

# ЛЕКЦИЯ

## Технология работы с данными Entity Framework

Лектор Крамар Ю.М.

# Содержание

1. Шаблоны проектирования доступа к данным
2. Концепция ORM
3. ADO.NET Entity Framework
4. Шаблоны Репозиторий и Единица работы  
(Repository, Unit of Work – UoW)

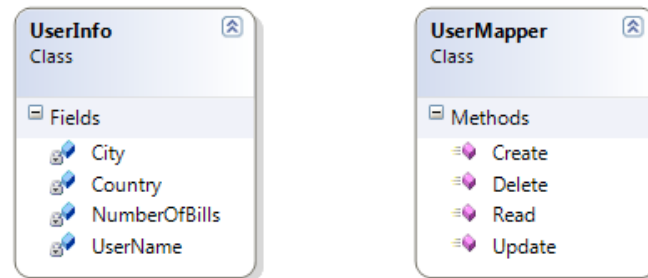
# Шаблоны проектирования доступа к данным

В фокусе – **уровень доступа к данным** – шаблоны проектирования и технологии реализации доступа к данным

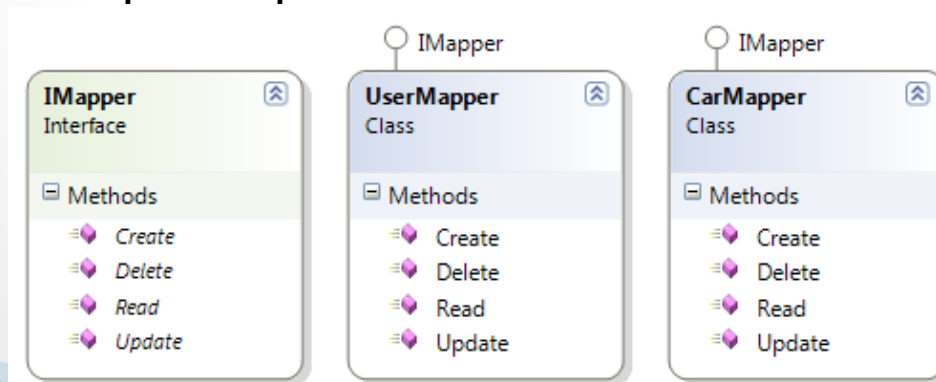


# Шаблоны проектирования доступа к данным

**Data Mapper** – позволяет инкапсулировать логику преобразования данных из объекта в формат данных источника. Таким образом при изменении источника или класса домена, изменения вносятся только в преобразователь.

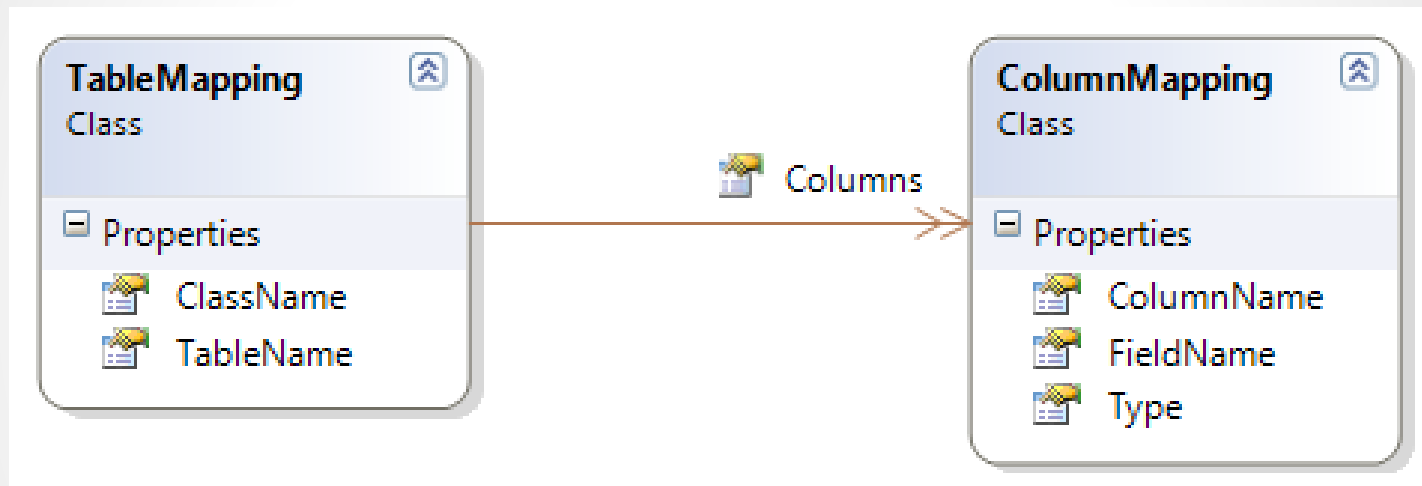


Можно удачно расширить этот шаблон:



# Шаблоны проектирования доступа к данным

**Metadata mapping** – позволяет максимально убрать дублирование кода доступа к базе:



# КОНЦЕПЦИЯ ORM

# Концепция ORM

**ORM** (Object–relational mapping – Объектно–реляционное отображение) означает технологию программирования, которая связывает базы данных с концепциями объектно–ориентированных языков программирования, т.е. ORM — прослойка между базой данных и кодом программиста, которая позволяет созданные в программе объекты помещать/считывать в/из БД:

- ▶ В ООП объекты в программе представляют объекты из реального мира. В качестве примера можно рассмотреть адресную книгу, которая содержит список людей с нулём или более телефонов и нулём или более адресов. В терминах объектно–ориентированного программирования они будут представляться объектами класса «Человек», которые будут содержать следующий список полей: имя, список (или массив) телефонов и список адресов.
- ▶ Суть задачи состоит в преобразовании таких объектов в форму, в которой они могут быть сохранены в файлах или базах данных, и которые легко могут быть извлечены в последующем, с сохранением свойств объектов и отношений между ними.

# Концепция ORM

Что бы понять, что такое **ORM**, рассмотрим простой пример:  
Классический подход (без ORM):

Код жестко привязан к источнику данных, программисту нужно хорошо знать **DDL**.



В нашем случае Data Definition Language – это SQL



# Концепция ORM

С ORM программист пишет код обращения к базе на используемом языке программирования. ORM преобразует этот код в соответствующий DDL и выполняет обращение к источнику:



- ▶ Разработано множество пакетов, устраняющих необходимость в преобразовании объектов для хранения в реляционных базах данных.
- ▶ Некоторые пакеты решают эту проблему, предоставляя библиотеки классов, способных выполнять такие преобразования автоматически. Имея список таблиц в базе данных и объектов в программе, они автоматически преобразуют запросы из одного вида в другой. В результате запроса объекта «человек» (из примера с адресной книгой) необходимый SQL-запрос будет сформирован и выполнен, а результаты «волшебным» образом преобразованы в объекты «номер телефона» внутри программы.

# Концепция ORM

- ▶ С точки зрения программиста система должна выглядеть как постоянное хранилище объектов. Он может просто создавать объекты и работать с ними как обычно, а они автоматически будут сохраняться в реляционной базе данных.
- ▶ На практике всё не так просто и очевидно. Все системы ORM обычно проявляют себя в том или ином виде, уменьшая в некотором роде возможность игнорирования базы данных. Более того, слой транзакций может быть медленным и неэффективным (особенно в терминах сгенерированного SQL). Все это может привести к тому, что программы будут работать медленнее и использовать больше памяти, чем программы, написанные «вручную».
- ▶ Но ORM избавляет программиста от написания большого количества кода, часто однообразного и подверженного ошибкам, тем самым значительно повышая скорость разработки. Кроме того, большинство современных реализаций ORM позволяют программисту при необходимости самому жёстко задать код SQL-запросов, который будет использоваться при тех или иных действиях (сохранение в базу данных, загрузка, поиск и т. д.) с постоянным объектом.

# Концепция ORM

Фактически **ORM** – набор шаблонов проектирования, соединенных вместе:

- ▶ Metadata mapping
- ▶ Data mapper
- ▶ Query object
- ▶ Lazy load
- ▶ Unit of work и Object collection
- ▶ И многое другое...

Реализовать свой ORM достаточно тяжело, но возможно, однако уже существует множество готовых решений:

- ▶ Linq to SQL
- ▶ ADO.NET Entity Framework
- ▶ NHibernate
- ▶ И другие

# ENTITY FRAMEWORK

# Entity Framework

Платформа Entity Framework представляет собой набор технологий ADO.NET, обеспечивающих разработку приложений, связанных с обработкой данных.

Entity Framework позволяет работать с данными в форме специфических для домена объектов и свойств, таких как клиенты и их адреса, без необходимости обращаться к базовым таблицам и столбцам базы данных, где хранятся эти данные. Это дает разработчикам возможность работать с данными на более высоком уровне абстракции, создавать и сопровождать приложения, ориентированные на данные, используя меньше кода, чем в традиционных приложениях.

# Entity Framework

- ▶ Entity Framework (EF) — это программная модель, которая представляет собой отражение конструкций базы данных на объектно-ориентированные конструкции.
- ▶ Используя EF, можно взаимодействовать с реляционными базами данных, не имея дело с кодом SQL (при желании).
- ▶ Исполняющая среда EF генерирует операторы SQL, соответствующие запросам LINQ к строго типизированным классам (сущностям, Entities).

# Entity Framework

- ▶ **Сущности (Entities)** – это концептуальная модель физической базы данных, которая отображается на предметную область. Эта модель называется *моделью сущностных данных (Entity Data Model — EDM)*. Не смотря на то, что сущности клиентской стороны в конечном итоге отображаются на таблицу базы данных, жесткая связь между именами свойств сущностных классов и именами столбцов таблиц с данными отсутствует.
- ▶ Модель EDM представляет собой набор классов клиентской стороны, которые отображаются на физическую базу данных.



# Entity Framework

Точкой входа для работы с моделью сущностных данных является класс `DbContext`. Его основное назначение заключается в том, что он:

- ▶ Отслеживает сущностные объекты, которые уже извлечены. Если объект запрашивается снова, он берется из контекста объектов.
- ▶ Хранит информацию состояния сущностей. Вы можете получить информацию о добавленных, модифицированных и удаленных объектах.
- ▶ Позволяет обновлять объекты из контекста объектов для записи изменений в лежащее в основе хранилище.



# Entity Framework

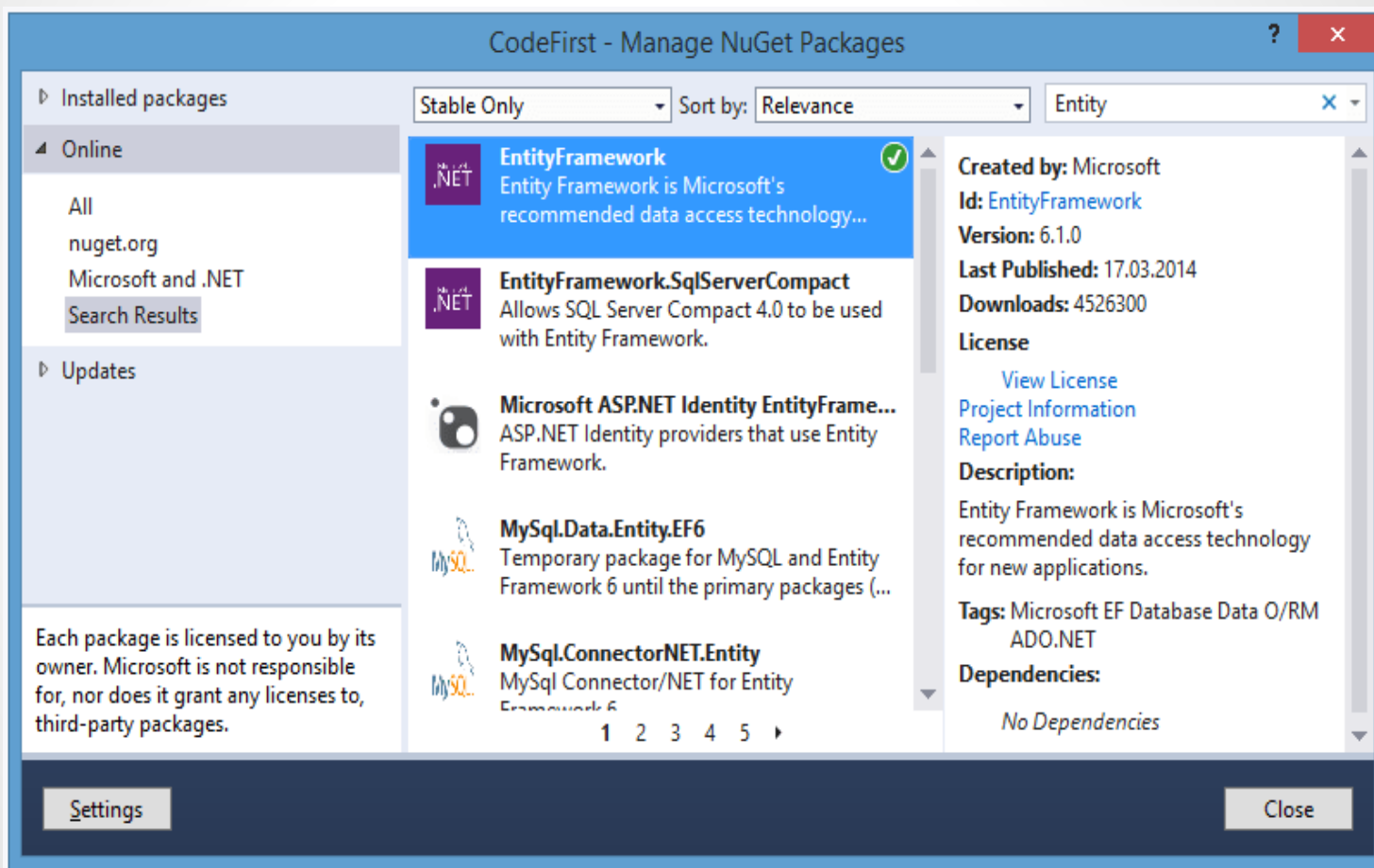
Способы работы с EF:

- ▶ **Data first** – генерация сущностей на основании структуры данных в источнике;
- ▶ **Model first** – создание сущностей при помощи дизайнера, а затем автогенерация хранилища;
- ▶ **Code first** – написание кода, а затем автогенерация сущностей и хранилища.

Подход **code first** позволяет писать реальный код, а затем вносить изменения в сгенерированную модель и структуру хранилища по ходу внесения изменений:

# Entity Framework

Для работы с EF необходимо подключить библиотеку:



# Entity Framework

Подход **code first** позволяет писать реальный код, а затем вносить изменения в сгенерированную модель и структуру хранилища по ходу внесения изменений:

Имеем классы:

**Author** с полями:

```
public int AuthorId { get; set; }
public string FirstName { get; set; }
public string LastName { get; set; }
public DateTime Registration { get; set; }
public string Email { get; set; }
public string AvatarUrl { get; set; }
public AuthenticationInfo AuthenticationInfo { get; set; }
```

**Blog** с полями:

```
public int BlogId { get; set; }
public string Title { get; set; }
public string Discription { get; set; }
public Author Author { get; set; } // ссылается на автора
public DateTime Created { get; set; }
public DateTime LastEntry { get; set; }
```

# Entity Framework

Теперь необходимо сгенерировать модель и хранилище:  
Для работы с моделью нужно реализовать класс:

```
public class PostITEntities : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Author> Authors { get; set; }
}
```

Этот класс наследует DbContext – контекстный объект EF.  
Так же надо создать инициализатор для EF:

```
public class DBInitializer :
DropCreateDatabaseIfModelChanges<PostITEntities>
{ }
```

Тип наследуемого класса сообщает о необходимых действиях при изменении модели.

# Entity Framework

## Проинициализировать EF:

```
protected void Application_Start()  
{  
    System.Data.Entity.Database.SetInitializer(new  
        DBInitializer());  
}
```

## И использовать EF в коде:

```
private PostITEntities dc = new PostITEntities();  
...  
var authors = dc.Authors;  
foreach(author a in authors)  
{  
    Console.WriteLine("{0}{1}", a.FirstName, a.LastName);  
}
```

# Entity Framework

Если не использовать инициализаторы, то необходимо описать конструктор класса контекста и соответствующую ему строку подключения к базе данных:

```
public class PostITEntities : DbContext
{
    public PostITEntities():base("DbConnection")
    { }
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Author> Authors { get; set; }
}
```

```
<connectionStrings>
    <add name="DBConnection" connectionString="data
source=(localdb)\v11.0;Initial Catalog=userstore.mdf; Integrated
Security=True;"
providerName="System.Data.SqlClient"/>
</connectionStrings>
```

# Entity Framework

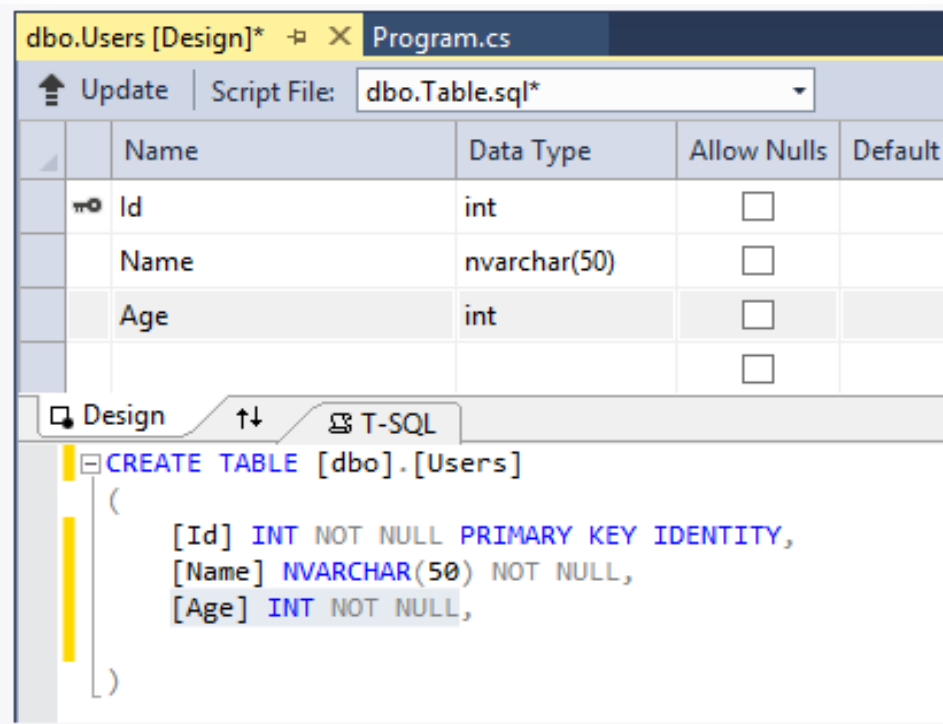
## Конфигурация EF:

```
<configSections>
  <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection
, EntityFramework, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" requirePermission="false" />
</configSections>
<startup>
  <supportedRuntime version="v4.0"
sku=".NETFramework,Version=v4.5" />
</startup>
<entityFramework>
  <defaultConnectionFactory
type="System.Data.Entity.Infrastructure.SqlConnectionFactory,
EntityFramework" />
  <providers>
    <provider invariantName="System.Data.SqlClient"
type="System.Data.Entity.SqlServer.SqlProviderServices,
EntityFramework.SqlServer" />
  </providers>
</entityFramework>
```

# Entity Framework

Подход **code first** применяется и к уже существующей базе данных.

Например имеется база данных с таблицей Users:



The screenshot displays the SQL Server Enterprise Designer interface for the 'dbo.Users' table. The top pane shows the 'Design' view with columns: Id (int, primary key, not null), Name (nvarchar(50), not null), and Age (int, not null). The bottom pane shows the 'T-SQL' view with the following script:

```
CREATE TABLE [dbo].[Users]
(
    [Id] INT NOT NULL PRIMARY KEY IDENTITY,
    [Name] NVARCHAR(50) NOT NULL,
    [Age] INT NOT NULL,
)
```

	Name	Data Type	Allow Nulls	Default
Id		int	<input type="checkbox"/>	
Name		nvarchar(50)	<input type="checkbox"/>	
Age		int	<input type="checkbox"/>	
			<input type="checkbox"/>	



# Entity Framework

Создадим конфигурацию для подключения к бд:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <!--остальное содержимое-->
  <connectionStrings>
    <add name="UserDB" connectionString="data
source=(localdb)\v11.0; AttachDbFilename=|DataDirectory|
\userstoredb.mdf;Integrated Security=True;"
    providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>
```

# Entity Framework

Определим классы модели данных User и контекста UserContext:

```
public class User
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

```
class UserContext : DbContext
{
    public UserContext():
        base("UserDB")
    { }

    public DbSet<User> Users { get; set; }
}
```

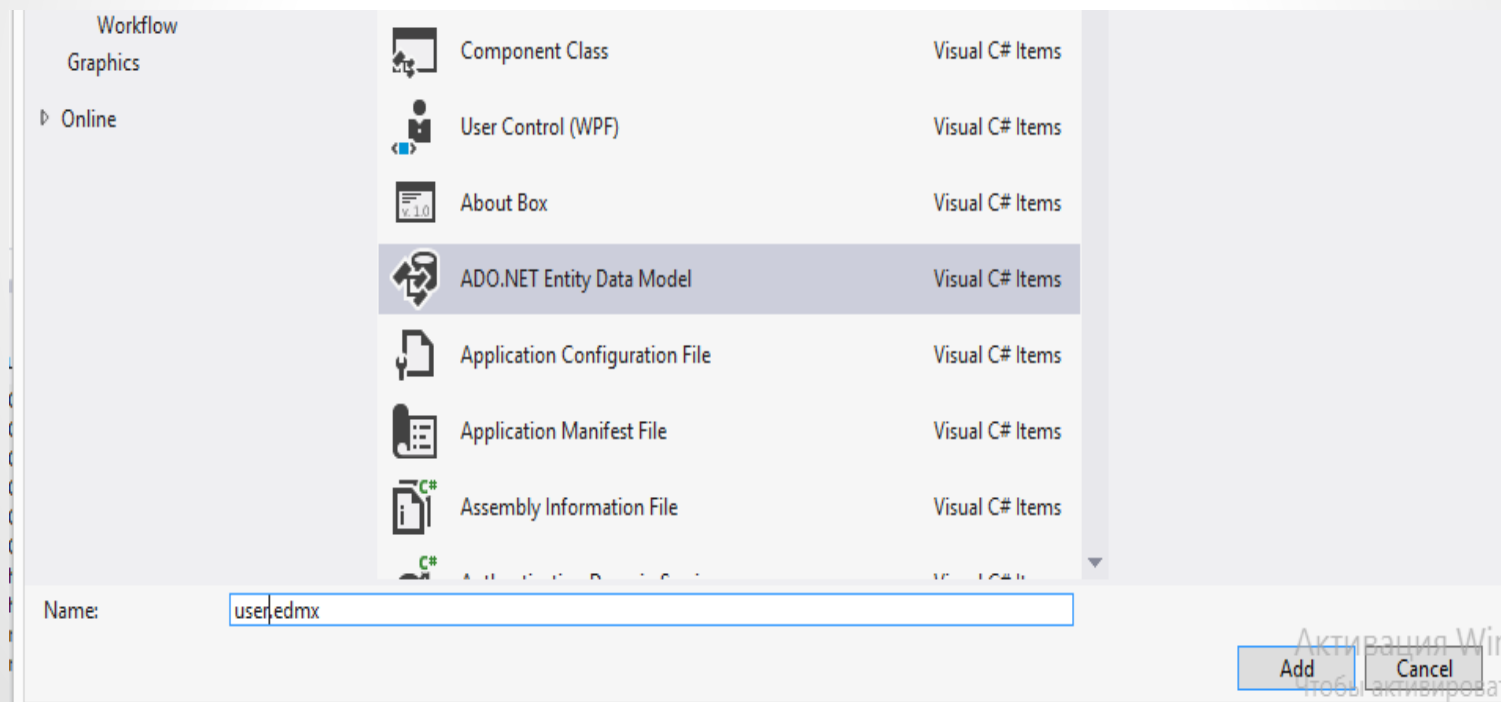
# Entity Framework

Для получения данных определим, например, следующий код консольного приложения:

```
using(UserContext db = new UserContext())
{
    var users = db.Users;
    foreach(User u in users)
    {
        Console.WriteLine("{0}.{1} - {2}", u.Id, u.Name, u.Age);
    }
}
```

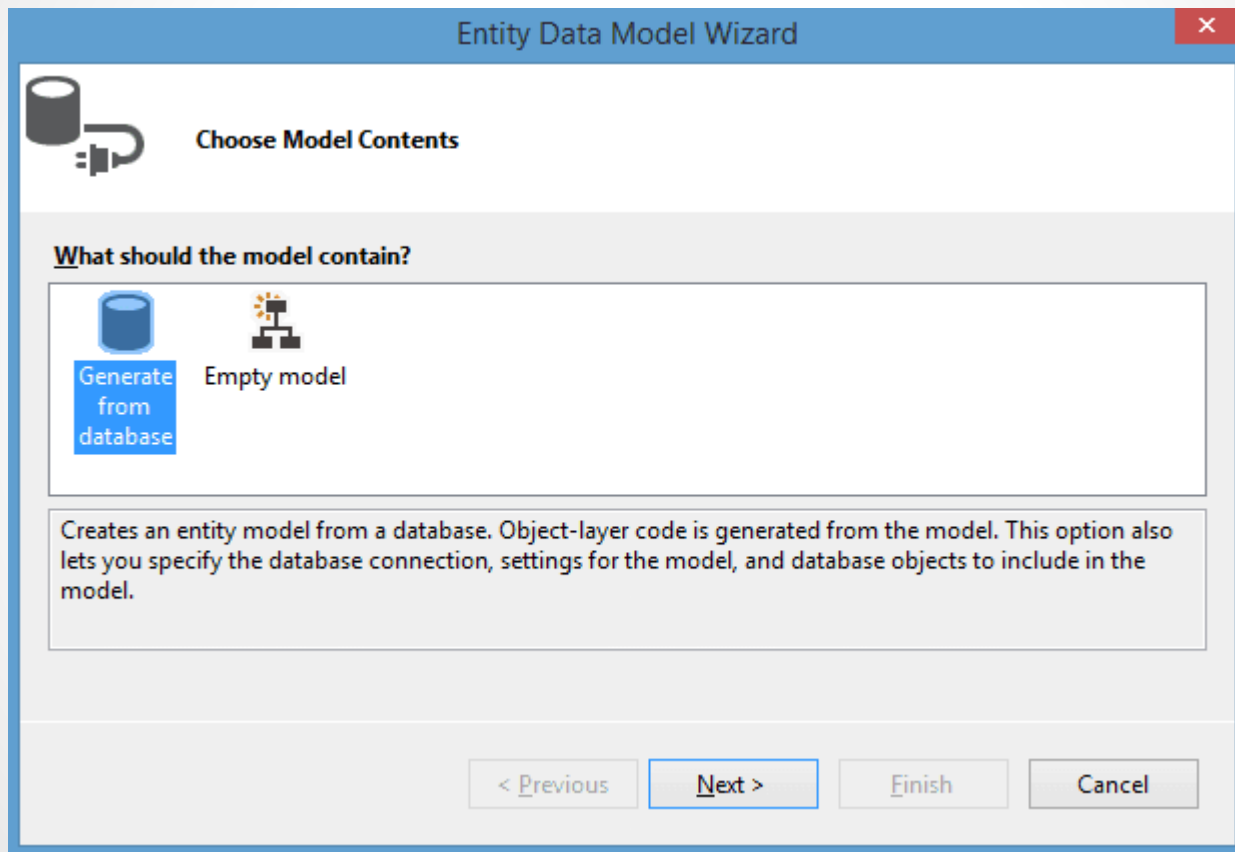
# Entity Framework

Подход **data first** подходит для тех случаев, когда разработчик уже имеет готовую базу данных, а в системе установлен соответствующий провайдер для работы с СУБД. Классы сущностей сгенерируются дизайнером Entity Data Model:



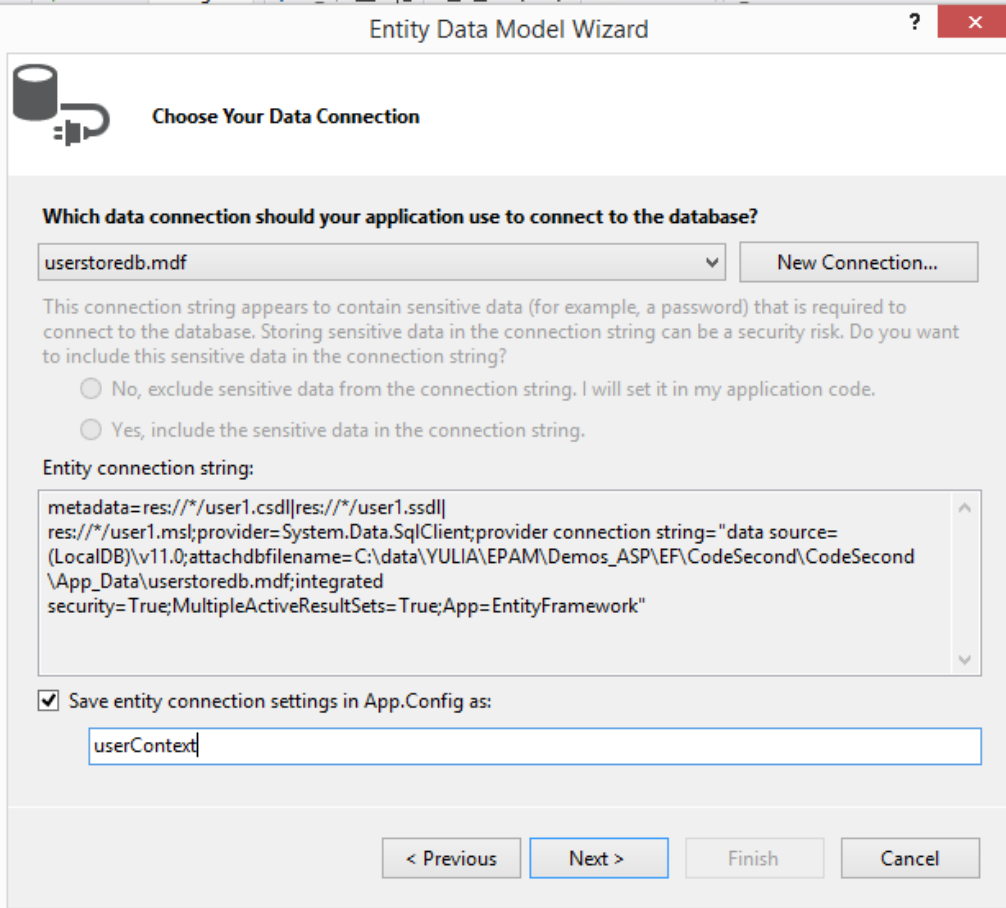
# Entity Framework

Генерируем модель из готовой бд:



# Entity Framework

Выбираем соединение с бд или создаем новое:



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 a question mark and a close button. Below the title bar is a header area with a database icon and the text 'Choose Your Data Connection'. The main content area asks 'Which data connection should your application use to connect to the database?'. There is a dropdown menu showing 'userstoredb.mdf' and a 'New Connection...' button. Below this, a warning message states: '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.' (selected) and 'Yes, include the sensitive data in the connection string.'. Below the radio buttons is a text box labeled 'Entity connection string:' containing a long connection string. At the bottom, there is a checkbox 'Save entity connection settings in App.Config as:' which is checked, followed by a text box containing 'userContext'. 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?

userstoredb.mdf 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.

Entity connection string:

```
metadata=res://*/user1.csdl|res://*/user1.ssdl|
res://*/user1.msl;provider=System.Data.SqlClient;provider connection string="data source=
(LocalDB)\v11.0;attachdbfilename=C:\data\YULIA\EPAM\Demos_ASP\EF\CodeSecond\CodeSecond
\App_Data\userstoredb.mdf;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
```

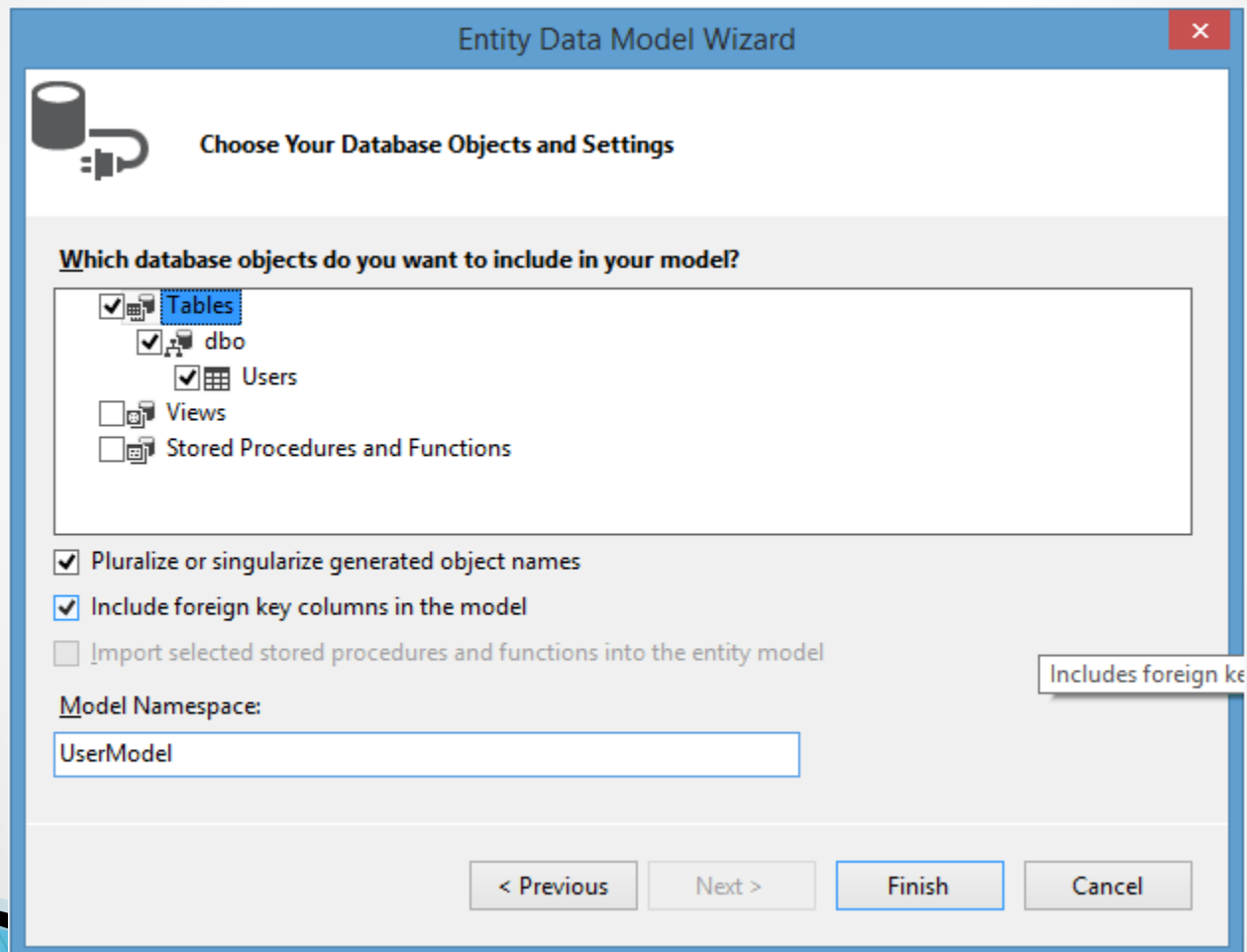
☒ Save entity connection settings in App.Config as:

userContext

< Previous Next > Finish Cancel

# Entity Framework

Определяем содержание модели (таблицы, хранимые процедуры:



The image shows the 'Entity Data Model Wizard' dialog box, titled 'Entity Data Model Wizard' with a close button (X) in the top right corner. The main heading is 'Choose Your Database Objects and Settings'. Below this, the question 'Which database objects do you want to include in your model?' is displayed. A list of database objects is shown with checkboxes: 'Tables' (checked), 'dbo' (checked), 'Users' (checked), 'Views' (unchecked), and 'Stored Procedures and Functions' (unchecked). Below the list, there are three checkboxes: 'Pluralize or singularize generated object names' (checked), 'Include foreign key columns in the model' (checked), and 'Import selected stored procedures and functions into the entity model' (unchecked). A tooltip 'Includes foreign ke' is visible next to the 'Include foreign key columns in the model' checkbox. Below these checkboxes, the 'Model Namespace:' is shown with a text box containing 'UserModel'. At the bottom, there are 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
  - ☒ dbo
  - ☒ Users
- ☐ 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:

UserModel

< Previous   Next >   Finish   Cancel

# Entity Framework

## Строка подключения EDM:

```
<connectionStrings>
  <add name="userContext" providerName="System.Data.EntityClient"
    connectionString="metadata=res://*/user.csd|res://*/user.ssd|r
es://*/user.msl;provider=System.Data.SqlClient;
    provider connection string="data source=HP-PC\SQLEXPRESS;
    initial catalog=persondb;integrated security=True;
    MultipleActiveResultSets=True;App=EntityFramework"" />
</connectionStrings>
```



# ADO.NET Entity Framework

Entity Framework имеет несколько уровней для отображения таблиц базы данных на объекты:

- ▶ логический — этот уровень определяет реляционные данные;
- ▶ концептуальный — этот уровень определяет классы .NET;
- ▶ отображения — этот уровень определяет отображение классов .NET на реляционные таблицы и ассоциации.
- ▶ В EF каждый из этих трех уровней фиксируется в XML-файле. В результате использования интегрированных визуальных конструкторов Entity Framework создается Entity Data Model-файл с расширением \*.edmx.

# Entity Framework

Логический уровень определен на языке SSDL (Store Schema Definition Language — язык определения схемы хранилища) и определяет структуру таблиц базы данных и их отношений.

```
<!-- SSDL content -->

<EntityType Name="Users">
  <Key>
    <PropertyRef Name="Id" />
  </Key>
  <Property Name="Id" Type="int" Nullable="false"
StoreGeneratedPattern="Identity" />
  <Property Name="Name" Type="nvarchar" Nullable="false"
MaxLength="50" />
  <Property Name="Age" Type="int" Nullable="false" />
</EntityType>
```

# Entity Framework

Концептуальный уровень определяет классы .NET. Этот уровень создается на языке CSDL (Conceptual Schema Definition Language — язык концептуального определения схемы):

```
<!-- CSDL content -->

<EntityType Name="User">
  <Key>
    <PropertyRef Name="Id" />
  </Key>
  <Property Name="Id" Type="Int32" Nullable="false"
p1:StoreGeneratedPattern="Identity" />
  <Property Name="Name" Type="String" Nullable="false"
MaxLength="50" Unicode="true" FixedLength="false" />
  <Property Name="Age" Type="Int32" Nullable="false" />
</EntityType>
```

# Entity Framework

Уровень отображения отображает определение типа сущности с CSDL на SSDL, используя язык MSL (Mapping Specification Language — язык спецификации отображения).

```
<!-- MSL content -->

<EntitySetMapping Name="Users">
  <EntityTypeMapping TypeName="userstoredbModel.User">
    <MappingFragment StoreEntitySet="Users">
      <ScalarProperty Name="Id" ColumnName="Id" />
      <ScalarProperty Name="Name" ColumnName="Name" />
      <ScalarProperty Name="Age" ColumnName="Age" />
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>
```

# Entity Framework

Папка `obj\Debug\edmxResourcesToEmbed` содержит три XML-файла EDM-модели, основанные на содержимом файла `*.edmx`:

- ▶ `user.csdl`,
- ▶ `user.msl`,
- ▶ `user.ssdl`.

Данные в этих файлах будут встроены в сборку как двоичные ресурсы.

Таким образом, приложение .NET обладает всей информацией, необходимой для понимания концептуального, физического и уровня отображения модели EDM.

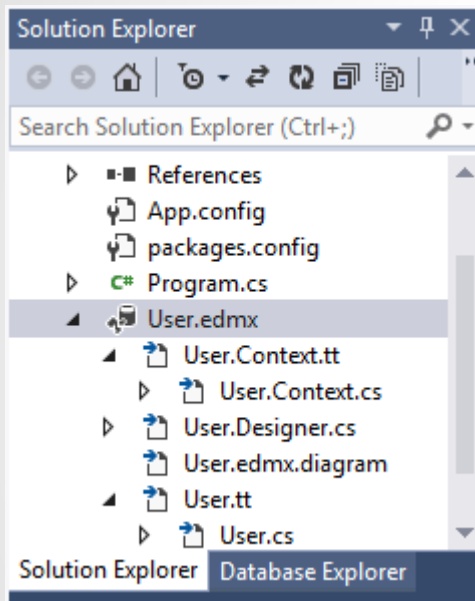
# Entity Framework

Просмотр содержимого модели User.edmx

и отображения (Mapping) :



Generated code

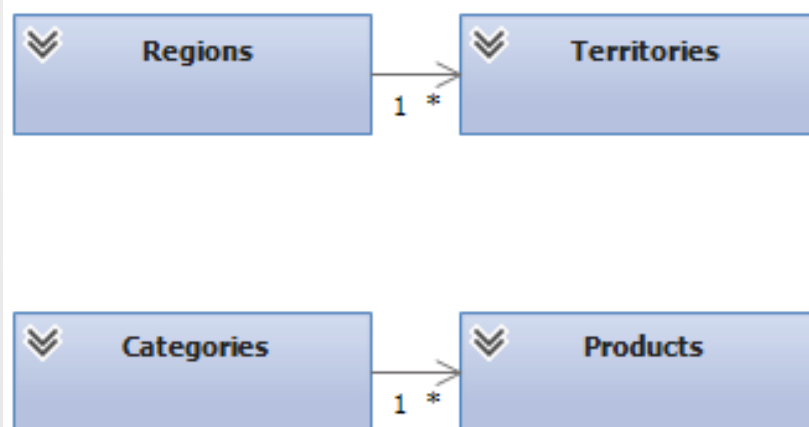


Mapping Details - User		
Column	Operator	Value / Property
<strong>Tables</strong>		
Maps to Users		
<Add a Condition>		
Column Mappings		
Id : int	↔	Id : Int32
Name : nvarchar	↔	Name : String
Age : int	↔	Age : Int32
<Add a Table or View>		

# Entity Framework



Создадим модель.edmx на основе готовой базы данных **NORTHWND.mdf**, в которую включим использование таблиц **Categories**, **Products**, **Regions** и **Territories**, а также хранимой процедуры **ProductCountInCategory**, принимающей строку и возвращающую целое:



StoredProc

```
CREATE PROCEDURE ProductCountInCategory  
  
    @CategoryName nvarchar(15),  
    @ProductCount int OUTPUT  
  
AS  
  
    SELECT @ProductCount =  
        count(ProductID) from Products  
        WHERE Products.CategoryID = (select  
            CategoryID from Categories where CategoryName  
            = @CategoryName)
```

# Entity Framework



## Строка подключения EDM:

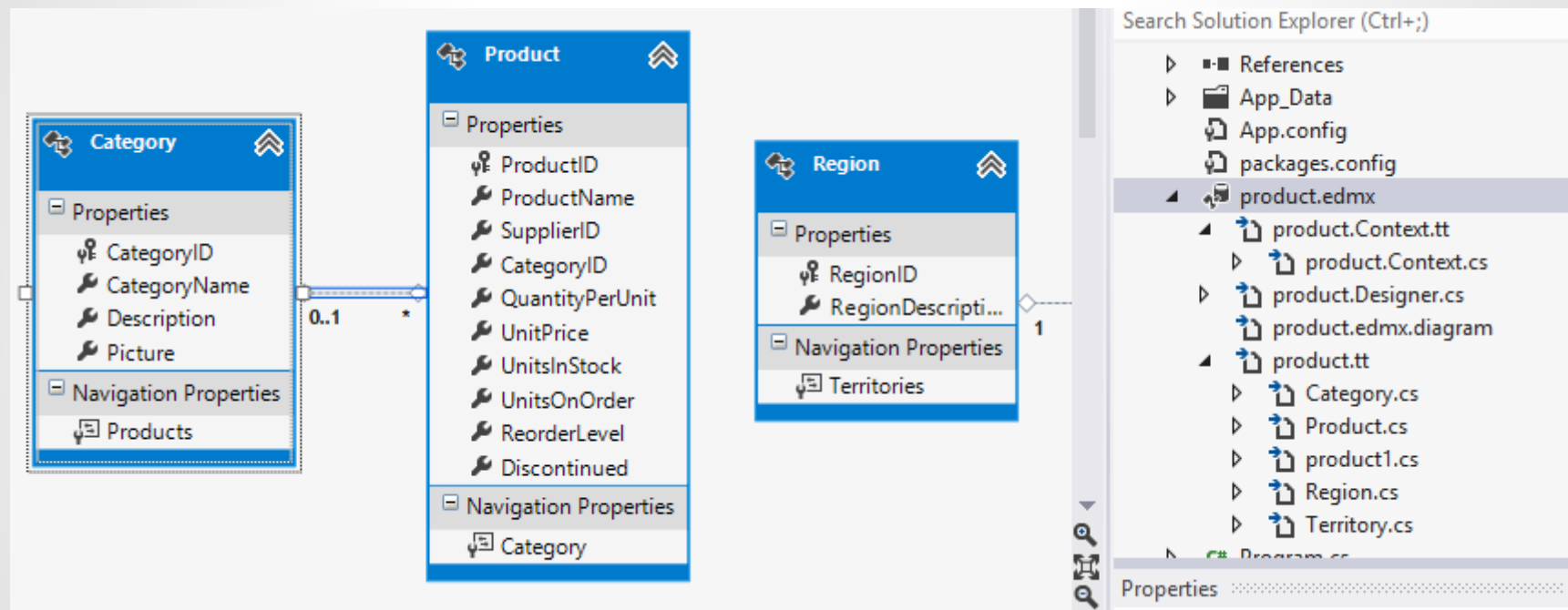
```
<connectionStrings>
  <add name="productContext"
connectionString="metadata=res://*/product.csdl|res://*/product.ssd
l|res://*/product.msl;provider=System.Data.SqlClient;provider
connection string=&quot;data
source=(LocalDB)\v11.0;attachdbfilename=C:\data\YULIA\EPAM\Demos_AS
P\EF\simpleEF\EFwork1\App_Data\NORTHWND.mdf;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework&quo
t;" providerName="System.Data.EntityClient" />
</connectionStrings>
```



# Entity Framework



Сгенерированная EDM:



# Entity Framework



## Содержимое класса контекста EDM:

```
public partial class productContext : DbContext
{
    public productContext()
        : base("name=productContext") {}

    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }
    public DbSet<Region> Regions { get; set; }
    public DbSet<Territory> Territories { get; set; }

    public virtual int ProductCountInCategory(string
categoryName, ObjectParameter productCount)
    {
        var categoryNameParameter = categoryName != null ?
            new ObjectParameter("CategoryName", categoryName) :
            new ObjectParameter("CategoryName",
typeof(string));
        return
            ((IObjectContextAdapter)this).ObjectContext.ExecuteFunction("Produc
tCountInCategory", categoryNameParameter, productCount);
    }
}
```



Generated code

# Entity Framework

Сегодня подход data first, включающий в себя создание модели .edmx используется не часто, и широкое распространение нашел подход code first как для создания бд через фреймворк EF так и использования уже существующей.

# Entity Framework

CRUD-операции в Entity Framework (контекст db содержит коллекцию сущностей Region):

```
public DbSet<Region> Regions { get; set; }
```

```
var region = new Region { RegionID = regionID, RegionDescription =  
regionDescription };  
db.Regions.Add(region);  
db.SaveChanges();
```

```
region.RegionDescription = regionDescription;  
db.Entry(region).State = System.Data.EntityState.Modified;  
db.SaveChanges();
```

```
Region region = db.Regions.FirstOrDefault(r => r.RegionID ==  
regionID);  
if (region != null)  
{  
    db.Regions.Remove(region);  
    db.SaveChanges();  
}
```

# Linq to Entities

Запросы LINQ to Entities должны быть преобразованы в SQL-запросы. Преобразование выполняется ORM-провайдером EntityClient, представляющим интерфейс для взаимодействия с провайдером ADO.NET для SQL Server.

Провайдер обеспечивает создание и использование служебных объектов EntityClient:

- EntityConnection – для взаимодействия с БД

- EntityCommand – для передачи запросов к БД

- EntityDataReader – для извлечения данных из БД

# Linq to Entities

При получении и обработке данных из БД с помощью технологии EF имеем дело с **Linq to Entities** и **Linq to Objects**.

Например, описаны связанные сущности EF Company и Phone:

```
public class Company
{
    public int Id { get; set; }
    ...
    public ICollection<Phone> Phones { get; set; }
    public Company()
    {
        Phones = new List<Phone>();
    }
}
public class Phone
{
    public int Id { get; set; }
    ...
    public int CompanyId { get; set; }
    public Company Company { get; set; }
}
```

# Linq to Entities

Для работы с сущностями описан контекст DbContext и выполнен Linq-запрос через контекст:

```
using (PhoneContext db = new PhoneContext())  
{  
    var phones = db.Phones.Where(p => p.CompanyId == 1);  
}
```

Linq-запрос превратится в SQL-запрос SELECT ... FROM ... WHERE  
Однако в запросе

```
var phones = db.Phones.Where(p => p.CompanyId ==  
1).ToList().Where(p => p.Id < 10);
```

Первый Where – транслируется в выражение SQL.

Второй Where – обращение к списку в памяти и выполнение запроса Linq to Object

# Linq to Entities

## Навигационные свойства и lazy loading

В реальности в базе данных может быть не одна, а несколько таблиц, которые связаны между собой различными связями.

```
class Team{  
    public int Id { get; set; }  
    public string Name { get; set; } // название команды  
    public string Coach { get; set; } // тренер  
}
```

```
class Player{  
    public int Id { get; set; }  
    public string Name { get; set; }  
    public string Position { get; set; }  
    public int Age { get; set; }  
  
    public int? TeamId { get; set; }  
    public Team Team { get; set; }  
}
```



# Link to Entities

Внешний ключ состоит из обычного свойства и навигационного.

Свойство `public Team Team { get; set; }` в классе `Player` называется **навигационным свойством** - при получении данных об игроке оно будет автоматически получать данные из БД.

Вторая часть внешнего ключа - свойство `TeamId`. Чтобы в связке с навигационным свойством образовать внешний ключ оно должно принимать одно из следующих вариантов имени:

- *Имя\_навигационного\_свойства + Имя ключа из связанной таблицы (TeamId)*
- *Имя\_класса\_связанной\_таблицы + Имя ключа из связанной таблицы (TeamId)*

Если тип обычного свойства во внешнем ключе – `int?`, то есть допускает значения `null`, то при создании бд соответствующее поле сможет принимать значения `NULL`: `[TeamId] INT NULL`, если `int`, то поле имело бы ограничение `NOT NULL`, а внешний ключ определял бы каскадное удаление.

# Linq to Entities

- ▶ Получение данных жадной загрузкой (eager loading). Ее суть заключается в том, чтобы использовать для подгрузки связанных по внешнему ключу данных метод **Include**

```
using(SoccerContext db = new SoccerContext())
{
    IEnumerable<Player> players = db.Players.Include(p=>p.Team);
    foreach(Player p in players)
    {
        MessageBox.Show(p.Team.Name);
    }
}
```

# Linq to Entities

- ▶ Получение данных ленивой загрузкой (lazy loading). При первом обращении к объекту, если связанные данные не нужны, то они не подгружаются. Однако при первом же обращении к навигационному свойству эти данные автоматически подгружаются из бд. Классы, использующие ленивую загрузку, должны быть публичными, а их свойства должны иметь модификаторы `public` и `virtual`.

```
class Team{  
    ...  
    public virtual ICollection<Player> Players { get; set; }  
  
    public Team()  
    {  
        Players = new List<Player>();  
    }  
}
```

```
class Player{  
    ...  
    public int? TeamId { get; set; }  
    public virtual Team Team { get; set; }  
}
```

# Linq to Entities

Связи один-к-одному между сущностями EF:

```
public class User
{
    public int Id { get; set; }
    public string Login { get; set; }
    public string Password { get; set; }

    public UserProfile Profile { get; set; }
}

public class UserProfile
{
    [Key]
    [ForeignKey("User")]
    public int Id { get; set; }

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

    public User User { get; set; }
}
```

# Linq to Entities

Связи один-ко-многим между сущностями EF:

```
public class Player
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Position { get; set; }
    public int Age { get; set; }

    public int? TeamId { get; set; }
    public Team Team { get; set; }
}
```

# Linq to Entities

## Связи один-ко-многим между сущностями EF:

```
public class Team
{
    public int Id { get; set; }
    public string Name { get; set; } // название команды

    public ICollection<Player> Players { get; set; }
    public Team()
    {
        Players = new List<Player>();
    }
}
```

# Linq to Entities

## Связи многие-ко-многим между сущностями EF:

```
public class Player
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }

    public ICollection<Team> Teams { get; set; }
    public Player()
    {
        Teams = new List<Team>
    }
}
```

# Linq to Entities

## Связи многие-ко-многим между сущностями EF:

```
public class Team
{
    public int Id { get; set; }
    public string Name { get; set; } // название команды

    public ICollection<Player> Players { get; set; }
    public Team()
    {
        Players = new List<Player>();
    }
}
```



# Linq to Entities

Так как для сущностей характерно наличие поля **Id**, то к методам Linq **First()/FirstOrDefault()**, в **DbSet** добавлен метод **Find()**:

```
Phone myphone = db.Phones.Find(3); // получение элемента с id=3
```

Это аналог:

```
Phone myphone = db.Phones.FirstOrDefault(p=>p.Id==3);  
if (myphone != null)  
    Console.WriteLine(myphone.Name);
```

Также работают операции сортировки **OrderBy**, **ThenBy**, объединения двух и более наборов **Join**, группировки **GroupBy**, операции с множествами: объединение **Union**, пересечение **Intersect**, разность **Except**, агрегатные операции.

# Linq to Entities

Когда контекст данных извлекает данные из базы данных, Entity Framework помещает извлеченные объекты в кэш и отслеживает изменения, которые происходят с этими объектами вплоть до использования метода `SaveChanges()`, который фиксирует все изменения в базе данных. Но не всегда необходимо отслеживать изменения. Например, нам надо просто вывести данные для просмотра.

Чтобы данные не помещались в кэш, применяется метод `AsNoTracking()`. При его применении возвращаемые из запроса данные не кэшируются. А это означает, что Entity Framework не производит какую-то дополнительную обработку и не выделяет дополнительное место для хранения извлеченных из БД объектов.

Метод `AsNoTracking()` применяется к набору IQueryable операции.

```
IEnumerable<Book> books2 = db.Books
    .Where(b => b.Price > 200)
    .AsNoTracking().ToList();
```

# Linq to Entities

Методы расширений LINQ могут возвращать два объекта: **IEnumerable** и **IQueryable**

**IEnumerable** – System.Collections. Объект IEnumerable представляет набор данных в памяти и может перемещаться по этим данным только вперед. Запрос к IEnumerable выполняется немедленно и полностью, поэтому получение данных приложением происходит быстро. При выполнении запроса IEnumerable загружает все данные, и если нам надо выполнить их фильтрацию, то сама фильтрация происходит на стороне клиента.

**IQueryable** – System.Linq. Объект IQueryable предоставляет удаленный доступ к базе данных и позволяет перемещаться по данным как в прямом порядке от начала до конца, так и в обратном порядке. В процессе создания запроса, возвращаемым объектом которого является IQueryable, происходит оптимизация запроса. В итоге в процессе его выполнения тратится меньше памяти, меньше пропускной способности сети, но в то же время он может обрабатываться чуть медленнее, чем запрос, возвращающий объект IEnumerable.

# Linq to Entities

С IEnumerable фильтрация результата, обозначенная с помощью метода Where (p=>p.Id>id), будет идти уже после выборки из бд в самом приложении:

```
IEnumerable<Phone> phoneIEnum = db.Phones;  
phoneIEnum = phoneIEnum.Where(p => p.Id > id);
```

Чтобы совместить фильтры:

```
IEnumerable<Phone> phoneIEnum = db.Phones.Where(p => p.Id > id);
```

С IQueryable все методы суммируются, запрос оптимизируется, и только потом происходит выборка из базы данных:

```
IQueryable<Phone> phoneIQuer = db.Phones;  
phoneIQuer=phoneIQuer.Where(p => p.Id > id);
```

*Если нужен весь набор возвращаемых данных – IEnumerable, предоставляющий максимальную скорость. Если нужны только отфильтрованные данные – IQueryable.*

# SQL в Entity Framework

## SQL в EF:

```
// выборка
using (PhoneContext db = new PhoneContext())
{
    var comps = db.Database.SqlQuery<Company>("SELECT * FROM Companies");
    foreach (var company in comps)
        Console.WriteLine(company.Name);
}
```

```
// выборка (SQL с параметром)
using (PhoneContext db = new PhoneContext())
{
    System.Data.SqlClient.SqlParameter param = new
    System.Data.SqlClient.SqlParameter("@name", "%Samsung%");
    var phones = db.Database.SqlQuery<Phone>("SELECT * FROM Phones
    WHERE Name LIKE @name", param);
    foreach (var phone in phones)
        Console.WriteLine(phone.Name);
}
```

# SQL в Entity Framework

## SQL в EF:

```
// вставка
int numberOfRowsInserted = db.Database.ExecuteSqlCommand("INSERT
INTO Companies (Name) VALUES ('HTC')");
// обновление
int numberOfRowsUpdated = db.Database.ExecuteSqlCommand("UPDATE
Companies SET Name='Nokia' WHERE Id=3");
// удаление
int numberOfRowsDeleted = db.Database.ExecuteSqlCommand("DELETE FROM
Companies WHERE Id=3");
```

```
// вызов хранимой процедуры
using (PhoneContext db = new PhoneContext())
{
    System.Data.SqlClient.SqlParameter param = new
System.Data.SqlClient.SqlParameter("@name", "Samsung");
    var phones = db.Database.SqlQuery<Phone>("GetPhonesByCompany
@name", param);
    foreach (var p in phones)
        Console.WriteLine("{0} - {1}", p.Name, p.Price);
}
```

# Аннотации

Аннотации представляют настройку сопоставления моделей и таблиц с помощью атрибутов. Большинство классов аннотаций располагаются в пространстве `System.ComponentModel.DataAnnotations`:

- ▶ **Key** – установка свойства в качестве первичного ключа
- ▶ **DatabaseGenerated(DatabaseGeneratedOption.Identity)** – автогенерация значения (для Id по умолчанию)
- ▶ **Required** – обязательное значение свойства
- ▶ **MaxLength** и **MinLength** – ограничения на длину значения свойства
- ▶ **Table** и **Column** – сопоставление с таблицей БД и столбцом



# Аннотации

```
public class Customer
{
    [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int CustomerId { get; set; }

    [Required]
    [MaxLength(30)]
    public string FirstName { get; set; }

    public string LastName { get; set; }

    [Range(8, 100)]
    [Column("ModelAge")]
    public int Age { get; set; }

    [Column(TypeName = "image")]
    public byte[] Photo { get; set; }

    public int? RegionId { get; set; }
    // ВНЕШНИМ КЛЮЧОМ ДЛЯ СВЯЗИ С МОДЕЛЬЮ Region будет служить свойство RegId, а не Region
    [ForeignKey("RegId")]
    public Region Region { get; set; }
}
```



# Fluent API

Настройка соглашений о конфигурации базы данных с помощью аннотаций является довольно простой, но она не может обеспечить проецирование сложной конфигурации, например создания отношения many-to-many между таблицами. *Fluent API* дает доступ к таким настройкам.

Преимуществом Fluent API перед аннотациями является то, что он не засоряет код модели и организует взаимосвязь модели с контекстом данных.

Fluent API передает контексту дополнительные данные для конфигурации, благодаря переопределению метода *DbContext.OnModelCreating()*, который вызывается перед тем, как контекст построит сущностную модель данных

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

# Fluent API

```
public class SampleContext : DbContext
{
    public SampleContext() : base("MyShop")
    { }
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Order> Orders { get; set; }
    protected override void OnModelCreating(DbModelBuilder
modelBuilder)
    {
        // настройка полей с помощью Fluent API
        modelBuilder.Entity<Customer>()
            .Property(c =>
c.FirstName).IsRequired().HasMaxLength(30);

        modelBuilder.Entity<Customer>()
            .Property(c => c.Email).HasMaxLength(100);

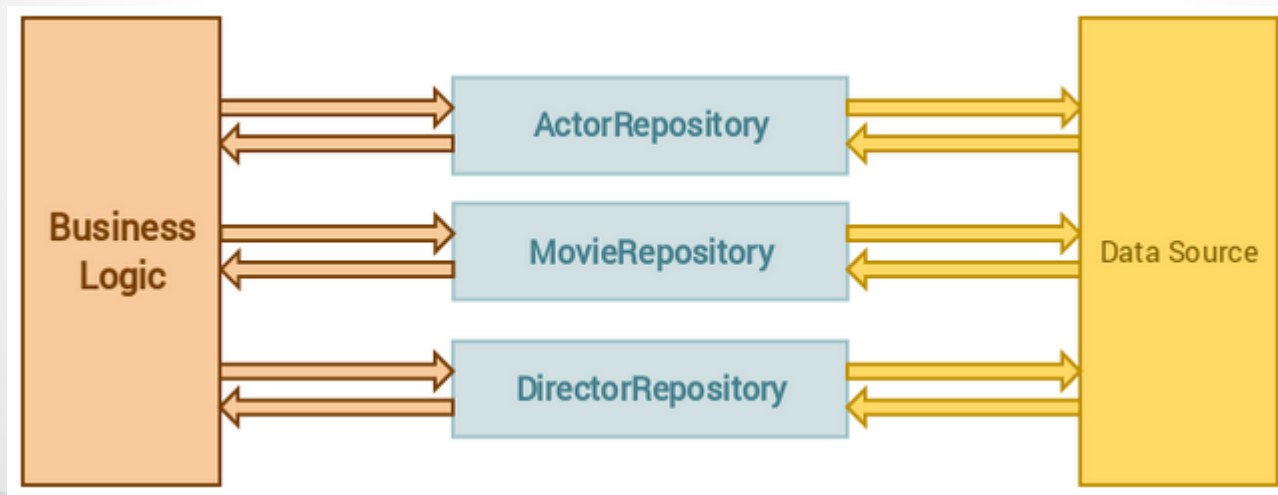
        modelBuilder.Entity<Customer>()
            .Property(c => c.Photo).HasColumnType("image");

        // настройка таблицы
        modelBuilder.Entity<Customer>().ToTable("NewName_Customer");
    }
}
```

# ШАБЛОН РЕПОЗИТОРИЙ (REPOSITORY)

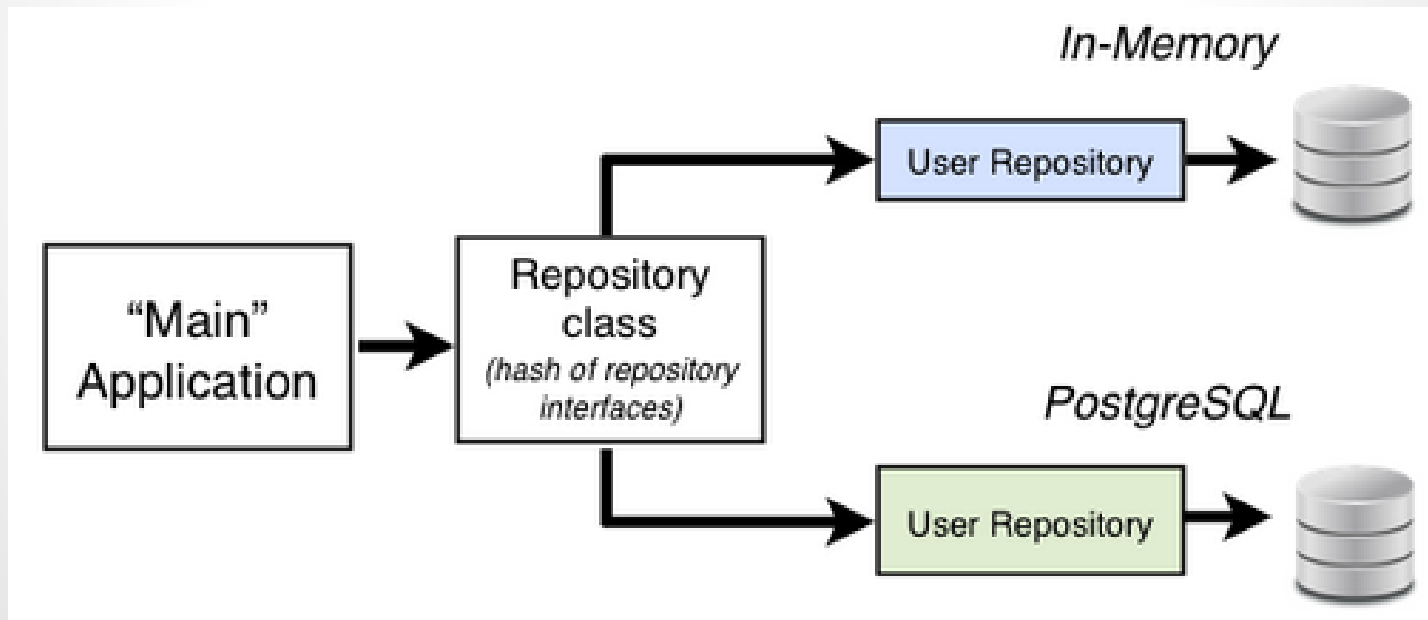
# Шаблон Репозиторий

Одним из наиболее часто используемых паттернов при работе с данными является паттерн 'Репозиторий'. Репозиторий позволяет абстрагироваться от конкретных подключений к источникам данных, используемых в программной системе, и является промежуточным звеном между классами, непосредственно взаимодействующими с данными, и остальной программой.



# Шаблон Репозиторий

Реализации репозиториев можно подменять, изменяя источники данных. Классы, использующие данные, поставляемые репозиториями, от типа источника данных никак не зависят



# Репозиторий

1. В программе есть следующий класс модели

```
public class Book
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Author { get; set; }
    public int Price { get; set; }
}
```

2. И класс контекста данных

```
public class BookContext : DbContext
{
    public BookContext() : base("DefaultConnection")
    { }
    public DbSet<Book> Books { get; set; }
}
```

# Репозиторий

## 3. определим интерфейс репозитория

```
interface IRepository<T> : IDisposable
    where T : class
{
    IEnumerable<T> GetBookList(); // получение всех объектов
    T GetBook(int id); // получение одного объекта по id
    void Create(T item); // создание объекта
    void Update(T item); // обновление объекта
    void Delete(int id); // удаление объекта по id
    void Save(); // сохранение изменений
}
```

# Репозиторий

## 4. создадим реализацию репозитория для MS SQL Server

```
public class SQLBookRepository : IRepository<Book>
{
    private BookContext db;

    public SQLBookRepository()
    {
        this.db = new BookContext();
    }

    public IEnumerable<Book> GetBookList()
    {
        return db.Books;
    }

    public Book GetBook(int id)
    {
        return db.Books.Find(id);
    }

    public void Create(Book book)
    {
        db.Books.Add(book);
    }
}
```



# Репозиторий

## 4. создадим реализацию репозитория для MS SQL Server

```
public void Update(Book book)
{
    db.Entry(book).State = EntityState.Modified;
}

public void Delete(int id)
{
    Book book = db.Books.Find(id);
    if(book!=null)
        db.Books.Remove(book);
}

public void Save()
{
    db.SaveChanges();
}

private bool disposed = false;

public virtual void Dispose(bool disposing)
{
    if(!this.disposed)

```

# Репозиторий

## 5. применим репозиторий в контроллере

```
public class HomeController : Controller
{
    IRepository<Book> db;

    public HomeController()
    {
        db = new SQLBookRepository();
    }

    public ActionResult Index()
    {
        return View(db.GetBookList());
    }
}
```



Code

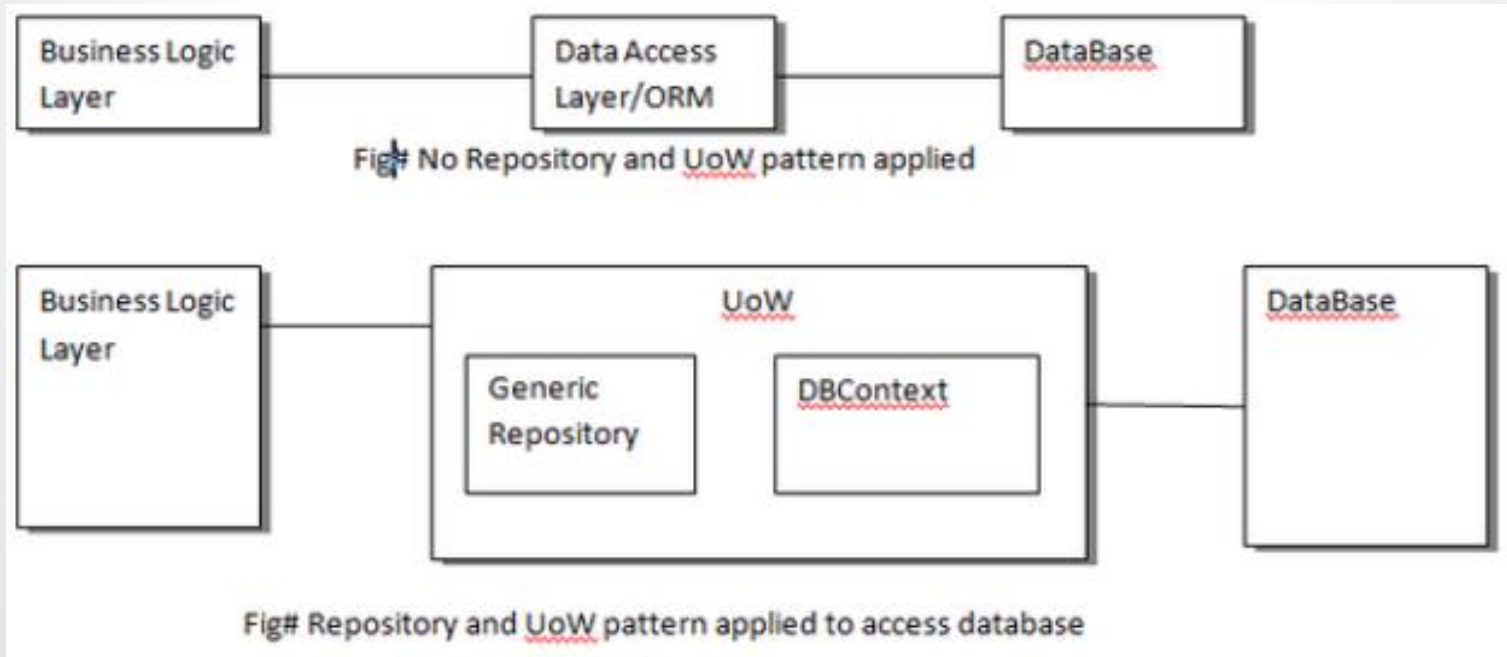
# Репозиторий

<b>Описание</b>	Все совместно используемые подсистемами данные хранятся в центральной базе данных, доступной всем подсистемам. Репозиторий является пассивным элементом, а управление им возложено на подсистемы.
<b>Рекомендации</b>	Логично использовать, если система обрабатывает большие объёмы данных.
<b>Преимущества</b>	<p>Совместное использование больших объёмов данных эффективно, поскольку не требуется передавать данные из одной подсистемы в другие. Подсистема не должна знать, как используются данные в других подсистемах - уменьшается степень связывания.</p> <p>В системах с репозиторием резервное копирование, обеспечение безопасности, управление доступом и восстановление данных централизованы, поскольку входят в систему управления репозиторием.</p>
<b>Недостатки</b>	<p>Все подсистемы должны быть согласованы со структурой репозитория (моделью данных). Модернизировать модель данных достаточно трудно</p> <p>К разным подсистемам предъявляются различные требования по безопасности, восстановлению и резервированию данных, а в паттерне Репозиторий ко всем подсистемам применяется одинаковая политика.</p>

# ШАБЛОН ЕДИНИЦА РАБОТЫ (UNIT OF WORK – UOW)

# Шаблон Единица работы

- ▶ Паттерн **Unit of Work** позволяет систематизировать работу с различными репозиториями и дает уверенность, что все репозитории будут использовать один и тот же контекст данных.



# Единица работы

1. Допустим, есть следующая пара моделей

```
public class Book
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}

public class Order
{
    public int Id { get; set; }
    public string Number { get; set; }
    public int BookId { get; set; }
    public Book Book { get; set; }
}
```

2. И контекст данных

```
public class OrderContext : DbContext
{
    public DbSet<Book> Books { get; set; }
    public DbSet<Order> Orders { get; set; }
}
```

# Единица работы

3. определен интерфейс репозитория и созданы две его отдельных реализации

```
interface IRepository<T> where T : class
{
    IEnumerable<T> GetAll();
    T Get(int id);
    void Create(T item);
    void Update(T item);
    void Delete(int id);
}
```

```
public class BookRepository : IRepository<Book>
{
    private OrderContext db;

    public BookRepository(OrderContext context)
    {
        this.db = context;
    }
}
```

```
public class OrderRepository : IRepository<Order>
{
    private OrderContext db;

    public OrderRepository(OrderContext context)
    {
        this.db = context;
    }
}
```

# Единица работы

4. Создается класс – точка входа в контекст данных с репозиториями

```
public class UnitOfWork : IDisposable
{
    private OrderContext db = new OrderContext();
    private BookRepository bookRepository;
    private OrderRepository orderRepository;

    public BookRepository Books
    {
        get
        {
            if (bookRepository == null)
                bookRepository = new BookRepository(db);
            return bookRepository;
        }
    }
}
```



# Единица работы

4. Создается класс – точка входа в контекст данных с репозиториями

```
public OrderRepository Orders
{
    get
    {
        if (orderRepository == null)
            orderRepository = new OrderRepository(db);
        return orderRepository;
    }
}

public void Save()
{
    db.SaveChanges();
}

private bool disposed = false;

public virtual void Dispose(bool disposing)
{
    if (!this.disposed)
    {
        if (disposing)
```

# Единица работы

## 5. применим единицу работы в контроллере

```
public class HomeController : Controller
{
    UnitOfWork unitOfWork;
    public HomeController()
    {
        unitOfWork = new UnitOfWork();
    }
    public ActionResult Index()
    {
        var books = unitOfWork.Books.GetAll();
        return View();
    }
}
```



Code

# Единица работы

Задача	При выполнении операций считывания или изменения объектов система должна гарантировать, что состояние базы данных останется согласованным. Например, на результат загрузки данных не должны влиять изменения, вносимые другими процессами.		
Решение	<p>Создается специальный объект, "отслеживающий" изменения, вносимые в базу данных. Данное типовое решение позволяет проконтролировать, какие объекты считываются и какие модифицируются и обслужить операции обновления содержимого базы данных.</p> <table><tr><th>ЕдиницаРаботы</th></tr><tr><td>зарегистрироватьНовый(объект) ЗарегистрироватьАктуальный(объект) зарегистрироватьВсе(объект) зарегистрироватьУдаленный(объект) Commit()</td></tr></table>	ЕдиницаРаботы	зарегистрироватьНовый(объект) ЗарегистрироватьАктуальный(объект) зарегистрироватьВсе(объект) зарегистрироватьУдаленный(объект) Commit()
ЕдиницаРаботы			
зарегистрироватьНовый(объект) ЗарегистрироватьАктуальный(объект) зарегистрироватьВсе(объект) зарегистрироватьУдаленный(объект) Commit()			
Преимущества	Нет необходимости явно вызывать методы сохранения, достаточно сообщить объекту Единица работы о необходимости фиксации (commit) результатов в базе данных. Вся сложная логика фиксации сосредоточена в одном месте, таким образом, координируется запись в базу данных.		

## Технология работы с данными Entity Framework

### Ресурсы:

1. **Э. Троелсен Язык программирования C# 5.0 и платформа .NET 4.5**
2. **<https://msdn.microsoft.com/ru-ru/library/bb399567%28v=vs.110%29.aspx>**
3. **<http://metanit.com/sharp/entityframework/1.1.php>**