عنوان: **آموزش BrightStarDb (قسمت اول)**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۳/۰۱/۲۸ ۰:۲۵
آدرس: www.dotnettips.info
گروه‌ها: NoSQL, BrightstarDB

در طی این پست ها با مفاهیم NoSql آشنا شدید. همچنین در این دوره مفاهیم و مبانی RavenDb (یکی از بی نقص‌ترین دیتابیس‌های NoSql) بررسی شد. اما قرار است در طی چند پست با یکی دیگر از انواع دیتابیس‌های NoSql طراحی شده برای دات نت به نام BrightStarDb یا به اختصار B*Db آشنا شویم.

*در دنیای NoSql پیاده سازی‌های متفاوتی از دیتابیس‌ها انجام شده است و هر دیتابیس، ویژگی‌ها و مزایا و معایب خاص خودش را دارد. باید قبول کرد که همیشه و همه جا نمی‌توان از یک پایگاه داده NoSql مشخص استفاده نماییم (دلایلی نظیر محدودیت‌های License، هزینه پیاده سازی و...). در نتیجه بررسی یک دیتابیس دیگر با شرایط و توانمندی‌های خاص آن خالی از سود نیست.
از ویژگی مهم این دیتابیس می‌توان به عناوین زیر اشاره کرد.
» این دیتابیس در گروه **Graph databasesها** قرار دارد و از زبان SPARQL (بخوانید Sparkle) برای کوئری گرفتن و کار با داده‌ها بهره می‌برد؛
» متن باز و رایگان است
» پشتیبانی از دات نت چهار به بعد؛
» قابل استفاده در Mobile Application - Windows phone 7 , 8؛
» بدون شما (Schema Less) و با قابلیت ذخیره سازی triple و به فرمت RDF
» پشتیبانی از Linq و OData؛
» پشتیبانی از تراکنش‌ها ؛
» پیاده سازی مدل برنامه به صورت Code First؛
» سرعت بالا جهت ذخیره سازی و لود اطلاعات؛
» نیاز به پیکربندی‌های خاص جهت پیاده سازی ندارد؛
» ارائه افزونه رایگان Polaris جهت کوئری گفتن و نمایش Visual داده ها.
و غیره که در ادامه با آن‌ها آشنا خواهید شد.

در B*Db دو روش برای ذخیره سازی اطلاعات وجود دارد:
» **Append Only** : در این روش تمامی تغییرات (عملیات نوشتن) در انتهای فایل index اضافه خواهد شد. این روش مزایای زیر را به دنبال خواهد داشت:
عملیات نوشتن هیچگاه عملیات خواندن اطلاعات را block نمی‌کند. در نتیجه هر تعداد عملیات خواندن فایل (منظور اجرای کوئری‌های SPQRL است) می‌تواند به صورت موازی بر روی Storeها اجرا شود.

به دلیل اینکه عمل ویرایش واقعی هیچ گاه انجام نمی‌شود (داده‌ها فقط اضافه خواهند شد) همیشه می‌توانید وضعیت Store را به حالت‌های قبلی بازگردانید.
عملیات نوشتن اطلاعات بسیار سریع خواهد بود.

از معایب این روش این است که حجم Storeها فقط با افزایش داده‌ها زیاد نمی‌شود، بلکه با هر عملیات ویرایش نیز حجم فایل‌های Store افزایش پیاده خواهد کرد. در نتیجه از این روش فقط زمانی که از نظر فضای هارد دیسک محدودیت ندارید استفاده کنید(روش پیش فرض در B*Db نیز همین حالت است)

» **Rewritable** : در این روش در هنگام اجرای عملیات نوشتن، ابتدا یک کپی از اطلاعات گرفته میشود. سپس داده‌های مورد نظر به کپی گرفته شده اعمال می‌شوند. تا زمانیکه عملیات نوشتن اطلاعات به پایان نرسد، هر گونه دسترسی به اطلاعات جهت عملیات Read بر روی داده اصلی اجرا می‌شود. با استفاده از این روش عملیات Read و Write هیچ گونه تداخلی با هم نخواهند داشت.

(چیزی شبیه به ^)

نکته ای که باید به آن دقت داشت این است که فقط در هنگام ساخت Storeها می‌توانید نوع ذخیره سازی آن را تعیین نمایید، بعد از ساخت Store امکان سوئیچ بین حالات امکان پذیر نیست.

همان طور که پیشتر گفته شد B*Db برای ذخیره سازی اطلاعات از سند RDF بهره می‌برد. البته با RDF Syntaxهای متفاوت :

RDF Syntax	File Extension (uncompressed)	File Extension (GZip compressed)
NTriples	.nt	.nt.gz
NQuads	.nq	.nq.gz
RDF/XML	.rdf	.rdf.gz
Turtle	.ttl	.ttl.gz
RDF/JSON	.rj or .json	.rj.gz or .json.gz

هم چنین در B*Db چهار روش برای دست یابی با داده‌ها (پیاده سازی عملیات CRUD) وجود دارد از قبیل:

« B*Db EntityFramewok

« Data Object Layer

« RDF Client Api

« Dynamic API

که هر کدام در طی پست‌های متفاوت بررسی خواهد شد.

بررسی یک مثال با روش B*Db EntityFramework

برای شروع ابتدا یک پروژه جدید از نوع Console Application ایجاد کنید. سپس از طریق Nuget اقدام به نصب Package زیر نمایید:

```
pm> install-Package BirghtStarDb
```

پکیج بالا تمام کتابخانه‌های لازم جهت کار با B*Db را شامل می‌شود. اگر قصد ندارید از افزونه‌های مربوط به EntityFramework و Code First استفاده نمایید می‌توانید Package زیر را نصب نمایید:

```
PM> Install-Package BirghtStarDbLibs
```

این پکیج فقط شامل کتابخانه‌های لازم جهت کار با استفاده از SPRQL است.
بعد از نصب پکیج‌های بالا یک فایل Text Template با نام MyEntityContext.tt نیز به پروژه افزوده خواهد شد. این فایل جهت تولید خودکار مدل‌های برنامه استفاده می‌شود. اما برای این کار لازم است به ازای هر مدل ابتدا یک اینترفیس ایجاد نمایید. برای مثال:

```
[Entity]
    public interface IBook
    {
        public int Code { get; set; }
        public string Title { get; set; }
```

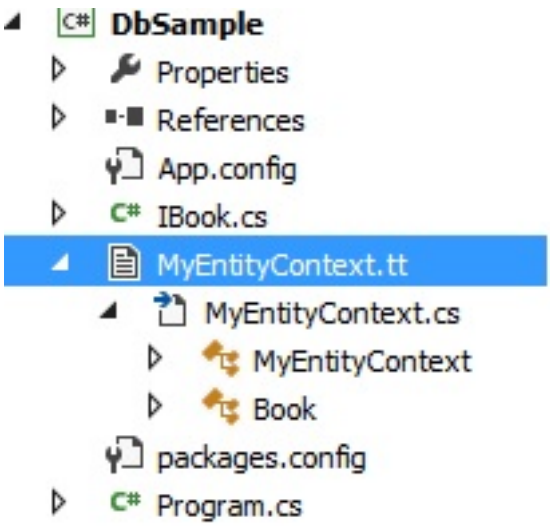
```
    }
```

نکته:

» نیاز به ایجاد یک خاصیت به عنوان Id وجود ندارد. به صورت پیش فرض خاصیت Id با نوع string برای هر مدل پیاده سازی می‌شود. اما اگر قصد دارید این نام خاصیت را تغییر دهید می‌توانید به صورت زیر عمل کنید:

```
[Entity]
    public interface IBook
    {
        [Identifier]
        public string MyId { get;  }
        public int Code { get; set; }
        public string Title { get; set; }
    }
```

در مثال بالا خاصیت MyId به جای خاصیت Id در نظر گرفته می‌شود. مزین شدن با Identifier و همچنین نداشتن متد set را فراموش نکنید. بعد از ایجاد اینترفیس مورد نظر و اجرای Run Custom Tool بر روی فایل Text Template.tt کلاسی به نام Book به صورت زیر ساخته می‌شود:



استفاده از اینترفیس برای ساخت مدل انعطاف پذیری بالایی را در اختیار ما قرار می‌دهد که بعدا مفصل بحث خواهد شد. برای عملیات درج داده می‌توان به صورت زیر عمل کنید:

```
MyEntityContext context = new
MyEntityContext("type=embedded;storesdirectory=c:\brightstar;storename=test");
            var book = context.Books.Create();
            book.Code = 1;
            book.Title = "Test";

            context.Books.Add(book);

            context.SaveChanges();
```

با یک نگاه می‌توان به شباهت مدل پیاده سازی شده بالا به EntityFramework پی برد. اما نکته مهم در مثال بالا ConnectionString پاس داده شده به Context پروژه است. در B*Db چهار روش برای دستیابی به اطلاعات ذخیره شده وجود دارد:

»embedded : در این حالت از طریق آدرس فیزیکی فایل مورد نظر می‌توان یک Connection ایجاد کرد.

»rest : یا استفاده از HTTP یا HTTPS می‌توان به سرویس B*Db دسترسی داشت.
»dotNetRdf : برای ارتباط با یک Store دیگر با استفاده از اتصال دهنده‌های DotNetRDf.
»sparql : اتصال به منبع داده ای دیگر با استفاده از پروتکل SPARQL
در هنگام ایجاد اتصال باید نوع مورد نظر را از حتما تعیین نمایید. با استفاده از storedirctory مکان فیزیکی فایل تعیین خواهد شد.

عنوان: **پیاده سازی UnitOfWork برای BrightStarDb**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۳/۰۲/۲۶ ۲۰:۰
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, NoSQL, UnitOfWork, BrightstarDB

در این پست با BrightStarDb و مفاهیم اولیه آن آشنا شدید. همان طور که پیش‌تر ذکر شد BrightStarDb از تراکنش‌ها جهت ذخیره اطلاعات پشتیبانی می‌کند. قصد داریم روش شرح داده شده در اینجا را بر روی BrightStarDb فعال کنیم. ابتدا بهتر است با روش ساخت مدل در B*Db آشنا شویم.
*یکی از پیش نیازهای این پست مطالعه این دو مطلب (^) و (^) می‌باشد.
فرض می‌کنیم در دیتابیس مورد نظر یک Store به همراه یک جدول به صورت زیر داریم:

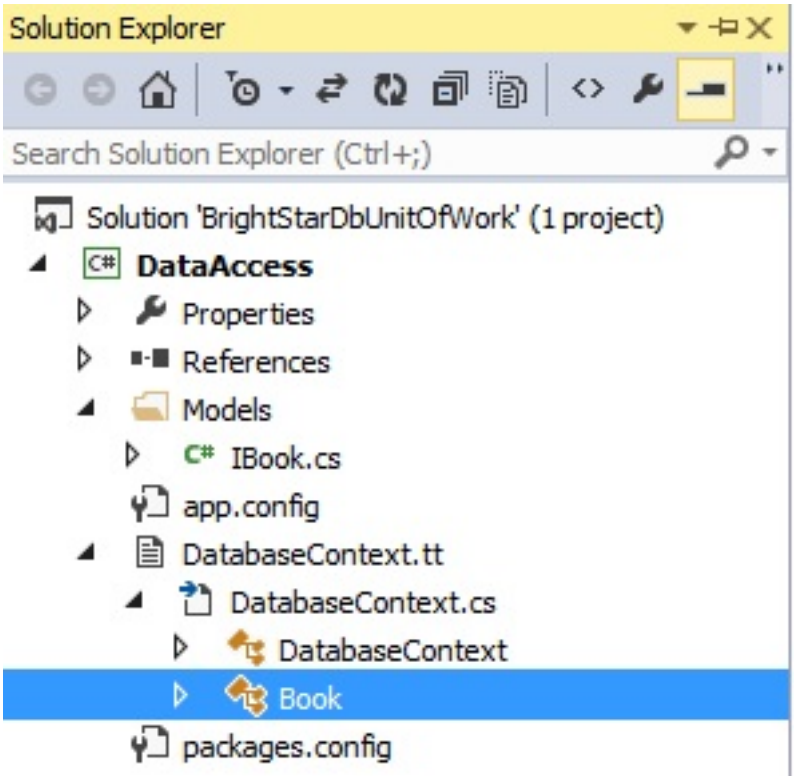
```
[Entity]
    public interface IBook
    {
        [Identifier]
        string Id { get; }

        string Title { get; set; }

        string Isbn { get; set; }
    }
```

بر روی پروژه مورد نظر کلیک راست کرده و گزینه Add new Item را انتخاب نمایید. از برگه Data گزینه BrightStar Entity Context را انتخاب کنید

بعد از انتخاب گزینه بالا یک فایل با پسوند tt به پروژه اضافه خواهد شد که وظیفه آن جستجو در اسمبلی مورد نظر و پیدا کردن تمام اینترفیس هایی که دارای EntityAttribute هستند و همچنین ایجاد کلاس‌های متناظر جهت پیاده سازی اینترفیس‌های بالا است. در نتیجه ساختار پروژه تا این جا به صورت زیر خواهد شد.



واضح است که فایلی به نام Book به عنوان پیاده سازی مدل IBook به عنوان زیر مجموعه فایل DatabaseContext.tt به پروژه اضافه شده است.

تا اینجا برای استفاده از Context مورد نظر باید به صورت زیر عمل نمود:

```
DatabaseContext context = new DatabaseContext();
  context.Books.Add(new Book());
```

Context پیش فرض ساخته شده توسط B*Db از Generic DbSetهای معادل EF پشتیبانی نمی‌کند و از طرفی IUnitOfWork مورد نظر به صورت زیر است

```
public interface IUnitOfWork
    {
        BrightstarEntitySet<T> Set<T>() where TEntity : class;
        void DeleteObject(object obj);
         void SaveChanges();
    }
```

در اینجا فقط به جای IDbSet از BrightStarDbSet استفاده شده است. همان طور که در این مقاله توضیح داده شده است، برای پیاده سازی مفهوم UnitOfWork؛ نیاز است تا کلاس DatabaseContext که نماینده BrightStarDbContext پروژه است، از اینترفیس IUnitOfWork طراحی شده ارث بری کند. جهت انجام این مهم و همچنین جهت اضافه کردن قابلیت ایجاد Generic DbSetها نیز باید کمی در فایل Template Generator تغییر ایجاد نماییم. این تغییرات را قبلا در طی یک پروژه ایجاد کرده‌ام و شما می‌توانید آن را از اینجا دریافت کنید. بعد از دانلود کافیست فایل DatabaseContext.tt مورد نظر را در پروژه خود کپی کرده و گزینه Run Custom Tools را فراخوانی نمایید.

نکته: برای حذف یک آبجکت از Store، باید از متد DeleteObject تعبیه شده در Context استفاده نماییم. در نتیجه متد مورد نظر نیز در اینترفیس بالا در نظر گرفته شده است.

استفاده از IOC Container جهت رجیستر کردن IUnitOfWrok

در این قدم باید IUnitOfWork را در یک IOC container رجیستر کرده تا در جای مناسب عملیات وهله سازی از آن میسر باشد.
من در اینجا از Castle Windsor Container استفاده کردم. کلاس زیر این کار را برای ما انجام خواهد داد:

```
public class DependencyResolver
    {
        public static void Resolve(IWindsorContainer container)
        {
            var context = new
DatabaseContext("type=embedded;storesdirectory=c:\brightstar;storename=test ");
            container.Register(Component.For<IUnitOfWork>().Instance(context).LifestyleTransient());
        }
    }
```

حال کافیست در کلاس‌های سرویس برنامه UnitOfWork رجیستر شده را به سازنده آن‌ها تزریق نماییم.

```
public class BookService
    {
        public BookService(IUnitOfWork unitOfWork)
        {
            UnitOfWork = unitOfWork;
        }

        public IUnitOfWork UnitOfWork
        {
            get;
            private set;
        }

        public IList<IBook> GetAll()
        {
            return UnitOfWork.Set<IBook>().ToList();
        }

        public void Add()
        {
            UnitOfWork.Set<IBook>().Add(new Book());
        }

        public void Remove(IBook entity)
        {
            UnitOfWork.DeleteObject(entity);
        }
    }
```

سایر موارد دقیقا معادل مدل EF آن است.
نکته: در حال حاضر امکان جداسازی مدل‌های برنامه (تعاریف اینترفیس) در قالب یک پروژه دیگر(نظیر مدل CodeFirst در EF) در B*Db امکان پذیر نیست.
نکته : برای اضافه کردن آیتم جدید به Store نیاز به وهله سازی از اینترفیس IBook داریم. کلاس Book ساخته شده توسط DatabaseContext.tt در عملیات Insert و update کاربرد خواهد داشت.