RavenDB 4.0

CONTENTS

WRITTEN BY ELEMAR JÚNIOR
CTO, GUIANDO

RavenDB is a mature, multi-platform, NoSQL document-oriented database, safe by default and optimized for efficiency. It is easy to learn and use with several languages — including C#, Java, Python, NodeJS, Ruby, and C++. Being fully transactional, RavenDB supports the best features from NoSQL and the relational world.

With RavenDB, you can achieve 150,000 writes per second and more than one million reads on commodity hardware. It offers a rich integrated GUI available for any license, including a free Community version.

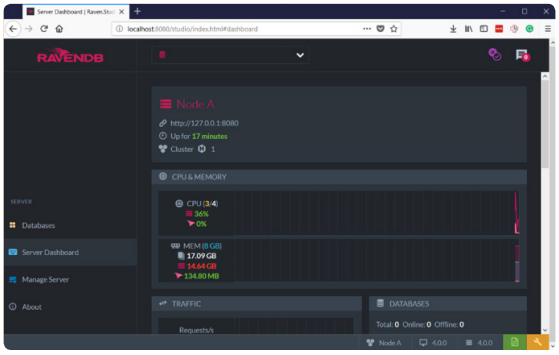
This RefCard is intended to help you get the best results from RavenDB, starting from the basics.

- For installation guidance, see https://ravendb.net/docs/article-page/4.1/csharp/start/installation/setup-wizard
- For a self-directed free learning course, see https://ravendb.net/learn/bootcamp
- For an in-depth reference, see https://ravendb.net/articles/inside-ravendb-book

Starting With the Basics

CREATING YOUR FIRST DATABASE

After installing RavenDB, you can use the Management Studio to easily create your first database.



To create your first database:

RAVENDB

DATA MADE FASTER

Step by Step Guide to Mastering the Document Database

Throughout this decade, the number of developers using a Document Database model for their data has tripled. Documents are the most natural way to store and use data. Not needing a schema, you have massive flexibility to start your application quickly and release your next build fast.

While relational data is traditionally modeled with a pen and paper, the document model is the ideal tool for designing domain-driven applications. Data can be combined from different sources and distributed to multiple places with ease, while enjoying reduced latency, boosting performance and maintaining high availability for your users.

This refcard will teach you to be a NoSQL Document Database expert using RavenDB, an open source Document Database that is fully transactional, giving you the opportunity to explore today's evolving methods of managing data while maintaining the unchanging demands of data integrity. Enjoy a no-cost, no-commitment option to make non-relational the latest tool in your developer arsenal.

HIGH PERFORMANCE

ACID TRANSACTIONS

MULTI-PLATFORM

HIGH AVAILABILITY

MULTI-MODEL ARCHITECTURE

MANAGEMENT STUDIO

FREE LICENSE

Using the NoSQL Document Database Model you will learn to:

- Create your first database with the RavenDB Management Studio GUI
- Import sample data to get started right away
- Understand the document model
- Query a Document Database
- Setup Indexing in RavenDB
- Perform Batch Operations
- Display Revisions and track document's history
- Use Data Subscriptions
- Set up a Distributed Data Cluster

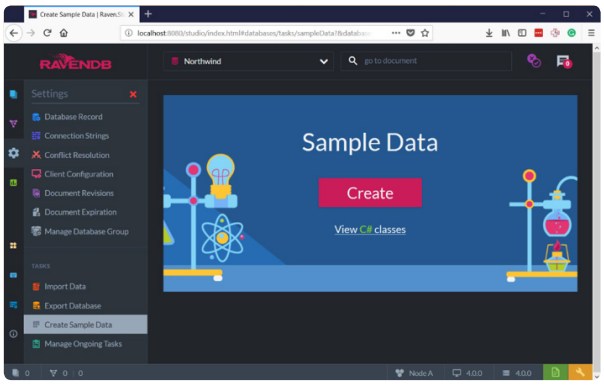
GET FREE LICENSE

Proudly developed by Hibernating Rhinos

1. Select the Database option in the left panel
2. Click on the New Database button
3. Type a name for the new database
4. Click on the Create button

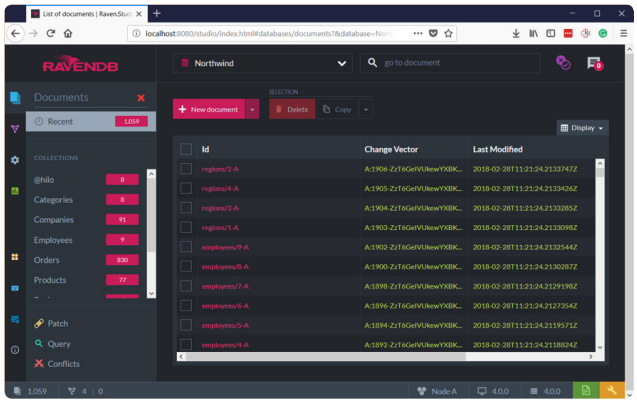
Also, RavenDB provides you with sample data for learning purposes. To load it:

1. Select Databases on the left panel
2. In the right panel, click on the name of the database you just created
3. In the left panel, click on Settings, and then Create Sample Data
4. Click on the big Create button



The sample data comes from the Northwind database, which is the original SQL server sample database that has been used for decades in the Microsoft community. If you are new to the NoSQL world, this is a valuable resource to learn from about NoSQL database modeling.

Going to the Documents section (left panel), you will see the sample data that was loaded into your database.



UNDERSTANDING THE DOCUMENT CONCEPT

A document is a self-describing, hierarchical tree data structure that can consist of maps, collections, and scalar values. RavenDB documents are created with simple JSON.

For example, assuming that you loaded the sample data into a database, you could use the Go To Document feature (the text box in the Studio toolbar), to go to document orders/101-A.

```
{
    "Company": "companies/86-A",
    "Employee": "employees/4-A",
    "Freight": 0.78,
    "Lines": [
        {
            "Discount": 0.15,
            "PricePerUnit": 14.4,
            "Product": "products/1-A",
            "ProductName": "Chai",
            "Quantity": 15
        },
        {
            "Discount": 0,
            "PricePerUnit": 7.2,
            "Product": "products/23-A",
            "ProductName": "Tunnbröd",
            "Quantity": 25
        }
    ],
    "OrderedAt": "1996-11-07T00:00:00.0000000",
    "RequireAt": "1996-12-05T00:00:00.0000000",
    "ShipTo": {
        "City": "Stuttgart",
        "Country": "Germany",
        "Line1": "Adenauerallee 900",
        "Line2": null,
        "Location": {
            "Latitude": 48.7794494,
            "Longitude": 9.1852878
        },
        "PostalCode": "70563",
        "Region": null
    },
    "ShipVia": "shippers/2-A",
    "ShippedAt": "1996-11-15T00:00:00.0000000",
    "@metadata": {
        "@collection": "Orders",
        "@flags": "HasRevisions"
    }
}
```

As you can see, you can aggregate related information into a common object, as in the case of the `ShipTo` property, which has all the shipping information. Also, you can reference other documents (as in the `ShipVia` property).

In a document-oriented database, documents are organized in collections.

UNDERSTANDING THE COLLECTION CONCEPT

Collections provide a good way to establish a level of organization. For example, documents holding customer data are very different

from documents holding product information, and you want to talk about groups of them. A document in RavenDB can "belong" to a collection, which is a string value that groups similar structured documents together (like "Customers" and "Products"). Every document belongs to exactly one collection.

Documents that are in the same collection can have completely different structures. Because RavenDB is schemaless, this is fine.

Querying Basics

Storing and querying for data are basic database operations. RavenDB makes querying easy through RQL.

RQL, the Raven Query Language, is a SQL-like language used to retrieve the data from the server when queries are being executed. It is designed to expose the RavenDB query pipeline in a way that is easy to understand, easy to use, and not overwhelming to the user.

For more information about RQL, see ravendb.net/docs/article-page/4.1/csharp/indexes/querying/what-is-rql

QUERYING FOR THE VERY FIRST TIME

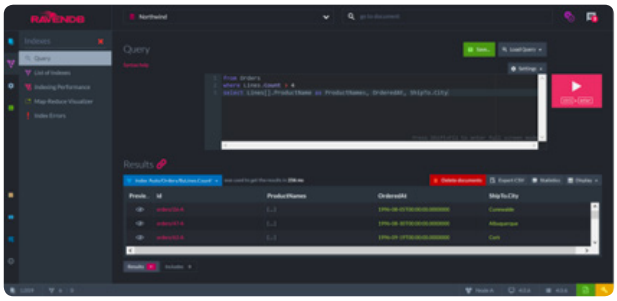
Writing queries is easy using the RavenDB Management Studio. Let's go step-by-step:

1. Open the RavenDB Management Studio
2. In the left panel, click on Databases
3. Open the database we created in the previous lesson (Northwind, if you followed our recommendation)
4. In the left panel, select the Documents section
5. Click on Query
6. Enter your query (using RQL)

```
from Employees
```

7. Click on Run

This query returns all the documents that belong to the Employees collection (from the sample data).



OTHER INTERESTING BASIC QUERIES

Getting all the documents from a collection is nice, but quite useless. Let's make something more exciting.

```
from Employees
where FirstName == "Nancy"
```

FirstName is the name of one of the properties present in the documents from the Employees collection.

The next query shows how to shape the data returned from the server. This query returns the name of all products, ordering date, and shipping city from all orders with more than four lines.

```
from Orders
where Lines.Count > 4
select Lines[].ProductName as ProductNames, OrderedAt,
ShipTo.City as City
```

We can also use JavaScript to define the shape of the query results. Also, this query shows how to combine two or more documents in a single result. As you noticed, the Company field of an Order document contains the ID of another document stored in the database. The load instruction is smart enough to get that document for you.

```
from Orders as o
load o.Company as c
select {
    Name: c.Name.toLowerCase(),
    Country: c.Address.Country,
    LinesCount: o.Lines.length
}
```

Another important basic querying feature is to aggregate data. The following query groups the Orders using the Company field as a grouping key. Also, there is a filter to get only groups with six documents at least, and ordering criteria by the number of elements per group in descending order. Finally, we are projecting the number of documents per group and the group key.

```
from Orders
group by Company
where count() > 5
order by count() desc
select count() as Count, key() as Company
```

Accessing RavenDB Using C#

Hibernating Rhinos provides official client APIs for the most popular programming languages. When using .NET, you need to install the RavenDB.Client NuGet package.

```
Install-Package RavenDB.Client
```

CONNECTING TO THE SERVER

To connect to the RavenDB server, you will need to create a `DocumentStore` object.

```
var documentStore = new DocumentStore
{
    Urls = new [] {"http://localhost:8080"},
    Database = "Northwind"
};

documentStore.Initialize();
```

The document store is the main client API object that establishes and manages the connection channel between an application and a database instance. It acts as the connection manager and also exposes methods to perform all the operations you can run against an associated server instance. The document store object has an array of URL addresses to the cluster nodes, and it can work against multiple databases that exist there.

For more information about the `DocumentStore`, see ravendb.net/docs/article-page/4.0/csharp/client-api/what-is-a-document-store

You should only need to create a single instance of the `DocumentStore` object for a program's lifetime. To do that, I would recommend that you provide it through a Singleton interface.

```
public static class DocumentStoreHolder
{
    private static readonly Lazy<IDocumentStore>
       LazyStore =
       new Lazy<IDocumentStore>(() =>
       {
           var store = new DocumentStore
           {
               Urls = new[] { "http://localhost:8080" },
               Database = "Northwind"
           };

           return store.Initialize();
        });

    public static IDocumentStore Store =>
        LazyStore.Value;
}
```

The use of Lazy ensures that the document store is only created once, without having to worry about locking or other thread safety issues.

YOUR FIRST SESSION

The session is the primary way your code interacts with RavenDB. You need to create a session via the document store, and then use the session methods to perform operations.

A session object is very easy to create and use. To create a session, simply call the `DocumentStore.OpenSession()` method.

```
using (var session =
    DocumentStoreHolder.Store.OpenSession())
{
    // ready to interact with the database
}
```

LOADING DOCUMENTS

To load documents, you will use the session's load method.

As the name implies, the Load method gives you the option of loading a document or a set of documents, passing the document(s) ID(s) as parameters. The result will be an object representing the document, or null if the document does not exist.

A document is loaded only once in a session. Even though we call the Load method twice passing the same document ID, only a single remote call to the server will be made. Whenever a document is loaded, it is added to an internal dictionary managed by the session.

```
using (var session =
    DocumentStoreHolder.Store.OpenSession())
{
    var p1 = session.Load<Product>("products/1-A");
    var p2 = session.Load<Product>("products/1-A");
    Debug.Assert(ReferenceEquals(p1, p2));
}
```

LOADING TWO RELATED DOCUMENTS WITH A SINGLE SERVER REQUEST

The easiest way to kill your application performance is to make a lot of remote calls. RavenDB provides a lot of features to help you significantly reduce calls and boost performance.

Consider the Northwind `products/1-A` document:

```
{
    "Name": "Chai",
    "Supplier": "suppliers/1-A",
    "Category": "categories/1-A",
    "QuantityPerUnit": "10 boxes x 20 bags",
    "PricePerUnit": 18,
    "UnitsInStock": 1,
    "UnitsOnOrder": 39,
    "Discontinued": false,
    "ReorderLevel": 10,
    "@metadata": {
        "@collection": "Products"
    }
}
```

As you can see, the Supplier and Category properties are references to other documents.

Considering you need to load the product and the related category, how would you write the code? Your first attempt might look something like this:

```
var p = session.Load<Product>("products/1-A");
var c = session.Load<Category>(p.Category);
```

This approach will make two remote calls , which is not good. Instead, use the following:

```
var p = session
    .Include<Product>(x => x.Category)
    .Load("products/1-A");
var c = session.Load<Category>(p.Category);
```

The Include session method changes the way RavenDB will process the request.

It will:

1. Find a document with the ID: products/1-A
2. Read its Category property value
3. Find a document with that ID
4. Send both documents back to the client

When the `session.Load(p.Category);` is executed, the document is already in the session cache, and no additional remote call is made.

Here is a powerful example of the application of this technique in a complex scenario.

```
var order = session
    .Include<Order>(x => x.Company)
    .Include(x => x.Employee)
    .Include(x => x.Lines.Select(l => l.Product))
    .Load("orders/1-A");
```

This code will, in a single remote call, load the order, including the company and employee documents, and also load *all* the products in all the lines in the order.

QUERYING FROM C#

While coding C#, you can make queries using LINQ or RQL.

For LINQ, you need to open a session and write your query using the session's Query method.

```
var orders = (
    from order in session.Query<Order>()
                         .Include(o => o.Company)
    where order.Company == companyReference
    select order
    ).ToList();
```

For RQL, you need to open a session and write your query using the session's RawQuery method.

```
var orders = session.Advanced.RawQuery<Order>(
```

CODE CONTINUED ON NEXT COLUMN

```
        @"from Orders
        where Company == $companyId
        include Company"
    ).AddParameter("companyId", companyReference);
```

It's important to know that all queries are translated to RQL before sending to the server.

STORING, MODIFYING, AND DELETING DOCUMENTS

The easiest way to learn how to store, modify, and delete documents from the database is with an example.

```
// storing a new document
string categoryId;
using (var session =
    DocumentStoreHolder.Store.OpenSession())
{
    var newCategory = new Category
    {
        Name = "My New Category",
        Description = "Description of the new category"
    };
    session.Store(newCategory);
    categoryId = newCategory.Id;
    session.SaveChanges();
}
// loading and modifying
using (var session =
    DocumentStoreHolder.Store.OpenSession())
{
    var storedCategory = session
        .Load<Category>(categoryId);
    storedCategory.Name = "abcd";
    session.SaveChanges();
}
// deleting
using (var session =
    DocumentStoreHolder.Store.OpenSession())
{
    session.Delete(categoryId);
    session.SaveChanges();
}
```

Any .NET object can be stored by RavenDB. It only needs to be serializable to JSON.

The `Store` method is responsible for registering the "storing" intention in the session. You can access the document right after the `Store` call was made, even though the document was not saved to the database yet. The `SaveChanges` method applies the registered actions in the session to the database.

When you change the state of an entity, the session is smart enough to detect it and update the matching document on the server side. The session keeps track of all the entities you have loaded or stored (with `Load` or `Query` methods), and when you call `SaveChanges`, all changes to those entities are sent to the database in a *single remote call*.

The `Delete` method, which we have used in the last part of the code, will delete the matching document on the server side. You can provide the document ID or an entity instance.

All the changes are applied on the server side only after you call the `SaveChanges` method. This is how RavenDB supports transactions.

A Few Words About Indexing

An index is a data structure that the RavenDB engine uses to perform all queries. Thanks to this data structure, RavenDB can quickly locate data without having to search every document in the database.

RavenDB is safe by default, and whenever you make a query, the query optimizer will try to select an appropriate index to use. **If there is no such appropriate index, then the query optimizer will create an index for you.**

For more detailed information about indexing, see ravendb.net/docs/article-page/4.1/csharp/indexes/what-are-indexes

CREATING YOUR FIRST INDEX

This is the basic process for creating an index:

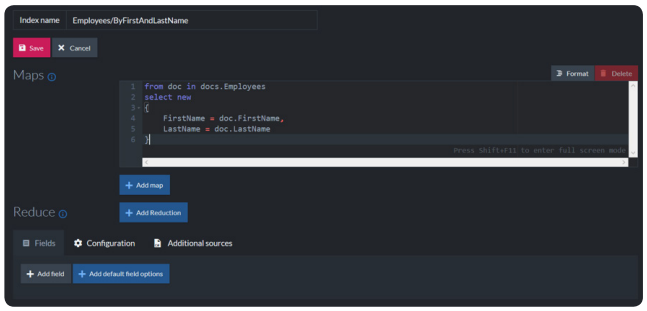
1. Access your database using the Management Studio
2. Go to the Indexes section, then List of Indexes, click on New Index
3. Specify the index name
4. Specify the Map function (LINQ syntax)

RavenDB does not have fixed naming rules, but I strongly recommend that you follow a naming convention. For example, Employees/ByFirstAndLastName (collection name/By selected fields Of filtering criteria)

Here is an example of a map function for indexing the Employees documents by FirstName and LastName.

```
from doc in docs.Employees
select new
{
    FirstName = doc.FirstName,
    LastName = doc.LastName
}
```

In the expression, the docs object represents the entire collection of documents stored in the database. The `docs.Employees` object represents all documents from the Employees collection.



RQL provides a special syntax when you want to specify what index you want to use. Here is an example:

```
from index 'Employees/ByFirstNameAndLastName' as e
where e.LastName = "Davolio"
```

DEFINING A MULTI-MAP INDEX USING C#

Multi-Map indexes allow you to index data from multiple collections e.g. polymorphic data or any common data between types.

Everything that you can do using the Management studio, you can do using the RavenDB client API.

The following code shows how to create a complex index that maps three different types of documents.

```
using System.Linq;
using Raven.Client.Indexes;

namespace MultimapIndexes
{
    public class People_Search :
        AbstractMultiMapIndexCreationTask<People_Search.
        Result>
    {
        public class Result
        {
            public string SourceId { get; set; }
            public string Name { get; set; }
            public string Type { get; set; }
        }

        public People_Search()
        {
            AddMap<Company>(companies =>
                from company in companies
                select new Result
                {
                    SourceId = company.Id,
                    Name = company.Contact.Name,
                    Type = "Company's contact"
                }
            );
```

CODE CONTINUED ON NEXT PAGE

```
            AddMap<Supplier>(suppliers =>
                from supplier in suppliers

                select new Result
                {
                    SourceId = supplier.Id,
                    Name = supplier.Contact.Name,
                    Type = "Supplier's contact"
                }
            );

            AddMap<Employee>(employees =>
                from employee in employees
                select new Result
                {
                    SourceId = employee.Id,
                    Name = $"{employee.FirstName}
                    {employee.LastName}",
                    Type = "Employee"
                }
            );

            Index(entry => entry.Name, FieldIndexing.
            Search);

            Store(entry => entry.SourceId, FieldStorage.
            Yes);
            Store(entry => entry.Name, FieldStorage.Yes);
            Store(entry => entry.Type, FieldStorage.Yes);
        }
    }
}
```

You can define as many map functions as you need. Each map function is defined using the `AddMap` method, and has to produce the same output type. The "source" collection is specified by the generic parameter type you specify in the `AddMap` function (the type is the same type that you use to retrieve documents as objects in the client-side).

The Index method here was used to mark the Name property as FieldIndexing.Search which enables full-text search with this field.

The `Store` method was used to store those defined properties along with the Index. When a query is made on these stored fields, the results come directly from the index, instead of having to load the document and get the fields from it. In most cases, this isn't an interesting optimization. RavenDB is already heavily optimized toward loading documents. Use this when you are creating new values in the indexing function and want to project them, not merely skip the (pretty cheap) loading of the document.

When creating indexes in the client-side, you need to remember registering them in the server. You do that during the `DocumentStore` object initialization.

```
var store = new DocumentStore
                {
                    Urls = new[] { "http://
                    localhost:8080" },
                    Database = "Northwind"
                };

store.Initialize();

var asm = Assembly.GetExecutingAssembly();
IndexCreation.CreateIndexes(asm, store);
```

PERFORMING ADVANCED QUERIES

For querying using a multi-map index, you need to use a special interface from the RavenDB client.

```
public static IEnumerable<People_Search.Result> Search(
    IDocumentSession session, string searchTerms
)
{
    var results = session
    .Query<People_Search.Result, People_Search>()
        .Search( r => r.Name, searchTerms )
        .ProjectInto<People_Search.Result>()
        .ToList();

    return results;
}
```

As you can see, you can specify the index to use and the shape of the result.

The query's Search method allows you to perform advanced querying using all the power of the Lucene engine (underlying used by RavenDB), including wildcards.

For more detailed information about querying, see ravendb.net/docs/article-page/4.1/csharp/indexes/querying/basics

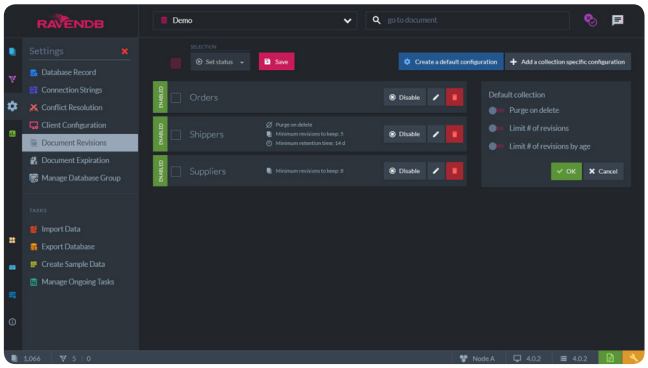
Revisions

Every time you update or delete a document, RavenDB 4 can create a snapshot (revision) of the previous document state. This is useful when you need to track the history of the documents or when you need a full audit trail.

You can choose to keep track of the last N revisions. If necessary, you could "track everything."

HOW TO ENABLE REVISIONS

You can configure the revisions feature using the Studio:



When activated, by default, RavenDB will track the history for all documents. Also by default, RavenDB will never purge old revisions.

As the administrator, you can configure this for all collections. You can also specify a different setup for a specific collection.

The options are:

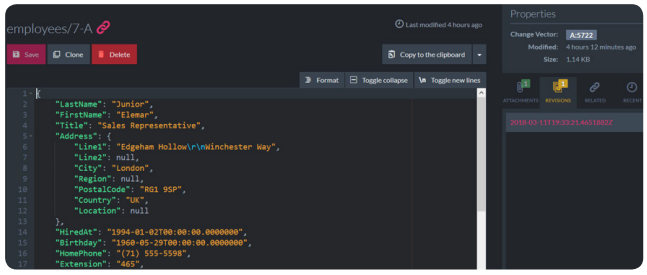
- **Purge on Delete**: Delete the revisions upon the document delete
- **Limit # of revisions**: How many revisions to keep
- **Limit # of revisions by age**: Configure a minimum retention time before the revisions can be expired

The settings can be changed according to your needs.

For how to configure revisions programmatically, see ravendb.net/docs/article-page/4.1/csharp/server/extensions/revisions

RETRIEVING A DOCUMENT'S REVISION

When using the Management Studio, it is easy to get access to historic information about a document by accessing the Revisions tab.



Programmatically, you need to use the session's advanced interface:

```
static void Main(string[] args)
{
    using (var session =
        DocumentStoreHolder.Store.OpenSession())
    {
        var revisions = session.Advanced.Revisions
            .GetFor<Employee>("employees/7-A");
```

CODE CONTINUED ON NEXT COLUMN

```
        foreach (var revision in revisions)
        {
            // process revision
        }
    }
}
```

Commands and Operations

The session is a high-level interface to RavenDB that tracks entities and provides LINQ queries. To do something low-level or advanced, you should start using Commands and Operations.

For more detailed information about commands and operations, see ravendb.net/docs/article-page/4.1/csharp/client-api/operations/what-are-operations.

ADDING AN ORDER'S LINE INTO AN ORDER DOCUMENT WITHOUT LOADING THE ENTIRE DOCUMENT

So far, we have seen how to change a document by loading it, modifying it, and then storing it again. But there is a cheaper way.

```
using (var session =
    DocumentStoreHolder.Store.OpenSession())
{
    session.Advanced.Defer(new PatchCommandData(
        id: "orders/816-A",
        changeVector: null,
        patch: new PatchRequest
        {
            Script = "this.Lines.push(args.NewLine)",
            Values =
            {
                {
                    "NewLine", new
                    {
                        Product = "products/1-a",
                        ProductName = "Chai",
                        PricePerUnit=18M,
                        Quantity=1,
                        Discount=0
                    }
                }
            }
        },
        patchIfMissing: null));

    session.SaveChanges();
}
```

In the above example, you use the `Patch` command which performs partial document updates without having to load, modify, and save a full document. The Script needs to be in JavaScript.

If you have types, and you probably do, you can also use a typed version:

```
static void Main()
{
    using (var session =
        DocumentStoreHolder.Store.OpenSession())
    {
        session.Advanced.Patch<Order,
                               OrderLine>("orders/816-A",
            x => x.Lines,
            lines => lines.Add(new OrderLine
            {
                Product = "products/1-a",
                ProductName = "Chai",
                PricePerUnit = 18M,
                Quantity = 1,
                Discount = 0
            }));

        session.SaveChanges();
    }
}
```

PERFORMING A BATCH OPERATION

Batch operations provide a good alternative to performing operations that affect a lot of documents.

The following example increments the price of all products that were not discontinued.

```
static void Main()
{
  var operation =
    DocumentStoreHolder.Store.Operations
    .Send(new PatchByQueryOperation(@"from Products as p
        where p.Discontinued = false
        update
           {
               p.PricePerUnit = p.PricePerUnit * 1.1
           }"));
    operation.WaitForCompletion();
}
```

For more interesting examples about batch operations, see ravendb.net/docs/article-page/4.1/csharp/client-api/operations/patching/set-based

How to Get Notified Whenever a Document Changes

RavenDB provides a very useful, reactive interface to share information whenever a document is changed.

```
using System;
using Raven.Client;
using Raven.Client.Documents;
namespace BasicsOfChangesAPI
```

CODE CONTINUED ON NEXT COLUMN

```
{
    using static Console;

    class Program
    {
        static void Main(string[] args)
        {
            using (var subscription =
            DocumentStoreHolder.Store
                .Changes()
                .ForAllDocuments()
                .Subscribe(change =>
                    WriteLine($"{change.Type} on
                    document {change.Id}")))
            {
                WriteLine("Press any key to exit...");
                ReadKey();
            }
        }
    }
}
```

If you are not familiar with reactive programming, the lambda function will be invoked whenever a change occurs in the server side.

You could use the `Where` function to introduce a filter.

IMPORTANT: To use the reactive extension, please add references to System.Reactive.Core and System.Reactive.Linq NuGet packages.

Data Subscriptions

Data Subscription is a powerful feature that is simpler to explain with an example. Consider the following query:

```
from Orders
where Lines.length > 5
```

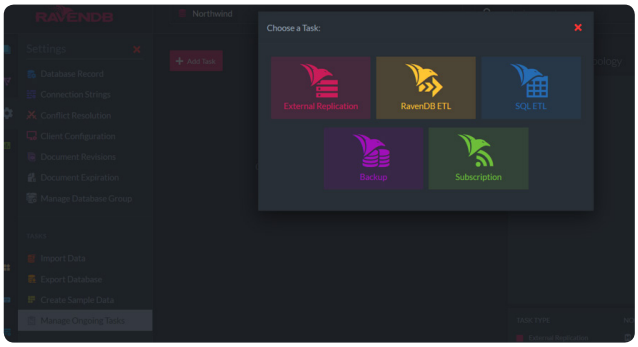
This query would retrieve all the big orders from your database. But what if a new big order is added after you run this query? What if you want to be notified whenever a big order occurs? This is what the Data Subscription feature allows you to do.

There are some important facts that you need to know to use this feature correctly.

- Documents that match the pre-defined subscription criteria are sent in batches from the server to the client.
- The client sends an acknowledgment to the server once it is done with processing the batch.
- The server keeps track of the latest document that the client has acknowledged so that the next batch sent to the client can continue from the latest acknowledged position if the connection to the client was paused or interrupted.

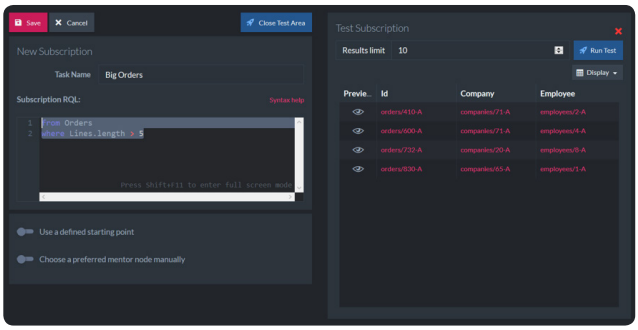
CREATING A DATA SUBSCRIPTION

To create a Data Subscription, you need to open the Settings section of your database. Click on the Manage Ongoing Tasks option, then click on the Add Task button.



Then, click on Subscription.

Enter a name for your subscription task (for example, Big Orders) and then you can provide the very same query we wrote before.



Test and save it!

CONSUMING A DATA SUBSCRIPTION

Data Subscriptions are consumed by clients, called subscription workers. Only a single worker can be connected to a specific data subscription in any given moment. A worker connected to a data subscription receives a batch of documents and gets to process it. When the worker is done (depending on the code that the client gave the worker, this can take seconds or hours), the client informs the server about the progress, so that the server can send the next batch.

Here is an example of how to consume a Data Subscription.

```
static void Main(string[] args)
{
    var subscriptionWorker =
        DocumentStoreHolder.Store.Subscriptions
        .GetSubscriptionWorker<Order>("Big Orders");

    var subscriptionRuntimeTask = subscriptionWorker.
        Run(batch =>
           {
```

CODE CONTINUED ON NEXT COLUMN

```
            foreach (var order in batch.Items)
            {
                // business logic here.
                Console.WriteLine(order.Id);
            }
        });

    WriteLine("Press any key to exit...");
    ReadKey();
}
```

Clustering

A RavenDB cluster is a set of machines that have been joined together. Each machine is a node in the cluster.

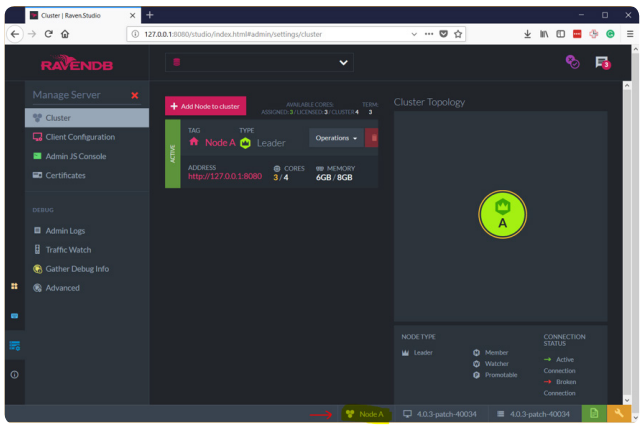
When you create a database, it can live on a single node (one machine in the cluster), some number of the nodes, or even all the nodes. Each node will hold a complete copy of the database and will be able to serve all queries, operations, and writes. RavenDB clusters implement multi-master replication.

The primary reason for duplicating data is to allow high availability. If a node goes down, the cluster will still have copies of the data. Clients are redirected to another node without realizing that anything happened.

The cluster distributes work among the nodes, handles failures, and adopts recovery procedures automatically.

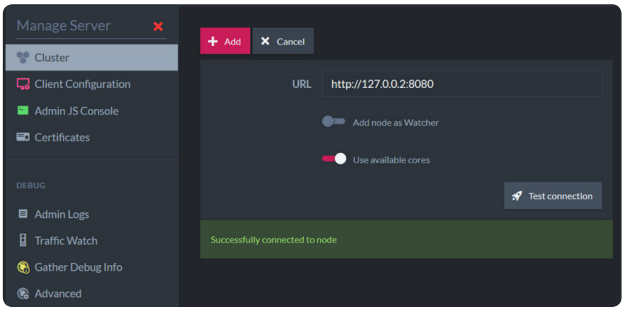
YOUR FIRST CLUSTER

Assuming that you have started the RavenDB server on your computer, you are already using a cluster. It's a cluster with a single node, but it is still a cluster.

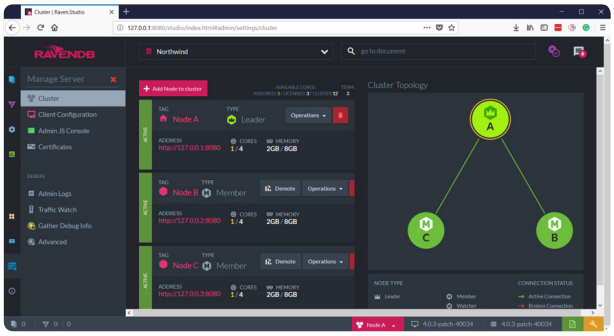


ADD OTHER SERVERS AS NODES IN THE CLUSTER

Adding nodes to an existing cluster is easy. Open the Management Studio from the first server, then, in the left panel, select the option Manage Server and then Cluster. Click the “Add Node To Cluster” button. Enter the address of the second server and hit Test Connection.



RavenDB will show you an updated topology:



SOME CONCEPTS THAT YOU NEED TO KNOW

Now that you know what a cluster is and how to add nodes to it, it's time to learn some new concepts.

- **Database** — the named database we're talking about, regardless of whether we're speaking about a specific instance or the whole database group
- **Database Instance** — exists on a single node
- **Database Group** — the grouping of all the different instances, typically used to explicitly refer to its distributed nature
- **Database Topology** — the specific nodes that all the database instances in a database group reside on at a particular point in time

Consider you have a cluster with five nodes. It is possible to create a database setting with a replication factor of just three. In this case, only three nodes will have instances of the database. The selected nodes will form the Database Topology, and all the three instances will constitute the Database Group.

DISTRIBUTED, YES! SO, CONSISTENT OR AVAILABLE?

Operations in RavenDB can be classified into two categories:

1. **Cluster-Wide Operations**: They impact the entire cluster, like creating a new database
2. **Internal Database Operations**: They impact a single database, like creating a new document

This distinction is vital because cluster-wide operations demand consensus between the nodes of the cluster. This is only possible when the majority of the nodes are working. RavenDB uses a consensus protocol called RAFT. To create a database in a cluster with three nodes, it would be necessary that at least two nodes are working (the majority). This is the reason why you should not have a cluster with only two nodes. When one is down, there is no majority.

There are a lot of details about how the RAFT consensus protocol is implemented in RavenDB that are beyond the scope of this bootcamp.

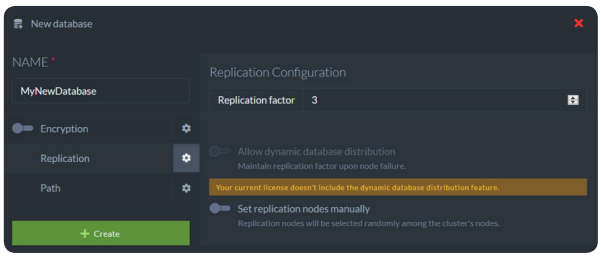
For an in-depth understanding about RavenDB's RAFT implementation, see ravendb.net/articles/inside-ravendb-book

Database operations, on the other hand, are treated quite differently. Each node in the cluster has a full copy of the topology that specifies which node hosts which database. The connection between the database instances does not use the consensus protocol. Instead, they're direct connections among the various nodes, forming a multi-master mesh. A write to a database son any node in the cluster is automatically replicated to all the other nodes that are in the database group.

Cluster Consensus provides firm consistency. Unfortunately, we also need to ensure the availability of the databases, and it is a tradeoff (look for the CAP theorem to understand the reasons). We have decided to provide consistency where it is indispensable and explicitly chose to ensure availability elsewhere.

CREATING A DATABASE WITH REPLICATION

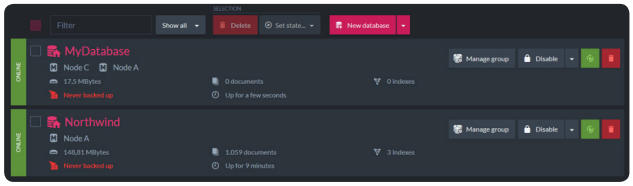
Whenever you create a new database, you can specify a replication factor. That is the number of the cluster nodes that will contain an instance of the database.



MANAGING THE DATABASE GROUP

As you already know, the database group is the set of cluster nodes containing the instances of a database.

You can manage the database group of a specified database by accessing the Databases option in the left panel.



The above image displays a database (MyDatabase) that has instances on two nodes, Node C and Node A.

You can see a status of Online, indicating that MyDatabase is active (meaning that I have an instance of this database running on this cluster node). When accessing the same screen on a server that has no instance of MyDatabase (Node B, running on http://127.0.0.2:8080 in my example), the status will show as Remote.

Clicking on the Manage Group button of the database that you want to manage allows you to modify the list of nodes that will contain instances of the selected database.

It is important to remember that managing a database group is a cluster-wide operation that requires a majority to work.

CONNECTING TO A CLUSTER

Whenever you create an instance of the DocumentStore, you should specify the list of nodes in the cluster that you will be connecting to.

```
var store = new DocumentStore
{
    Urls =
    {
      "http://127.0.0.1:8080" ,
      "http://127.0.0.2:8080" ,
      "http://128.0.0.3:8080"
    },
    Database = "Northwind"
};
store.Initialize();
```

As we mentioned earlier, all cluster nodes contain the full topology for all the databases hosted in the cluster. The very first thing that a client will do upon initialization is query the defined URLs to find which nodes contain the database that you are trying to connect to.

Written by **Elemar Júnior,** *CTO at Guiando*

Elemar has been developing world-class software for more than 20 years. He is a CTO at Guiando and RavenDB evangelist who helps developers, architects, and IT executives to produce software that meets their business needs. Familiar with low-level coding, he has been developing parsers, compilers, and interpreters all his life. He is a co-author of the Code-Cracker project, an analyzer library for C# and VB that uses Roslyn to produce refactorings, code analysis, and other niceties. He has been blogging and speaking at events like QCon, TDC, and Microsoft TechEd about software architecture, domain-driven design, innovation strategy, code performance, process-driven architecture, software integration, automation, and NoSQL for years.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC

888.678.0399 919.678.0300