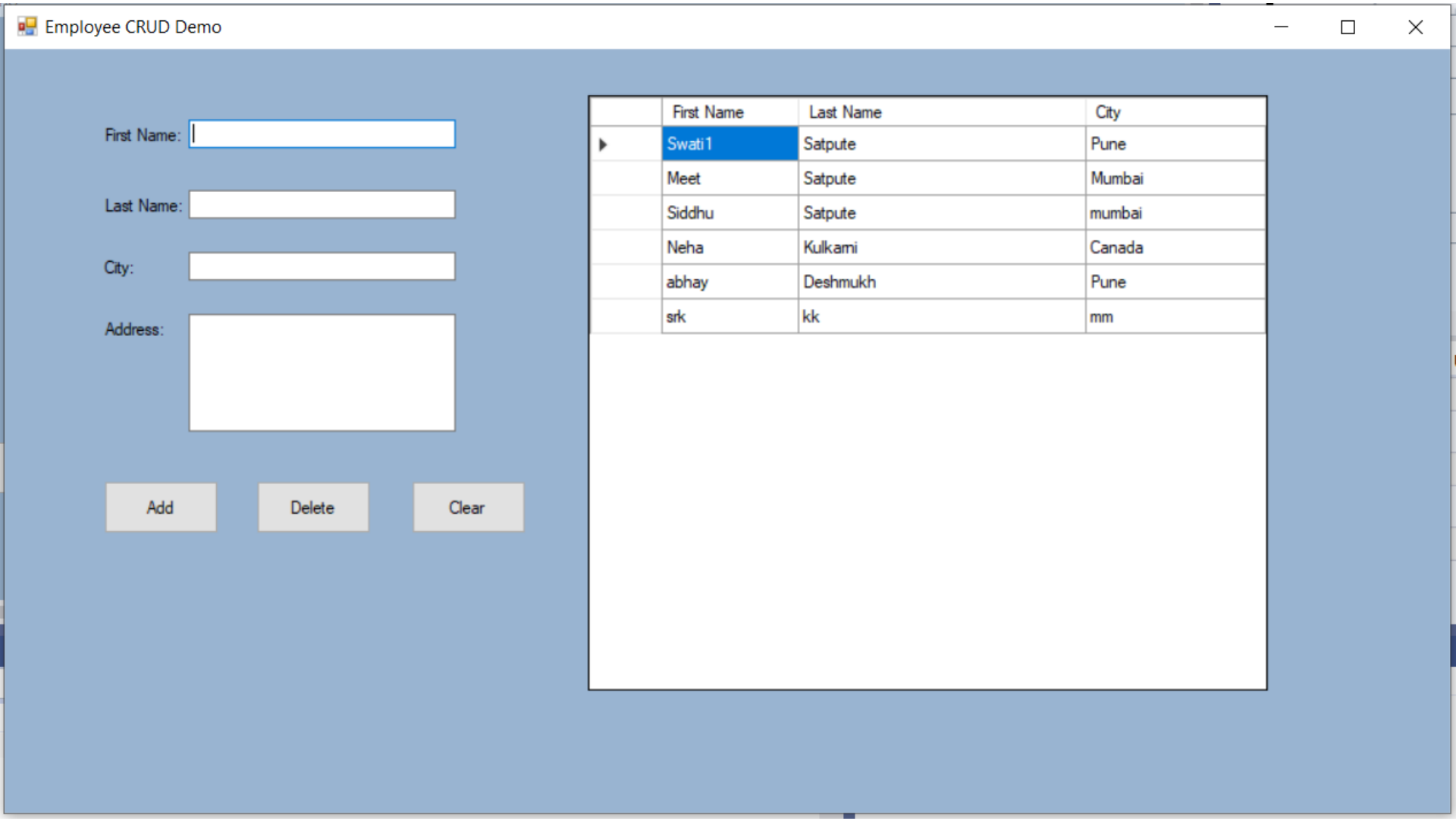
Configure Form

Demo



Database First Approach

Demo

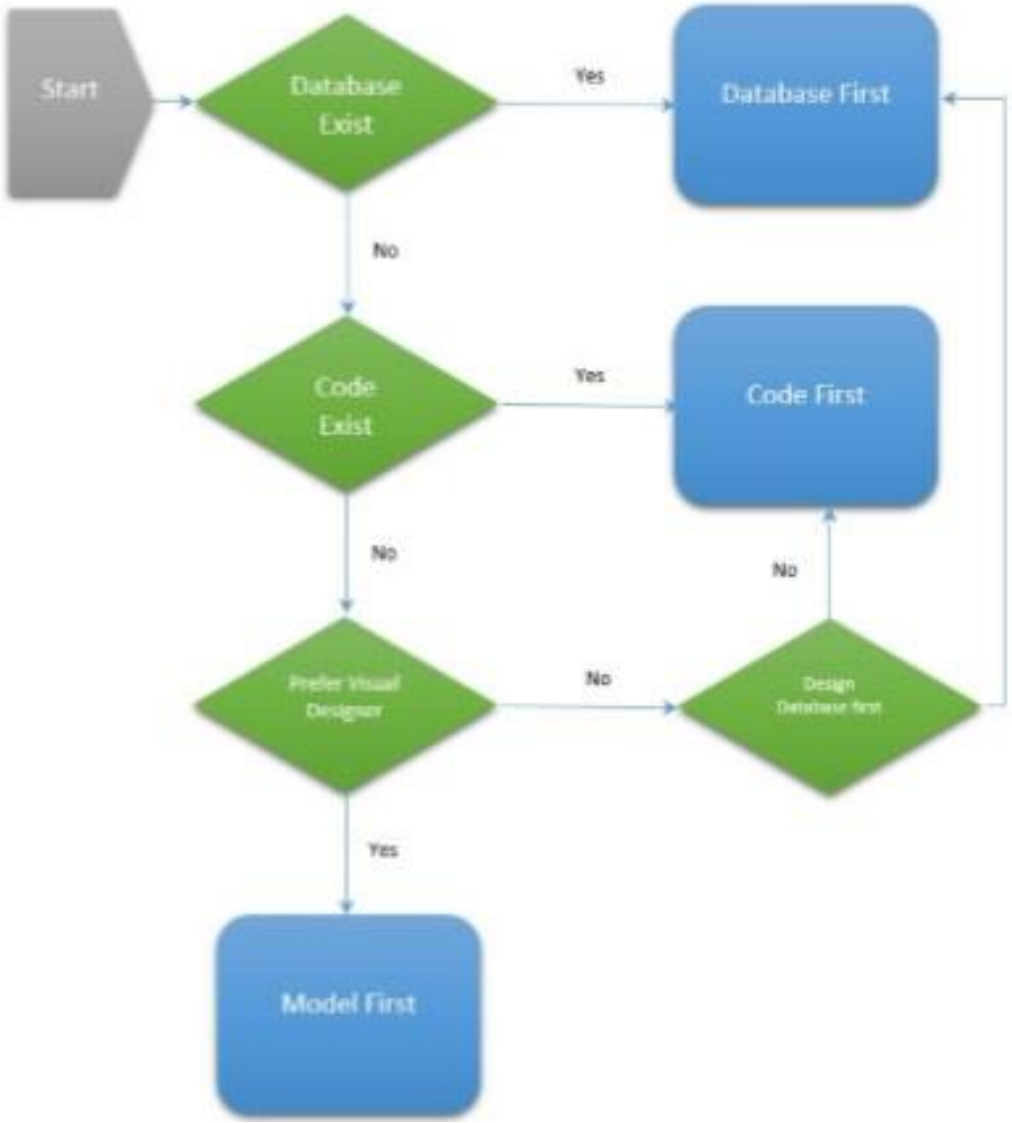

Employee CRUD Demo
First Name:
Last Name:
City:
Address:
Add
Delete
Clear
First Name
Last Name
City
Swati1
Satpute
Pune
Meet
Satpute
Mumbai
Siddhu
Satpute
mumbai
Neha
Kulkarni
Canada
abhay
Deshmukh
Pune
srk
kk
mm

How to choose developer approach?

Choose Developer Approach

- Database First approach and Model First approach uses EF - Designer View.
- Code First approach avoids EF - designer view and edmx thing completely, allowing focusing on entities.
- If Database exists, go with DB First Approach.
- If domain classes already exist and database is not created, go for code first approach.
- If want to use EF – Designer and Database in not present, go for Model First approach

Choose Developer Approach



CRUD operations

- D:\Work\Excellon 2021\Excellon_Projects\DataBindingWithEntityFramework\Category.cs

Stored Procedure

```
IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =
   OBJECT_ID(N'[dbo].[GetStudentGrades]') AND type in (N'P', N'PC'))

BEGIN

   EXEC dbo.sp_executesql @statement = N'
   CREATE PROCEDURE [dbo].[GetStudentGrades]
   @StudentID int
   AS
   SELECT EnrollmentID, Grade, CourseID, StudentID FROM dbo.StudentGrade
   WHERE StudentID = @StudentID
   '
END
GO
```

```
using (var context = new UniContextEntities()) {

   int studentID = 22;
   var studentGrades = context.GetStudentGrades(studentID);

   foreach (var student in studentGrades) {
      Console.WriteLine("Course ID: {0}, Title: {1}, Grade: {2} ",
         student.CourseID, student.Course.Title, student.Grade);
   }

   Console.ReadKey();
```

Transaction handling

Transaction handler is similar to ADO.NET

- BeginTransaction()
- Transaction.Commit();
- Transaction.Rollback();

- e.g.

```
var dbContextTransaction = context.Database.BeginTransaction()
....
dbContextTransaction.Commit();
dbContextTransaction.Rollback();
```

```
using (var dbContextTransaction = context.Database.BeginTransaction())
{

    try
    {
        Student student = new Student()
        {
            ID = 200,
            FirstMidName = "Ali",
            LastName = "Khan",
            EnrollmentDate = DateTime.Parse("2015-12-1")
        };

        context.Students.Add(student);

        context.Database.ExecuteSqlCommand(@"UPDATE Course SET Title =
                'Calculus'" + "WHERE CourseID = 1045");

        var query = context.Courses.Where(c ⇒ c.CourseID == 1045);

        foreach (var item in query)
        {
            Console.WriteLine(item.CourseID.ToString()
               + " " + item.Title + " " + item.Credits);
        }

        context.SaveChanges();

        dbContextTransaction.Commit();
    }
    catch (Exception)
    {
        dbContextTransaction.Rollback();
    }
```

Eager Loading & Lazy Loading

- Eager loading is the process whereby a **query for one type of entity also loads related entities** as part of the query.
- Eager loading is achieved by the use of the **Include method**.
- Related data be returned along with query results from the database.
- Initial data load will be higher.

```
using (var context = new UniContextEntities()) {
    // Load all students and related enrollments
    var students = context.Students
        .Include(s ⇒ s.Enrollments).ToList();

    foreach (var student in students) {
        string name = student.FirstMidName + " " + student.LastName;
        Console.WriteLine("ID: {0}, Name: {1}", student.ID, name);
```

Lazy Loading

- Lazy loading is the process whereby an entity or **collection of entities is automatically loaded from the database the first time that a property referring to the entity/entities is accessed**.

- Delaying the loading of related data, until specifically requested.

- Lazy loading is the default configuration.

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

Lazy Loading

```
public partial class Student {

   public Student() {
      this.Enrollments = new HashSet<Enrollment>();
   }

   public int ID { get; set; }
   public string LastName { get; set; }
   public string FirstMidName { get; set; }
   public System.DateTime EnrollmentDate { get; set; }

   public virtual ICollection<Enrollment> Enrollments { get; set; }
}
```

- when using the Student entity class, the related Enrollments will be loaded the first time the Enrollments navigation property is accessed.

Steps for Development approach

- **ModelFirst approach**
  - Define Entities, properties, association
  - create Empty database
  - we generated Sql by using "generate database from model“
    - **Note: DbContext and entity classes gets generated automatically after this step**

- Execute that sql on Database

Steps for Development approach

DB First

- Create project
- Add new item
- Select ADO.NET EntityModel and proceed the wizard

Code first approach

- create classes for entities
  -- student.cs
  -- Course.cs
- create dbContext class
  - schooldbContext.cs :DbContext
- CRUD operation in program.cs
- Note -
  database gets created in the backend with appname_<dbcontext>

References