Advanced Databases Project

NoSQL Database : RavenDB

Mohamed Chajii

Julio Cesar Velasco A.

Introduction

Our project is based in a NoSQL database called RavenDB, an open source second generation NoSQL document database written in .NET. With its schema-less and flexible data model, it is designed to meet the real-world applications and allow to build high-performance applications quickly and efficiently.

1 NoSQL Databases

1.1 What is NoSQL

NoSQL is a term used when talking about non-traditional, refers to any data store that does not follow the traditional RDBMS model, specifically, the data is non-relational and they don't use the Codd's model[1].

It basically defines a database that uses other Data Manipulation Language (DML) than SQL. This is why NoSQL is literally a combination of two words: No and SQL and it means "**Not only SQL**".

In a relational database, we can have a table and in this table an ID column which is used as a foreign key in another table. NoSQL Database does not have relational enforcement and it does not use SQL language to interact with stored data.

A NoSQL System aims to be easy to scale out. Scalability is the ability of a system to change its size while maintaining proportions and this usually means to increase throughput with addition of resources to address load increases. The problem with Relational Database Management System (RDBMS) isn't that they don't scale, it's that it becomes cost prohibitive to scale vertically, versus scaling horizontally. Vertical scaling means that you scale by adding more power (CPU, RAM, Hard disk space) to your existing machine, while horizontal scaling means that you scale by adding more machines into your pool of resources.

1.2 Why NoSQL

According to the last paragraph, It doesn't mean that we should look at NoSQL only when we start reaching the problems of scale. NoSQL databases have a lot more to offer than just solving the problems of scale like :

- **Schemaless data representation :** Almost all NoSQL implementations offer schemaless data representation. This means that you don’t have to think too far ahead to define a structure and you can continue to evolve over time, including adding new fields or even nesting the data, for example, in case of JSON representation (Custom Serialization / Deserialization in RavenDB).

- **Development time :** Nowadays, searching on internet references, It is very common heard stories about reduced development time because one doesn’t have to deal with complex SQL queries.

[1] E.F. Codd was a famous mathematician who introduced 12 rules for the relational model for databases commonly known as Codd's rules. The rules mainly define what is required for a DBMS for it to be considered relational.

• **Speed :** Even with the small amount of data that an app can have, if you can deliver in milliseconds rather than hundreds of milliseconds (especially over mobile and other intermittently connected devices) you have much higher probability of winning users over.

• **Plan ahead for scalability :** Your application can be quite elastic, it can handle sudden spikes of load.

1.3 Types of NoSQL Databases

NoSQL is a very large concept and it involves a large number of technologies. In this section, we will introduce the different types of NoSQL but we will focus later only on the RavenDB NoSQL Database type, the document-oriented database.

Currently, the NoSQL databases types can be categorized in more than ten types. But we will show only the four major different types:

- **Key-value databases** : A key-value database has a key and a corresponding value together as dataset. In key-value models, the key has to be unique and the value is associated with that key. Values can be of different types such as strings, integers, floats, or byte arrays. Some databases do not even care about the data type of the value, they just store the data sent. These inputs are called **blobs**, meaning an arbitrary binary object, which the database does not need to interpret.

  Examples of key-value NoSQL Databases are Amazon Dynamo, Redis.

- **Column-oriented databases** : The basic idea behind column-oriented databases is that the data is not a dataset with its attributes stored in one unit as in SQL databases (**row oriented**) but one attribute of a set of datasets is stored in one unit (**column oriented**). The column-oriented databases store data as columns as opposed to rows that is prominent in RDBMS.

  Examples of column-oriented NoSQL Databases are Google's BigTable, Cassandra, HBase, Hypertable.

- **Graph-oriented databases** : The graph-oriented databases originated from the graph theory, where you have vertices (singular: vertex) and edges connecting the vertices. In databases, vertices are entities like persons and edges are relations between entities. These relations are similar to the relations in RDBMS. Twitter uses FlockDB the graph-oriented database, Facebook and LinkedIn use also a graph-oriented NoSQL database.

  Another examples of graph-oriented NoSQL Databases are Neo4j, OrientDB and InfiniteGraph.

- **Document-oriented databases :** The document-oriented databases use entire documents of different types as datasets. The document types are structured human readable data format such as **XML** or **JSON**. Document-oriented databases can be interpreted as particular cases of a key-value database.

The database knows the format of document and it interprets it. This is the biggest difference to key-value storage. Document store databases are schema free. In a schema free database you don't have to predefine what your documents looks like.

For the storage, this does not seem to be a problem, because the documents do not depend on each other. When the documents have some kind of structure or schema it is called semi-structured.

Examples of document-oriented databases are CouchDB, MongoDB, and obviously RavenDB.

RavenDB is a NoSQL document store database. In the context of RavenDB or a document-oriented database, you can think of a document as a web page with a hierarchical nested structure and all this data can be retrieved and fetched in one time. A document store can have a key and then it will have a document associated with that key.

2 RavenDB

2.1 What is RavenDB ?

RavenDB is an open source document-oriented NoSQL designed especially for the .NET/ Windows platform. It requires commercial licensing for some features that we will see in the next pages. A free edition is available for open source projects.

RavenDB supports multiple databases and, as other database servers, a database acts as a container of data. RavenDB can easily handle hundreds or thousands of databases on the same instance and was explicitly designed with multi-tenancy[2] in mind. This allows RavenDB to manage large numbers of databases, but at any given time, only one database is active and taking resources.

Each RavenDB database contains a set of documents which can be divided into collections[3]. A Collection is a logical way of thinking of documents groups. Within a Collection, documents share the same entity name.

A Document is a unit of data and it contains a set of fields or key-value pairs. The keys are strings; the values can be of various types such as strings, integers, etc. You can even store a document as the value of a field. RavenDB database contains Indexes which works differently than RDBMS indexes. RavenDB uses indexes to satisfy queries. You may index the whole query's field or specify the fields you want to index in a document. RavenDB takes the query, analyses it and extracts an index which can answer the query.

RavenDB uses JSON (JavaScript Object Notation) to store documents. JSON is a way to serialize data and is a **lightweight data-interchange** format.

[2] Multi-tenancy is a term that is used often in the context of SaaS applications. In general, multi-tenancy refers to the ability to run multiple users of an application on a shared infrastructure. The main motivation for doing this is efficiency, or in other words, reducing the cost per user in comparison to a dedicated system where each user has their own dedicated environment.

[3] RavenDB's collections are very similar to MongoDB's collections which is another document-oriented NoSQL database.

As said in the beginning, RavenDB does not have a schema (schema-less) and documents do not have to have a specific field or column. Also, there are no explicit relations among the documents that exist in RavenDB.

RavenDB is fully transactional. That means, it fully supports both implicit and explicit transactions and we can take full advantage of this feature. Unlike most NoSQL databases that adhere to the BASE properties, RavenDB supports the ACID properties for write operations. ACID, as we all know, stands for Atomicity, Consistency, Isolation, and Durability, and support for ACID is what makes us so comfortable using RDBMS systems with respect to data integrity. BASE, on the other hand, stands for Basically Available, Soft state, Eventual consistency.

2.2 Structure Comparison to Relational databases

The figure 1 gives a quick view of a relational data model and their equivalent with those of the RavenDB data model :

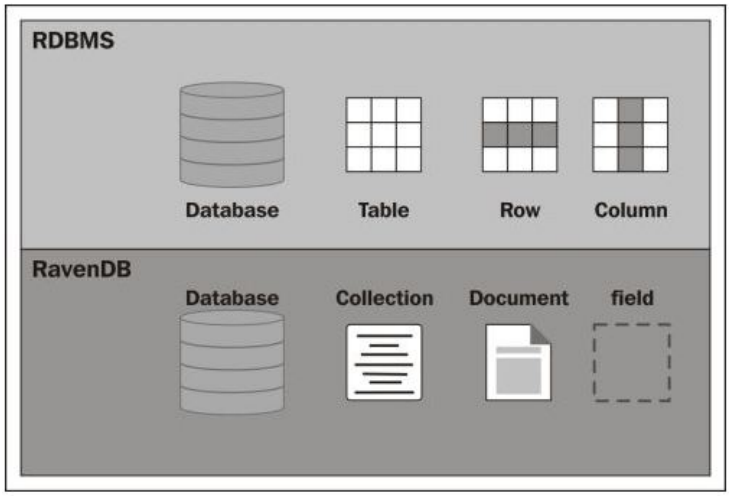


Figure 1 : RavendDB data model comparison to RDBMS

When working with RavenDB, It is very important to consider that RavenDB is non-relational. The reason is that RavenDB treats each document as an independent entity. There are no foreign keys in a collection and as a result, there are no JOIN queries. Constraint management is typically handled in the application layer. Also, because of its flexible schema property, there is no need for the expensive ALTER TABLE statement in RavenDB and you can have user defined content much more easily.

RavenDB is able to store large amounts of data and also is able to optimize the way documents are stored and managed. It supports **Horizontal scaling** and each document can be stored on any node of the horizontal resource pool (sharding[4]).

[4] More information can be found in the official documentation : http://ravendb.net/docs/2.0/server/scaling-out/sharding

2.2 Advantages of adopting RavenDB

The benefits of NoSQL solutions depend on the case that it will be used. When considering a NoSQL solution, users must choose the appropriate database management system that best fits their applications. An appropriate choice brings best performance or decrease a costs.

Some advantages found are :

- RavenDB is written in C#, .NET, and it is easy to learn how to use it. This is a real advantage. Data can be queried efficiently using LINQ queries from .NET code or using RESTful (REpresentational State Transfer) APIs.

- In RavenDB, the database schema is no longer fixed, data is stored schema-less as the JSON documents, so the documents can have arbitrary structures and attributes associated with them. Internally, RavenDB makes use of Indexes which are automatically created based on your usage, or were created explicitly by the consumer.

- RavenDB is highly scalable and is built for **web-scale**.

- RavenDB is fully transactional with the ACID support.

RavenDB can be used in many cases. For example, RavenDB can be used to archive a huge number of documents, it can be used as a content management database, to store orders, inventory, and suppliers in an e-commerce solution. Examples found on internet shows that there are a lot of real estate / rental people using RavenDB.

But the case in that it is not recommended to use RavenDB for is reporting. This is because in many cases, reporting requires dynamic data aggregation over large dataset, and that isn't an OLTP (**Online Transaction Processing**) task, which is what RavenDB was designed for.

2.3 How does RavenDB works

When you are using RavenDB, basically you have a client application that communicates with a database server. The client application will communicate with RavenDB database server over HTTP. It will look like any web service and REST (Representational State Transfer) based web service.

REST is an architecture that uses the strengths of the web to build services. It proposes a set of constraints that simplifies development and encourages more scalable designs. Client applications interact with resources using four main HTTP verbs.

In the RavenDB architecture model, the data is communicated across the network in JSON format from the server to the client. The figure 2 shows the basics of RavenDB client/ server application architecture:

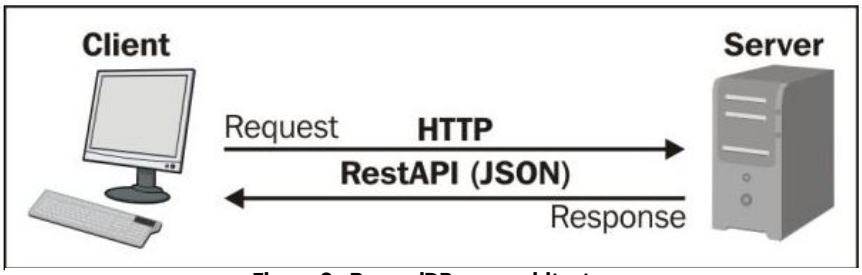


Figure 2 : RavendDB app architecture

2.4 JSON in RavenDB

JSON (JavaScript Object Notation) is a generic text format easy for humans to read and write and it's easy for machines to parse it and generate it. This format is used by RavenDB to store data and databases metadata. RavenDB can write the JSON document directly, simplifying the writing / updating process.

The JSON format is built on two structures:

- A collection of name/ value pairs. Programming languages support this data structure in different names like : object, record, struct, dictionary, hash table, keyed list, or associative array.

- An ordered list of values. In various programming languages, this is realized as an array, vector, list, or sequence.

The following screenshot illustrates a Document entity as it will be stored in RavenDB using JSON format:

```
{
  "ComputerID": "1234",
  "ComputerName": "KTAWIN8",
  "UserId": "963258",
  "Buying": "2010-03-06T20:35:08.8068415Z",
  "Inventory": [
    {
      "SoftwareName": "Adobe Reader",
      "SoftwareVersion": "10",
      "InstallDate": "2012-10-25T12:55:18.8058315Z"
    },
    {
      "SoftwareName": "Microsoft Office",
      "SoftwareVersion": "2010",
      "InstallDate": "2011-05-02T09:15:10.4068815Z"
    }
  ]
}
```

Figure 3 : Document format in JSON

Explanation of this JSON example : "*Describes the Computer entity, the object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name /*

value pairs are separated by , (comma). So, the "CompuerID" entity represents the key name and "12345" represents the value assigned to that key name.

The "Inventory" key name is an array and is an ordered collection of values. In JSON, an array begins with [(left bracket) and ends with] (right bracket). Values are separated by , (comma). In this example, the "Inventory" array contains two object instances where each one is composed of three name/ value pairs "SoftawareName", "SoftwareVersion", and "InstallDate".

In the website "http:// www.json.org", the JSON structure is well explained.

2.4 Download and Install RavenDB

Basically, the primary installation target of RavenDB is Microsoft Windows. The Microsoft's .NET Framework should also be installed on the computer where RavenDB will run and where the application client will run. The RavenDB package comes with different client versions.

We recommend to always use the latest .Net Framework so RavenDB can run with no problems. The RavenDB Management Studio is a Silverlight application and it needs Microsoft's Silverlight plugin to be installed on the web browser.

To download RavenDB, head to the download page on the RavenDB official website :

http://ravendb.net/download

You can choose between an Installer or a zip file. We recommend to choose the latest version available on the website, but as said in the beginning (about Licenses), in order to try some features we need an older RavenDB Server (build # 2230). The Management Studio is opened automatically when RavenDB server is launched using the "Start.cmd" file. To open it manually, you can type the next address into your browser :

http://localhost:8080

or

http://127.0.0.1:8080

By default RavenDB is listening on port 8080.

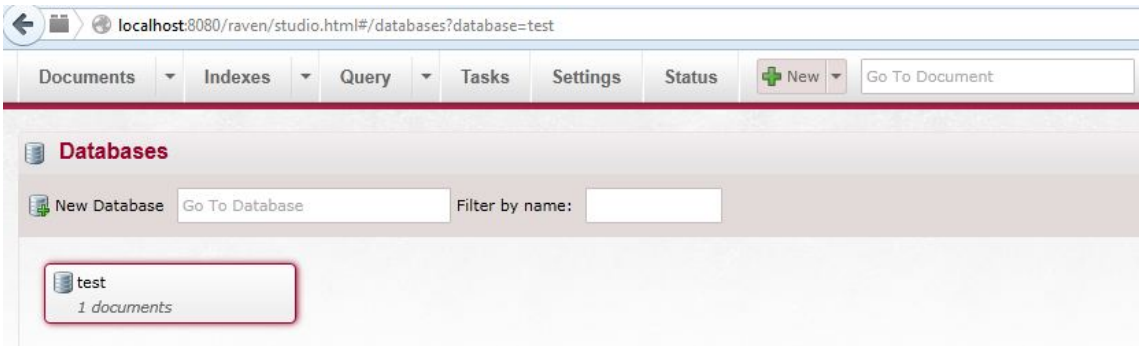


Figure 4 : RavenDB Management Studio

2.5 RavenDB Management Studio

The RavenDB Management Studio is a graphical integrated environment for accessing, configuring, managing, and administering the RavenDB server.

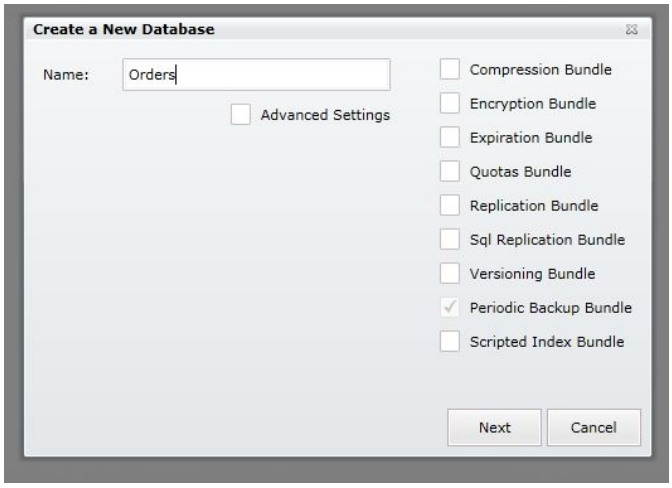
The RavenDB Management Studio is a Silverlight based application which is used to manage the RavenDB server. It is a very useful tool to look at the server databases and the activity logs to understand what's going on at the server-side. With the Management Studio, we can create **new databases** and **new documents**; we can **modify data directly** if we need to.

The main Management Studio user interface has a few different tabs on the top: Document, Indexes, Query, Tasks, Settings and Status (varies depending on the version used). We can use these tabs to access the different screens and manage different parts of the database server.

2.5.1 Creating a new Database

To begin using the Management Studio, we need to create a new database.

1. In the Management Studio, click on the Databases link.
2. Click on the New Database button to open the Create a New Database form.
3. Enter "Orders" for the Name of new database and click on the Next button to create the new database.

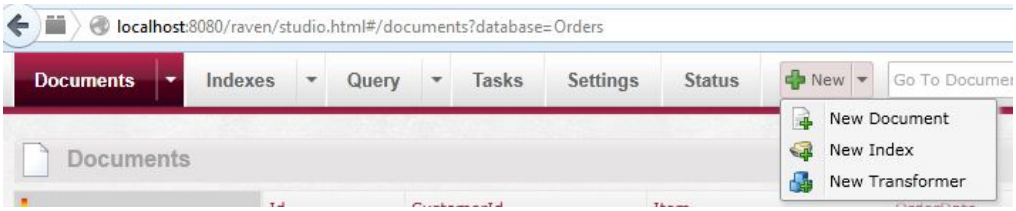


The database "Orders" will become the current database and will be added to the hosted databases list.

2.5.2 Creating a new Document

Follow these steps to create a new Document :

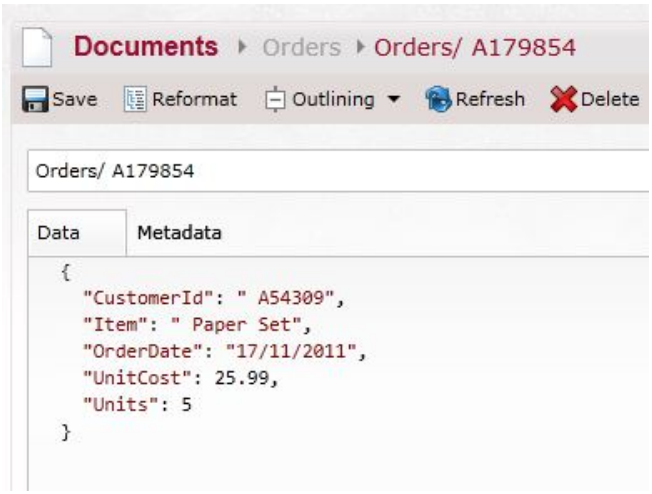
1. In the Management Studio toolbar, click on the arrow near the New button and select New Document.



2. Enter the document ID = Orders/A179854 and enter the following document data values:

```
{
        "CustomerId":" A54309",
        "Item":" Paper Set",
        "OrderDate":" 17/11/2011",
        "UnitCost": 25.99,
        "Units": 5
}
```

3. After that, click on the Save button to insert the document into the Orders database.



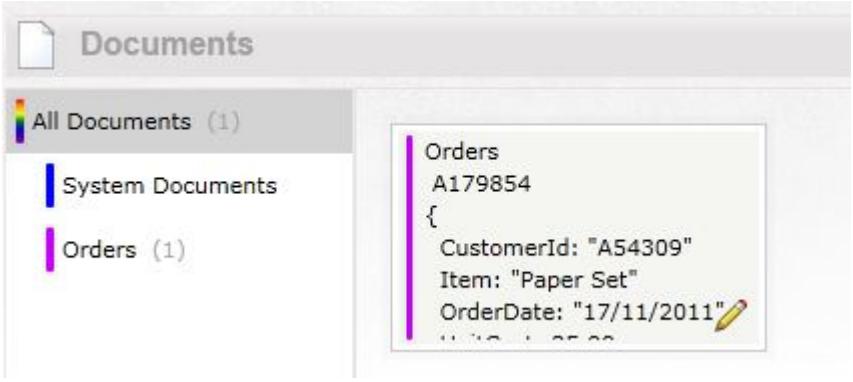
After clicking on the Next button, the Orders database is created. It will become the current database and will be added to the hosted databases list.

2.5.3 Modifying a Document

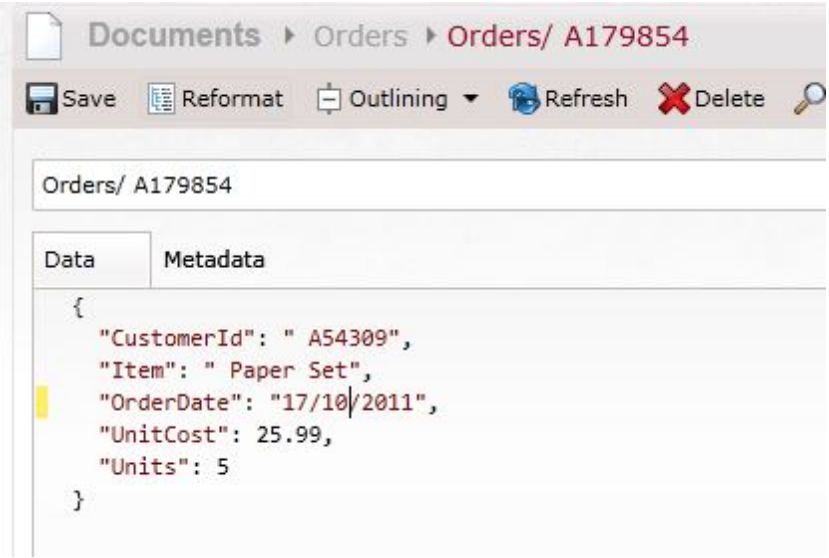
Under the Documents screen section[5], we see all documents related to the "Orders" collection which is the active selection in the Collections section.

[5] Consider that there are six different view modes we can switch between: Details, ID Only, Small Card, Medium Card, Large Card, and Extra Large Card.

1. Depending on the view mode, you can edit a document clicking on the pencil image or just double clicking in the document



2. You can edit the data displayed, once you start changing any value , you will see a yellow bar in the left of the row :



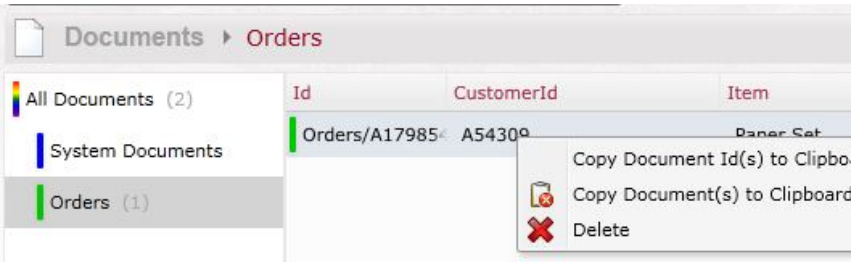
When modifying a document be sure that the JSON structure is still valid otherwise you will get an error which will be displayed in the *Errors* section

3. After that, click on the Save button to update the document.

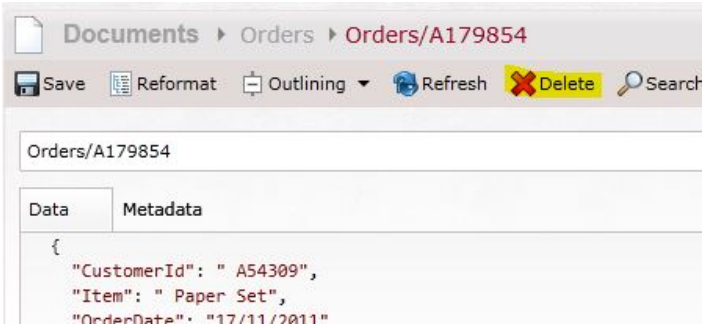
2.5.4 Deleting a Document

Depending on the view mode, you can delete a document :

- Under the Documents screen section, we see all documents related to the "Orders" collection which is the active selection in the Collections section. You right-click on the document row and select "Delete"



- Entering to the document full view, we can see the "Delete" button on document actions menu :



Now that we have the server running and we also know the CRUD operations with the RavenDB interface, we will introduce you some tips about relationships (RDBMS background) with RavenDB.

3 A pattern-based approach to modeling relationships in RavenDB

RavenDB does not directly support entity relationships. However, relationships exist in the real world and these patterns will help you follow best practices when modeling them.

- **One-to-one relationships :** In one-to-one relationships, the subordinate data should be embedded in the primary document. An example is as follows:

```
// person/ 1
{  "name": "Sam Smith",
   "address" : {
            "street": "1720 Center Blvd",
            "city": "Jacksonville",
            "state": "FL",
            "postal": "32256"
    }
}
```

- **One-to-many relationships** : In one-to-many relationships, a collection is embedded in the parent document representing many children. If items in the collection only make sense in the context of the parent, they should be embedded. If the items in the collections are aggregates, the collection should contain references. An example is as follows:

```
// manufacturers/ 1
{  "name": "Toyota",
   "models": [
            { "name":" Prius" },
            { "name":" Camry" },
            { "name":" Highlander} ]
 }
```

Be careful when embedding collections that may become large. When the collection reaches a certain size, you should either split the document or use the referencing technique. Here is an example of splitting the document:

```
// manufacturers/ 1
{
       "name": "Toyota",
}

// manufacturers/ 1/ models
{ "models": [
            { "name":" Prius" },
            { "name":" Camry" },
           { "name":" Highlander}
           ]
 }
```

By splitting the document, the models collection can be loaded only when needed, instead of loading every time the manufacture document is loaded.

- **Many-to-many relationships :** In many-to-many relationships, one document maintains a list of document identifiers for the related documents. An example is as follows:

```
// books/ 1
 {
     "title": "CLR via C#",
     "ISBN": "978-0735667457",
     "published": 2012
}

 // books/ 2
{
        "title": "Programming Patterns",
        "ISBN": "978-0735614222",
        "published": 2002
}

 // authors/ 1
{
         "name": "Jeffrey Richter",
        "books": [
                    { "id":" books/ 1", "title":" CLR via C#"},
```

```
                { "id":" books/ 2", "title":" Programming Patterns "}
            ]
}

// authors/ 2
{
    "name": "Maarten van de Bospoort",
    "books": [" id":" books/ 1", "title":" CLR via C#"}]
}
```

Note that the books collection not only contains the ID reference, but also the title. By adding this **denormalization**, the display of the author and related books can be optimized. When building an application, look for these opportunities to limit the need for multiple database reads.

3 RavenDB HTTP API

RavenDB HTTP API RavenDB allows interactions over the HTTP protocol and is exposing itself as a RESTful web service. This is very important because using programming languages that supports RESTful web services (C#, Java, PHP, Python, Ruby) we can access to the RavenDB server documents. To simplify the REST requests composition, we will use an HTTP tool to do some interaction with RavenDB through the RavenDB RESTful API.

We will describe and explain these two points :

- The RavenDB HTTP API
- Performing requests using the main REST verbs (GET, PUT, POST, PATCH, and DELETE)

3.1 RavenDB for Web apps

RavenDB fully supports an API based on HTTP that basically allows to interact with the database engine with simple HTTP requests. This means that we can have an application or environment that can talk HTTP, you can communicate through the RavenDB HTTP API. This HTTP API follows commonly understood RESTful principles and the interactions are entirely based around the HTTP protocol.

When using the HTTP protocol to access RESTful resources, the resource identifier is the URL of the resource and the standard operation to be performed on that resource is one of the HTTP methods such as **GET, PUT, DELETE, POST,** or **PATCH**.

RavenDB is exposing itself as the RESTful web service. Utilizing a RESTful HTTP API allows an application functionality to be consistently used across different platforms. It's possible to write a fully functioning RavenDB application just using JavaScript, HTML, and the HTTP API.

3.2 REST (REpresentational State Transfer)

It is a way of interacting with resources on the web via plain human-readable URLs. Any interaction of a RESTful API is an interaction with a resource (resource from RavenDB). In fact, the API can be considered simply as mapping and endpoint, or resource identifier (URL), to a resource. Resources are sources of information, typically documents or services. An HTTP-

based REST API makes communicating with the database easier, because so many modern environments are capable of talking HTTP. The simple structure of HTTP resources and methods is easy to understand and develop with.

The REST interface defines four commonly used HTTP main verbs: GET, POST, PUT, and DELETE. The GET verb tells the service that the client wishes to get a read-only representation of a resource. POST indicates the desire to create a new resource. PUT is typically used for modifying an existing resource also for resource creation. And DELETE indicates that a client wishes to delete a resource.

3.2.1 REST request URL Structure

REST requests are URL-based. In order to get or put any document in RavenDB, we will basically create a request and use a RavenDB target structure. Then we will specify what type of entity it is and the document ID or the data to be sent to the server within the RavenDB target structure. Then we will specify what type of entity it is and the document ID or the data to be sent to the server :

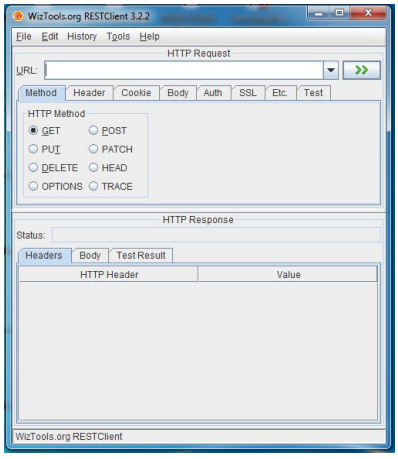
http://url:port/databases/databaseName/target/RequestData

- **url** : This represents the URL where RavenDB is running.
- **port** : This is the TCP port number, by default the port number is 8080.
- **databaseName** : This is the name of the database where the documents are stored.
- **target** : This represents the target structure in RavenDB we want to deal with. This might be the docs structure, the indexes structure, the queries structure, and so on.
- **RequestData** : This represents the data resources for the request. It might be the document ID on which the action is performed (create, retrieve, update, or delete) or all other data needed to perform the action.

3.2.2 RESTClient tool

To experiment and interact with the RavenDB HTTP API, we need to use a REST client tool to create HTTP requests and display HTTP responses ealy. You can download this tool from : http:// www.wiztools.org.

Be sure to download the GUI Executable Jar file.



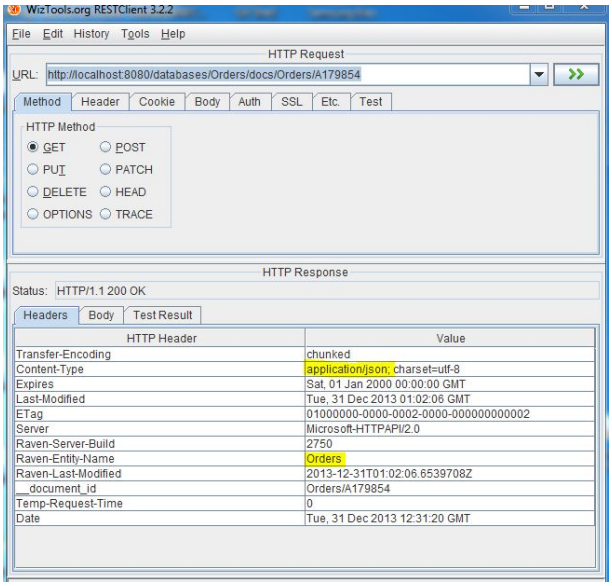
3.3 Describing GET, POST, PUT and DELETE requests

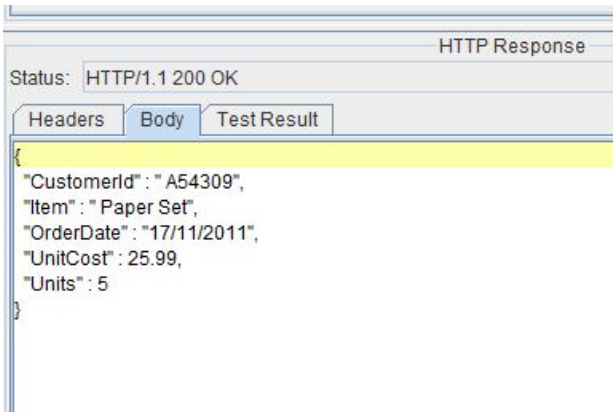
3.3.1 GET request

The GET verb is used for read-only operations. In order to create a Get Request, we can retrieve the document created before, with the ID = "Orders/A179854"

So according to the structure presented in the point 3.2.1, the URL must look like this :

http://localhost:8080/databases/Orders/docs/Orders/A179854





As you can see we didn't have any problem retrieving this document. Check that (left image), the content-type is json and the data retrieved belongs to "Orders" according to the Raven-Entity-Name value. The right image displays the json returned from the server.

3.3.2 PUT request

The PUT verb is an HTTP method that can be used to create a resource, with the information in the request, for the URL specified by the client. It is important to note that a PUT verb in RavenDB will always create the specified document at the requested URL. If the resource already exists, PUT simply replaces what existed with the new information.

In RESTClient, select PUT as HTTP Method, then URL must look like this :

http://localhost:8080/databases/Orders/docs/Orders/C676332

The body tab must be filled with the info of the document to create :

{

"CustomerId": "A55689",

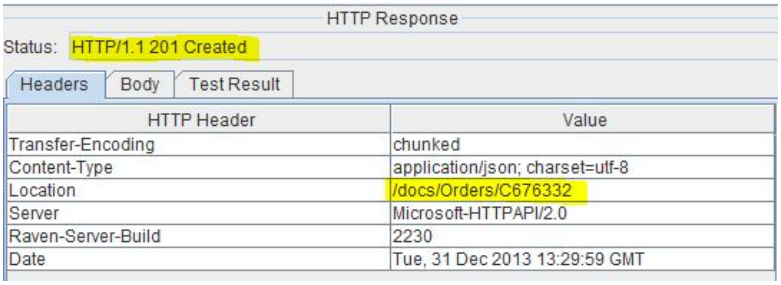
"Item": "Mouse Pad",

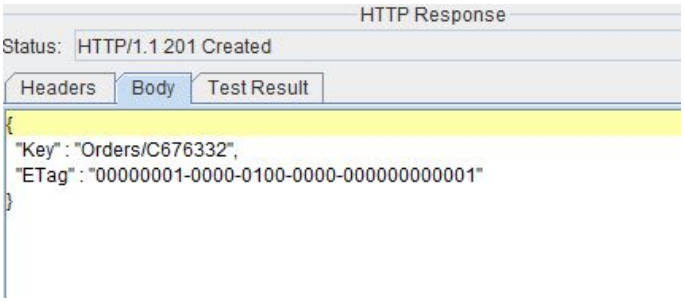
"OrderDate": "03/ 11/ 2011",

"UnitCost": 8.5, "Units": 10

}

After executing the PUT request the result is :





While testing the PUT request with the latest RavenDB server (build #2750), we had some issues with the License, because we will need a "Valid License" to perform this request from an anonymous user, in order to run it, we used an older RavenDB server (build # 2230), that lets us execute this request.

Performing a PUT request needs writing privileges on the server. By default, RavenDB grants a read-only access to anonymous users. We can change the "Raven/ AnonymousAccess", instead of "Admin", we must put "All" in the Raven.Server.exe.config file.

3.3.3 POST request

The POST verb can be used to either create a resource or update an existing resource. When we perform a POST request to the RavenDB's docs structure, it will create the specified document and allow RavenDB to assign a unique ID to it. We have to specify the document data in JSON format and will not specify the document ID.

The POST request URL is very similar to the PUT request URL with the difference that we do not specify the document ID in the URL.

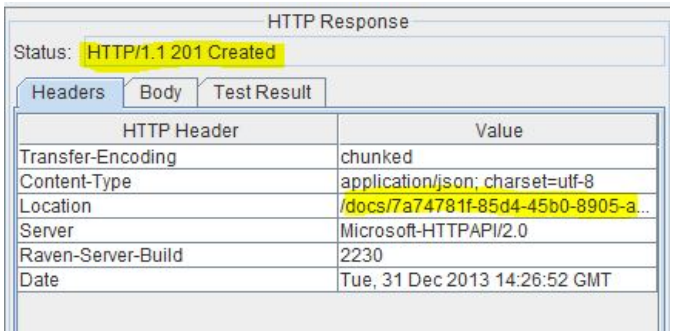
- Click on the Method tab and select the POST HTTP method. In the HTTP Request section, enter this URL:

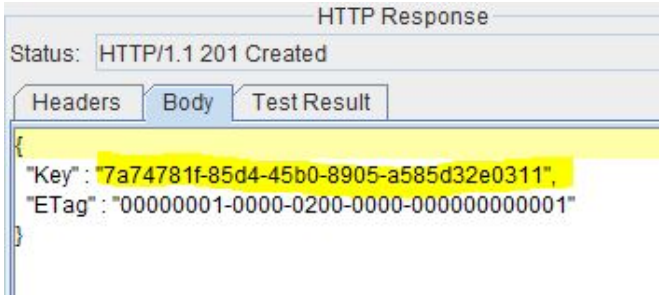
http://localhost:8080/databases/Orders/docs

- Click on the Body tab and select String body from the drop-down list and enter the following string:

```
{
        "CustomerId": "B66689",
       "Item": "USB Key",
       "OrderDate": "01/07/ 2012",
        "UnitCost": 28.5,
       "Units": 5
}
```

After executing the POST request the result is :





3.3.4 PATCH request

The PATCH request allows any single document to be updated without replacing the entire document as it is happening with the PUT request. The PATCH command accepts an array of commands, so it is possible to issue multiple modifications for the same document. The PATCH command keys are case sensitive and they have to be specified with the correct **Pascal Casing**.

The six different command keys which you need to specify are listed as follows:

1. **Type**: This represents the operation type. RavenDB supports the following patch operations:
   - **Set**: It is a property to a new value. (Optionally, creating the property).
   - **Inc**: Use this to increment a property value by a given value. (Optionally, creating the property).
   - **Unset**: Use this to remove a property.

- **Add**: Use this to add a value to an existing array.
- **Insert**: Use this to insert a value to an existing array at the specified position.
- **Remove**: Use this to remove a value from an existing array at a specified position.
- **Modify**: Use this to apply nested patch operation to an existing property value.
- **Copy**: Use this to copy a property.
- **Rename**: Use this to rename a property.

2. **Name**: This is used to identify a property by its name.
3. **Position**: This is used to specify the position (index) in an array field.
4. **Value**: This represents the new value for a property.
5. **PrevVal**: This is the old property value.

Now with these info, we can perform a PATCH request :

- Click on the Method tab and select the PATCH HTTP method.
- In the HTTP Request section, enter this URL :

http://localhost:8080/databases/orders/docs/Orders/C676332

- Click on the Header tab and ensure that the Raven-Entity-Name metadata key is sent to the RavenDB server with the Value = Orders.
- Add this string in the body tab :

```
[
        { Type: 'Set', Name: 'Colour', Value: "Black"},
       { Type: 'Inc', Name: 'discount', Value: 2},
]
```

After executing the PATCH request the result is :

3.3.5 Delete request

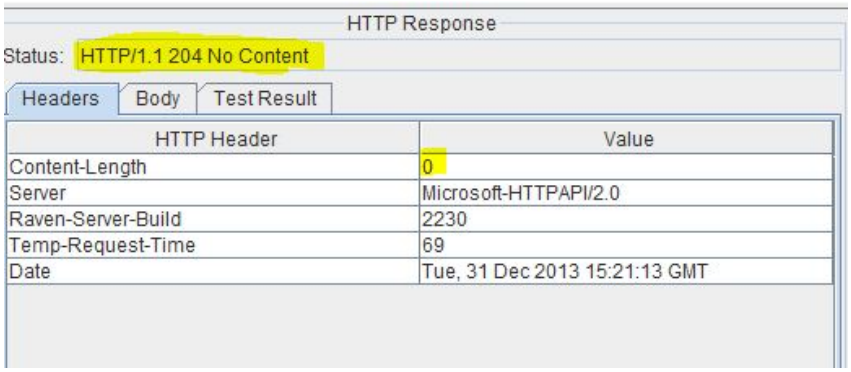
The DELETE verb asks the server to delete the resources. This will instruct RavenDB to delete the JSON document specified by the URL.

Follow these steps to perform a DELETE request for the document ID = **Orders/C676332** :

- Click on the Method tab and select the DELETE HTTP method.
- In the HTTP Request section, enter this URL :

http://localhost:8080/databases/orders/docs/Orders/C676332

After executing the DELETE request, the result is :



Bibliography

- Gaurav V. (2013) *Getting Started with NoSQL by Gaurav Vaish*, UK : Packt Publishing Ltd.
- Pramod J. S. & M. Fowler (2012) *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence* , US - Person Education, Inc.
- Ritchie, Brian (2013) *RavenDB High Performance*, UK : Packt Publishing
- Tannir, Khaled (2013) *RavenDB 2.x beginner's guide*, UK : Packt Publishing

- RavenDB Official Website - *http://www.ravendb.net/*
- NoSQL references - retrieved from *http://www.nosql-database.org/*
- Forum NoSQL - *https://groups.google.com/forum/#!forum/nosql-discussion*
- NoSQL references - *http://en.wikipedia.org/wiki/NoSQL*
- JSON structure - *http:// www.json.org*
- REST concepts - *http://en.wikipedia.org/wiki/Representational_state_transfer*
- RESTClient tool - *http:// www.wiztools.org/*