





Entity Framework – teme abordate

- **Modele de programare**
 - ✓ **Database First;**
 - ✓ **Code First.**
 - ✓ **Model Design First.**
- **Entity Data Model – EDM**
- **Terminologie EF**
- **ObjectContext, ObjectSet**
- **DbContext, DbSet**
- **EF – Incarcare date relationate**
 - ✓ **Cereri cu DbContext.**
 - ✓ **Cereri pe date locale.**
 - ✓ **Iterare proprietati de navigare.**
 - ✓ **Lazy loading**
 - ✓ **Eager loading**
 - ✓ **Explicit loading**
 - ✓ **Incarcare continut proprietati de navigare tip colectie.**
- **Adaugare, modificare si stergere entitati.**
- **Entity Framework si aplicatii n-Tier**

Bibliografie

-  **MSDN**
-  **Julia Lerman: SECOND EDITION, Programming Entity Framework**
-  **Julia Lerman, Rowan Miller: Programming Entity Framework: DbContext**
-  **Brian Driscoll, Nitin Gupta, Rob Vettor, Zeeshan Hirani, Larry Tenny : Entity Framework 6 Recipes 2nd Edition**

Entity Framework - EF

Entity Framework este un set de tehnologii in ADO.NET ce suporta dezvoltarea de aplicatii software cu baze de date, aplicatii orientate pe obiecte. Comenzile din ADO.NET lucreaza cu scalar (date la nivel de coloana dintr-o tabela) in timp ce ADO.NET Entity Framework lucreaza cu obiecte (din baza de date se returneaza obiecte).

Arhitectura ADO.NET Entity Framework consta din urmatoarele:

- *Provideri specifici pentru sursa de date (Data source)* ce abstractizeaza interfetele ADO.NET pentru conectare la baza de date cand programam folosind schema conceptuala (model conceptual).
- *Provider specific* bazei de date ce translateaza comenzile Entity SQL in cereri native SQL (comenzi SQL din limbajul de definire a bazei de date, limbajul de cereri).
- *Parser EDM (Entity Data Model) si mapare vizualizari* prin tartarea specificatiilor SDL (Storage Data language – model de memorare) al modelului de date, stabilirea asociatiilor dintre modelul relational (baza de date) si modelul conceptual. Din schema relationala se creaza vizualizari ale datelor ce corespund modelului conceptual. Informatii din mai multe tabele sunt agregate intr-o entitate. Actualizarea bazei de date (apel metoda **SaveChanges()**) are ca efect construirea comenzilor SQL specifice fiecarei tabele ce apare in acea entitate.
- *Servicii pentru metadata* ce gestioneaza metadata entitatilor, relatiilor si maparilor.
- *Tranzactii* – pentru a suporta posibilitatile tranzactionale ale bazei de date.
- *Utilitare pentru proiectare* – Mapping Designer – incluse in mediul de dezvoltare.
- *API pentru nivelul conceptual* – runtime ce expune modelul de programare pentru a scrie cod folosind nivelul conceptual. Se folosesc obiecte de tip Connection, Command asemanator cu ADO.NET si se returneaza rezultate de tip EntityResultSets sau EntitySets.
- *Componente deconectate* – realizeaza un cache local pentru dataset si multimele entitati.
- *Nivel de programare* – ce expune EDM ca o constructie programabila si poate fi folosita in limbajele de programare ceea ce inseamna *generare automata de cod* pentru clasele CLR ce expun aceleasi proprietati ca o entitate, permitand instantierea entitatilor ca obiecte .NET sau expun entitatile ca *servicii web*.

Modele de programare

1. Aplicatie centrata pe **baza de date** - Baza de date este in centrul aplicatiei.
 2. Aplicatie centrata pe **model** – modelul este in centrul aplicatiei. Accesul la date este facut pe baza modelului conceptual, model ce reflecta obiectele problemei de rezolvat. In acest caz exista posibilitatea de a folosi : *Code first* sau *Model design first*.
-
1. Modelul de programare **centrat pe baza de date**, presupune ca baza de date este creata si apoi se genereaza modelul logic ce contine tipurile folosite in logica aplicatiei. Acest lucru se face folosind mediul de dezvoltare VStudio. Se genereaza clasele POCO (EF versiune mai mare ca 4 si VS 2012/VS 2013) si fisierele necesare pentru nivelul conceptual, nivelul de mapare si nivelul de memorare.

2. Aplicatia centrata pe **model** poate fi dezvoltata alegand una din variantele :
- Code first.
 - Model design first.

In cazul **Code First**, dezvoltatorul scrie toate clasele (POCO) modelului si clasa derivata din **DbContext** cu toate entitatile necesare si apoi cu ajutorul mediului de dezvoltare se creaza si genereaza baza de date, tabelele din baza de date si informatiile aditionale necesare pentru EF.

In cazul **Model Design First**, mediul de dezvoltare permite dezvoltarea unei diagrame a modelului aplicatiei si pe baza acesteia se va crea si genera baza de date, tabelele din baza de date si informatiile aditionale necesare pentru EF.

ADO.NET clasic presupune obiecte **DataReader** / **DataAdapter** pentru a citi informatii din baza de date si obiecte **Command** pentru a executa insert, update, delete, etc. **DataAdapter** din ADO .NET este folosit pentru **DataSet**.

In EF randurile si coloanele din tabele sunt returnate ca **obiecte** si nu se foloseste in mod direct **Command**, se translateaza datele din forma tabelara in obiecte.

Observatie

Obiecte de tip **Command** pot fi folosite pentru accesul direct la baza de date sau pentru cazurile pe care EF nu le poate rezolva.

EF foloseste un model numit *Entity Data Model* (EDM), dezvoltat din *Entity Relationship Modeling* (ERM).

Conceptele principale introduse de EDM sunt:

- **Entity**: entitatile sunt instante ale tipului Entity (de exemplu *Customer*, *Order*). Acestea reprezinta structura unei inregistrari identificata printr-o cheie. Entitatile sunt grupate in multimi de Entity (*Entity-Sets*).
- **Relationship**: relatiile asociaza entitatile si sunt instante ale tipurilor Relationship (de exemplu *Order* postat de *Customer*). Relatiile sunt grupate in multimi de relatii – *Relationship-Sets*.

EDM suporta diverse constructii ce extind aceste concepte de entitate si relatie:

- **Inheritance**: tipurile entitate pot fi definite astfel incat sa fie derivate din alte tipuri. Mostenirea in acest caz este una structurala, adica nu se mosteneste comportarea ci numai structura tipului entitate de baza.; in plus la aceasta mostenire a structurii, o instanta a tipului entitate derivat satisface relatia “is a”.
- **Tipuri complexe**: EDM suporta definirea tipurilor complexe si folosirea acestora ca membri ai tipurilor entitate. De exemplu se poate defini tipul (clasa) **Address** ce are proprietatile *Street*, *City* si *Telephone* si apoi sa folosim o proprietate de tip **Address** in tipul entitate *Customer*.

ERM defineste o schema a entitatilor si a relatiilor dintre acestea.

Entitățile definesc schema unui obiect, dar nu și comportarea acestuia. Entitatea este asemănătoare cu schema unei tabele din baza de date numai că aceasta descrie schema obiectelor problemei de rezolvat - pe scurt *modelul*.

ERM apare pentru prima dată în lucrarea lui Dr. Peter Chen: “The Entity-Relationship Model—Toward a Unified View of Data” (<http://csc.lsu.edu/news/erd.pdf>) – 1976.

EDM este un model pe partea de client și constituie fundamentul pentru EF.

EDM constă din trei niveluri, care sunt independente :

- ✓ nivelul conceptual (sau model conceptual) ;
- ✓ nivelul de mapare (sau model de mapare);
- ✓ nivelul de memorare (sau model de memorare).

Clasele entitate sunt continute în *modelul conceptual* din EDM.

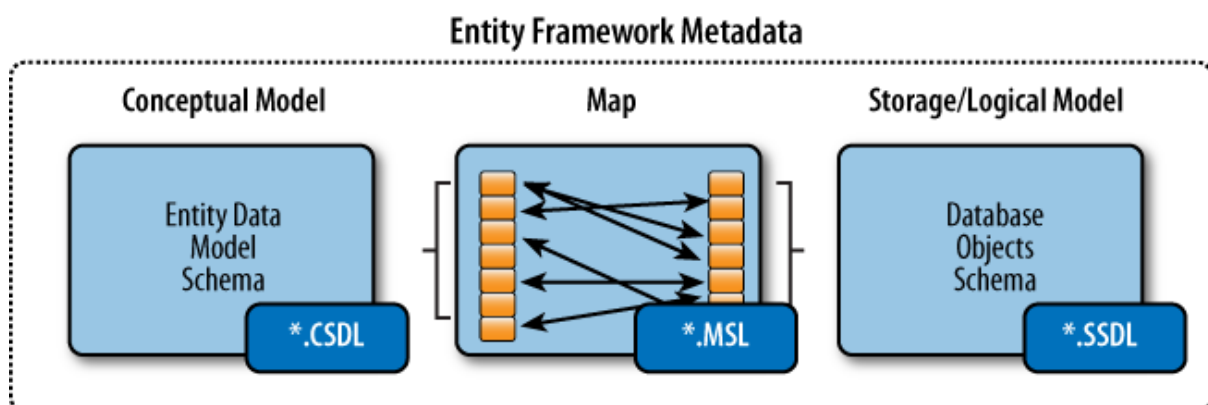
Nivelul conceptual poate fi modelat prin scriere directă de cod (*model de programare code first*) – modelul este în centrul aplicației – sau folosind un utilitar pentru generarea entităților în situația când avem baza de date proiectată (*model de programare database first*). Sintaxa pentru nivelul conceptual este definită de *Conceptual Schema Definition Language* (CSDL).

Nivelul de memorare din EDM definește tabele, coloane, relații și tipuri de date ce sunt mapate la baza de date. Sintaxa pentru modelul de memorare este definită de *Store Schema Definition Language* (SSDL).

Nivelul de mapare definește maparea (legătura) dintre nivelul conceptual și nivelul de memorare. Printre altele, acest nivel definește cum sunt mapate proprietățile din clasele entitate la coloanele tabelor din baza de date.

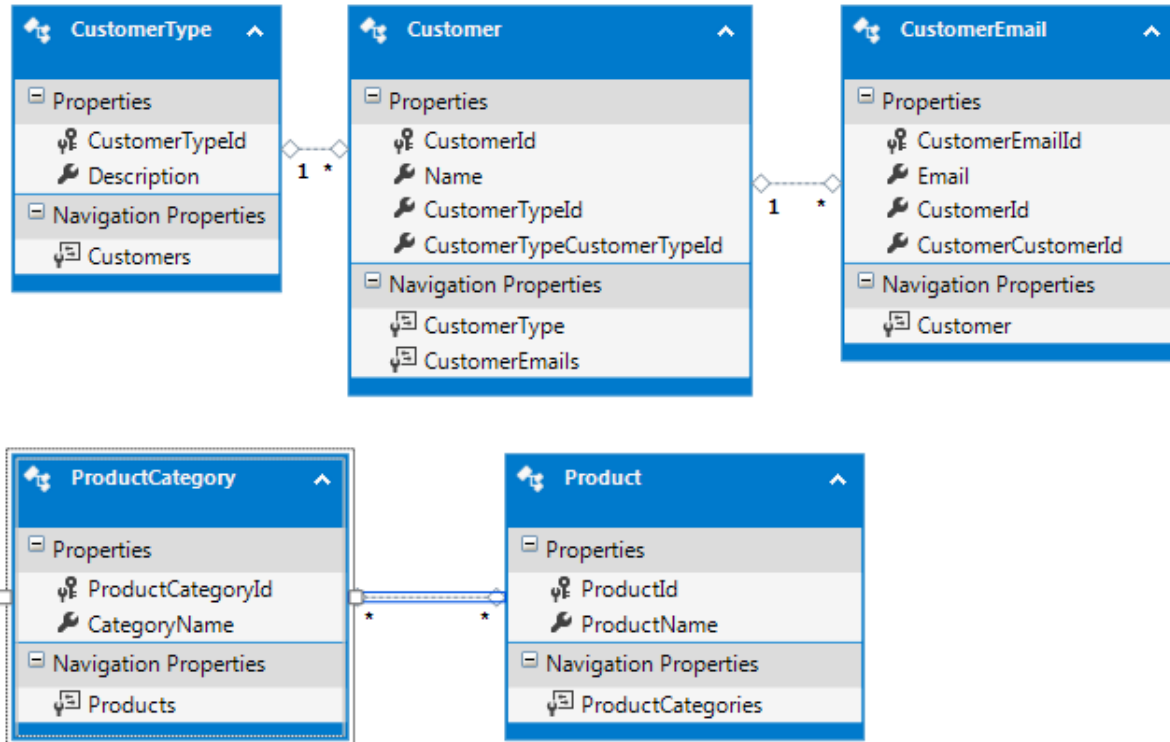
Mapping Specification Language (MSL) definește sintaxa pentru nivelul de memorare.

În EF, EDM este reprezentat în timpul proiectării de un singur fișier XML din care la runtime se creează trei fișiere cu extensiile : *csdl*, *msl*, *ssdl*. Aceste fișiere conțin descrierile pentru modelul conceptual, modelul de mapare și modelul de memorare.



Terminologie

EntityType reprezinta o clasa din model. O instanta a **EntityType** este referita ca o entitate. In mediul de dezvoltare, **EntityType** se reprezinta ca un dreptunghi cu diverse proprietati indicate in interior.



EntityType poate contine proprietati. Aceste proprietati se impart in :

Proprietati de navigare care se refera la alte entitati relationate – in mod obisnuit reprezentate de relatiile furnizate de *foreign key* din bazele de date. Proprietatile de navigare pot fi de tip *referinta* sau de tip *colectie*.

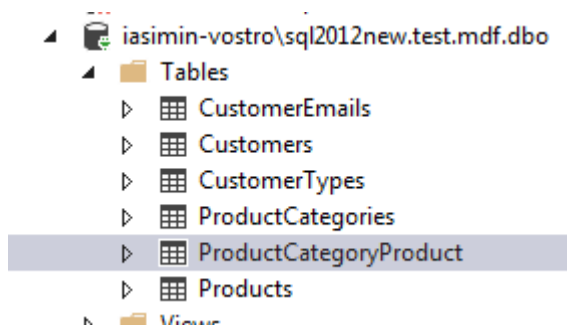
Proprietati scalare : alte proprietati in afara celor de navigare – proprietati automate in C# si reprezinta in fapt numele coloanelor dintr-o tabela si tipul de data asociat acestora.

O relatie intre doua entitati se numeste **asociere**. Asocierea poate avea multiplicitate (cardinalitate) : 1-0..1, 1-n, n-n.

EntityKey – identifica in mod unic entitatea in Entity Framework, si cel mai adesea este mapata la cheia primara corespunzatoare tablei din baza de date. **EntityKey** este o proprietate in sens C#.

Cele cinci entitati (exemplul de mai sus) formeaza impreuna un **obiect graf**. Un obiect graf este o vizualizare a unei entitati date, impreuna cu entitatile cu care relationeaza la un moment dat. De exemplu, la un moment dat, un **Customer** cu Id = 10 poate avea numele “Alfa”, **CustomerType.Name** = “Preferat” si o colectie de 5 obiecte de tip **CustomerEmail**.

Fiecarei clase de mai sus ii corespunde o tabela in baza de date. Numele tabelor sunt: **CustomerTypes**, **Customers**, **CustomerEmails**, **Products**, **ProductCategories** si **ProductCategoryProduct**.



Observatie

1. Numele tabelelor corespunde cu numele entitatilor din clasa *ModelContainer*. Descrierea acestei clase, derivata din **DbContext**, este data mai jos.
2. Relatia *many-to-many* are ca efect construirea unei tabeli suplimentare, de legatura, intre tabelele implicate in aceasta relatie. In cazul de fata tabela suplimentara este *ProductCategoryProduct* care are definite doua chei straine ce puncteaza la cheile primare din tabelele implicate in relatie.

Tabela de legatura *ProductCategoryProduct*, intre tabelele *ProductCategories* si *Products* este definita (de catre mediul de dezvoltare) astfel :

```
CREATE TABLE [dbo].[ProductCategoryProduct] (
    [ProductCategories_ProductCategoryId] INT NOT NULL,
    [Products_ProductId] INT NOT NULL,
    CONSTRAINT [PK_ProductCategoryProduct] PRIMARY KEY CLUSTERED
([ProductCategories_ProductCategoryId] ASC, [Products_ProductId] ASC),
    CONSTRAINT [FK_ProductCategoryProduct_ProductCategory] FOREIGN KEY
([ProductCategories_ProductCategoryId]) REFERENCES [dbo].[ProductCategories]
([ProductCategoryId]),
    CONSTRAINT [FK_ProductCategoryProduct_Product] FOREIGN KEY ([Products_ProductId])
REFERENCES [dbo].[Products] ([ProductId])
);
```

Tabela are o cheie primara definita de coloanele *ProductCategories_ProductCategoryId* si *Products_ProductId*, si doua *foreign keys*.

Pentru diagrama de mai sus clasele generate sunt (dau numai primele doua clase):

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

    public int CustomerTypeId { get; set; }
    public string Description { get; set; }

    public virtual ICollection<Customer> Customers { get; set; }
}

public partial class Customer
```

```
{
    public Customer()
    {
        this.CustomerEmails = new HashSet<CustomerEmail>();
    }

    public int CustomerId { get; set; }
    public string Name { get; set; }
    public string CustomerTypeId { get; set; }
    public int CustomerTypeCustomerId { get; set; }

    public virtual CustomerType CustomerType { get; set; }
    public virtual ICollection<CustomerEmail> CustomerEmails { get; set; }
}
```

Contextul este definit dupa cum urmeaza:

```
public partial class Model1Container : DbContext
{
    public Model1Container()
        : base("name=Model1Container") { }

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

    public virtual DbSet<Customer> Customers { get; set; }
    public virtual DbSet<CustomerType> CustomerTypes { get; set; }
    public virtual DbSet<CustomerEmail> CustomerEmails { get; set; }
    public virtual DbSet<ProductCategory> ProductCategories { get; set; }
    public virtual DbSet<Product> Products { get; set; }
}
```

Observati cum sunt modelate relatiile 1->* (one-to-many) in C# prin definirea proprietatilor de navigare de tip referinta si/sau de tip colectie. Un obiect *CustomerType* poate avea asociat mai multe obiecte de tip *Customer*, deci in clasa *CustomerType* se defineste proprietatea de navigare de tip colectie:

```
public virtual ICollection<Customer> Customers { get; set; }
```

Un obiect de tip *Customer* face referire la un singur *CustomerType*, si ca atare in clasa *Customer* se defineste proprietatea de navigare de tip referinta :

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

In acest mod se implementeaza in EF relatiile definite de chei primare si chei straine. In cazul relatiilor *->* (many-to-many) in fiecare clasa se defineste o proprietate de navigare de tip colectie. Aceste colectii contin instante ale tipului cu care intra in relatie tipul dat.

Contextul in Entity Framework

Pana la versiunea 4 (Visual Studio 2008), contextul in EF era definit numai de clasele *ObjectContext* si *ObjectSet*. Incepand cu versiunea 4.1 s-au creat clasele *DbContext* si *DbSet*, clase ce sunt wrapper-e pentru *ObjectContext* si *ObjectSet*. Pentru a intelege

mai bine calsele `DbContext` si `DbSet` vom studia mai intai clasele `ObjectContext` si `ObjectSet`. Nu vom prezenta exemple cu EF versiunea 4.

Clasa **ObjectContext**

Context object (clasa `ObjectContext`) reprezinta accesul la serviciile din EF.

ObjectContext

- expune obiecte entitate;
 - gestioneaza conexiunile la baza de date;
 - genereaza SQL parametrizat;
 - transfera date din / in baza de date;
 - realizeaza cache la obiecte;
- jurnalizeaza modificarile efectuate;
- materializeaza sau transforma o multime rezultat fara tip intr-o colectie de obiecte puternic tipizate.

`ObjectContext` interactioneaza cu clasele `ObjectSet`, `ObjectQuery`.

Observatie

`DbContext` interactioneaza cu clasele `DbSet`, `DbQuery`.

Clasa **ObjectContext** permite interactiunea cu date ca obiecte ce sunt instante ale tipurilor entitati, definite in modelul conceptual.

O instanta a tipului **ObjectContext** contine informatii despre:

- ✓ conexiune la baza de date, in forma unui obiect **EntityConnection**.
- ✓ Metadata ce descrie modelul, in forma unui obiect **MetadataWorkspace**.
- ✓ Un obiect **ObjectStateManager** ce gestioneaza obiectele memorate in cache.

Clasa ce reprezinta **EntityContainer** pentru model, este derivata din **ObjectContext**.

Metode (MSDN) uzuale pentru ObjectContext.

Name	Description
AcceptAllChanges	Accepts all changes made to objects in the object context.
AddObject	Adds an object to the object context.
CreateDatabase	Creates the database by using the current data source connection and the metadata in the StoreItemCollection .
DeleteDatabase	Deletes the database that is specified as the database in the current data source connection.
DeleteObject	Marks an object for deletion.
DetectChanges	Ensures that ObjectStateEntry changes are synchronized with changes in all objects that are tracked by the ObjectStateManager .
ExecuteStoreCommand	Executes an arbitrary command directly against the data

	source using the existing connection.
SaveChanges ()	Persists all updates to the data source and resets change tracking in the object context.
SaveChanges (SaveOptions)	Persists all updates to the data source with the specified SaveOptions.
Attach	Attaches an object or object graph to the object context when the object has an entity key.
AttachTo	Attaches an object or object graph to the object context in a specific entity set.

Clasa ObjectSet

Un obiect din **ObjectSet** reprezinta o multime entitate tipizata folosita pentru operatii de creare, modificare, citire si stergere din baza de date.

Exemplu de declarare (EF versiunea 4).

Proprietatea *Customers* va returna multimea inregistrarilor din tabela “*Customer*” din baza de date (asa a fost definit modelul), folosind conexiunea data de un obiect de tip *CustomerOrderEntities*.

```
public partial class CustomerOrderEntities :ObjectContext
{
    // cod lipsa
    public ObjectSet<Customer> Customers
    {
        get
        {
            if ((_Customers == null))
            {
                _Customers = base.CreateObjectSet<Customer>("Customers");
            }
            return _Customers;
        }
    }
    private ObjectSet<Customer> _Customers;
}
```

Clasa *Customer* va contine numai proprietati. Aceste proprietati corespund coloanelor tabelului *Customer* existenta in baza de date. Observati modul de dfinire al proprietatii *Customers*.

Metode importante pentru ObjectSet (MSDN)

Name	Description
AddObject	Adds an object to the object context in the current entity set.
ApplyCurrentValues	Copies the scalar values from the supplied object into the object in the ObjectContext that has the same key.

ApplyOriginalValues	Sets the OriginalValues property of an ObjectStateEntry to match the property values of a supplied object.
Attach	Attaches an object or object graph to the object context in the current entity set.
DeleteObject	Marks an object for deletion.
Detach	Removes the object from the object context.
Execute	Executes the object query with the specified merge option. (Inherited from ObjectQuery(T) .)
Include	Specifies the related objects to include in the query results. (Inherited from ObjectQuery(T) .)

Etapele necesare pentru adaugare, regasire si stergere informatii din baza de date sunt definite in continuare.

Adaugare

Pentru a *adauga* o inregistrare in tabela *Customer* (in cazul nostru) procedam in modul urmator :

1. Cream o instanta a tipului *Customer*. Completam proprietatile corespunzatoare.
2. `Customer customer = new Customer(...)` ;
3. Adaugam obiectul la colectie. Vom folosi metoda :
4. `AddObject()` din `ObjectSet` sau `AddObjectTo` definita in clasa *Customer*.
5. Salvare in baza de date : folosim metoda `SaveChanges()` pe un obiect de tip `ObjectContext`.

Modificare

Pentru a *modifica* o inregistrare trebuie sa efectuam urmatoorii pasi :

1. Conexiune la baza de date – creare instanta tip derivat din `ObjectContext`.
2. Determinare in cadrul colectiei a obiectului (lor) ce trebuie(sc) modificat(e). Metode de tip `select`, `Find` din LINQ, etc.
3. Modificare obiect.
4. Salvare in baza de date : folosim metoda `SaveChanges()` pe un obiect de tip `ObjectContext`.

Stergere

Pentru a *sterge* o inregistrare trebuie sa efectuam urmatoorii pasi :

1. Conexiune la baza de date – creare instanta tip derivat din `ObjectContext`.
2. Determinare in cadrul colectiei a obiectului (lor) ce trebuie(sc) sters(e). Metode de tip `select`, `Find` din LINQ, etc.
3. Apel metoda `DeleteObject`, ce are ca efect marcarea obiectului pentru stergere.
4. Stergere efectiva - salvare in baza de date : folosim metoda `SaveChanges()` pe un obiect de tip `ObjectContext`.

Metodele din `ObjectSet` lucreaza pe colectii de obiecte.

Metodele `ExecuteStoreQuery` si `ExecuteStoreCommand` – acces direct la baza de date

Metodele `ExecuteStoreQuery` si `ExecuteStoreCommand` din `ObjectContext` permit executia de comenzi SQL direct la nivel de baza de date, fara a folosi modelul conceptual generat. Din EDM se foloseste doar conexiunea.

Exista doua prototipuri pentru `ExecuteStoreQuery` (MSDN).

```
public ObjectResult<TElement> ExecuteStoreQuery<TElement>(
    string commandText,
    params Object[] parameters
)
```

Type Parameters

TElement

Parameters

commandText

Type: [System.String](#)

The command to execute, in the native language of the data source.

parameters

Type: [System.Object\[\]](#)

An array of parameters to pass to the command.

Return Value

Type: [System.Data.Objects.ObjectResult<TElement>](#)

An enumeration of objects of type *TResult*.

Metoda `ExecuteStoreQuery` foloseste conexiunea existenta pentru a executa o comanda in mod direct asupra sursei de date. Se foloseste tranzactia existenta, daca o asemenea tranzactie exista.

`ExecuteStoreQuery` echivalentul metodei `ExecuteReader` din clasa `DbCommand` cu observatia ca `ExecuteStoreQuery` returneaza entitati si nu valori scalare precum `ExecuteReader`.

Putem apela metoda `Translate` pentru a transforma un `DbDataReader` in obiecte entitate in situatia cand reader-ul contine inregistrari ce sunt mapate la un tip entitate specificat.

Valoarea pentru *parameters* poate fi un array de `DbParameter` sau un array de valori pentru parametru.

Al doilea prototip pentru `ExecuteStoreQuery` este:

```
public ObjectResult<TEntity> ExecuteStoreQuery<TEntity>(
    string commandText,
    string entitySetName,
    MergeOption mergeOption,
    params Object[] parameters
)
```

)

Type Parameters

TEntity

Parameters

commandText

Type: [System.String](#)

The command to execute, in the native language of the data source.

entitySetName

Type: [System.String](#)

The entity set of the *TResult* type. If an entity set name is not provided, the results are not going to be tracked.

mergeOption

Type: [System.Data.Objects.MergeOption](#)

The [MergeOption](#) to use when executing the query. The default is [AppendOnly](#).

parameters

Type: [System.Object\[\]](#)

An array of parameters to pass to the command.

Return Value

Type: [System.Data.Objects.ObjectResult](#)<*TEntity*>

An enumeration of objects of type *TResult*.

Exemple (MSDN)

Pasare parametri la o comanda SQL

Trei modalitati diferite pentru a pasa parametri la o comanda SQL.

Cererile returneaza un tip string.

```
using (SchoolEntities context = new SchoolEntities())
{
    // 1. Pattern substitutie parametru.
    // Returneaza string.
    foreach (string name in context.ExecuteStoreQuery<string>
        ("Select Name from Department where DepartmentID < {0}", 5))
    {
        Console.WriteLine(name);
    }

    // 2. Utilizare folosind sintaxa parametrizata. Asemnator cu 1.
    foreach (string name in context.ExecuteStoreQuery<string>
        ("Select Name from Department where DepartmentID < @p0", 5))
    {
        Console.WriteLine(name);
    }

    // 3. Utilizare explicita SqlParameter.
```

```
foreach (string name in context.ExecuteStoreQuery<string>
    ("Select Name from Department where DepartmentID < @p0",
     new SqlParameter { ParameterName = "p0", Value = 5 }))
{
    Console.WriteLine(name);
}
}
```

Exemplu (MSDN) : Metodele ExecuteStoreCommand si ExecuteStoreQuery

Se citesc inregistrari din tabela *Department* si se genereaza obiecte de tip *DepartmentInfo*. Vedeti metoda **ExecuteStoreQuery** prezentata mai jos.

Se creaza o clasa *DepartmentInfo* ce are numele proprietatilor aceleasi cu numele coloanelor din tabela *Department*. Se vor insera inregistrari in tabela *Department*, se va obtine un obiect *DepartmentInfo* executand o cerere asupra tabeli *Department* si in final se va sterge inregistrarea adaugata.

```
public class DepartmentInfo
{
    private DateTime _startDate;
    private String _name;
    private Int32 _departmentID;
    // Se definesc proprietatile pentru cele trei date membru private
    public Int32 DepartmentID
    {
        get { return _departmentID; }
        set { _departmentID = value; }
    }
    public String Name
    {
        get { return _name; }
        set { _name = value; }
    }
    public DateTime StartDate
    {
        get { return _startDate; }
        set { _startDate = value; }
    }
}

public static void ExecuteStoreCommands()
{
    // Executie comenzi SQL direct asupra bazei de date. Din EF
    // se foloseste numai conexiunea
```

```
using (SchoolEntities context = new SchoolEntities())
{
    int DepartmentID = 21;
    // Insert inregistrare in tabela Department.
    // Se foloseste substitutia parametrilor.
    int rowsAffected = context.ExecuteStoreCommand(
        "Insert Department values ({0}, {1}, {2}, {3}, {4})",
        DepartmentID, "Engineering", 350000.00, "2009-09-01", 2);
    Console.WriteLine("Number of affected rows: {0}", rowsAffected);

    // Obtinem obiectul DepartmentInfo.
    DepartmentInfo department =
        context.ExecuteStoreQuery<DepartmentInfo>
        ("Select * from Department where DepartmentID= {0}",
        DepartmentID).FirstOrDefault();

    Console.WriteLine("ID: {0}, Name: {1} ", department.DepartmentID,
        department.Name);

    // Stergere inregistrare din Departament
    rowsAffected = context.ExecuteStoreCommand(
        "Delete from Department where DepartmentID = {0}", DepartmentID);
    Console.WriteLine("Number of affected rows: {0}", rowsAffected);
}}
```

Exemplu (MSDN) : In actiune metoda Translate.

Se creaza un **DataReader** si apoi tipurile scalare din **DataReader** sunt transformate in obiecte de tip *Department*. In prima etapa se foloseste pentru conexiune clasa **SqlConnection**, iar in a doua etapa se foloseste conexiunea data de EF. Se returneaza un **DbDataReader** si apoi se translateaza **DbDataReader** in obiecte ale tipului *Department*.

```
// Initializare string conexiune
SqlConnectionStringBuilder sqlBuilder = new SqlConnectionStringBuilder();

sqlBuilder.DataSource = ".";
sqlBuilder.InitialCatalog = "School";
sqlBuilder.IntegratedSecurity = true;
// Conectare la baza de date folosind SqlConnection
SqlConnection con = new SqlConnection(sqlBuilder.ToString());
{
    con.Open();
    SqlCommand cmd = con.CreateCommand();
```

```
cmd.CommandText = @"SELECT * FROM Department";

// Creare reader ce contine inregistrari din Department.
using (DbDataReader rdr =
    cmd.ExecuteReader(CommandBehavior.SequentialAccess))
{
    // Conectare la baza de date folosind contextul din EF
    using (SchoolEntities context = new SchoolEntities())
    {
        // Translateza reader-ul la obiecte de tipul Department.
        foreach (Department d in context.Translate<Department>(rdr))
        {
            Console.WriteLine("DepartmentID: {0} ", d.DepartmentID);
        }
    }
}
con.Close();
}
```

Exemplu (MSDN): ObjectQuery<TEntity>

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    // Se returneaza un obiect
    string esqlQuery = @"SELECT VALUE Contact
        FROM AdventureWorksEntities.Contacts as Contact where
        Contact.LastName = @ln";

    // Urmatoarea cerere returneaza o colectie de obiecte Contact.
    ObjectQuery<Contact> query = new ObjectQuery<Contact>(esqlQuery,
        context, MergeOption.NoTracking);
    query.Parameters.Add(new ObjectParameter("ln", "Zhou"));

    // Iterare colectie.
    foreach (Contact result in query)
        Console.WriteLine("Contact First Name: {0}; Last Name: {1}",
            result.FirstName, result.LastName);
}
```

sau folosind Linq

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
```



```
ObjectQuery<Contact> query= context.Contacts.Where("it.LastName==@ln",
    new ObjectParameter("ln", "Zhou"));

// Iterare colectie.
foreach (Contact result in query)
    Console.WriteLine("Contact First Name: {0}; Last Name: {1}",
        result.FirstName, result.LastName);}
```

Clasa DbContext

Conceptual aceasta clasa este similara cu **ObjectContext**. Se foloseste pentru a executa cereri asupra EDM si a efectua actualizarea bazei de date (insert, update, delete, cereri). Aceasta clasa este folosita in versiunile EF mai mari sau egale cu 4.1.

Exemplu de clasa derivata din DbContext :

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

Observatie

Daca folosim modelul de programare *Code First*, atunci aceasta clasa trebuie sa o scriem noi. Daca folosim modelul de programare *Database first*, atunci aceasta clasa este generata de un utilitar apelat de catre mediul de dezvoltare Visual Studio.

Odata ce avem un context, putem adauga (vezi metodele **Add** si **Attach**) sau sterge (vezi metoda **Remove**) entitati din acest context folosind proprietatile definite in cadrul lui (in exemplu *Categories* si *Products*). Accesarea unei proprietati **DbSet** din cadrul contextului reprezinta startul unei cereri ce returneaza toate entitatile de tipul specificat. Faptul ca accesam o proprietate nu inseamna ca cererea se executa. O cerere este executata cand:

- Este enumerata in *foreach*.
- Este enumerata de o operatiune din colectie cum ar fi *ToArray*, *ToDictionary* sau *ToList*.
- Sunt specificati in cadrul cererii operatorii LINQ *First* sau *Any* .
- Sunt apelate urmatoarele metode: metoda extinsa *Load* pe un *DbSet*, *DbEntityEntry.Reload* si *Database.ExecuteSqlCommand*.

Timpul de viata al contextului – recomandari

Un context este valabil din momentul in care s-a creat o instanta a acestuia si este disponibil pana cand se apeleaza metoda **Dispose** sau este luata in considerare de catre garbage collector. Se recomanda folosirea lui *using*.

```
using (var context = new ProductContext())
{
    // Acces la date folosind contextul
}
```

Recomandarile Microsoft pentru lucrul cu contextul in EF sunt urmatoarele:

- Performanta aplicatiei poate scadea in cazul cand incarcam in memorie multe obiecte din cadrul aceluiasi context. Consum sporit de memorie plus accesarea obiectelor are nevoie de mai mult timp procesor.
- Daca o exceptie are ca efect trecerea contextului intr-o stare pe care sistemul nu o poate gestiona este posibil ca aplicatia sa se termine.
- Daca timpul dintre incarcarea datelor unei cereri si actualizarea acestora este mare, e posibil sa apara probleme legate de concurenta.
- In aplicatiile Web, trebuie sa folosim instanta unui context pe fiecare cerere.
- In aplicatiile WPF sau Windows Forms se recomanda utilizarea unei instante a contextului la nivel de fereastră (formular).

DbContext se foloseste de regula cu un tip derivat ce contine proprietati DbSet<TEntity> pentru entitatile modelului. Aceste multipli sunt initializate in mod automat cand se creaza o instanta a clasei derivate. Aceasta comportare poate fi suprimata daca aplicam atributul SuppressDbSetInitializationAttribute, fie la intreaga clasa context, fie la proprietati individuale din cadrul clasei.

Proprietati pentru DbContext :

ChangeTracker	Provides access to features of the context that deal with change tracking of entities.
Configuration	Provides access to configuration options for the context.
Database	Creates a Database instance for this context that allows for creation/deletion/existence checks for the underlying database.

Metode pentru DbContext (MSDN)

Name	Description
Entry (Object) sau forma generica Entry<TEntity> (TEntity)	Gets a DbEntityEntry object for the given entity providing access to information about the entity and the ability to perform actions on the entity.

OnModelCreating()	This method is called when the model for a derived context has been initialized, but before the model has been locked down and used to initialize the context. The default implementation of this method does nothing, but it can be overridden in a derived class such that the model can be further configured before it is locked down.
SaveChanges()	Saves all changes made in this context to the underlying database.
SaveChangesAsync() SaveChangesAsync(CancellationToken)	Asynchronously saves all changes made in this context to the underlying database.
Set(Type)	Returns a non-generic DbSet instance for access to entities of the given type in the context, the ObjectStateManager, and the underlying store.
Set<TEntity>()	Returns a DbSet instance for access to entities of the given type in the context, the ObjectStateManager, and the underlying store.

Executie comenzi SQL la nivel de baza de date – proprietatea Database

Prin proprietatea **Database** obtinem un obiect Database. Acest obiect expune metode ce pot executa comenzi SQL (DDL/DML) la nivel de baza de date.

Clasa **Database** expune proprietatea **Connection** ce permite recrearea unui obiect **DbConnection** daca acesta nu exista.

Metode importante din clasa - Database:

Name	Description
BeginTransaction()	Begins a transaction on the underlying store connection.
BeginTransaction(IsolationLevel)	Begins a transaction on the underlying store connection using the specified isolation level.
Create	Creates a new database on the database server for the model defined in the backing context. Note that calling this method before the database initialization strategy has run will disable executing that strategy.
CreateIfNotExists	Creates a new database on the database server for the model defined in the backing context, but only if a database with the same name does not already exist on the server.

Delete()	Deletes the database on the database server if it exists, otherwise does nothing. Calling this method from outside of an initializer will mark the database as having not been initialized. This means that if an attempt is made to use the database again after it has been deleted, then any initializer set will run again and, usually, will try to create the database again automatically.
ExecuteSqlCommand (String, Object[])	Executes the given DDL/DML command against the database.
ExecuteSqlCommandAsync (String, CancellationToken, Object[])	Asynchronously executes the given DDL/DML command against the database.
SqlQuery (Type, String, Object[]) Cazul generic SqlQuery<TElement> (String, Object[])	Creates a raw SQL query that will return elements of the given (generic) type. The type can be any type that has properties that match the names of the columns returned from the query, or can be a simple primitive type. The type does not have to be an entity type. The results of this query are never tracked by the context even if the type of object returned is an entity type. Use the SqlQuery(String, Object[]) method to return entities that are tracked by the context.

Exemplu. Cereri cu SqlQuery pentru entitati

Cereri SQL pentru *entitati* – Blogs este o entitate :

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.SqlQuery("SELECT * FROM dbo.Blogs").ToList();
}
```

Cereri SqlQuery pentru tipuri non-entitate – acces direct la tabela Blogs :

```
using (var context = new BloggingContext())
{
    var blogNames = context.Database.SqlQuery<string>(
        "SELECT Name FROM dbo.Blogs").ToList();
}
```

Clasa DbSet

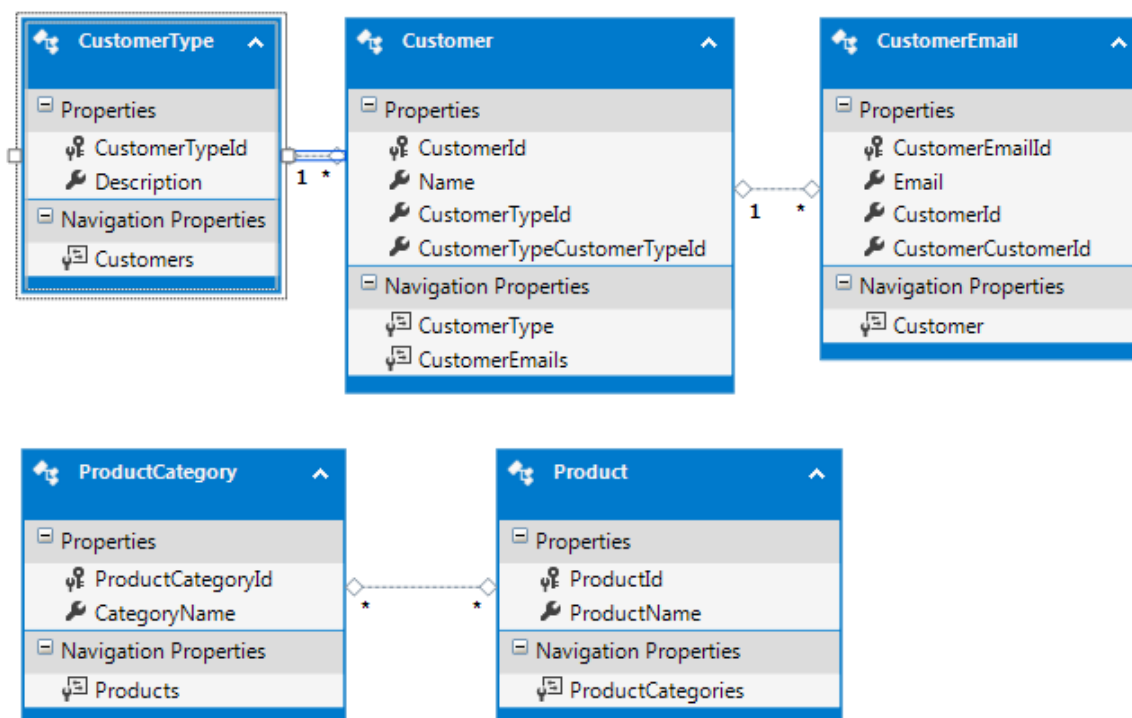
Aceasta clasa este folosita cand tipul entitatii nu este cunoscut in momentul constructiei. Este versiunea non-generica pentru `DbSet<TEntity>`.

Metode pentru DbSet (MSDN)

Name	Description
Add AddRange	Adds the given entity to the context underlying the set in the Added state such that it will be inserted into the database when <code>SaveChanges</code> is called.
Attach	Attaches the given entity to the context underlying the set. That is, the entity is placed into the context in the Unchanged state, just as if it had been read from the database.
Create() Create<TDerivedEntity>()	Creates a new instance of an entity for the type of this set. Note that this instance is NOT added or attached to the set. The instance returned will be a proxy if the underlying context is configured to create proxies and the entity type meets the requirements for creating a proxy.
Find	Finds an entity with the given primary key values. If an entity with the given primary key values exists in the context, then it is returned immediately without making a request to the store. Otherwise, a request is made to the store for an entity with the given primary key values and this entity, if found, is attached to the context and returned. If no entity is found in the context or the store, then null is returned.
Include	Specifies the related objects to include in the query results. (Inherited from DbQuery<TResult> .)
Remove RemoveRange	Marks the given entity as Deleted such that it will be deleted from the database when <code>SaveChanges</code> is called. Note that the entity must exist in the context in some other state before this method is called.
SqlQuery	Creates a raw SQL query that will return entities in this set. By default, the entities returned are tracked by the context; this can be changed by calling <code>AsNoTracking</code> on the DbSqlQuery<TEntity> returned. Note that the entities returned are always of the type for this set and never of a derived type. If the table or tables queried may contain data for other entity types, then the SQL query must be written appropriately to ensure that only entities of the correct type are returned.

Reprezentari ale modelului conceptual

Un model conceptual este o reprezentare specifica a structurii unor date vazute ca entitati si relatii.



In VS avem o reprezentare vizuala a EDM-ului si a membrilor sai. In spatele acestei reprezentari vizuale exista un fisier XML.

Metadata modelului (vezi cele trei fisiere)

- Baza de date
- Maparea

Alte denumiri utile

Conceptual Schema Definition Language (CSDL)

- • Conceptual layer
- • Conceptual schema
- • Conceptual model
- • C-side

Store Schema Definition Language (SSDL)

- • Store/storage layer
- • Store/storage schema
- • Store/storage model

- • Store/storage metadata
- • Store/storage metadata schema
- • S-side

Mapping Specification Language (MSL)

- • Mapping layer
- • Mapping specification
- • C-S mapping (referring to “conceptual to store”)

Ce exista in fisierul XML ce descrie EDM?

Doua sectiuni principale :

- informatii pentru runtime ;
 - modelul de memorare ;
 - modelul conceptual ;
 - maparea ;
- informatii pentru Designer.

CSDL: Schema conceptuala

EntityContainer
EntitySet
EntityType

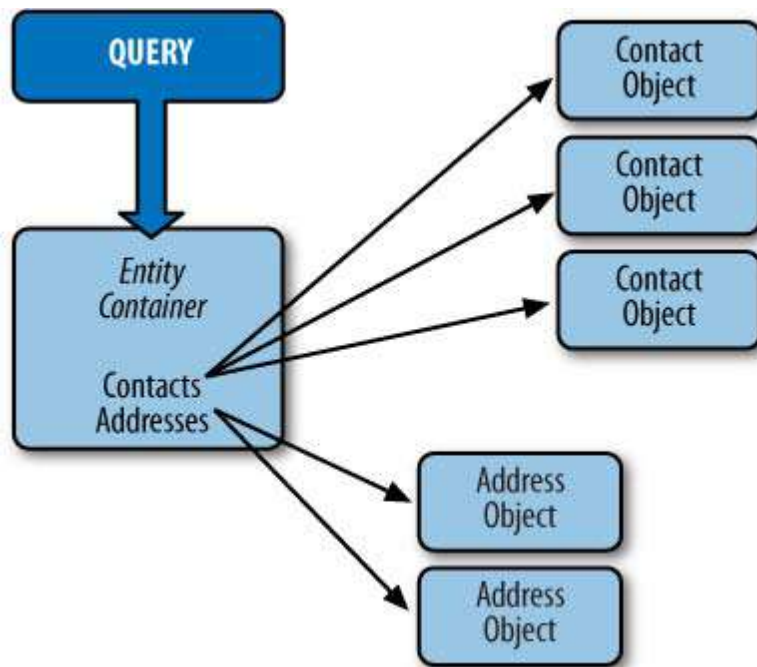
EntityContainer

Constituie punctul principal pentru a executa cereri asupra modelului.

Un **EntityContainer** este o grupare logica de multimi de entitati si multimi de asociatii.

La nivel conceptual, clasa **EntityContainer** reprezinta un container ce va fi mapat la un obiect baza de date in modelul de memorare. In modelul de memorare, clasa **EntityContainer** reprezinta o descriere a tabelii si/sau relatiilor.

EntityContainer expune **EntitySets** care dau acces la obiectele pe care le contin.



Proprietatea **LazyLoadingEnabled** determina modul cum sunt incarcate in memorie datele cu care relateaza un anumit obiect.

EntitySet

Un **EntitySet** este un container de tip entitate si are attributele **Name** si **EntityType**.

```

<EntitySet Name="Addresses" EntityType="SampleModel.Address" />
<EntitySet Name="Contacts" EntityType="SampleModel.Contact" />
    
```

EntityType

EntityType este tipul de data din model.

```

<EntityType Name="Address">
  <Key>
    <PropertyRef Name="addressID" />
  </Key>
  <Property Name="addressID" Type="Int32" Nullable="false"
    annotation:StoreGeneratedPattern="Identity" />
  <Property Name="Street1" Type="String" MaxLength="50"
    Unicode="true" FixedLength="true" />
  <Property Name="Street2" Type="String" MaxLength="50"
    Unicode="true" FixedLength="true" />
  ...
  <NavigationProperty Name="Contact"
    Relationship="SampleModel.FK_Address_Contact"
    FromRole="Address" ToRole="Contact" />
</EntityType>
    
```


.NET Framework Data Provider pentru Entity Framework.

Spatiul de nume **EntityClient**.

Provider-ul **EntityClient** foloseste clase specifice ADO.NET Data Provider si mapeaza metadata pentru interactiune cu modelele de date entitate. **EntityClient** translateaza operatiile efectuate pe entitatile conceptuale in operatii efectuate pe sursa de date fizica, translateaza multimile rezultat returnate din sursa fizica de date in entitati conceptuale.

Claselor din acest namespace sunt:

Class	Description
EntityCommand	Represents a command for the conceptual layer.
EntityConnection	Contains a reference to a conceptual model and a data source connection. This class cannot be inherited.
EntityConnectionStringBuilder	Provides a simple way to create and manage the contents of connection strings used by the EntityClient.
EntityDataReader	Reads a forward-only stream of rows from a data source.
EntityParameter	Represents a parameter used in EntityCommand.
EntityParameterCollection	Represents a collection of parameters associated with a EntityCommand .
EntityProviderFactory	Represents a set of methods for creating instances of a provider's implementation of the data source classes.
EntityTransaction	Specifies the transaction for an EntityCommand.

Observatie

Aceste clase se aseamana cu cele din ADO.NET situate in nivelul cu conexiune – mai putin **DataAdapter** care nu are corespondent aici pentru ca nu exista **DataSet** in EF.

In continuare va fi descrisa fiecare clasa.

Clasa EntityConnection

Namespace: [System.Data.EntityClient](#)

Assembly: System.Data.Entity (in System.Data.Entity.dll)

Contine o referinta la modelul conceptual si stabileste o conexiune la sursa de date.

Mod de lucru asemanator cu **Connection** din ADO.NET.

Cand construim un obiect **EntityConnection** trebuie sa furnizam stringul de conexiune.

Acest lucru poate fi facut in urmatoarele moduri:

- furnizam un string de conexiune complet;
- furnizam un string formatat ce contine cuvantul cheie **Name** astfel (bazat pe exemplul din curs):

Name=ModelContainer

unde *ModelContainer* este numele stringului de conexiune din fisierul de configurare.

Proprietatea **StoreConnection**, de tip **DbConnection**, din **EntityConnection** reprezinta o conexiune fizica la baza de date.

Proprietati EntityConnection

Name	Description
ConnectionString	Gets or sets the EntityConnection connection string.
ConnectionTimeout	Gets the number of seconds to wait when attempting to establish a connection before ending the attempt and generating an error.
Container	Gets the IContainer that contains the Component .
Database	Gets the name of the current database, or the database that will be used after a connection is opened.
DataSource	Gets the name or network address of the data source to connect to.
ServerVersion	Gets a string that contains the version of the data source to which the client is connected.
State	Gets the ConnectionState property of the underlying provider if the EntityConnection is open. Otherwise, returns Closed .
StoreConnection	Provides access to the underlying data source connection that is used by the EntityConnection object.

Metode EntityConnection

Name	Description
BeginTransaction ()	Begins a transaction by using the underlying provider.
BeginTransaction (IsolationLevel)	Begins a transaction with the specified isolation level by using the underlying provider.
Close	Closes the connection to the database.
CreateCommand	Creates a new instance of an EntityCommand ,

	with the Connection set to this EntityConnection.
CreateObjRef	Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object.
GetMetadataWorkspace	Returns the MetadataWorkspace associated with this EntityConnection.
GetSchema(String)	Returns schema information for the data source of this DbConnection using the specified string for the schema name.
Open, OpenAsync	Establishes a connection to the data source by calling the underlying data provider's Open method.

Exemplul cu EntityConnection, EntityCommand si EntityDataReader.

```

Console.WriteLine("Test EntityConnection, EntityCommand si
EntityDataReader");
using (var conn = new EntityConnection("Name = ModellContainer"))
{
    conn.Open();
    // Interogare MetadataWorkSpace pentru a obtine EntitySet
    MetadataWorkspace workspace = conn.GetMetadataWorkspace();
    var sets = from container in
                workspace.GetItems<EntityContainer>(DataSpace.CSpace)
                from set in container.BaseEntitySets
                where set is EntitySet
                select set;

    #region Afisare EntitySet din MetadataWorkSpace
    foreach (EntitySet set in sets)
    {
        Console.WriteLine(set.ElementType.Name);
    }
    #endregion

    string eSQL = "SELECT VALUE (c) FROM ModellContainer.Customers AS c " +
        " where c.CustomerId > 1";
    using (EntityCommand command = new EntityCommand(eSQL, conn))
    {
        EntityDataReader reader =
            command.ExecuteReader(CommandBehavior.SequentialAccess);
        while (reader.Read())
        {
            // coloana 1 contine Name din Customer.
            Console.WriteLine("Name = " + reader.GetString(1));
        }
    }
}

```

Clasa EntityConnectionStringBuilder

Furnizeaza o modalitate de a crea si gestiona continutul stringului de conexiune folosit de EntityClient.

Exemplu

```
// Specify the provider name, server and database.
string providerName = "System.Data.SqlClient";
string serverName = ".";
string databaseName = "AdventureWorks";

// Initialize the connection string builder for the
// underlying provider.
SqlConnectionStringBuilder sqlBuilder =
    new SqlConnectionStringBuilder();

// Set the properties for the data source.
sqlBuilder.DataSource = serverName;
sqlBuilder.InitialCatalog = databaseName;
sqlBuilder.IntegratedSecurity = true;

// Build the SqlConnection connection string.
string providerString = sqlBuilder.ToString();

// Initialize the EntityConnectionStringBuilder.
EntityConnectionStringBuilder entityBuilder =
    new EntityConnectionStringBuilder();

//Set the provider name.
entityBuilder.Provider = providerName;

// Set the provider-specific connection string.
entityBuilder.ProviderConnectionString = providerString;

// Set the Metadata location.
entityBuilder.Metadata = @"res://*/AdventureWorksModel.csdl|
                           res://*/AdventureWorksModel.ssdl|
                           res://*/AdventureWorksModel.msl";
Console.WriteLine(entityBuilder.ToString());

using (EntityConnection conn =
    new EntityConnection(entityBuilder.ToString()))
{
    conn.Open();
    Console.WriteLine("Just testing the connection.");
    conn.Close();
}
```

}

Clasa EntityCommand

Reprezinta o comanda pentru nivelul conceptual.

Constructorii sunt asemanatori cu cei de la clasa **Command** din ADO.NET.

Pot avea ca parametrii: fraza SQL; conexiunea; tranzactia.

Metodele mai importante sunt (diverse prototipuri):

ExecuteNonQuery; ExecuteReader; ExecuteScalar;

Vedeti exemplul de la clasa EntityConnection.

Clasa EntityDataReader

Folosita pentru a crea cursoare de tip forward dintr-o baza de date.

Proprietati EntityDataReader

Name	Description
FieldCount	Gets the number of columns in the current row. (Overrides DbDataReader.FieldCount .)
HasRows	Gets a value that indicates whether this EntityDataReader contains one or more rows. (Overrides DbDataReader.HasRows .)
IsClosed	Gets a value indicating whether the EntityDataReader is closed. (Overrides DbDataReader.IsClosed .)
Item[Int32]	Gets the value of the specified column as an instance of Object . (Overrides DbDataReader.Item .)
Item[String]	Gets the value of the specified column as an instance of Object . (Overrides DbDataReader.Item .)
RecordsAffected	Gets the number of rows changed, inserted, or deleted by execution of the SQL statement. (Overrides DbDataReader.RecordsAffected .)

Metode EntityDataReader

Read – citire din data reader.

Metode de tip **Get...** pentru preluare date (GetInt32, GetDecimal, etc.)

IsDBNull() – test coloana daca are valoare completata sau este “null” (“null” in sensul bazelor de date).

NextResult() – trece la urmatoarea inregistrare din reader.

Exemplu Vedeti exemplul de la clasa EntityConnection.

Clasa EntityParameter

Reprezinta un parametru folosit de **EntityCommand**.

Constructori

Name	Description
EntityParameter()	Initializes a new instance of the EntityParameter class using the default values.
EntityParameter(String, DbType)	Initializes a new instance of the EntityParameter class using the specified parameter name and data type.
EntityParameter(String, DbType, Int32)	Initializes a new instance of the EntityParameter class using the specified parameter name, data type and size.

Clasa EntityTransaction

Infrastructura *System.Transactions* furnizeaza un model de programare explicit bazat pe clasa **Transaction** si un model de programare implicit bazat pe clasa **TransactionScope**, in acest din urma caz tranzactiile sunt gestionate in mod automat de infrastructura.

TransactionScope face un bloc de cod ca fiind tranzactional. Tranzactiile se aplica la nivel de baza de date si nu la multimea de obiecte din EF.

EntityTransaction specifica tranzactia pentru **EntityCommand**.

O tranzactie se specifica in cadrul unei conexiuni, metoda de tip **BeginTransaction()**.

Tranzactia obtinuta astfel se foloseste ca parametru la crearea unei comenzi (**EntityCommand**).

Proprietati – clasa Transaction

Name	Description
Connection	Gets EntityConnection for this EntityTransaction.
IsolationLevel	Gets the isolation level of this EntityTransaction. (Overrides DbTransaction.IsolationLevel .)

Metode - clasa Transaction

Cele mai importante metode sunt: **Commit()** si **Rollback()** care se aplica pe obiectul obtinut cu metoda **BeginTransaction()**.

In **TransactionScope**, metoda **Complete()** are rolul de a executa *commit* asupra bazei de date.

Daca in blocul de cod unde se foloseste **TransactionScope** apare o exceptie se va executa **Dispose()**, echivalent cu apelul metodei **Rollback()** din clasa **Transaction**.

Exemplu (MSDN) pentru TransactionScope

Scenariul este urmatorul: se lucreaza cu doua baze de date. Metoda **Complete()** va face *commit* –in caz de succes – pe ambele baze de date sau rollback in caz de insucces..

Infrastructura foloseste DTC – Distributed Transaction Coordinator.

Fiecare baza de date foloseste un alt string pentru conexiune.

```
// This function takes arguments for 2 connection strings and commands to
// create a transaction involving two SQL Servers. It returns a value > 0
// if the transaction is committed, 0 if the
// transaction is rolled back. To test this code, you can connect to two
// different databases on the same server by altering the connection
// string, or to another 3rd party RDBMS by
// altering the code in the connection2 code block.
static public int CreateTransactionScope(
    string connectString1, string connectString2,
    string commandText1, string commandText2)
{
    // Initialize the return value to zero and create a
    // StringWriter to display results.
    int returnValue = 0;
    System.IO.StringWriter writer = new System.IO.StringWriter();

    try
    {
        // Create the TransactionScope to execute the commands,
```

```
// guaranteeing that both commands can commit or roll back as a
// single unit of work.
using (TransactionScope scope = new TransactionScope())
{
    using (SqlConnection connection1 =
        new SqlConnection(connectString1))
    {
        // Opening the connection automatically enlists it in the
        // TransactionScope as a lightweight transaction.
        connection1.Open();

        // Create the SqlCommand object and execute the first
        // command.
        SqlCommand command1 =
            new SqlCommand(commandText1, connection1);
        returnValue = command1.ExecuteNonQuery();
        writer.WriteLine("Rows to be affected by command1: {0}",
            returnValue);

        // If you get here, this means that command1 succeeded.
        // By nesting the using block for connection2 inside that
        // of connection1, you conserve server and network
        // resources as connection2 is opened only when
        // there is a chance that the transaction can commit.
        using (SqlConnection connection2 =
            new SqlConnection(connectString2))
        {
            // The transaction is escalated to a full distributed
            // transaction when connection2 is opened.
            connection2.Open();

            // Execute the second command in the second database.
            returnValue = 0;
            SqlCommand command2 =
                new SqlCommand(commandText2, connection2);
            returnValue = command2.ExecuteNonQuery();
            writer.WriteLine("Rows to be affected by command2:" +
                {0}", returnValue);
        }
    }

    // The Complete method commits the transaction.
    // If an exception has been thrown,
    // Complete is not called and the transaction is rolled back.
    scope.Complete();
}
```



```

    }
    catch (TransactionAbortedException ex)
    {
        writer.WriteLine("TransactionAbortedException Message: {0}", " +
            ex.Message);
    }
    catch (ApplicationException ex)
    {
        writer.WriteLine("ApplicationException Message: {0}", ex.Message);
    }

    // Display messages.
    Console.WriteLine(writer.ToString());

    return returnValue;
}

```

Cereri pe un Entity Data Model - EDM

EF foloseste informatiile din model si fisierele de mapare pentru a traduce cererile asupra obiectelor din modelul conceptual in cereri specifice sursei de date.

EF furnizeaza urmatoarele modalitati de a interoga modelul conceptual si a returna obiecte:

- **LINQ to Entities.** Furnizeaza suport pentru Language-Integrated Query (LINQ) in vederea interogarii tipurilor de entitati definite in modelul conceptual.
- **Entity SQL.** Un dialect SQL independent de mediul de memorare ce lucreaza direct cu entitatile in modelul conceptual.
- Metode de constructie a cererilor folosind metodele din **LINQ to Objects**.
- **SQL** nativ.

EF include furnizorul de date **EntityClient**, furnizor ce gestioneaza conexiunile, parseaza cererile asupra entitatilor in cereri specifice sursei de date si returneaza un data reader folosit de EF pentru a materializa datele din obiecte.

Observatie – Ce folosim si cum?

1. Linq to Entities – depinde deObjectContext deci exista ChangeTracking si returneaza obiecte

```

var items = from c in context.Customers Where c.Name="AB" Select
c)

```

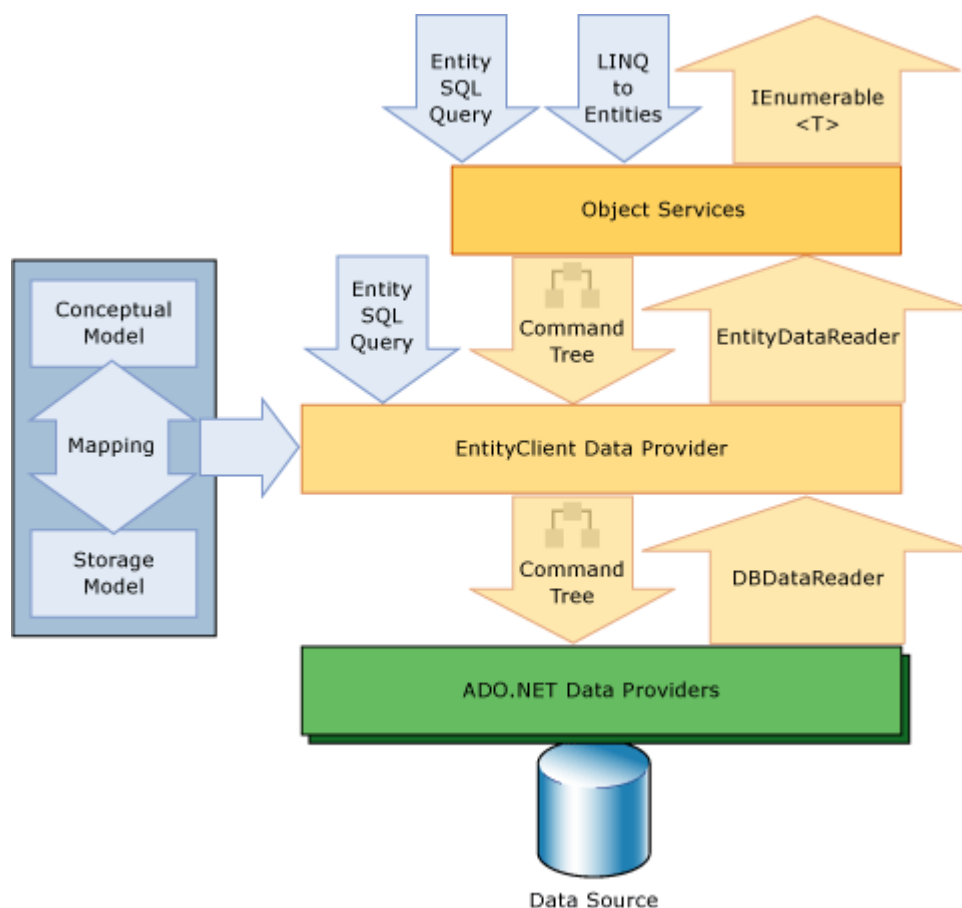
2. EntitySQL cu Object Services.

```
ObjectContext.CreateQuery<Customer>("Select VALUE c from Customer
as c WHERE c.Name=' AB '")
```

EntitySQL with EntityClient.

EntityClient este un provider pentru baza de date similar cu **SQLClient** sau **OleDbClient**. Diferenta este ca suntem conectati la modelul de date si nu la baza de date. Vom folosi **EntityConnection**, **EntityCommand**, **EntityDataReader**, etc.

Urmatoarea diagrama (MSDN) ilustreaza arhitectura EF pentru accesarea datelor:

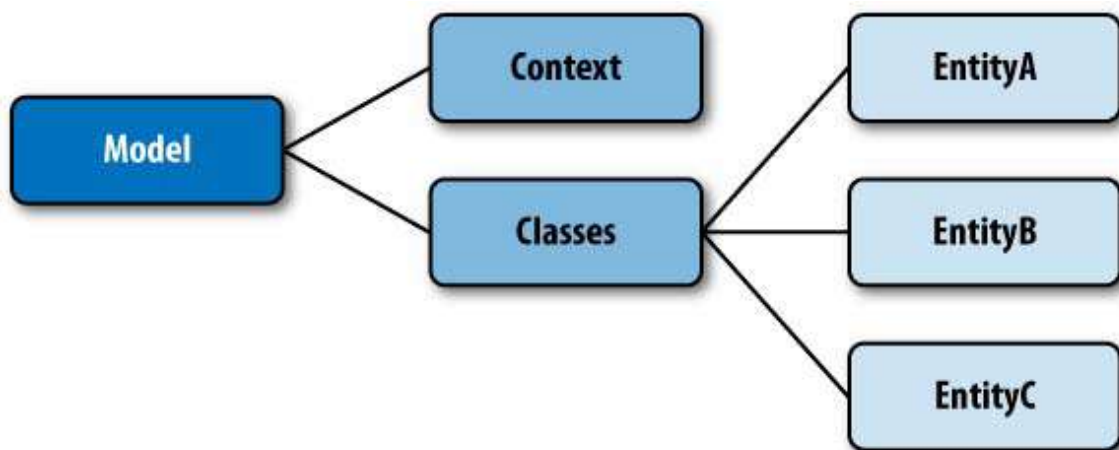


Utilitarele pentru EDM pot genera o clasa derivata din **ObjectContext** sau **DbContext**, clasa ce reprezinta un container in modelul conceptual.

Cererile se executa asupra modelului, nu asupra bazei de date. Folosind informatiile din modelul de mapare si cel de memorare se vor genera comenzi SQL specifice pentru a incarca in memorie datele din baza de date.

```
private static void QueryContacts()
```

```
{
    using (var context = new Model1Container())
    {
        var customers = context.Customers;
        foreach (var customer in customers)
        {
            Console.WriteLine("{0} {1}",
                customer.Name. toString(),
                customer.CustomerId);
        }
    }
    Console.Write("Press Enter...");
    Console.ReadLine();
}
```



Metodele CreateQuery si Execute dinObjectContext

namespace: **System.Data.Objects**

Metode folosite: **CreateQuery**, **Execute**.

Deoarece am folosit **DbContext** iar metodele **CreateQuery**, **Execute** sunt in **ObjectContext** trebuie sa facem conversie din **DbContext** la **ObjectContext**. Codul pentru conversie poate fi:

```
var adapter = (IObjectContextAdapter)ctx;
var contextObject = adapter.ObjectContext;
```

Observatie

O alta modalitate ar fi sa scriem o proprietate in clasa derivata din **DbContext**, proprietate ce returneaza un **ObjectContext**:

```
public ObjectContext ObjectContext
{
    get { return (this as IObjectContextAdapter).ObjectContext; }
}
```

Folosind modelul dat in curs putem scrie urmatoarea cerere in LINQ SQL:

```
using(var ctx = new ModellContainer())
{
    var adapter = (IObjectContextAdapter)ctx;
    var contextObject = adapter.ObjectContext;
    string queryString = "Select Value c From Customers as c where
                          c.CustomerId > 1";

    // se returneaza obiecte Customer
    ObjectQuery<Customer> customers =
        contextObject.CreateQuery<Customer>(queryString);
    IList<Customer> results =
        customers.Execute(MergeOption.AppendOnly).ToList();

    Console.WriteLine(results.Count<Customer>());
    foreach(var c in results)
    {
        Console.WriteLine("CustomerId = {0}, Name = {1}",
            c.CustomerId, c.Name);
    }
}
```

Observatie

Adaugati referinte si apoi using pentru:

```
using System.Data.Entity.Core;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.Core.Objects;
```

Cea mai simpla cerere poate fi, de aceasta data cu LINQ To Entities:

```
var context = new ModellContainer();
ObjectSet<Customer> customers = context.Customers;
context.Dispose();
```

In cele ce urmeaza se descriu cateva metode (interogare, insert, update, delete) pentru a lucra cu Entity Framework.

Modelul contine: *contextul si clasele*.

Exemplele se bazeaza pe o baza de date SQL Server ce contine doua tabele: *Customer* si *Order*.

Structurile tabelor sunt:

```
CREATE TABLE [dbo].[Customer] (
    [Id] INT NOT NULL,
    [Nume] NCHAR (10) NOT NULL,
    [Email] NCHAR (10) NOT NULL,
    PRIMARY KEY CLUSTERED ([Id] ASC)
);

CREATE TABLE [dbo].[Order] (
    [Id] INT IDENTITY (1, 1) NOT NULL,
    [CustomerId] INT NOT NULL,
    [Valoare] REAL NOT NULL,
    PRIMARY KEY CLUSTERED ([Id] ASC),
    CONSTRAINT [FK_Order_To_Customer] FOREIGN KEY ([CustomerId])
        REFERENCES [dbo].[Customer] ([Id])
);
```

);

La un proiect de tip Console Application s-a adaugat un articol de tip *ADO.NET Entity Data Model*. Parte din codul generat este redat in continuare.

Acesta e contextul.

```
namespace EFDbSet
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;

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

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

        public DbSet<Customer> Customers { get; set; }
        public DbSet<Order> Orders { get; set; }
    }
}
```

C clasele generate sunt:

```
namespace EFDbSet
{
    using System;
    using System.Collections.Generic;

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

        public int Id { get; set; }
        public string Nume { get; set; }
        public string Email { get; set; }

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

namespace EFDbSet
{
    using System;
    using System.Collections.Generic;

    public partial class Order
    {

```

```

    public int Id { get; set; }
    public int CustomerId { get; set; }
    public float Valoare { get; set; }

    public virtual Customer Customer { get; set; }
}

```

Observatie

In baza de date exista tabelele *Customer* si *Order* ce au drept coloane proprietatile din clasele *Customer* si *Order*. Tabela *Customer* are cheie primara *Id*, iar tabela *Order* are cheie straina *CustomerId* si cheie primara *Id* (conventie din Code First).

Clasa *CustomerOrdersEntities* derivata din *DbContext*, are un constructor definit astfel:

```

public CustomerOrdersEntities()
    : base("name=CustomerOrdersEntities")
{ }

```

Acest constructor nu face altceva decat sa citeasca din *App.config* urmatoarea linie:

```

<connectionStrings>
  <add name="CustomerOrdersEntities"
    connectionString="metadata=res://*/ModelCustomerOrder.csdl|res://*/ModelCustomerOrder.ssdl|res://*/ModelCustomerOrder.msl;
    provider=System.Data.SqlClient;provider_connection_string="
    data source=(LocalDB)\v11.0;attachdbfilename=
    D:\Documente\Cursuri\Net\2012\BazadeDate\Invoice.mdf;integrated
    security=True;
    connect timeout=30;MultipleActiveResultSets=True;
    App=EntityFramework";
    providerName="System.Data.EntityClient" />
</connectionStrings>

```

ce reprezinta stringul de conectare la baza de date. Pe baza acestui string de conectare se construiesc obiectul ce reprezinta in final conexiunea la baza de date.

CustomerOrdersEntities are doua proprietati (in acest caz):

```

public DbSet<Customer> Customers { get; set; }
public DbSet<Order> Orders { get; set; }

```

Aceste proprietati reprezinta in fapt multi de entitati. In exemplul de mai sus entitatile sunt *Customer* si *Order*.

Folosind acest model vom analiza modul de a actiona asupra bazei de date. Ceea ce ne intereseaza in mod special sunt comenzile *Select*, *Insert*, *Update*, *Delete* executate asupra bazei de date.

Interogari

Observatie

In majoritatea exemplelor variabila *context* este definita astfel:

```

CustomerOrdersEntities context = new CustomerOrdersEntities();

```

O prima interogare pe model ar putea fi :

```
CustomerOrdersEntities context = new CustomerOrdersEntities();
var setCustomers = context.Set<Customer>();

foreach (var x in setCustomers)
    Console.WriteLine(x.Id.ToString() + " -> " + x.Nume);
```

Observatie

Metoda folosita este **Set<>** din clasa **DbContext**.

Definitia metodei in MSDN este:

```
public DbSet<TEntity> Set<TEntity>() where TEntity : class
```

Metoda returneaza un **DbSet** pentru tipul specificat, si permite sa fie executate operatii CRUD pentru entitatea data in cadrul contextului.

C : Create - INSERT / POST
 R : Read (Retrieve) - SELECT / GET
 U : Update (Modify) - UPDATE / PUT
 D : Delete - DELETE / DELETE

Observatie

Varianta a doua la CRUD se refera la HTTP

2. Cereri cu Linq to Entities

```
var customers = from c in context.Set<Customer>() where c.Id == 2 select c;

foreach (var y in customers)
    Console.WriteLine(y.Id.ToString() + " " + y.Nume);
```

3. Cereri SQL pentru entitati

Metoda **SqlQuery** – returneaza instante ale entitatii.

SqlQuery pe **DbSet** permite executia de cereri SQL ce vor returna instante ale entitatii. Obiectele returnate sunt gestionate de context ca si cum ar fi returnate de cereri Linq.

```
using (var context = new CustomerOrdersEntities())
{
    var result = context.Customers.SqlQuery("SELECT * FROM customer").
        ToList();
}
```

Observatie

Cererea este executata imediat deoarece am folosita metoda **ToList()**.

4. Incarcare entitati folosind proceduri stocate

```
using (var context = new CustomerOrdersEntities())
{
    var custs = context.Customers.SqlQuery("dbo.GetCustomers").ToList();
}
```

unde *GetCustomers* este procedura stocata ce returneaza tipuri *Customer*.

Apel procedura cu parametri:

```
using (var context = new CustomerOrdersEntities())
{
    var Id = 1;

    var custs = context.Customers.SqlQuery("dbo.GetCustomerById @p0", Id)
        .Single();
}
```

GetCustomerById este procedura stocata ce are un parametru. Deoarece Id in Customer este cheie primara se va returna cel mult o inregistrare.

5. Cereri SQL pentru tipuri non-entitati

Acestea reprezinta o cerere SQL ce returneaza instante de orice tip, incluzand tipuri primitive; poate fi creata folosind metoda **SqlQuery** din clasa **Database**.

```
using (var context = new CustomerOrdersEntities())
{
    var items = context.Database.SqlQuery<string>(
        "SELECT Name FROM dbo.Persoana").ToList();
}
```

Observatie

Persoana este o tabela din baza de date dar care nu e in EDM creat. EDM creat contine numai entitatile Customer si Order.

Comenzi: Insert, Update, Delete

Metoda folosita este **ExecuteSqlCommand** din clasa **Database**.

```
using (var context = new CustomerEntities())
{
    context.Database.SqlCommand(
        "UPDATE dbo.Customer SET Nume = 'New Name' WHERE Id = 1");
}
```

Observatie

Dupa executia acestei comenzi, contextul trebuie reincarcat pentru a reflecta modificarile efectuate.

Posibilitati pentru *refresh* date.

Se poate folosi una din metodele de mai jos:

```
var objectContext = ((IObjectContextAdapter) context).ObjectContext;
```



```
ObjectContext.Refresh();
```

sau

```
var modifiedEntries = context.ChangeTracker.Entries()
    .Where(e => e.State == EntityState.Modified);
foreach (var modifiedEntry in modifiedEntries)
{
    modifiedEntry.Reload();
}
```

Exemple cu Insert, Update, Delete

Insert

```
Customer c = new Customer();
c.Id = id;
c.Nume = "New Customer";
c.Email = "email@email.world";

context.Set<Customer>().Add(c);
context.SaveChanges();
```

Update

```
Customer temp = new Customer();
temp.Id = id;
Customer cm = context.Customers.Find(temp.Id);
cm.Nume = " MmXx";
context.Entry<Customer>(cm).CurrentValues.SetValues(cm);
context.SaveChanges();
```

Delete

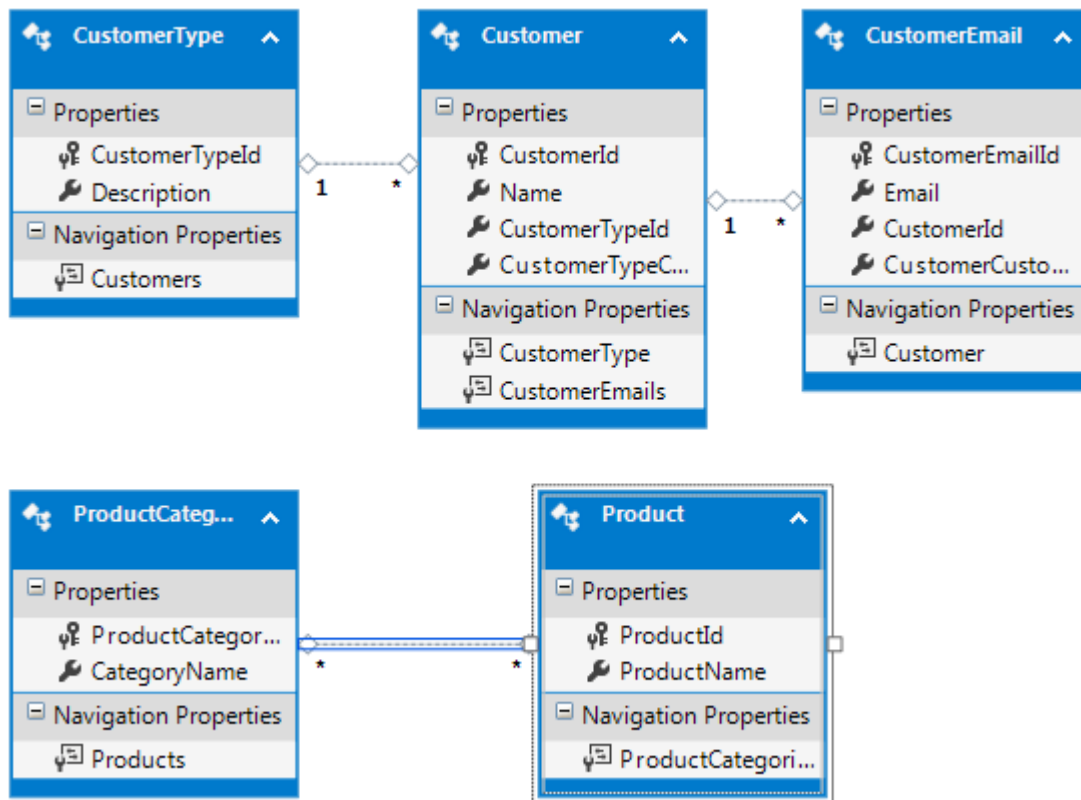
```
Customer temp = new Customer();
temp.Id = id;
Customer cm = context.Customers.Find(temp.Id);
context.Set<Customer>().Remove(cm);
context.SaveChanges();
```

Model folosit in exemple

Exemplele ce urmeaza au la baza modelul de mai jos.

Relatii 1 -> * (one-to-many) si * -> * (many-to-many)

In modul designer din VS 2013, cream aceste entitati si asocierele corespunzatoare.



Codul generat pentru acest model este dat mai jos (partial a fost dat si in cursul anterior).
 Generarea bazei de date se face selectand din meniul contextual: *Generate Database from Model...*

Observatie

Relatia “many-to-many” presupune existenta unei tabele suplimentare in baza de date.
 Scriptul de mai jos (redat partial) contine acea tabela

Relatia “many-to-many” presupune crearea unei tabele intermediare (

```
-- Creating table 'ProductCategoryProduct'
CREATE TABLE [dbo].[ProductCategoryProduct] (
    [ProductCategories_ProductCategoryId] int NOT NULL,
    [Products_ProductId] int NOT NULL
);
```

Clasele generate sunt prezentate in continuare.

```
namespace NetSpecial
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;

    public partial class Model1Container : DbContext
    {
        public Model1Container(): base("name=Model1Container")
        { }
    }
}
```

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

public virtual DbSet<Customer> Customers { get; set; }
public virtual DbSet<CustomerType> CustomerTypes { get; set; }
public virtual DbSet<CustomerEmail> CustomerEmails { get; set; }
public virtual DbSet<ProductCategory> ProductCategories { get; set; }
public virtual DbSet<Product> Products { get; set; }
}
}
```

Clașele ce corespund fiecărei tabele din baza de date.

```
public partial class CustomerType
{
    public CustomerType()
    {
        this.Customers = new HashSet<Customer>();
    }
    // Proprietati scalare
    public int CustomerTypeId { get; set; }
    public string Description { get; set; }
    // Entitate colectie - proprietate de navigare de tip colectie
    public virtual ICollection<Customer> Customers { get; set; }
}
public partial class Customer
{
    public Customer()
    {
        this.CustomerEmails = new HashSet<CustomerEmail>();
    }
    // Proprietati scalare
    public int CustomerId { get; set; }
    public string Name { get; set; }
    public string CustomerTypeId { get; set; }
    public int CustomerTypeCustomerId { get; set; }
    // Entitate referinta - proprietate de navigare de tip referinta
    public virtual CustomerType CustomerType { get; set; }
    // Entitate colectie
    public virtual ICollection<CustomerEmail> CustomerEmails { get; set; }
}
public partial class CustomerEmail
{
    // Proprietati scalare
    public int CustomerEmailId { get; set; }
    public string Email { get; set; }
    public string CustomerId { get; set; }
    public int CustomerCustomerId { get; set; }
    // Entitate referinta
    public virtual Customer Customer { get; set; }
}
```

Partea a doua a – Relatie “mai multe la mai multe” (many-to-many)

Observam ca fiecare clasa contine o colectie de tipuri ale celeilalte clase.

```
public partial class ProductCategory
{
    public ProductCategory()
```

```

{
    this.Products = new HashSet<Product>();
}

public int ProductCategoryId { get; set; }
public string CategoryName { get; set; }

public virtual ICollection<Product> Products { get; set; }
}
public partial class Product
{
    public Product()
    {
        this.ProductCategories = new HashSet<ProductCategory>();
    }

    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public virtual ICollection<ProductCategory> ProductCategories { get; set; }
}

```

Cele cinci entitati formeaza impreuna un *obiect graf*. Un *obiect graf* este o vizualizare, la un moment dat de timp, a unei entitati date impreuna cu entitatile cu care relationeaza.

Conversie DbContext laObjectContext

Clasa DbContext este declarata astfel:

```
public class DbContext : IDisposable, IOObjectContextAdapter
```

Cu ajutorul interfetei **IOObjectContextAdapter** putem face conversia unui obiect **DbContext** in **ObjectContext** si sa apelam metode din **ObjectContext**.

In exemplul urmatoar apelam metoda **CreateQuery** din **ObjectContext**.

Obtinerea unui ObjectContext este realizata prin urmatoarele linii de cod:

```

var adapter = (IOObjectContextAdapter)ctx;
var contextObject = adapter.ObjectContext;

```

Observatie

Daca folosim des in cod obiecte **ObjectContext**, atunci putem scrie o proprietate in clasa derivata din **DbContext**, in cazul de fata *Model1Container*, care sa returneze obiectul **ObjectContext** dorit. Codul poate fi urmatoarul:

```

public partial class Model1Container : DbContext
{
    public ObjectContext ObjectContext
    {
        get { return (this as IOObjectContextAdapter).ObjectContext; }
    }
    // Urmeaza cod din clasa
}

```

Cereri cu DbContext

Cea mai simpla cerere este creata prin iterarea unui DbSet din clasa derivata din DbContext (in cazul de fata *Model1Container*). In exemplul ce urmeaza se aduc toate inregistrarile din tabela Customers.

```
using (var context = new Model1Container())
{
    var query = context.Customers.ToList();
    return query;
}
sau
using (var context = new Model1Container())
{
    var query = context.CustomerTypes;
    foreach(var ct in query)
        Console.WriteLine(ct.Description);
}
```

Observatie

Daca nu folosim pattern-ul **using** pentru obiectul derivat din **DbContext**, atunci trebuie sa folosim metoda **Dispose()** pe acest obiect.

Exemplu - pattern using

```
// ...
using (var context = new Model1Container())
{
    var query= from d in context.Customers
               orderby d.Name
               select d;
    return query.ToList();
}
```

Exemplu cu Dispose()

```
var context = new Model1Container()
var query= from d in context.Customers
           orderby d.Name
           select d;
var result = query.ToList();
context.Dispose();
```

Cereri folosind LINQ to Entities

Iteram colectia *Customers* : obtinem toate inregistrarile din baza de date din tabela Customers. Pentru a obtine inregistrarile din tabellele *CustomerTypes* si *CustomerEmails* cu care relationeaza un *Customer* folosim proprietatile de navigare (entitate referinta, entitate colectie) din clasa *Customer*.

```
// cod lipsa
using (var context = new Model1Container())
{
    var query= from d in context.Customers
               orderby d.Name
               select d;
    return query.ToList();
}
```

Cand iteram continutul unui **DbSet**, EF va cere de la baza de date de fiecare data inregistrările pentru tipul solicitat. Pentru a nu deteriora performanta aplicatiei trebuie sa analizam ce cereri vor fi *amanate* la executie si ce cereri trebuie sa se execute imediat. In exemplul de mai sus cererea se executa imediat, se foloseste **ToList()**.

Iterarea pe oricare *DbSet* definit in *ModelContainer* are ca efect aducerea din baza de date a inregistrărilor din tabela ce corespunde tipului selectat. EF va emite comenzile SQL necesare catre baza de date, noi in cod iteram o multime de obiecte.

Exemplu – se aduc toate inregistrările din tabela *Customer*. In termeni ADO .NET asta inseamna:

- deschiderea unei conexiuni la baza de date – creare obiect *Connection* si apel metoda *Open()*;
- crearea unei comenzi (obiect de tip *Command*) ce contine cererea “**Select * From Customer**”;
- crearea unui *DataReader*;
- citirea inregistrărilor din *DataReader*;
- inchidere *DataReader* si conexiune la baza de date.

In EF toate aceste actiuni sunt exprimate in codul de mai jos.

```
// cod lipsa
using (var context = new ModelContainer())
{
    foreach(var item in context.Customers)
        Console.WriteLine(item.Name);
}
```

Gasirea unui singur obiect

Scenariul cel mai comun este de a scrie o cerere ce va returna un singur obiect cu o anumita cheie data. **DbContext** API are implementata metoda **Find** pentru acest scenariu care returneaza obiectul cautat sau *null* in caz contrar.

Regulile folosite de **Find** pentru a localiza un obiect sunt:

1. Cauta in memorie pentru existenta entitatii. Entitatile au fost incarcate din baza de date sau atasate la context.
2. Cauta in obiectele adaugate dar care nu au fost inca salvate in baza de date.
3. Cauta in baza de date pentru entitati care nu au fost inca incarcate in memorie.

Entitatile atasate la context pot sa nu fie salvate in baza de date in momentul respectiv.

Un exemplu cu metoda **Find()**

```
int id = 1; // valoare hard coded
using (var ctx = new ModelContainer())
{
    CustomerType ct = ctx.CustomerTypes.Find(id);
    if (ct != null)
    {
        ct.Description = "Valoare noua description";
        ctx.SaveChanges();
    }
}
```

Cereri pe date locale versus din baza de date – proprietatea Local

Datele sunt incarcate in memorie in urma executiei unei cereri. Datele astfel incarcate se numesc date locale. Datele locale pot fi accesate prin intermediul proprietatii **Local**.

Proprietatea **Local** returneaza **ObservableCollection<TEntity>**.

Cum obtinem datele din baza de date?

1. Definire cerere.
 - ✓ amanata la executie;
 - ✓ executie imediata – se aduc datele din baza de date.
2. Iterare cerere.

Pentru a evita iterarea unei cereri si sa incarcam totusi datele in memorie **DbContext** pune la dispozitie metoda **Load()**.

Metoda **Load()** este folosita in scrierea unei cereri. Vezi exemplul urmator.

Exemplu: se incarca toate inregistrarile din tabela CustomerTypes

```
using (var ctx = new Model1Container())
{
    var items = ctx.CustomerTypes.Load();
    // cod
}
```

Exemplu: se incarca in memorie numai anumite inregistrari

```
using (var ctx = new Model1Container())
{
    var query = from ct in ctx.CustomerTypes
                where (ct.CustomerTypeId > 2)
                select ct;
    query.Load();
    // cod
}
```

Proprietatea **Local** returneaza o colectie de obiecte pe care putem folosi LINQ To Objects.

*Exemplu – cererea **items** lucreaza pe date locale. Cererea **query** foloseste LINQ to Entities pe cand cererea **items** foloseste LINQ to Objects. In general baza de date nu este case sensitive cand compara siruri de caractere in timp ce LINQ to Objects este case sensitive. Operatori din LINQ to Objects nu pot fi aplicati pe LINQ to Entities. LINQ to Objects are operatorii **Last**, **Max**, etc. ceea ce nu gasim in LINQ to Entities.*

```
using (var ctx = new Model1Container())
{
    var query = from ct in ctx.CustomerTypes
                where (ct.CustomerTypeId > 2)
                select ct;
    query.Load();
    var items = from ct in ctx.CustomerTypes.Local
                orderby ct.CustomerTypeId
                select ct;
    foreach(CustomerType x in items)
    {
        //cod
    }
    // cod
}
```

ObservableCollection returnat de Local

Deoarece **Local** returneaza **ObservableCollection<TEntity>** exista posibilitatea notificarii cand obiecte sunt adaugate sau eliminate din colectie. **Local** va genera evenimentul **CollectionChanged** ori de cate ori continutul colectiei se modifica – se adauga noi obiecte sau se marcheaza pentru stergere anumite obiecte din colectie.

```
using (var context = new Model1Container())
{
    // add handler pentru tratare event CollectionChanged
    context.CustomerTypes.Local.CollectionChanged +=
        Local_CollectionChanged;

    CustomerType ct = new CustomerType();
    ct.Description = this.CustomerTypeDescription.Text.Trim();
    context.CustomerTypes.Add(ct);
    ...
}
```

si handler tratare eveniment:

```
void Local_CollectionChanged(object sender,
    System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
{
    IList ct = e.NewItems; // obiect nou adaugat la colectie
    MessageBox.Show("S-a adaugat CustomerType: " +
        ((CustomerType)ct[0]).Description);
}
```

Observatie

Proprietatile **NewItems** si **OldItems** din parametrii evenimentului returneaza o colectie de obiecte adaugate si/sau eliminate. In exemplul de mai sus am folosit **NewItems**.

Incarcare date relationate

Tipul **CustomerType** contine o colectie de tipuri **Customer**. Proprietatea *Customers* definita mai jos se numeste proprietate de navigare de tip colectie (entitate colectie).

```
public virtual ICollection<Customer> Customers { get; set; }
```

Cand aplicatia va cere informatii despre *CustomerType* se vor incarca in memorie si toate entitatile cu care relationeaza aceasta (in cazul de fata *Customer*), ceea ce in anumite situatii nu e de dorit pentru ca ar crea un impact negativ asupra performantei aplicatiei cat si asupra memoriei consumate.

Un alt exemplu, la un moment dat, un *Customer* cu Id = 10 poate avea numele “Alfa”, *CustomerType.Description* = “Preferat” si o colectie de 5 obiecte de tip *CustomerEmail*.

In mod implicit, EF incarca numai entitatile cerute de aplicatie. Acest lucru poate fi benefic sau nu functie de ceea ce dorim sa incarcam din baza de date la un moment dat.

In definitia entitatii *Customer* este definita proprietatea de navigare de tip colectie – entitate colectie:


```
// Entitate colectie
public virtual ICollection<CustomerEmail> CustomerEmails { get; set; }
```

si proprietatea de navigare scalara – entitate referinta:

```
// Entitate referinta
public virtual CustomerType CustomerType { get; set; }
```

Observatie

Metoda aleasa depinde de specificul problemei de rezolvat in acea parte de cod a aplicatiei.

In EF putem controla si optimiza numarul de cereri executate asupra bazei de date. EF implementeaza trei modele pentru incarcarea entitatilor relationate:

1. *lazily loading* – comportare implicita.
2. *eagerly loading* – incarcarea anumitor entitati relationate, partial sau in totalitate.
3. *explicitly loading* – renuntarea la incarcarea anumitor entitati relationate.

Lazy loading

Lazy loading este declarata implicit la crearea modelului.

Cerintele pe care trebuie sa le indeplineasca clasele POCO definite sunt:

- Clasele POCO trebuie sa fie **public** dar **nu sealed**.
- Proprietatile de navigare ce dorim sa foloseasca *lazy loading* trebuie sa fie marcate ca **virtual** in caz contrar *lazy loading* nu functioneaza.

Observatie

DbContext poate fi configurat pentru lazy loading sau nu folosind proprietatea **DbContext.Configuration.LazyLoadingEnabled**. Implicit este setata pe *true*.

Modelul *lazy loading* presupune crearea unui proxy dinamic de catre EF. Acest proxy dinamic va crea si incarca toate tipurile cu care relationeaza tipul dat. Din aceasta cauza proprietatile de navigare trebuie sa fie declarate virtual. Lazy loading este declarata implicit la construirea modelului.

Exemplu

```
using (var ctx = new Model1Container())
{
    var query = ctx.CustomerTypes.Take(5);
    foreach (var ct in query)
    {
        // pentru fiecare obiect din query se trimite o noua cerere
        // la baza de date pentru a aduce in memorie Customers
        foreach (var c in ct.Customers)
            Console.WriteLine("Description = {0}, Name customer = {1}",
                               ct.Description, c.Name);
    }
}
```

Utilizarea incorecta a modelului *lazy loading* poate duce la incarcarea serverului de baze de date prin trimiterea multor comenzi SQL de regasire a informatiilor. De exemplu daca vrem sa aducem informatii despre zece (10) *CustomerType* si apoi sa regasim inregistrarile din

Customer care sunt relationate cu fiecare *CustomerType* se vor executa 10 cereri asupra bazei de date, o cerere pentru a regasi cele 10 inregistrari si apoi pentru fiecare *CustomerType* sa regasim inregistrările din *Customer*. In total vor fi 11 cereri emise catre baza de date. In acest ultim scenariu este mai bine de folosit *eager loading*.

Eager loading

Eager loading permite EF sa incarce datele relationate intr-o singura cerere trimisa la baza de date.

Metoda folosita in acest caz este **Include()**. Exista doua proptotipuri pentru metoda **Include**. Un prototip are ca parametru un string iar celalalt prototip are ca parametru o expresie lambda.

Exemple

```
using (var ctx = new Model1Container())
{
    var query = ctx.CustomerTypes.Include("Customers").Take(5);
    foreach (var x in query)
    {
        foreach (var c in x.Customers)
            Console.WriteLine("Description = {0}, Name customer = {1}",
                               x.Description, c.Name);
    }
}
```

Observatie

Nu intotdeauna a trimite mai putine cereri asupra bazei de date inseamna ca e mai bine. Metoda **Include()** realizeaza in fapt un Join. O cerere poate sa contina mai multe apeluri ale metodei **Include()** si deci mai multe Join-uri la nivel de baza de date ceea ce poate avea ca efect diminuarea performantei aplicatiei. In acest caz e preferat sa se execute cereri simple si multiple asupra bazei de date.

Putem folosi **Include()** ca parte a unei cereri prin adaugarea acestei metode la **DbSet**-ul pe care executam cererea:

```
var query = from ct in context.CustomerTypes.Include(ct=>ct.Customers)
             where ct.Description == "Preferat"
             select ct;
```

Daca folosim forma C# a cererii atunci putem scrie:

```
var query = context.CustomerTypes
    .Include(ct => ct.Customers)
    .Where(ct => ct.Description == "Preferat");
```

Include() este definita ca o metoda extinsa pe **IQueryable<T>** si deci poate fi adaugata la o cerere in orice punct.

Exemplu

```
var query = from ct in context.CustomerTypes
             where ct.Description == "Preferat"
             select ct;
```

```
query.Include(ct=>ct.Customers);
```

Explicit loading

Ca si in cazul *lazy loading* incarcarea datelor relationate se face dupa ce datele principale au fost incarcate. Incarcarea acestor date nu se face automat. In cazul incarcarii explicite a datelor nu mai e nevoie ca proprietatile de navigare sa fie declarate **virtual**.

Metoda folosita pentru incarcarea explicita a datelor este **DbContext.Entry()** ce returneaza **DbEntityEntry**.

Instantele clasei **DbEntityEntry** furnizeaza acces la informatia despre entitatile definite in **DbContext**.

Dupa ce avem o intrare pentru o entitate data, putem folosi metodele **Collection()** si **Reference()** pentru a detalia informatia si operatiile pentru proprietatile de navigare.

Metoda **Load()** poate fi folosita pentru a incarca continutul *proprietatii de navigare de tip colectie*.

Exemplu

```
private void ExplicitLoading()
{
    // Metoda Collection
    using (var ctx = new Model1Container())
    {
        //Disable Lazy loading
        ctx.Configuration.LazyLoadingEnabled = false;

        var itemsCt = from ct in ctx.CustomerTypes
                      where ct.CustomerTypeId == 2
                      select ct;
        var single = itemsCt.Single();
        ctx.Entry(single).Collection(d => d.Customers).
            Load();
        foreach (var c in single.Customers)
            Console.WriteLine("Explicit loading. Customer name = {0}", c.Name);
    }
}
```

Incarcarea explicita a unei proprietati de navigare de tip referinta se face cu ajutorul metodei **Reference()** si este exemplificata in continuare. Proprietatea de navigare este de tip *Entity Reference*.

```
using (var ctx = new Model1Container())
{
    Console.WriteLine("Explicit loading. Reference method.");
    var items = from c in ctx.Customers
                where c.CustomerId >= 2
                select c;
    var single = items.First();
    ctx.Entry(single).Reference(d => d.CustomerType).
        Load();
}
```

```

Console.WriteLine("CustomerType = {0}; Customer Name =
{1}",single.CustomerType.Description, single.Name);
}

```

Alaturi de aceste doua metode **Collection()** si **Reference()** mai exista si metoda **Property()**. Metoda **Property()** lucreaza pe proprietati scalare. Se poate folosi proprietatea **IsModified** pentru a determina daca o proprietate scalara a fost sau nu modificata.

Metoda Collection() lucreaza pe proprietati de navigare de tip colectie, entitati colectie.

Metoda Reference() lucreaza pe proprietati de navigare de tip referinta, entitate referinta.

Se poate folosi proprietatea **IsLoaded** pentru a determina daca entitatea referinta a fost sau nu incarcata in memorie.

Proprietatea **IsLoaded** poate fi verificata pentru a determina daca continutul unei proprietati de navigare a fost incarcat in memorie. Proprietatea este valabila pentru cele trei modalitati de incarcare a proprietatii de navigare.

Interogare continut pentru o proprietate de navigare Collection

Filtrarea continutului unei proprietati de navigare poate fi facut astfel:

- dupa ce datele au fost incarcate in memorie se foloseste LINQ to Objects;
- aplicare de filtre pe cererile ce se trimit la baza de date.

Metoda **Query()** poate fi utilizata dupa ce am apelat metodele **Entry()** si **Collection()**.

Aplicare de filtre pe cererile ce se trimit la baza de date

Exemplu

```

context.Entry(customerType)
.Collection(d => d.Customers)
.Query()
.Where(l => l.Name.Contains("test"))
.Load();

```

Adaugare, modificare si stergere entitati

Stergere din baza de date fara a aduce datele in memorie, e ca si cum s-ar executa o comanda SQL Delete.

Metoda **Attach()** – presupune ca datele care se doresc a fi sterse nu sunt aduse in memorie.

Pentru a realiza acest lucru se foloseste metoda **Attach()** ca in exemplul urmator:

```

var context = new Model1Container();
var toDelete = new CustomerEmails(CustomerEmailId = 1);
context.CustomerEmails.Attach(toDelete);
context.CustomerEmails.Remove(toDelete);
context.SaveChanges();
context.Dispose();

```

Observatie

Daca entitatea exista in memorie, codul de mai sus va genera o exceptie de tip **InvalidOperationException**.

Metoda `DbContext.Database.ExecuteSqlCommand` permite scrierea directa a comenzii SQL, **Delete**.

Exemplu

```
context.Database.ExecuteSqlCommand(
    "DELETE FROM CustomerEmails WHERE CustomerEmailId = 1");
```

`ExecuteSqlCommand` poate fi folosita cu orice comanda din SQL, Insert, Update, Delete.

Stergere obiect cu date relationate

Daca stergem obiecte ce au date relationate, trebuie sa actualizam datele relationate pentru ca stergerea sa se efectueze. Actualizarea datelor relationate depinde de tipul relatiei: optionala sau nu (required). Relatia optionala presupune ca entitatea “copil” poate exista in baza de date fara a fi atribuita unei entitati “parinte”. Relatia obligatorie (required) presupune ca entitatea “copil” nu poate exista daca nu are o entitate “parinte” la care face referire (*foreign key* cu optiunea *delete in cascade* din baza de date).

Daca stergem o entitate care este parintele unei relatii, relatie optionala, relatia dintre parinte si copil poate fi stearsa de asemenea. Foreign key va fi setata pe null in baza de date. EF va sterge in mod automat relatia pentru entitatea copil cu conditia ca entitatea copil sa fi fost incarcata in memorie din baza de date.

In cazul cand exista relatii obligatorii (required) intre entitatea parinte si cea copil operatia de stergere a entitatii parinte va realiza si stergerea entitatilor copil.

Entity Framework si aplicatii N-tier

Scenariul principal consta in a lucra cu entitati in instante diferite ale contextului si de asemenea de a lucra cu entitati deconectate.

EF jurnalizeaza modificarile efectuate asupra obiectelor in doua moduri:

- *snapshot change tracking* sau
- *change tracking proxies*.

Snapshot change tracking

Codul folosit pana acum foloseste *snapshot change tracking*.

Clasele din cadrul modelului sunt POCO si nu contin logica necesara pentru a notifica EF cand valoarea unei proprietati se schimba. Cand EF are nevoie sa stie ce schimbari au fost facute asupra obiectelor din memorie, va scana fiecare obiect si-i va compara valoarea cu cea existenta in snapshot. Acest proces de scanare se realizeaza prin apelul metodei **DetectChanges** din **ChangeTracker**.

Change tracking proxies

Mecanismul dat de *change tracking proxies*, permite EF sa fie notificat de schimbarile pe care le-am facut asupra proprietatilor obiectelor. *Change tracking proxies* creaza proxy in mod dinamic pentru obiectele create din **DbContext** si notifica contextul asupra modificarilor. Acest mecanism cere sa respectam urmatoarele reguli, adica sa structuram clasele din context astfel incat:

- Clasele sa fie *publice*.
- Clasele sa *nu* fie *sealed*.
- Proprietatile sa fie *virtuale*. Getter-i si setter-i sa fie publici.
- Proprietatile de navigare de tip colectie sa fie de tipul **ICollection<T>**.

Modificarile in clase trebuiesc facute manual.

Clasa *CustomerType* ar trebuie sa aiba urmatoarea structura in acest caz. Modificarile sunt evidentiata cu culoarea rosie.

```
public partial class CustomerType
{
    public CustomerType()
    {
        this.Customers = new HashSet<Customer>();
    }
    // Proprietati scalare
    public virtual int CustomerTypeId { get; set; }
    public virtual string Description { get; set; }
    // Entitate colectie
    public virtual ICollection<Customer> Customers { get; set; }
}
```

Cand Entity Framework creaza proxy dinamic pentru monitorizarea modificarilor (change tracking), EF va implementa interfata **IEntityChangeTracker**.

Urmatorul cod ne arata daca *customer* este proxy sau nu.


```
private static void TestForChangeTrackingProxy()
{
    using (var context = new Modell())
    {
        var customer = context.CustomerTypes.First();
        var isProxy = customer is IEntityWithChangeTracker;
        Console.WriteLine("customertype este un proxy: {0}", isProxy);
    }
}
```

In acest caz se va afisa **true** daca *CustomerType* respecta regulile enuntate mai sus.





Interfata **IEntityChangeTracker** este definita astfel (spatiul de nume - System.Data.Entity.Core.Objects.DataClasses)

public interface IEntityChangeTracker

Properties

	Name	Description
	EntityState	Gets current state of a tracked object.

Methods

	Name	Description
	EntityComplexMemberChanged(String, Object, String)	Notifies the change tracker that a property of a complex type has changed.
	EntityComplexMemberChanging(String, Object, String)	Notifies the change tracker of a pending change to a complex property.
	EntityMemberChanged(String)	Notifies the change tracker that a property of an entity type has changed.
	EntityMemberChanging(String)	Notifies the change tracker of a pending change to a property of an entity type.

Observatie

[EntityObject](#) si [ComplexObject](#) sunt clasele de baza pentru tipurile entitate si tipurile complexe generate de utilitarul Entity Data Models. Ambele clase folosesc IEntityChangeTracker pentru a raporta modificarile proprietatii.

Cand se detecteaza in mod automat schimbarile efectuate?

Exista evenimente care sunt generate automat si apeleaza metoda **DetectChanges**. Metodele care fac acest lucru sunt:

- **DbSet.Add**
- **DbSet.Find**
- **DbSet.Remove**
- **DbSet.Local**
- **DbContext.SaveChanges**
- **Rulare cerere LINQ asupra unui DbSet**
- **DbSet.Attach**
- **DbContext.GetValidationErrors**
- **DbContext.Entry**
- **DbChangeTracker.Entries**

Detectarea modificarilor poate fi setata pe *on* sau *off* astfel:

```
context.Configuration.AutoDetectChangesEnabled = false; // true = on
```

Determinarea modificării unui obiect se face prin scanarea colecției ce conține acel obiect și testarea proprietății **State**.

```
using (var context = new Model1Container())
{
    context.Configuration.AutoDetectChangesEnabled = false;
    var customer = (from d in context.Customers
                    where d.Name == "Alfa"
                    select d).Single();
    customer.Name = "Alfa Beta";
    Console.WriteLine("Inainte de DetectChanges: {0}",
        context.Entry(customer).State);
    // Apel pentru detectarea modificarilor
    context.ChangeTracker.DetectChanges();
    Console.WriteLine("Dupa DetectChanges: {0}",
        context.Entry(customer).State);
}
```

Deoarece *AutoDetectChangesEnabled = false*; (setat pe off) se va afișa:

Inainte de DetectChanges: Unchanged
Dupa DetectChanges: Modified

EF presupune implicit ca **DetectChanges()** este apelată înaintea oricărui apel API din **DbContext**. Acest lucru implică apelul metodei **DetectChanges()** înainte de a executa orice cerere asupra bazei de date. Omiterea apelului metodei **DetectChanges()** poate produce efecte colaterale nedorite și uneori dificil de depistat bug-uri în cod.

Observatie

Dacă efectuez o serie de apeluri API **DbContext** fără a modifica obiectele atunci e corect să dezactivăm apelul metodei **DetectChanges()**.

DetectChanges() și relațiile între obiecte

Metoda **DetectChanges()** este responsabilă și pentru a sincroniza relațiile existente între entități, *entity reference* sau *entity collection*. Acest lucru este vizibil atunci când folosim *data binding* în aplicații GUI și facem modificări asupra proprietăților de navigare. Modificările efectuate trebuie să se reflecte în interfața GUI a aplicației.

În exemplul de cod ce urmează se apelează metoda **DetectChanges()** și astfel are loc o sincronizare a relațiilor dintre obiecte.

În baza de date avem situația următoare: *Customer* cu *CustomerId = 1* este în relație cu *CustomerType* ce are *CustomerType.Description = "Normal"*. Dorim ca acest *Customer* să-l punem în relație cu *CustomerType* ce are *CustomerType.Description = "Preferat"*, deci schimbăm asocierea existentă.

Codul trebuie să execute următoarele acțiuni:

- (1) să aducă în memorie *Customer* cu *CustomerId = 1*.
- (2) să aducă în memorie relația existentă între acest obiect și *CustomerType*.
- (3) să aducă în memorie *CustomerType* la care vreau să asociez obiectul *Customer*.

- (4) sa actualizeze proprietatea de navigare de tip colectie, *Customers*, din *CustomerType* apoi sa afiseze in continuare pentru a verifica ca EF nu a “vazut” modificarea relatiei.
- (5) sa apelez **DetectChanges()** pentru a forta EF sa-si reactualizeze informatiile despre modificarile efectuate si apoi sa afisez rezultatele.

In acest cod nu facem salvarile in baza de date.

```
using (var context = new Model1Container())
{
    context.Configuration.AutoDetectChangesEnabled = false;

    // (1)
    var customer = (from d in context.Customers
                    where d.CustomerId == 1
                    select d).Single();

    // (2)
    context.Entry(customer)
        .Reference(1 => 1.CustomerType)
        .Load();

    // (3)
    var customerType = (from l in context.CustomerTypes
                        where l.Description == "Preferat"
                        select l).Single();

    // (4)
    customerType.Customers.Add(customer);
    Console.WriteLine("Inainte de DetectChanges: {0}",
        customer.CustomerType.Description);

    // (5)
    context.ChangeTracker.DetectChanges();
    Console.WriteLine("Dupa DetectChanges: {0}",
        customer.CustomerType.Description); }


```

Rezultatul este

```
Inainte de DetectChanges: Normal
Dupa DetectChanges: Preferat
```

Observatie

Nu e necesar sa facem disable **DetectChanges** cand folosim change tracking proxies. EF va sti cand sa scaneze sau nu o colectie pentru a detecta modificarile efectuate asupra obiectelor.

EF va crea automat proxies pentru rezultatele unei cereri pe care o executam. Daca cream obiectele folosind constructorul claselor POCO, atunci acest proxy nu va fi creat. Pentru a crea proxy trebuie sa apelam metoda **DbSet.Create()** ce va returna o instanta a tipului si in acelasi timp si proxy-ul necesar.

EF lucreaza si cu tipuri cu proxy si cu tipuri fara proxy. Pentru a elimina ambiguitatea se recomanda crearea instantelor tipurilor prin folosirea metodei **DbSet.Create()**. Nu se adauga instanta la colectie (DbSet). Trebuie folosita apoi metoda **Add**. **DbSet.Create()** are sens atunci cand atasam o entitate existenta la context si apoi putem obtine datele relationate de aceasta entitate.

Exemplu

```
public class Customer
{
    public virtual int CustomerId { get; set; }
    public virtual ICollection<Order> Orders { get; set; }
}

public class Order
{
    public virtual int OrderId { get; set; }
    public string Name { get; set; }
}
```

Urmatorul cod va returna datele relationate

```
using (var context = new ModelCustomerContext())
{
    var customer = context.Customers.Create();
    customer.CustomerId = 1; // pp ca exista in bd
    context.Customers.Attach(customer);

    foreach (var order in customer.Orders)
    {
        var name = order.Name;
        // ...
    }
}
```

Exemplu

Se lucreaza cu instante ale tipului ce au creat sau nu proxy. Nu e de dorit acest lucru, adica de a lucra mixt cu instante ce au sau nu au proxy. EF recomanda apel metoda **Create()** in majoritatea situatiilor.

```
using (var context = new ModellContainer())
{
    var nonProxy = new CustomerType();
    nonProxy.Description = "Non-proxy Description";
    nonProxy.Customers = new List<Customer>();

    var proxy = context.CustomerTypes.Create();
    proxy.Description = "Proxy Description";
    context.CustomerTypes.Add(proxy);
    context.CustomerTypes.Add(nonProxy);

    var cust = (from l in context.Customers
                where l.Name == "Alfa srl"
                select l).Single();
    context.Entry(cust)
        .Reference(l => l.CustomerType)
        .Load();

    Console.WriteLine("Inainte de modificari: {0}",
        cust.CustomerType.Description);

    nonProxy.CustomerTypes.Add(cust);
    Console.WriteLine("Adaugat la non-proxy: {0}",
        cust.CustomerType.Description);
}
```

```
proxy.Customers.Add(cust);
Console.WriteLine("Adaugat la proxy: {0}",
    cust.CustomerType.Description);
}
```

Observatie

Pentru tipurile derivate exista metoda **Create<TEntity>**, cu generice.

Dezactivarea directa pentru *change tracking* poate fi facuta la executia unei cereri in mod direct. In scena intra metoda **AsNoTracking()**, ca in exemplul urmator.

```
using (var context = new Model1Container())
{
    foreach (var c in context.Customers.AsNoTracking())
    {
        Console.WriteLine(c.Name);
    }
}
```

Observatie

Daca modificam obiecte din cererea de mai sus si apelam **SaveChanges()**, modificarile nu vor fi salvate in baza de date.

N-Tier – WCF Data Service

Entitatile care nu sunt jurnalizate (contextul nu are informatii daca entitatile s-au modificat sau nu) de catre context se numesc entitati deconectate.

Aplicatiile single-tier presupun o interfata client si un nivel de acces la baza de date, toate acestea ruland in cadrul aceluiasi proces.

Operatiile pe entitati deconectate sunt comune aplicatiilor N-tier, aplicatii ce presupun aducerea de pe server a anumitor informatii, prin retea, pe masina clientului; clientul executa operatii asupra acestor date si apoi salveaza datele pe server.

O prima problema care apare aici este aceea ca se folosesc contexte diferite la citirea datelor respectiv salvarea acestora in baza de date. O alta problema care trebuie avuta in vedere aici este cea legata de serializarea datelor.

Solutia propusa de Microsoft pentru EF este folosirea WCF Data Service.

Alte solutii posibile sunt discutate in continuare. Fiecare prezinta avantaje si dezavantaje privitoare la modul de scriere al codului pe partea de server si pe partea clientului. Alegerea unei solutii depinde de tipul aplicatiei si de proiectantul acesteia.

Operatii explicite pe partea de server

In acest scenariu clientul identifica exact schimbarile ce trebuiesc facute. Acest lucru inseamna ca pe partea clientului vom scrie metode care sa lucreze cu o anumita entitate. De exemplu, pentru *CustomerType*, vom scrie o metoda pentru adaugare, una pentru

modificare, alta pentru stergere din baza de date. Aceste metode vor fi punctuale pentru o anumita entitate. Avand in vedere relatiile dintre entitati si numarul acestora intr-un anumit context constatam ca logica pe partea clientului devine complexa iar serverul va expune mult mai multe operatii. Operatiile individuale conduc la o cantitate mai mare de cod, dar codul este mai simplu de scris, testat si depanat.

Reluare schimbari pe partea de server

O alta posibilitate este de a avea pe server o operatie generalizata ce accepta un graf de entitati si lasa ca EF sa cunoasca starea fiecărei entitati din acest graf. Acest lucru presupune iterarea de catre EF a starii entitatilor pentru a determina modificarile survenite pe partea clientului. Acest proces este cunoscut sub de numele de *“painting the state”*, altfel spus serverul determina starea fiecărei entitati din graf si va proceda ca atare la executia metodei **SaveChanges()**. In general implementarea unei asemenea logici presupune crearea unei *clase de baza* sau a unei *interfete*. Entitatile trebuie sa fie derivate din acestea. Acestea expun informatii ce permit codului de pe partea de server sa execute actiuni conform starii fiecărei entitati.

Inainte de a analiza arhitectura aplicatiei n-tier cu EF este necesar sa intelegem rolul proprietatii **State** din **DbContext**.

DbContext – Proprietatea State – enum EntityState

Pentru exemplificare vom considera o singura entitate.

Metodele pe care le vom implementa pot fi expuse de un serviciu web. In exemplele prezentate in continuare nu folosim servicii web, e numai o simulare.

In loc de a serializa entitatile, vom utiliza un context temporar pentru a aduce datele si apoi simulam modificarile facute pe partea de client. Vom transfera aceste obiecte in operatiile de pe un pseudo server, operatii ce vor utiliza un nou context ce nu va sti nimic despre instantele entitatilor, si vom salva modificarile in baza de date. EF are o lista a starilor pe care *change tracker* le foloseste pentru a inregistra starea fiecărei entitati. Starile pe care le foloseste EF sunt (vezi enumerarea **EntityState**):

Added

Entitatea este evidentiata (tracking) de context dar nu exista in baza de date. **SaveChanges()** va genera o instructiune **INSERT**.

Unchanged

Entitatea exista deja in baza de date si nu a fost modificata de cand a fost adusa din baza de date. **SaveChanges()** nu proceseaza entitatea.

Modified

Entitatea exista deja in baza de date si a fost modificata in memorie. **SaveChanges()** va genera o instructiune **UPDATE**. Se mentine si o lista cu coloanle ce vor fi modificate.

Deleted

Entitatea exista in baza de date si a fost marcata pentru stergere. **SaveChanges()** va genera o instructiune **DELETE**.

Detached

Entitatea nu este evidentiata de context.

Utilizare EntityState

In continuare voi prezenta exemple de cod pentru fiecare stare descrisa mai sus.

Vor fi prezentate doua versiuni de cod pentru o aceeasi actiune. O versiune (V1) va lasa EF sa determine ce obiecte au fost modificate si sa emita codul corect la **SaveChanges()**, iar cealalta versiune (V2) va marca exact obiectul care se modifica urmand ca **SaveChanges()** sa emita codul corect.

EntityState.Added

V1: Nu se foloseste proprietatea State

```
private static void AddCustomerTypeAdd(CustomerType ct)
{
    using (var context = new ModellContainer())
    {
        context.CustomerTypes.Add(ct);
        context.SaveChanges();
    }
}
```

V2: Utilizare proprietate State

```
private static void AddCustomerTypeEntry(CustomerType ct)
{
    using (var context = new ModellContainer())
    {
        context.Entry(ct).State = EntityState.Added;
        context.SaveChanges();
    }
}
```

Observatie

Diferenta majora intre cele doua versiuni ale metodei de inserare in tabela *CustomerTypes* este aceea ca in varianta V1, EF va inspecta garful de obiecte pentru a determina ce anume s-a modificat iar in V2 indicam exact entitatea care se schimba.

EntityState.Unchanged

```
private static void TestAttachCustomerType()
{
    CustomerType ct;
    using (var context = new ModellContainer())
    {
        ct = (from x in context.CustomerTypes
```

```

        where d.Description == "Preferat"
        select x).Single();
    }
    AttachCustomerType(ct);
}

```

v1: Nu se foloseste proprietatea State

```

private static void AttachCustomerType(CustomerType ct)
{
    using (var context = new ModellContainer())
    {
        context.CustomerTypes.Attach(ct);
        context.SaveChanges();
    }
}

```

v2: Utilizare proprietate State

```

private static void AttachCustomerType(CustomerType ct)
{
    using (var context = new ModellContainer())
    {
        context.Entry(ct).State = EntityState.Unchanged;
        context.SaveChanges();
    }
}

```

Observatie

Obiectul *CustomerType* este obtinut intr-un context si utilizat in alt context. Cele doua versiuni fac acelasi lucru.

EntityState.Modified

In acest caz nu exista versiunea (V1), adica sa gasesc datele intr-un cotext si sa le modific in alt context fara a folosi **EntityState.Modified**.

```

private static void TestUpdateCustomerType()
{
    CustomerType ct;
    using (var context = new ModellContainer())
    {
        ct = (from d in context.CustomerTypes
              where d.Description == "Preferat"
              select d).Single();
    }
    ct.Description = "Normal";
    UpdateCustomerType(ct);
}

```

v2: Utilizare proprietate State

```

private static void UpdateCustomerType(CustomerType ct)
{
    using (var context = new ModellContainer())
    {
        context.Entry(ct).State = EntityState.Modified;
        context.SaveChanges();
    }
}

```

```
}
```

EntityState.Deleted

DbSet.Remove() este folosită la a marca pentru ștergere o entitate. Dacă entitatea pe care se apelează **DbSet.Remove()** este în starea **Added** atunci aceasta va trece în starea **Detached**. Următorul exemplu exemplifică acest lucru:

```
private void AddAndDeleteSameEntity()
{
    Console.WriteLine("Add and Delete the same entity in the same
                        context...");
    using (var context = new ModellContainer())
    {
        CustomerType ct = new CustomerType();
        ct.Description = "Add and Delete";
        context.CustomerTypes.Add(ct);
        Console.WriteLine("State dupa Add: " + context.Entry(ct).State);
        context.CustomerTypes.Remove(ct);
        Console.WriteLine("State dupa Remove: " + context.Entry(ct).State);
    }
}
```

Rezultatul executiei acestei metode:

Add and Delete the same entity in the same context...

State dupa Add: Added

State dupa Remove: Detached

Din exemplul de mai sus observăm că entitatea pe care dorim să o ștergem trebuie să fie încărcată în context pentru ca EF să poată determina dacă entitatea este una nouă sau în care va fi trecută în starea **Detached** și va fi ignorată de EF la **SaveChanges()**, sau una adusă din baza de date în care **SaveChanges()** va genera instrucțiunea **DELETE**.

O entitate *deconectată* nu poate folosi metoda **Remove()**. Entitatea trebuie întâi atasată la context și apoi să folosim metoda **Remove()**. Obținem entitatea într-un context și o folosim în alt context.

```
private static void TestDeleteCustomerType()
{
    CustomerType ct;
    using (var context = new ModellContainer())
    {
        ct = (from d in context.CustomerTypes
              where d.Description == "Preferat"
              select d).Single();
    }
    DeleteCustomerType(ct);
}
```

V1: Nu se folosește proprietatea *State*

```
private static void DeleteCustomerType(CustomerType ct)
{
    using (var context = new ModellContainer())
    {
        context.CustomerTypes.Attach(ct);
        context.CustomerTypes.Remove(ct);
    }
}
```

```

        context.SaveChanges();
    }
}

v2: Utilizare proprietate State
private static void DeleteCustomerType(CustomerType ct)
{
    using (var context = new ModellContainer())
    {
        context.Entry(ct).State = EntityState.Deleted;
        context.SaveChanges();
    }
}

```

Stergere folosind “stub entity”

Aceasta metoda presupune folosirea cheii primare pentru a identifica entitatea ce va fi stearsa. Instructiunea DELETE va folosi aceasta cheie primara. Nu mai sunt aduse date in context.

```

private static void DeleteCustomerType(int customerId)
{
    using (var context = new ModellContainer())
    {
        var ct = new CustomerType { CustomerTypeId = customerId };
        context.Entry(ct).State = EntityState.Deleted;
        context.SaveChanges();
    }
}

```

Lucrul cu relatii - cu si fara chei straine (FK)

Relatiile ce includ o proprietate cheie straina se numesc “asociatii cheie straina” – in unele documentatii notata **PK->FK**.

Daca marcam o entitate ca “**Added**”, aceasta va folosi valoarea curenta atribuita proprietatii cheie straina. In cazul cand se fac modificari, cheia straina va fi modificata la fel ca celelalte proprietati.

Reluam un exemplu (putin modificat) prezentat mai inainte in acest curs:

```

private void TestChangeFKCustomers()
{
    //CustomerType customerType;
    using (var context = new ModellContainer())
    {
        // (1) : Incarc in context un Customer
        var customer = (from d in context.Customers
                        where d.CustomerId == 1
                        select d).Single();

        // (2) : Incarc in context parintele relatiei,
        //         adica catre ce CustomerType
        //         puncteaza cheia straina din Customers
        context.Entry(customer)
            .Reference(1 => 1.CustomerType)
            .Load();

        // (3) : Incarc in context un CustomerType.
        //         Vrem sa modificam relatia intre Customer din (1)
        //         si CustomerType din (3)
        var customerTypeInNewRelation =
    
```



```

        (from l in context.CustomerTypes
         where l.Description == "Preferat"
         select l).Single();

// (cw: 1) : Situatia existenta inainte de modificare relatie
Console.WriteLine("Inainte de modificare relatie PK-FK");
Console.WriteLine("Customer. CustomerId = {0}, Name = {1},
    Apartine de {2}", customer.CustomerId, customer.Name,
    customer.CustomerType.Description);

// (4) : Schimbam relatia intre Customers si CustomerTypes
customer.CustomerType = customerTypeInNewRelation;
// (4.1) : Modificam valoarea unei proprietati scalare
//         (Name in acest caz)
customer.Name = "Name modificat";

context.Entry(customer).State = EntityState.Modified;

// (cw: 2) : Situatia dupa modificarea relatiei in cadrul contextului
Console.WriteLine("Inainte de salvare in baza de date");
Console.WriteLine("Customer. CustomerId = {0}, Name = {1},
    Apartine de {2}", customer.CustomerId, customer.Name,
    customer.CustomerType.Description);

// (5) : Salvare modificare relatie in baza de date
context.SaveChanges();
}
}

```

Rezultat executie cod:

```

Inainte de modificare relatie PK-FK
Customer. CustomerId = 1, Name = Alfa srl, Apartine de Normal
Inainte de salvare in baza de date
Customer. CustomerId = 1, Name = Name modificat, Apartine de Preferat

```

Utilizare proprietati de navigare pentru a defini relatii

Foreign Keys sunt vitale in scenariile N-Tier si din acest motiv EF nu expune metodele pentru schimbarea starii relatiilor independente in *API DbContext*. Metodele se gasesc in *APIObjectContext*. Reamintesc cu aceasta ocazie folosirea interfetei **IObjContextAdapter**.

```

var context = new ModelContainer()
var contextObject = ((IObjContextAdapter)context).ObjectContext;

```

Contextul monitorizeaza schimbarile numai pentru proprietatile scalare.

Cand schimbam o proprietate “cheie straina”, cum ar fi *Customer.CustomerTypeCustomerId*, contextul este informat despre acest lucru. Cand schimbam o proprietate de navigare, nu exista nimic de evidentiat chiar daca schimbam starea entitatii in *Modified*. Cand folosim asocieri independente, contextul mentine o evidenta a acestor autorelatii. El are un obiect ce contine cheile a doua instante relationate si acest obiect va fi folosit de context pentru a retine modificarile si actualizarile pe care trebuie sa le faca in baza de date.

Cand entitatea nu e conectata la context, contextul nu poate modifica obiectele ce sunt in relatie. Cand reconectam entitatile la un context, trebuie sa modificam manual obiectele relationate din cadrul contextului.

Sa urmarim exemplul in care sunt modificate starile relatiilor.

```
private static void UpdateCustomer(Customer customer,
    CustomerType previousCustomerType)
{
    using (var context = new ModellContainer())
    {
        context.Entry(customer).State = EntityState.Modified;
        context.Entry(customer.CustomerType).State = EntityState.Unchanged;
        if (customer.CustomerType.CustomerTypeId !=
            previousCustomerType.CustomerTypeId)
        {
            context.Entry(previousCustomerType).State = EntityState.Unchanged;
           ObjectContext objectContext =
                ((IOBJECTContextAdapter) context).ObjectContext

            objectContext.ObjectStateManager
                .ChangeRelationshipState(customer,
                    customer.CustomerType,
                    1 => 1.CustomerType,
                    EntityState.Added);

            objectContext.ObjectContext
                .ObjectStateManager
                .ChangeRelationshipState(customer,
                    previousCustomerType, 1 => 1.CustomerType,
                    EntityState.Deleted);
        }
        context.SaveChanges();
    }
}
```

Pattern-uri pentru lucrul cu “object graph” intr-un context

Metoda	Rezultat	Observatie
<i>Add Root</i>	Fiecare entitate din graf va fi marcata cu starea Added.	E posibil sa salvam date care deja exista.
<i>Attach Root</i>	Fiecare entitate din graf va fi marcata cu starea Unchanged.	Noile entitati nu vor fi inserate in baza de date.
<i>Add</i> sau <i>Attach Root</i> si apoi obtine starea din graf (paint state)	Entitatile vor avea starea corecta cand procesul de determinare al starii s-a incheiat.	E de preferat sa folosim Add in detrimentul lui Attach care poate crea conflicte cu cheile existente.

Setarea proprietatii State pentru entitatile dintr-un graf

Explicam folosirea acestei proprietati pe baza unui exemplu.
Ceea ce trebuie mentionat de la inceput in acest exemplu este urmatorul lucru:

- Cheile primare din tabele sunt de tip int si cu autoincrementare.
- Cand EF adauga o entitate noua la context, cheia primara a acelei entitati este zero. Codul de mai jos testeaza cu zero valoarea cheii primare si determinam daca am modificat sau nu entitatea.

```
private static void SaveCustomerTypeAndCustomers (
    CustomerType ct,
    List<Customer> deletedCustomers)
{
    // Se pp ca parametrii sunt furnizati corect
    using (var context = new ModellContainer())
    {
        context.CustomerTypes.Add(ct);
        if (ct.CustomerTypeId > 0)
        {
            context.Entry(ct).State = EntityState.Modified;
        }
        foreach (var customer in ct.Customers)
        {
            if (customer.CustomerId > 0)
            {
                context.Entry(customer).State = EntityState.Modified;
            }
        }
        foreach (var customer in deletedCustomers)
        {
            context.Entry(customer).State = EntityState.Deleted;
        }
        context.SaveChanges();
    }
}
```

iar metoda ce o apeleaza poate avea urmatorul cod:

```
private static void TestSaveCustomerTypeAndCustomers ()
{
    CustomerType ct;
    using (var context = new ModellContainer())
    {
        ct = (from d in context.CustomerTypes.Include(d => d.Customers)
              where d.Description == "Preferat"
              select d).Single();
    }
    ct.Description = "Mai preferat"; // modificare proprietate
    ct.Customers.Add(new Customer { Name = "New customer"});

    var firstCustomer = ct.Customers.ElementAt(0);
    firstCustomer.Name = "Alfa Nou srl ";
    var secondCustomer = ct.Customers.ElementAt(1);
    var deletedCustomers = new List<Customer>();
    ct.Customers.Remove(secondCustomer);
    deletedCustomers.Add(secondCustomer);
    SaveCustomerTypeAndCustomers(ct, deletedCustomers);
}
```

Construire model generic pentru gestionare proprietate State

Se pot folosi clase de baza sau interfete.

In situatia cand se folosesc clase de baza, pentru fiecare clasa POCO din model trebuie sa implementam o clasa de baza cu proprietati corespunzatoare.

Scenariul cu interfete

Construim o interfata ce contine valoarea proprietatii **State**. Toate clasele din model trebuie sa implementeze aceasta interfata.

Cand datele sunt incarcate din baza de date, **ObjectContext** genereaza evenimentul **ObjectMaterialized**. Ideea este de a scrie o metoda pentru acest eveniment, iar in metoda vom seta proprietatea **State** pe valoarea *Unchanged*. Definim interfata urmatoare:

```
public interface IObjectWithState
{
    State State { get; set; }
}

public enum State
{
    Added,
    Unchanged,
    Modified,
    Deleted
}
```

Clasele din model trebuie sa fie derivate din *IObjectWithState*.

```
public class CustomerType: IObjectWithState
{
    public State State { get; set; }
    // ...
}
// urmeaza si celelalte clase
```

In ctor *Model1Container* subscriere la eveniment

```
public Model1Container()
{
    ((IOBJECTContextAdapter)this).ObjectContext
    .ObjectMaterialized += (sender, args) =>
    {
        var entity = args.Entity as IObjectWithState;
        if (entity != null)
        {
            entity.State = State.Unchanged;
        }
    };
    // cod din ctor
}
```

În acest moment avem nevoie de o metoda ce foloseste aceste informatii pentru a lua un graf deconectat și a aplica modificările efectuate pe partea clientului la un context prin setarea corectă a stării fiecărei entități din graf. Metoda poate avea urmatorul cod:

```
public static EntityState ConvertState(State state)
{
    switch (state)
    {
        case State.Added:
            return EntityState.Added;
        case State.Modified:
            return EntityState.Modified;
        case State.Deleted:
            return EntityState.Deleted;
        default:
            return EntityState.Unchanged;
    }
}

public static void SaveCustomerTypeGraph(CustomerType customerType)
{
    using (var context = new ModellContainer())
    {
        context.CustomerTypes.Add(customerType);
        foreach (var entry in context.ChangeTracker
            .Entries<IObjectWithState>())
        {
            IObjectWithState stateInfo = entry.Entity;
            entry.State = ConvertState(stateInfo.State);
        }
        context.SaveChanges();
    }
}
```

Observatie

Deoarece toate clasele POCO din model implementează interfața *IObjectWithState*, putem folosi metoda **Entries** în forma generică.

Următorul cod testează metoda scrisă mai sus. Codul marcat arată acțiunile pe care dorim să le efectuăm asupra bazei de date.

```
private static void TestSaveCustomerTypeGraph()
{
    CustomerType ct;
    using (var context = new ModellContainer())
    {
        ct = (from d in context.CustomerTypes.Include(d => d.Customers)
            where d.Description == "Preferat"
            select d).Single();
        // context distrusts
        ct.Description = "Super preferat";
        ct.State = State.Modified;

        var firstCustomer = ct.Customers.First();
        firstCustomer.Name = "Alfa Omega srl";
        firstCustomer.State = State.Modified;

        var secondCustomer = ct.Customers.Last();
```

```
secondCustomer.State = State.Deleted;

ct.Customers.Add(new Customer
{
    Name = "EON srl",
    State = State.Added
});
SaveCustomerTypeGraph(ct);
}
```

Generalizare

Metoda `SaveCustomerTypeGraph` poate fi generalizata astfel:

```
private static void ApplyChanges<TEntity>(TEntity root)
    where TEntity : class, IObjectWithState
{
    using (var context = new ModellContainer())
    {
        context.Set<TEntity>().Add(root);
        // cazul cand o clasa din model nu implementeaza
        // interfata IObjectWithState
        CheckForEntitiesWithoutStateInterface(context);

        foreach (var entry in context.ChangeTracker
            .Entries<IObjectWithState>())
        {
            IObjectWithState stateInfo = entry.Entity;
            entry.State = ConvertState(stateInfo.State);
        }
        context.SaveChanges();
    }
}
```

Daca clasele din model nu implementeaza interfata *IObjectWithState* atunci se genereaza o exceptie pentru entitatile din graful dat si prelucrarea se termina.

```
private static void CheckForEntitiesWithoutStateInterface(
    ModellContainer context)
{
    var entitiesWithoutState = from e in context.ChangeTracker.Entries()
        where !(e.Entity is IObjectWithState)
        select e;

    if (entitiesWithoutState.Any())
    {
        throw new NotSupportedException(
            "Toate entitatile trebuie sa implementeze IObjectWithState");
    }
}
```

Scenariul cu clasa de baza abstracta

Se considera o clasa de baza abstracta ce va monitoriza modificarile facute asupra entitatilor. In acest caz se impune ca clasele POCO sa fie derivate din aceasta clasa de baza. *State* nu este memorat in baza de date. Crearea propriului sistem de evidentiare al modificarilor ne da posibilitatea de a renunta la anumite dependente pe care le cere EF. In

fișierul ce conține contextul (clasa derivată din `DbContext`) vom scrie cod astfel încât EF să nu mapeze proprietatea `State` la tabele din baza de date. Codul va fi scris în metoda **`OnModelCreating()`**.

Structura clasei de bază este furnizată în continuare. Aceasta folosește un enum `State` (același ca la scenariul cu interfețe):

```
public abstract class BaseEntity
{
    protected BaseEntity()
    {
        State = State.Unchanged;
    }
    public State State { get; set; }
}
public enum State
{
    Added,
    Unchanged,
    Modified,
    Deleted
}
```

Fiecare clasă POCO va fi derivată din `BaseEntity`. Exemplu:

```
public partial class Customer: BaseEntity
{
    // nu se adauga nimic in codul existent
}
```

În clasa ce are definit contextul, scriem cod în metoda **`OnModelCreating()`** pentru a “spune” EF că proprietatea `State` nu este memorată în baza de date. Această proprietate este folosită intern pentru a gestiona starea entităților deconectate, în cadrul a diverse servicii.

```
public class Model1Container : DbContext
{ // cod lipsa ...
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Types<BaseEntity>().Configure(x => x.Ignore(y => y.State));
    }
}
```

Ca și la scenariul cu interfețe, și aici avem nevoie de o metodă ce va atașa starea recunoscută de EF, deci va face conversia pentru `State` din entitate la `State` din EF.

Această metodă poate fi definită în cadrul unei clase statice, de exemplu `EntityConvertState`:

```
public static EntityState ConvertState(State state)
{
    switch (state)
    {
        case State.Added:
            return EntityState.Added;
        case State.Modified:
            return EntityState.Modified;
    }
}
```

```

        return EntityState.Modified;
    case State.Deleted:
        return EntityState.Deleted;
    default:
        return EntityState.Unchanged;
    }
}

```

În cadrul metodelor ce fac update trebuie să apelăm această metodă, ca în exemplul următor:

```

public bool UpdateCustomer(Customer customer)
{
    using (var context = new Model1Container())
    {
        // Adaug un obiect graf la context, starea fiind "Added".
        // Adaugarea parintelui la context are ca efect atasarea
        // întregului graf (parinte și entități copii) la context
        // setarea stării în "Added" pentru toate entitățile.

        context.Customers.Add(customer);
        foreach (var entry in context.ChangeTracker.Entries<BaseEntity>())
        {
            entry.State = EntityConvertState.ConvertState(entry.Entity.State);
            if (entry.State == EntityState.Modified)
            {
                // Pentru actualizarea entității, aducem din baza de date
                // copia curentă a entității și îi atribuim noile valori
                // din obiectul Entry. EF va detecta aceste modificări și
                // va marca fiecare entitate ca fiind modificată.
                // Initial setăm starea entității la valoarea "Unchanged".

                entry.State = EntityState.Unchanged;
                var databaseValues = entry.GetDatabaseValues();
                entry.OriginalValues.SetValues(databaseValues);
            }
        }
        context.SaveChanges();
    }
    return true;
}

```

Concurența EF

Proiectantul fiecărei aplicații stabilește regulile ce vor guverna concurența în cadrul sistemului pe care-l construiește.

Putem întâlni diverse tipuri de probleme de concurență.

1. Ce modificare luăm în considerare, ultima, penultima, etc.
2. Drepturi asupra coloanelor și tabelelor supuse modificării.
3. Concurența la nivel de bază de date, concurența la nivel de tabele, concurența la nivel de înregistrare.

Mai mult de atat unele actiuni nu trebuiesc supuse unor reguli stricte de concurenta (rapoarte periodice informative) in timp ce alte actiuni sunt vitale pentru buna executie a aplicatiei (actualizari stocuri, conturi bancare, etc.).

Intelegerea conflictelor de concurenta intr-o baza de date

Concurenta pesimista: ne putem astepta la ce e mai rau.

Concurenta optimista: speram pentru ce e mai bine.

Ce inseamna concurenta pesimista?

O inregistrare dintr-o tabela este blocata cand utilizatorul o acceseaza si este eliberata cand utilizatorul a terminat lucrul cu ea. Nimeni in acest interval de timp nu poate accesa acea inregistrare. Potentialul uni conflict este mic in acest caz.

Dezavantaje: concurenta pesimista nu e scalabila pentru ca mentine o conexiune deschisa la baza de date si poate cauza o blocare excesiva, lunga ca durata si in unele situatii chiar “deadlocks”.

EF nu suporta concurenta pesimista.

Ce inseamna concurenta optimista?

Concurenta optimista nu blocheaza inregistrari din baza de date si da posibilitatea dezvoltatorului de a furniza logica in cadrul aplicatiei pentru gestionarea potentialelor conflicte la actualizari (update).

EF furnizeaza utilitare pentru a rezolva problema concurentei.

ClientWins – salvare ultime modificari, se actualizeaza toate campurile. Asemenator ca la proceduri stocate unde se pierd actuaizarile facute de alt utilizator.

StoreWins – readucere din baza de date si apoi salvare.

Determinare scop schimbări

Verificare pentru orice schimbare in cadrul inregistrarii – proprietatea **ConcurrencyMode** ce poate fi setata pe **Fixed** pentru orice proprietate pe care dorim sa o verificam.

Verificare pentru schimbarile unui camp particular

Campul *rowversion* (adica timestamp) este folosit in acest caz. De mentionat ca nu toate serverele de baze de date suporta acest camp. Acest *timestamp* se aplica la toata inregistrarea si ca atare nu avem informatii despre ce camp s-a modificat.

Verificare daca campurile pe care le actualizam au fost modificate intre timp.

Solutii:

- Cache inregistrare initiala (de catre dezvoltator).

- Verificare timestamp.

- Verificare inregistrare din baza de date cu ceea ce este in cache – asta inseamna ca la un moment dat o inregistrare poate necesita spatiu de memorie de cel mult trei ori lungimea inregistrarii, neluand in seama spatiul de memorie cerut de serverul de baze de date.

Idei de urmat in gestionarea concurentei

Urmare modificare proprietati:

1. Gestionare proprietati modificate pe partea de client si pasarea unei liste cu aceste modificari pe partea de server.
2. Memorare date originale entitati inainte de a fi pasate la client.
3. Readucere date din baza de date cand entitatile au fost transferate catre server.

Explicatie

1. Se doreste pastrarea unei liste cu proprietatile modificate din cadrul unei entitati, lista ce va fi pasata serverului. Ideea este de a face update numai pe proprietatile modificate. In cazul proprietatilor complexe, EF va genera update pentru toate proprietatile tipului complex indiferent ca s-a modificat sau nu valoarea. Pentru a pastra cat mai simplu modelul in interfata *IObjectWithState* ar trebui sa adaugam ceva de genul:

```
List<string> ModifiedProperties { get; set; }
```

2. Pentru fiecare proprietate modificata se pastreaza valoarea veche.
3. In baza de date exista valoarea inainte de incarcare in context.

Discutie – date modificate intr-un context si salvate in alt context.

```
private static void AttachCustomerType(CustomerType ct)
{
    using (var context = new ModellContainer())
    {
        context.CustomerTypes.Attach(ct);
        context.SaveChanges();
    }
}
```

Dupa executia acestei metode, *AttachCustomerType*, nu se intampla nimic.

```
private static void SaveCustomerType(CustomerType ct)
{
    using (var context = new ModellContainer())
    {
        context.Entry(ct).State = EntityState.Added;
        context.SaveChanges();
    }
}
```

In acest caz datele sunt salvate in baza de date.

Exemplu corect cu doua metode ce lucreaza cu contexte diferite.

```
private static void TestUpdateCustomerType()
{

```

```

CustomerType ct;
using (var context = new ModellContainer())
{
    ct = (from d in context.CustomerTypes
          where d.Description == "Preferat"
          select d).Single();
}
// ct != null ?
UpdateCustomerType(ct);
}

private static void UpdateCustomerType(CustomerType ct)
{
    using (var context = new ModellContainer())
    {
        context.Entry(ct).State = EntityState.Modified;
        context.SaveChanges();
    }
}

```

Putem realiza aplicatii n-tier utilizand Web API, WCF, ASP.NET MVC.

Urmeaza exemple pentru fiecare din aceste cazuri.