FREE eBook

LEARNING ravendb

Free unaffiliated eBook created from
Stack Overflow contributors.

#ravendb

Table of Contents

About

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.

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

Chapter 1: Getting started with ravendb

Remarks

This section provides an overview of what ravendb is, and why a developer might want to use it.

It should also mention any large subjects within ravendb, and link out to the related topics. Since the Documentation for ravendb is new, you may need to create initial versions of those related topics.

Examples

Installation or Setup

Detailed instructions on getting ravendb set up or installed.

A simple RavenDB console application

For this example we will use the Live Test RavenDB instance.

We will build a simple console app here which demonstrates the most basic operations:

- Creation
- Retrieval by Id
- Querying
- Updating
- Deletion

Begin by creating a new Visual Studio solution and add a Console Application project to it. Let's call it RavenDBDemoConsole. The usage of the client library should be similar if the Reader uses VS Code or his/her favorite editor.

Next, we need to add the required references. Right-click the References node in the Solution Explorer pane and select Manage NuGet packages. Browse online for 'RavenDb.Client'. I'll be using the latest stable release, which is - as of this writing - 3.5.2.

Let's write some code, shall we? Begin by adding the following using statements:

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

These allow us to use RavenDB's `IDocumentStore` and `DocumentStore`, which is an interface and its out-of-the-box implementation to connect to a RavenDB instance. This is the top-level object we need to use to connect to a server and the RavenDB documentation of it advises that it is used as a singleton in the application.

So we will go ahead and create one, but for simplicity, we will not implement the singleton wrapper around it - we will just dispose it when the program exits so the connection is closed in a clean way. Add the following code to your main method:

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

As said at the beginning, we use the Live Test RavenDB Instance, we use its address as the `Url` property of the `DocumentStore`. We also specify a default database name, in this case, "Pets". If the database does not yet exist, RavenDB creates it when trying to access it. If it does exist, then the client can use the existing one. We need to call the `Initialize()` method so we can start operating on it.

In this simple application, we will maintain owners and pets. We think about their connection as one owner may have arbitrary number of pets but one pet may have only one owner. Even though in the real world, one pet might have arbitrary number of owners, for example, a husband and a wife, we will opt to this assumption as many-to-many relationships in a document database is somewhat differently handled from that in a relational database and deserves a topic of its own. I've chosen this domain because it is common enough to grasp.

So we should now define our domain object types:

```
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);
    }
}
```

There are some things to note here:

Firstly, our `Owner`s can contain zero or more `Pet`s. Note that the `Owner` class has a property called `Id` whereas the `Pet` class does not. This is because the `Pet` objects will be stored *inside* the `Owner` objects, which is fairly different from how this sort of relationship would be implemented in a relational database.

One may argue that this should not be implemented like this - and it *may* be right, it really depends on the requirements. As a rule of the thumb, if a `Pet` makes sense to exist without an `Owner` then it should not be embedded but exist on its own with an own identifier. In our application, we assume that a `Pet` is only considered a pet if it has an owner, otherwise it would be a critter or a beast. For this reason, we do not add an `Id` property to the `Pet` class.

Secondly, note that the identifier of the owner class is a string as it is generally shown in the examples in the RavenDB documentation. Many developers used to relational databases may consider this a bad practice, which usually makes sense in the relational world. But because RavenDB uses Lucene.Net to perform its tasks and because Lucene.Net specializes in operating with strings it is perfectly acceptable here - also, we are dealing with a document database which stores JSON and, afterall, basically everything is represented as a string in JSON.

One more thing to note about the `Id` property is that it is not mandatory. In fact, RavenDB attaches its own metadata to any document we save, so even if we didn't define it, RavenDB would have no problems with our objects. It is, however, generally defined for easier access.

Before we see how we can use the RavenDB from our code, let's define a few common helper methods. These should be self-explanatory.

```
// 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();
}
```

And our main method:

```
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");
    }
}
```

Creation

Let's see how we can save some objects into RavenDB. Let's define the following common methods:

```
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();
    }
}
```

Now let's see how it works. We've defined some simple C# logic to create `Owner` objects and keep creating and assigning to it `Pet` objects until the user so desires. The part in which RavenDB is concerned and thus is the focus of this article is how we save objects.

In order to save the newly created `Owner` along with its `Pet`s, we first need to open a session, which implements `IDocumentSession`. We can create one by calling the `OpenSession` on the document store object.

So, note the difference, whereas the document store is a permanent object that generally exists during the whole lifetime of the application, the `IDocumentSession` is a short-lived, lightweight object. It represents a series of operations that we want to perform in one go (or at least, in just a few calls to the database).

RavenDB emphasizes (and somewhat forces) that you avoid excessive numbers of roundtrips to the server, something that they call 'Client-Server chatter protection' on the RavenDB website. For this very reason, a session has a default limit to how many database calls it will tolerate, and so one must pay attention to when a session is opened and disposed of. Because in this example, we treat the creation of an `Owner` and its `Pet`s as an operation which should be executed on its own, we do this in one session and then we dispose it.

We can see two more method calls that are of interest to us:

- `session.Store(owner)`, which registers the object for saving, and additionally, sets the `Id` property of the object if it is not yet set. The fact that the identifier property is called `Id` is therefore a convention.
- `session.Savehanges()` sends the actual operations to execute to the RavenDB server, commiting all pending operations.

Retrieval by Id

Another common operation is to get an object by its identifier. In the relational world, we normally do this using a `Where` expression, specifying the identifier. But because in RavenDB, every *query* is done using indexes, which may be *stale*, it is not the approach to take - in fact, RavenDB throws an exception if we attempt to query by id. Instead, we should use the `Load<T>` method, specifying the id. With our menu logic already in place, we just need to define the method which actually loads the requested data and displays its details:

```
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();
}
```

All that is RavenDB-related here is, once again, the initialization of a session, then using the `Load` method. The RavenDB client library will return the deserialized object as the type we specify as the type parameter. It is important to know that RavenDB does not enforce any sort of compatibility here - all mappable properties get mapped and the nonmappable ones don't.

RavenDB needs the document type prefix prepended to the `Id` - that is the reason for the calling of `NormalizeOwnerId`. If a document with the specified Id does not exist, then `null` is returned.

Querying

We will see two types of queries here: one in which we query over the own properties of `Owner` documents and one in which we query over the embedded `Pet` objects.

Let's start with the simpler one, in which we query the `Owner` documents whose `Name` property starts with the specified string.

```
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();
}
```

One again, because we would like to perform the query as an independent work, we open a session. We can query over a document collection by calling `Query<TDocumentType>` on the session object. It returns an `IRavenQueryable<TDocumentType>` object, on which we can call the usual LINQ-methods, as well as some RavenDB-specific extensions. We do a simple filtering here, and the condition is that the value of the `Name` property starts with the entered string. We take the first 10 items of the result set and create a list of it. One must pay attention to properly specifying the result set size - here is another defensive enforcement at play done by RavenDB called Unbounded result set protection. It means that (by default) only the first 128 items are returned.

Our second query looks like the following:

```
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();
}
```

This one isn't much more complicated, I've written it to demonstrate how naturally it is possible to query over embedded object properties. This query simply returns the first 10 `Owner`s who have at least one `Pet` whose name starts with the entered value.

Deletion

We have two options to perform deletion. One is to pass the document identifier, which is useful if we don't have the object itself in memory but we do have the identifier and we would like to prevent an otherwise avoidable roundtrip to the database. The other way, obviously then, is to pass an actual object saved to RavenDB. We'll look at the first option here, the other one is just using an other overload and passing an appropriate object:

```
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();
    }
}
```

Once again, we need to open a session to perform our work. As mentioned earlier, here we delete the desired object by passing its identifier to the `Delete` method. The identifier prefix should be in place here as well, just like it was the case with the `Load` method. To actually send the delete command to the database, we need to call the `SaveChanges` method, which will do just that, along with any other pending operations that are registered in the same session.

Updating

And finally, we'll take a look at how to update documents. Basically, we have two ways of doing this. The first is straightforward, we load a document, update its properties as necessary, then passing it to the `Store` method. This should be straightforward according to the demonstration of loading and saving, but there are a few things worthy of note.

First, the RavenDB client library uses a change tracker which makes it possible to update any document without actually passing it to `Store` as long as the session that loaded the document is still open. In this case, calling `SaveChanges` on the session is enough for the update to take place.

Secondly, for this to work, the object obviously needs its identifier to be set so that RavenDB can figure out what to update.

With these said, we'll only take a look at the other way of updating. There is a concept called patching, which can be used to update documents. Just like it was the case with delete, it has its own usage scenarios as well. Using the previous method to perform an update is a good way if we already have the object in memory and/or we want to use its type safety. Using patching is the option if we want to avoid an otherwise unnecessary roundtrip to the database if we don't already have the object in memory. The downside is that we lose some of the type safety, since we must specify the properties to update by using plain strings (nothing that some LINQ-magic couldn't solve). Let's see the code:

```
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
        }
    });
```

```
}
```

That wraps it up. You should be able to see the example working by pasting the code fragments into a console app.

Read Getting started with ravendb online: https://riptutorial.com/ravendb/topic/9335/getting-started-with-ravendb

Credits

S. No	Chapters	Contributors
1	Getting started with ravendb	Balázs, Community