

# LINQ in Layered Applications

Choices & Patterns



# Topics

**Operations:  
Lazy  
or  
Greedy?**

**Queries:  
Composable  
or  
Predicatable?**

**Abstractions:  
Build  
or  
Borrow?**

**Session Scope:  
Local  
Or  
Open?**

**Testing:  
Mocks  
or  
Fakes?**

# IQueryable

- **What we are talking about applies to most remote LINQ providers**
  - NHibernate
  - Entity Framework
  - WCF Data Services Client
- **We'll use EF in the demos**

# Abstractions – Build or Borrow?

- **Repository Pattern**

- Build custom repositories (like IRepository<T>)
- Use a built-in abstraction (like IObjectSet<T>)

- **Unit of Work Pattern**

- Build your own unit of work
- Use a built-in abstraction (ISession,ObjectContext)

# Pros and Cons

**ISession**  
Interface  
→ IDisposable

**Properties**

**Methods**

- BeginTransaction (+ 1 overload)
- CancelQuery
- Clear
- Close
- Contains
- CreateCriteria<T> (+ 5 overloads)
- CreateFilter
- CreateMultiCriteria
- CreateMultiQuery
- CreateQuery (+ 1 overload)
- CreateSQLQuery
- Delete (+ 4 overloads)
- DisableFilter
- Disconnect
- EnableFilter
- Evict
- Flush
- Get (+ 4 overloads)
- GetCurrentLockMode
- GetEnabledFilter
- GetEntityName

**ObjectContext**  
Class

**Properties**

**Methods**

- AcceptAllChanges
- AddObject
- ApplyCurrentValues<TEntity>
- ApplyOriginalValues<TEntity>
- ApplyPropertyChanges
- Attach
- AttachTo
- CreateDatabase
- CreateDatabaseScript
- CreateEntityKey
- CreateObject<T>
- CreateObjectSet<TEntity> (...)
- CreateProxyTypes
- CreateQuery<T>
- DatabaseExists
- DeleteDatabase
- DeleteObject
- Detach
- DetectChanges
- Dispose (+ 1 overload)
- ExecuteFunction<TElement> ...
- ExecuteStoreCommand

**IRepository<T>**  
GenericInterface

**Methods**

- Add
- FindAll
- FindById
- FindWhere
- Remove

**IUnitOfWork**  
Interface

**Properties**

- Employees
- TimeCards

**Methods**

- Commit

# Fully Composable Queries?

- There are two worlds in LINQ
  - Both are lazy by default



**IQueryable<T>**

**IEnumerable<T>**

# Pros and Cons

- **IQueryable Pros**

- Allows a LINQ provider to take a holistic view of a query
- Generated queries are as efficient as possible

- **Cons**

- Universe of possible queries isn't as predictable
- Easier to miss an index

# Queries – What are They?

- What does a query produce?
  - A concrete list?
  - A data structure for future execution?

```
public IQueryable<T> FindWhere(Expression<Func<T, bool>> predicate) {  
    return _objectSet.Where(predicate);  
}
```



# Stopping Deferred Execution

- Use one of the "greedy" operators
  - ToList, ToArray, ToDictionary
- Produce a concrete type
  - Sum, First, Single, Count

```
public IList<T> FindWhere(Expression<Func<T, bool>> predicate) {  
    return _objectSet.Where(predicate).ToList();  
}
```

# Deferred Execution

- **Pros**

- We might not need the result

- **Cons**

- It doesn't fail fast
  - It might execute more than once!

# Testing – Mocks or Fakes?

- Test doubles for IEnumerable and IQueryable are easy
  - For fakes use AsQueryable operator
- IRepository<T>, IUnitOfWork, ISession are easy, too
  - ObjectContext – not so much

# Pros and Cons

- **Fakes**

- Pros: state testing all the way
- Cons: test data can be cumbersome

- **Mocks**

- Pros: can test state or interaction and fewer fakes to write
- Cons: will still need test data in most scenarios, end result might be more code

- **Other alternatives**

- Embedded databases

- **Note: you still need integration tests**

- Some LINQ queries only work against objects ....

# Context/Session Scope

- Contexts and Sessions are IDisposable (and often transactional)
- Local to a method?
- Local to a HTTP request or form?

# Managing Contexts

- **Keeping context local to a method**
  - Cons: No deferred execution, no lazy loading, no work done outside the method (or is that a pro?)
  - Pros: Easy to see the scope of a unit of work
- **Keeping a context per request or per form**
  - Cons: Need a way to manage the context
  - Pros: No worries on closing a session too early

# Summary

- **LINQ offers a powerful abstraction**
  - Easy to implement the Repository and Unit of Work design patterns
  - In most cases just delegate to ORM framework
  - Easy to test with fakes or mocks
  - Tradeoffs around performance and predictability