EBook Gratis

APRENDIZAJE ravendb

Free unaffiliated eBook created from
Stack Overflow contributors.

#ravendb

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: ravendb

It is an unofficial and free ravendb ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official ravendb.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con ravendb

Observaciones

Esta sección proporciona una descripción general de qué es ravendb y por qué un desarrollador puede querer usarlo.

También debe mencionar cualquier tema grande dentro de ravendb, y vincular a los temas relacionados. Dado que la Documentación para ravendb es nueva, es posible que deba crear versiones iniciales de los temas relacionados.

Examples

Instalación o configuración

Instrucciones detalladas sobre cómo configurar o instalar ravendb.

Una aplicación de consola RavenDB simple

Para este ejemplo usaremos la instancia de Live Test RavenDB .

Construiremos una aplicación de consola simple aquí que demuestra las operaciones más básicas:

- Creación
- Recuperación por ID
- Preguntando
- Actualizando
- Supresión

Comience creando una nueva solución de Visual Studio y agregue un proyecto de aplicación de consola. Llamémoslo RavenDBDemoConsole. El uso de la biblioteca del cliente debe ser similar si el Reader utiliza VS Code o su editor favorito.

A continuación, necesitamos agregar las referencias requeridas. Haga clic con el botón derecho en el nodo Referencias en el panel Explorador de soluciones y seleccione Administrar paquetes de NuGet. Busque en línea para 'RavenDb.Client'. Estaré usando la última versión estable, que es, a partir de este escrito, 3.5.2.

Vamos a escribir un código, ¿de acuerdo? Comience agregando las siguientes declaraciones de uso:

```
using Raven.Client;
using Raven.Client.Document;
```

Estos nos permiten usar el `IDocumentStore` y el `DocumentStore` de `IDocumentStore` , que es una

interfaz y su implementación lista para usar para conectarnos a una instancia de RavenDB. Este es el objeto de nivel superior que debemos usar para conectarnos a un servidor y la documentación de RavenDB indica que se utiliza como un solo elemento en la aplicación.

Así que seguiremos adelante y crearemos uno, pero para simplificar, no implementaremos el envoltorio Singleton a su alrededor, solo lo desecharemos cuando el programa salga para que la conexión se cierre de manera limpia. Agregue el siguiente código a su método principal:

```
using (IDocumentStore store = new DocumentStore
{
    Url = "http://live-test.ravendb.net",
    DefaultDatabase = "Pets"
})
{
    store.Initialize();
}
```

Como se dijo al principio, usamos la instancia de Live Test RavenDB, usamos su dirección como la propiedad `Url` del `DocumentStore` . También especificamos un nombre de base de datos predeterminado, en este caso, "Mascotas". Si la base de datos aún no existe, RavenDB la crea cuando intenta acceder a ella. Si existe, entonces el cliente puede usar el existente. Necesitamos llamar al método `Initialize()` para que podamos comenzar a operar en él.

En esta sencilla aplicación, mantendremos dueños y mascotas. Pensamos en su conexión, ya que un propietario puede tener un número arbitrario de mascotas, pero una mascota puede tener un solo propietario. Aunque en el mundo real, una mascota puede tener un número arbitrario de propietarios, por ejemplo, un esposo y una esposa, optaremos por este supuesto, ya que las relaciones de muchos a muchos en una base de datos de documentos se manejan de manera diferente a la de un animal. Base de datos relacional y merece un tema propio. He elegido este dominio porque es lo suficientemente común como para comprenderlo.

Así que ahora debemos definir nuestros tipos de objetos de dominio:

```
public class Owner
{
    public Owner()
    {
        Pets = new List<Pet>();
    }

    public string Id { get; set; }
    public string Name { get; set; }
    public List<Pet> Pets { get; set; }

    public override string ToString()
    {
        return
            "Owner's Id: " + Id + "\n" +
            "Owner's name: " + Name + "\n" +
            "Pets:\n\t" +
            string.Join("\n\t", Pets.Select(p => p.ToString()));
    }
}
```

```
public class Pet
{
    public string Color { get; set; }
    public string Name { get; set; }
    public string Race { get; set; }

    public override string ToString()
    {
        return string.Format("{0}, a {1} {2}", Name, Color, Race);
    }
}
```

Hay algunas cosas a tener en cuenta aquí:

En primer lugar, nuestro `Owner` s puede contener cero o más `Pet` . Tenga en cuenta que la clase `Owner` tiene una propiedad llamada `Id` mientras que la clase `Pet` no. Esto se debe a que los objetos `Pet` se almacenarán *dentro de* los objetos `Owner` , que es bastante diferente de cómo se implementaría este tipo de relación en una base de datos relacional.

Se puede argumentar que esto no debería implementarse de esta manera, y *puede* ser correcto, realmente depende de los requisitos. Como regla general, si una `Pet` tiene sentido que exista sin un `Owner` , no debe estar incrustada sino que existe por sí misma con un identificador propio. En nuestra aplicación, asumimos que una `Pet` solo se considera una mascota si tiene un dueño, de lo contrario sería una criatura o una bestia. Por esta razón, no agregamos una propiedad de `Id` a la clase de `Pet` .

En segundo lugar, tenga en cuenta que el identificador de la clase propietaria es una cadena, ya que generalmente se muestra en los ejemplos en la documentación de RavenDB. Muchos desarrolladores acostumbrados a las bases de datos relacionales pueden considerar esto como una mala práctica, lo que generalmente tiene sentido en el mundo relacional. Pero como RavenDB usa Lucene.Net para realizar sus tareas y porque Lucene.Net se especializa en operar con cadenas, aquí es perfectamente aceptable. También, estamos tratando con una base de datos de documentos que almacena JSON y, después de todo, básicamente todo se representa como una cadena. en JSON.

Una cosa más a tener en cuenta sobre la propiedad `Id` es que no es obligatorio. De hecho, RavenDB adjunta sus propios metadatos a cualquier documento que guardamos, por lo que incluso si no lo definiéramos, RavenDB no tendría problemas con nuestros objetos. Sin embargo, generalmente se define para facilitar el acceso.

Antes de ver cómo podemos usar RavenDB desde nuestro código, definamos algunos métodos de ayuda comunes. Estos deben ser autoexplicativos.

```
// Returns the entered string if it is not empty, otherwise, keeps asking for it.
private static string ReadNotEmptyString(string message)
{
    Console.WriteLine(message);
    string res;
    do
    {
        res = Console.ReadLine().Trim();
        if (res == string.Empty)
```

```
        {
            Console.WriteLine("Entered value cannot be empty.");
        }
    } while (res == string.Empty);

    return res;
}

// Will use this to prevent text from being cleared before we've read it.
private static void PressAnyKeyToContinue()
{
    Console.WriteLine();
    Console.WriteLine("Press any key to continue.");
    Console.ReadKey();
}

// Prepends the 'owners/' prefix to the id if it is not present (more on it later)
private static string NormalizeOwnerId(string id)
{
    if (!id.ToLower().StartsWith("owners/"))
    {
        id = "owners/" + id;
    }

    return id;
}

// Displays the menu
private static void DisplayMenu()
{
    Console.WriteLine("Select a command");
    Console.WriteLine("C - Create an owner with pets");
    Console.WriteLine("G - Get an owner with its pets by Owner Id");
    Console.WriteLine("N - Query owners whose name starts with...");
    Console.WriteLine("P - Query owners who have a pet whose name starts with...");
    Console.WriteLine("R - Rename an owner by Id");
    Console.WriteLine("D - Delete an owner by Id");
    Console.WriteLine();
}
```

Y nuestro principal método:

```
private static void Main(string[] args)
{
    using (IDocumentStore store = new DocumentStore
    {
        Url = "http://live-test.ravendb.net",
        DefaultDatabase = "Pets"
    })
    {
        store.Initialize();

        string command;
        do
        {
            Console.Clear();
            DisplayMenu();

            command = Console.ReadLine().ToUpper();
            switch (command)
```

```
            {
                case "C":
                    Creation(store);
                    break;
                case "G":
                    GetOwnerById(store);
                    break;
                case "N":
                    QueryOwnersByName(store);
                    break;
                case "P":
                    QueryOwnersByPetsName(store);
                    break;
                case "R":
                    RenameOwnerById(store);
                    break;
                case "D":
                    DeleteOwnerById(store);
                    break;
                case "Q":
                    break;
                default:
                    Console.WriteLine("Unknown command.");
                    break;
            }
        } while (command != "Q");
    }
}
```

Creación

Veamos cómo podemos guardar algunos objetos en RavenDB. Vamos a definir los siguientes métodos comunes:

```
private static Owner CreateOwner()
{
    string name = ReadNotEmptyString("Enter the owner's name.");

    return new Owner { Name = name };
}

private static Pet CreatePet()
{
    string name = ReadNotEmptyString("Enter the name of the pet.");
    string race = ReadNotEmptyString("Enter the race of the pet.");
    string color = ReadNotEmptyString("Enter the color of the pet.");

    return new Pet
    {
        Color = color,
        Race = race,
        Name = name
    };
}

private static void Creation(IDocumentStore store)
{
    Owner owner = CreateOwner();
```

```
    Console.WriteLine(
        "Do you want to create a pet and assign it to {0}? (Y/y: yes, anything else: no)",
        owner.Name);

    bool createPets = Console.ReadLine().ToLower() == "y";
    do
    {
        owner.Pets.Add(CreatePet());

        Console.WriteLine("Do you want to create a pet and assign it to {0}?", owner.Name);
        createPets = Console.ReadLine().ToLower() == "y";
    } while (createPets);

    using (IDocumentSession session = store.OpenSession())
    {
        session.Store(owner);
        session.SaveChanges();
    }
}
```

Ahora veamos cómo funciona. Hemos definido una lógica C # simple para crear objetos `Owner` y seguir creando y asignando objetos `Pet` hasta que el usuario así lo desee. La parte en la que se refiere a RavenDB y, por lo tanto, es el tema central de este artículo es cómo guardamos los objetos.

Para guardar el nuevo `Owner` creado junto con sus `Pet` , primero debemos abrir una sesión, que implemente `IDocumentSession` . Podemos crear uno llamando a `OpenSession` en el objeto de almacén de documentos.

Por lo tanto, tenga en cuenta la diferencia, mientras que el almacén de documentos es un objeto permanente que generalmente existe durante toda la vida útil de la aplicación, `IDocumentSession` es un objeto ligero y de corta duración. Representa una serie de operaciones que queremos realizar de una sola vez (o al menos, en unas pocas llamadas a la base de datos).

RavenDB enfatiza (y de alguna manera obliga) que evite un número excesivo de viajes de ida y vuelta al servidor, algo que ellos llaman 'Protección de chatter Cliente-Servidor' en el sitio web de RavenDB. Por esta misma razón, una sesión tiene un límite predeterminado de cuántas llamadas a la base de datos tolerará, por lo que uno debe prestar atención cuando se abre y se elimina una sesión. Debido a que en este ejemplo, tratamos la creación de un `Owner` y sus `Pet` como una operación que debe ejecutarse por sí sola, lo hacemos en una sesión y luego la eliminamos.

Podemos ver dos llamadas a métodos más que nos interesan:

- `session.Store(owner)` , que registra el objeto para guardar, y además, establece la propiedad de `Id` del objeto si aún no está establecido. El hecho de que la propiedad del identificador se llame `Id` es, por lo tanto, una convención.
- `session.Savehanges()` envía las operaciones reales para ejecutarse al servidor RavenDB, comprometiendo todas las operaciones pendientes.

Recuperación por ID

Otra operación común es obtener un objeto por su identificador. En el mundo relacional,

normalmente lo hacemos utilizando una expresión `Where` , especificando el identificador. Pero debido a que en RavenDB, cada *consulta* se realiza mediante índices, que pueden estar *obsoletos* , no es el enfoque a seguir, de hecho, RavenDB lanza una excepción si intentamos realizar una consulta por id. En su lugar, deberíamos usar el método `Load<T>` , especificando el id. Con nuestra lógica de menú ya implementada, solo tenemos que definir el método que realmente carga los datos solicitados y muestra sus detalles:

```
private static void GetOwnerById(IDocumentStore store)
{
    Owner owner;
    string id = NormalizeOwnerId(ReadNotEmptyString("Enter the Id of the owner to display."));

    using (IDocumentSession session = store.OpenSession())
    {
        owner = session.Load<Owner>(id);
    }

    if (owner == null)
    {
        Console.WriteLine("Owner not found.");
    }
    else
    {
        Console.WriteLine(owner);
    }

    PressAnyKeyToContinue();
}
```

Todo lo que está relacionado con RavenDB aquí es, una vez más, la inicialización de una sesión, y luego se usa el método de `Load` . La biblioteca cliente RavenDB devolverá el objeto deserializado como el tipo que especificamos como parámetro de tipo. Es importante saber que RavenDB no impone ningún tipo de compatibilidad aquí: todas las propiedades asignables se asignan y las que no son intercambiables no.

RavenDB necesita que el prefijo de tipo de documento esté precedido por el `Id` . Ese es el motivo de la llamada de `NormalizeOwnerId` . Si un documento con el ID especificado no existe, se devuelve un `null` .

Preguntando

Veremos dos tipos de consultas aquí: una en la que consultamos sobre las propiedades propias de los documentos del `Owner` y otra en la que consultamos sobre los objetos `Pet` incrustados.

Comencemos con el más simple, en el que consultamos los documentos del `Owner` cuya propiedad `Name` comienza con la cadena especificada.

```
private static void QueryOwnersByName(IDocumentStore store)
{
    string namePart = ReadNotEmptyString("Enter a name to filter by.");

    List<Owner> result;
    using (IDocumentSession session = store.OpenSession())
```

```
    {
        result = session.Query<Owner>()
           .Where(ow => ow.Name.StartsWith(namePart))
           .Take(10)
           .ToList();
    }

    if (result.Count > 0)
    {
        result.ForEach(ow => Console.WriteLine(ow));
    }
    else
    {
        Console.WriteLine("No matches.");
    }
    PressAnyKeyToContinue();
}
```

Una vez más, porque nos gustaría realizar la consulta como un trabajo independiente, abrimos una sesión. Podemos consultar sobre una colección de documentos llamando a `Query<TDocumentType>` en el objeto de sesión. Devuelve un `IRavenQueryable<TDocumentType>` , en el que podemos invocar los métodos LINQ habituales, así como algunas extensiones específicas de RavenDB. Hacemos un filtrado simple aquí, y la condición es que el valor de la propiedad `Name` comience con la cadena ingresada. Tomamos los primeros 10 elementos del conjunto de resultados y creamos una lista de ellos. Uno debe prestar atención a la especificación correcta del tamaño del conjunto de resultados: aquí hay otra aplicación defensiva en juego realizada por RavenDB llamada protección de conjuntos de resultados sin límites. Esto significa que (por defecto) solo se devuelven los primeros 128 artículos.

Nuestra segunda consulta se parece a la siguiente:

```
private static void QueryOwnersByPetsName(IDocumentStore store)
{
    string namePart = ReadNotEmptyString("Enter a name to filter by.");

    List<Owner> result;
    using (IDocumentSession session = store.OpenSession())
    {
       result = session.Query<Owner>()
           .Where(ow => ow.Pets.Any(p => p.Name.StartsWith(namePart)))
           .Take(10)
           .ToList();
    }

    if (result.Count > 0)
    {
        result.ForEach(ow => Console.WriteLine(ow));
    }
    else
    {
        Console.WriteLine("No matches.");
    }
    PressAnyKeyToContinue();
}
```

Este no es mucho más complicado, lo escribí para demostrar cuán naturalmente es posible

consultar sobre las propiedades de los objetos incrustados. Esta consulta simplemente devuelve los 10 primeros `Owner` que tienen al menos una `Pet` cuyo nombre comienza con el valor ingresado.

Supresión

Tenemos dos opciones para realizar la eliminación. Una es pasar el identificador del documento, que es útil si no tenemos el objeto en la memoria, pero sí tenemos el identificador y nos gustaría evitar un viaje de ida y vuelta a la base de datos que de otra manera sería evitable. La otra forma, obviamente entonces, es pasar un objeto real guardado en RavenDB. Veremos la primera opción aquí, la otra es solo usar otra sobrecarga y pasar un objeto apropiado:

```
private static void DeleteOwnerById(IDocumentStore store)
{
    string id = NormalizeOwnerId(ReadNotEmptyString("Enter the Id of the owner to delete."));

    using (IDocumentSession session = store.OpenSession())
    {
        session.Delete(id);
        session.SaveChanges();
    }
}
```

Una vez más, necesitamos abrir una sesión para realizar nuestro trabajo. Como se mencionó anteriormente, aquí eliminamos el objeto deseado pasando su identificador al método `Delete` . El prefijo del identificador también debe estar en su lugar aquí, como en el caso del método `Load` . Para enviar realmente el comando de eliminación a la base de datos, debemos llamar al método `SaveChanges` , que hará exactamente eso, junto con cualquier otra operación pendiente que se registre en la misma sesión.

Actualizando

Y finalmente, echaremos un vistazo a cómo actualizar los documentos. Básicamente, tenemos dos formas de hacer esto. El primero es sencillo, cargamos un documento, actualizamos sus propiedades según sea necesario y luego lo pasamos al método de `Store` . Esto debería ser sencillo de acuerdo con la demostración de cargar y guardar, pero hay algunas cosas dignas de mención.

Primero, la biblioteca cliente de RavenDB usa un rastreador de cambios que hace posible actualizar cualquier documento sin pasarlo realmente a la `Store` siempre que la sesión que cargó el documento todavía esté abierta. En este caso, llamar a `SaveChanges` en la sesión es suficiente para que se realice la actualización.

En segundo lugar, para que esto funcione, el objeto obviamente necesita que se establezca su identificador para que RavenDB pueda descubrir qué actualizar.

Dicho esto, solo echaremos un vistazo a la otra forma de actualizar. Existe un concepto llamado parcheo, que se puede usar para actualizar documentos. Al igual que en el caso de la eliminación, también tiene sus propios escenarios de uso. Usar el método anterior para realizar una actualización es una buena manera si ya tenemos el objeto en la memoria y / o queremos

usar su tipo de seguridad. El uso de parches es la opción si queremos evitar un viaje de ida y vuelta innecesario a la base de datos si aún no tenemos el objeto en la memoria. El inconveniente es que perdemos parte del tipo de seguridad, ya que debemos especificar las propiedades para actualizar mediante el uso de cadenas sencillas (nada de lo que algunos LINQ-magic no pudieron resolver). Veamos el código:

```
private static void RenameOwnerById(IDocumentStore store)
{
    string id = NormalizeOwnerId(ReadNotEmptyString("Enter the Id of the owner to rename."));
    string newName = ReadNotEmptyString("Enter the new name.");

    store.DatabaseCommands.Patch(id, new Raven.Abstractions.Data.PatchRequest[]{
        new Raven.Abstractions.Data.PatchRequest
        {
            Name = "Name",
            Value = newName
        }
    });
}
```

Eso lo envuelve. Debería poder ver cómo funciona el ejemplo pegando los fragmentos de código en una aplicación de consola.

Lea Empezando con ravendb en línea: https://riptutorial.com/es/ravendb/topic/9335/empezando-con-ravendb

Creditos

S. No	Capítulos	Contributors
1	Empezando con ravendb	Balázs, Community