



# Module "Data processing"

## Submodule "Data processing technologies"

ADO.NET Entity Framework Pt.1

# Contents

- ADO.NET EF
- EDM
- Entities and DbContext
- Entity relations
- Database First, Code First, Model First
- Configuration
- Initializers

# ADO.NET Entity Framework

# Entity Framework Overview

The Entity Framework is a set of technologies in ADO.NET that support the development of data-oriented software applications.

- Work with data in the form of domain-specific objects and properties
- Work at a higher level of abstraction
- Write less code

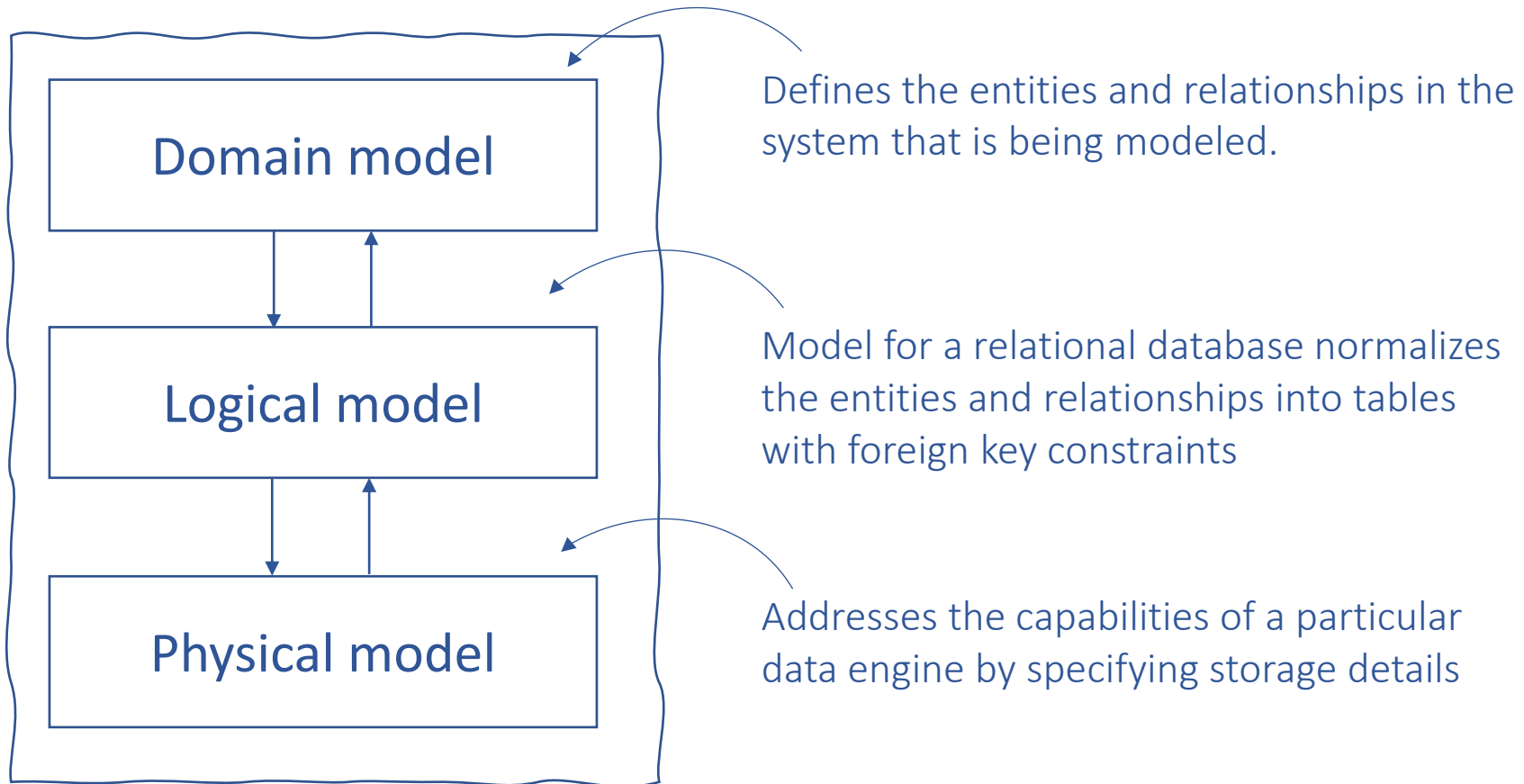
“...How am I supposed to implement this domain model if storage might change, also I’m so done with writing SQL queries...”



“your ever heard about the entity framework?”

# Entity Framework Overview

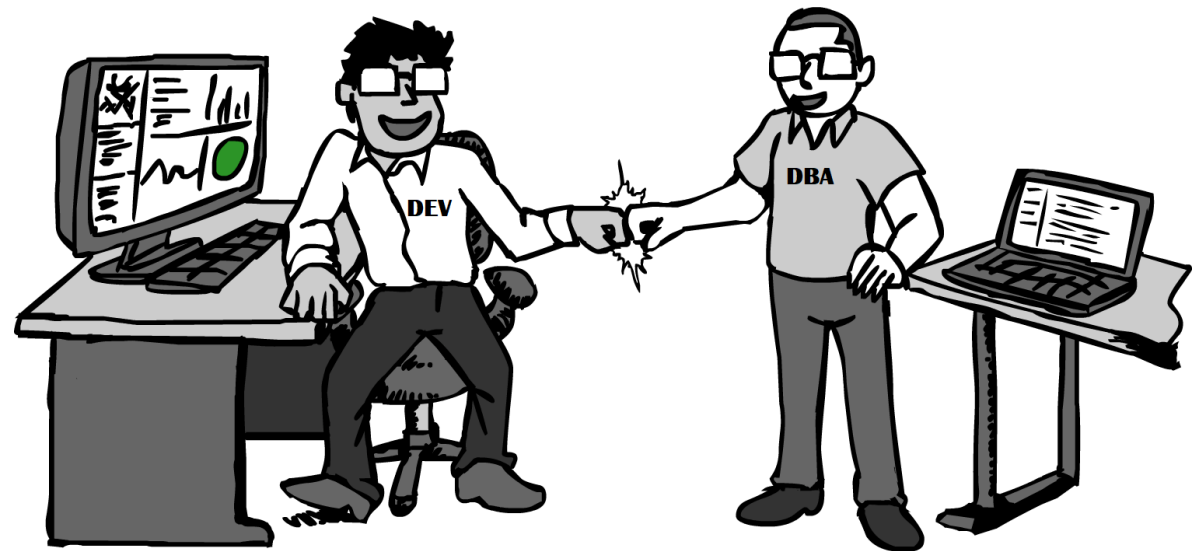
Common application\service design approach – split the system into three parts:



## Give Life to Models

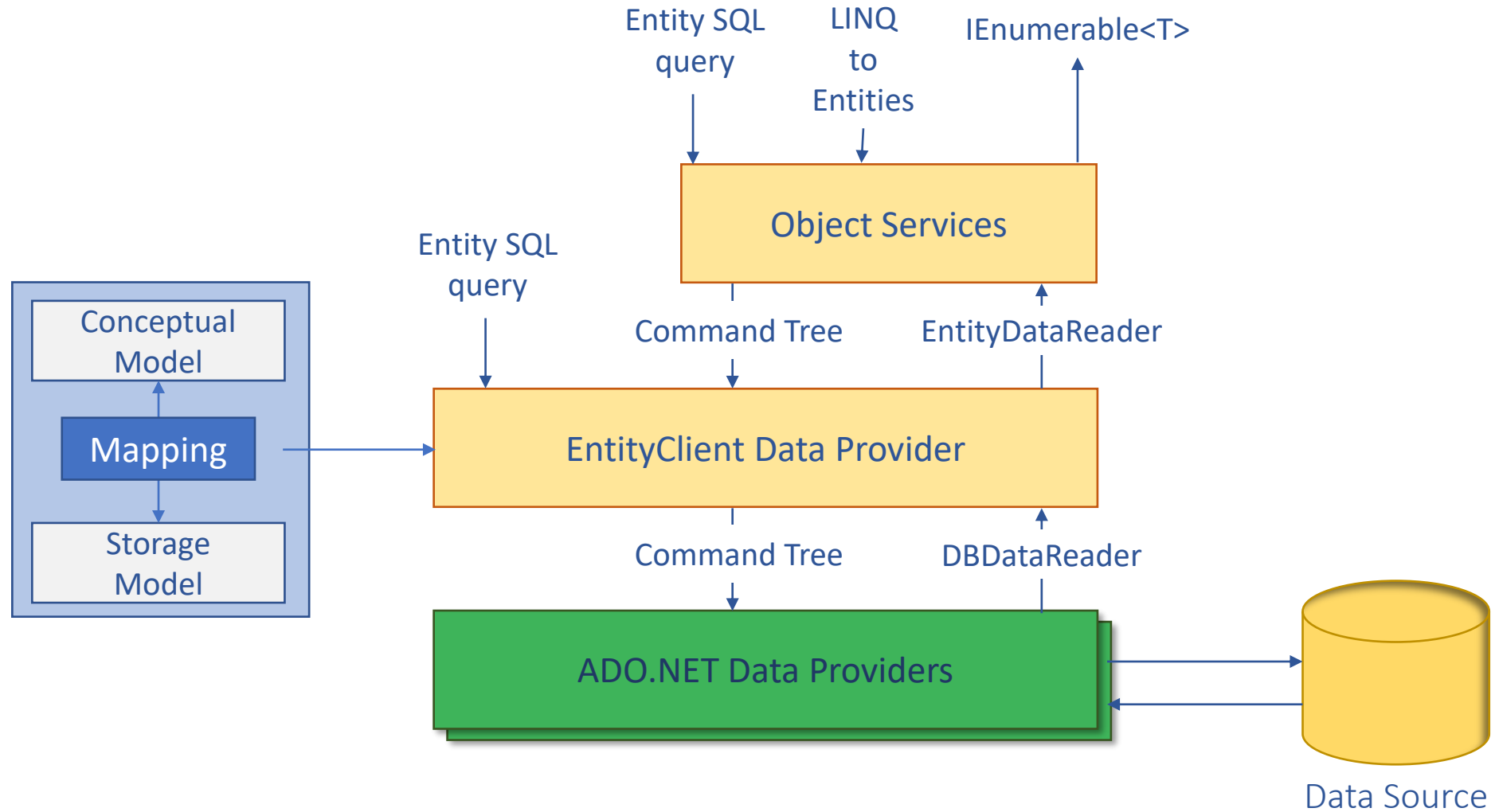
The physical model is refined by database administrators to improve performance

Programmers writing application code primarily confine themselves to working with the logical model by writing SQL queries and calling stored procedures



# Entity Framework

\*\*



## Define Models

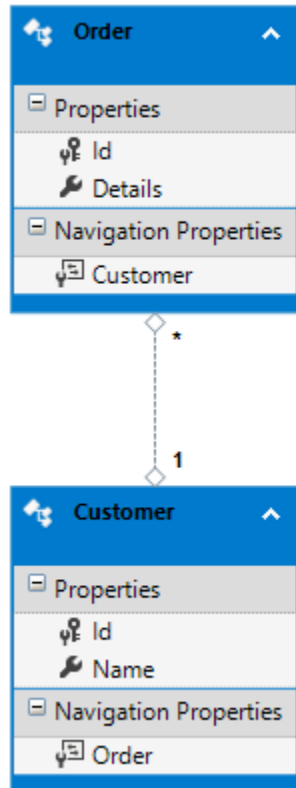
When working with the Entity Data Model Tools, the conceptual model, the storage model, and the mappings between the two are expressed in XML-based schemas and defined in files that have corresponding name extensions:

- Conceptual schema definition language (CSDL) defines the conceptual model.
- Store schema definition language (SSDL) defines the storage model
- Mapping specification language (MSL) defines the mappings between the storage and conceptual models



# Map Objects to Data

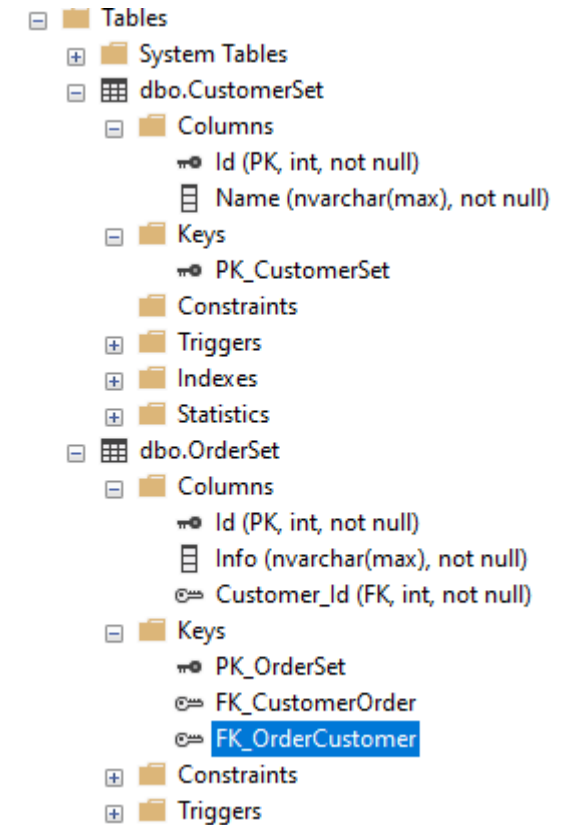
Define database representation with models:



```
public partial class Customer
{
    public Customer()
    {
        this.Order = new HashSet<Order>();
    }

    public int Id { get; set; }
    public string Name { get; set; }

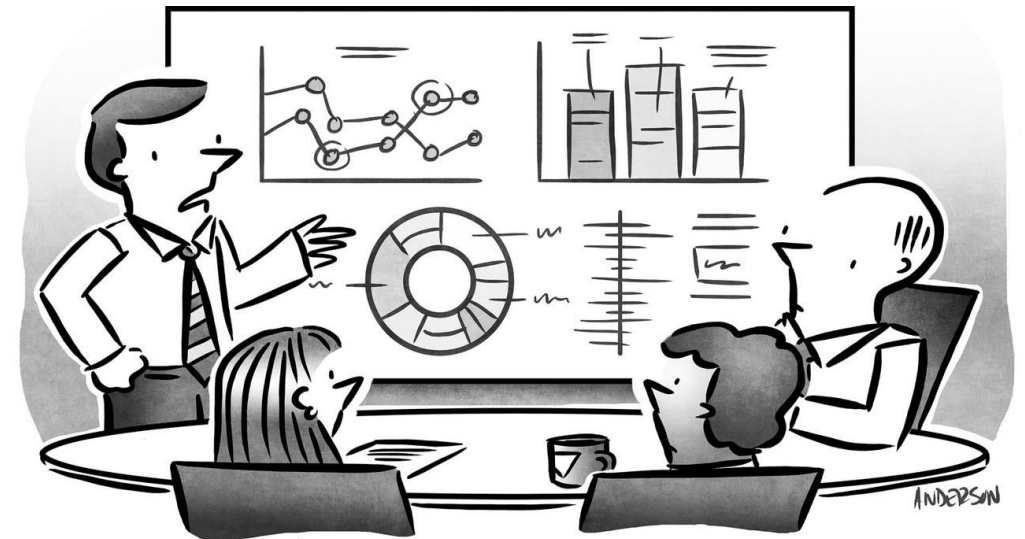
    public virtual ICollection<Order> Order { get; set; }
}
```



## Access and change entity data

The Entity Framework provides the following ways to query a conceptual model and return objects:

- LINQ to Entities. Provides Language-Integrated Query (LINQ) support for querying entity types that are defined in a conceptual model.
- Entity SQL. A storage-independent dialect of SQL that works directly with entities in the conceptual model and that supports Entity Data Model concepts



“How much data you want?” – “YES!”

# Entity Data Model

# Entity Data Model

The Entity Data Model (EDM) is a set of concepts that describe the **structure of data, regardless of its stored form.**

The EDM borrows from the Entity-Relationship Model described by Peter Chen in 1976, but it also builds on the Entity-Relationship Model and extends its traditional uses.

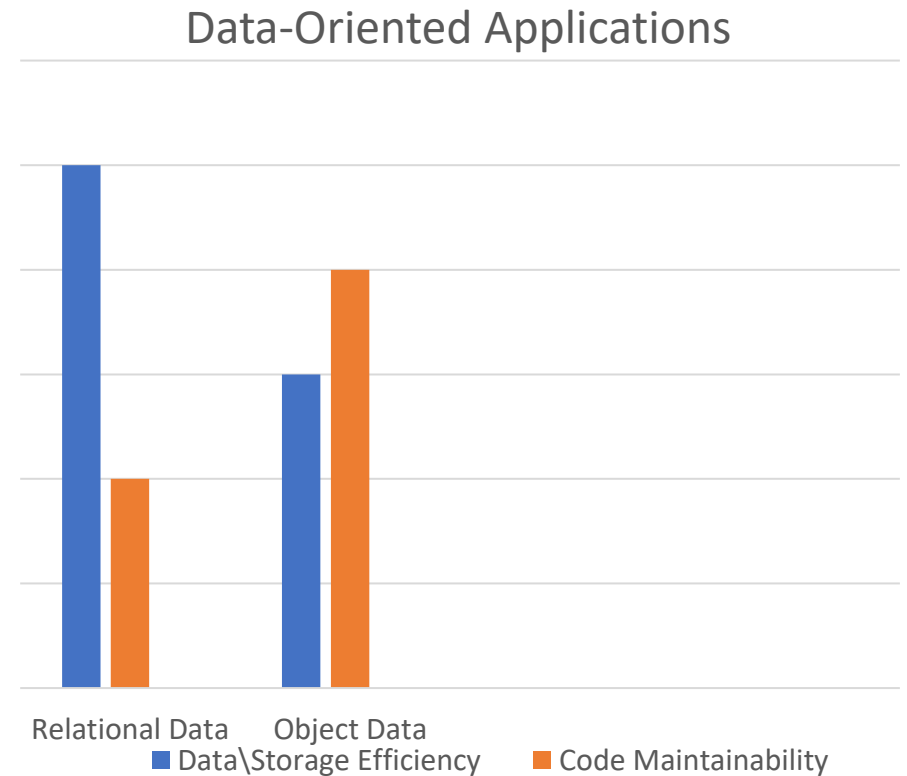


Dr. Peter Chen

# Entity Data Model

Challenges in data modeling:

- When designing a data-oriented application, the challenge is to write efficient and maintainable code without sacrificing efficient data access, storage, and scalability.
- When data has a relational structure, data access, storage, and scalability are very efficient, but writing efficient and maintainable code becomes more difficult



## Entity Data Model. Entity Type

Entity type - fundamental building block for describing the structure of data with the Entity Data Model.

In a conceptual model, entity types are constructed from **properties** and describe the structure of **top-level concepts**, such as a **customers** and **orders** in a business application.



## Entity Data Model. Association Type

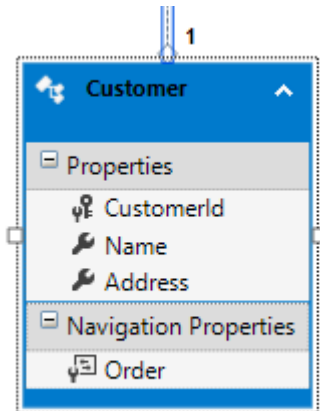
An association type (association) is another fundamental building block for describing relationships in the Entity Data Model.

Each association end also 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 (\*).

## Entity Data Model. Property

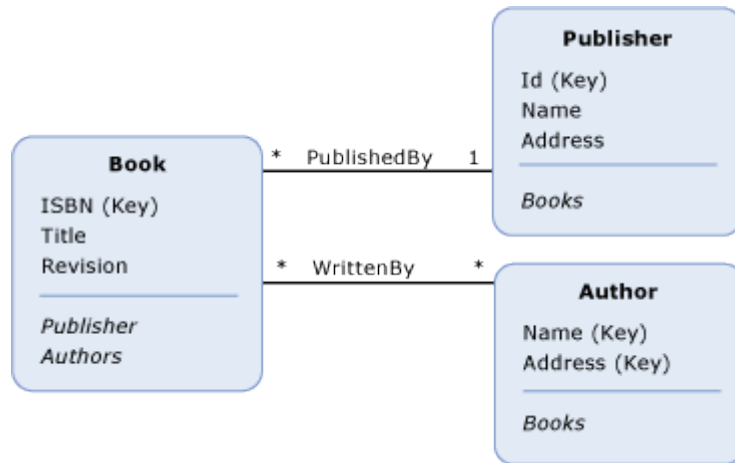
Entity types contain properties that define their structure and characteristics. For example, a **Customer** entity type may have properties such as **CustomerId**, **Name**, and **Address**.





# Representations of a Conceptual Model

A *conceptual model* is a specific representation of the structure of some data as entities and relationships.



```
<Schema xmlns="http://schemas.microsoft.com/ado/2008/09/edm"
  xmlns:cg="http://schemas.microsoft.com/ado/2006/04/codegeneration"
  xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"
  Namespace="BooksModel" Alias="Self">
  <EntityContainer Name="BooksContainer" >
    <EntitySet Name="Books" EntityType="BooksModel.Book" />
    <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
    <EntitySet Name="Authors" EntityType="BooksModel.Author" />
    <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
      <End Role="Book" EntitySet="Books" />
      <End Role="Publisher" EntitySet="Publishers" />
    </AssociationSet>
    <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
      <End Role="Book" EntitySet="Books" />
      <End Role="Author" EntitySet="Authors" />
    </AssociationSet>
  </EntityContainer>
  <EntityType Name="Book">
    <Key>
      <PropertyRef Name="ISBN" />
    </Key>
    <Property Type="String" Name="ISBN" Nullable="false" />
    <Property Type="String" Name="Title" Nullable="false" />
    <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
    <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
      FromRole="Book" ToRole="Publisher" />
    <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
      FromRole="Book" ToRole="Author" />
  </EntityType>
  <EntityType Name="Publisher">
    <Key>
      <PropertyRef Name="Id" />
    </Key>
    <Property Type="String" Name="Id" Nullable="false" />
```

# DbContext and Entites

## Working with DbContext

You can use a **DbContext** associated to a model to:

- Write and execute queries
- Materialize query results as entity objects
- Track changes that are made to those objects
- Persist object changes back on the database
- Bind objects in memory to UI controls

## Defining a DbContext derived class

The recommended way to work with context is to define a class that derives from **DbContext** and exposes **DbSet** properties that represent collections of the specified entities in the context.

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

Once you have a context, you would query for, add (using **Add** or **Attach** methods ) or remove (using **Remove**) entities in the context through these properties.

## DbContext class

Note that just accessing a property will not execute the query. A query is executed when:

- It is enumerated by a **foreach** statement.
- It is enumerated by a collection operation such as **ToArray**, **ToDictionary**, or **ToList**.
- LINQ operators such as **First** or **Any** are specified in the outermost part of the query.
- One of the following methods are called: the **Load** extension method, **DbEntityEntry.Reload**, **Database.ExecuteSqlCommand**, and **DbSet<T>.Find**, if an entity with the specified key is not found already loaded in the context.

## Context Lifetime

The lifetime of the context begins when the instance is created and ends when the instance is either disposed or garbage-collected.

```
public void UseProducts()
{
    using (var context = new ProductContext())
    {
        // Perform data access using the context
    }
}
```

## Context Lifetime and Connections

Some general guidelines when deciding on the lifetime of the context:

- When working with Web applications, use a context instance per request.
- When working with Windows Presentation Foundation (WPF) or Windows Forms, use a context instance per form.
- If the context instance is created by a dependency injection container, it is usually the responsibility of the container to dispose the context.
- If the context is created in application code, remember to dispose of the context when it is no longer required.

By default, the context manages connections to the database. The context opens and closes connections as needed. For example, the context opens a connection to execute a query, and then closes the connection when all the result sets have been processed.

# Entities

Entity in Entity Framework – is a class that is being mapped to database table.

Example:

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

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

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

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

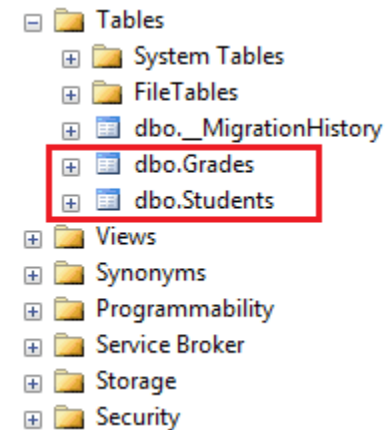


# Entities

The class becomes an entities when they are included as DbSet<TEntity> properties in a context class

```
public class SchoolContext : DbContext
{
    public SchoolContext() : base("schooldb") { }

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

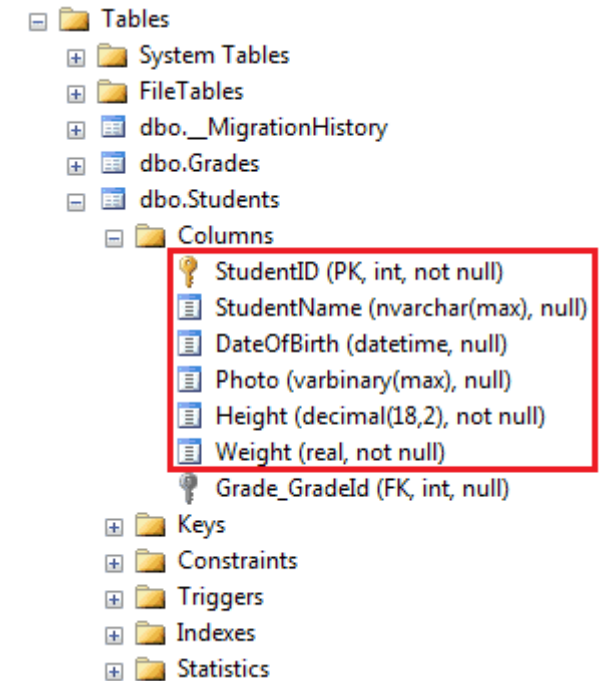


## Entities. Scalar Properties

The primitive type properties are called scalar properties. Each scalar property maps to a column in the database table which stores an actual data.

```
public class Student
{
    // scalar properties
    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; }

    //reference navigation properties
    public Grade Grade { get; set; }
}
```

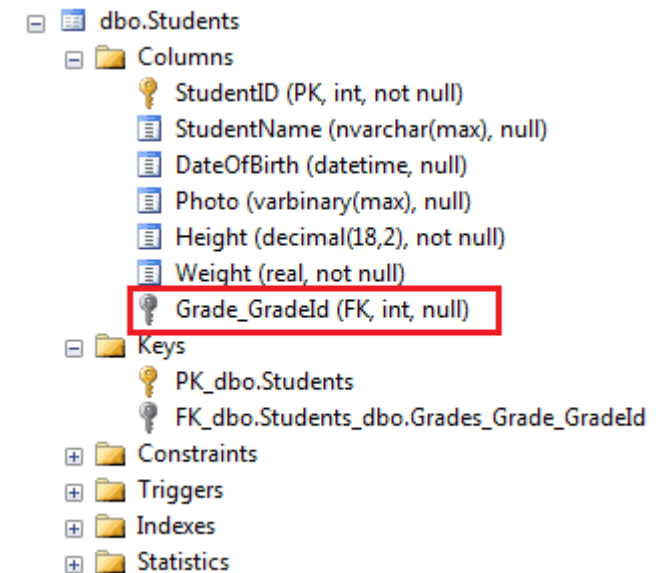


## Entities. Navigation Properties

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

```
public class Student
{
    // scalar properties
    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; }

    //reference navigation properties
    public Grade Grade { get; set; }
}
```

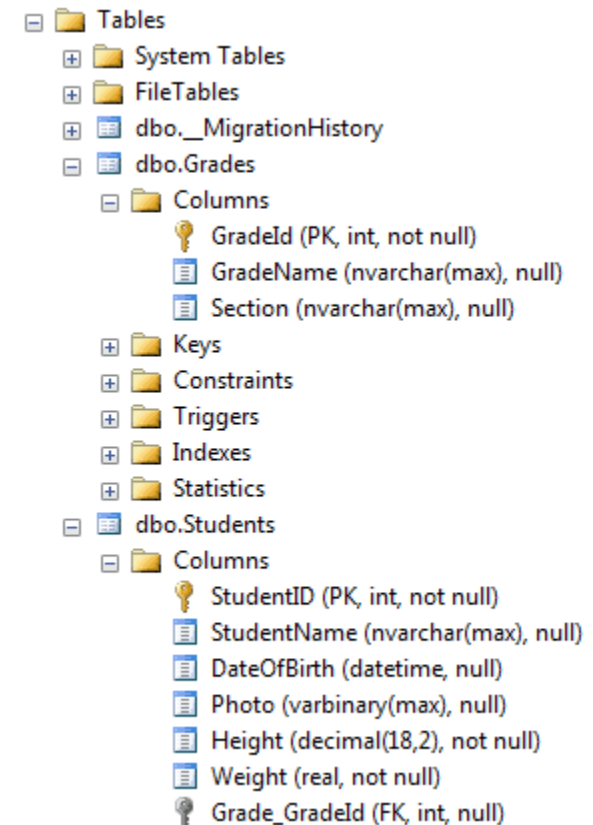


## Entities. Navigation Properties

If an entity includes a property of generic collection of an entity type, it is called a collection navigation property. It represents multiplicity of many (\*).

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

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



# Types of Entities in Entity Framework

There are two types of Entities in Entity Framework: POCO Entities and Dynamic Proxy Entities.

POCO Entities (Plain Old CLR Object)

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

# Types of Entities in Entity Framework

Dynamic Proxy Entities (POCO Proxy)

```
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; }

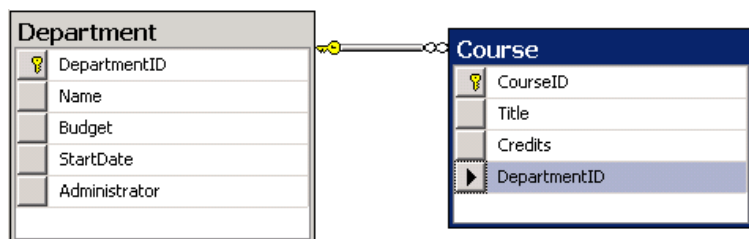
    public virtual StudentAddress StudentAddress { get; set; }
    public virtual Grade Grade { get; set; }
}
```

Dynamic Proxy is a runtime proxy class which wraps POCO entity. Dynamic proxy entities allow lazy loading.

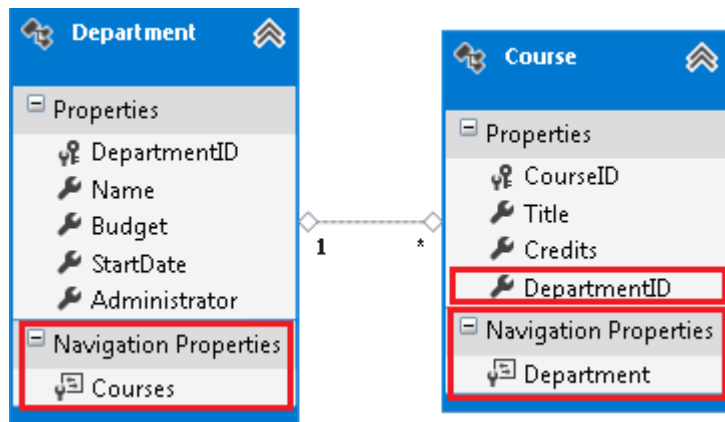
# Relationships

# Relationships in EF

Database tables:



Entity Framework Designer model:



```
public class Course
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    public int DepartmentID { get; set; }
    public virtual Department Department { get; set; }
}
```

```
public class Department
{
    public Department()
    {
        this.Courses = new HashSet<Course>();
    }
    public int DepartmentID { get; set; }
    public string Name { get; set; }
    public decimal Budget { get; set; }
    public DateTime StartDate { get; set; }
    public int? Administrator { get; set; }
    public virtual ICollection<Course> Courses { get; set; }
}
```



## Creating and modifying relationships

In a foreign key association, when you change the relationship, the state of a dependent object with an **EntityState.Unchanged** state changes to **EntityState.Modified**.

In an independent relationship, changing the relationship does not update the state of the dependent object.

You can assign a new value to a foreign key property:

```
course.DepartmentID = newCourse.DepartmentID;
```

Remove a relationship by setting the foreign key to null:

```
course.DepartmentID = null;
```

## Creating and modifying relationships

You can assign a new object to a navigation property:

```
course.Department = department;
```

\*To delete the relationship, set the navigation property to *null*:

```
context.Entry(course).Reference(c => c.Department).Load();  
course.Department = null;
```

\*\*You can set the relationship to *null* without loading the related end:

```
context.Entry(course).Reference(c => c.Department).CurrentValue = null;
```

\*.NET 4.0

\*\*EF 5.0 based on .NET 4.5 onward

## Loading related objects

In Entity Framework you commonly use navigation properties to load entities that are related to the returned entity by the defined association.

```
// Get the course where currently DepartmentID = 2.
Course course = context.Courses.First(c => c.DepartmentID == 2);

// Use DepartmentID foreign key property
// to change the association.
course.DepartmentID = 3;

// Load the related Department where DepartmentID = 3
context.Entry(course).Reference(c => c.Department).Load();
```

# Entity States

## Entity states and SaveChanges

An entity can be in one of five states as defined by the **EntityState** enumeration. These states are:

- **Added**: the entity is being tracked by the context but does not yet exist in the database
- **Unchanged**: the entity is being tracked by the context and exists in the database, and its property values have not changed from the values in the database
- **Modified**: the entity is being tracked by the context and exists in the database, and some or all of its property values have been modified
- **Deleted**: the entity is being tracked by the context and exists in the database, but has been marked for deletion from the database the next time SaveChanges is called
- **Detached**: the entity is not being tracked by the context

## Entity states and SaveChanges

SaveChanges does different things for entities in different states:

- Unchanged entities are not touched by SaveChanges. Updates are not sent to the database for entities in the Unchanged state.
- Added entities are inserted into the database and then become Unchanged when SaveChanges returns.
- Modified entities are updated in the database and then become Unchanged when SaveChanges returns.
- Deleted entities are deleted from the database and are then detached from the context.

## Adding a new entity to the context

A new entity can be added to the context by calling the **Add** method on **DbSet**:

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Name = "ADO.NET Blog" };
    context.Blogs.Add(blog);
    context.SaveChanges();
}
```

Another way to add a new entity to the context is to change its state to **Added**.

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Name = "ADO.NET Blog" };
    context.Entry(blog).State = EntityState.Added;
    context.SaveChanges();
}
```

## Adding a new entity to the context

Finally, you can add a new entity to the context by hooking it up to another entity that is already being tracked.

```
using (var context = new BloggingContext())
{
    // Add a new User by setting a reference from a tracked Blog
    var blog = context.Blogs.Find(1);
    blog.Owner = new User { UserName = "johndoe1987" };

    // Add a new Post by adding to the collection of a tracked Blog
    blog.Posts.Add(new Post { Name = "How to Add Entities" });

    context.SaveChanges();
}
```



## Attaching an existing entity to the context

If you have an entity that you know already exists in the database but which is not currently being tracked by the context then you can tell the context to track the entity using the **Attach** method on **DbSet**.

```
var existingBlog = new Blog { Id = 1, Name = "ADO.NET Blog" };

using (var context = new BloggingContext())
{
    context.Blogs.Attach(existingBlog);

    // Do some more work...

    context.SaveChanges();
}
```

## Attaching an existing entity to the context

Another way to attach an existing entity to the context is to change its state to **Unchanged**

```
existingBlog = new Blog { Id = 1, Name = "ADO.NET Blog" };

using (var context = new BloggingContext())
{
    context.Entry(existingBlog).State = EntityState.Unchanged;

    // Do some more work...

    context.SaveChanges();
}
```

## Changing the state of a tracked entity

You can change the state of an entity that is already being tracked by setting the State property on its entry

```
existingBlog = new Blog { Id = 1, Name = "ADO.NET Blog" };

using (var context = new BloggingContext())
{
    context.Blogs.Attach(existingBlog);
    context.Entry(existingBlog).State = EntityState.Unchanged;

    // Do some more work...

    context.SaveChanges();
}
```

## Insert or update pattern

Example:

```
public static void InsertOrUpdate(Blog blog)
{
    using (var context = new BloggingContext())
    {
        context.Entry(blog).State = blog.Id == 0 ?
            EntityState.Added :
            EntityState.Modified;

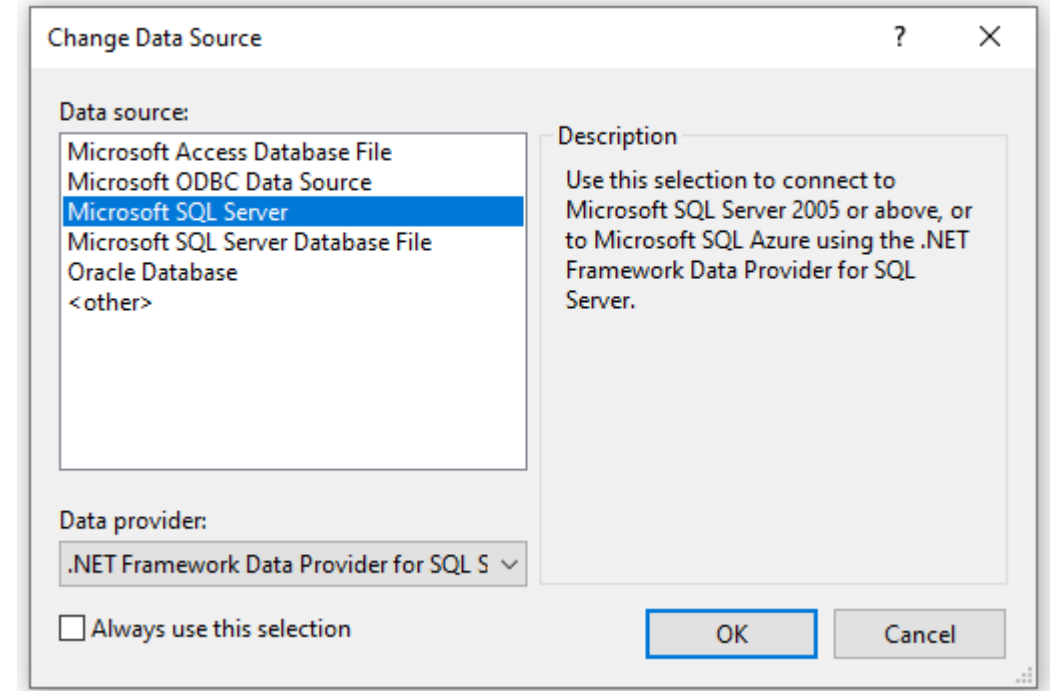
        context.SaveChanges();
    }
}
```

# EF Workflows

## Database First

## Create an Existing Database

- Open Visual Studio
- View -> Server Explorer
- Right click on Data Connections -> Add Connection...
- If you haven't connected to a database from Server Explorer before you'll need to select Microsoft SQL Server as the data source



## Create an Existing Database

Add Connection

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source: Microsoft SQL Server (SqlClient) Change...

Server name: (localdb)\MSSQLLocalDB Refresh

Log on to the server

Authentication: Windows Authentication

User name: Password: Save my password

Connect to a database

Select or enter a database name: DatabaseFirst.Blogging

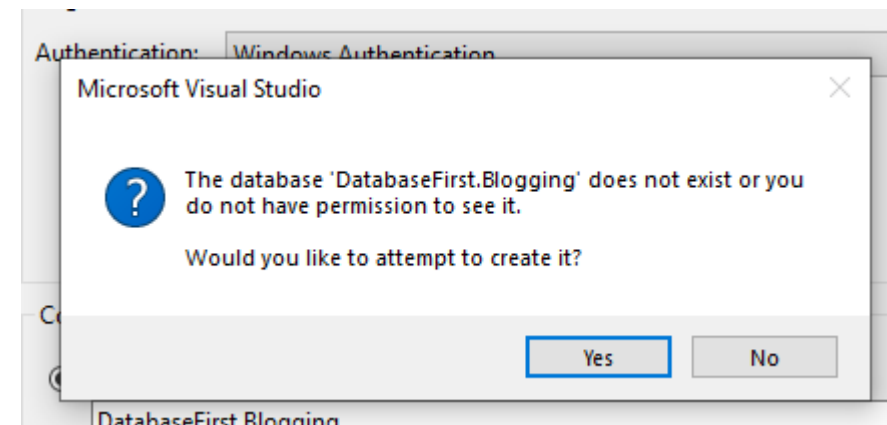
Attach a database file: Browse...

Logical name:

Advanced...

Test Connection OK Cancel

- Connect to either LocalDB or SQL Express, depending on which one you have installed, and enter the database name



## Create an Existing Database

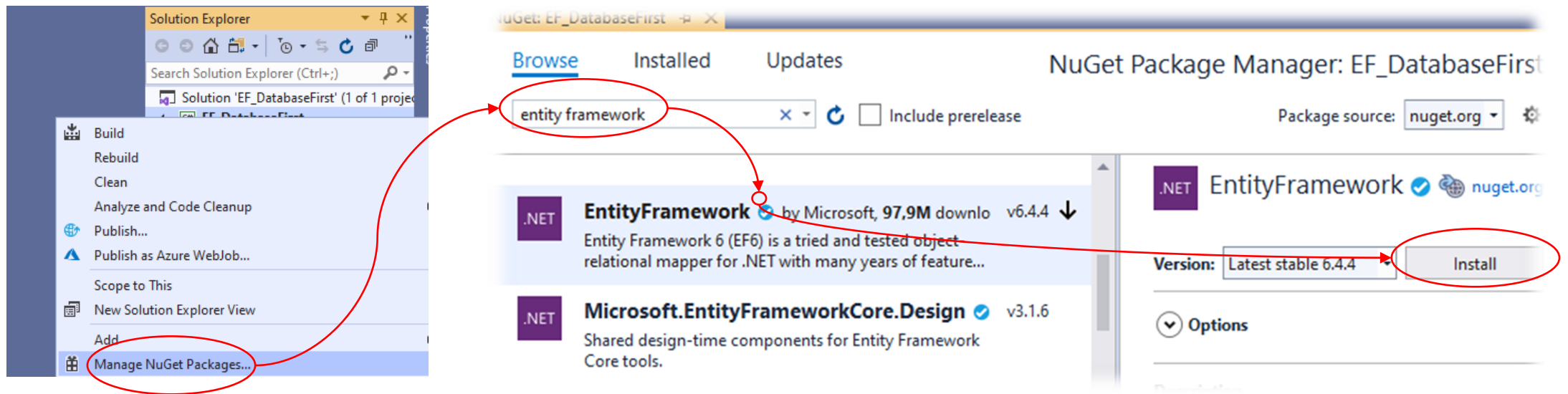
Add a few related tables into it

```
1 CREATE TABLE [dbo].[Blogs] (  
2     [BlogId] INT IDENTITY (1, 1) NOT NULL,  
3     [Name] NVARCHAR (200) NULL,  
4     [Url] NVARCHAR (200) NULL,  
5     CONSTRAINT [PK_dbo.Blogs] PRIMARY KEY CLUSTERED ([BlogId] ASC)  
6 );  
7  
8 CREATE TABLE [dbo].[Posts] (  
9     [PostId] INT IDENTITY (1, 1) NOT NULL,  
10    [Title] NVARCHAR (200) NULL,  
11    [Content] NTEXT NULL,  
12    [BlogId] INT NOT NULL,  
13    CONSTRAINT [PK_dbo.Posts] PRIMARY KEY CLUSTERED ([PostId] ASC),  
14    CONSTRAINT [FK_dbo.Posts_dbo.Blogs_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [dbo].[Blogs] ([BlogId])  
15    ON DELETE CASCADE  
16 )
```

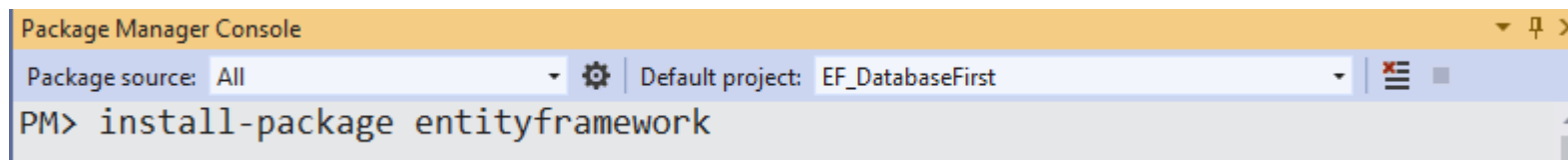


## Create an Application

Create a new (console) application and add the latest Entity Framework package to it:

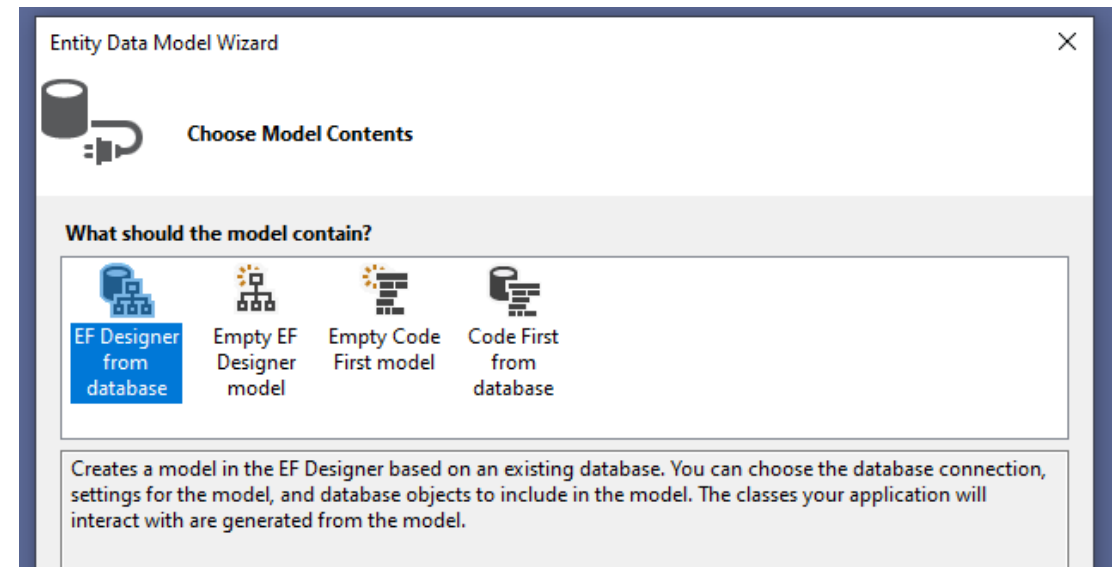


Or just use your package manager console:



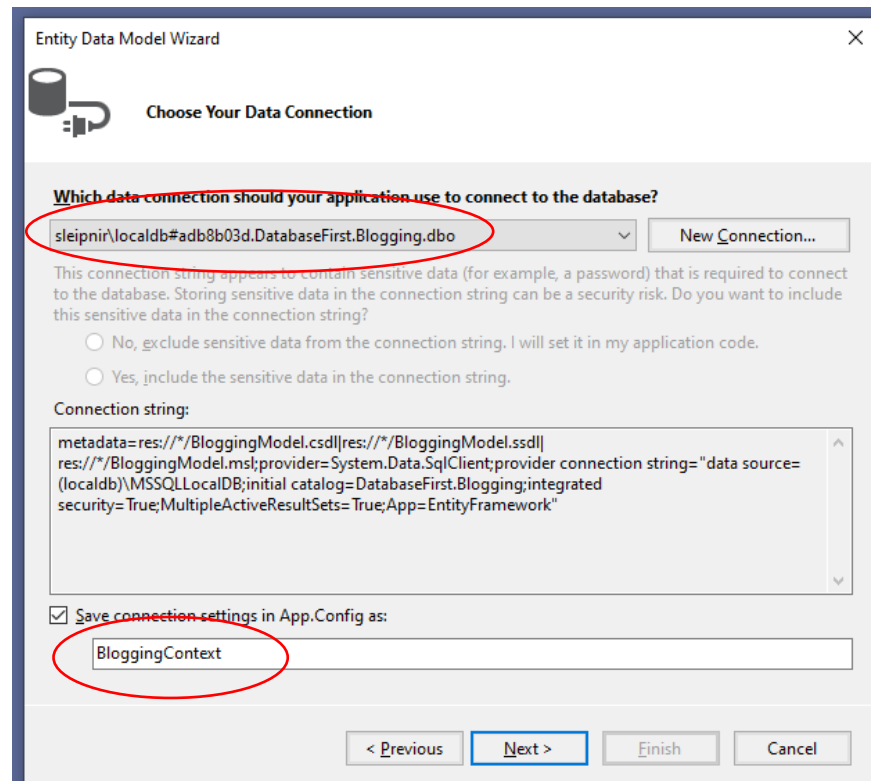
# Reverse Engineer Model

- Project -> Add New Item...
- Select Data from the left menu and then ADO.NET Entity Data Model
- Enter the model name and click OK
- This launches the Entity Data Model Wizard
- Pick “EF Designer from database” and hit Next



# Reverse Engineer Model

- Select the connection to the database you created in the first section, enter the name of the connection string and click Next



The image shows a screenshot of the 'Entity Data Model Wizard' dialog box, specifically the 'Choose Your Data Connection' step. The dialog has a title bar with 'Entity Data Model Wizard' and a close button. Below the title bar is a section with a database icon and the text 'Choose Your Data Connection'. The main area contains the question 'Which data connection should your application use to connect to the database?'. Below this is a dropdown menu with the selected value 'sleipnir\\localdb#adb8b03d.DatabaseFirst.Blogging.dbo', which is circled in red. To the right of the dropdown is a 'New Connection...' button. Below the dropdown is a warning message: '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?'. There are two radio buttons: 'No, exclude sensitive data from the connection string. I will set it in my application code.' and 'Yes, include the sensitive data in the connection string.'. Below this is a section labeled 'Connection string:' with a text area containing the connection string: 'metadata=res://\*/BloggingModel.csdl|res://\*/BloggingModel.ssdl|res://\*/BloggingModel.msl;provider=System.Data.SqlClient;provider connection string="data source=(localdb)\\MSSQLLocalDB;initial catalog=DatabaseFirst.Blogging;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework"'. At the bottom, there is a checkbox labeled 'Save connection settings in App.Config as:' which is checked. Below the checkbox is a text box containing 'BloggingContext', which is also circled in red. At the very bottom are four buttons: '< Previous', 'Next >', 'Finish', and 'Cancel'.

Entity Data Model Wizard

Choose Your Data Connection

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

sleipnir\\localdb#adb8b03d.DatabaseFirst.Blogging.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://\*/BloggingModel.csdl|res://\*/BloggingModel.ssdl|res://\*/BloggingModel.msl;provider=System.Data.SqlClient;provider connection string="data source=(localdb)\\MSSQLLocalDB;initial catalog=DatabaseFirst.Blogging;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework"

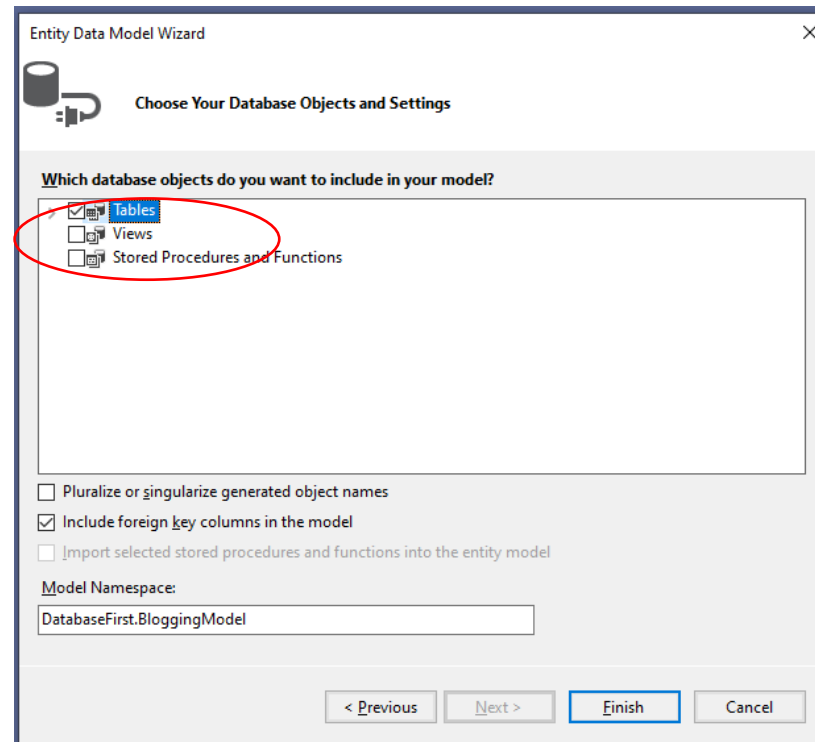
☒ Save connection settings in App.Config as:

BloggingContext

< Previous Next > Finish Cancel

# Reverse Engineer Model

- Pick whatever you may need from database and finish



The image shows a screenshot of the 'Entity Data Model Wizard' dialog box. The title bar reads 'Entity Data Model Wizard'. Below the title bar, there is a database icon and the text 'Choose Your Database Objects and Settings'. The main section is titled 'Which database objects do you want to include in your model?'. It contains a list of three items: 'Tables' (checked), 'Views' (unchecked), and 'Stored Procedures and Functions' (unchecked). The 'Tables' item is highlighted with a red circle. Below this list, there are three checkboxes: 'Pluralize or singularize generated object names' (unchecked), 'Include foreign key columns in the model' (checked), and 'Import selected stored procedures and functions into the entity model' (unchecked). At the bottom, there is a text field labeled 'Model Namespace:' containing the text 'DatabaseFirst.BloggingModel'. The bottom right corner has four buttons: '< Previous', 'Next >', 'Finish', and 'Cancel'.

Entity Data Model Wizard

Choose Your Database Objects and Settings

Which database objects do you want to include in your model?

- ☒ Tables
- ☐ Views
- ☐ Stored Procedures and Functions

☐ Pluralize or singularize generated object names

☒ Include foreign key columns in the model

☐ Import selected stored procedures and functions into the entity model

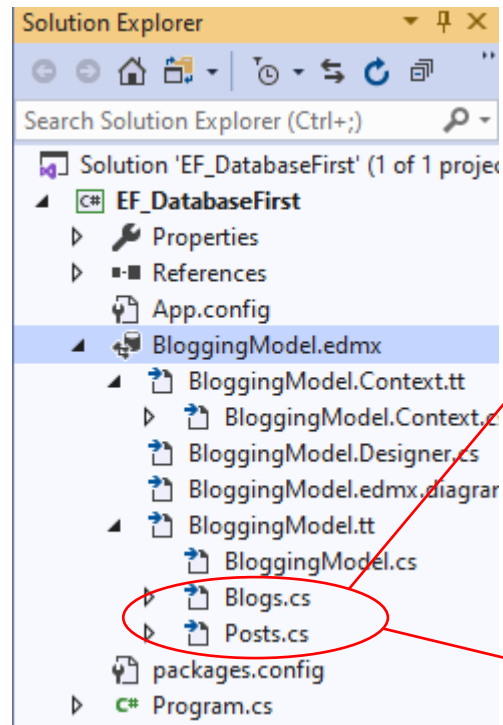
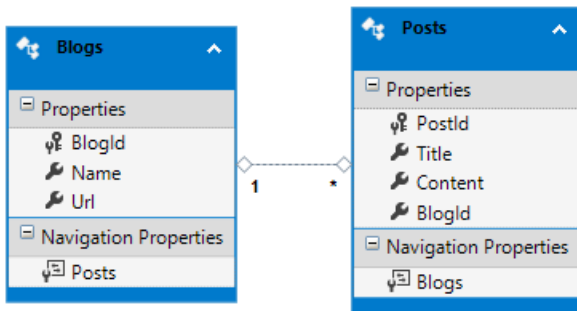
Model Namespace:

DatabaseFirst.BloggingModel

< Previous Next > Finish Cancel

# Reverse Engineer Model

- Explore your newly created model



```
public partial class Posts
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public int BlogId { get; set; }

    public virtual Blogs Blogs { get; set; }
}

public partial class Blogs
{
    public Blogs()
    {
        this.Posts = new HashSet<Posts>();
    }

    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }

    public virtual ICollection<Posts> Posts { get; set; }
}
```

## Read and Write your Data

```
static void Main(string[] args)
{
    using (var db = new BloggingContext())
    {
        // Create and save a new Blog
        var blog = new Blogs { Name = "ADO.NET blog" };
        db.Blogs.Add(blog);
        db.SaveChanges();

        // Display all Blogs from the database
        var query = from b in db.Blogs
                    orderby b.Name
                    select b;

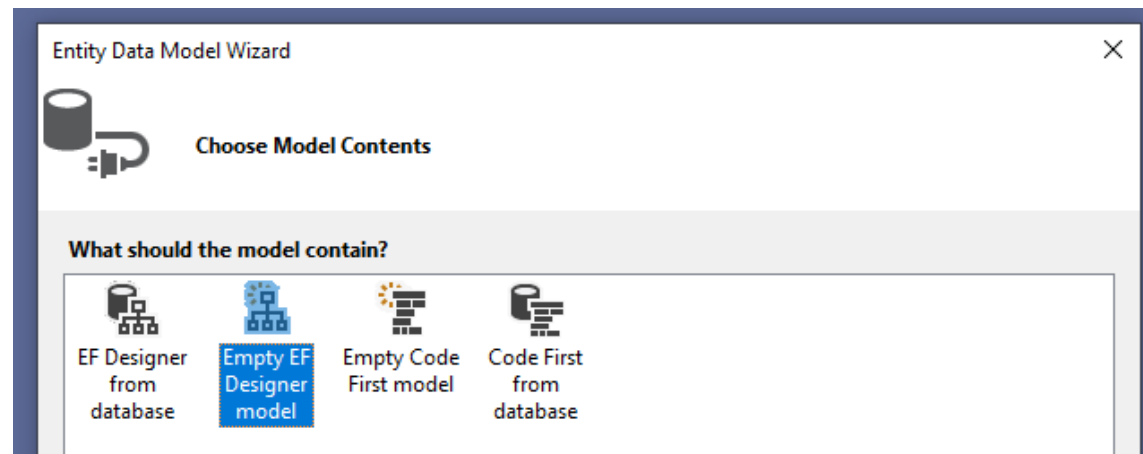
        foreach (var item in query)
        {
            Console.WriteLine(item.Name);
        }
    }
}
```

# Model First

## Create a Model

Assuming the application is created, create a model with Entity Framework Designer:

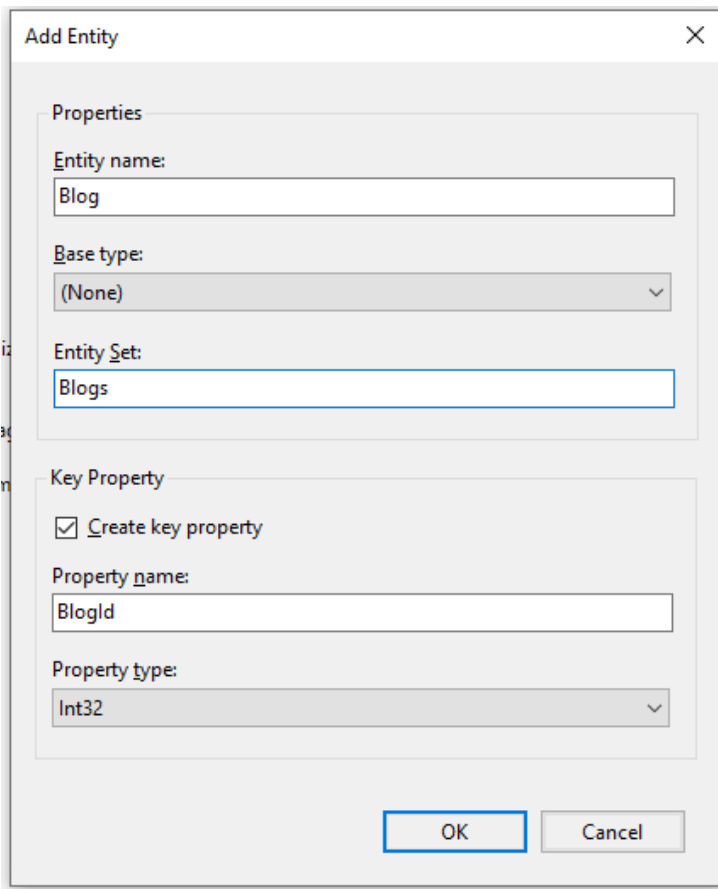
- Project -> Add New Item...
- Select Data from the left menu and then ADO.NET Entity Data Model
- Enter the name and click OK, this launches the Entity Data Model Wizard
- Select Empty Model and click Finish





# Create a Model

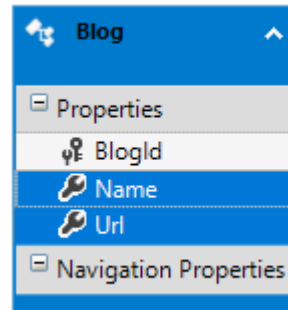
Create Entities. Right click on a surface -> Add New -> Entity



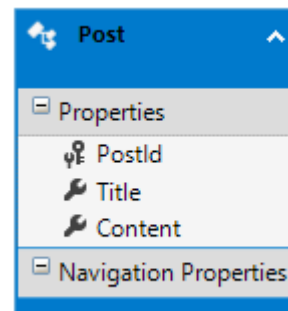
The 'Add Entity' dialog box is shown with the following fields and options:

- Properties**
  - Entity name: Blog
  - Base type: (None)
  - Entity Set: Blogs
- Key Property**
  - ☒ Create key property
  - Property name: BlogId
  - Property type: Int32

Buttons: OK, Cancel



Add some properties



# Create a Model

Create an association.

Add Association

Association Name:  
BlogPost

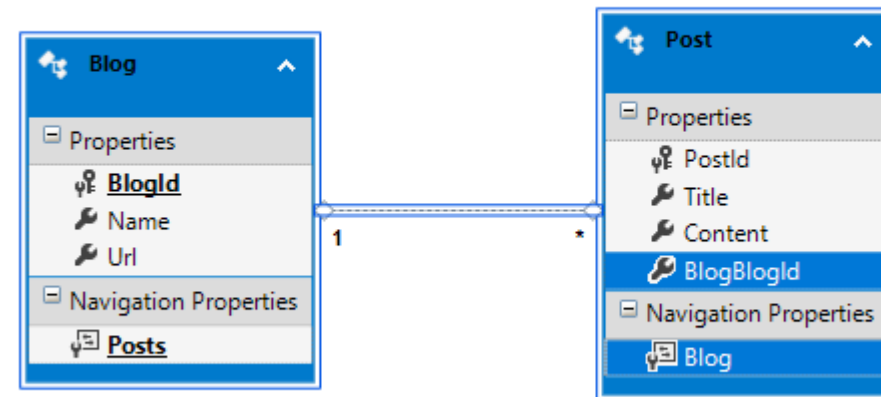
End Entity: Blog	End Entity: Post
Multiplicity: 1 (One)	Multiplicity: * (Many)
<input checked="" type="checkbox"/> Navigation Property: Post	<input checked="" type="checkbox"/> Navigation Property: Blog

☒ Add foreign key properties to the 'Post' Entity

Blog can have \* (Many) instances of Post. Use Blog.Post to access the Post instances.

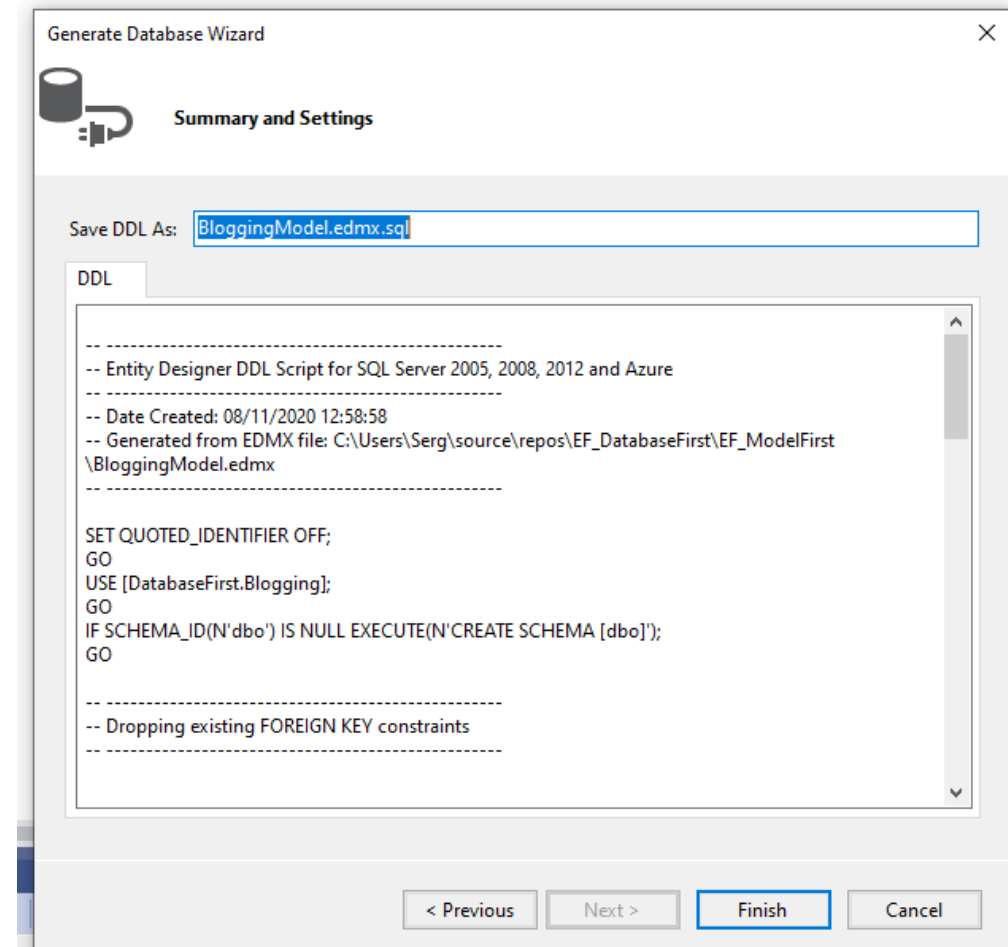
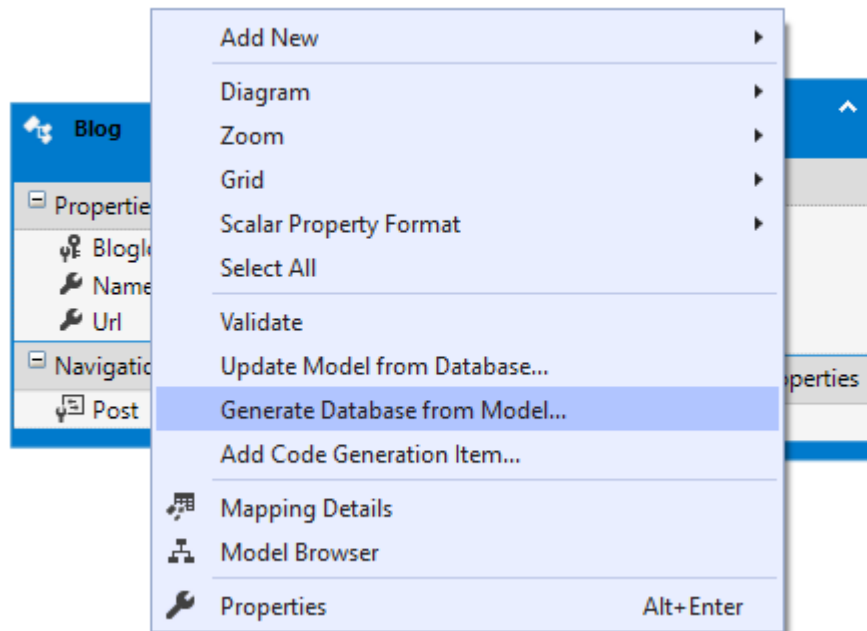
Post can have 1 (One) instance of Blog. Use Post.Blog to access the Blog instance.

OK Cancel



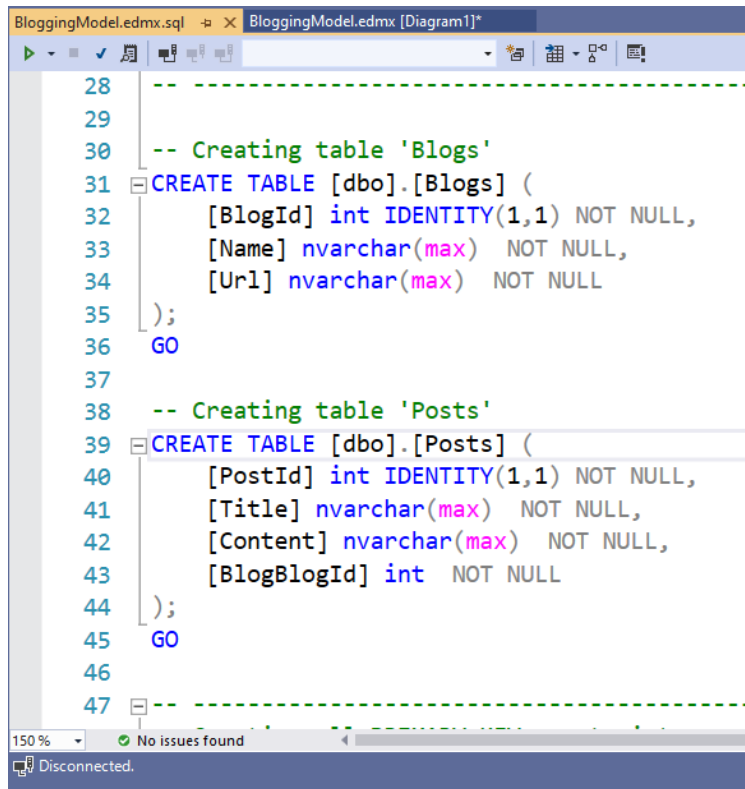
# Create Database from Model

Create an association.

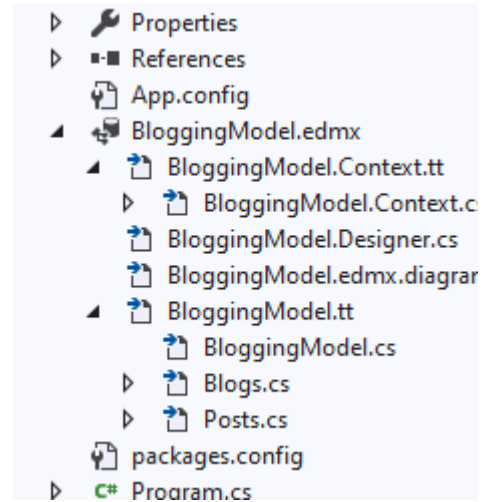


# Create Database from Model

Run your database creation script, full model description will be generated aswell:



```
28 -----
29
30 -- Creating table 'Blogs'
31 CREATE TABLE [dbo].[Blogs] (
32     [BlogId] int IDENTITY(1,1) NOT NULL,
33     [Name] nvarchar(max) NOT NULL,
34     [Url] nvarchar(max) NOT NULL
35 );
36 GO
37
38 -- Creating table 'Posts'
39 CREATE TABLE [dbo].[Posts] (
40     [PostId] int IDENTITY(1,1) NOT NULL,
41     [Title] nvarchar(max) NOT NULL,
42     [Content] nvarchar(max) NOT NULL,
43     [BlogBlogId] int NOT NULL
44 );
45 GO
46
47 -----
```



# Code First

## Create the Model

Create a simple model:

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }

    public virtual List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public virtual Blog Blog { get; set; }
}
```

## Create a Context

Install EF package and create your very own Context class:

```
using System.Data.Entity;

namespace EF_CodeFirst
{
    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }
    }
}
```

## Read and Write your Data

```
static void Main(string[] args)
{
    using (var db = new BloggingContext())
    {
        // Create and save a new Blog
        var blog = new Blogs { Name = "ADO.NET blog" };
        db.Blogs.Add(blog);
        db.SaveChanges();

        // Display all Blogs from the database
        var query = from b in db.Blogs
                    orderby b.Name
                    select b;

        foreach (var item in query)
        {
            Console.WriteLine(item.Name);
        }
    }
}
```



# Configuration

## Code-based configuration

Code-based configuration in EF6 and above is achieved by creating a subclass of **System.Data.Entity.Config.DbConfiguration**. The following guidelines should be followed when subclassing **DbConfiguration**:

- Create only one **DbConfiguration** class for your application
- Place your **DbConfiguration** class in the same assembly as your **DbContext** class
- Give your **DbConfiguration** class a public parameterless constructor
- Set configuration options by calling protected **DbConfiguration** methods from within this constructor

## Code-based configuration

A class derived from DbConfiguration might look like this:

```
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.SqlServer;

namespace EF_CodeFirst
{
    public class MyConfiguration : DbConfiguration
    {
        public MyConfiguration()
        {
            SetExecutionStrategy("System.Data.SqlClient", () => new SqlAzureExecutionStrategy());
            SetDefaultConnectionFactory(new LocalDbConnectionFactory("mssqllocaldb"));
        }
    }
}
```

## Configuration File Settings

The **entityFramework** section was automatically added to the configuration file of your project when you installed the EntityFramework NuGet package.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkId=294655 -->
    <section name="entityFramework"
      type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework,
      Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
      requirePermission="false"/>
  </configSections>
```

## Connection Strings

Connection strings go in the standard `connectionStrings` element and do not require the `entityFramework` section.

Code First based models use normal ADO.NET connection strings. For example:

```
<connectionStrings>
  <add name="BlogContext"
        providerName="System.Data.SqlClient"
        connectionString="Server=(localdb)\MSSQLLocalDB;Database=Blogging;Integrated Security=True;" />
</connectionStrings>
```

## Connection Strings

EF Designer based models use special EF connection strings. For example:

```
<connectionStrings>
  <add name="BloggngModelContainer"
    connectionString=
    "metadata=
      res://*/BloggngModel.csdl|
      res://*/BloggngModel.ssdl|
      res://*/BloggngModel.msl;
    provider=System.Data.SqlClient;
    provider connection string=
      &quot;data source=(localdb)\MSSQLLocalDB;
      initial catalog=ModelFirst.Bloggng;
      integrated security=True;
      MultipleActiveResultSets=True;
      App=EntityFramework&quot;;"
    providerName="System.Data.EntityClient" />
</connectionStrings>
```

## Connection Strings and Models

Typically an Entity Framework application uses a class derived from **DbContext**. This derived class will call one of the constructors on the base **DbContext** class to control:

- How the context will connect to a database — that is, how a connection string is found/used
- Whether the context will use calculate a model using Code First or load a model created with the EF Designer
- Additional advanced options

## Connection Strings by convention

If you have not done any other configuration in your application, then calling the parameterless constructor on **DbContext** will cause **DbContext** to run in **Code First mode** with a database connection created by convention. For example:

```
namespace Demo.EF
{
    public class BloggingContext : DbContext
    {
        public BloggingContext()
        // C# will call base class parameterless constructor by default
        {
        }
    }
}
```

DbContext will use full name of derived context class as db name - **Demo.EF.BloggingContext**



## Connection Strings by convention

If you have not done any other configuration in your application, then calling the string constructor on **DbContext** with the database name you want to use will cause **DbContext** to run in **Code First mode** with a database connection created by convention to the database of that name. For example:

```
namespace Demo.EF
{
    public class BloggingContext : DbContext
    {
        public BloggingContext()
            : base("BloggingDatabase")
        {
        }
    }
}
```

## Connection Strings and \*.config

You may choose to put a connection string in your `app.config` or `web.config` file. For example:

```
<configuration>
  <connectionStrings>
    <add name="BloggingCompactDatabase"
         providerName="System.Data.SqlServerCe.4.0"
         connectionString="Data Source=Blogging.sdf"/>
  </connectionStrings>
</configuration>
```

You can specify connections string name in context constructor, this works either way:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
        : base("BloggingCompactDatabase")
    {
    }
}
```

```
public class BloggingContext : DbContext
{
    public BloggingContext()
        : base("name=BloggingCompactDatabase")
    {
    }
}
```

# Code-First Initialization Strategies

# Initialization Strategies

The DbContext class supports four initialization strategies:

- CreateDatabaseIfNotExists
- DropCreateDatabaseIfModelChanges
- DropCreateDatabaseAlways
- Custom initializer

## Initialization Strategies

Initial strategy is set in derived DbContext class constructor:

```
namespace EF_CodeFirst
{
    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }

        public BloggingContext() : base("name=BlogContext")
        {
            Database.SetInitializer<BloggingContext>(
                new CreateDatabaseIfNotExists<BloggingContext>());
        }
    }
}
```

## Initialization Strategies

Initial strategy is set in derived DbContext class constructor:

```
namespace EF_CodeFirst
{
    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }

        public BloggingContext() : base("name=BlogContext")
        {
            Database.SetInitializer<BloggingContext>(
                new CreateDatabaseIfNotExists<BloggingContext>());
        }
    }
}
```

## Custom Initializers

To stuff data into the database during the initialization process, it's required that you create a custom database initializer:

```
public BloggingContext() : base("name=BlogContext")
{
    Database.SetInitializer(new MyBloggingContextInitializer());
}

public class MyBloggingContextInitializer : DropCreateDatabaseAlways<BloggingContext>
{
    protected override void Seed(BloggingContext context)
    {
        base.Seed(context);
        //populate your tables with initial data
    }
}
```

## Disable initialization

You can disable initializers entirely:

```
public class BloggingContext : DbContext
{
    public BloggingContext() : base("name=BlogContext")
    {
        Database.SetInitializer<BloggingContext>(null);
    }

    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
}
```



# Q&A



DRIVEN



CANDID



CREATIVE



ORIGINAL



INTELLIGENT



EXPERT

UA .NET Online LAB