عنوان: **شروع به کار با RavenDB**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۱۴ ۱۳:۵۳
آدرس: www.dotnettips.info
گروه‌ها: NoSQL, RavenDB

پیشنیازهای بحث

- مروری بر مفاهیم مقدماتی NoSQL
- رده‌ها و انواع مختلف بانک‌های اطلاعاتی NoSQL
- چه زمانی بهتر است از بانک‌های اطلاعاتی NoSQL استفاده کرد و چه زمانی خیر؟

لطفا یکبار این پیشنیازها را پیش از شروع به کار مطالعه نمائید؛ چون بسیاری از مفاهیم پایه‌ای و اصطلاحات مرسوم دنیای NoSQL در این سه قسمت بررسی شده‌اند و از تکرار مجدد آن‌ها در اینجا صرفنظر خواهد شد.

RavenDB چیست؟

RavenDB یک بانک اطلاعاتی سورس باز NoSQL سندگرای تهیه شده با دات نت است. ساختار کلی بانک‌های اطلاعاتی NoSQL سندگرا، از لحاظ نحوه ذخیره سازی اطلاعات، با بانک‌های اطلاعاتی رابطه‌ای متداول، کاملا متفاوت است. در اینگونه بانک‌های اطلاعاتی، رکوردهای اطلاعات، به صورت اشیاء JSON ذخیره می‌شوند. اشیاء JSON یا JavaScript Object Notation بسیار شبیه به anonymous objects سی شارپ هستند. JSON روشی است که توسط آن JavaScript اشیاء خود را معرفی و ذخیره می‌کند. به عنوان رقیبی برای XML مطرح است؛ نسبت به XML اندکی فشرده‌تر بوده و عموما دارای اسکیمای خاصی نیست و در بسیاری از اوقات تفسیر المان‌های آن به مصرف کننده واگذار می‌شود.
در JSON عموما سه نوع المان پایه مشاهده می‌شوند:
- اشیاء که به صورت {object} تعریف می‌شوند.
- مقادیر "key":"value" که شبیه به نام خواص و مقادیر آن‌ها در دات نت هستند.
- و آرایه‌ها به صورت [array]
همچنین ترکیبی از این سه عنصر یاد شده نیز همواره میسر است. برای مثال، یک key مشخص می‌تواند دارای مقداری حاوی یک آرایه یا شیء نیز باشد.

```
JSON: JavaScript Object Notation

document :{
   key: "Value",
   another_key: {
      name: "embedded object"
   },
   some_date: new Date(),
   some_number: 12
}

C# anonymous object

var Document = new {
   Key= "Value",
   AnotherKey= new {
      Name = "embedded object"
   },
   SomeDate = DateTime.Now(),
   SomeNumber = 12
};
```

به این ترتیب می‌توان به یک ساختار دلخواه و بدون اسکیما، از هر سند به سند دیگری رسید. اغلب بانک‌های اطلاعاتی سندگرا، اینگونه اسناد را در زمان ذخیره سازی، به یک سری binary tree تبدیل می‌کنند تا تهیه کوئری بر روی آن‌ها بسیار سریع شود. مزیت دیگر استفاده از JSON، سادگی و سرعت بالای Serialize و Deserialize اطلاعات آن برای ارسال به کلاینت‌ها و یا دریافت آن‌ها است؛ به همراه فشرده‌تر بودن آن نسبت به فرمت‌های مشابه دیگر مانند XML.

یک نکته مهم

اگر پیشنیازهای بحث را مطالعه کرده باشید، حتما بارها با این جمله که دنیای NoSQL از تراکنش‌ها پشتیبانی نمی‌کند، برخورد داشته‌اید. این مطلب در مورد RavenDB صادق نیست و این بانک اطلاعاتی NoSQL خاص، از تراکنش‌ها پشتیبانی می‌کند. RavenDB در Document store خود ACID عملکرده و از تراکنش‌ها پشتیبانی می‌کند. اما تهیه ایندکس‌های آن بر مبنای مفهوم عاقبت یک دست شدن عمل می‌کند.

مجوز استفاده از RavenDB

هرچند مجموعه سرور و کلاینت RavenDB سورس باز هستند، اما این مورد به معنای رایگان بودن آن نیست. مجوز استفاده از RavenDB نوع خاصی به نام AGPL است. به این معنا که یا کل کار مشتق شده خود را باید به صورت رایگان و سورس باز ارائه دهید و یا اینکه مجوز استفاده از آن‌را برای کارهای تجاری بسته خود خریداری نمائید. نسخه استاندارد آن نزدیک به هزار دلار است و نسخه سازمانی آن نزدیک به 2800 دلار به ازای هر سرور.

شروع به نوشتن اولین برنامه کار با RavenDB

ابتدا یک پروژه کنسول ساده را آغاز کنید. سپس کلاس‌های مدل زیر را به آن اضافه نمائید:

```
using System.Collections.Generic;

namespace RavenDBSample01.Models
{
    public class Question
    {
        public string By { set; get; }
        public string Title { set; get; }
        public string Content { set; get; }

        public List<Comment> Comments { set; get; }
        public List<Answer> Answers { set; get; }

        public Question()
        {
            Comments = new List<Comment>();
            Answers = new List<Answer>();
        }
    }
}

namespace RavenDBSample01.Models
{
    public class Comment
    {
        public string By { set; get; }
        public string Content { set; get; }
    }
}

namespace RavenDBSample01.Models
{
    public class Answer
    {
        public string By { set; get; }
        public string Content { set; get; }
    }
}
```

سپس به کنسول پاور شل نیوگت در ویژوال استودیو مراجعه کرده و دستورات ذیل را جهت افزوده شدن وابستگی‌های مورد نیاز RavenDB، صادر کنید:

```
PM> Install-Package RavenDB.Client
PM> Install-Package RavenDB.Server
```

به این ترتیب بسته‌های کلاینت (مورد نیاز جهت برنامه نویسی) و سرور RavenDB به پروژه جاری اضافه خواهند شد (نگارش 2.5

در زمان نگارش این مطلب؛ جمعا نزدیک به 75 مگابایت).
اکنون به پوشه packages\RavenDB.Server.2.5.2700\tools مراجعه کرده و برنامه Raven.Server.exe را اجرا کنید تا سرور RavenDB شروع به کار کند. این سرور به صورت پیش فرض بر روی پورت 8080 اجرا می‌شود. از این جهت که در RavenDB نیز همانند سایر Document Stores مطرح، امکان دسترسی به اسناد از طریق REST API و Urlها وجود دارد.
البته لازم به ذکر است که RavenDB در 4 حالت برنامه کنسول (همین سرور فوق)، نصب به عنوان یک سرویس ویندوز NT، هاست شدن در IIS و حالت مدفون شده یا Embedded قابل استفاده است.

خوب؛ همین اندازه برای برپایی اولیه RavenDB کفایت می‌کند.

```
using Raven.Client.Document;
using RavenDBSample01.Models;

namespace RavenDBSample01
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var store = new DocumentStore
                               {
                                   Url = "http://localhost:8080"
                               }.Initialize())
            {
                using (var session = store.OpenSession())
                {
                    session.Store(new Question
                    {
                        By = "users/Vahid",
                        Title = "Raven Intro",
                        Content = "Test...."
                    });

                    session.SaveChanges();
                }
            }
        }
    }
}
```

اکنون کدهای برنامه کنسول را به نحو فوق برای ذخیره سازی اولین سند خود، تغییر دهید.
کار با ایجاد یک DocumentStore که به آدرس سرور اشاره می‌کند و کار مدیریت اتصالات را برعهده دارد، شروع خواهد شد. اگر نمی‌خواهید Url را درون کدهای برنامه مقدار دهی کنید، می‌توان از فایل کانفیگ برنامه نیز برای این منظور کمک گرفت:

```
<connectionStrings>
   <add name="ravenDB" connectionString="Url=http://localhost:8080"/>
</connectionStrings>
```

در این حالت باید خاصیت ConnectionStringName شیء DocumentStore را مقدار دهی نمود.
سپس با ایجاد Session در حقیقت یک Unit of work آغاز می‌شود که درون آن می‌توان انواع و اقسام دستورات را صادر نمود و سپس در پایان کار، با فراخوانی SaveChanges، این اعمال ذخیره می‌گردند. در RavenDB یک سشن باید طول عمری کوتاه داشته باشد و اگر تعداد عملیاتی که در آن صادر کرده‌اید، زیاد است با خطای زیر متوقف خواهید شد:

```
The maximum number of requests (30) allowed for this session has been reached.
```

البته این نوع محدودیت‌ها عمدی است تا برنامه نویس به طراحی بهتری برسد.

در یک برنامه واقعی، ایجاد DocumentStore یکبار در آغاز کار برنامه باید انجام گردد. اما هر سشن یا هر واحد کاری آن، به ازای تراکنش‌های مختلفی که باید صورت گیرند، بر روی این DocumentStore، ایجاد شده و سپس بسته خواهند شد. برای مثال در یک برنامه ASP.NET، در فایل Global.asax در زمان آغاز برنامه، کار ایجاد DocumentStore انجام شده و سپس به ازای هر درخواست رسیده، یک سشن RavenDB ایجاد و در پایان درخواست، این سشن آزاد خواهد شد.

برنامه را اجرا کنید، سپس به کنسول سرور RavenDB که پیشتر آن‌را اجرا نمودیم مراجعه نمائید تا نمایی از عملیات انجام شده را بتوان مشاهده کرد:

```
Raven is ready to process requests. Build 2700, Version 2.5.0 / 6dce79a Server started in 14,438 ms
Data directory: D:\Prog\RavenDBSample01\packages\RavenDB.Server.2.5.2700\tools\Database\System
HostName: <any> Port: 8080, Storage: Esent
Server Url: http://localhost:8080/
Available commands: cls, reset, gc, q
Request #   1: GET     -   514 ms - <system>   - 404 - /docs/Raven/Replication/Destinations
Request #   2: GET     -   763 ms - <system>   - 200 -
/queries/?&id=Raven%2FHilo%2Fquestions&id=Raven%2FServerPrefixForHilo
Request #   3: PUT     -   185 ms - <system>   - 201 - /docs/Raven/Hilo/questions
Request #   4: POST    -   103 ms - <system>   - 200 - /bulk_docs
        PUT questions/1
```

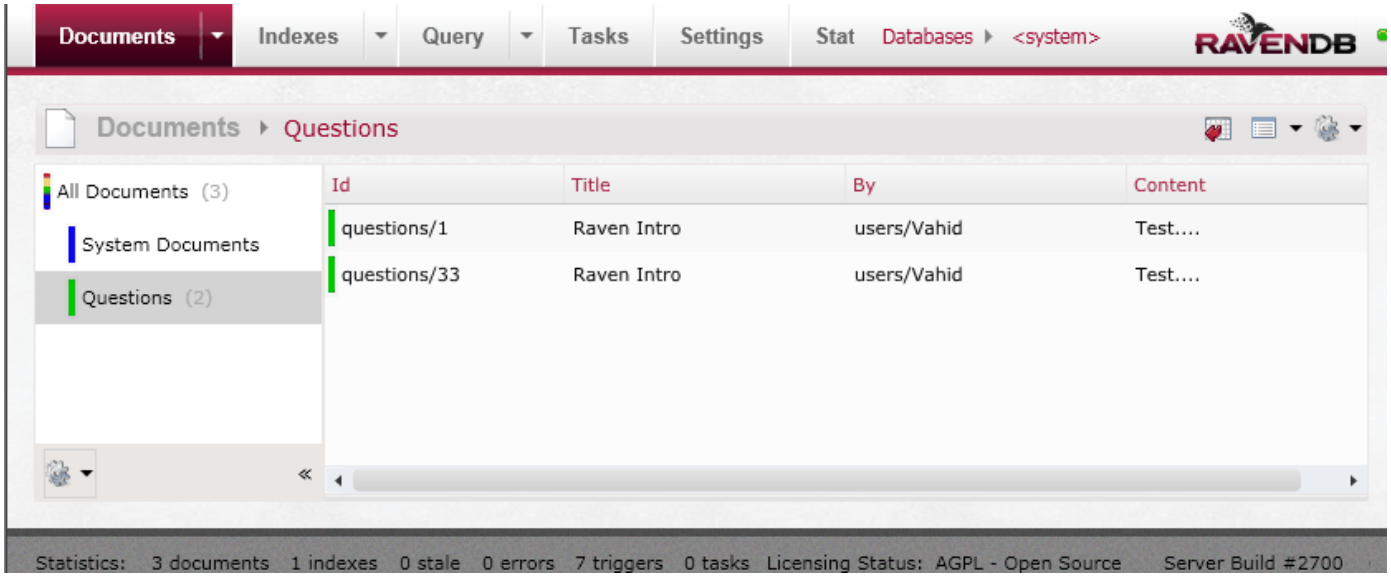
زمانیکه سرور RavenDB در حالت دیباگ در حال اجرا باشد، لاگ کلیه اعمال انجام شده را در کنسول آن می‌توان مشاهده نمود.
همانطور که مشاهده می‌کنید، یک کلاینت RavenDB با این بانک اطلاعاتی با پروتکل HTTP و یک REST API ارتباط برقرار می‌کند. برای نمونه، کلاینت در اینجا با اعمال یک HTTP Verb خاص به نام PUT، اطلاعات را درون بانک اطلاعاتی ذخیره کرده است. تبادل اطلاعات نیز با فرمت JSON انجام می‌شود.
عملیات PUT حتما نیاز به یک Id از پیش مشخص دارد و این Id، پیشتر در سطری که Hilo در آن ذکر شده (یکی از الگوریتم‌های محاسبه Id در RavenDB)، محاسبه گردیده است. برای نمونه در اینجا الگوریتم Hilo مقدار "questions/1" را به عنوان Id محاسبه شده بازگشت داده است.
در سطری که عملیات Post به آدرس bulk_docs سرور ارسال گردیده است، کار ارسال یکباره چندین شیء JSON برای کاهش رفت و برگشت‌ها به سرور انجام می‌شود.

و برای کوئری گرفتن مقدماتی از اطلاعات ثبت شده می‌توان نوشت:

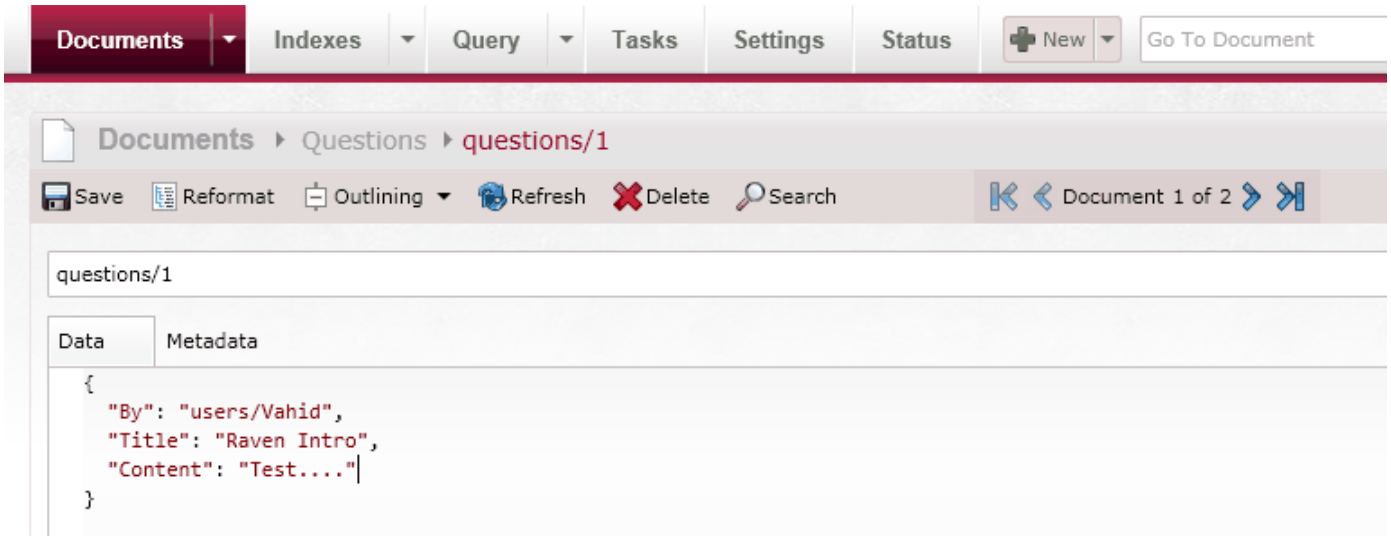
```
using (var session = store.OpenSession())
{
  var question1 = session.Load<Question>("questions/1");
  Console.WriteLine(question1.By);
}
```

نگاهی به بانک اطلاعاتی ایجاد شده

در همین حال که سرور RavenDB در حال اجرا است، مرورگر دلخواه خود را گشوده و سپس آدرس http://localhost:8080 را وارد نمائید. بلافاصله، کنسول مدیریتی تحت وب این بانک اطلاعاتی که با سیلورلایت نوشته شده است، ظاهر خواهد شد:



و اگر بر روی هر سطر اطلاعات دوبار کلیک کنید، به معادل JSON آن نیز خواهید رسید:



اینبار برنامه را به صورت زیر تغییر دهید تا روابط بین کلاس‌ها را نیز پیاده سازی کند:

```
using System;
using Raven.Client.Document;
using RavenDBSample01.Models;

namespace RavenDBSample01
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var store = new DocumentStore
                                {
                                    Url = "http://localhost:8080"
                                }.Initialize())
            {
                using (var session = store.OpenSession())
                {
```

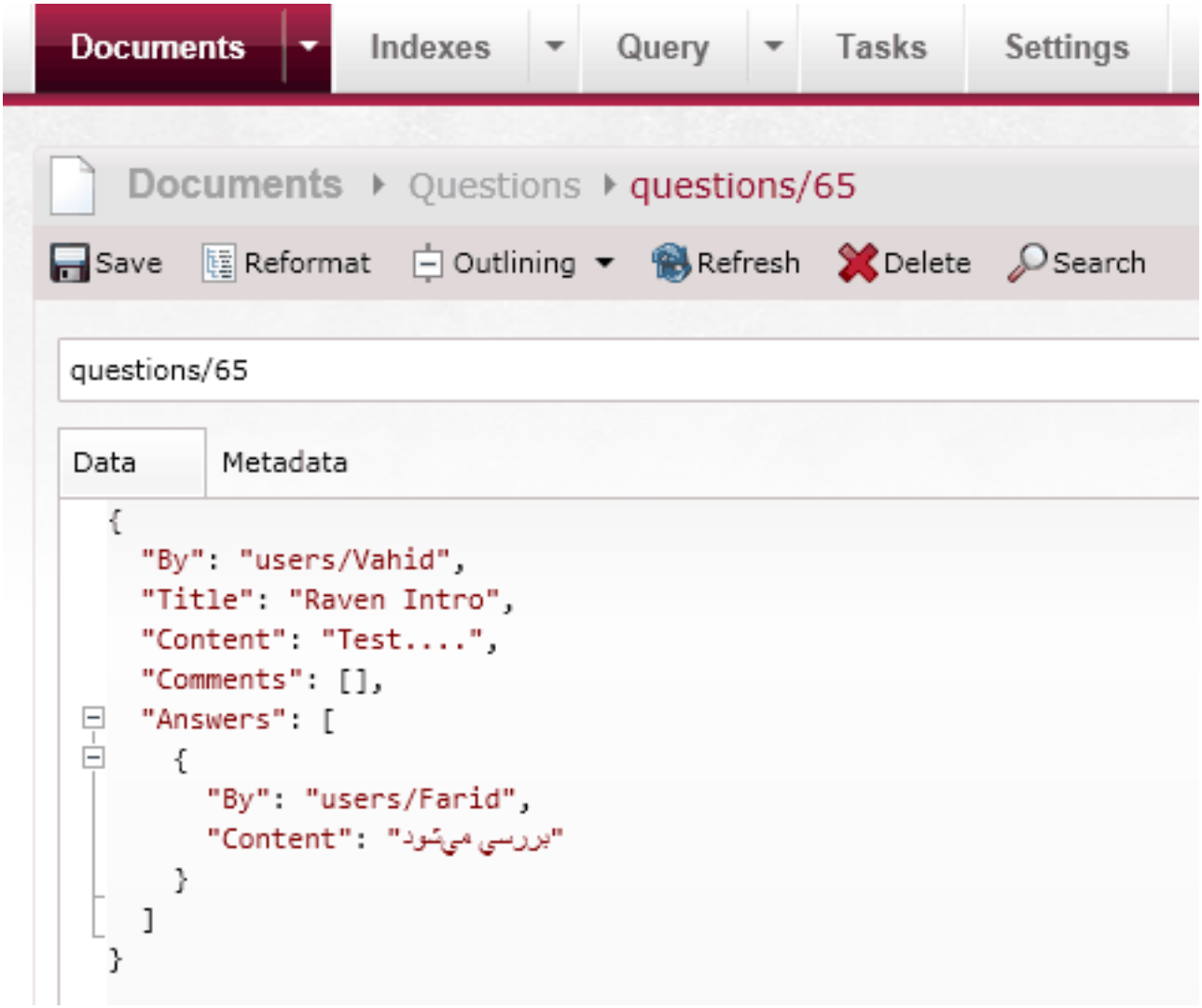
```
                    var question = new Question
                    {
                        By = "users/Vahid",
                        Title = "Raven Intro",
                        Content = "Test...."
                    };
                    question.Answers.Add(new Answer
                    {
                         By = "users/Farid",
                         Content = "بررسی می‌شود"
                    });
                    session.Store(question);

                    session.SaveChanges();
                }

                using (var session = store.OpenSession())
                {
                    var question1 = session.Load<Question>("questions/1");
                    Console.WriteLine(question1.By);
                }
            }
        }
    }
}
```

در اینجا یک سؤال به همراه پاسخی به آن تعریف شده است. همچنین در مرحله بعد، نحوه کوئری گرفتن مقدماتی از اطلاعات را بر اساس Id سند مرتبط، مشاهده می‌کنید. چون یک Session، الگوی واحد کار را پیاده سازی می‌کند، اگر پس از Load یک سند، خواصی از آن‌را تغییر دهیم و در پایان Session متد SaveChanges فراخوانی شود، به صورت خودکار این تغییرات به بانک اطلاعاتی نیز اعمال خواهند شد (روش به روز رسانی اطلاعات). این مورد بسیار شبیه است به مباحث پایه ای Change tracking که در بسیاری از ORMهای معروف تاکنون پیاده سازی شده‌اند. روش حذف اطلاعات نیز به همین ترتیب است. ابتدا سند مورد نظر یافت شده و سپس متد session.Delete بر روی این شیء یافت شده فراخوانی گردیده و در پایان سشن باید SaveChanges جهت نهایی شدن تراکنش فراخوانی گردد.

اگر برنامه فوق را اجرا کرده و به ساختار اطلاعات ذخیره شده نگاهی بیندازیم به شکل زیر خواهیم رسید:



نکته جالبی که در اینجا وجود دارد، عدم نیاز به join نویسی برای دریافت اطلاعات وابسته به یک شیء است. اگر سؤالی وجود دارد، پاسخ‌های به آن و یا سایر نظرات، یکجا داخل همان سؤال ذخیره می‌شوند و به این ترتیب سرعت دسترسی نهایی به اطلاعات بیشتر شده و همچنین قفل گذاری روی سایر اسناد کمتر. این مساله نیز به ذات NoSQL و یا غیر رابطه‌ای RavenDB بر می‌گردد. در بانک‌های اطلاعاتی NoSQL، مفاهیمی مانند کلیدهای خارجی، JOIN بین جداول و امثال آن وجود خارجی ندارند. برای نمونه اگر به کلاس‌های مدل‌های برنامه دقت کرده باشید، خبری از وجود Id در آن‌ها نیست. RavenDB یک Document store است و نه یک Relation store. در اینجا کل درخت تو در توی خواص یک شیء دریافت و به صورت یک سند ذخیره می‌شود. به حاصل این نوع عملیات در دنیای بانک‌های اطلاعاتی رابطه‌ای، Denormalized data هم گفته می‌شود.

البته می‌توان به کلاس‌های تعریف شده خاصیت رشته‌ای Id را نیز اضافه کرد. در این حالت برای مثال در حالت فراخوانی متد Load، این خاصیت رشته‌ای، با Id تولید شده توسط RavenDB مانند "questions/1" مقدار دهی می‌شود. اما از این Id برای تعریف ارجاعات به سؤالات و پاسخ‌های متناظر استفاده نخواهد شد؛ چون تمام آن‌ها جزو یک سند بوده و داخل آن قرار می‌گیرند.

نظرات خوانندگان

نویسنده: رضا ساکت
تاریخ: ۱۳۹۲/۰۸/۰۵ ۱۹:۲۰

با سلام و احترام

بیان کردید "البته لازم به ذکر است که RavenDB در 4 حالت برنامه کنسول (همین سرور فوق)، نصب به عنوان یک سرویس ویندوز NT، هاست شدن در IIS و حالت مدفون شده یا Embedded قابل استفاده است.
" خواهش میکنم راجع به هر مورد توضیح دهید.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۸/۰۵ ۲۰:۱۱

در ادامه دوره در مطلب « بررسی حالت‌های مختلف نصب RavenDB » در این مورد بیشتر بحث شده.

نویسنده: vici
تاریخ: ۱۳۹۲/۱۰/۲۹ ۱۱:۳۸

سلام
آقای نصیری موارد کاربردهای Raven.DB چی هست؟
ممنون

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۰/۲۹ ۱۱:۴۴

لطفا پیشنیازهای بحث را که در ابتدای مطلب عنوان شده مطالعه کنید تا با مفاهیم اولیه و علت وجودی بانک‌های اطلاعاتی NoSQL آشنا شوید.

نویسنده: علی رضایی
تاریخ: ۱۳۹۳/۱۰/۲۶ ۱۹:۴۴

با سلام؛ در مفاهیم پایه آمده است بانک‌های اطلاعاتی no sql خصوصیت **Non-schematized/schema free یا بدون اسکیما دارند،** اما در این مطلب ابتدا شمای بانک مثل کلاس Question یا Answer تعریف شده و در نتیجه ابتدا این ساختار در بانک تشکیل میشود و سپس داده‌ها در آن قرار میگیرد. ممنون میشوم بیشتر توضیح بدید.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۲۶ ۱۹:۵۹

« ابتدا این ساختار در بانک تشکیل میشود »
خیر. این فقط ساختار یک سند است. سند بعدی را هر طور که علاقمند بودید طراحی و ثبت کنید. متد session.Store محدودیتی ندارد. همچنین جایی هم در برنامه این ساختار در ابتدای کار به بانک اطلاعاتی معرفی یا ثبت نمی‌شود. وجود یک کلاس در برنامه به معنی تشکیل ساختار آن در بانک اطلاعاتی نیست.
بدون اسکیما یعنی هر رکورد با رکورد قبلی یا بعدی خودش می‌تواند ساختار کاملا متفاوتی داشته باشد.

نویسنده: علی رضایی
تاریخ: ۱۳۹۳/۱۰/۲۶ ۲۰:۱۷

با سلام و احترام

در انتهای بحث فرمودید « نکته جالبی که در اینجا وجود دارد، عدم نیاز به join نویسی برای دریافت اطلاعات وابسته به یک شیء است ». با توجه به این مطلب من دقیقا متوجه نشدم در مسئله زیر چه باید کرد:
هر answer و question توسط By مشخص میشود توسط چه کسی ارسال شده (مثلاً : "users/Vahid":"By")، و در سند users اطلاعات کامل هر کاربر مثل نام کاریری، ایمیل، آدرس و شماره تماس و... در زمان ثبت نام ذخیره شده است. سوال: اگر لازم باشد به ازای هر سوال یا جواب ایمیل و شماره تماس فرستنده را هم به کاربر نهایی نشان دهیم، باید در زمان ارسال سوال یا جواب ابتدا کاربر را با استفاده از By در users یافته و ایمیل و شماره تماس وی را خوانده و در سند آن سوال یا جواب ذخیره کنیم؛ **یا** باید هر زمان که سوال یا جوابی ارسال شد فقط نام کاربری را ثبت کنیم و در زمان نمایش سوال‌ها و جواب‌ها به کاربر مشخصات فرستنده را از users خوانده و همراه با خصوصیات دیگر به کاربر نشان دهیم، که این مورد آخر همان بانک‌های اطلاعاتی رابطه ای است.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۲۶ ۲۱:۲۹

«که همان بانک‌های اطلاعاتی رابطه ای است»
خیر. در اینجا در صورت لزوم کل اطلاعات مورد نیاز یک سند را می‌توان داخل آن قرار داد. دید طراحی اسناد NoSQL با جداول normalize شده بانک‌های اطلاعاتی رابطه‌ای یکی نیست. این مساله به همراه جزئیات بیشتری در قسمت‌های بعد مانند مدل سازی داده‌ها در RavenDB بحث شده‌است.

عنوان: **کار با RavenDB از طریق REST API آن**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۱۴ ۱۴:۱۲
آدرس: www.dotnettips.info
برچسب‌ها: NoSQL, RavenDB

در این قسمت قصد داریم برخلاف رویه معمول کار با RavenDB که از طریق کتابخانه‌های کلاینت آن انجام می‌شود، با استفاده از REST API آن، ساز و کار درونی آن‌را بیشتر بررسی کنیم.

REST چیست؟

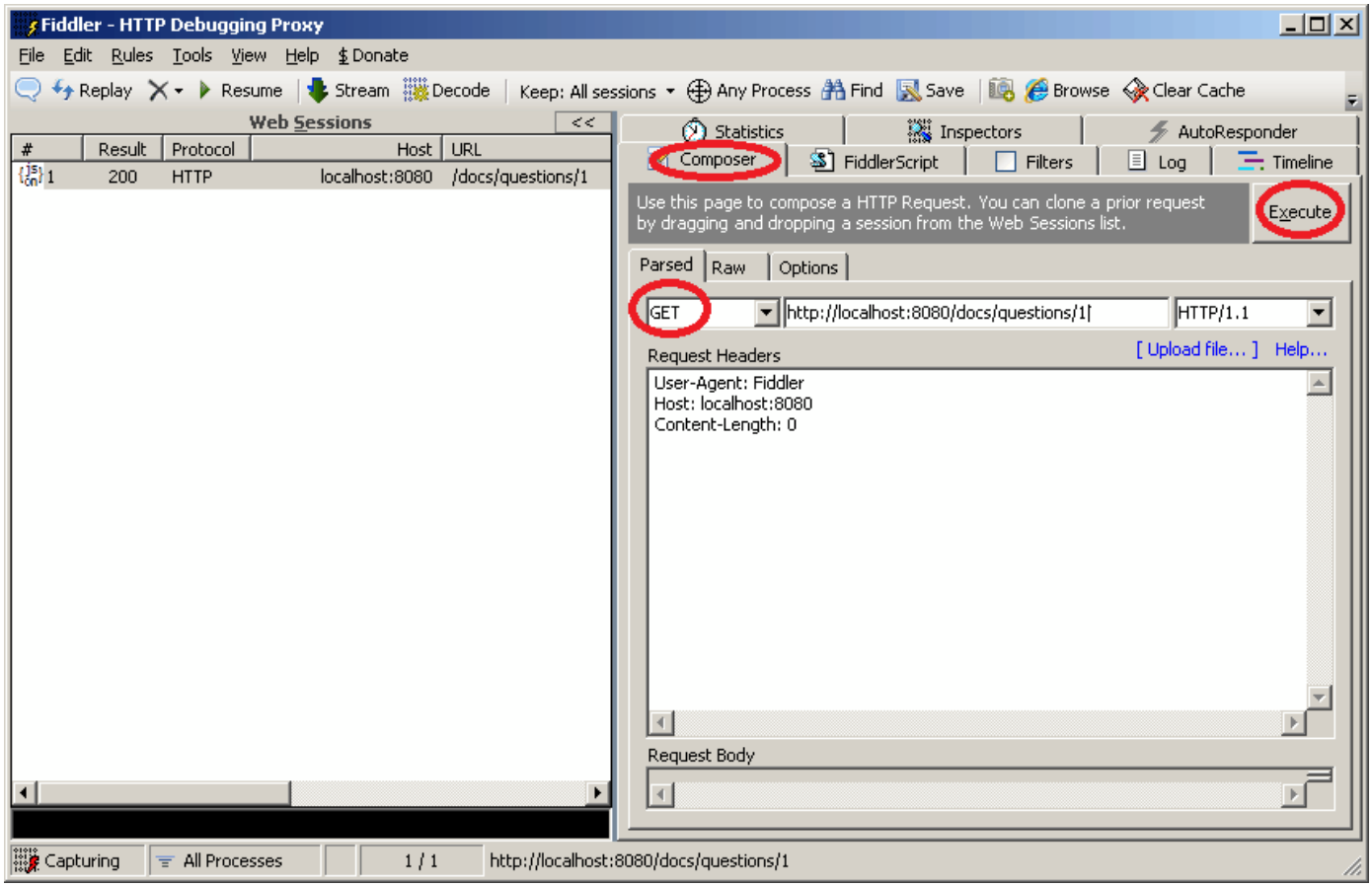
برای درک ساختار پشت صحنه RavenDB نیاز است با مفهوم REST آشنا باشیم؛ زیرا سرور این بانک اطلاعاتی، خود را به صورت یک RESTful web service در اختیار مصرف کنندگان قرار می‌دهد.
REST مخفف representational state transfer است و این روزها هر زمانیکه صحبت از آن به میان می‌آید منظور یک RESTful web service است که با استفاده از تعدادی HTTP Verb استاندارد می‌توان با آن کار کرد؛ مانند GET، POST، PUT و DELETE. با استفاده از GET، یک منبع ذخیره شده بازگشت داده می‌شود. با استفاده از فعل PUT، اطلاعاتی به منابع موجود اضافه و یا جایگزین می‌شوند. POST نیز مانند PUT است با این تفاوت که نوع اطلاعات ارسالی آن اهمیتی نداشته و تفسیر آن به سرور واگذار می‌شود. از DELETE نیز برای حذف یک منبع استفاده می‌گردد.

چند مثال

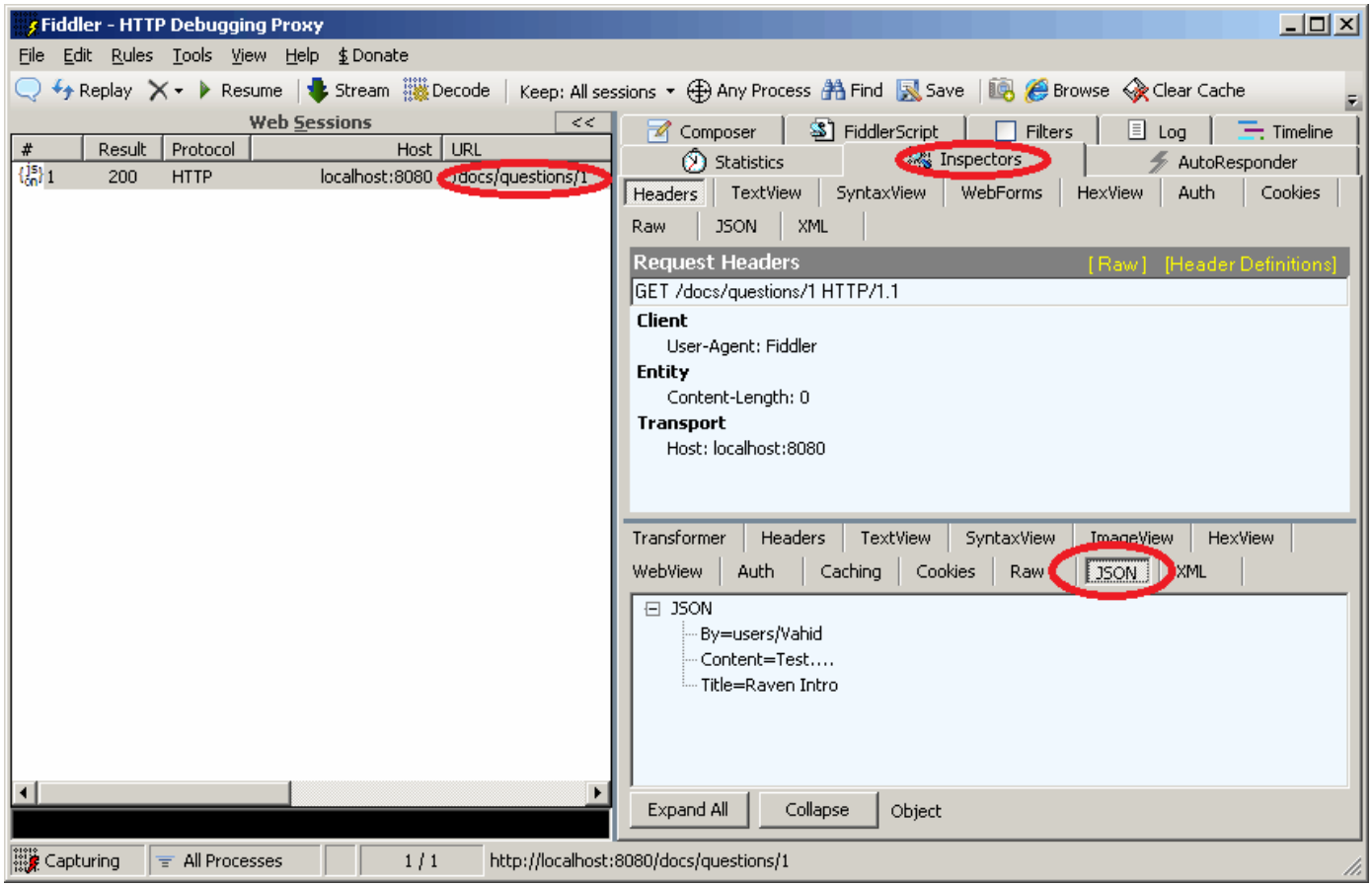
فرض کنید REST API برنامه‌ای از طریق آدرس http://myapp.com/api/questions در اختیار شما قرار گرفته است. در این آدرس، به questions منابع یا Resource گفته می‌شود. اگر دستور GET پروتکل HTTP بر روی این آدرس اجرا شود، انتظار ما این است که لیست تمام سؤالات بازگشت داده شود و اگر از دستور POST استفاده شود، باید یک سؤال جدید به مجموعه منابع موجود اضافه گردد.
اکنون آدرس http://myapp.com/api/questions/1 را درنظر بگیرید. در اینجا عدد یک معادل Id اولین سؤال ثبت شده است. بر اساس این آدرس خاص، اینبار اگر دستور GET صادر شود، تنها اطلاعات سؤال یک بازگشت داده خواهد شد و یا اگر از دستور PUT استفاده شود، اطلاعات سؤال یک با مقدار جدید ارسالی جایگزین می‌شود و یا با فراخوانی دستور DELETE، سؤال شماره یک حذف خواهد گردید.

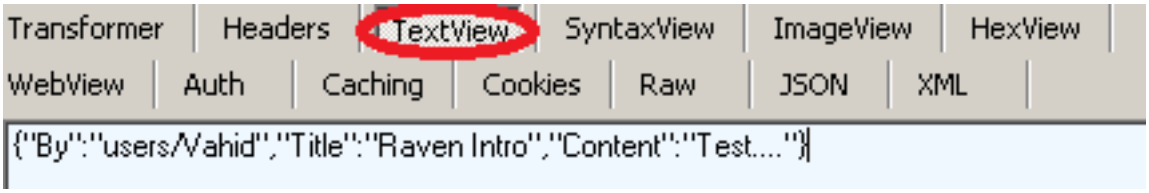
کار با دستور GET

در ادامه، به مثال قسمت قبل مراجعه کرده و تنها سرور RavenDB را اجرا نمائید (برنامه Raven.Server.exe)، تا در ادامه بتوانیم دستورات HTTP را بر روی آن امتحان کنیم. همچنین نیاز به برنامه معروف فیدلر نیز خواهیم داشت. از این برنامه برای ساخت دستورات HTTP استفاده خواهد شد.
پس از دریافت و نصب فیدلر، برگه Composer آن‌را گشوده و http://localhost:8080/docs/questions/1 را در حالت GET اجرا کنید:

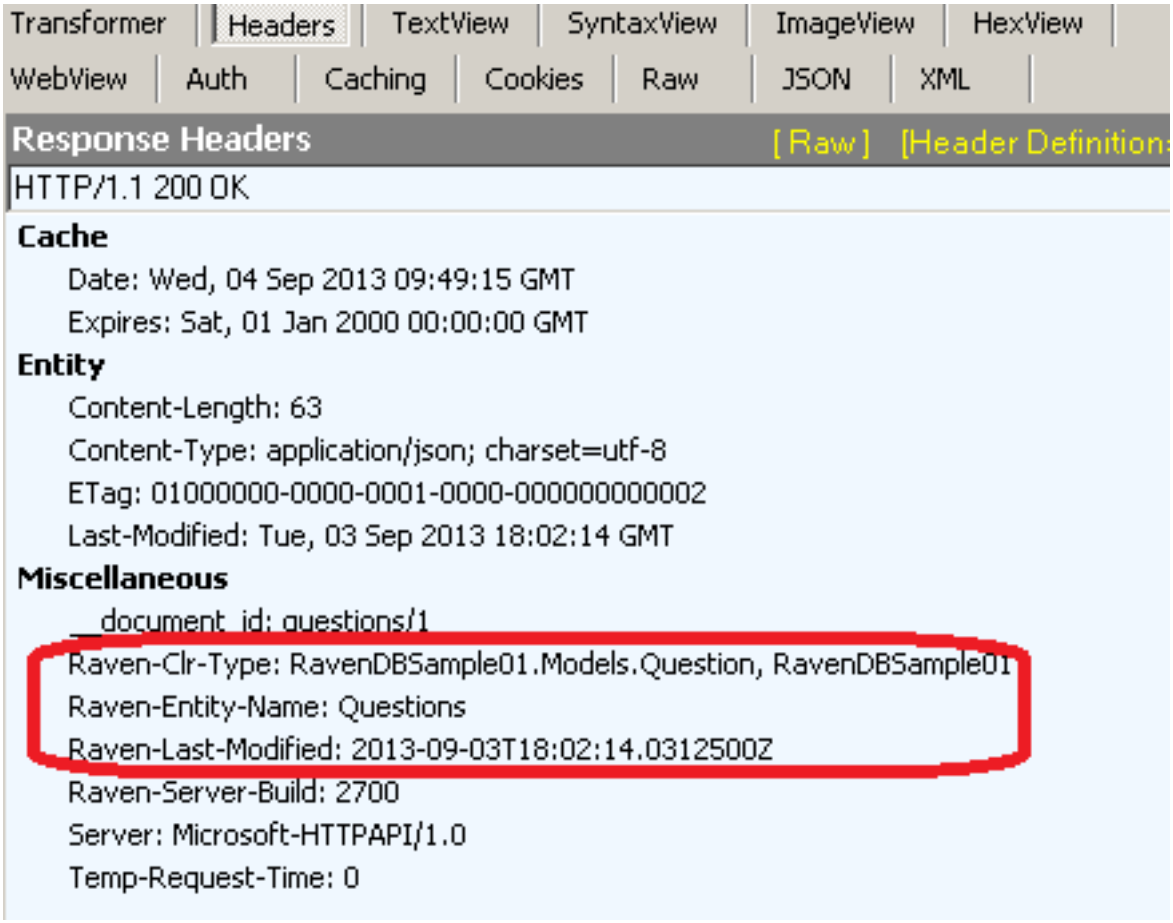


در این حالت دستور بر روی بانک اطلاعاتی اجرا شده و خروجی را در برگه Inspectors آن می‌توان مشاهده کرد:



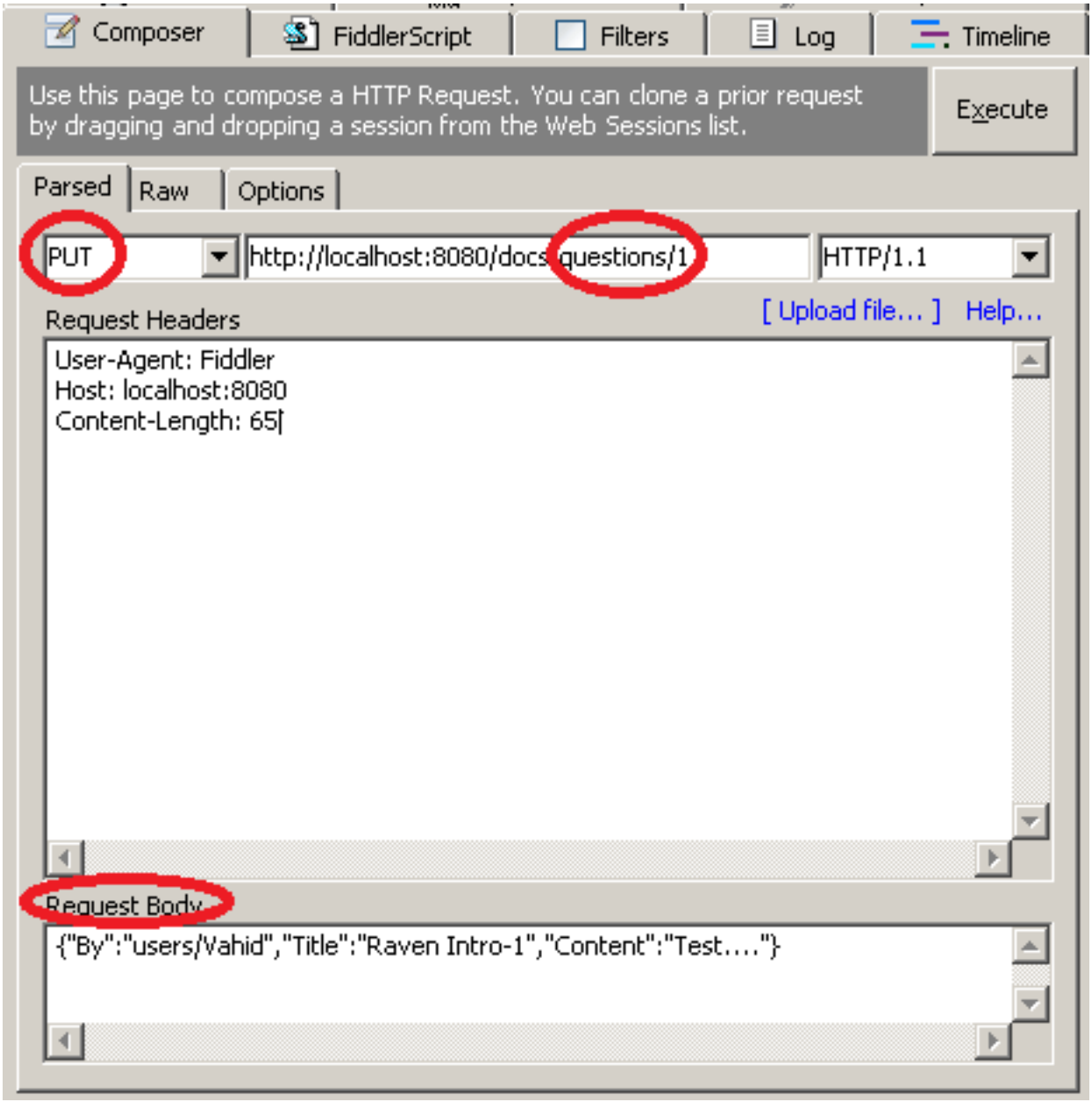


به علاوه در اینجا یک سری هدر اضافی (یا متادیتا) را هم می‌توان مشاهده کرد که RavenDB جهت سهولت کار کلاینت خود ارسال کرده است:



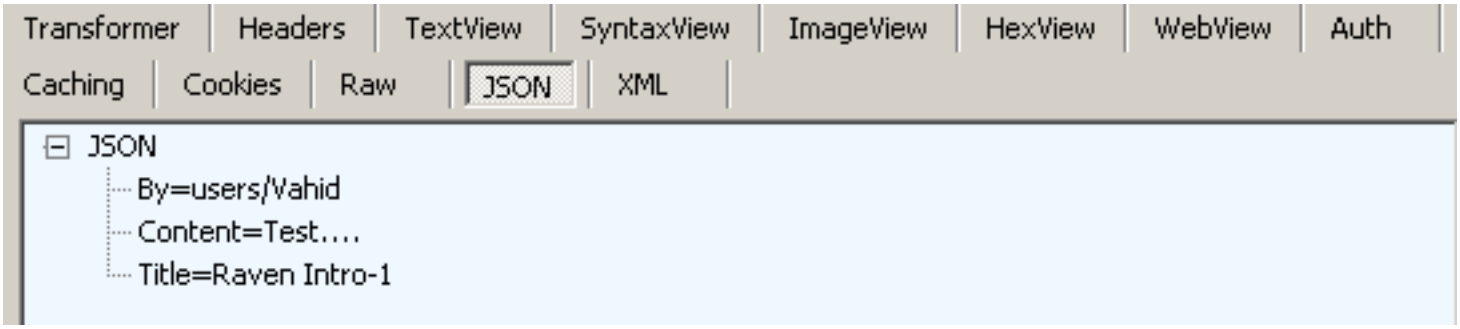
یک نکته: اگر آدرس `http://localhost:8080/docs/questions` را اجرا کنید، به معنای درخواست دریافت تمام سؤالات است. اما RavenDB به صورت پیش فرض طوری طراحی شده‌است که تمام اطلاعات را بازگشت ندهد و شعار آن `Safe by default` است. به این ترتیب مشکلات مصرف حافظه بیش از حد، پیش از بکارگیری یک سیستم در محیط کاری واقعی، توسط برنامه نویس یافت شده و مجبور خواهد شد تا برای نمایش تعداد زیادی رکورد، حتما صفحه بندی اطلاعات را پیاده سازی کرده و هربار تعداد معقولی از رکوردها را واکشی نماید.

کار با دستور PUT



اینبار نوع دستور را به PUT و آدرس را به http://localhost:8080/docs/questions/1 تنظیم می‌کنیم. همچنین در قسمت Request body، مقداری را که قرار است در سؤال یک درج شود، با فرمت JSON وارد می‌کنیم.

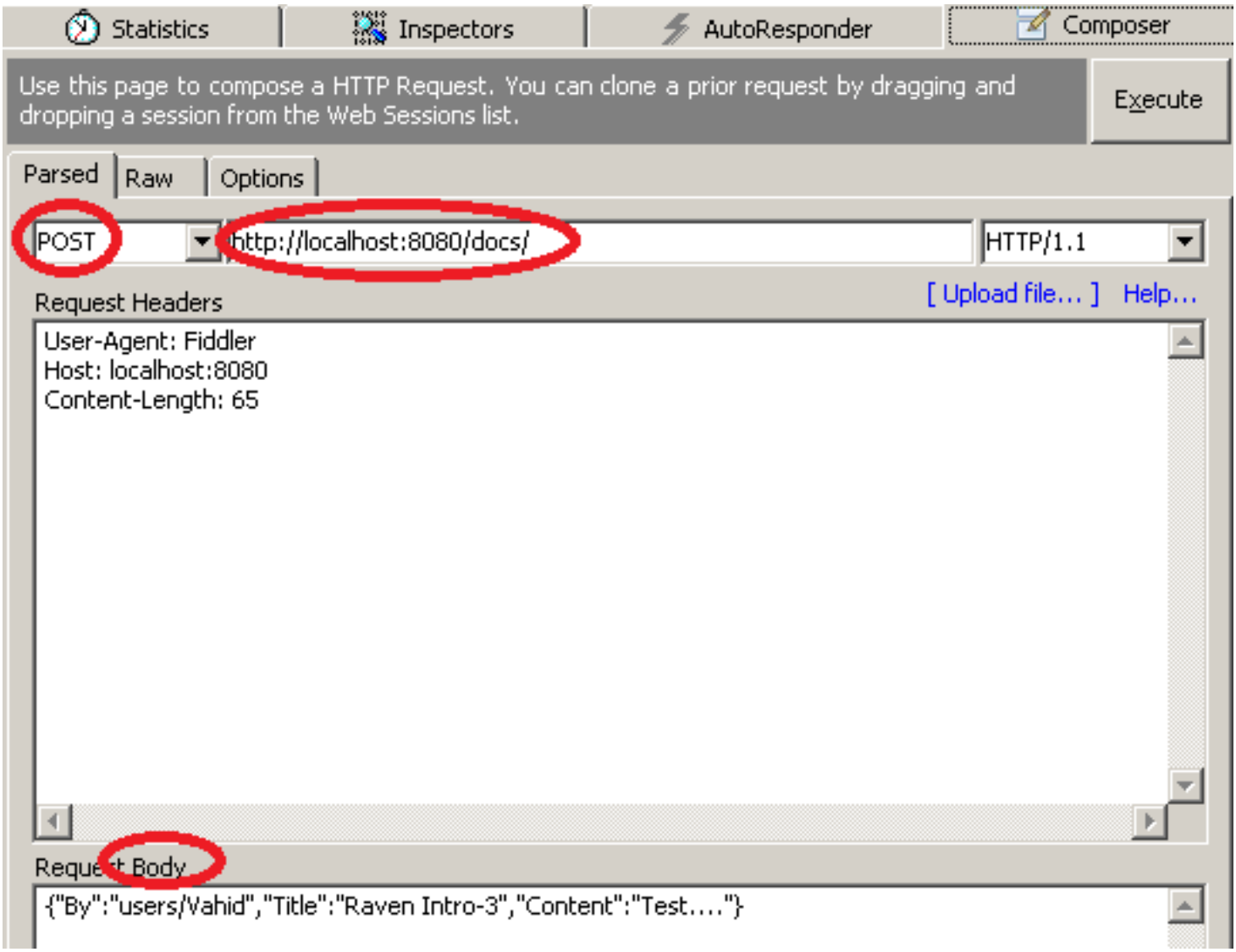
برای آزمایش صحت عملکرد آن، مرحله کار با دستور GET را یکبار دیگر تکرار نمائید:



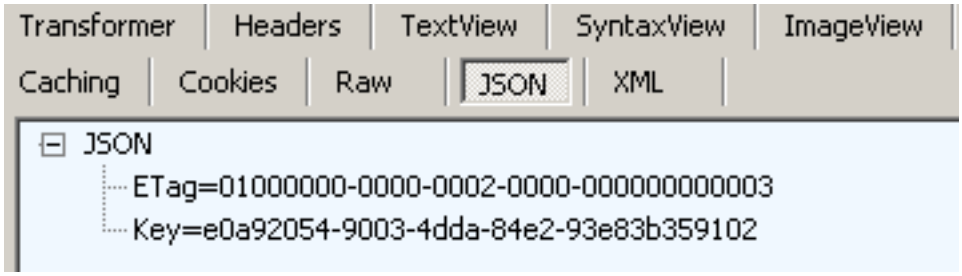
همانطور که مشاهده می‌کنید، تغییر ما در عنوان سؤال یک، با موفقیت اعمال شده است.

کار با دستور POST

در حین کار با دستور PUT، نیاز است حتما Id سؤال مورد نظر برای به روز رسانی (و یا حتی ایجاد نمونه جدید، در صورت عدم وجود) ذکر شود. اگر نیاز است اطلاعاتی به سیستم اضافه شوند و Id آن توسط RavenDB انتساب داده شود، بجای دستور PUT از دستور POST استفاده خواهیم کرد.



مطابق تصویر، اطلاعات شیء مدنظر را با فرمت JSON به آدرس http://localhost:8080/docs/ ارسال خواهیم کرد. در این حالت اگر به برگه‌ی Inspectors مراجعه نمائیم، یک چنین خروجی JSON ایی دریافت می‌گردد:



Key در اینجا شماره منحصربفرد سند ایجاد شده است و برای دریافت آن تنها کافی است که دستور GET را بر روی آدرس زیر که نمایانگر Key دریافتی است، اجرا کنیم:

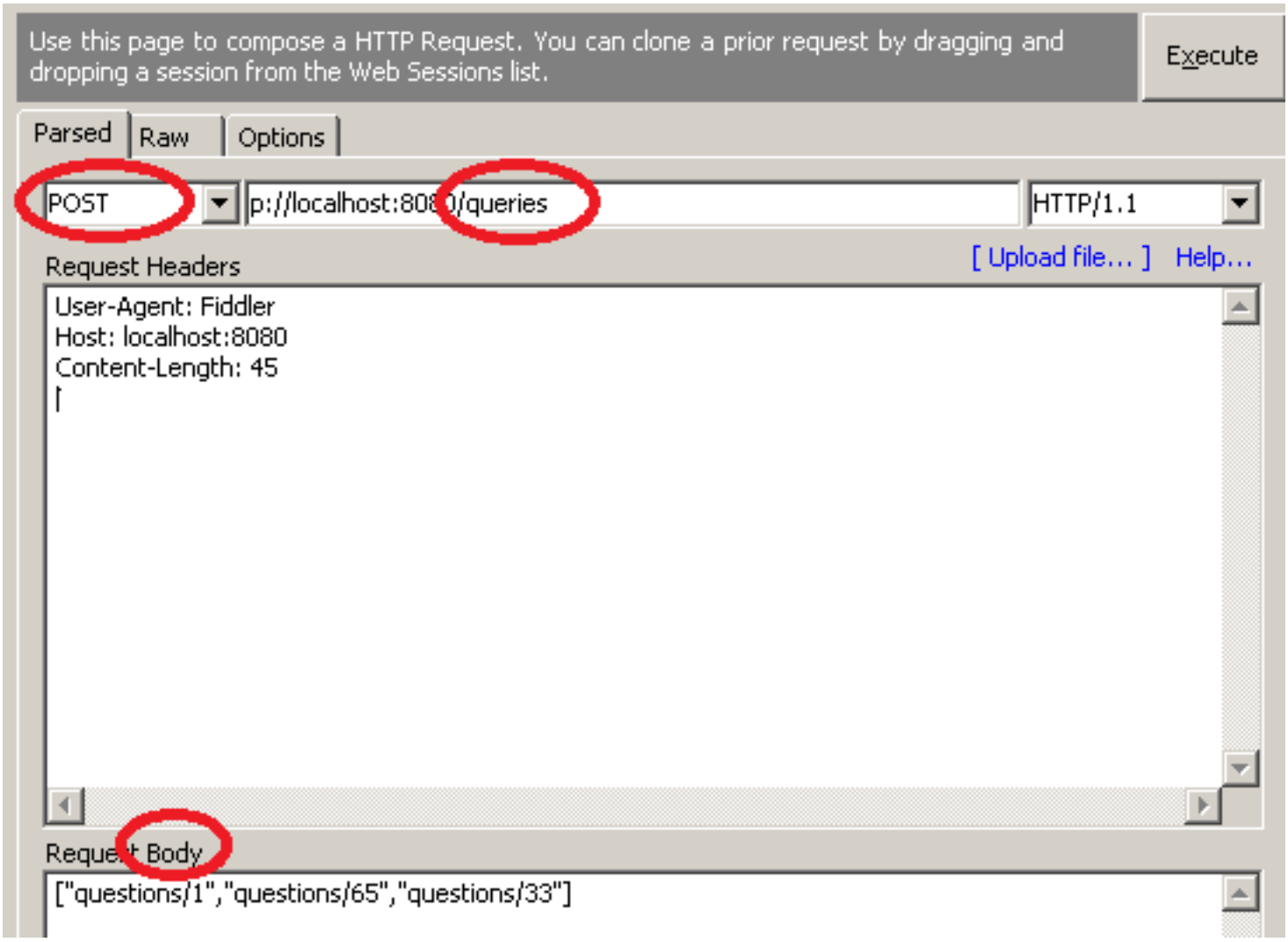
http://localhost:8080/docs/e0a92054-9003-4dda-84e2-93e83b359102

کار با دستور DELETE

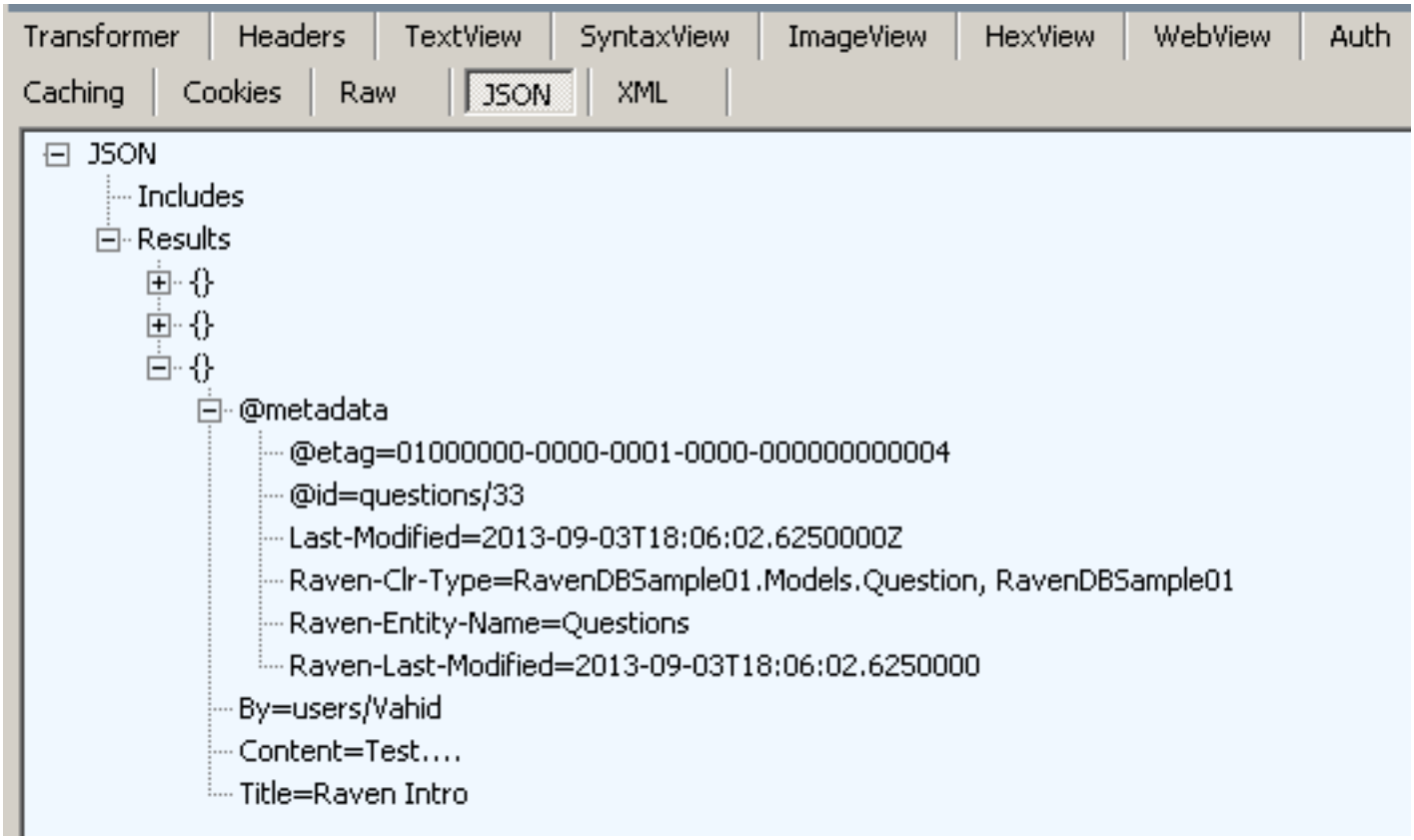
برای حذف یک سند تنها کافی است آدرس آن‌را وارد کرده و نوع دستور را بر روی Delete قرار دهیم. برای مثال اگر دستور Delete را بر روی آدرس فوق که به همراه Id تولید شده توسط RavenDB است اجرا کنیم، بلافاصله سند از بانک اطلاعاتی حذف خواهد شد.

بازگشت چندین سند از بانک اطلاعاتی RavenDB

برای نمونه، در فراخوانی‌های Ajaxایی نیاز است چندین رکورد با هم بازگشت داده شوند. برای این منظور باید یک درخواست Post ویژه را مهیا کرد:



در اینجا آدرس ارسال اطلاعات، آدرس خاص `http://localhost:8080/queries` است. اطلاعات ارسالی به آن، آرایه‌ای از Idهای اسنادی است که به اطلاعات آن‌ها نیاز داریم.



بنابراین برای کار با RavenDB در برنامه‌های وب و خصوصا کدهای سمت کلاینت آن، نیازی به کلاینت یا کتابخانه ویژه‌ای نیست و تنها کافی است یک درخواست Ajax از نوع post را به آدرس کوئری‌های سرور RavenDB ارسال کنیم تا نتیجه نهایی را به شکل JSON دریافت نمائیم.

نظرات خوانندگان

نویسنده: شهروز جعفری
تاریخ: ۱۳۹۲/۰۷/۲۳ ۱۴:۳۰

چرا در post از http://localhost:8080/docs/ استفاده شده و questions در url دیده نمیشه ولی در put و get دیده میشه؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۷/۲۳ ۱۵:۲۱

put کار به روز رسانی و یا حتی ایجاد یک id مشخص رو انجام می‌دهد. اگر id مشخص نیست، از post استفاده خواهد شد تا محاسبه آن id خودکار شده و به سیستم واگذار شود.

نویسنده: رضا ساکت
تاریخ: ۱۳۹۲/۰۸/۱۶ ۲۱:۲۴

سلام

با این وصف کسی که نشانی سرور دیتابیس را داشته باشد مستقیم و به راحتی می‌تواند اطلاعات را دستکاری نماید. همینطور است؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۸/۱۶ ۲۱:۳۶

خیر. این دسترسی‌ها در این مثال میسر شد چون حالت پیش فرض نصب آزمایشی سورس باز آن، حالت دسترسی ادمین است .

نویسنده: رضا ساکت
تاریخ: ۱۳۹۲/۰۸/۱۶ ۲۱:۵۹

بنابراین برای کار ایمن در RavenDB میبایست آنرا خرید

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۸/۱۶ ۲۲:۶

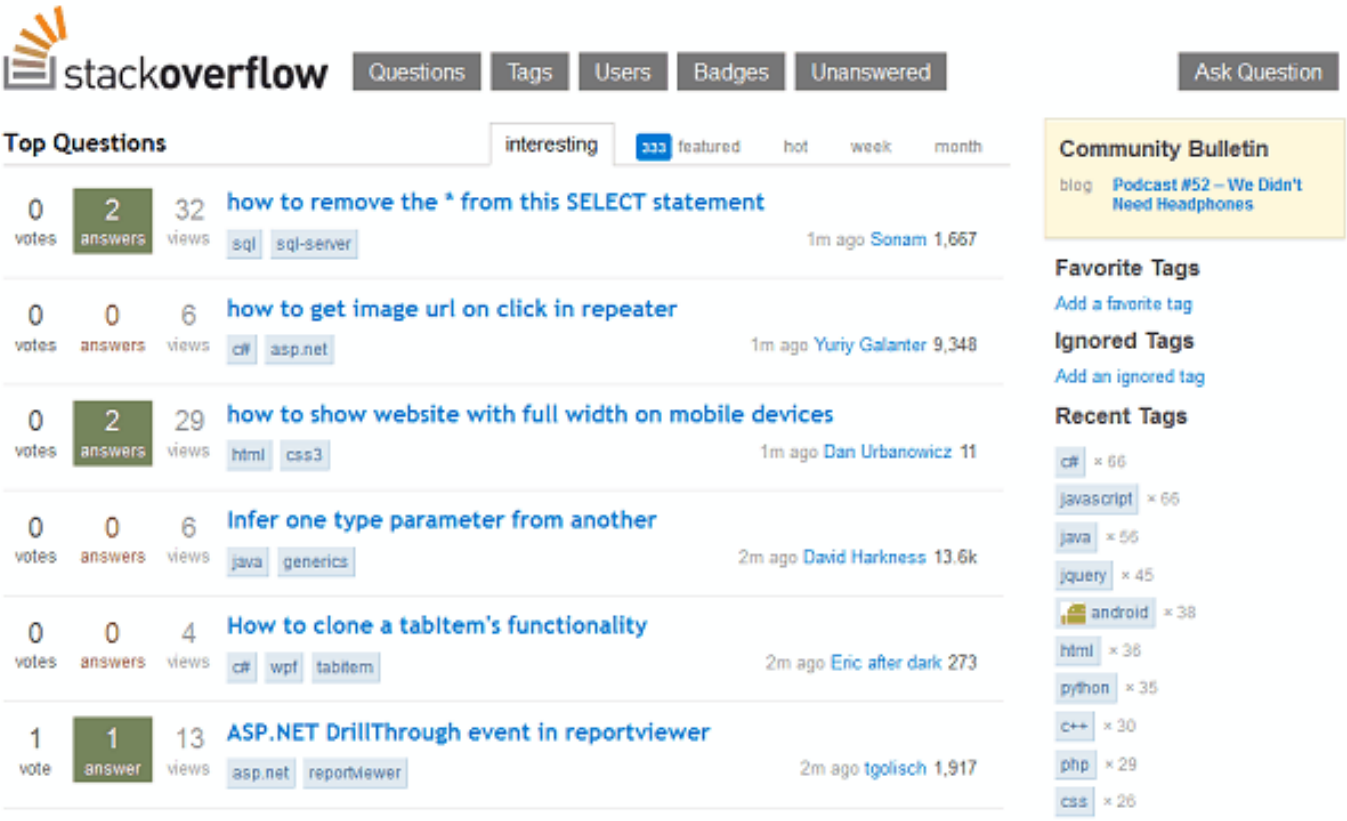
- بله.
- البته می‌شود پورت‌های دسترسی خارجی به یک سرور را با فایروال بست. به این ترتیب فقط برنامه نصب شده در آن سرور امکان اتصال را خواهد داشت (خیلی‌ها با SQL Server هم به همین نحو کار می‌کنند؛ یک برنامه وب و یک برنامه سرور SQL دارند روی یک سرور. برنامه وب سفارشی، لایه اتصال امن به بانک اطلاعاتی است).
- همچنین حالت نصب embedded آن دسترسی از بیرون ندارد و فقط از طریق برنامه قابل استفاده است.

عنوان: **مدل سازی داده‌ها در RavenDB**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۱۴ ۱۴:۴۱
آدرس: www.dotnettips.info
برچسب‌ها: NoSQL, RavenDB

در مطلب جاری، به صورت اختصاصی، مبحث مدل سازی اطلاعات و رسیدن به مدل ذهنی مرسوم در طراحی‌های NoSQL سندگرا را در مقایسه با دنیای Relational، بررسی خواهیم کرد.

تفاوت‌های دوره ما با زمانیکه بانک‌های اطلاعاتی رابطه‌ای پدیدار شدند

- دنیای بانک‌های اطلاعاتی رابطه‌ای برای Write بهینه سازی شده‌اند؛ از این جهت که تاریخچه پیدایش آن‌ها به دهه 70 میلادی بر می‌گردد، زمانیکه برای تهیه سخت دیسک‌ها باید هزینه‌های گزافی پرداخت می‌شد. به همین جهت الگوریتم‌ها و روش‌های بسیاری در آن دوره ابداع شدند تا ذخیره سازی اطلاعات، حجم کمتری را به خود اختصاص دهند. اینجا است که مباحثی مانند Normalization بوجود آمدند تا تضمین شود که داده‌ها تنها یکبار ذخیره شده و دوبار در جاهای مختلفی ذخیره نگردند. جهت اطلاع در سال 1980 میلادی، یک سخت دیسک 10 مگابایتی حدود 4000 دلار قیمت داشته است.
- تفاوت مهم دیگر دوره ما با دهه‌های 70 و 80 میلادی، پدیدار شدن UI و روابط کاربری بسیار پیچیده، در مقایسه با برنامه‌های خط فرمان یا حداکثر فرم‌های بسیار ساده ورود اطلاعات در آن زمان است. برای مثال در دهه 70 میلادی تصور UI ایی مانند صفحه ابتدایی سایت Stack overflow احتمالا به ذهن هم خطور نمی‌کرده است.



تهیه چنین UI ایی نه تنها از لحاظ طراحی، بلکه از لحاظ تامین داده‌ها از جداول مختلف نیز بسیار پیچیده است. برای مثال برای رندر صفحه اول سایت استک اورفلو ابتدا باید تعدادی سؤال از جدول سؤالات واکشی شوند. در اینجا در ذیل هر سؤال نام شخص مرتبط را هم مشاهده می‌کنید. بنابراین اطلاعات نام او، از جدول کاربران نیز باید دریافت گردد. یا در اینجا تعداد رای‌های هر سؤال را نیز مشاهده می‌کنید که به طور قطع اطلاعات آن در جدول دیگری نگه داری می‌شود. در گوشه‌ای از صفحه،

برچسب‌های مورد علاقه و در ذیل هر سؤال، برچسب‌های اختصاصی هر مطلب نمایش داده شده‌اند. تگ‌ها نیز در جدولی جداگانه قرار دارند. تمام این قسمت‌های مختلف، نیاز به واکشی و رندر حجم بالایی از اطلاعات را دارند.
- تعداد کاربران برنامه‌ها در دهه‌های 70 و 80 میلادی نیز با دوره ما متفاوت بوده‌اند. اغلب برنامه‌های آن دوران تک کاربره طراحی می‌شدند؛ با بانک‌های اطلاعاتی که صرفا جهت کار بر روی یک سیستم طراحی شده بودند. اما برای نمونه سایت استک اور فلویی که مثال زده شده، توسط هزاران و یا شاید میلیون‌ها نفر مورد استفاده قرار می‌گیرد؛ با توزیع و تقسیم اطلاعات آن بر روی سرورها مختلف.

معرفی مفهوم Unit of change

همین پیچیدگی‌ها سبب شدند تا جهت ساده‌سازی حل اینگونه مسایل، حرکتی به سمت دنیای NoSQL شروع شود. ایده اصلی مدل سازی داده‌ها در اینجا کم کردن تعداد اعمالی است که باید جهت رسیدن به یک نتیجه واحد انجام داد. اگر قرار است یک سؤال به همراه تگ‌ها، اطلاعات کاربر، رای‌ها و غیره واکشی شوند، چرا باید تعداد اعمال قابل توجهی جهت مراجعه به جداول مختلف مرتبط صورت گیرد؟ چرا تمام این اطلاعات را یکجا نداشته باشیم تا بتوان همگی را در طی یک واکشی به دست آورد و به این ترتیب دیگر نیازی نباشد انواع و اقسام JOINها را به چند ده جدول موجود نوشت؟
اینجا است که مفهومی به نام Unit of change مطرح می‌شود. در هر واحد تغییر، کلیه اطلاعات مورد نیاز برای رندر یک شیء قرار می‌گیرند. برای مثال اگر قرار است با شیء محصول کار کنیم، تمام اطلاعات مورد نیاز آن‌را اعم از گروه‌ها، نوع‌ها، رنگ‌ها و غیره را در طی یک سند بانک اطلاعاتی NoSQL سندگرا، ذخیره می‌کنیم.

محدوده‌های تراکنشی یا Transactional boundaries

محدوده‌های تراکنشی در Domain driven design به Aggregate root نیز معروف است. هر محدود تراکنشی حاوی یک Unit of change قرار گرفته داخل یک سند است. ابتدا بررسی می‌کنیم که در یک Read به چه نوع اطلاعاتی نیاز داریم و سپس کل اطلاعات مورد نیاز را بدون نوشتن JOIN ایی از جداول دیگر، داخل یک سند قرار می‌دهیم.
هر محدوده تراکنشی می‌تواند به محدوده تراکنشی دیگری نیز ارجاع داده باشد. برای مثال در RavenDB شماره‌های اسناد، یک سری رشته هستند؛ برخلاف بانک‌های اطلاعاتی رابطه‌ای که بیشتر از اعداد برای مشخص سازی Id استفاده می‌کنند. در این حالت برای ارجاع به یک کاربر فقط کافی است برای مثال مقدار خاصیت کاربر یک سند به "users/1" تنظیم شود. "users/1" نیز یک Id تعریف شده در RavenDB است.
مزیت این روش، سرعت واکشی بسیار بالای دریافت اطلاعات آن است؛ دیگر در اینجا نیازی به JOINهای سنگین به جداول دیگر برای تامین اطلاعات مورد نیاز نیست و همچنین در ساختارهای پیچیده‌تری مانند ساختارهای تو در تو، دیگر نیازی به تهیه کوئری‌های بازگشتی و استفاده از روش‌های پیچیده مرتبط با آن‌ها نیز وجود ندارد و کلیه اطلاعات مورد نظر، به شکل یک شیء JSON داخل یک سند حاضر و آماده برای واکشی در طی یک Read هستند.
به این ترتیب می‌توان به سیستم‌های مقیاس پذیری رسید. سیستم‌هایی که با بالا رفتن حجم اطلاعات در حین واکشی‌های داده‌های مورد نیاز، کند نبوده و بسیار سریع پاسخ می‌دهند.

Denormalization داده‌ها

اینجا است که احتمالا ذهن رابطه‌ای تربیت شده‌ی شما شروع به واکنش می‌کند! برای مثال اگر نام یک محصول تغییر کرد، چطور؟ اگر آدرس یک مشتری نیاز به ویرایش داشت، چطور؟ چگونه یکپارچگی اطلاعاتی که اکنون به ازای هر سند پراکنده شده‌است، مدیریت می‌شود؟
زمانیکه به این نوع سؤالات رسیده‌ایم، یعنی Denormalization رخ داده است. در اینجا سندهایی را داریم که کلیه اطلاعات مورد نیاز خود را یکجا دارند. به این مساله از منظر نگاه به داده‌ها در طی زمان نیز می‌توان پرداخت. به این معنا که صحیح است که آدرس مشتری خاصی امروز تغییر کرده است، اما زمانیکه سندی برای او در سال قبل صادر شده است، واقعا آدرس آن مشتری که سفارشی برایش ارسال شده، دقیقا همان چیزی بوده است که در سند مرتبط، ثبت شده و موجود می‌باشد. بنابراین سند قبلی با اطلاعات قبلی مشتری در سیستم موجود خواهد بود و اگر سند جدیدی صادر شد، این سند بدیهی است که از اطلاعات امروز مشتری استفاده می‌کند.

ملاحظات اندازه‌های داده‌ها

زمانیکه سندها بسیار بزرگ می‌شوند چه رخ خواهد داد؟ از لحاظ اندازه داده‌ها سه نوع سند را می‌توان متصور بود:
الف) سندهای محدود، مانند اغلب اطلاعاتی که تعداد فیلدهای مشخصی دارند با تعداد اشیاء مشخصی.
ب) سندهای نامحدود اما با محدودیت طبیعی. برای مثال اطلاعات فرزندان یک شخص را درنظر بگیرید. هرچند این اطلاعات نامحدود هستند، اما به صورت طبیعی می‌توان فرض کرد که سقف بالایی آن عموما به 20 نمی‌رسد!
ج) سندهای نامحدود، مانند سندهایی که آرایه‌ای از اطلاعات را ذخیره می‌کنند. برای مثال در یک سایت فروشگاه، اطلاعات فروش یک گروه از اجناس خاص را درنظر بگیرید که عموما نامحدود است. اینجا است که باید به اندازه اسناد نیز دقت داشت. برای مدیریت این مساله حداقل از دو روش استفاده می‌شود:
- محدود کردن تعداد اشیاء. برای مثال در هر سند حداکثر 100 اطلاعات فروش یک محصول بیشتر ثبت نشود. زمانیکه به این حد رسیدیم، یک سند جدید ایجاد شده و Id سند قبلی مثلا "products/1" در سند دوم ذکر خواهد شد.
- محدود کردن تعداد اطلاعات ذخیره شده بر اساس زمان
RavenDB برای مدیریت این مساله، مفهوم Includes را معرفی کرده است. در اینجا با استفاده از متد الحاقی Include، کار زنجیر کردن سندهای مرتبط صورت خواهد گرفت.

یک مثال عملی: مدل سازی داده‌های یک بلاگ در RavenDB

پس از این بحث مقدماتی که جهت معرفی ذهنیت مدل سازی داده‌ها در دنیای غیر رابطه‌ای NoSQL ضروری بود، در ادامه قصد داریم مدل‌های داده‌های یک بلاگ را سازگار با ساختار بانک اطلاعاتی NoSQL سندگرای RavenDB طراحی کنیم.
در یک بلاگ، تعدادی مطلب، نظر، برچسب (گروه‌های مطالب) و امثال آن وجود دارند. اگر بخواهیم این اطلاعات را به صورت رابطه‌ای مدل کنیم، به ازای هر کدام از این موجودیت‌ها یک جدول نیاز خواهد بود و برای رندر صفحه اصلی بلاگ، چندین و چند کوئری برای نمایش اطلاعات مطالب، نویسنده(ها)، برچسب‌ها و غیره باید به بانک اطلاعاتی ارسال گردد، که تعدادی از آن‌ها مستقیما بر روی یک جدول اجرا می‌شوند و تعدادی دیگر نیاز به JOIN دارند.
مشکلاتی که روش رابطه‌ای دارد:
- تعداد اعمالی که باید برای نمایش صفحه اول سایت صورت گیرد، بسیار زیاد است و این مساله با تعداد بالای کاربران از دید مقیاس پذیری سیستم مشکل ساز است.
- داده‌های مرتبط در جداول مختلفی پراکنده‌اند.
- این سیستم برای Write بهینه سازی شده است و نه برای Read. (همان بحث گران بودن سخت دیسک‌ها در دهه‌های قبل که در ابتدای بحث به آن اشاره شد)

مدل سازی سازگار با دنیای NoSQL یک بلاگ

در اینجا چند کلاس مقدماتی را مشاهده می‌کنید که تعریف آن‌ها به همین نحو صحیح است و نیاز به جزئیات و یا روابط بیشتری ندارند.

```
namespace RavenDBSample01.BlogModels
{
    public class BlogConfig
    {
        public string Id { set; get; }
        public string Title { set; get; }
        public string Description { set; get; }
        // ... more items here
    }

    public class User
    {
        public string Id { set; get; }
        public string FullName { set; get; }
        public string Email { set; get; }
        // ... more items here
    }
}
```

اما کلاس مطالب بلاگ را به چه صورتی طراحی کنیم؟ هر مطلب، دارای تعدادی نظر خواهد بود. اینجا است که بحث unit of change مطرح می‌شود و درج اطلاعاتی که در طی یک read نیاز است از بانک اطلاعاتی جهت رندر UI واکشی شوند. به این ترتیب به این نتیجه می‌رسیم که بهتر است کلیه کامنت‌های یک مطلب را داخل همان شیء مطلب مرتبط قرار دهیم. از این جهت که یک نظر، خارج از یک مطلب بلاگ دارای مفهوم نیست.
اما این طراحی نیز یک مشکل دارد. درست است که ساختار یک صفحه مطلب، از مطالب وبلاگ به همین نحوی است که توضیح داده شد؛ اما در صفحه اول سایت، هیچگاه کامنت‌های مطالب درج نمی‌شوند. بنابراین نیازی نیست تا تمام کامنت‌ها را داخل یک مطلب ذخیره کرد. به این ترتیب برای نمایش صفحه اول سایت، حجم کمتری از اطلاعات واکشی خواهند شد.

```
public class Post
    {
        public string Id { set; get; }
        public string Title { set; get; }
        public string Body { set; get; }

        public ICollection<string> Tags { set; get; }

        public string AuthorId { set; get; }

        public string PostCommentsId { set; get; }
        public int CommentsCount { set; get; }
    }

    public class Comment
    {
        public string Id { set; get; }
        public string Body { set; get; }
        public string AuthorName { set; get; }
        public DateTime CreatedAt { set; get; }
    }

    public class PostComments
    {
        public List<Comment> Comments { set; get; }
        public string LastCommentId { set; get; }
    }
```

در اینجا ساختار Post و Commentهای بلاگ را مشاهده می‌کنید. جایی که ذخیره سازی اصلی کامنت‌ها صورت می‌گیرد در شیء PostComments است. یعنی PostCommentsId شیء Post به یک وهله از شیء PostComments که حاوی کلیه کامنت‌های آن مطلب است، اشاره می‌کند.
به این ترتیب برای نمایش صفحه اول سایت، فقط یک کوئری صادر می‌شود. برای نمایش یک مطلب و کلیه کامنت‌های متناظر با آن دو کوئری صادر خواهند شد.

بنابراین همانطور که مشاهده می‌کنید، در دنیای NoSQL، طراحی مدل‌های داده‌ای بر اساس «سناریوهای Read» صورت می‌گیرد و نه صرفا طراحی یک مدل رابطه‌ای بهینه سازی شده برای حالت Write.

سورس کامل ASP.NET MVC این بلاگ‌را که « راکن بلاگ » نام دارد، از GitHub نویسندگان اصلی RavenDB می‌توانید دریافت کنید.

عنوان: **کوئری نویسی مقدماتی در RavenDB**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۱۵ ۰:۴۰
آدرس: www.dotnettips.info
برچسب‌ها: NoSQL, RavenDB

با شروع کوئری نویسی مقدماتی در RavenDB، در قسمت اول این مباحث، توسط فراخوانی متد Load یک سشن، آشنا شدید. در ادامه مباحث تکمیلی آن‌را مرور خواهیم کرد.

امکان استفاده از LINQ در RavenDB

RavenDB از LINQ جهت کوئری نویسی پشتیبانی می‌کند. برای استفاده از آن، در ادامه مطلب اول، ابتدا سرور RavenDB را اجرا نموده و سپس برنامه کنسول را به نحو ذیل تغییر دهید:

```
using System;
using System.Linq;
using Raven.Client.Document;
using RavenDBSample01.Models;

namespace RavenDBSample01
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var store = new DocumentStore
            {
                Url = "http://localhost:8080"
            }.Initialize())
            {
                using (var session = store.OpenSession())
                {
                    var questions = session.Query<Question>().Where(x => x.Title.StartsWith("Raven"));
                    foreach (var question in questions)
                    {
                        Console.WriteLine(question.Title);
                    }
                }
            }
        }
    }
}
```

در RavenDB برای دسترسی به امکانات LINQ، کار با متد Query یک سشن آغاز می‌شود و پس از آن، امکان استفاده از متدهای متداول LINQ مانند مثال فوق وجود خواهد داشت. البته بدیهی است مباحثی مانند JOIN و امثال آن در یک بانک اطلاعاتی NoSQL پشتیبانی نمی‌شود. ضمنا باید درنظر داشت که مبحث safe by default در اینجا نیز اعمال می‌شود. برای مثال اگر به کنسول سرور RavenDB که در حال اجرا است مراجعه کنید، یک چنین خروجی را حین اجرای مثال فوق می‌توان مشاهده کرد که در آن pageSize پیش فرضی اعمال شده است:

```
Available commands: cls, reset, gc, q
Request #   1: GET     -   179 ms - <system>   - 404 - /docs/Raven/Replication/Destinations
Request #   2: GET     - 3,818 ms - <system>   - 200 -
/indexes/dynamic/Questions?&query=Title%3ARaven*&pageSize=128
        Query: Title:Raven*
        Time: 3,494 ms
        Index: Auto/Questions/ByTitle
        Results: 2 returned out of 2 total.
```

یعنی در عمل کوئری‌را که اجرا کرده است، شبیه به کوئری ذیل می‌باشد و یک Take پیش فرض بر روی آن اعمال شده است:

```
var questions = session.Query<Question>().Where(x => x.Title.StartsWith("Raven")).Take(128);
```

علت این مساله نیز به تصمیم نویسنده اصلی آن بر می‌گردد؛ ایشان پیش از شروع به تهیه RavenDB، کار تهیه انواع و اقسام

پروفایلرهای مهم ORMهای معروف مانند NHibernate و Entity framework را انجام داده است و در این حین، یکی از مهم‌ترین مشکلاتی را که با آن‌ها در کدهای متداول برنامه نویس‌ها یافته است، unbounded queries است. کوئری‌هایی که حد و مرزی برای بازگشت اطلاعات قائل نمی‌شوند. داشتن این نوع کوئری‌ها با تعداد بالای کاربر، یعنی مصرف بیش از حد RAM بر روی سرور، به همراه بار پردازشی بیش از حد و غیر ضروری. چون عملا حتی اگر 10 هزار رکورد بازگشت داده شوند، عموم برنامه نویس‌ها حداکثر 100 رکورد آن‌را در یک صفحه نمایش می‌دهند و نه تمام رکوردها را.

ارتباط Lucene.NET و RavenDB

کل LINQ API تهیه شده در RavenDB یک محصور کننده امکانات Lucene.NET است. اگر پیشتر با Lucene.NET کار کرده باشید، در خروجی حالت دیباگ کنسول سرور فوق، سطر «Query: Title:Raven*» آشنا به نظر خواهد رسید. دقیقا کوئری LINQ نوشته شده به یک کوئری با Syntax مخصوص Lucene.NET ترجمه شده‌است. برای نمونه اگر علاقمند باشید که مستقیما کوئری‌های خاص لوسین را در RavenDB اجرا کنید، از Syntax ذیل می‌توان استفاده کرد:

```
var questions = session.Advanced.LuceneQuery<Question>().Where("Title:Raven*").ToList();
```

و یا اگر علاقمند به حفظ کردن Syntax خاص لوسین نیستید، یک سری متد الحاقی خاص نیز در اینجا برای LuceneQuery تدارک دیده شده است. برای مثال کوئری رشته‌ای فوق، معادل کوئری strongly typed ذیل است:

```
var questions = session.Advanced.LuceneQuery<Question>().WhereStartsWith(x => x.Title,
"Raven").ToList();
```

استفاده مجدد از کوئری‌ها در RavenDB

در RavenDB، متد Query به صورت immutable تعریف شده است و متد LuceneQuery حالت mutable دارد (ترکیبات آن نیز یک وهله است).
یک مثال:

```
var query = session.Query<User>().Where(x => x.Name.StartsWith("A"));
var ageQuery = query.Where(x => x.Age > 21);
var eyeQuery = query.Where(x => x.EyeColor == "blue");
```

در اینجا از کوئری ابتدایی، در دو کوئری مجزا استفاده مجدد شده است. ترجمه خروجی سه کوئری فوق به نحو زیر است:

```
query - Name:A*
ageQuery - (Name:A*) AND (Age_Range:{Ix21 TO NULL})
eyeQuery - (Name:A*) AND (EyeColor:blue)
```

به این معنا که زمانیکه به eyeQuery رسیدیم، نتیجه ageQuery با آن ترکیب نمی‌شود؛ چون متد Query از نوع immutable است.
در ادامه اگر همین سه کوئری فوق را با فرمت LuceneQuery تهیه کنیم، به عبارات ذیل خواهیم رسید:

```
var luceneQuery = session.Advanced.LuceneQuery<User>().WhereStartsWith(x => x.Name, "A");
var ageLuceneQuery = luceneQuery.WhereGreaterThan(x => x.Age, 21);
var eyeLuceneQuery = luceneQuery.WhereEquals(x => x.EyeColor, "blue");
```

در خروجی‌های این سه کوئری، مورد سوم مهم است:

```
luceneQuery - Name:A*
ageLuceneQuery - Name:A* Age_Range:{Ix21 TO NULL}
eyeLuceneQuery - Name:A* Age_Range:{Ix21 TO NULL} EyeColor:blue
```

همانطور که مشاهده می‌کنید، کوئری سوم، عبارت کوئری دوم را نیز به همراه دارد؛ این مورد دقیقا مفهوم اشیاء mutable یا تک

وهله‌ای است مانند LuceneQuery در اینجا.

And و Or شدن کوئری‌های ترکیبی در RavenDB

در مثال استفاده مجدد از کوئری‌ها، زمانیکه از Where استفاده شد، بین عبارات حاصل AND قرار گرفته است. این مورد را به نحو ذیل می‌توان تنظیم کرد و مثلا به OR تغییر داد:

```
session.Advanced.LuceneQuery<User>().UsingDefaultOperator(QueryOperator.And);
```

صفحه بندی اطلاعات در RavenDB

در ابتدای بحث عنوان شد که کوئری LINQ اجرا شده در RavenDB، یک Take مخفی و پیش فرض تنظیم شده به 128 آیتم را دارد. اکنون سؤال این خواهد بود که چگونه می‌توان اطلاعات را به صورت صفحه بندی شده، بر اساس شماره صفحه خاصی نمایش داد.

```
using System;
using System.Linq;
using Raven.Client.Document;
using RavenDBSample01.Models;

namespace RavenDBSample01
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var store = new DocumentStore
            {
                Url = "http://localhost:8080"
            }.Initialize())
            {
                using (var session = store.OpenSession())
                {
                    int pageNumber = 0;
                    int resultsPerPage = 2;

                    var questions = session.Query<Question>()
                                           .Where(x => x.Title.StartsWith("Raven"))
                                           .Skip(pageNumber * resultsPerPage)
                                           .Take(resultsPerPage);
                    foreach (var question in questions)
                    {
                        Console.WriteLine(question.Title);
                    }
                }
            }
        }
    }
}
```

برای انجام صفحه بندی در RavenDB، کافی است از متدهای Skip و Take بر اساس محاسباتی که مشاهده می‌کنید، استفاده گردد.

دریافت اطلاعات آماری کوئری اجرا شده

در RavenDB امکان دریافت یک سری اطلاعات آماری از کوئری اجرا شده نیز وجود دارد؛ برای مثال یک کوئری چند ثانیه طول کشیده است، چه تعدادی رکورد را بازگشت داده است و امثال آن. برای پیاده سازی آن، نیاز است از متد الحاقی Statistics به نحو ذیل استفاده کرد:

```
using System;
using System.Linq;
using Raven.Client.Document;
using RavenDBSample01.Models;
using Raven.Client;

namespace RavenDBSample01
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            using (var store = new DocumentStore
            {
                Url = "http://localhost:8080"
            }.Initialize())
            {
                using (var session = store.OpenSession())
                {
                    int pageNumber = 0;
                    int resultsPerPage = 2;
                    RavenQueryStatistics stats;
                    var questions = session.Query<Question>()
                                           .Statistics(out stats)
                                           .Where(x => x.Title.StartsWith("Raven"))
                                           .Skip(pageNumber * resultsPerPage)
                                           .Take(resultsPerPage);
                    foreach (var question in questions)
                    {
                        Console.WriteLine(question.Title);
                    }

                    Console.WriteLine("TotalResults: {0}", stats.TotalResults);
                }
            }
        }
    }
}
```

متد الحاقی Statistics پس از متد Query که نقطه آغازین نوشتن کوئری‌های LINQ است، فراخوانی شده و یک پارامتر out از نوع RavenQueryStatistics تعریف شده در فضای نام Raven.Client را دریافت می‌کند. پس از پایان کوئری می‌توان از این خروجی جهت نمایش اطلاعات آماری کوئری استفاده کرد.

امکانات ویژه فضای نام Raven.Client.Linq

یک سری متد الحاقی خاص جهت تهیه ساده‌تر کوئری‌های LINQ در فضای نام Raven.Client.Linq قرار دارند که پس از تعریف آن قابل دسترسی خواهند بود:

```
var list = session.Query<Question>().Where(q => q.By.In<string>(arrayOfUsers))).ToArray()
```

برای مثال در اینجا متد الحاقی جدید In را مشاهده می‌کنید که شبیه به کوئری SQL ذیل در دنیای بانک‌های اطلاعاتی رابطه‌ای عمل می‌کند:

```
SELECT * FROM tbl WHERE data IN (1, 2, 3)
```

اتصال به RavenDB با استفاده از برنامه معروف LINQPad

اگر علاقمند باشید که کوئری‌های خود را در محیط برنامه معروف LINQPad نیز آزمایش کنید، درایور مخصوص RavenDB آن‌را از آدرس ذیل می‌توانید دریافت نمائید:

https://github.com/ronnieoverby/RavenDB-Linqpad-Driver

عنوان: **ایندکس‌ها در RavenDB**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۱۵ ۱۴:۷
آدرس: www.dotnettips.info
برچسب‌ها: NoSQL, RavenDB

RavenDB یک Document database است و در این نوع بانک‌های اطلاعاتی، اسکیما و ساختار مشخصی وجود ندارد. شاید اینطور به نظر برسد، زمانیکه با دات نت کلاینت RavenDB کار می‌کنیم، یک سری کلاس مشخص دات نتی داشته و این‌ها ساختار اصلی کار را مشخص می‌کنند. اما در عمل RavenDB چیزی از این کلاس‌ها و خواص نمی‌داند و این کلاس‌های دات نتی صرفا کمکی هستند جهت سهولت اعمال Serialization و Deserialization اطلاعات. زمانیکه اطلاعاتی را در RavenDB ذخیره می‌کنیم، هیچ نوع قیدی در مورد ساختار نوع سندی که در حال ذخیره است، اعمال نمی‌شود.
خوب؛ اکنون این سؤال مطرح می‌شود که RavenDB چگونه اطلاعاتی را در این اسناد بدون اسکیما جستجو می‌کند؟ اینجا است که مفهوم و کاربرد ایندکس‌ها مطرح می‌شوند. ما در قسمت قبل که کوئری نویسی مقدماتی را بررسی کردیم، عملا ایندکس خاصی را به صورت دستی جهت انجام جستجوها ایجاد نکردیم؛ از این جهت که خود RavenDB به کمک امکانات dynamic indexing آن، پیشتر اینکار را انجام داده است. برای نمونه به سطر ارسال کوئری به سرور، که در قسمت قبل ارائه شد، دقت کنید. در اینجا ارسال کوئری به indexes/dynamic کاملا مشخص است:

```
Request #   2: GET     - 3,818 ms - <system>   - 200 -
/indexes/dynamic/Questions?&query=Title%3ARaven*&pageSize=128
```

Dynamic Indexes یا ایندکس‌های پویا

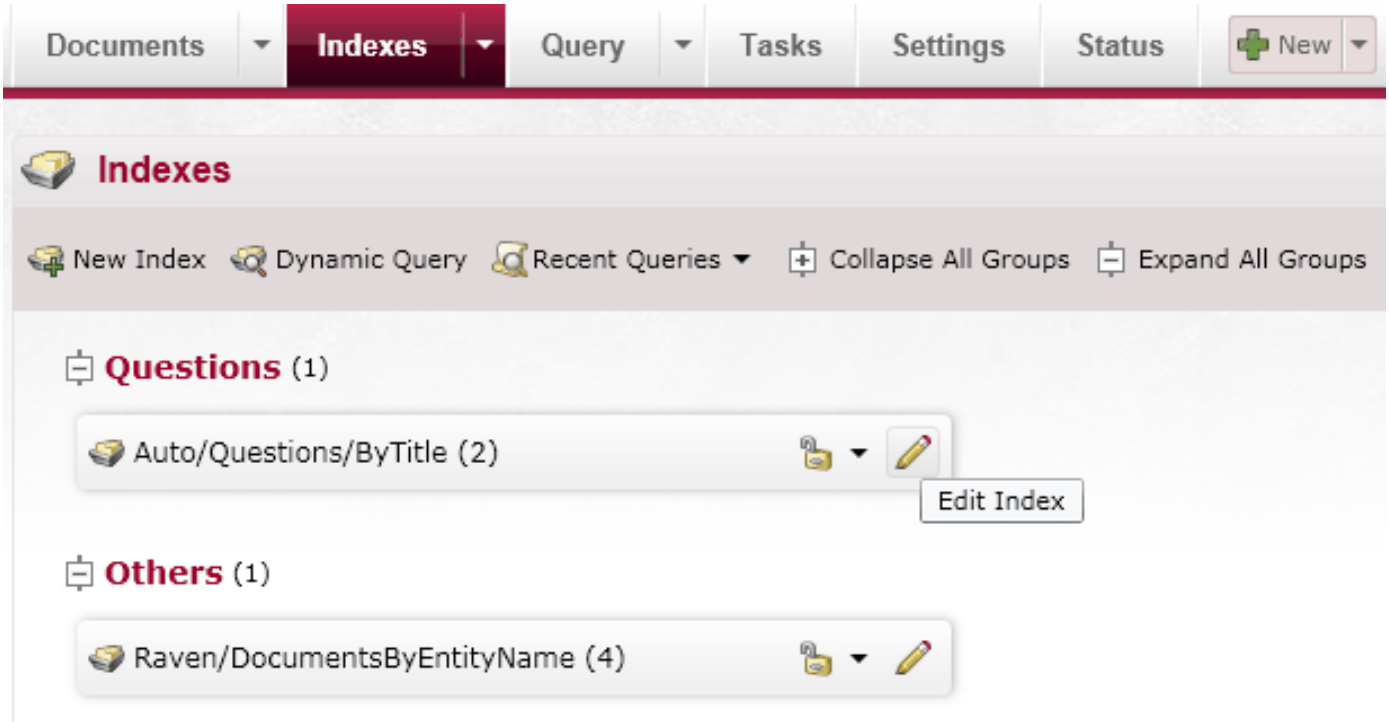
ایندکس‌های پویا زمانی ایجاد خواهند شد که ایندکس صریحی توسط برنامه نویس تعریف نگردد. برای مثال زمانیکه یک کوئری LINQ را صادر می‌کنیم، RavenDB بر این اساس و برای مثال فیلدهای قسمت Where آن، ایندکس پویایی را تولید خواهد کرد. ایجاد ایندکس‌ها در RavenDB از اصل عاقبت یک دست شدن پیروی می‌کنند. یعنی مدتی طول خواهد کشید تا کل اطلاعات بر اساس ایندکس جدیدی که در حال تهیه است، ایندکس شوند. بنابراین تولید ایندکس‌های پویا در زمان اولین بار اجرای کوئری، کوئری اول را اندکی کند جلوه خواهند داد؛ اما کوئری‌های بعدی که بر روی یک ایندکس آماده اجرا می‌شوند، بسیار سریع خواهند بود.

Static indexes یا ایندکس‌های ایستا

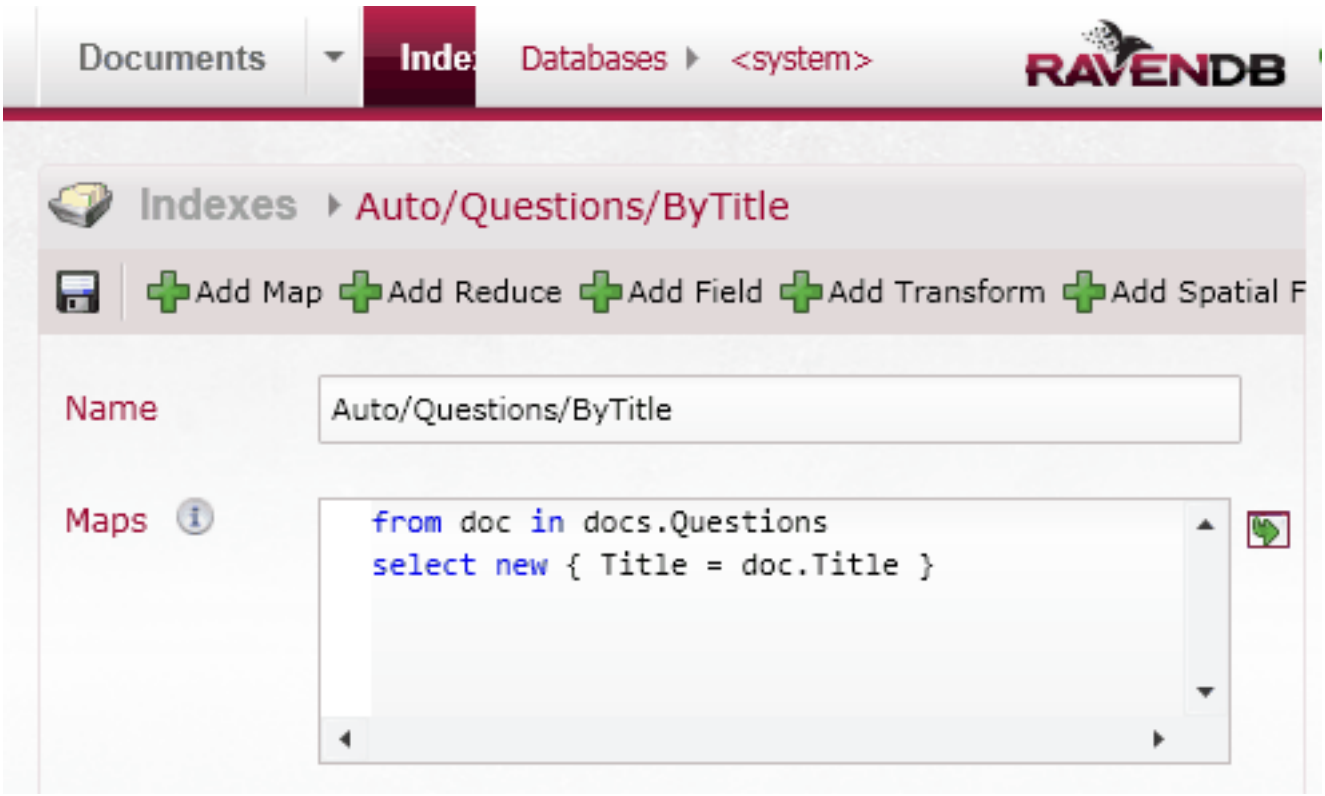
ایندکس‌های پویا به دلیل وقفه ابتدایی که برای تولید آن‌ها وجود خواهد داشت، شاید آنچنان مطلوب به نظر نرسند. اینجا است که مفهوم ایندکس‌های ایستا مطرح می‌شوند. در این حالت ما به RavenDB خواهیم گفت که چه چیزی را ایندکس کند. برای تولید ایندکس‌های ایستا، از مفاهیم Map/Reduce که در پیشنیازهای دوره جاری در مورد آن بحث شد، استفاده می‌گردد. خوشبختانه تهیه Map/Reduceها در RavenDB پیچیده نبوده و کل عملیات آن توسط کوئری‌های LINQ قابل پیاده سازی است.
تهیه ایندکس‌های پویا نیز در تردهای پس‌زمینه انجام می‌شوند. از آنجائیکه RavenDB برای اعمال Read، بهینه سازی شده است، با ارسال یک کوئری به آن، این بانک اطلاعاتی، کلیه اطلاعات آماده را در اختیار شما قرار خواهد داد؛ صرفنظر از اینکه کار تهیه ایندکس تمام شده است یا خیر.

چگونه یک ایندکس ایستا را ایجاد کنیم؟

اگر به کنسول مدیریتی سیلورلایت RavenDB مراجعه کنیم، حاصل کوئری‌های LINQ قسمت قبل را در برگه‌ی ایندکس‌های آن می‌توان مشاهده کرد:



در اینجا بر روی دکمه Edit کلیک نمائید، تا با نحوه تهیه این ایندکس پویا آشنا شویم:



این ایندکس، یک نام داشته به همراه قسمت Map از پروسه Map/Reduce که توسط یک کوئری LINQ تهیه شده است. کاری که در اینجا انجام شده، ایندکس کردن کلیه سؤالات، بر اساس خاصیت عنوان آن‌ها است.

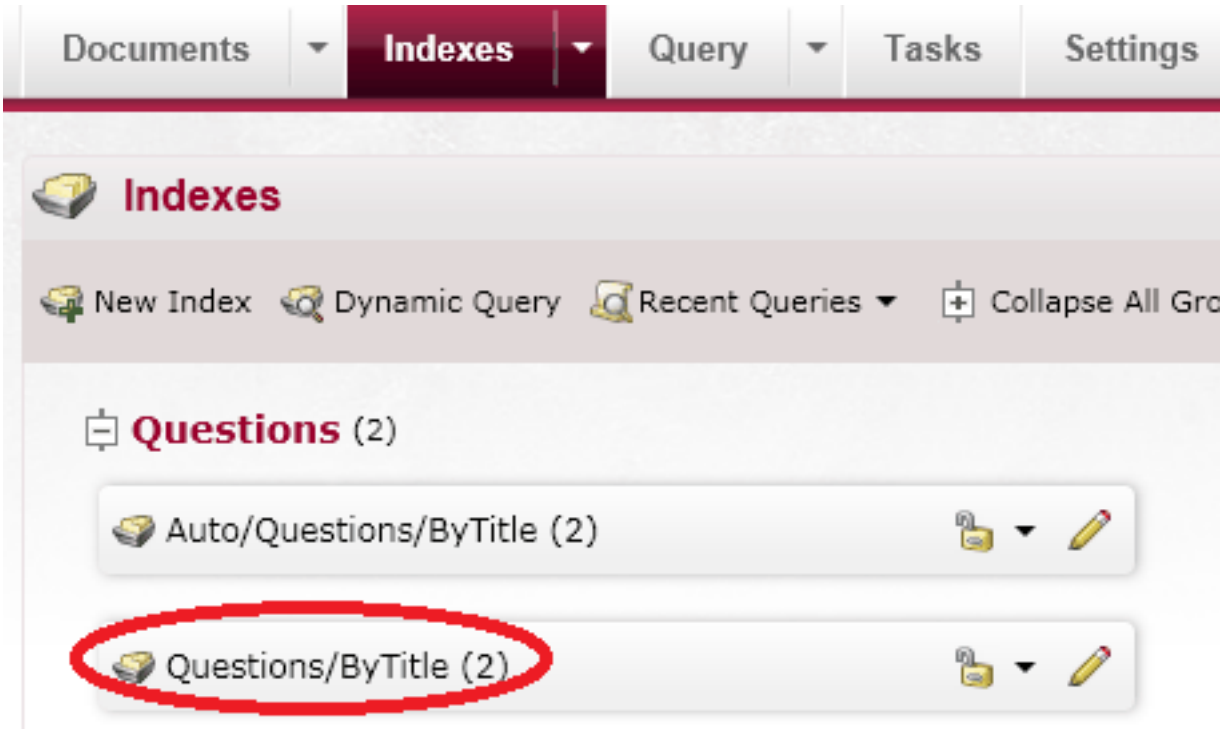
اکنون اگر بخواهیم همین کار را با کدنویسی انجام دهیم، به صورت زیر می‌توان عمل کرد:

```
using System;
using System.Linq;
using Raven.Client.Document;
using RavenDBSample01.Models;
using Raven.Client;
using Raven.Client.Linq;
using Raven.Client.Indexes;

namespace RavenDBSample01
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var store = new DocumentStore
            {
                Url = "http://localhost:8080"
            }.Initialize())
            {
                store.DatabaseCommands.PutIndex(
                name: "Questions/ByTitle",
                indexDef: new IndexDefinitionBuilder<Question>
                {
                    Map = questions => questions.Select(question => new { Title = question.Title } )
                });
            }
        }
    }
}
```

کار با شیء DatabaseCommands یک DocumentStore شروع می‌شود. سپس توسط متد PutIndex آن می‌توان یک ایندکس جدید را تعریف کرد. این متد نیاز به نام ایندکس ایجاد شده و همچنین حداقل، متد Map آن‌را دارد. برای این منظور از شیء IndexDefinitionBuilder برای تعریف نحوه جمع آوری اطلاعات ایندکس کمک خواهیم گرفت. در اینجا خاصیت Map آن‌را باید توسط یک کوئری LINQ که فیلدهای مدنظر را بازگشت می‌دهد، مقدار دهی کنیم.

برنامه را اجرا کرده و سپس به کنسول مدیریتی تحت وب RavenDB، قسمت ایندکس‌های آن مراجعه کنید. در اینجا می‌توان ایندکس جدید ایجاد شده را مشاهده کرد:



هرچند همین اعمال را در کنسول مدیریتی نیز می‌توان انجام داد، اما مزیت آن در سمت کدها، دسترسی به intellisense و نوشتن کوئری‌های strongly typed است.

روش استفاده از store.DatabaseCommands.PutIndex اولین روش تولید Index در RavenDB با کدنویسی است. روش دوم، بر اساس ارث بری از کلاس AbstractIndexCreationTask شروع می‌شود و مناسب است برای حالتیکه نمی‌خواهید کدهای تولید ایندکس، با کدهای سایر قسمت‌های برنامه مخلوط شوند:

```
public class QuestionsByTitle : AbstractIndexCreationTask<Question>
    {
        public QuestionsByTitle()
        {
            Map = questions => questions.Select(question => new { Title = question.Title });
        }
    }
```

در اینجا با ایجاد یک کلاس جدید و ارث بری از کلاس AbstractIndexCreationTask کار شروع می‌شود. سپس در سازنده این کلاس، خاصیت Map را مقدار دهی می‌کنیم. مقدار آن نیز یک کوئری LINQ است که کار Select فیلدهای شرکت دهنده در کار تهیه ایندکس را انجام می‌دهد.
اکنون برای معرفی آن به برنامه باید از متد IndexCreation.CreateIndexes استفاده کرد. این متد، نیاز به دریافت اسمبلی محل تعریف کلاس‌های تولید ایندکس را دارد. به این ترتیب تمام کلاس‌های مشتق شده از AbstractIndexCreationTask را یافته و ایندکس‌های متناظری را تولید می‌کند.

```
using (var store = new DocumentStore
            {
                Url = "http://localhost:8080"
            }.Initialize())
            {
                IndexCreation.CreateIndexes(typeof(QuestionsByTitle).Assembly, store);
            }
```

این روش، قابلیت نگهداری و نظم بهتری دارد.

استفاده از ایندکس‌های ایستای ایجاد شده

تا اینجا موفق شدیم ایندکس‌های ایستای خود را با کد نویسی ایجاد کنیم. در ادامه قصد داریم از این ایندکس‌ها در کوئری‌های خود استفاده نمائیم.

```
using (var store = new DocumentStore
            {
                Url = "http://localhost:8080"
            }.Initialize())
            {
                using (var session = store.OpenSession())
                {
                    var questions = session.Query<Question>(indexName: "QuestionsByTitle")
                                           .Where(x => x.Title.StartsWith("Raven")).Take(128);
                    foreach (var question in questions)
                    {
                        Console.WriteLine(question.Title);
                    }
                }
            }
```

استفاده از ایندکس تعریف شده نیز بسیار ساده می‌باشد. تنها کافی است نام آن‌را به متد Query ارسال نمائیم. اینبار اگر به خروجی کنسول سرور RavenDB دقت کنیم، از ایندکس indexes/QuestionsByTitle بجای ایندکس‌های پویا استفاده کرده است:

```
Request # 147: GET     -    58 ms - <system>   - 200 -
/indexes/QuestionsByTitle?&query=Title%3ARaven*&pageSize=128
```

```
        Query: Title:Raven*
        Time: 7 ms
        Index: QuestionsByTitle
        Results: 2 returned out of 2 total.
```

روش مشخص سازی نام ایندکس با استفاده از رشته‌ها، با هر دو روش store.DatabaseCommands.PutIndex و استفاده از AbstractIndexCreationTask سازگار است. اما اگر ایندکس‌های خود را با ارث بری از AbstractIndexCreationTask ایجاد کرده‌ایم، می‌توان نام کلاس مشتق شده را به صورت یک آرگومان جنریک دوم به متد Query به شکل زیر ارسال کرد تا از مزایای تعریف strongly typed آن نیز بهره‌مند شویم:

```
var questions = session.Query<Question, QuestionsByTitle>()
                                        .Where(x => x.Title.StartsWith("Raven")).Take(128);
```

ایجاد ایندکس‌های پیشرفته با پیاده سازی Map/Reduce

حالتی را در نظر بگیرید که در آن قصد داریم تعداد عنوان‌های سؤالات مانند هم را بیابیم (یا تعداد مطالب گروه‌های مختلف یک وبلاگ را محاسبه کنیم). برای انجام اینکار با سرعت بسیار بالا، می‌توانیم از ایندکس‌هایی با قابلیت محاسباتی در RavenDB استفاده کنیم. کار با ارث بری از کلاس AbstractIndexCreationTask شروع می‌شود. آرگومان جنریک اول آن، نام کلاسی است که در تهیه ایندکس شرکت خواهد داشت و آرگومان دوم (و اختیاری) ذکر شده، نتیجه عملیات Reduce است:

```
public class QuestionsCountByTitleReduceResult
    {
        public string Title { set; get; }
        public int Count { set; get; }
    }

    public class QuestionsCountByTitle : AbstractIndexCreationTask<Question,
QuestionsCountByTitleReduceResult>
    {
        public QuestionsCountByTitle()
        {
            Map = questions => questions.Select(question =>
                                                new
                                                {
                                                    Title = question.Title,
                                                    Count = 1
                                                });
            Reduce = results => results.GroupBy(x => x.Title)
                                       .Select(g =>
                                                new
                                                {
                                                    Title = g.Key,
                                                    Count = g.Sum(x => x.Count)
                                                });
        }
    }
```

در اینجا یک ایندکس پیشرفته را تعریف کرده‌ایم که در آن در قسمت Map، کار ایندکس کردن تک تک عنوان‌ها انجام خواهد شد. به همین جهت مقدار Count در این حالت، عدد یک است. در قسمت Reduce، بر روی نتیجه قسمت Map کوئری LINQ دیگری نوشته شده و تعداد عنوان‌های همانند، با گروه بندی اطلاعات، شمارش گردیده است.
اکنون برای استفاده از این ایندکس، ابتدا توسط متد IndexCreation.CreateIndexes، کار معرفی آن به RavenDB صورت گرفته و سپس متد Query سشن باز شده، دو آرگومان جنریگ را خواهد پذیرفت. اولین آرگومان، همان نتیجه Map/Reduce است و دومین آرگومان نام کلاس ایندکس جدید تعریف شده می‌باشد:

```
using (var store = new DocumentStore
            {
                Url = "http://localhost:8080"
            }.Initialize())
            {
                IndexCreation.CreateIndexes(typeof(QuestionsCountByTitle).Assembly, store);

                using (var session = store.OpenSession())
                {
```

```
                    var result = session.Query<QuestionsCountByTitleReduceResult,
QuestionsCountByTitle>()
                                        .FirstOrDefault(x => x.Title == "Raven") ?? new
QuestionsCountByTitleReduceResult();
                    Console.WriteLine(result.Count);
                }
            }
```

در کوئری فوق چون عملیات بر روی نتیجه نهایی باید صورت گیرد از FirstOrDefault استفاده شده است. این کوئری در حقیقت بر روی قسمت Reduce پیشتر محاسبه شده، اجرا می‌شود.

عنوان: **تراکنش‌ها در RavenDB**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۱۶ ۲۱:۳۵
آدرس: www.dotnettips.info
برچسب‌ها: NoSQL, RavenDB

پیش از شروع به بحث در مورد تراکنش‌ها و نحوه مدیریت آن‌ها در RavenDB، نیاز است با مفهوم ACID آشنا شویم.

ACID چیست؟

ACID از 4 قاعده تشکیل شده است (Atomic, Consistent, Isolated, and Durable) که با کنار هم قرار دادن آن‌ها یک تراکنش مفهوم پیدا می‌کند:

الف) Atomic: به معنای همه یا هیچ
اگر تراکنشی از چندین تغییر تشکیل می‌شود، همه‌ی آن‌ها باید با موفقیت انجام شوند، یا اینکه هیچکدام از تغییرات نباید فرصت اعمال نهایی را بیابند.
برای مثال انتقال مبلغ X را از یک حساب، به حسابی دیگر درنظر بگیرید. در این حالت X ریال از حساب شخص کسر و X ریال به حساب شخص دیگری واریز خواهد شد. اگر موجودی حساب شخص، دارای X ریال نباشد، نباید مبلغی از این حساب کسر شود. مرحله اول شکست خورده است؛ بنابراین کل عملیات لغو می‌شود. همچنین اگر حساب دریافت کننده بسته شده باشد نیز نباید مبلغی از حساب اول کسر گردد و در این حالت نیز کل تراکنش باید برگشت بخورد.

ب) Consistent یا یکپارچه
در اینجا consistency علاوه بر اعمال قیود، به معنای اطلاعاتی است که بلافاصله پس از پایان تراکنشی از سیستم قابل دریافت و خواندن است.

ج) Isolated: محصور شده
اگر چندین تراکنش در یک زمان با هم در حال اجرا باشند، نتیجه نهایی با حالتی که تراکنش‌ها یکی پس از دیگری اجرا می‌شوند باید یکی باشد.

د) Durable: ماندگار
اگر سیستم پایان تراکنشی را اعلام می‌کند، این مورد به معنای 100 درصد نوشته شدن اطلاعات در سخت دیسک باید باشد.

مراحل چهارگانه ACID در RavenDB به چه نحوی وجود دارند؟

RavebDB از هر دو نوع تراکنش‌های implicit و explicit پشتیبانی می‌کند. Implicit به این معنا است که در حین استفاده معمول از RavenDB (و بدون انجام تنظیمات خاصی)، به صورت خودکار مفهوم تراکنش‌ها وجود داشته و اعمال می‌شوند. برای نمونه به متد ذیل توجه نمائید:

```
public void TransferMoney(string fromAccountNumber, string toAccountNumber, decimal amount)
{
   using(var session = Store.OpenSession())
   {
         session.Advanced.UseOptimisticConcurrency = true;

         var fromAccount = session.Load<Account>("Accounts/" + fromAccountNumber);
         var toAccount = session.Load<Account>("Accounts/" + toAccountNumber);

         fromAccount.Balance -= amount;
         toAccount.Balance += amount;

         session.SaveChanges();
   }
}
```

در این متد مراحل ذیل رخ می‌دهند:
- از document store ایی که پیشتر تدارک دیده شده، جهت بازکردن یک سشن استفاده شده است.
- به سشن صراحتا عنوان شده است که از Optimistic Concurrency استفاده کند. در این حالت RavenDB اطمینان حاصل می‌کند که اکانت‌های بارگذاری شده توسط متدهای Load، تا زمان فراخوانی SaveChanges تغییر پیدا نکرده‌اند (و در غیراینصورت یک استثناء را صادر می‌کند).
- دو اکانت بر اساس Id آن‌ها از بانک اطلاعاتی واکشی می‌شوند.
- موجودی یکی تقلیل یافته و موجودی دیگر، افزایش می‌یابد.
- متد SaveChanges بر روی شیء سشن فراخوانی شده است. تا زمانیکه این متد فراخوانی نشده است، کلیه تغییرات در حافظه نگهداری می‌شوند و به سرور ارسال نخواهند شد. فراخوانی آن سبب کامل شدن تراکنش و ارسال اطلاعات به سرور می‌گردد.
بنابراین شیء سشن بیانگر یک atomic transaction ماندگار و محصور شده است (سه جزء ACID تاکنون محقق شده‌اند). محصور شده بودن آن به این معنا است که:
الف) هر تغییری که در سشن اعمال می‌شود، تا پیش از فراخوانی متد SaveChanges از دید سایر تراکنش‌ها مخفی است.
ب) اگر دو تراکنش همزمان رخ دهند، تغییرات هیچکدام بر روی دیگری اثری ندارد.

اما Consistency یا یکپارچگی در RavenDB بستگی دارد به نحوه‌ی خواندن اطلاعات و این مورد با دنیای رابطه‌ای اندکی متفاوت است که در ادامه جزئیات آن‌را بیشتر بررسی خواهیم کرد.

عاقبت یک دست شدن یا eventual consistency

درک Consistency مفهوم ACID در RavenDB بسیار مهم است و عدم آشنایی با نحوه عملکرد آن می‌تواند مشکل‌ساز شود. در دنیای بانک‌های اطلاعاتی رابطه‌ای، برنامه نویس‌ها به «immediate consistency» عادت دارند (یکپارچگی آنی). به این معنا که هرگونه تغییری در بانک اطلاعاتی، پس از پایان تراکنش، بلافاصله در اختیار کلیه خوانندگان سیستم قرار می‌گیرد. در RavenDB و خصوصا دنیای NoSQL، این یکپارچگی آنی دنیای رابطه‌ای، به «eventual consistency» تبدیل می‌شود (عاقبت یک‌دست شدن).
عاقبت یک دست شدن در RavenDB به این معنا است که اگر تغییری به یک سند اعمال گردیده و ذخیره شود؛ کوئری انجام شده بر روی این اطلاعات تغییر یافته ممکن است «stale data» باز گرداند. واژه stale در RavenDB به این معنا است که هنوز اطلاعاتی در دیتابیس موجود هستند که جهت تکمیل ایندکس‌ها پردازش نشده‌اند. به این مورد در قسمت بررسی ایندکس‌ها در RavenDB اشاره شد.
در RavenDB یک سری تردهای پشت صحنه، مدام مشغول به کار هستند و بدون کند کردن عملیات سیستم، کار ایندکس کردن اطلاعات را انجام می‌دهند. هر زمانیکه اطلاعاتی را ذخیره می‌کنیم، بلافاصله این تردها تغییرات را تشخیص داده و ایندکس‌ها را به روز رسانی می‌کنند. همچنین باید درنظر داشت که RavenDB جزو معدود بانک‌های اطلاعاتی است که خودش را بر اساس نحوه استفاده شما ایندکس می‌کند! (نمونه‌ای از آن‌را در قسمت ایندکس‌های پویای حاصل از کوئری‌های LINQ پیشتر مشاهده کرده‌اید)

نکته مهم
در RavenDB اگر از کوئری‌های LINQ استفاده کنیم، ممکن است به علت اینکه هنوز تردهای پشت صحنه‌ی ایندکس سازی اطلاعات، کارشان تمام نشده است، تمام اطلاعات یا آخرین اطلاعات را دریافت نکنیم (که به آن stale data گفته می‌شود). هر آنچه که ایندکس شده است دریافت می‌گردد (مفهوم عاقبت یک دست شدن ایندکس‌ها). اما اگر نیاز به یکپارچگی آنی داشتیم، متد Load یک سشن، مستقیما به بانک اطلاعاتی مراجعه می‌کند و اطلاعات بازگشت داده شده توسط آن هیچگاه احتمال stale بودن را ندارند.
بنابراین برای نمایش اطلاعات یا گزارشگیری، از کوئری‌های LINQ استفاده کنید. RavenDB خودش را بر اساس کوئری شما ایندکس خواهد کرد و نهایتا به کوئری‌هایی فوق العاده سریعی در طول کارکرد سیستم خواهیم رسید. اما در صفحه ویرایش اطلاعات بهتر است از متد Load استفاده گردد تا نیاز به مفهوم immediate consistency یا یکپارچگی آنی برآورده شود.

تنظیمات خاص کار با ایندکس سازها برای انتظار جهت اتمام کار آن‌ها

عنوان شد که اگر ایندکس سازهای پشت صحنه هنوز کارشان تمام نشده است، در حین کوئری گرفتن، هر آنچه که ایندکس شده بازگشت داده می‌شود.

در اینجا می‌توان به RavenDB گفت که تا چه زمانی می‌تواند یک کوئری را جهت دریافت اطلاعات نهایی به تاخیر بیندازد. برای اینکار باید اندکی کوئری‌های LINQ آن‌را سفارشی سازی کنیم:

```
RavenQueryStatistics stats;
var results = session.Query<Product>()
    .Statistics(out stats)
    .Where(x => x.Price > 10)
    .ToArray();

if (stats.IsStale)
{
    // Results are known to be stale
}
```

توسط امکانات آماری کوئری‌های LINQ در RavenDB مطابق کدهای فوق، می‌توان دریافت که آیا اطلاعات دریافت شده stale است یا خیر.
همچنین زمان انتظار تا پایان کار ایندکس ساز را نیز توسط متد Customize به نحو ذیل می‌توان تنظیم کرد:

```
RavenQueryStatistics stats;
var results = session.Query<Product>()
    .Statistics(out stats)
    .Where(x => x.Price > 10)
    .Customize(x => x.WaitForNonStaleResults(TimeSpan.FromSeconds(5)))
    .ToArray();
```

به علاوه می‌توان کلیه کوئری‌های یک documentStore را وارد به صبر کردن تا پایان کار ایندکس سازی کرد (متد Customize پیش فرضی را با WaitForNonStaleResultsAsOfLastWrite مقدار دهی و اعمال می‌کند):

```
documentStore.Conventions.DefaultQueryingConsistency = ConsistencyOptions.QueryYourWrites;
```

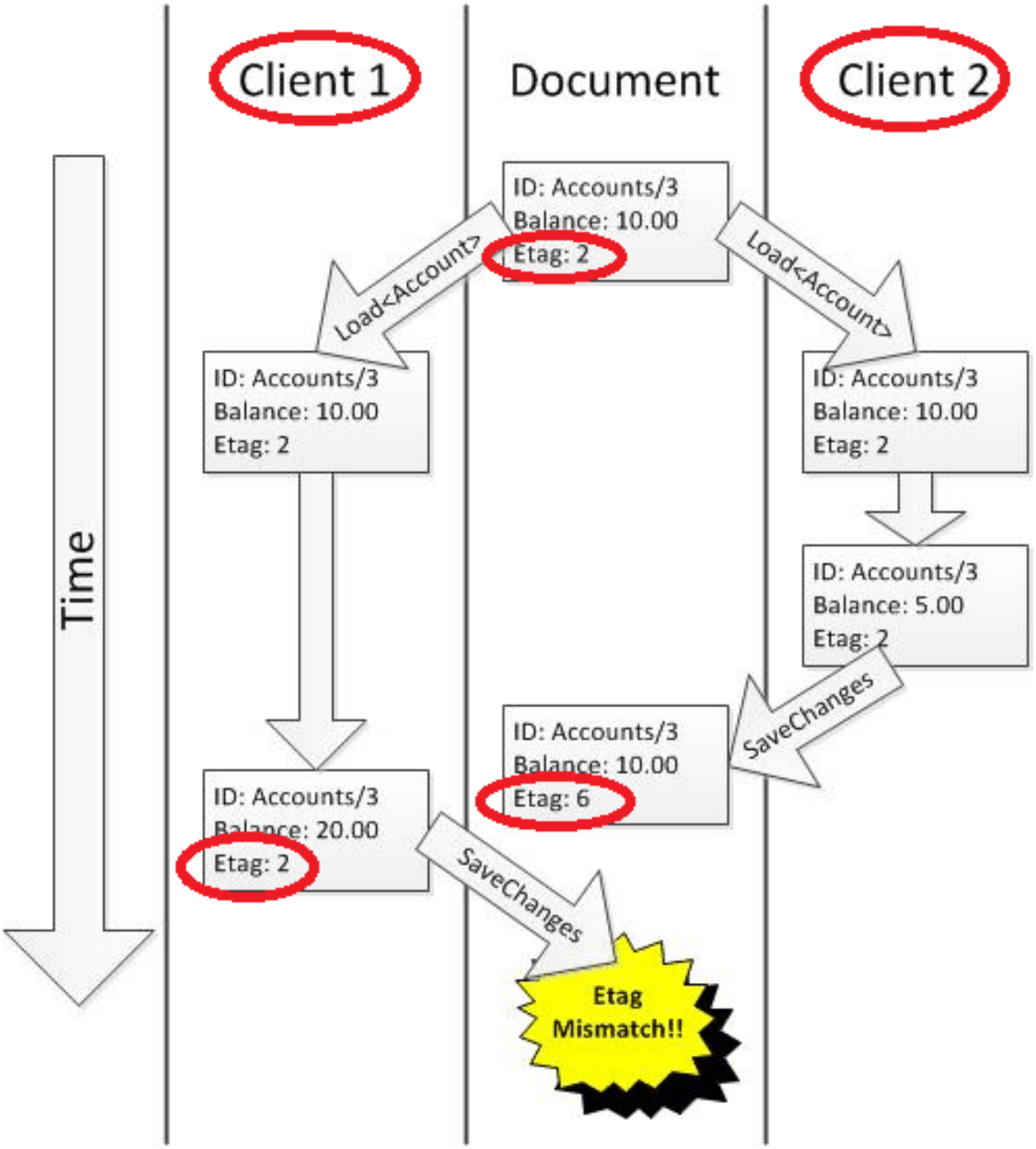
این مورد در برنامه‌های وب توصیه نمی‌شود چون کل سیستم در حین آغاز کار با آن بر اساس یک documentStore سینگلتون باید کار کند و همین مساله صبر کردن‌ها، با بالا رفتن حجم اطلاعات و تعداد کاربران، پاسخ دهی سیستم را تحت تاثیر قرار خواهد داد.
به علاوه این تنظیم خاص بر روی کوئری‌های پیشرفته Map/Reduce کار نمی‌کند. در این نوع کوئری‌های ویژه، برای صبر کردن تا پایان کار ایندکس شدن، می‌توان از روش زیر استفاده کرد:

```
while (documentStore.DatabaseCommands.GetStatistics().StaleIndexes.Length != 0)
{
    Thread.Sleep(10);
}
```

مقابله با تداخلات همزمانی

با تنظیم session.Advanced.UseOptimisticConcurrency = true، اگر سندی که در حال ویرایش است، در این حین توسط کاربر دیگری تغییر کرده باشد، استثنای ConcurrencyException صادر خواهد شد. همچنین این استثناء در صورتیکه شخصی قصد بازنویسی سند موجودی را داشته باشد نیز صادر خواهد شد (شخصی بخواهد سندی را با ID سند موجودی ذخیره کند). اگر از optimistic concurrency استفاده نشود، آخرین ترد نویسنده یا به روز کننده اطلاعات، برنده خواهد شد و اطلاعات نهایی موجود در بانک اطلاعاتی متعلق به او و حاصل بازنویسی آن ترد است.
optimistic concurrency به زبان ساده به معنای به خاطر سپردن شماره نگارش یک سند است، زمانیکه آن‌را بارگذاری می‌کنیم و سپس ارسال آن به سرور، زمانیکه قصد ذخیره آن‌را داریم. در SQL Server اینکار توسط RowVersion انجام می‌شود. در بانک‌های اطلاعاتی سندگرا چون تمایل به استفاده از HTTP در آن‌ها زیاد است (مانند RavenDB) از مکانیزمی به نام E-Tag برای این منظور کمک گرفته می‌شود. هر زمانیکه تغییری به یک سند اعمال می‌شود، E-Tag آن به صورت خودکار افزایش خواهد یافت.
برای مثال فرض کنید کاربری سندی را با E-Tag مساوی 2 بارگذاری کرده است. قبل از اینکه این کاربر در صفحه ویرایش اطلاعات کارش با این سند خاتمه یابد، کاربر دیگری در شبکه، این سند را ویرایش کرده است و اکنون E-Tag آن مثلا مساوی 6 است. در این زمان اگر کاربر یک سعی به ذخیره سازی اطلاعات نماید، چون E-Tag سند او با E-Tag سند موجود در سرور دیگر یکی نیست، با

استثنای ConcurrencyException متوقف خواهد شد.



مشکل! در برنامه‌های بدون حالت وب، چون پس از نمایش صفحه ویرایش اطلاعات، سشن RavenDB نیز بلافاصله Dispose خواهد شد، این E-Tag را از دست خواهیم داد. همچنین باید دقت داشت که سشن RavenDB به هیچ عنوان نباید در طول عمر یک برنامه باز نگهداشته شود و برای طول عمری کوتاه طراحی شده است. راه حلی که برای آن درنظر گرفته شده است، ذخیره سازی این E-Tag در بار اول دریافت آن از سشن می‌باشد. برای این منظور تنها کافی است خاصیتی را به نام Etag با ویژگی JsonIgnore (که

سبب عدم ذخیره سازی آن در بانک اطلاعاتی خواهد شد) تعریف کنیم:

```
public class Person
{
    public string Id { get; set; }

    [JsonIgnore]
    public Guid? Etag { get; set; }

    public string Name { get; set; }
}
```

اکنون زمانیکه سندی را از بانک اطلاعاتی دریافت می‌کنیم، با استفاده از متد session.Advanced.GetEtagFor، می‌توان این Etag واقعی را دریافت کرد و ذخیره نمود:

```
public Person Get(string id)
{
    var person = session.Load<Person>(id);
    person.Etag = session.Advanced.GetEtagFor(person);
    return person;
}
```

و برای استفاده از آن ابتدا باید UseOptimisticConcurrency به true تنظیم شده و سپس در متد Store این Etag دریافتی از سرور را مشخص نمائیم:

```
public void Update(Person person)
{
    session.Advanced.UseOptimisticConcurrency = true;
    session.Store(person, person.Etag, person.Id);
    session.SaveChanges();
    person.Etag = session.Advanced.GetEtagFor(person);
}
```

تراکنش‌های صریح

همانطور که عنوان شد، به صورت ضمنی کلیه سشن‌ها، یک واحد کار را تشکیل داده و با پایان آن‌ها، تراکنش خاتمه می‌یابد. اگر به هر علتی قصد تغییر این رفتار ضمنی پیش فرض را دارید، امکان تعریف صریح تراکنش‌های نیز وجود دارد:

```
using (var transaction = new TransactionScope())
{
   using (var session1 = store.OpenSession())
   {
     session1.Store(new Account());
     session1.SaveChanges();
   }

   using (var session2 = store.OpenSession())
   {
     session2.Store(new Account());
     session2.SaveChanges();
   }

   transaction.Complete();
}
```

باید دقت داشت که پایان یک تراکنش، یک non-blocking asynchronous call است و مباحث stale data که پیشتر در مورد آن بحث شد، برقرار هستند.

عنوان: طراحی روابط و ارجاعات در RavenDB
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۱۸ ۱:۰
آدرس: www.dotnettips.info
گروه‌ها: NoSQL, RavenDB

در قسمت‌های قبل، با پیش زمینه‌ی ذهنی طراحی مدل‌های RavenDB به همراه اصول مقدماتی کوئری نویسی آن آشنا شدیم. در این قسمت قصد داریم معادل‌های روابط موجود در بانک‌های اطلاعاتی رابطه‌ای را در RavenDB و مطابق ذهنیت غیر رابطه‌ای آن، مدلسازی کنیم و مثال‌های بیشتری را بررسی نمائیم.

مدیریت روابط در RavenDB

یکی از اصول طراحی مدل‌ها در RavenDB، مستقل بودن اسناد یا documents است. به این ترتیب کلیه اطلاعاتی که یک سند نیاز دارد، داخل همان سند ذخیره می‌شوند (به این نوع شیء، Root Aggregate هم گفته می‌شود). اما این اصل سبب نخواهد شد تا نتوان یا نباید ارتباطی را بین اسناد تعریف کرد. بنابراین سؤال مهم اینجا است که چه اطلاعات مرتبطی باید داخل یک سند ذخیره شوند و چه اطلاعاتی باید به سند دیگری ارجاع داده شوند. برای پاسخ به این سؤال سه روش ذیل را باید مدنظر داشت:

الف) Denormalized references

فرض کنید در دنیای رابطه‌ای دو جدول سفارش و مشتری را دارید. در این حالت، جدول سفارش تنها شماره آی دی اطلاعات مشتری را از جدول مشتری یا کاربران سیستم، در خود ذخیره خواهد کرد. به این ترتیب از تکرار اطلاعات مشتری در جدول سفارشات جلوگیری می‌گردد. اما اگر اطلاعات پرکاربرد مشتری را در داخل جدول سفارش قرار دهیم به آن denormalized reference گفته می‌شود.

ایجاد denormalized reference یکی از روش‌های مرسوم در دنیای NoSQL و RavenDB است؛ خصوصا جهت سهولت نمایش اطلاعات. به این ترتیب ارجاع به سندهای دیگر کمتر شده و ترافیک شبکه نیز کاهش می‌یابد. برای مثال در اینجا نام و آدرس مشتری را داخل سند ثبت شده قرار می‌دهیم و از سایر اطلاعات او (که اهمیت نمایشی ندارند) مانند کلمه عبور و امثال آن صرفنظر خواهیم کرد.

اینجا است که یک سری از سؤالات مطرح خواهند شد مانند : «اگر آدرس مشتری تغییر کرد، چطور؟»

بنابراین بهترین حالت استفاده از روش denormalized references محدود خواهد شد به موارد ذیل:

الف) قید اطلاعاتی که به ندرت تغییر می‌کنند. برای مثال نام یک شخص یا نام یک کشور، استان یا شهر.

ب) ثبت اطلاعات تکراری که در طول زمان تغییر می‌کنند، اما باید تاریخچه‌ی آن‌ها حفظ شوند. برای مثال اگر آدرس مشتری تغییر کرده است، واقعا اجناس سندهای قبلی او، صرفنظر از آدرس جدیدی که اعلام کرده است، به آدرس قبلی او ارسال شده‌اند و این تاریخچه باید در سیستم حفظ شوند.

ج) اطلاعاتی که ممکن است بعدها حذف شوند؛ اما نیاز است سابقه اسناد قبلی تخریب نشوند. برای مثال کارخانه‌ای را درنظر بگیرید که امسال یک سری چینی خاص را تولید می‌کند و می‌فروشد. سال بعد خط تولید خود را عوض کرده و سری اجناس دیگری را شروع به تولید و فروش خواهد کرد. در بانک‌های اطلاعاتی رابطه‌ای نمی‌توان اجناسی را که در جداول دیگر ارجاع دارند، به این سادگی‌ها حذف کرد. در اینجا باید از روش‌هایی مانند تعریف فیلد بیتی IsDeleted برای مخفی کردن ظاهری رکوردهای موجود کمک گرفت. اما در دنیای رابطه‌ای، اطلاعات مهم محصول را در سند اصلی ثبت کنید. بعد هر زمانیکه نیازی به محصول نبود، کلا تعریف آن‌را حذف نمائید.

ب) Includes

Includes در RavenDB برای پوشش مشکلات denormalization ارائه شده است. در اینجا بجای اینکه یک شیء کپی اطلاعات پرکاربرد شیءایی دیگر را در خود ذخیره کند، تنها ارجاعی (یک Id رشته‌ای) از آن شیء را در سند مرتبط ذخیره خواهد کرد.

```
public class Order
{
    public string CustomerId { get; set; }
    public LineItem[] LineItems { get; set; }
    public double TotalPrice { get; set; }
}

public class Customer
```

```
{
    public string Name { get; set; }
    public string Address { get; set; }
    public short Age { get; set; }
    public string HashedPassword { get; set; }
}
```

برای نمونه در کلاس Order شاهد یک Id رشته‌ای ارجاع دهنده به کلاس Customer هستیم. هرگاه که نیاز به بارگذاری اطلاعات شیء Order به همراه کل اطلاعات مشتری او تنها در یک رفت و برگشت به بانک اطلاعاتی باشد، می‌توان از متد الحاقی Include مختص RavenDB استفاده کرد:

```
var order = session.Include<Order>(x => x.CustomerId)
                   .Load("orders/1234");

این کوئری از کش سشن خوانده می‌شود و کاری به سرور ندارد //
var cust = session.Load<Customer>(order.CustomerId);
```

همانطور که مشاهده می‌کنید، با ذکر متد Include، اعلام کرده‌ایم که مایل هستیم تا اطلاعات سند مشتری متناظر را نیز داشته باشیم. در این حالت در Load بعدی که بر اساس Id مشتری انجام شده، دیگر رفت و برگشتی به سرور انجام نشده و اطلاعات مشتری از کش سشن جاری که پیشتر با فراخوانی Include مقدار دهی شده است، دریافت می‌گردد.
حتی می‌توان چند سند مرتبط را با هم بارگذاری کرد؛ با حداقل رفت و برگشت به سرور:

```
var orders = session.Include<Order>(x => x.CustomerId)
    .Load("orders/1234", "orders/4321");

foreach (var order in orders)
{
    // این کوئری‌ها سمت کلاینت هستند و به سرور ارسال نمی‌شوند
    var cust = session.Load<Customer>(order.CustomerId);
}
```

همچنین امکان استفاده از متد Include در LINQ API نیز پیش بینی شده است. برای این منظور باید از متد Customize استفاده کرد:

```
var orders = session.Query<Order>()
    .Customize(x => x.Include<Order>(o => o.CustomerId))
    .Where(x => x.TotalPrice > 100)
    .ToList();

foreach (var order in orders)
{
    // این کوئری‌ها سمت کلاینت اجرا می‌شوند
    var cust = session.Load<Customer>(order.CustomerId);
}
```

Includeهای یک به چند

اکنون فرض کنید به کلاس سفارش، آرایه تامین کننده‌ها نیز افزوده شده است (رابطه یک به چند):

```
public class Order
{
    public string CustomerId { get; set; }
    public string[] SupplierIds { get; set; }
    public LineItem[] LineItems { get; set; }
    public double TotalPrice { get; set; }
}
```

بارگذاری یکباره روابط یک به چند نیز با Include میسر است:

```
var orders = session.Include<Order>(x => x.SupplierIds)
    .Load("orders/1234", "orders/4321");

foreach (var order in orders)
{
    foreach (var supplierId in order.SupplierIds)
    {
        // از کش سشن خوانده می‌شود
        var supp = session.Load<Supplier>(supplierId);
    }
}
```

Includeهای چند سطحی

در اینجا کلاس سفارشی را در نظر بگیرید که دارای خاصیت ارجاع دهنده نیز هست. این خاصیت به شکل یک کلاس تعریف شده است و نه به شکل یک آی دی رشته‌ای:

```
public class Order
{
    public string CustomerId { get; set; }
    public string[] SupplierIds { get; set; }
    public Referral Refferal { get; set; }
    public LineItem[] LineItems { get; set; }
    public double TotalPrice { get; set; }
}

public class Referral
{
    public string CustomerId { get; set; }
    public double CommissionPercentage { get; set; }
}
```

متد Include امکان ارجاع به خواص تو در تو را نیز دارد:

```
var order = session.Include<Order>(x => x.Refferal.CustomerId)
    .Load("orders/1234");

// از کش سشن خوانده می‌شود
var referrer = session.Load<Customer>(order.Refferal.CustomerId);
```

همچنین این متد با مجموعه‌ها نیز کار می‌کند. برای مثال اگر تعریف متد LineItem به صورت زیر باشد:

```
public class LineItem
{
    public string ProductId { get; set; }
    public string Name { get; set; }
    public int Quantity { get; set; }
    public double Price { get; set; }
}
```

برای بارگذاری یکباره اسناد مرتبط می‌توان به روش ذیل عمل کرد:

```
var order = session.Include<Order>(x => x.LineItems.Select(li => li.ProductId))
    .Load("orders/1234");

foreach (var lineItem in order.LineItems)
{
    // از کش سمت کلاینت خوانده می‌شود
    var product = session.Load<Product>(lineItem.ProductId);
}
```

و به صورت خلاصه برای باگذاری اسناد مرتبط، دیگر از دو کوئری پشت سر هم ذیل استفاده نکنید:

```
var order = session.Load<Order>("orders/1");
var customer = session.Load<Customer>(order.CustomerId);
```

این دو کوئری یعنی دوبار رفت و برگشت به سرور. با استفاده از Include می‌توان تعداد رفت و برگشت‌ها و همچنین ترافیک شبکه را کاهش داد. به علاوه سرعت کار نیز افزایش خواهد یافت.

ج) تفاوت بین Reference و Relationship

برای درک اینکه آیا اطلاعات یک شیء مرتبط را بهتر است داخل شیء اصلی (Aggregate rooe) ذخیره کرد یا خیر، باید مفاهیم ارجاع و ارتباط را بررسی کنیم.
اگر به مثال سفارش و مشتری دقت کنیم، یک سفارش را بدون مشتری نیز می‌توان تکمیل کرد. برای مثال بسیاری از فروشگاه‌ها به همین نحو عمل می‌کنند و اگر شماره Id مشتری را به سندی اضافه می‌کنیم، صرفا جهت این است که بدانیم این سند متعلق به شخص دیگری نیست. بنابراین «ارجاعی» به کاربر در جدول سفارش می‌تواند وجود داشته باشد.
اکنون اقلام سفارش را درنظر بگیرید. هر آیتم سفارش تنها با بودن آن سفارش خاص است که معنا پیدا می‌کنند و نه بدون آن. این آیتم می‌تواند ارجاعی به محصول مرتبط داشته باشد. اینجا است که می‌گوییم اقلام سند با سفارش «در ارتباط» هستند؛ اما یک سند ارجاعی دارد به مشتری.
از این دو مفهوم برای تشخیص تشکیل Root Aggregate استفاده می‌شود. به این ترتیب تشخیص داده‌ایم اقلام سند، Root Aggregate را تشکیل می‌دهند؛ بنابراین ذخیره سازی تمام آن‌ها داخل یک سند RavenDB معنا پیدا می‌کند.

چند مثال برای درک بهتر نحوه طراحی اسناد در RavenDB

الف) Stackoverflow
صفحه نمایش یک سؤال و پاسخ‌های آن و همچنین رای‌های هر آیتم را درنظر بگیرید. در اینجا کاربران همزمانی ممکن است به یک سؤال رای بدهند، پاسخ‌هایی را ارائه دهند و یا کاربر اصلی، سؤال خویش را ویرایش کند. به این ترتیب با قرار دادن کلیه آیتم‌های این سند داخل آن، به مشکلات همزمانی برخواهیم خورد. برای مثال واقعا نمی‌خواهیم که به علت افزوده شدن یک پاسخ، کل سند قفل شود.
بنابراین ذخیره سازی سؤال در یک سند و ذخیره سازی لیست پاسخ‌ها در سندی دیگر، طراحی بهتری خواهد بود.

ب) سبد خرید و آیتم‌های آن
زمانیکه کاربری مشغول به خرید آنلاین از سایتی می‌شود، لیست اقلام انتخابی او یک سفارش را تشکیل داده و به تنهایی معنا پیدا نمی‌کنند. به همین جهت ذخیره سازی اقلام سفارش به صورت یک Root aggregate در اینجا مفهوم داشته و متداول است.

ج) یک بلاگ و کامنت‌های آن
در اینجا نیز کاربران، مجزای از مطلب اصلی ارسال شده ممکن است نظرات خود را ویرایش کنند یا اینکه بخواهیم نظرات را جداگانه لیست کنیم. بنابراین این دو (مطالب و نظرات) موضوعاتی جداگانه بوده و نیازی نیست به صورت یک Root aggregate تعریف شوند.

بنابراین در حین طراحی اسناد NoSQL باید به اعمال و «محدوده‌های تراکنشی» انجام شده دقت داشت تا اینکه صرفا عنوان شود این یک رابطه یک به چند یا چند به چند است.

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۲۴ ۰:۴۳

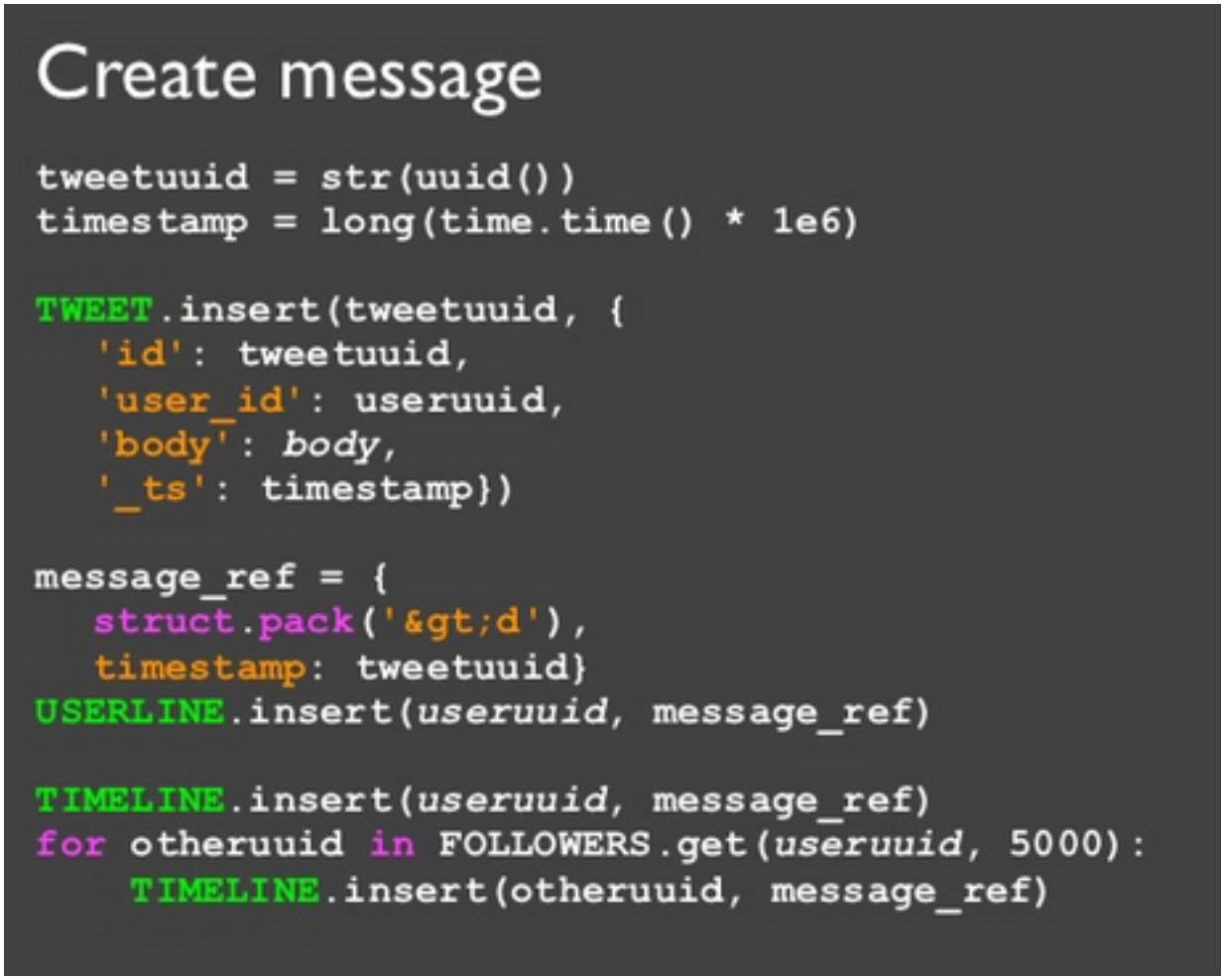
یک مثال تکمیلی

شبیه سازی سایت Stackoverflow با RavenDB

توضیحات مفصل آن به صورت یک ویدیو

نویسنده: سینا شهرکی
تاریخ: ۱۳۹۳/۱۱/۰۳ ۲۰:۴۱

با سلام.
یک سوال در خصوص طراحی روابط یک بلاگ دارم:
فرض کنید میخواهیم بخش تایم لاین را به برنامه اضافه کنیم بدین شرح: هر کاربری بتواند مشترک بلاگ‌های مورد علاقه اش شود و هر بار که به صفحه اول برنامه مراجعه میکند جدیدترین پست‌های بلاگ هایی که مشترک آنها بوده ببیند.
شاید یک نوع طراحی اینگونه باشد که جدولی داشته باشیم به نام «اشتراک» که در آن فیلد «نام کاربری نویسنده و نام کاربری مشترک» مورد نظر درج شود. سپس یک جدول هم داشته باشیم مثلاً به نام Timeline با فیلدهای «نام کاربری نویسنده، نام کاربری گیرنده، متن کامل مطلب و تاریخ ارسال».
هر زمان مطلب جدیدی منتشر شد، به ازای هر مشترک در جدول اشتراک، یک رکورد در جدول Timeline درج شود، در این حالت کار خواندن مطالب جدیدی که باید به کاربر نشان دهیم ساده میشود اما اگر یک کاربر مثلاً 10000 تا مشترک داشته باشد پس به ازای هر مطلب جدیدی که مینویسد باید 10000 رکورد در جدول Timeline درج شود، و اگر 100 نفر بخواهند مطلب بنویسند فکر کنم سیستم از کار بی افتد.
ممکن است من را راهنمایی کنید.
ویرایش:
در این لینک صفحه 26 مطلب زیر رو هم پیدا کرم.



نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۰۳ ۲۱:۳

Timeline یک جدول نیست؛ یک گزارش هست از اطلاعات موجود (گزارش از اینکه یک کاربر به چه بلاگ‌هایی علاقمند است). به ازای هر View جدید مورد نیاز از بانک اطلاعاتی، یک جدول جدید ایجاد نمی‌کنند. همچنین در اینجا چیزی به نام جدول نداریم. این بانک اطلاعاتی، سندگرا است. هر رکورد آن یک سند JSON است و مجموعه‌ای از آن‌ها تا حدودی شبیه به یک جدول بانک اطلاعاتی رابطه‌ای است (تا حدودی از این جهت که هر سند JSON آن می‌تواند ساختار متفاوتی با قبلی داشته باشد، یا نداشته باشد؛ بسته به انتخاب و طراحی). در مورد «denormalized references» در متن بحث شده: «... بنابراین بهترین حالت استفاده از روش denormalized references محدود خواهد شد به موارد ذیل ... ». یعنی همه جا قرار نیست کار رفع نرمال سازی در بانک‌های اطلاعاتی NoSQL سندگرا انجام شود. سه مورد مهم دارد که در بحث ذکر شده‌است.

در اینجا برای طراحی حالت بلاگ‌های مورد علاقه یک شخص در RavenDB فقط کافی است از مفهوم Includes آن استفاده کنید (نمونه آن «Includeهای یک به چند» در بحث). داخل کلاس User، یک آرایه شبیه به SupplierIds (مثال زده شده) به نام FavoriteBlogIds خواهید داشت. بارگذاری و گزارشگیری از آن برای نمایش لیست این بلاگ‌ها و سپس مطالب آن‌ها، مانند مثال‌های Include و Load ایی است که ارائه شد.

بنابراین در اینجا به چیزی مانند دو جدول مجزای کاربران و جدول ذخیره سازی لیست بلاگ‌های محبوب آن‌ها نیازی نیست. لیست و آرایه Idهای بلاگ‌های مورد علاقه‌ی یک کاربر، داخل سند JSON همان کاربر قرار می‌گیرد.

عنوان: **نگاهی به افزونه‌های کار با اسناد در RavenDB**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۱۹ ۰:۳۷
آدرس: www.dotnettips.info
برچسب‌ها: NoSQL, RavenDB

توانمندی‌های RavenDB جهت کار با اسناد، صرفا به ذخیره و ویرایش آن‌ها محدود نمی‌شوند. در ادامه، مباحثی مانند پیوست فایل‌های باینری به اسناد، نگهداری نگارش‌های مختلف آن‌ها، حذف آبشاری اسناد و وصله کردن آن‌ها را مورد بررسی قرار خواهیم داد. تعدادی از این قابلیت‌ها توکار هستند و تعدادی دیگر توسط افزونه‌های آن فراهم شده‌اند.

پیوست و بازیابی فایل‌های باینری

امکان پیوست فایل‌های باینری نیز به اسناد RavenDB وجود دارد. برای مثال به کلاس سؤالات قسمت اول این سری، خاصیت FileId را اضافه کنید:

```
public class Question
{
    public string FileId { set; get; }
}
```

اکنون برای ذخیره فایلی و همچنین انتساب آن به یک سند، به روش ذیل باید عمل کرد:

```
using (var store = new DocumentStore
            {
                Url = "http://localhost:8080"
            }.Initialize())
            {
                using (var session = store.OpenSession())
                {
                    store.DatabaseCommands.PutAttachment(key: "file/1",
                                                         etag: null,
                                                         data:
System.IO.File.OpenRead(@"D:\Prog\packages.config"),
                                                         metadata: new RavenJObject
                                                         {
                                                            { "Description", "توضیحات فایل" }
                                                         });

                    var question = new Question
                    {
                        By = "users/Vahid",
                        Title = "Raven Intro",
                        Content = "Test....",
                        FileId = "file/1"
                    };
                    session.Store(question);

                    session.SaveChanges();
                }
            }
```

کار متد store.DatabaseCommands.PutAttachment، ارسال اطلاعات یک استریم به سرور RavenDB است که تحت کلید مشخصی ذخیره خواهد شد. متد استاندارد System.IO.File.OpenRead روش مناسبی است برای دریافت استریم‌ها و ارسال آن به متد PutAttachment. در قسمت metadata این فایل، توسط شیء RavenJObject، یک دیکشنری از key-valueها را جهت درج اطلاعات اضافی مرتبط با هر فایل می‌توان مقدار دهی کرد. پس از آن، جهت انتساب این فایل ارسال شده به یک سند، تنها کافی است کلید آن‌را به خاصیت FileId انتساب دهیم.
در این حالت اگر به خروجی دیباگ سرور نیز دقت کنیم، مسیر ذخیره سازی این نوع فایل‌ها مشخص می‌شود:

```
Request # 2: PUT   - 200 ms - <system> - 201 - /static/file/1
```

بازیابی فایل‌های همراه با اسناد نیز بسیار ساده است:

```
using (var store = new DocumentStore
            {
                Url = "http://localhost:8080"
            }.Initialize())
            {
                using (var session = store.OpenSession())
                {
                    var question = session.Load<Question>("questions/97");
                    var file1 = store.DatabaseCommands.GetAttachment(question.FileId);
                    Console.WriteLine(file1.Size);
                }
            }
```

فقط کافی است سند را یکبار Load کرده و سپس از متد store.DatabaseCommands.GetAttachment برای دستیابی به فایل پیوست شده استفاده نمائیم.

وصله کردن اسناد

سند سؤالات قسمت اول و پاسخ‌های آن، همگی داخل یک سند هستند. اکنون برای اضافه کردن یک آیتم به این لیست، یک راه، واکشی کل آن سند است و سپس افزودن یک آیتم جدید به لیست پاسخ‌ها و یا در این حالت، جهت کاهش ترافیک سرور و سریعتر شدن کار، RavenDB مفهوم Patching یا وصله کردن اسناد را ارائه داده است. در این روش بدون واکشی کل سند، می‌توان قسمتی از سند را وصله کرد و تغییر داد.

```
using (var store = new DocumentStore
            {
                Url = "http://localhost:8080"
            }.Initialize())
            {
                using (var session = store.OpenSession())
                {
                    store.DatabaseCommands.Patch(key: "questions/97",
                                                 patches: new[]
                                                          {
                                                             new PatchRequest
                                                             {
                                                                Type = PatchCommandType.Add,
                                                                Name = "Answers",
                                                                Value = RavenJObject.FromObject(new
Answer{ By= "users/Vahid", Content="data..."})
                                                             }
                                                          });

                }
            }
```

برای وصله کردن اسناد از متد store.DatabaseCommands.Patch استفاده می‌شود. در اینجا ابتدا Id سند مورد نظر مشخص شده و سپس آرایه‌ای از تغییرات لازم را به صورت اشیاء PatchRequest ارائه می‌دهیم. در هر PatchRequest، خاصیت Type مشخص می‌کند که حین عملیات وصله کردن چه کاری باید صورت گیرد؛ برای مثال اطلاعات ارسالی اضافه شوند یا ویرایش و امثال آن. خاصیت Name، نام خاصیت در حال تغییر را مشخص می‌کند. برای مثال در اینجا می‌خواهیم به مجموعه پاسخ‌های یک سند، آیتم جدیدی را اضافه کنیم. خاصیت Value، مقدار جدید را دریافت خواهد کرد. این مقدار باید با فرمت JSON تنظیم شود؛ به همین جهت از متد توکار RavenJObject.FromObject برای اینکار استفاده شده است.

افزونه‌های RavenDB

قابلیت‌های ذکر شده فوق جهت کار با اسناد به صورت توکار در RavenDB مهیا هستند. این سیستم افزونه پذیر است و تاکنون افزونه‌های متعددی برای آن تهیه شده‌اند که در اینجا به آن‌ها Bundles گفته می‌شوند. برای استفاده از آن‌ها تنها کافی است فایل DLL مرتبط را درون پوشه Plugins سرور، کپی کنیم. دریافت آن‌ها نیز از طریق NuGet پشتیبانی می‌شود؛ و یا سورس آن‌ها را دریافت کرده و کامپایل کنید. در ادامه تعدادی از این افزونه‌ها را بررسی خواهیم کرد.

حذف آبشاری اسناد

```
PM> Install-Package RavenDB.Bundles.CascadeDelete -Pre
```

فایل افزونه حذف آبشاری اسناد را از طریق دستور نیوگت فوق می‌توان دریافت کرد. سپس فایل Raven.Bundles.CascadeDelete.dll دریافتی را درون پوشه plugins کنار فایل exe سرور RavenDB کپی کنید تا قابل استفاده شود. استفاده مهم این افزونه، حذف پیوست‌های باینری اسناد و یا حذف اسناد مرتبط با یک سند، پس از حذف سند اصلی است (که به صورت پیش فرض انجام نمی‌شود).
یک مثال:

```
var comment = new Comment
{
   PostId = post.Id
};
session.Store(comment);

session.Advanced.GetMetadataFor(post)["Raven-Cascade-Delete-Documents"] = RavenJToken.FromObject(new[]
{ comment.Id });
session.Advanced.GetMetadataFor(post)["Raven-Cascade-Delete-Attachments"] =
RavenJToken.FromObject(new[] { "picture/1" });

session.SaveChanges();
```

برای استفاده از آن باید از متد session.Advanced.GetMetadataFor استفاده کرد. در اینجا شیء post که دارای تعدادی کامنت است، مشخص می‌شود. سپس با مشخص سازی Raven-Cascade-Delete-Documents و ذکر Id کامنت‌های مرتبطی که باید حذف شوند، تمام این اسناد با هم پس از حذف post، حذف خواهند شد. همچنین دستور Raven-Cascade-Delete-Attachments سبب حذف فایل‌های مشخص شده با Id مرتبط با یک سند، می‌گردد.

نگهداری و بازیابی نگارش‌های مختلف اسناد

```
PM> Install-Package RavenDB.Bundles.Versioning
```

فایل افزونه Versioning اسناد را از طریق دستور نیوگت فوق می‌توان دریافت کرد. سپس فایل dll دریافتی را درون پوشه plugins کنار فایل exe سرور RavenDB کپی کنید تا قابل استفاده شود. فایل Raven.Bundles.Versioning.dll باید در پوشه افزونه‌ها کپی شود و فایل Raven.Client.Versioning.dll به برنامه ما ارجاع داده خواهد شد.
با استفاده از قابلیت document versioning می‌توان تغییرات اسناد را در طول زمان، ردیابی کرد؛ همچنین حذف یک سند، این سابقه را از بین نخواهد برد.
تنظیمات اولیه آن به این صورت است که توسط شیء VersioningConfiguration به سشن جاری اعلام می‌کنیم که چند نگارش از اسناد را ذخیره کند. اگر Exclude آن به true تنظیم شود، اینکار صورت نخواهد گرفت.

```
session.Store(new VersioningConfiguration
{
  Exclude = false,
  Id = "Raven/Versioning/DefaultConfiguration",
  MaxRevisions = 5
});
```

تنظیم Id به Raven/Versioning/DefaultConfiguration، سبب خواهد شد تا VersioningConfiguration فوق به تمام اسناد اعمال شود. اگر نیاز است برای مثال تنها به BlogPosts اعمال شود، این Id را باید به Raven/Versioning/BlogPosts تنظیم کرد.
بازیابی نگارش‌های مختلف یک سند، صرفا از طریق متد Load میسر است و در اینجا شماره Id نگارش به انتهای Id سند اضافه می‌شود. برای مثال "blogposts/1/revisions/1" به نگارش یک مطلب شماره یک اشاره می‌کند.
برای بدست آوردن سه نگارش آخر یک سند باید از متد ذیل استفاده کرد:

```
var lastThreeVersions = session.Advanced.GetRevisionsFor<BlogPost>(post.Id, 0, 3);
```

عنوان: **بررسی حالت‌های مختلف نصب RavenDB**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۱۹ ۲۳:۴۸
آدرس: www.dotnettips.info
برچسب‌ها: NoSQL, RavenDB

چهار روش مختلف برای نصب، استفاده و توزیع RavenDB وجود دارند. ساده‌ترین روش آن‌را که اجرای فایل Raven.Server.exe است، تاکنون بررسی کردیم. این روش صرفا جهت دیباگ و کار برنامه نویسی مناسب است. در ادامه سه روش دیگر را بررسی خواهیم کرد.

الف) استفاده از RavenDB در حالت مدفون شده یا Embedded

حالت Embedded به این معنا است که RavenDB درون پروسه برنامه شما اجرا خواهد شد و نه به صورت پروسه‌ای مجزا. این حالت برای ارائه ساده برنامه‌های دسکتاپ بسیار مناسب است؛ یا حتی توزیع برنامه‌های سبک ASP.NET بدون نیاز به نصب بانک اطلاعاتی خاصی بر روی وب سرور.
برای کار با RavenDB در حالت Embedded ابتدا فایل‌های مورد نیاز آن‌را از طریق نیوگت دریافت کنید:

```
PM> Install-Package RavenDB.Embedded -Pre
```

در این حالت فایل کلاینت مورد نیاز، اسمبلی Raven.Client.Embedded.dll خواهد بود. سپس در کدهای قبلی خود بجای استفاده از new DocumentStore، اینبار خواهیم داشت new EmbeddableDocumentStore.

```
var documentStore = new EmbeddableDocumentStore { DataDirectory = @"~/app_data/ravendb" };
documentStore.Initialize();
```

سایر قسمت‌های برنامه نیازی به تغییر نخواهند داشت.
امکان تعریف DataDirectory در فایل کانفیگ برنامه نیز وجود دارد. فقط در این حالت باید دقت داشت که نام مسیر، با DataDir شروع می‌شود و نه DataDirectory :

```
<connectionStrings>
  <add name="Local" connectionString="DataDir = ~\Data"/>
```

سپس همانند قبل، مقدار خاصیت رشته اتصالی EmbeddableDocumentStore به نام مدخل فوق باید تنظیم گردد.

چند نکته جالب در مورد حالت Embedded

- امکان اجرای درون حافظه‌ای RavenDB نیز وجود دارد:

```
var documentStore = new EmbeddableDocumentStore{RunInMemory = true}.Initialize()
```

در اینجا فقط کافی است خاصیت RunInMemory شیء EmbeddableDocumentStore به true تنظیم شود. این مورد بسیار مناسب است برای انجام آزمون‌های واحد بسیار سریع که پس از پایان کار برنامه، اثری از بانک اطلاعاتی آن باقی نخواهد ماند.
- اجرای حالت Embedded به صورت HTTP:
در حالت Embedded دیگر دسترسی به برنامه سیلورلایت Raven studio وجود ندارد. اگر علاقمند به کار با آن بودید، خاصیت UseEmbeddedHttpServer شیء EmbeddableDocumentStore را به true تنظیم کنید. سپس فایل Raven.Studio.xap را در ریشه وب سایت خود قرار دهید. اکنون مانند قبل آدرس localhost:8080/raven/studio.html برقرار خواهد بود.
همچنین سرور Http این بانک اطلاعاتی را نیز می‌توان دستی راه اندازی کرد. متد NonAdminHttp.EnsureCanListenToWhenInNonAdminContext بررسی می‌کند که آیا برنامه مجوز راه اندازی یک سرور را بر روی پورت مثلا 8080 دارد یا خیر.

```
NonAdminHttp.EnsureCanListenToWhenInNonAdminContext(8080);

// Start the HTTP server manually
var server = new RavenDbHttpServer(documentStore.Configuration, documentStore.DocumentDatabase);
server.Start();
```

ب) نصب RavenDB به صورت سرویس ویندوز NT

اگر مایل باشیم تا RavenDB را نیز مانند SQL Server به صورت یک سرویس ویندوز NT نصب کنیم تا همواره در پس زمینه سرور در حال اجرا باشد، کنسول پاورشل ویندوز را گشوده و سپس فرمان ذیل را صادر کنید:

```
d:\ravendb\server> .\raven.server.exe /install
```

اکنون اگر به کنسول مدیریتی سرویس‌های ویندوز یا services.msc مراجعه کنید، ravendb را به صورت یک آیتم جدید در لیست سرویس‌های ویندوز خواهید یافت.
و اگر خواستید این سرویس را عزل کنید، دستور ذیل را در پاورشل ویندوز صادر کنید:

```
d:\ravendb\server> .\raven.server.exe /uninstall
```

ج) نصب RavenDB به صورت یک پروسه IIS (یا اجرا شده توسط IIS)

فایل‌های مورد نیاز حالت اجرای RavenDB را به صورت یک پروسه مجزای IIS از نیوگت دریافت کنید:

```
PM> Install-Package RavenDB.AspNetHost -Pre
```

در این حالت، پوشه bin، فایل xap و فایل کانفیگ برنامه وب مورد نیاز دریافت خواهند شد. پس از آن، تنها کافی است یک دایرکتوری مجازی را در IIS به این پوشه جدید اختصاص داده و همچنین بهتر است یک Application pool جدید را نیز برای آن تهیه کنید تا واقعا این برنامه در پروسه‌ی مجزای خاص خودش اجرا شود. حتی در این حالت می‌توان شماره پورت دیگری را به این برنامه اختصاص داد. به علاوه در این حالت تنظیمات Recycling مربوط به IIS را هم باید مدنظر داشت (در قسمت تنظیمات Application pool برنامه) و مثلا تنظیم کرد که برنامه پس از چه مدت فعال نبودن از حافظه خارج شود.

یک نکته

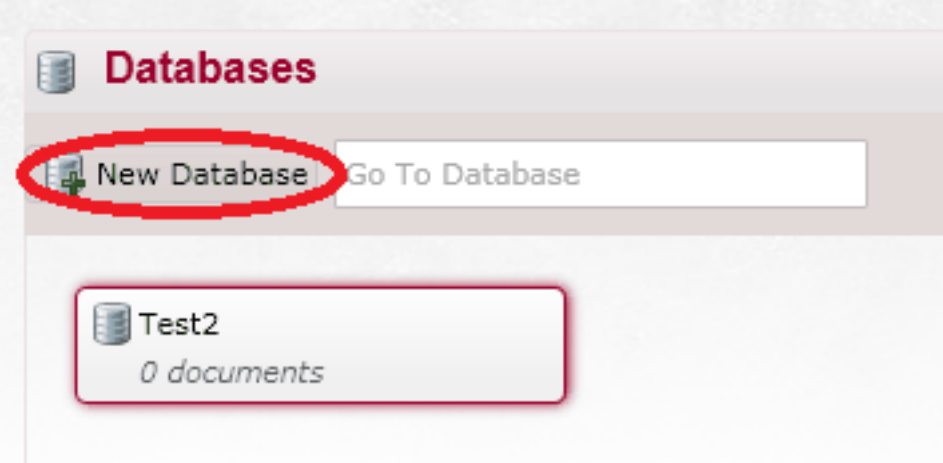
تمام بسته‌های مورد نیاز را یکجا از آدرس http://ravendb.net/download نیز می‌توان دریافت کرد. در نگارش‌های جدید، بسته نصاب نیز برای این بانک اطلاعاتی تهیه شده است که برای نمونه توزیع آن‌را جهت حالت نصب در IIS ساده‌تر می‌کند.

عنوان: **نگهداری سرور RavenDB**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۴:۱۱
آدرس: www.dotnettips.info
برچسب‌ها: NoSQL, RavenDB

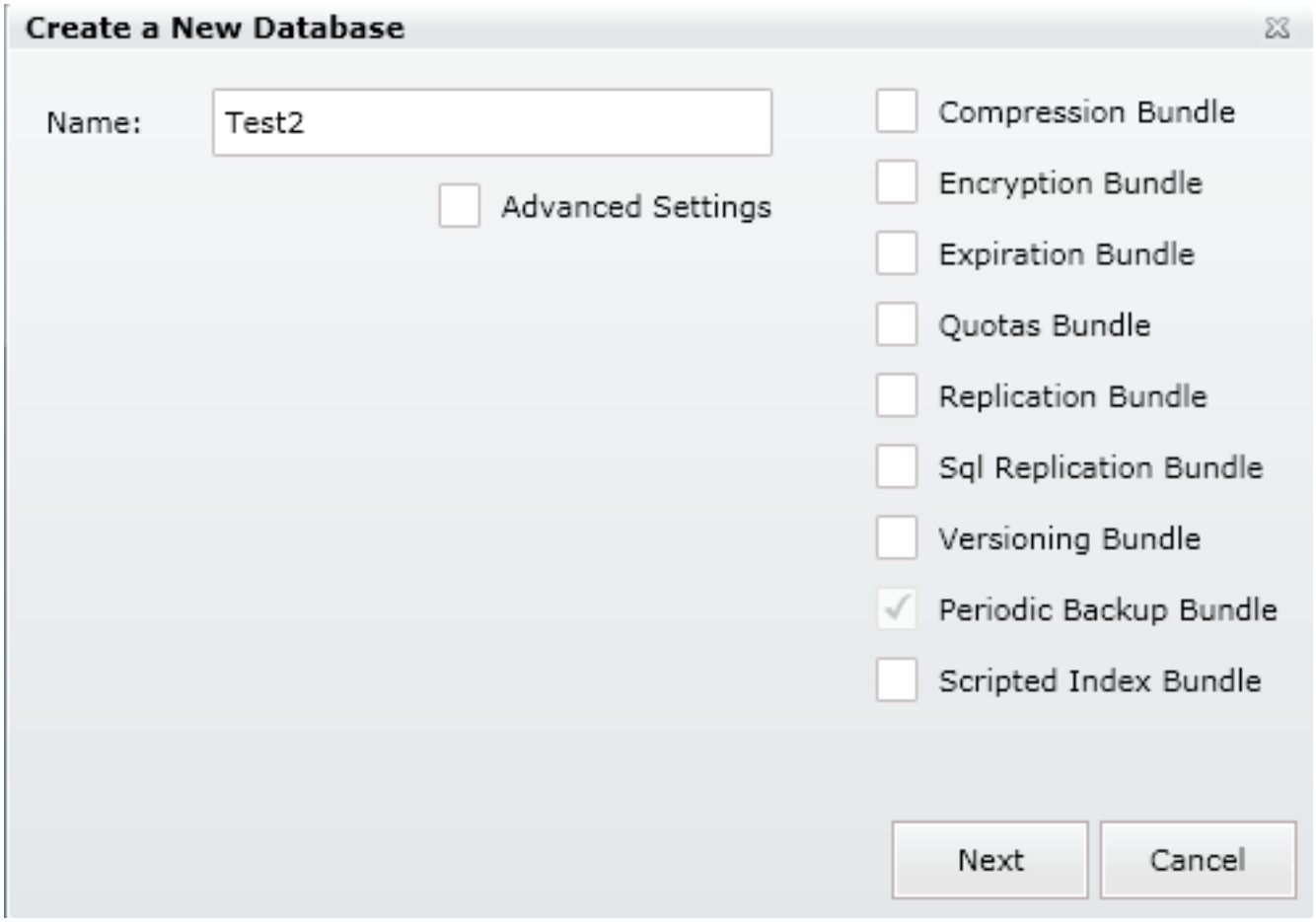
نگهداری سرور RavenDB شامل مواردی است مانند مدیریت فایل‌های آن، اضافه کردن یا حذف بانک‌های اطلاعاتی و تهیه پشتیبان از آن‌ها که در ادامه بررسی خواهند شد.

ایجاد و حذف بانک‌های اطلاعاتی جدید

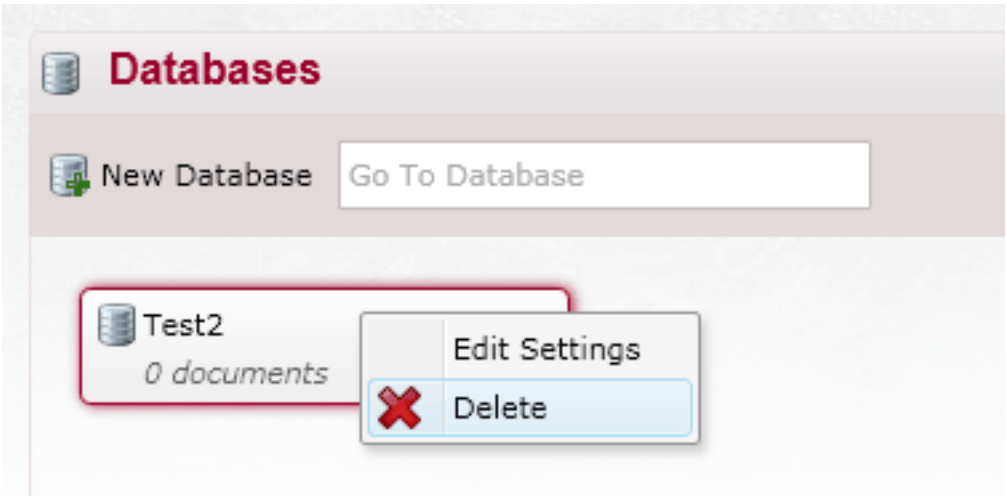
برای این منظور به آدرس http://localhost:8080 مراجعه و از طریق کنسول مدیریتی تحت وب RavenDB بر روی دکمه New database کلیک کنید.



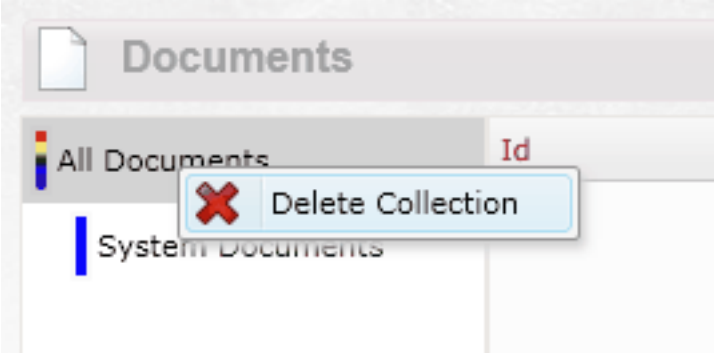
در صفحه باز شده می‌توان نام دیتابیس را مشخص کرد و همچنین در صورت نیاز افزونه‌هایی مانند فشرده سازی یا رمزنگاری اطلاعات را نیز فعال نمود.



پس از ایجاد دیتابیس، برای حذف آن، بر روی نام دیتابیس کلیک راست کرده و گزینه Delete را انتخاب کنید



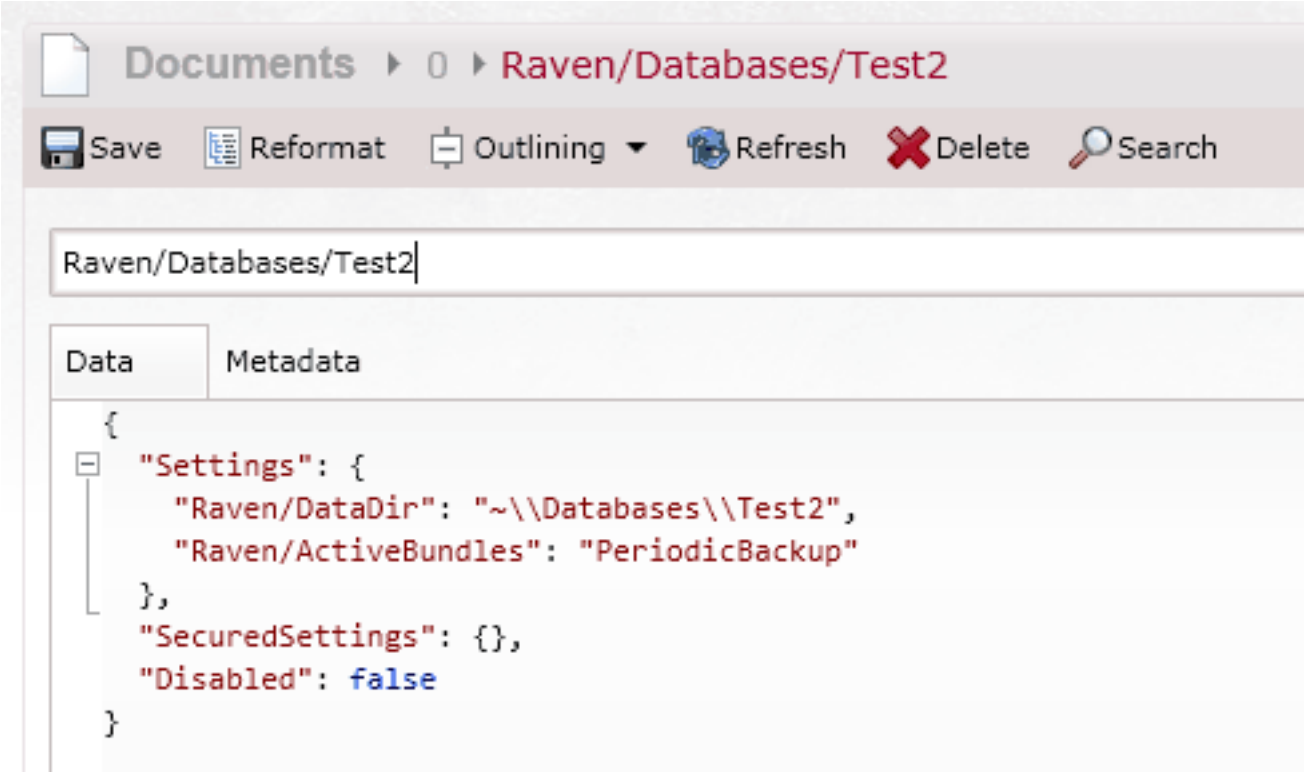
روش دیگر حذف اطلاعات یک بانک اطلاعاتی، مراجعه به سندهای آن و سپس کلیک راست بر روی گروهی از آن‌ها برای حذف می‌باشد:



و یا سرور RavenDB را خاموش یا stop کنید. سپس به پوشه Database کنار فایل Raven.Server.exe مراجعه کرده، بانک اطلاعاتی خود را یافته و سپس کل پوشه آن‌را Delete کنید.

سؤال: چگونه با دیتابیس‌های ایجاد شده کار کنیم؟

تاکنون تمام مثال‌های برنامه با بانک اطلاعاتی پیش فرض RavenDB کار کردند (چیزی شبیه به master database در اس کیوال سرور) و هیچگاه ابتدا یک دیتابیس جدید و مستقل را برای انجام آزمایشات خود، ایجاد نکردیم. بدیهی است این روش برای محیط‌های کاری توصیه نمی‌شود.



برای نمونه در اینجا به System database این مجموعه وارد شده‌ایم که تعریف جزئیات بانک اطلاعاتی جدید ایجاد شده را در خود دارد.

جهت استفاده از بانک اطلاعاتی جدید ایجاد شده در کدهای خود، فقط کافی است خاصیت DefaultDatabase یک

DocumentStore مقدار دهی شود:

```
using (var store = new DocumentStore
{
    Url = "http://localhost:8080",
    DefaultDatabase = "Test2"
}.Initialize())
{
  //...
}
```

تهیه پشتیبان از بانک‌های اطلاعاتی و بازیابی آن‌ها

ابتدا نیاز است تمام بسته‌های مورد نیاز را یکجا از آدرس http://ravendb.net/download تهیه کنید. سپس به پوشه backup آن مراجعه کرده و از فایل اجرایی Raven.Backup.exe آن می‌توان جهت تهیه پشتیبان از بانک اطلاعاتی خاصی استفاده نمود. لازم به ذکر است که این برنامه باید با سطح دسترسی ادمین اجرا شود.

```
Raven.Backup.exe --url==http://localhost:8080 --dest=d:\backup
```

برنامه backup، آدرس سرور را گرفته و سپس فایل‌های پشتیبان تهیه شده را در آدرس مشخصی ذخیره می‌کند. برای مدیریت اجرای روزانه آن نیز از برنامه استاندارد windows task schedule manager استفاده نمائید. به علاوه امکانات Shadow copy ویندوز نیز در اینجا مفید خواهند بود.

برای بازیابی و Restore یک بانک اطلاعاتی ابتدا دستور Raven.Server.exe /help را صادر کنید تا کلیه سوئیچ‌های این برنامه مشخص شوند. یکی از آن‌ها Restore نام دارد که پارامترهای dest و src را دریافت می‌کند (کجا بازیابی شود و از کجا اطلاعات را بخواند).

همچنین بجای backup و restore، امکان export و import نیز وجود دارند و برای انجام آن از برنامه Raven.Smuggler.exe که کنار Raven.Server.exe قرار دارد، می‌توان استفاده کرد.
برای تهیه خروجی (که در حقیقت تهیه یک dump فشرده شده از اسناد JSON موجود است):

```
Raven.Smuggler.exe out http://localhost:8080/ dump.raven
```

و برای بازیابی خروجی تولید شده:

```
Raven.Smuggler.exe in http://localhost:8080/ dump.raven
```

عنوان: **الگوریتم HiLo در RavenDB چیست؟**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۲۱ ۱۱:۱۲
آدرس: www.dotnettips.info
برچسب‌ها: NoSQL, RavenDB

یکی از مشکلاتی که با اکثر سیستم‌های رابطه‌ای وجود دارد، نیاز به تکمیل یک تراکنش برای دریافت Id رکورد ثبت شده است. همین مساله علاوه بر خاتمه یک تراکنش و شروع تراکنشی دیگر، به معنای رفت و برگشت اضافه‌ای به بانک اطلاعاتی نیز می‌باشد. به همین جهت در RavenDB برای ارائه راه حلی برای اینگونه مشکلات از الگوریتم HiLo برای تولید کلیدهای اسناد استفاده می‌گردد.

HiLo چیست؟

HiLo روشی است برای ارائه idهای عددی افزایش یابنده، جهت استفاده در محیط‌های چندکاربری. در این حالت، هنوز هم سرور تولید idها را کنترل می‌کند، اما هربار بازه‌ای از Idها را در اختیار کلاینت‌ها قرار می‌دهد. به این ترتیب کلاینت، بدون نیاز به رفت و برگشت اضافه‌ای به سرور، می‌تواند Id مورد نیاز خود را از بازه ارائه شده تامین کرده و هر زمانیکه این بازه به پایان رسید، سری دیگری را درخواست کند.
بدیهی است در این حالت، ارائه کلیه Idهای یک بازه از طرف سرور ممکن است کاری سنگین باشد. به همین جهت سرور در درخواست ابتدایی کلاینت، یک تک id را به نام «Hi» جهت مشخص سازی ابتدای بازه تولید idها، در اختیار او قرار می‌دهد. قسمت «Lo» توسط خود کلاینت مدیریت می‌شود و در این بین به هر کلاینت یک capacity یا تعداد مجاز idهایی را که می‌تواند تولید کند، از پیش اختصاص داده شده است:

```
(currentHi - 1)*capacity + (++currentLo)
```

بنابراین فرمول تولید idهای جدید در سمت کلاینت به نحو فوق خواهد بود. currentLo تا جایی افزایش می‌یابد که capacity آن کلاینت تنظیم شده است. سپس Hi جدیدی درخواست شده و Lo صفر می‌شود.

حالت پیش فرض کار کلاینت‌های RavenDB نیز به همین نحو است و الگوریتم HiLo در DocumentConvention آن، از قبل تنظیم شده است و مزیت مهم روش HiLo این است که با هربار فراخوانی متد Store یک سشن، بدون رفت و برگشت اضافه‌تری به سرور، Idهای لازم در اختیار شما قرار خواهد گرفت و نیازی نیست تا زمان SaveChanges صبر کرد. به این ترتیب batch operations در RavenDB کارآیی بسیار بالایی خواهند یافت.

عنوان: **استفاده از RavenDB در ASP.NET MVC به همراه تزریق وابستگی‌ها**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۲۱ ۱۵:۵۳
آدرس: www.dotnettips.info
گروه‌ها: NoSQL, RavenDB

جهت تکمیل مباحث این دوره می‌توان به نحوه مدیریت سشن‌ها و document store بانک اطلاعاتی RavenDB با استفاده از یک IoC Container مانند StructureMap در ASP.NET MVC پرداخت. اصول کلی آن به تمام فناوری‌های دات نتی دیگر مانند وب فرم‌ها، WPF و غیره نیز قابل بسط است. تنها پیشنیاز آن مطالعه «کامل» دوره « بررسی مفاهیم معکوس سازی وابستگی‌ها و ابزارهای مرتبط با آن » می‌باشد.

هدف از بحث

ارائه راه حلی جهت تزریق یک وهله از واحد کار تشکیل شده (همان شیء سشن در RavenDB) به کلیه کلاس‌های لایه سرویس برنامه و همچنین زنده نگه داشتن شیء document store آن در طول عمر برنامه است. ایجاد شیء document store که کار اتصال به بانک اطلاعاتی را مدیریت می‌کند، بسیار پرهزینه است. به همین جهت این شیء تنها یکبار باید در طول عمر برنامه ایجاد شود.

ابزارها و پیشنیازهای لازم

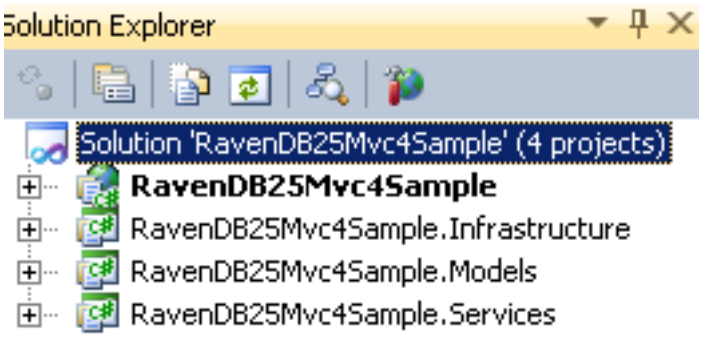
ابتدا یک برنامه جدید ASP.NET MVC را آغاز کنید. سپس ارجاعات لازم را به کلاینت RavenDB، سرور درون پروسه‌ای آن (RavenDB.Embedded) و همچنین StructureMap با استفاده از نیوگت، اضافه نمائید:

```
PM> Install-Package RavenDB.Client
PM> Install-Package RavenDB.Embedded -Pre
PM> Install-Package structuremap
```

دریافت کدهای کامل این مثال

RavenDB25Mvc4Sample.zip

این مثال، به همراه فایل‌های باینری ارجاعات یاد شده، نیست (جهت کاهش حجم 100 مگابایتی آن). برای بازیابی آن‌ها می‌توانید به مطلبی در اینباره در سایت مراجعه کنید.
این پروژه از چهار قسمت مطابق شکل زیر تشکیل شده است:



الف) لایه سرویس‌های برنامه

```
using RavenDB25Mvc4Sample.Models;
using System.Collections.Generic;

namespace RavenDB25Mvc4Sample.Services.Contracts
{
    public interface IUsersService
    {
```

```
        User AddUser(User user);
        IList<User> GetUsers(int page, int count = 20);
    }
}

using System.Collections.Generic;
using System.Linq;
using Raven.Client;
using RavenDB25Mvc4Sample.Models;
using RavenDB25Mvc4Sample.Services.Contracts;

namespace RavenDB25Mvc4Sample.Services
{
    public class UsersService : IUsersService
    {
        private readonly IDocumentStore _documentStore;
        private readonly IDocumentSession _documentSession;
        public UsersService(IDocumentStore documentStore, IDocumentSession documentSession)
        {
            _documentStore = documentStore;
            _documentSession = documentSession;
        }

        public User AddUser(User user)
        {
            _documentSession.Store(user);
            return user;
        }

        public IList<User> GetUsers(int page, int count = 20)
        {
            return _documentSession.Query<User>()
                                   .Skip(page * count)
                                   .Take(count)
                                   .ToList();
        }

        //todo: سایر متدهای مورد نیاز در اینجا

    }
}
```

نکته مهمی که در اینجا وجود دارد، استفاده از اینترفیس‌های خود RavenDB است. به عبارتی IDocumentSession، تشکیل دهنده الگوی واحد کار در RavenDB است و نیازی به تعاریف اضافه‌تری در اینجا وجود ندارد.
هر کلاس لایه سرویس با یک اینترفیس مشخص شده و اعمال آن‌ها از طریق این اینترفیس‌ها در اختیار کنترلرهای برنامه قرار می‌گیرند.

ب) لایه Infrastructure برنامه

در این لایه کدهای اتصالات IoC Container مورد استفاده قرار می‌گیرند. کدهایی که به برنامه جاری وابسته‌اند، اما حالت عمومی و مشترکی ندارند تا در سایر پروژه‌های مشابه استفاده شوند.

```
using Raven.Client;
using Raven.Client.Embedded;
using RavenDB25Mvc4Sample.Services;
using RavenDB25Mvc4Sample.Services.Contracts;
using StructureMap;

namespace RavenDB25Mvc4Sample.Infrastructure
{
    public static class IoCConfig
    {
        public static void ApplicationStart()
        {
            ObjectFactory.Initialize(x =>
            {
                // داکیومنت استور سینگلتون تعریف شده چون باید در طول عمر برنامه زنده نگه داشته شود
                x.ForSingletonOf<IDocumentStore>().Use(() =>
                       {
                           return new EmbeddableDocumentStore
                           {
                               DataDirectory = "App_Data"
                           }.Initialize();
                       });
```

```
                // سشن در برنامه وب هیبرید تعریف شده تا در طول عمر یک درخواست زنده نگه داشته شود
                // در برنامه‌های ویندوزی حالت هیبرید را حذف کنید
                x.For<IDocumentSession>().HybridHttpOrThreadLocalScoped().Use(context =>
                    {
                        return context.GetInstance<IDocumentStore>().OpenSession();
                    });

                // اتصالات لایه سرویس در اینجا
                x.For<IUsersService>().Use<UsersService>();
                // ...
            });
        }

        public static void ApplicationEndRequest()
        {
            ObjectFactory.ReleaseAndDisposeAllHttpScopedObjects();
        }
    }
}
```

تعاریف اتصالات StructureMap را در اینجا ملاحظه می‌کنید.
IDocumentStore و IDocumentSession، دو اینترفیس تعریف شده در کلاینت RavenDB هستند. اولی کار اتصال به بانک اطلاعاتی را مدیریت خواهد کرد و دومی کار مدیریت الگوی واحد کار را انجام می‌دهد. IDocumentStore به صورت Singleton تعریف شده است؛ چون باید در طول عمر برنامه زنده نگه داشته شود. اما IDocumentStore در ابتدای هر درخواست رسیده، وهله سازی شده و سپس در پایان هر درخواست در متد ApplicationEndRequest به صورت خودکار Dispose خواهد شد.
اگر به فایل Global.asax.cs پروژه وب برنامه مراجعه کنید، نحوه استفاده از این کلاس را مشاهده خواهید کرد:

```
using System;
using System.Globalization;
using System.Web.Mvc;
using System.Web.Routing;
using RavenDB25Mvc4Sample.Infrastructure;
using StructureMap;

namespace RavenDB25Mvc4Sample
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            IoCConfig.ApplicationStart();

            AreaRegistration.RegisterAllAreas();

            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            //Set current Controller factory as StructureMapControllerFactory
            ControllerBuilder.Current.SetControllerFactory(new StructureMapControllerFactory());
        }

        protected void Application_EndRequest(object sender, EventArgs e)
        {
            IoCConfig.ApplicationEndRequest();
        }
    }

    public class StructureMapControllerFactory : DefaultControllerFactory
    {
        protected override IController GetControllerInstance(RequestContext requestContext, Type
controllerType)
        {
            if (controllerType == null)
                throw new InvalidOperationException(string.Format("Page not found: {0}",

requestContext.HttpContext.Request.Url.AbsoluteUri.ToString(CultureInfo.InvariantCulture)));
            return ObjectFactory.GetInstance(controllerType) as Controller;
        }
    }
}
```

در ابتدای کار برنامه، متد IoCConfig.ApplicationStart جهت برقراری اتصالات، فراخوانی می‌شود. در پایان هر درخواست نیز شیء سشن جاری تخریب خواهد شد. همچنین کلاس StructureMapControllerFactory نیز جهت وهله سازی خودکار کنترلرهای برنامه به همراه تزریق وابستگی‌های مورد نیاز، تعریف گشته است.

ج) استفاده از کلاس‌های لایه سرویس در کنترلرهای برنامه

```
using System.Web.Mvc;
using Raven.Client;
using RavenDB25Mvc4Sample.Models;
using RavenDB25Mvc4Sample.Services.Contracts;

namespace RavenDB25Mvc4Sample.Controllers
{
    public class HomeController : Controller
    {
        private readonly IDocumentSession _documentSession;
        private readonly IUsersService _usersService;
        public HomeController(IDocumentSession documentSession, IUsersService usersService)
        {
            _documentSession = documentSession;
            _usersService = usersService;
        }

        [HttpGet]
        public ActionResult Index()
        {
            return View(); //نمایش صفحه ثبت
        }

        [HttpPost]
        public ActionResult Index(User user)
        {
            if (this.ModelState.IsValid)
            {
                _usersService.AddUser(user);
                _documentSession.SaveChanges();

                return RedirectToAction("Index");
            }
            return View(user);
        }
    }
}
```

پس از این مقدمات و طراحی اولیه، استفاده از کلاس‌های لایه سرویس در کنترل‌ها، ساده خواهند بود. تنها کافی است اینترفیس‌های مورد نیاز را از طریق روش تزریق در سازنده کلاس‌ها تعریف کنیم. سایر مسایل وهله سازی آن خودکار خواهند بود.

عنوان: **تنظیمات امنیتی دسترسی به سرور RavenDB**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۲۲ ۱۲:۲۵
آدرس: www.dotnettips.info
برچسب‌ها: NoSQL, RavenDB

تا اینجا اگر مباحث را دنبال کرده باشید، برای اتصال به RavenDB از اعتبارسنجی خاصی استفاده نشد و در حالت پیش فرض، بدون تنظیم خاصی، موفق به اتصال به سرور آن شدیم. بدیهی این مورد در دنیای واقعی به دلایل امنیتی قابل استفاده نیست و نیاز است دسترسی به سرور RavenDB را محدود کرد. برای مثال SQL Server حداقل از دو روش Windows authentication و روش توکار خاص خودش برای اعتبارسنجی دسترسی به داده‌ها استفاده می‌کند. اما RavenDB چطور؟

حالت پیش فرض دسترسی به سرور RavenDB

اگر فایل Raven.Server.exe.config را در یک ویرایشگر متنی باز کنید، یک چنین تنظیماتی در آن قابل مشاهده هستند:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="Raven/Port" value="*"/>
    <add key="Raven/DataDir" value="~\Database\System"/>
    <add key="Raven/AnonymousAccess" value="Admin"/>
  </appSettings>
<runtime>
<loadFromRemoteSources enabled="true"/>
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
<probing privatePath="Analyzers;Plugins"/>
</assemblyBinding>
</runtime>
</configuration>
```

کلید Raven/AnonymousAccess چندین مقدار مختلف را می‌تواند داشته باشد، مانند All ، Get و None.
حالت پیش فرض دسترسی به RavenDB برای کاربران اعتبارسنجی نشده، حالت Get است (خواندن اطلاعات) و هیچگونه دسترسی تغییر اطلاعات آن‌را ندارند (حالت Read only). اگر این کلید به All تنظیم شود، کلیه کاربران، قابلیت Read و Write را خواهند داشت. حالت None به این معنا است که تنها کاربران اعتبارسنجی شده می‌توانند به دیتابیس دسترسی پیدا کنند.
اگر علاقمند هستید که مجوزهای یک کاربر متصل را مشاهده کنید، از فرمان ذیل استفاده نمائید:

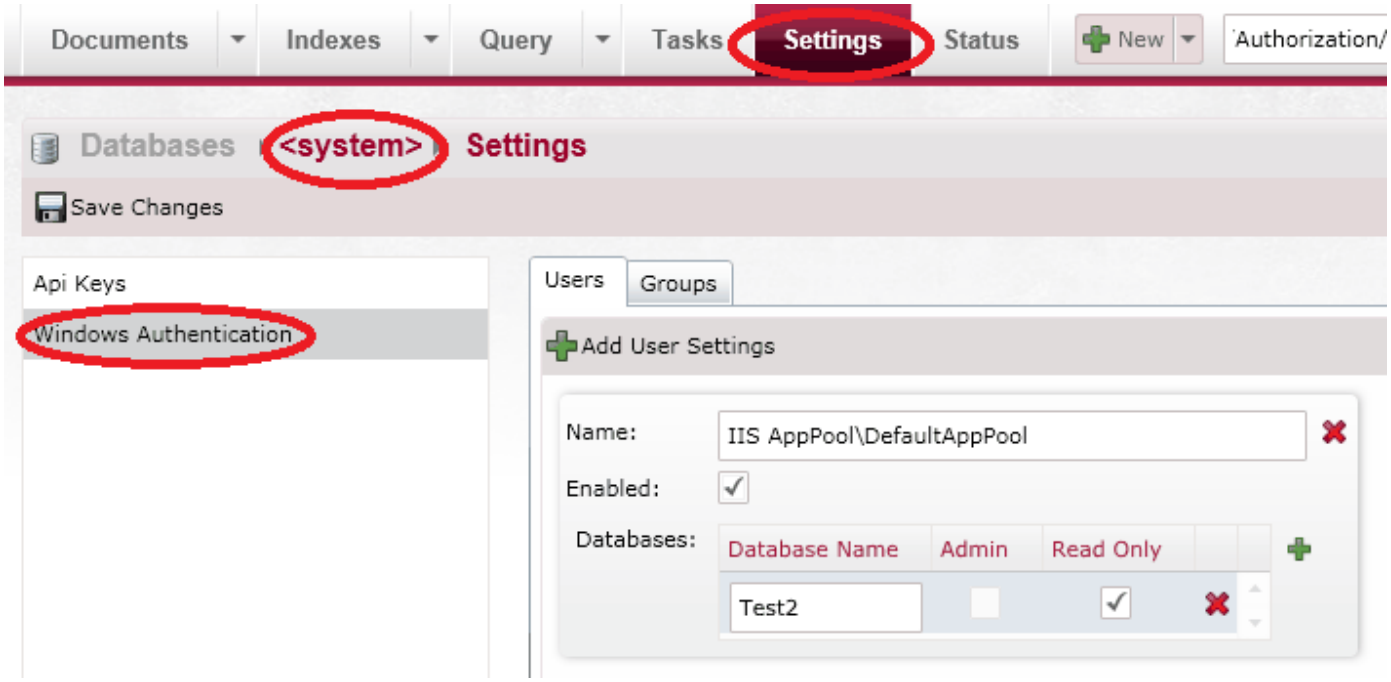
```
var json = ((ServerClient) store.DatabaseCommands).CreateRequest("GET", "/debug/user-
info").ReadResponseJson();
```

نکته بسیار مهم

اگر مجوز RavenDB را نخریده باشید، مقدار Admin تنها مقداری است که در اینجا می‌توانید تنظیم کنید. به این معنا که کلیه کاربران، دسترسی Admin را به سرور خواهند داشت. (و بدیهی است فقط برای آزمایش سیستم مناسب است)
سعی در تنظیم حالت اعتبار سنجی زمانیکه از مجوز AGPL استفاده می‌کنید، با یک استثناء از طرف سرور متوقف خواهد شد.

Windows authentication

اعتبار سنجی پیش فرض مورد استفاده نیز Windows authentication است. به این معنا که تنها کاربری با دارا بودن اکانت معتبری بر روی سیستم و یا دومین ویندوزی، امکان کار با RavenDB را خواهد داشت. در این حالت کلیه کاربران دومین به سرور دسترسی خواهند داشت. اگر این حالت مطلوب شما نیست، می‌توان از گروه‌های ویژه کاربران تعریف شده بر روی سیستم و یا بر روی دومین ویندوزی استفاده کرد.
این تنظیمات باید بر روی دیتابیس System صورت گیرند، در قسمت Settings و حالت Windows authentication :

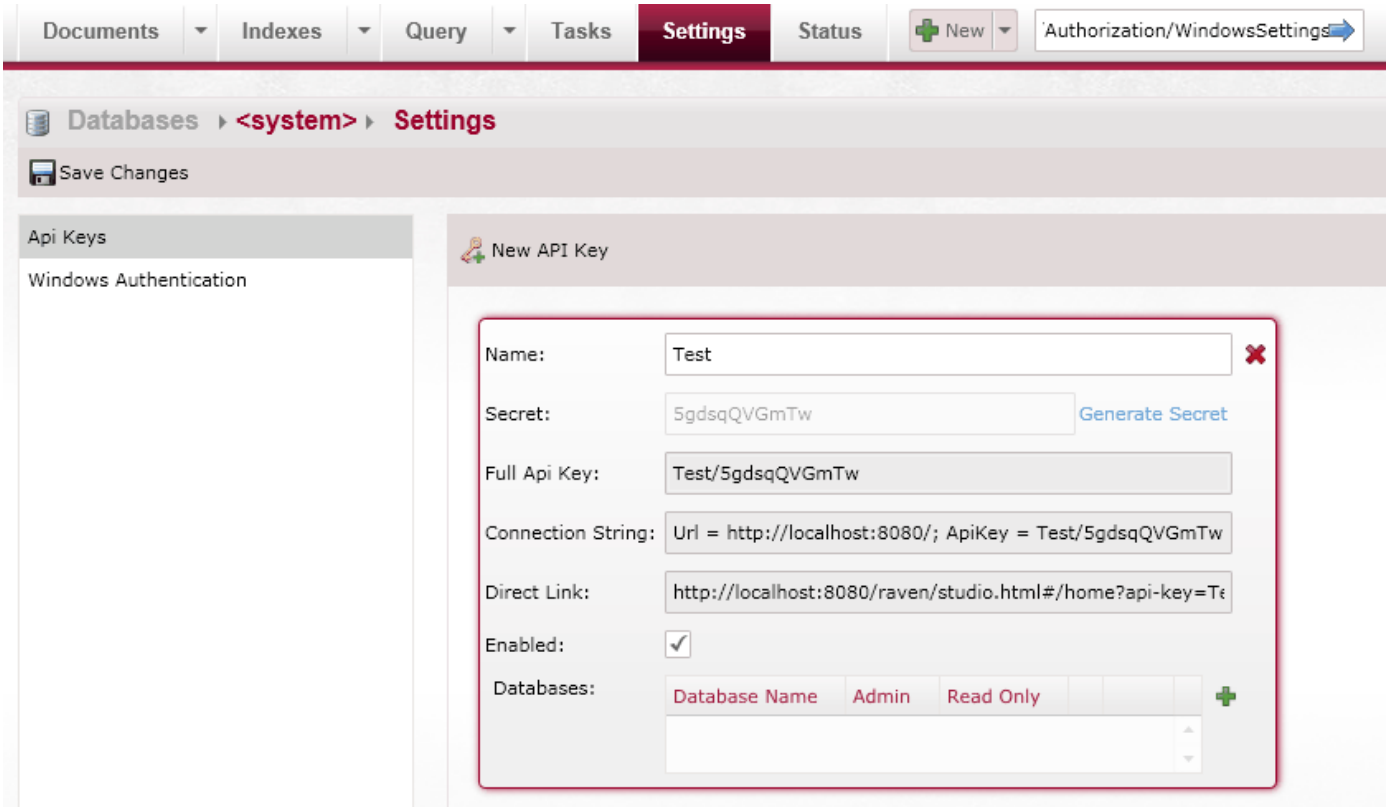


اعتبارسنجی OAuth

شاید دسترسی به سرور RavenDB همیشه از طریق Windows authentication مطلوب نباشد. برای این حالت از روش اعتبارسنجی سفارشی خاصی به نام OAuth نیز پشتیبانی می‌شود. این حالت به صورت توکار در سرور RavenDB پیاده سازی شده است و یا می‌توان با پیاده سازی اینترفیس IAuthenticateClient کنترل بیشتری را اعمال کرد. البته با دریافت افزونه Raven.Bundles.Authentication به یک نمونه پیاده سازی شده آن دسترسی خواهید داشت. پس از دریافت آن، فایل اسمبلی مربوطه را به درون پوشه افزونه‌های سرور کپی کنید تا آماده استفاده شود.

```
PM> Install-Package RavenDB.Bundles.Authentication -Pre
```

کار با آن هم بسیار ساده است. ابتدا کلیدهای لازم را در سمت سرور، در قسمت تنظیمات بانک اطلاعاتی سیستم ایجاد کنید:



فایل کانفیگ سرور را برای افزودن سطر ذیل ویرایش کنید:

```
<add key="Raven/AuthenticationMode" value="OAuth"/>
```

سپس DocumentStore کلاینت به نحو ذیل باید آغاز شود:

```
var documentStore = new DocumentStore
{
  ApiKey = "sample/ThisIsMySecret",
  Url = "http://localhost:8080/"
};
```

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۲۲ ۱۴:۴۹

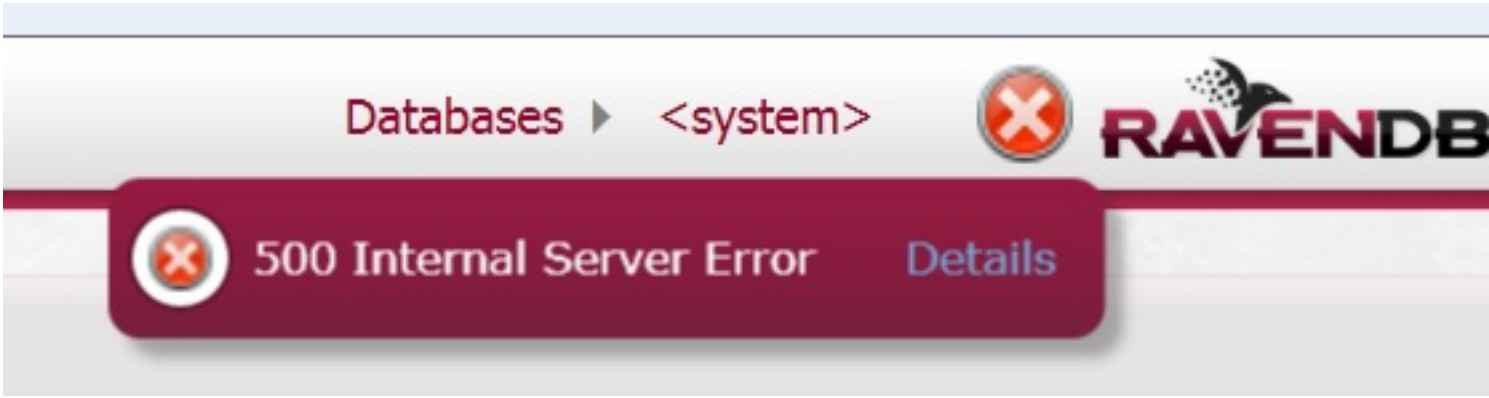
یک نکته تکمیلی

کدهای تعیین اعتبار مجوز RavenDB رو در اینجا می‌توانید ملاحظه کنید.

نویسنده: رضا ساکت
تاریخ: ۱۳۹۲/۰۸/۰۸ ۹:۲۲

سلام

اعتبارسنجی OAuth را مطابق آنچه گفته شد انجام دادم منتها در هنگام ثبت پیام خطا میدهد!



نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۸/۰۸ ۹:۳۳

به «نکته بسیار مهم » ذکر شده در متن دقت کنید. ویژگی‌های Authentication فقط در نسخه تجاری قابل استفاده هستند. نسخه عمومی به دسترسی Admin محدود است.
توضیحات بیشتر در اینجا Security system - OSS vs commercial use