Introdução a LINQ - Revista easy .net Magazine 30

Neste artigo veremos o conceito de LINQ e alguns exemplos de implementações com o mesmo que podem ser utilizados no nosso dia a dia.

Artigo do tipo **Exemplos Práticos**
Recursos especiais neste artigo:
Conteúdo sobre boas práticas.

Introdução a LINQ

LINQ (Language Integrated Query) é uma funcionalidade muito importante do .NET framework, trazendo uma sintaxe próxima do SQL, porém simples para ser utilizada em qualquer fonte de dados, de arrays e listas em memória a implementações para o uso com banco de dados.

Neste artigo veremos seu conceito e alguns exemplos de implementações com o mesmo que podem ser utilizados no nosso dia a dia.

Em que situação o tema é útil

Este tema é útil na recuperação de dados de forma prática, eficiente e produtiva, fornecendo ao desenvolvedor um modo elegante de acessar seus dados estejam eles em memória, arquivos XML ou bases de dados.

Lidar com coleções de dados é algo que desenvolvedores necessitam muitas vezes, sejam dados vindos de um banco de dados, dados de XML, dados em listas de objetos dentre outros veículos, porém o caminho mais comum é ter para cada fonte de dados uma forma diferente de trabalhar.

Eis então que surge no .NET 3.5 a Language-Integrated Query, conhecida como LINQ, tornando a manipulação de itens de dados, sejam objetos, arquivos XML ou retornos de banco de dados, em objetos tendo praticamente a mesma forma para todos eles. Assim se tornou mais fácil trabalhar com os dados de forma unificada.

Tendo surgido no .NET 3.5 ela é suportada a partir do C# 3.0 em diante e também por outras linguagens da plataforma, como por exemplo, o VB.Net.

A LINQ é então uma sub linguagem de paradigma declarativo que roda por cima da CLR podendo ser usada misturada no contexto de outras linguagens, por exemplo, seu projeto pode ser Vb.Net ou C#, entre outras, e usar trechos de código LINQ no meio dos seus códigos.

Temos então a **Figura 1** que mostra onde a LINQ está na arquitetura do .NET Framework:

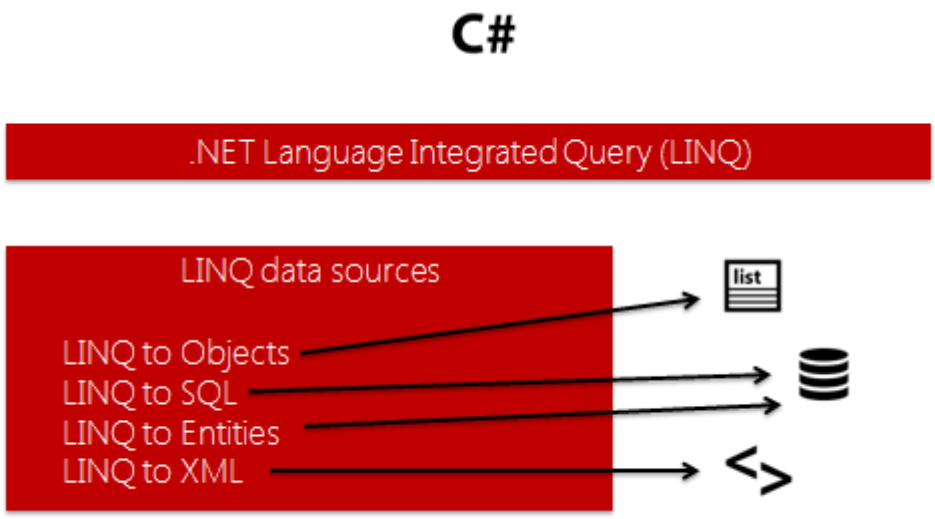


Figura 1. A LINQ dentro do .Net Framework

Como pode ver a LINQ é disponível para qualquer linguagem do .NET como um recurso adicional. Os chamados LINQ-Enabled Data Sources seriam implementações da LINQ para diversos cenários, como a LINQ to SQL, primeiro esboço de ORM da Microsoft, substituído depois pelo Entity Framework que utiliza a LINQ to Entities, além da LINQ to XML em que podemos usar para consultar nós no XML como itens de dados em uma estrutura OO.

Nota: A LINQ consegue trabalhar com qualquer estrutura de objetos que implemente a interface IEnumerable<T>.

As principais vantagens apontadas pela Microsoft para o uso da LINQ são a validação de sintaxe em tempo de execução, suporte a IntelliSense e tipagem estática.

Sintaxe

A linguagem LINQ possui duas formas de pesquisa: sintaxe de consulta (muito próxima da linguagem SQL) e a sintaxe de método.

Sintaxe de consulta

A sintaxe de consulta é chamada de Comprehension Queries e utiliza de palavras chaves em comum com a SQL como from, where e select, porém em uma outra ordem. Segundo o time da LINQ eles fizeram de uma forma mais lógica do que a SQL: primeiro o quê, depois de onde, depois a clausula where e por fim o select. Sempre começando com from e terminando com select, como podemos notar no exemplo da **Listagem 1**.

Listagem 1. Selecionando uma string dentro de um array

```
string[] linguagens = { "C#", "PHP", "Java", "Scala", "C++", "Groovy", "Ruby", "Python" };
              var Favorita = from l in linguagens
```

```
                    where l.Equals("C#")
                    select l;
        foreach (string f in Favorita) {
            Console.WriteLine(f);
        }
```

Na **Listagem 1** nós iniciamos um array com nomes de linguagens, depois buscamos a linguagem favorita para armazená-la em uma variável sendo que para isso indicamos com o **from** o que seria manipulado e completamos com o **in** de onde seria a origem da informação. Depois utilizamos uma clausura where filtrando C#, seguido do comando **select**, sendo que o fluxo da consulta pode ser melhor compreendido observando-se a **Figura 2**.

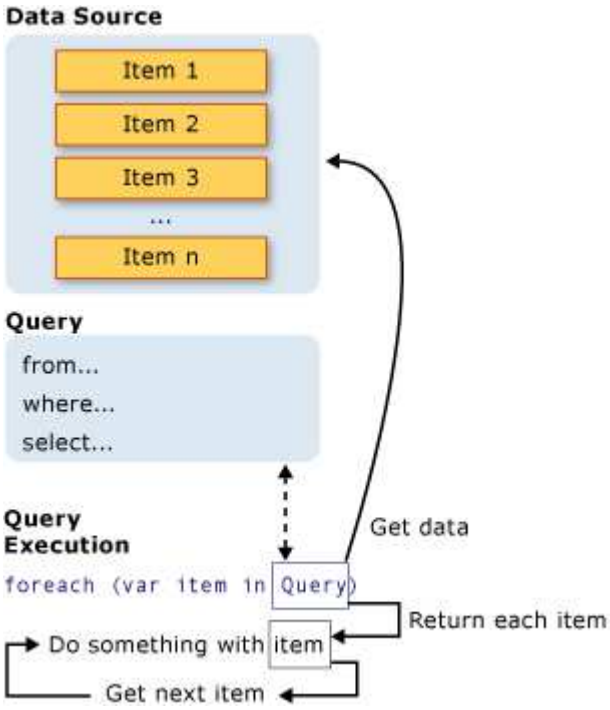


Figura 2. Fluxo de consulta LINQ

Na consulta LINQ também podemos ordenar, agrupar e moldar os resultados. Na MSDN é descrito que a LINQ só será executada quando a variável de consulta for chamada. Dessa forma temos uma execução retardada, porém se quisermos forçar uma execução imediata podemos usar um método de consulta, como First ou Count. Assim teríamos algo como na **Listagem 2.**

Listagem 2. Forçando uma execução de consulta

```
string Favorita =  (from l in linguagens
                    where l.Equals("C#")
                    select l).First();

  Console.WriteLine(Favorita);
```

Sintaxe de método

O básico da sintaxe de método (sinaxy Extension Method) é usar métodos e não somente query para trabalhar com os dados. Um where, por exemplo, pode ser chamado como um método e passado uma expressão lambda para ele.

Um exemplo é demonstrado na **Listagem 3**.

Listagem 3. Exemplo com a mesma query com sintaxe de método

```
var Favorita = linguagens.Where(l => l.Equals("C#"));
              foreach (string f in Favorita)
              {
                  Console.WriteLine(f);
              }
```

Dessa mesma forma você pode fazer joins, order by e várias outras opções de consulta, porém com métodos.

Enquanto isso no CLR

Por termos a LINQ de forma declarativa o CLR não a entende diretamente, por isso no momento de compilação as expressões com LINQ são quebradas em métodos e expressões lambda pela árvore de expressões.

Não há, portanto, praticamente nenhuma diferença de desempenho, pois o mesmo código escrito com LINQ ou com Lambda será passado da mesma forma no IL e será compilado.

O CLR é o ambiente gerenciado do .NET, onde são executados programas .NET antes de serem passados para código de máquina, enquanto que o IL é a Intermediate Language, a linguagem em que todas as linguagens do .NET framework são traduzidas para rodarem por cima do CLR.

Depois disso, no caso das implementações da LINQ que vão ao banco de dados, as querys em IL viram querys SQL que são passadas ao servidor de banco de dados que então irá executá-las e retornar o resultado.

Alguns exemplos de querys

Com LINQ você pode fazer vários tipos de querys, usando ordenação, filtros, paginação, totalizações, dentre outros comandos.

Order by

O Order by serve para ordenar a busca. Você pode indicar qual o campo a ser considerado na ordenação e se é crescente ou decrescente. No caso como estamos trabalhando com um vetor de string o campo é o próprio item. Veja na **Listagem 4** um exemplo de ordenação com sintaxe de query e na **Listagem 5** com sintaxe de método:

Listagem 4. Exemplo com order by com sintaxe de query

```
var Favorita = from l in linguagens
                               orderby l descending
                               select l;
```

Listagem 5. Exemplo com order by com sintaxe de método

```
var Favorita = linguagens.OrderByDescending(l => l);
```

Paginação

O Take é responsável pelo número máximo de itens que serão retornados dentre os primeiros resultados da consulta. Para pegar determinados itens além dos primeiros, podemos combinar o take com o Skip, que pulará X registros do início da consulta.

Para visualizar o exemplo imagine que temos 300 itens de uma lista de produtos e queremos mostrar apenas de 10 em 10. Usaremos então o Skip para indicar onde começa a contagem dos itens que serão retornados da consulta e o Take para indicar a quantidade a ser retornada, como podemos ver na **Listagem 6**.

Listagem 6. Exemplo de paginação

```
var listagem = (from p in produtos select p).Skip((pagina – 1)
* 10).Take(10);
```

No exemplo da **Listagem 6** não utilizamos nenhum filtro (condicional where), mas observe que o Skip recebe como parâmetro o número de onde começa a contar os próximo 10 valores, no caso o número da página anterior multiplicado pela quantidade de itens presentes em cada página (10).

Há também duas variações, uma de cada. O TakeWhere e o SkipWhere. Ambos são alterações que adicionam um delegate condicional para os métodos.

Implementações da LINQ

As primeiras implementações da LINQ foram a LINQ to SQL, LINQ to DataSets, a LINQ to XML e existe também a LINQ to Entities, implementação da LINQ preparada a lidar com objetos do Entity Framework.

LINQ to SQL

Também conhecida como DLINQ essa implementação é especifica para se trabalhar com Sql Server traduzindo seus comando para SQL puro.

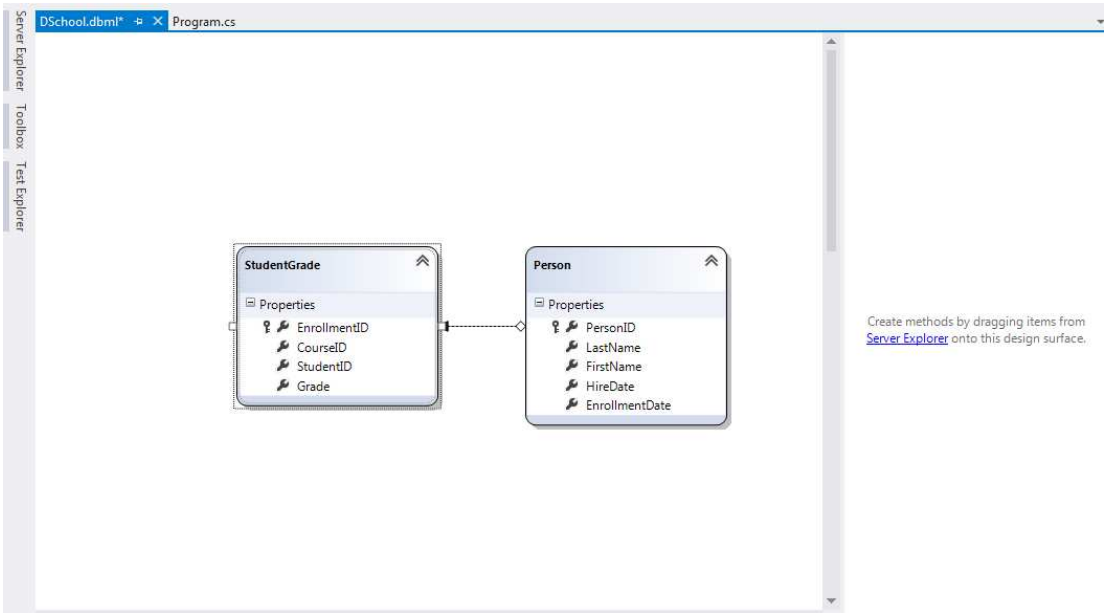
A LINQ to SQL foi uma ferramenta de mapeamento objeto relacional (**BOX 1**), porém sua descontinuidade foi oficializada no lançamento do .NET 4.0, fazendo do Entity Framework a ferramenta de ORM utilizada pela Microsoft.

BOX 1. Ferramenta de Mapeamento Objeto Relacional

E uma ferramenta que permite a criação de um mapeamento entre as tabelas do banco de dados e os objetos de da aplicação, fazendo a ponte entre o modelo relacional do banco de dados e o modelo de objetos da aplicação.

Sendo uma implementação da LINQ, LINQ to SQL faz parte de outro namespace. Para usar de seus no código é preciso adicionar a referencia a System.Data.Linq.

No Visual Studio temos uma ferramenta que nos auxilia no trabalho com LINQ to SQL, o Object/Relational Designer (O/R Designer), que é uma ferramenta visual onde podemos trabalhar o mapeamento de objetos e seus relacionamentos com o clicar e arrastar do mouse.



[abrir imagem em nova janela]

Figura 3. Visão do O/R Designer

Nota: Para esse exemplo estamos usando o banco de dados School, um banco criado para exemplo e estudos e está disponível pela MSDN. Veja onde conseguir o banco nos links do final desse artigo.

Para trabalhar com o O/R Designer é só adicionar no projeto um novo item, escolhendo a opção LINQ to SQL Classes dentro das opções de Acesso a Dados. A tela que abrirá estará em branco, para começar a arrastar suas classes que representam tabelas você pode primeiro ir à aba de Server Explorer e se conectar a uma base de dados.

Depois de se conectar a uma base de dados você pode abrir o nó da base e escolhe uma tabela e clicar e arrastar a tabela ao O/R Designer que representará de forma visual uma classe representando a tabela.

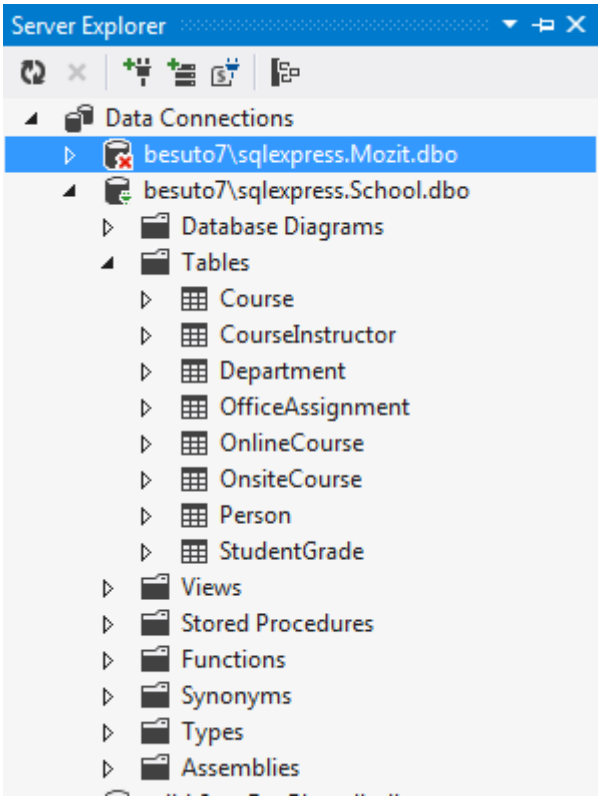


Figura 4. Visão do Server Explorer

Na **Figura 4** temos uma aba do Server Explorer mostrando conexões e uma delas está conectada, a que liga ao banco School no SQL Server Express.

Essa representação é construída com dois arquivos: um .cs com o código das classes representadas e um código XML com o design visual representando as posições e características visuais.

Depois de mapear todas as tabelas desse exemplo, o banco School, podemos conectar e usar LINQ e não mais SQL para trabalhar com a base de dados, como demonstrado no exemplo da **Listagem 7**.

Listagem 7. Listando as pessoas da base de dados

```
DSchoolDataContext db = new DSchoolDataContext();
              var Pessoas = (from pessoas in db.Persons select pessoas).ToList();
              foreach(var pessoa in Pessoas)
              {
                  Console.WriteLine(pessoa.FirstName);
              }
```

Observando a primeira linha da **Listagem 7**, podemos ver que o código cria a instancia de um objeto do tipo DschoolDataContext. Essa classe é a classe que vai representar nossa base de dados. Ela abre conexão com o banco de dados e cuida do ciclo de vida das nossas classes baseadas em tabelas. Uma instancia dela é sempre necessária para manipularmos os dados. Na **Listagem 8** temos a mesma consulta com sintaxe de método:

Listagem 8. Listando as pessoas da base de dados com sintaxe de método

```
DSchoolDataContext db = new DSchoolDataContext();
              var Pessoas = db.Persons.ToList();

              foreach (var pessoa in Pessoas)
              {
                  Console.WriteLine(pessoa.FirstName);
              }
```

Como estamos listando todos os itens, sem condicional (where), só o método ToList basta.

O método ToList é um dos métodos para trazer o resultado das querys, ele faz a consulta retornar uma lista genérica. Outro método muito usado é o First que retorna somente o primeiro item buscado.

Além de consultarmos, também podemos inserir dados na base de dados através da LINQ to SQL, bastando para isso utilizarmos o método InsertOnSubmit de nosso contexto, como mostra a linha 7 da **Listagem 9**.

Listagem 9. Incluindo um item com LINQ to SQL

```
01     DSchoolDataContext db = new DSchoolDataContext();
  02     Person pessoa = new Person();
  03     Console.WriteLine("Digite um primeiro nome");
  04     pessoa.FirstName = Console.ReadLine();
  05     Console.WriteLine("Digite um último nome");
  06     pessoa.LastName = Console.ReadLine();
  07     db.Persons.InsertOnSubmit(pessoa);
  08     db.SubmitChanges();
```

A diferença aqui foi que criamos uma instancia de pessoa, adicionamos valores a ela depois incluímos ela no contexto para por fim submetermos as mudanças.

O método InsertOnSubmit pega o item que passarmos e adiciona ao contexto que fica em memória. Nesse momento nada irá ao banco de dados, somente quando as mudanças forem submetidas.

A alteração de dados consiste em buscarmos um item já existente no banco. Alterar uma propriedade e depois submeter as alterações, como ilustrado na **Listagem 10**.

Listagem 10. Exemplo de alteração com LINQ to SQL

```
DSchoolDataContext db = new DSchoolDataContext();
              var Pessoa = db.Persons.Where(p => p.LastName.Equals("Machado")).FirstOrDefault();
              Pessoa.FirstName = "Luiza";
              db.SubmitChanges();
```

O contexto verifica as mudanças ocorridas nas classes e envia a mesma ao banco de dados quando executado o método SubmitChanges.

A Exclusão segue o mesmo principio, onde primeiro precisamos buscar um item existente, para então indicarmos que ele será deletado e submetermos as mudanças, como mostrado na **Listagem 11**.

Listagem 11. Exemplo de alteração com LINQ to SQL

```
DSchoolDataContext db = new DSchoolDataContext();
      var Pessoa = db.Persons.Where
      (p => p.LastName.Equals("Machado")).FirstOrDefault();
  db.Persons.DeleteOnSubmit(Pessoa);
  db.SubmitChanges();
```

LINQ to XML

Outra das primeiras implementações da LINQ foi a LINQ to XML. No .NET há várias maneiras de se trabalhar com XML, maneiras na qual você precisa manipular cada nó do XML e apesar de não ser difícil não é tão rápido de se usar. A LINQ to XML, ou XLINQ, traz facilidade a esse manuseio. Para compreendermos e exemplificarmos, vamos considerar o XML de exemplo da **Listagem 12** como sendo nossa fonte de dados.

Listagem 12. XML de exemplo

```
<?xml version="1.0" encoding="utf-8" ?>
  <escola>
    <aluno>
      <id>1</id>
      <nome>Priscila</nome>
      <sobrenome>Sato</sobrenome>
    </aluno>
    <aluno>
      <id>2</id>
      <nome>Lucas</nome>
      <sobrenome>Machado</sobrenome>
    </aluno>
    <aluno>
      <id>3</id>
      <nome>Ricardo</nome>
      <sobrenome>Coelho</sobrenome>
    </aluno>
    <aluno>
      <id>4</id>
      <nome>André</nome>
```

```
      <sobrenome>Farias</sobrenome>
    </aluno>
  </escola>
```

Vamos agora entender como a LINQ to XML realiza a leitura deste XML: escola é o nome da nossa base, nosso agrupamento de dados, cada aluno é um item dessa base. Imaginando um XML como uma base de dados, conseguimos agora visualizar como seriam as querys.

Listagem 13. Uma consulta com LINQ to XML

```
XElement xml = XElement.Load(@"..\..\Escola.xml");

             var alunos = from a in xml.Elements("aluno")
                          select a;

             foreach (var a in alunos)
             {
                 Console.WriteLine(a.Element("nome").Value);
             }
```

A primeira linha da **Listagem 13** inicializa o contexto do XML. A classe XElement é responsável por carregar o XML para memória. Um pouco diferente de LINQ to SQL temos aqui o **Element**, responsável por achar nós do XML. Por isso na nossa query indicamos que queremos os elementos com nome “aluno” dentro do contexto na memória. Na hora de resgatar um valor usamos o atributo Value.

Se quisermos fazer um where podemos continuar a usar do Element para indicar qual nó fará parte da condicional. Vejamos um exemplo simples na **Listagem 14**.

Listagem 14. Exemplo de condicional where

```
var alunos = from a in xml.Elements("aluno")
                         where a.Element("nome").Value.Equals("Priscila")
                         select a;
```

E usando Element conseguimos fazer as operações normais extendendo a LINQ, como ordenação e paginação. Você também pode usar XAtribute que procura um atributo dentro do nó atual do XML. É útil quando estamos lidando com elementos do XML que trabalham usando uma única linha e não um grupo como no exemplo anterior.

Também é possível adicionarmos um novo nó ao XML. O conceito é um pouco diferente onde precisamos selecionar o elemento container que vai receber um novo elemento, para então adicionarmos o mesmo. Na **Listagem 15** temos um exemplo onde inserimos um novo aluno no elemento escola.

Listagem 15. Adicionando um novo elemento

```
XElement xml = XElement.Load(@"..\..\Escola.xml");
              xml.FirstNode.AddAfterSelf(new XElement("aluno", new XElement("id", "5"),
              new XElement("nome", "Ricardo"), new XElement("sobrenome", "Pontes")));
              xml.Save(@"..\..\Escola.xml");
```

Se você preferir consultar um nó (elemento) antes de adicionar um elemento filho também é possível, neste caso atente-se em usar o método Add e não AddAfterSelf.

Se observarmos no código da **Listagem 15** iniciamos um novo elemento e dentro dele passamos como parametro novos elementos. Se o nó fosse de uma única linha poderíamos criar atributos, e não elementos.

Para editar um elemento podemos pesquisar por ele e usar o SetElementValue ou SetAtributeValue para editar um nó dele ou um atributo. Veja um exemplo desta edição na **Listagem 16**.

Listagem 16. Editando um elemento

```
XElement xml = XElement.Load(@"..\..\Escola.xml");
  XElement aluno = xml.Elements("aluno").Where(e =>
  e.Element("nome").Value.Equals("Priscila")).First();
  aluno.SetElementValue("nome", "Priscila Mayumi");
  xml.Save(@"..\..\Escola.xml");
```

No exemplo da **Listagem 16** nós selecionamos um elemento com determinado nome, usando sintaxe de método, e usamos SetElementValue pois “nome” é um nó dentro de “aluno”. Por fim salvamos o documento com o método Save. Para excluir um elemento o procedimento é parecido, como mostrado na **Listagem 17**.

Listagem 17. Excluindo um elemento

```
XElement xml = XElement.Load(@"..\..\Escola.xml");
  xml.Elements("aluno").Where(e =>
  e.Element("sobrenome").Value.Equals("Pontes")).First().Remove();
  xml.Save(@"..\..\Escola.xml");
```

Para remover elementos não precisamos pesquisar por eles primeiro, sendo necessário apenas indicarmos a condição direto ao contexto xml.

Outras implementações da LINQ

LINQ to Entities

A LINQ to Entities é a implementação especifica para lidar com entidades do Entity Framework, muito similar a LINQ to SQL. A primeira diferença que se nota é a classe de contexto, no Entity

Framework temos a DBContext. Outras mudanças são referentes aos métodos, como o método que indica que as alterações podem ser persistidas no banco de dados é o SaveChanges.

Como implementação ao Entity Framework a LINQ to Entities veio somente no Service Pack do Visual Studio 2008, quando o Entity Framework foi lançado, sendo a LINQ to SQL existente já desde o lançamento o VS 2008.

Por ser mais performático e por continuar a ser atualizado constantemente, usar LINQ to Entites (é claro, dentro do Entity Framework) torna-se mais interessante do que LINQ to SQL.

LINQ to NHibernate

NHibernate é um framework de mapeamento objeto relacional muito usado na comunidade .NET. Mantido por uma grande comunidade, ele ganhou uma implementação da LINQ para ser usada em suas querys. Podemos ver um exemplo na **Listagem 18**, onde podemos observar que ele não foge ao padrão do LINQ, com exceção do uso de sua session para execução da instrução LINQ.

Listagem 18. Exemplo de LINQ to NHibernate

```
ISession session = getSession();
              var query = from palestra in session.Linq<Palestras>()
                          where palestra.Id == id
                          select palestra;
```

LINQ em outras plataformas

A LINQ foi portada para outras plataformas por meio de comunidades open source. Dentre algumas implementações ou mecanismos baseados na LINQ podemos citar a asq, feita para python e declarada como baseada na LINQ.

Conclusão

A LINQ é um linguagem muito poderosa e prática. Hoje existem ports da LINQ para outras linguagens, todos buscando a facilidade de lidar com conjuntos de dados.

O uso da LINQ não causa nenhum problema de desempenho, lembrando que quase sempre ela terá o mesmo código IL que uma expressão lambda. Sendo assim é um ótimo recurso a ser usado no dia a dia

Links

O Projeto LINQ
http://msdn.microsoft.com/pt-br/library/bb308959.aspx

Creating the School Sample Database
http://msdn.microsoft.com/en-us/library/vstudio/bb399731(v=vs.100).aspx

Microsoft 101 Linq Exemples

http://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b