LINQ To SQL Part II

Inside the DataContext and Modifying Data

Overview

- Identity – objects versus rows
- Entity lifecycle and the unit of work
- Change tracking
- Updating associations
- Attach and Detach
- Concurrency Management

Modifying Data

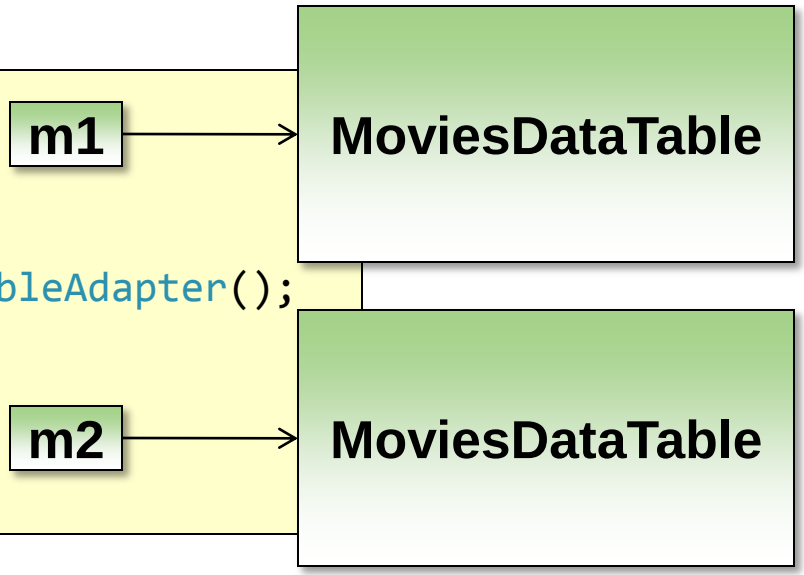
- **Object Relational Mappers want you to think about objects!**
- **CUD operations with ADO.NET typically not about objects.**
  - Insert records by passing parameters
  - Update records by passing parameters
  - Delete records by passing a primary key value
  - All three are data centric approaches
- **In ADO.NET – two objects can represent the same row**
  - The result of two successive invocations of a SQL command
  - This doesn't make sense from an object viewpoint …

Row Identity

- **How do objects relate to rows in the database?**
  - Rows in a database table have a unique primary key
- **What happens if you query for the same movie twice?**
  - Think about the ADO.NET DataSet / SqlReader scenario

```
MoviesDataTable m1;
MoviesDataTable m2;

MoviesTableAdapter adapter = new MoviesTableAdapter();
m1 = adapter.GetData();
m2 = adapter.GetData();
```

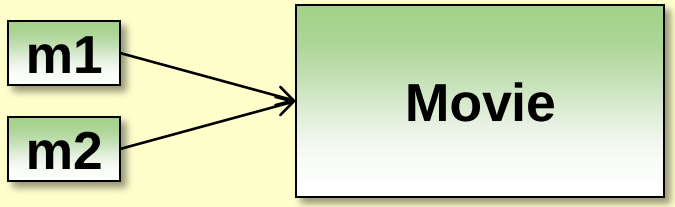


Object Identity

- **What happens if you query for the same movie twice?**
  - As CLR programmers we expect see the same object reference, not two unique objects with the same values.
  - Think about asking a Dictionary<K,T> for an object by unique key

```
Movie m1;
Movie m2;

using (MoviesDataContext dc =
       new MoviesDataContext(connectionString))
{
    m1 = dc.Movies.Where(movie => movie.ID == 1).First();
    m2 = dc.Movies.Where(movie => movie.ID == 1).First();
}
```



Identity Map Pattern

- **An Identity Map keeps a record of all objects that have been read from the database in a single business transaction. – Fowler**
- **LINQ to SQL implements an identity map**
  - Retrieved rows are tracked by primary key value.
  - Asking for a previously retrieved row will return the previous object instance
  - The type of query used to retrieve the row is not important
- **Each DataContext instance maintains it's own Identity Map**
  - Query for the same movie in two different DataContexts will return two different objects.
  - We will talk about a "unit of work" with the DataContext soon …

Consequences of the Identity Map

- **Any changes from "outside" are not visible to our current DataContext (if we've already retrieved a row)**
  - We want consistency and integrity inside our working context
  - The only changes we see are local changes
  - We will revisit concurrency later
- **LINQ to SQL cannot update a table with no primary key**
  - No way to ensure uniqueness and integrity of retrieved objects

Unit of work Pattern

- ***Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.* – Fowler**
- **LINQ to SQL DataContext is designed to be used in a unit of work**
  - For web apps, a unit of work may represent the processing of a single request
  - For smart client, a unit of work may be the life of a form
  - Unit of work may be encapsulated inside a single method
- **DataContext is inexpensive to create**
  - Create as needed
  - Don’t cache or create a singleton DataContext
  - DataContext is not thread safe

Entity Lifecycle

- **Object becomes an entity when DataContext becomes aware of the object**
  - Beginning of the lifecycle
  - Can happen when object is retrieved from database
  - Can also insert new objects and attach existing objects
- **Lifecycle ends when DataContext no longer needed**
  - DataContext and object eligible for garbage collection

Updates

- **Retrieve an entity from the DataContext**
- **Update the entity as you would any object instance**
- **Use SubmitChanges to conclude the current unit of work**
  - SubmitChanges will “flush” *all* changes to the database

```
using (MoviesDataContext context =
                    new MoviesDataContext(connectionString))
{
    Movie movie = context.Movies.Where(m => m.ID == 1).First();
    movie.ReleaseDate = movie.ReleaseDate.AddDays(1);
    context.SubmitChanges();
}
```

Change Tracking

- **DataContext tracks changes for you**
  - DataContext uses list of changes to generate SQL commands
- **How does the DataContext know what changed?**

Change Tracking with POCOs

- **For strict POCOs, LINQ to SQL will take a snapshot of the object when it begins life as an entity.**
  - All original values are copied
  - During SubmitChanges, LINQ to SQL must compare existing values to original values
  - Some expense incurred
  - Turn off this feature by with the DataContext's ObjectTrackingEnabled property

INotifyPropertyChanging

- **INotifyPropertyChanging is an optimization for LINQ to SQL**
  - Does not need a snapshot until a PropertyChanging event fires
  - Implementing this interface and you don't pay for change tracking unless you need it

```
[Column(Name="movie_id“, Storage = "_movie_id”]
public int ID
{
    get { return this._movie_id; }
    set {
          if ((this._movie_id != value)) {
            this.SendPropertyChanging();
            this._movie_id = value;
        }
    }
}
private int _movie_id;
```

Updating Associations

- **Changing an object's relationship to other objects in a graph requires some work**
  - Object needs to change it's parent reference
  - Object needs to be removed from the original parent's collection
  - Object needs to be added to it's new parent's collection

Updating Associations with POCOs

- **Never update a foreign key field manually.**
  - LINQ to SQL will figure this out
- **LINQ can will figure out the updates, inserts, deletes**
  - But its not always obvious how to get there…

```
Movie m1 = context.Movies.Where(m => m.ID == 1).First();
Movie m2 = context.Movies.Where(m => m.ID == 2).First();

m2.Reviews.AddRange(m1.Reviews);
m1.Reviews.Clear();
context.SubmitChanges(); // nothing happens
```

```
Review[] reviews = m1.Reviews.ToArray();
foreach(Review r in reviews) {
    m1.Reviews.Remove(r);
    m2.Reviews.Add(r);
    r.Movie = m2;  // must change the parent
}
context.SubmitChanges(); // this works!!
```

Using EntitySet<T>

- **EntitySet<T> helps manage associations**
  - As does generated code …

```
public Movie() {
    Action<Review> onAdd = r => r.Movie = this;
    Action<Review> onRemove = r => r.Movie = null;
    _reviews = new EntitySet<Review>(onAdd, onRemove);
}

[Association(ThisKey="ID", OtherKey = "MovieID", Storage="_reviews")]
public EntitySet<Review> Reviews
{
    get { return _reviews; }
    set { _reviews.Assign(_reviews);}
}
private EntitySet<Review> _reviews;

// ...
```

```
m2.Reviews.AddRange(m1.Reviews);
context.SubmitChanges();     // this works!!
```

Inserts

- LINQ to SQL will compute an INSERT statement for all new objects in the graph
- LINQ to SQL can retrieve autogenerated IDs

```
Movie movie = new Movie {
  Title = "Hairspray",
  ReleaseDate = new DateTime(2007, 6, 1)
};

Review myReview = new Review {
  Rating = 10, Reviewer = "scott",
  ReviewText = "I want to see it again and again!",
  Summary = "Fantastic!"
};

movie.Reviews.Add(myReview);
context.Movies.InsertOnSubmit(movie);
context.SubmitChanges();
```

Deletes

- **LINQ to SQL will calculate ordering of command to avoid key violations**
- **Associated entities are not deleted**
  - This behavior is configurable

```
Movie movie = context.Movies.Where(m => m.ID == 1).First();

context.Movies.DeleteOnSubmit(movie);
context.Reviews.DeleteAllOnSubmit(movie.Reviews);
context.SubmitChanges();
```

Concurrency Management

- **Optimistic concurrency checks by default**
  - Control in mapping with UpdateCheck: Always, Never, WhenChanged
- **Optimization: use a version column**
  - In mapping: IsVersion = true
  - In SQL: use timestamp or rowversion type

```
UPDATE [movies]
SET [release_date] = @p3
WHERE ([movie_id] = @p0) AND
      ([title] = @p1) AND
      ([release_date] = @p2)
```

```
[Column(Name="version", IsVersion=true,
        IsDbGenerated=true)]
public Binary Version { get; set; }
```

```
UPDATE [movies]
SET [release_date] = @p2
WHERE ([movie_id] = @p0) AND ([version] = @p1)
```

Concurrency Violations

- **SubmitChanges will throw an exception**
  - ChangeConflictException
  - DataContext includes ChangeConflict details (original value, submitted value, database value)
- **SubmitChanges is atomic - all changes roll back**
- **DataContext left unchanged**
  - Changes can be resubmitted

Transactions

- **Use the promotable TransactionScope from System.Transactions**

```
using(TransactionScope txn = new TransactionScope())
using (MoviesDataContext context = new MoviesDataContext(...))
{
    Movie movie = context.Movies.Where(m => m.ID == 1).First();
    movie.ReleaseDate = movie.ReleaseDate.AddDays(1);
    context.SubmitChanges();

    txn.Complete();
}
```

Detached Entities

- **Detached entities are entities that “leave” their DataContext**
  - Sent over the wire in a web service call
  - Sent to a client browser for editing
- **Later the entity can be re-attached**
  - But you have to describe how the entity has changed
  - One approach is to query for the current entity in the database then apply changes
  - Entities cannot move between DataContext instances easily

LINQ to SQL Limitations

- **Mapping limitations**
  - Inheritance mapping with discriminators only
  - No mapping for value types (a domain driven design concept)
- **Platform limitations**
  - Currently no support beyond SQL Server
- **Design limitations**
  - No bulk inserts or massive database updates (slow)
- **Other issues to know about**
  - Will not use default values in database
  - Change tracking for detached entities

Summary

- DataContext is the unit of work for LINQ to SQL
  - Maintains a change tracking service
  - Maintains an identity map
- LINQ to SQL uses optimistic concurreny
- DataContext will work with System.Transactions
- Think of objects, not database operations