Boost (v 0.3)

An open source framework to aid .NET developers with certain data manipulations and mapping scenarios. It tries to use the simplest syntax and be as light weight as possible.

Boost like other frameworks tries to apply convention over configuration where possible. This is achieved in Boost by minimizing the need for inheritance and implementing Interfaces, while staying as extensible as possible. These goals can often be at odds with each other. Boost solves this dilemma by supplying API's which are replace with your own code.

Behind the name

The name Boost is derived and takes inspiration from a combination of puns and admired software products.

- *Rails uses the Action, and Action Pack monikers. Boost is en exaggeration of Action.*
- *Boost can be broken into the composite word for Boot and Strap. A fairly well known software idea for system. In itself a pun from the phrase "pulling oneself by one's own bootstrap".*
- It also lends itself to additional puns, boost pack, turbo boost, nitro boost, booster.
- Minimally inspired by Ruby on Rails, Cocoa, and others.

Overview

The initial motivation for this library came from POCO (PONO) support for Linq ORM frameworks specifically Microsoft Entity Framework. However shortly after a prototype it became clear that the approach could serve to Linq enable object or more specifically domain libraries with little effort to any Linq provider. More importantly than even opening the door on using any ORM, it allows for a strong abstraction in the application and data. Things like caching and or proper delegation can now be put in place instead of the a particular Linq provider's implementation. In some ways it's inverting the control, or putting the developer back in control over the data.

Summary

After implementing Linq, the effort quickly turn to other aspects of real world applications such as mapping, modeling, etc. The mapping layer exists as a set of loosely coupled methods (in the form of delegates) and which take full advantage of C# 3.0 features.

Goals

Lower the barrier to common tasks found in most business applications. Allow for elegant code, while sustaining productivity and happiness with tool. Embrace the consistency found in .NET and expand to use Linq.

- Simple - Easy to use
- Open
- Fun

Technologies
.NET 3.5, Linq, C# 3.0 or (higher).

Prior knowledge of:
IEnumerable, IQueryable, IQueryable Operators,Extenstion methods, Lambda Expressions, Expression trees.

DataQuery

The corner stone is a class that implements IQueryable (Linq interfaces extending IEnumerable) named DataQuery.

Rather than use extension methods that way Linq extends IQueryable and IEnumerable, DataQuery infact implements the a subset of the Linq operators, rather than delegating to a referenced Extension namespace (The way Extension methods work).

The consequences and reasons are numerous, but let's enumerate through several to provide some insight.

Extension Methods allow for extensions to object type (as opposed to class types) without inheritance. The benefit of adding instance method to Interfaces that can serve as the default implementation, without requiring the concrete classes to override. Overriding is done by implementing. Because of these advantages it is also possible to inadvertently reference Namespaces with additional extensions that would create collision or undesired side effects.

Limitations
Use DataQuery instead of IQueryable otherwise the compiler will reference Extension method implementation since IQueryable does not explicitly define any operations.

[Future]
This may change in the future, but because IQueryableProviders may be inconsistent in the expression composition. This is significant because the implementation re-writes part of the expression tree, mainly to replace ParameterExpression of T with ParameterExpression of K. A future implementation may become a full ExpressionVisitor in order to support IQueryable transparently. Implications of performance and other factors are unknown and the future direction may change from these original ideas.

Subordinates

Inheritance Support

Because domain objects can inherit from a querying targeting the base class can only filter I access members in common.

The feature desired is an up cast to the derived subclass from the base query.

Luckily there is already a Linq operator for this type of behavior and a generic version OfType and OfType<T> respectively.

The usage may be...

[Supertype] [SubType]
Contact <-- Person

```
IDataQuery<Person> query = contactStore.Query().OfType<Person>()
        .Where(p=> p.LastName == “Smith”);
```

Without OfType method you would only be able to do

```
// Same signatures as IQueryable
IDataQuery<Contact> query = contactStore.Query().Where(c=> c.Id == 1);
```

Since LastName does not exist on Contact, but is defined in Person.

IStorageStrategy := object that handles CRUD operations for a particular type.

Query
Create
Save
Delete

ObjectMapping:= object containing converters I mappers that allow forward and backward transformations between 2 objects.

The ObjectMapping object uses two standard Delegates definitions from the System namespace.
Action<T1, T2> where T1 and T2 are input method parameters.
Converter<TInput,TOutput>

Implementations

Currently, the implementations of IDataQuery<T> are DataQuery<T> and DataQueryProxy<T,K> and as the name implies it relays supported Linq operators to an underlying IQueryable<K>. In this context K is another type.

[Foreseeably this would work with any underlying IQueryable, but known implementations have not been tested.]

The proxy delegates to the Linq provider of K. In order for this to work a tranformation between T and K must be supplied, so that when the DataQuery<T> is enumerated (via the GetEnumerator method as definied by IEnumerable). As each K item returned by the enumerator, it is then transformed to T and finally returned to the client.

This implementation relies on several conventions.
1. T and K must have the same property names.
2. The common properties must be the same data type or be implicitly convertible.

Providers supported

Entity Framework Storage
Currently only the Entity Framework has been tested and implemented.

Benefits of approach

Enum query support
...
IDataQuery<Person> query = personQuery.Where(p=> p.Gender == Gender.Male);

Modeling

A model without composition or aggregation is not only very useful or realistic for most applications. Boost enables these scenarios by applying a convention. As is true for all conventions in Boost, the guidance can be avoided as long as there is willingness to do additional steps.

Associations

IAssociation

Mapping

The mapping layer contains support for the complex mapping interactions faced in most applications. Boost takes a declarative approach for defining these interactions and uses the features found in C# 3.0 liberally. The mapping contracts exposed use delegates defined in the System namespace with Action<TInput, TOutput> serving as the glue. In addition, the defaults can be full overridden, not a single class/interface has to remain. Best of all, the entire mapping layer can be removed, not a single interface is mandatory in the Boost Linq implementation. Further the existing interfaces and factory/helper methods (Extension methods) can be replaced on as per needed basis (hybrid mode) or a provider can choose to replace.

A Unique Approach
Many frameworks, including commercial ORMs, either use extensive code generation, subclassing or interfaces for transformations. Under most programming tasks that may be a reasonable approach, but with so many different storage strategies, schema architectures and coding practices it would be impossible to begin to support a universal model. Rather letting the user define such transformations was the only choice.
However no software should create API's that are not at least consumed by 1 client as such a fully implemented mapping stack is supplied as part of the core framework. This stack can serve as both a starting point or used for learning purposes. Whichever the case, applications can be created quickly and easily without the need to spend lots of hours creating custom approaches.

A Lesson from Linq
An interesting api lessons exhibited in Linq was interfaces with implementations, and a seemingly new language that mimicked other popular ones. And best of all, all of the machinery was replaceable. As a result much of the default mapping functionality is implemented as Extension Methods over Interfaces using existing delegates as the contract. Additionally, lambda syntax lend itself well to the "template like" semantics that mapping exhibit. The use of lambda declarations along with the fluent interface shorten the amount of code the user needs to write.

Helpers to the rescue

Rather than complicating even the interfaces that are shipped and used as part of the defaults.

[Remember the contracts for mapping are System delegates Action and Converter]

Helpers were added as extension methods as this felt like the natural approach to enable extensibility scenarios with minimum code additions.

- Boost ships default implementations of the Interfaces, by extending the construction and composition of mappers via Extension methods, some or all of these default class implementations can be removed and both the interfaces and helpers would still work.
- Further, new helpers can be developed and used replacing these and yet allowing the previous scenario, a mixing or discarding the mapping classes.
- Finally all can be discarded in choice of another methodology.

Mappers in action

```
var mapper = ...
mapper.Fields((c1, c2 ) => c1.Prop1 = c2.Prop1)
          .Fields((c1, c2 ) => c1.Prop2 = c2.Prop2)
          .Fields((c1, c2 ) => c1.Prop2 = c2.Prop2)
          .Reference(c => c.RefProperty, refMapper)
          .ReferenceList( c => c.MyCollection, colMapper);
```

[This scenario covers simple (primitive types) as well as Reference types with an accessible property (setter). And a collection type that implements ICollection, the “Add” method is called and newly mapped objects are aggregated.]

As demonstrated above, even complicated mappings can be described with a couple simply parameters rather many objects or lengthy descriptions.

This magic is achieved achieved by convention, where the 2 objects being mapped have the same property names.

Alternatively further refinement is done by supplying additional information.

```
var mapper = ...
mapper.Fields((c1, c2 ) => c1.Prop1 = c2.Prop1)
          .Fields((c1, c2 ) => c1.Prop2 = c2.Prop2)
          .Fields((c1, c2 ) => c1.Prop2 = c2.Prop2)
(1)       .Reference(c => c.RefProperty, refMapper).Setter((o,p)=> o.Prop = p) // Explicit setter.
(2)       .ReferenceList( c => c.MyCollection, colMapper, () => new Reference()); // Provide a
constructor method.
```

1. A setter method is defined, which will be used to set the mapped Reference Object.
2. A construction method [Factory] is defined which will be used to instanciate the mapped objects to be added to the collection.

Digging Deeper
Every mapping function is treated equally, even in composition and thus the following statement is true.

FieldMaps = Execute.

Because of this fact, Execute methods become compose-able. This can be added to the FieldMap variable utilizing the existing “+=” operators on Delegate types.

Whenever implementing additional helper method, replacing existing ones, yet using the IMapper family of interfaces. Attempts should be made to append to the FieldMaps, rather than implementing further colleciton semantics unless there is an unavoidable reason.

Even non trivial issues like setting properties of reference types disappear, and become subtle conventions.

Pure Interface based approach, exposing state, overlaying implementation, construction via FactoryMethod.
Exposing state is needed to make it more testable instead of marker interfaces (see ASP.NET and to a certain extent Linq). While this may raise complexity, it also more openly allows construction of interfaces without included extension methods. Making the API more transparent.

Object Construction During Mapping
At times it is necessary to create new instances of object to add to a collection, or set on properties. For value types it is encouraged to implement implicit casting to aid in setter. For references however, this can be a challenge. In order to maintain convention yet allow extensions, there are 3 approaches allowed out of the box.

3. Helper methods enforce the ":new()" generic constraint to minimize reflection cost.
4. Activator <T> creates a new instance where T is the desired mapped type.
5. Supply a parameterless method (delegate) that creates a new instance.

As one can see these are reasonable cost for gradual extension.

Auto Mapping
The auto mapping feature allows reduction of property accessor definition by simply calling the "AutoMap" method (Extension method) on an IMapper.

```
var mapper = ...
mapper.AutoMap();
```

Internally, properties are enumerated and registered using the same process as manual mapping would have. This works as long as the convention for naming the attributes with the same name is followed between two objects being mapped.

Do call this method first before additional map registration.

Consider that nothing prevents code from registering the same attribute more than once, does extra care most be taken when using Auto Mapping.

Current implementation relies on case sensitive Property Name match.
This is a new feature and not fully vetted. Some unit test coverage. Feature does not stabilize mapping as it is a layer on top and using some new Linq expression magic.
[2009.03.09]

Why delegates again?
1. Boost relies on delegates already
2. Delegates are as loose of a contract as is currently allowed in the CLR.
3. Yields much better performance compared to Reflection.

The alternatives would have been code generation or reflection.
Code generation leads to the following:
1. Re-generation as things change
2. Code bloat
3. Brittle maintenance
4. Could create bigger breaks in future versions.

Some historical context for reference.
Following an accidental encounter with a blog post that seemed to have similar concerns relating to mapping. In this case, as it related to exposing DTO's as POCOs

rather than business objects as well as mentioning a framework. The AutoMapper (http://www.codeplex.com/AutoMapper) framework did has currently eerily similar public API. However the implementation seems bloated and reflection heavy.

More history...
Whilst the motivating factor was adherence to the DRY principle by doing away with most of the scalar property defining. Several bloggers deserve mention for inspiration as well as insight into how delegates work. There are some benchmarks as well included as reference.

- http://msmvps.com/blogs/jon_skeet/archive/2008/08/09/making-reflection-fly-and-exploring-delegates.aspx
- http://www.delphicsage.com/home/blog.aspx/d=131/title=Using_Net_3x_Lambda_Expressions_to_Write_More_Concise_Code
- http://rogeralsing.com/2008/02/26/linq-expressions-access-private-fields/

Roadmap

The plan is to release the Boost as an Open Source project, assuming there is interest, and allow greater community involvement.

It also makes sense to include validation/constraints to the mapping. A recent release of Fluent NHibernate posses this feature. This also could lead to a an Auto validation registration via the metadata available at the time of mapping.

View Helpers

Network Helpers

Emailing Helpers

Boost Scaffolding for most common scenarios, including views, controllers, etc.

Post v 1
Transparent Linq support, no longer requiring reference to IDataQuery. Instead implementing full Provider. Allowing mix and matching of Provider execution, although this seems to bring up dangerous leaky abstractions.

Designer, stronger Data layer support, some code gen.

Appendix

Scalar
Delegate