Administrarea și Dezvoltarea Aplicațiilor cu Baze de Date

Curs 8. LINQ to SQL

Cuprins

LINQ to SQL

1. Introducere

LINQ to SQL este o tehnologie de programare de la Microsoft care oferă o modalitate eficientă de accesare și manipulare a bazelor de date folosind limbajul de programare .NET, în special C# sau VB.NET. "LINQ" este acronimul pentru "Language Integrated Query" și este integrat în limbajul de programare, permițând dezvoltatorilor să scrie interogări de baze de date într-un mod mai natural și mai expresiv, folosind sintaxa limbajului de programare.

Principalele caracteristici ale LINQ to SQL includ:

1. **Integrare în .NET**: LINQ to SQL este strâns integrat cu limbajele .NET, ceea ce face posibilă scrierea de interogări de baze de date direct în C# sau VB.NET.
2. **Obiect-Relațional Mapping**: Permite maparea tabelelor dintr-o bază de date SQL la clase .NET. Aceasta înseamnă că fiecare tabel poate fi reprezentat ca o clasă, iar rândurile tabelei ca instanțe ale acelei clase.
3. **Interogări**: Cu LINQ to SQL, interogările sunt scrise în limbajul .NET și sunt traduse automat în SQL de către LINQ to SQL. Acest lucru face interogările mai puternice și mai flexibile.
4. **Gestionarea Modificărilor**: LINQ to SQL urmărește modificările aduse obiectelor și generează automat comenzi SQL necesare pentru actualizarea bazei de date.
5. **Suport pentru Tranzacții**: Permite gestionarea tranzacțiilor pentru a asigura consistența datelor.
6. **Performanță**: Deși nu este la fel de rapid ca interogările SQL brute, LINQ to SQL este optimizat pentru a oferi performanțe bune pentru majoritatea aplicațiilor.

LINQ to SQL este util în aplicații unde se dorește o integrare strânsă între baza de date și logica de programare, facilitând scrierea de cod mai clar și mai ușor de întreținut. Cu toate acestea, pentru scenarii mai complexe sau pentru performanțe maximizate, uneori se preferă alte soluții, cum ar fi Entity Framework sau interogări SQL directe.

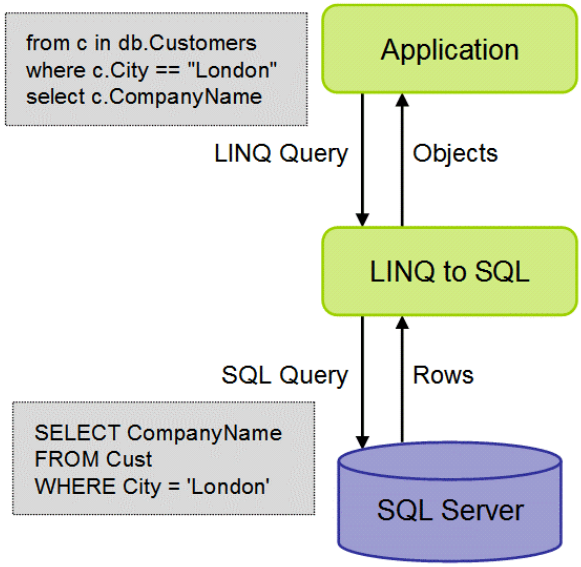

Fig1. Exemplu ORM LINQ to SQL. Sursa imagine: https://codesamplez.com/database/insert-update-delete-linq-to-sql

Figura 1 prezintă un flux de lucru ce descrie interacțiunea dintre o aplicație, LINQ to SQL și SQL Server:

1. **Aplicație**: Este nivelul de sus al diagramei și reprezintă aplicația software unde este scrisă interogarea LINQ. Aplicația poate fi orice program care rulează pe .NET Framework și care utilizează LINQ pentru a interoga baze de date.
2. **Interogarea LINQ (LINQ Query)**: În interogare LINQ se selectează numele companiei dintr-un set de clienți în baza de date, filtrând pentru acele companii care sunt situate în "London".
3. **LINQ to SQL**: Este tehnologia care acționează ca un intermediar între aplicație și SQL Server. Ea preia interogarea LINQ de la aplicație și o traduce într-o interogare SQL pe care serverul de baze de date o poate înțelege.
4. **Interogarea SQL (SQL Query)**: Este codul SQL generat de LINQ to SQL ca rezultat al interogării LINQ de mai sus. Este un exemplu de interogare SQL care ar fi executată pe un server SQL pentru a obține datele solicitate.
5. **SQL Server**: Este sistemul de gestionare a bazelor de date care execută interogarea SQL și returnează rezultatele. În acest caz, este un tabel sau o vizualizare numită "Cust".
6. **Rânduri (Rows)**: Rezultatele interogării SQL sunt returnate de SQL Server sub forma unor rânduri de date.
7. **Obiecte (Objects)**: LINQ to SQL mapează aceste rânduri la obiecte în cadrul aplicației .NET, permițând dezvoltatorului să lucreze cu aceste date ca obiecte în cadrul limbajului de programare folosit.

Diagrama evidențiază procesul de transformare a interogărilor de înalt nivel scrise într-un limbaj .NET în interogări SQL native și fluxul de date între aplicație și baza de date.

2. Crearea unui nou proiect și conectarea la baza de date folosind LINQ

Deschideti Visual Studio si creati un nou proiect Console Application (.Net Framework). Click dreapta pe nodul proiectului din Solution Explorer -> Add -> New item -> LINQ to SQL class.

Obs: Daca aceasta nu exista, porniti installer-ul de Visual Studio -> Modify -> Individual Components (tab) -> Selectati LINQ to SQL tools. Redeschideti Visual Studio.

Denumiti clasa Northwind.dbml (database markup language). Se va crea o fereastra cu doua panouri in care puteti adauga obiecte din meniul Server Explorer.

Daca nu aveti o conexiune la SQL Server configurata, din Server Explorer -> click dreapta pe nodul Data Connections -> Add Connection -> Microsoft SQL Server. Completati cu . pentru server-ul local si Northwind pentru baza de date. Testati conexiunea si creati-o.

Adaugati tabelele Customers, Orders, OrderDetails, Employee si procedurile stocate CustOrderHist si CustOrderDetails.

Explorati NorthwindDataContext. Observati ca s-a creat automat un connectionString in AppConfig care este apelat in constructorul clasei. Urmariti cu atentie fiecare proprietate a clasei.

Exemplu:

```
            var context = new NorthwindDataContext(); var
            employees = from emp in context.Employees
                        where emp.LastName.StartsWith("D")
                        select emp;

            foreach(var employee in employees)
            {
                Console.WriteLine("{0} \t {1} \t {2}", employee.FirstName,
employee.LastName, employee.Title);
            }
```

3. Eager Loading vs. Lazy Loading

Cand se specifica proprietatile unei entitati in cadrul unei interogari se poate efectua **eager loading** sau **lazy loading**. Lazy loading este de asemenea cunoscut si ca **delay loading**, iar eager loading este cunoscut ca **pre-fetch loading**. Comportamentul implicit este de a se aplica eager loading asupra proprietatilor, ceea ce inseamna ca **proprietatile sunt incarcate atunci cand se executa interogarea care face referinta la proprietatile respective.**

Lazy loading este configurat din designer-ul LINQ to SQL prin selectarea unei proprietati a unei entitati si apoi setarea *Delay loading* ca *true* in fereastra *Properties*.

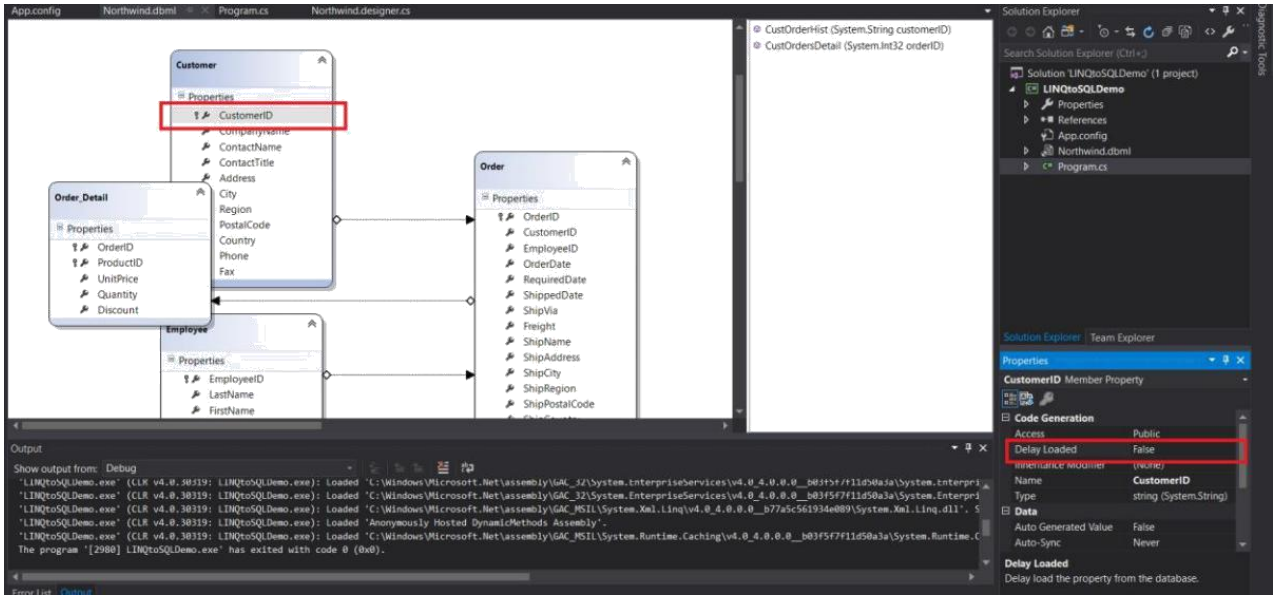

Fig.2. Configurare Delay Loaded

Atunci cand se foloseste lazy loading, proprietatea nu este incarcata pana cand nu este accesata explicit. Atunci cand luati in calcul folosirea lazy loading trebuie sa puneti in balanta criteriile de performanta ale aplicatiei. **In cazul lazy loading de fiecare data cand proprietatea lazy este folosita, va aparea un cost in delay-ul aplicatiei cauzat de stabilirea unei noi conexiuni catre baza de date.** Decizia de a folosi lazy loading depinde de cat de multe date vor fi transferate catre utilizator.

Exemplu:

```
private static void LazyExample()
{
    var context = new NorthwindDataContext();
    var sw = new StringWriter();
    context.Log = sw;
    var employee = (from emp in context.Employees
                    where emp.LastName.StartsWith("Davolio")
                    select emp).First();
     Console.WriteLine("----1----");
    Console.Write(sw.GetStringBuilder().ToString());
    Console.WriteLine();
    Console.WriteLine();
    sw = new StringWriter();
    context.Log = sw;
    Console.WriteLine("----2----");
    var photo = new MemoryStream(employee.Photo.ToArray());
    Console.Write(sw.GetStringBuilder().ToString());
}
```

In exemplul anterior proprietatea Photo din entitatea Employee este configurata in prealabil ca lazy. Clasa de baza *DataContext* pe care o mosteneste *NorthwindDataContext* are proprietatea *Log* care este de tipul *TextWriter* ce ne permite sa vizualizam interogarile trimise de aplicatie salvate sub forma unui jurnal in cadrul unui obiect *StringWriter*.

Observati in cele doua output-uri momentul in care este selectata proprietatea *Log*: atunci cand este folosita.

4. Executia interogarilor folosind LINQ to SQL

LINQ to SQL poate executa o diversitate de interogări care corespund operațiunilor standard SQL. Exemple de interogări ce pot fi realizate:

1. **Selectări**: Datele pot fi extrase din tabele și proiectate în obiecte. Se pot selecta, de exemplu, numele tuturor clienților dintr-un anumit oraș.
2. **Filtrare**: Se poate utiliza clauza **where** pentru a filtra datele conform unor condiții specifice.
3. **Ordonare**: Rezultatele pot fi sortate în ordine ascendentă sau descendentă folosind **orderby** sau **orderbydescending**.
4. **Grupare**: Se pot grupa datele după anumite coloane utilizând **groupby** și se pot efectua calcule pe aceste grupuri.
5. **Join-uri**: LINQ to SQL permite realizarea join-urilor interne și externe pentru a combina date din mai multe tabele.
6. **Agregare**: Funcțiile de agregare precum **sum**, **avg**, **min**, **max**, și **count** sunt disponibile pentru calculul valorilor agregate.
7. **Subinterogări**: Interogările în interogări, sau subinterogări, pot fi utilizate pentru a efectua operațiuni mai complexe.
8. **Paginare**: Se pot selecta segmente specifice dintr-un set de rezultate utilizând **skip** și **take**.
9. **Interogări Compuse**: Interogările pot fi combinate utilizând operatori precum **Concat**, **Union**, **Intersect**, și **Except**.
10. **Interogări Dinamice**: Interogările pot fi construite dinamic utilizând expresii și funcții LINQ pentru a crea interogări flexibile în timpul execuției.
11. **Update**: Actualizările se pot face prin modificarea obiectelor obținute și apoi aplicând aceste modificări în baza de date.
12. **Insert**: Se pot insera noi rânduri în tabele prin crearea de noi instanțe ale claselor de entitate și adăugarea lor în contextul datelor.
13. **Delete**: Rândurile pot fi șterse din tabele prin eliminarea obiectelor din contextul datelor și confirmarea modificărilor.

Aceste interogări sunt exprimate în LINQ utilizând metode de extensie și sintaxa limbajului .NET, care sunt traduse în SQL de către LINQ to SQL pentru execuția în baza de date. Astfel, LINQ to SQL funcționează ca un strat de abstracție care facilitează manipularea datelor într-un mod orientat pe obiecte.

4.1. Filtrare si Sortare

În LINQ to SQL, filtrarea și sortarea sunt două operațiuni esențiale care permit manipularea și prezentarea datelor în moduri specifice și eficiente.

Filtrarea în LINQ to SQL este procesul de restrângere a setului de rezultate la acele elemente care îndeplinesc o anumită condiție. Acest lucru este similar cu utilizarea clauzei **WHERE** în SQL. Filtrarea se realizează în LINQ prin utilizarea clauzei **where**, care poate fi aplicată colecțiilor de obiecte. Aceasta permite dezvoltatorilor să specifice condiții logice pentru alegerea înregistrărilor care trebuie incluse în rezultat. De exemplu, se pot extrage doar clienții dintr-un oraș specific sau

produsele care se încadrează într-o anumită gamă de prețuri. Filtrarea este un instrument puternic pentru a îmbunătăți performanța interogărilor prin limitarea datelor care trebuie recuperate și procesate.

Sortarea în LINQ to SQL se referă la organizarea rezultatelor unei interogări într-o anumită ordine, fie ascendentă, folosind **orderby**, fie descendentă, folosind **orderbydescending**. Acest proces este echivalent cu clauza **ORDER BY** în SQL. Sortarea poate fi aplicată pe orice tip de date care poate fi ordonat, cum ar fi numerele, șirurile de caractere și datele. Sortarea poate fi de asemenea lanțuită pentru a sorta după mai multe coloane, ceea ce permite o mai mare finețe în prezentarea datelor.

Exemplu

```
        private static void BasicExample()
        {
            var context = new NorthwindDataContext();

            var customers = from c in context.Customers
                            where c.CompanyName.Contains("Restaurant")
                            orderby c.PostalCode
                            select c;

           foreach(var customer in customers)
           {
                Console.WriteLine("{0} \t {1} \t {2}", customer.ContactName,
customer.ContactTitle, customer.CompanyName);
            }
        }
```

4.2. Proiectii

Proiecțiile în LINQ to SQL sunt operațiuni care transformă rezultatele unei interogări într-o formă diferită. În esență, o proiecție ia datele de intrare și le „proiectează" într-un nou format, fie el un obiect anonim, un tip de date primitiv, sau un obiect complex definit de utilizator. Acest proces este similar cu selectarea unui set specific de coloane dintr-un tabel în SQL.

În LINQ, proiecțiile sunt realizate folosind operatorul `select`. Când folosiți select, puteți specifica exact care proprietăți ale obiectelor din sursa de date doriți să le includeți în rezultate. De exemplu, dacă aveți o listă de obiecte Customer și sunteți interesat doar de numele și orașul fiecărui client, puteți utiliza proiecția pentru a crea o nouă colecție care conține doar aceste două proprietăți.

Exemplul 1.

```
var customerNamesAndCities = from c in db.Customers

                             select new { c.Name, c.City };
```

În exemplul de mai sus, **customerNamesAndCities** va fi o colecție de obiecte anonime, fiecare având proprietăți **Name** și **City**.

Proiecțiile sunt puternice deoarece vă permit să formați datele exact în forma în care aveți nevoie pentru o anumită sarcină. Asta înseamnă că puteți reduce cantitatea de date transferate din baza de date către aplicația client, ceea ce poate avea un impact semnificativ asupra performanței aplicației.

LINQ to SQL convertește aceste proiecții în clauze **SELECT** corespunzătoare în SQL, astfel încât baza de date să returneze doar coloanele solicitate. Aceasta optimizează interogările și reduce încărcătura pe rețea și pe server, deoarece doar datele necesare sunt transmise.

Exemplul 2.

```
        private static void ProjectionExample()
        {
            var context = new NorthwindDataContext();

            var customers = from c in context.Customers
                            where c.CompanyName.Contains("Restaurant")
                            orderby c.PostalCode
                            select new
                            {
                                c.ContactName,
                                c.ContactTitle,
                                c.CompanyName
                            };

            foreach (var customer in customers)
            {
                Console.WriteLine("{0} \t {1} \t {2}", customer.ContactName,
customer.ContactTitle, customer.CompanyName);
            }
        }
```

4.3. Joins

Join-urile în LINQ to SQL sunt folosite pentru a combina rânduri din două sau mai multe tabele bazate pe o condiție de relație, de obicei unde există coloane comune între tabele. Aceasta este analogă operării de join în limbajul SQL, care leagă tabelele pe baza unei condiții specificate și returnează un set de rezultate care include coloane din ambele tabele.

În LINQ to SQL, join-urile sunt realizate utilizând diferite clauze și metode de extensie:

1. **Inner Join**: Acesta este cel mai comun tip de join și este folosit pentru a selecta înregistrările care au valori potrivite în ambele tabele. În LINQ, acesta se realizează folosind clauza **join**.

```
var innerJoinQuery = from customer in db.Customers join order in db.Orders
on customer.CustomerID equals order.CustomerID select new { customer.Name,
order.OrderDate };
```

2. **Group Join**: Group join-ul în LINQ to SQL este similar cu un left outer join în SQL, în care toate elementele din prima (stânga) tabelă sunt returnate indiferent dacă există o potrivire în cea de-a doua tabelă (dreapta), împreună cu elementele grupate din a doua tabelă. În LINQ, acesta este realizat cu ajutorul metodei de extensie **GroupJoin** sau prin combinarea **join** cu **into**.

```
var groupJoinQuery = from customer in db.Customers join order in db.Orders
on customer.CustomerID equals order.CustomerID into orders select new {
customer.Name, Orders = orders };
```

3. **Left Outer Join**: În LINQ, un left outer join implică utilizarea clauzei **DefaultIfEmpty** și poate fi folosit pentru a include toate înregistrările din tabelul din stânga, chiar dacă nu există o corespondență în tabelul din dreapta.

```
var leftOuterJoinQuery = from customer in db.Customers join order in
db.Orders on customer.CustomerID equals order.CustomerID into orders from
order in orders.DefaultIfEmpty() select new { customer.Name, OrderDate =
order?.OrderDate };
```

4. **Cross Join**: Cross join-ul produce un produs cartezian între două tabele, adică combină fiecare rând din prima tabelă cu fiecare rând din cea de-a doua tabelă. În LINQ, acest lucru se poate realiza prin selectarea din ambele tabele fără o condiție de join specifică.

```
var crossJoinQuery = from customer in db.Customers from product in
db.Products select new { customer.Name, product.ProductName };
```

Aceste operații de join permit combinarea și manipularea datelor complexe din diferite tabele într-un mod care este atât expresiv, cât și eficient din punct de vedere al performanței. LINQ to SQL se ocupă de traducerea acestor operații în interogările SQL corespunzătoare pentru execuție în baza de date.

Exemplul 1

```
private static void InnerJoinExample()
{
    var context = new NorthwindDataContext();

    var results = from c in context.Customers
                  join o in context.Orders
                  on c.CustomerID equals o.CustomerID
                  orderby c.CustomerID, o.OrderID
                  select new
                  {
                      c.CustomerID,
                      c.CompanyName,
                      o.OrderID,
                      o.OrderDate
                  };

    foreach (var result in results)
    {
        Console.WriteLine("{0} \t {1} \t {2} \t {3}",
            result.CustomerID,
            result.CompanyName,
            result.OrderID,
            result.OrderDate);
    }
}
```

Metoda **InnerJoinExample** efectuează următoarele operații utilizând LINQ to SQL în contextul unei baze de date numită **NorthwindDataContext**:

1. Inițiază o conexiune la baza de date Northwind printr-o instanță a **NorthwindDataContext**.
2. Definește o interogare LINQ care realizează un join intern (**inner join**) între tabelul **Customers** și tabelul **Orders** pe baza unei chei comune, **CustomerID**. Aceasta înseamnă că

interogarea va selecta doar acele perechi de clienți și comenzi unde **CustomerID** se potrivește în ambele tabele.
3. Ordonează rezultatele întâi după **CustomerID** și apoi după **OrderID**, asigurându-se că datele sunt într-o ordine consistentă.
4. Selectează un set de proprietăți specifice din tabelele combinate și creează un obiect nou pentru fiecare pereche de client și comandă corespunzătoare. Proprietățile selectate sunt **CustomerID**, **CompanyName**, **OrderID** și **OrderDate**.
5. Parcurge colecția de rezultate obținute prin interogare și afișează fiecare **CustomerID**, **CompanyName**, **OrderID** și **OrderDate** în consolă folosind **Console.WriteLine**.

Exemplul 2

```
private static void InnerJoinWithLambdaExample()
{
    var context = new NorthwindDataContext();

    var results = context.Customers.Join(
        context.Orders,
        c => c.CustomerID,
        o => o.CustomerID,
        (c, o) => new
            {
                c.CustomerID,
                c.CompanyName,
                o.OrderID,
                o.OrderDate
            })
        .OrderBy(r => r.CustomerID)
        .ThenBy((r => r.OrderID));

    foreach(var result in results)
    {
        Console.WriteLine("{0} \t {1} \t {2} \t {3}",
            result.CustomerID,
            result.CompanyName,
            result.OrderID,
            result.OrderDate);
    }
}
```

Metoda **InnerJoinWithLambdaExample** efectuează operațiuni similare cu **InnerJoinExample**, dar folosește funcții lambda pentru a specifica condițiile de join și ordonarea rezultatelor:

1. Inițializează o conexiune la baza de date Northwind prin crearea unei instanțe a **NorthwindDataContext**.
2. Folosește metoda de extensie **Join** pentru a realiza un join intern între colecția **Customers** și **Orders** din contextul bazei de date. Această metodă primește patru argumente:
    - Colecția **Orders** cu care se face join-ul.
    - O funcție lambda **c => c.CustomerID** care extrage cheia de join din colecția **Customers**.
    - O funcție lambda **o => o.CustomerID** care extrage cheia de join din colecția **Orders**.
    - O funcție lambda **(c, o) => new { ... }** care creează un nou obiect anonim pentru fiecare pereche de client și comandă care au **CustomerID**-uri care se potrivesc.
3. Lanțuiește metodele de extensie **OrderBy** și **ThenBy**, utilizând funcții lambda pentru a specifica că rezultatele join-ului trebuie să fie ordonate mai întâi după **CustomerID** și apoi după **OrderID**.

4. Parcurge colecția ordonată de rezultate cu o instrucțiune **foreach** și folosește **Console.WriteLine** pentru a afișa proprietățile selectate (**CustomerID**, **CompanyName**, **OrderID**, **OrderDate**) pentru fiecare obiect anonim rezultat din join.

4.4. Trimiterea modificarilor catre baza de date

Atunci cand intoarceti obiecte de la SQL Server prin intermediul unui obiect *DataContext*, LINQ to SQL stocheaza automat informatii despre obiectele respective. Obiectele *DataContext* folosesc un serviciu de urmarire a starilor obiectelor LINQ to SQL. Obiectele LINQ to SQL se afla intotdeauna intr-o anumita stare. Aceste stari sunt descrise in tabelul urmator.

STATE	DESCRIPTION
Untracked	An object not tracked by LINQ to SQL due to one of the following reasons: You instantiated the object yourself. The object was created through deserialization. The object came from a different *DataContext* object.
Unchanged	An object retrieved by using the current *DataContext* object and not known to have been modified since it was created.
PossiblyModified	An object that is attached to a *DataContext* object will be in this state unless you specify otherwise when you attach.
ToBeInserted	An object not retrieved by using the current *DataContext* object, which will send an *insert* statement to the database.
ToBeUpdated	An object known to have been modified since it was retrieved, which sends an *update* statement to the database.
ToBeDeleted	An object marked for deletion, which will send a *delete* statement to the database.
Deleted	An object that has been deleted in the database; this state is final and does not allow for additional transitions.

Fig. 3 – Tipurile de stări ale obiectelor LINQ to SQL. Sursa imagine: https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/object-states-and-change-tracking

Fig.3 prezintă stările prin care trece un obiect gestionat de LINQ to SQL în cadrul ciclului său de viață:

- **Untracked**: Obiectul nu este urmărit de LINQ to SQL, de exemplu, un obiect nou creat care nu a fost interogat prin **DataContext** curent.
- **Unchanged**: Obiectul a fost recuperat folosind **DataContext** curent și nu se știe că a fost modificat de când a fost creat.
- **PossiblyModified**: Obiectul este atașat la un **DataContext**, dar nu se știe cu certitudine dacă a fost modificat.
- **ToBeInserted**: Obiectul nu a fost recuperat prin **DataContext** curent și va cauza o inserție în baza de date la executarea lui **SubmitChanges**.
- **ToBeUpdated**: Obiectul este cunoscut ca fiind modificat de când a fost recuperat și va cauza un update în baza de date la executarea lui **SubmitChanges**.
- **ToBeDeleted**: Obiectul este marcat pentru ștergere și va cauza o ștergere în baza de date la executarea lui **SubmitChanges**.
- **Deleted**: Obiectul a fost șters din baza de date și acesta este stadiul final, fără tranziții suplimentare permise.

4.5. Modificarea entitatilor existente

Atunci cand LINQ to SQL creeaza un nou obiect, obiectul se afla in starea *Unchanged.* Daca obiectul respectiv este modificat, obiectul *DataContext* este notificat prin intermediul metodei SendPropertyChanging care este apelata in cadrul metodei *Set* de la nivelul fiecarei proprietati. Starea obiectului modificat va fi modificata in *ToBeUpdated* de catre obiectul *DataContext*. Prin acest mecanism, obiectul *DataContext* va sti ce modificari trebuie sa trimita catre baza de date la apelarea metodei *SubmitChanges*.

```
private static void ModifyExample()
{
    var context = new NorthwindDataContext(); var
    customer = (from c in context.Customers
                where c.CustomerID == "ALFKI"
                select c).First();

    customer.ContactName = "John Doe";
    context.SubmitChanges();
}
```

4.6. Adaugarea de noi entitati la DataContext

Adăugarea de noi entități în LINQ to SQL se realizează prin următorii pași:

1. Se creează o instanță a contextului de date, care servește ca intermediar între aplicație și baza de date.
2. Se inițializează o nouă instanță a entității dorite, care corespunde unui tabel din baza de date.
3. Noua instanță este adăugată la setul aferent de entități din contextul de date.
4. Se invocă metoda **SubmitChanges()** a contextului de date pentru a persista noile entități în baza de date.

Exemplul 1

```
using (var context = new DataContext()) {
var entity = new Entity { Proprietate1 = valoare1, Proprietate2 = valoare2
};
context.SetDeEntitati.InsertOnSubmit(entity);
context.SubmitChanges(); }
```

În acest cod, **DataContext** este clasa de context, **Entity** este clasa entității, iar **SetDeEntitati** este colecția contextului care reține entitățile. Proprietățile entității sunt inițializate cu valorile necesare înainte de a fi adăugate la context și înainte ca schimbările să fie trimise la baza de date.

Exemplul 2

```
private static void CreateEntity()
{
    var context = new NorthwindDataContext();
    var employee = new Employee
    {
        FirstName = "John",
        LastName = "Doe"
```

```
        };
        context.Employees.InsertOnSubmit(employee);
        context.SubmitChanges();
    }
```

Codul de mai sus definește o metodă numită **CreateEntity** care adaugă o nouă entitate în tabelul **Employees** al bazei de date Northwind:

1. Se inițializează o conexiune la baza de date prin crearea unei instanțe a **NorthwindDataContext**.
2. Se creează o nouă instanță a obiectului **Employee**.
3. Se setează proprietățile **FirstName** și **LastName** ale noii instanțe cu valorile "John" și "Doe".
4. Se adaugă noua instanță de **Employee** la colecția **Employees** din contextul de date folosind metoda **InsertOnSubmit**.
5. Se apelează **SubmitChanges** pe context pentru a efectua operația de inserare în baza de date.

4.7. Stergerea unei Entitati

Ștergerea entităților în LINQ to SQL implică următoarele etape:

1. Se obține entitatea sau entitățile care trebuie șterse, de obicei printr-o interogare LINQ care le extrage din contextul de date.
2. Se utilizează metoda **DeleteOnSubmit(entitate)** pentru a marca entitatea pentru ștergere în cadrul setului de entități din context.
3. Se invocă metoda **SubmitChanges()** pe contextul de date pentru a efectua operațiunea de ștergere și a reflecta modificările în baza de date.

```
using (var context = new DataContext()) {
var entityToDelete = context.EntitySet.First(e => e.ID == idSpecific);
context.EntitySet.DeleteOnSubmit(entityToDelete);
context.SubmitChanges(); }
```

Acest cod șterge entitatea cu ID-ul specificat din colecția **EntitySet** a contextului de date și transmite modificarea în baza de date.

Exemplu

```
    private static void DeleteEntity()
    {
        var context = new NorthwindDataContext();
        var employee = (from emp in context.Employees
                        where emp.EmployeeID == 10
                        select emp).First();
        context.Employees.DeleteOnSubmit(employee);
        context.SubmitChanges();
    }
```

4.8. Folosirea Procedurilor Stocate

Procedurile stocate pot fi utilizate în LINQ to SQL prin maparea lor la metode în contextul de date. Acest proces implică următorii pași:

1. **Maparea Procedurii**: În designerul OR/M al LINQ to SQL, procedura stocată este mapată la o metodă în cadrul unei clase de context de date. Această mapare poate fi realizată automat de Visual Studio atunci când procedurile stocate sunt importate în modelul de date.
2. **Apelarea Procedurii**: După mapare, procedura stocată poate fi apelată ca orice altă metodă a clasei de context. La apelare, LINQ to SQL se ocupă de executarea procedurii stocate în baza de date și de manipularea rezultatelor.
3. **Manipularea Rezultatelor**: Rezultatele procedurii stocate pot fi returnate ca o colecție de obiecte, care apoi pot fi utilizate în aplicație. Dacă procedura stocată returnează un set de rezultate, acesta poate fi mapat la o colecție de entități cunoscute de LINQ to SQL.

Această abordare permite utilizarea caracteristicilor avansate ale procedurilor stocate, cum ar fi manipularea eficientă a unor cantități mari de date sau executarea operațiunilor complexe în baza de date, îmbinând avantajele procedurilor stocate cu flexibilitatea și puterea de expresie a LINQ.

Exemplu

```
private static void StoredProcedureExample()
{
    var context = new NorthwindDataContext();

    var results = context.CustOrderHist("ALFKI");

    foreach(var result in results)
    {
        Console.WriteLine("{0} \t {1}", result.ProductName, result.Total);
    }
}
```

Codul definește o metodă numită **StoredProcedureExample** care execută o procedură stocată în baza de date Northwind:

1. Inițializează o instanță a **NorthwindDataContext** pentru a accesa baza de date.
2. Apelează procedura stocată **CustOrderHist** cu un parametru de intrare ("ALFKI"), care este codul unui client. Procedura stocată este presupusă a fi mapată în cadrul contextului de date **NorthwindDataContext**.
3. Parcurge fiecare rezultat returnat de procedura stocată și afișează **ProductName** și **Total** pentru fiecare în consolă.

Scopul acestei metode este de a obține și de a afișa istoricul comenzilor pentru un client specific, folosind o procedură stocată care efectuează această interogare în baza de date.

Bibliografie

1. Language Integrated Query (LINQ) in C# - C# | Microsoft Learn
2. LINQ to SQL - ADO.NET | Microsoft Learn