

RavenDB IN ACTION

Itamar Syn-Hershko

FREE

MANNING

www.dbooks.org



RavenDB in Action

Copyright 2017 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

For online information and ordering of these and other Manning books, please visit http://manning.com/
The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

No part of this publication may be reproduced , stored in retrieval system, or transmitted, in any form or by any means electronic, mechanical, photocopying, or otherwise, without prior permission of the publisher.

Many of designations used by manufacturer and sellers to distinguish their products are claimed as trademarks- Where those designations appear in the book. and Manning Publications was aware of trademark claim, the designations have been printed in initial caps or all caps.

Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish on acid free-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is least 15 percent recycled and processed without the use of elemental chlorine.



Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Cover designer: Marija Tudor

ISBN: 9781617294600
Printed in the United states of America
1 2 3 4 5 6 7 8 9 10 - EBM - 21 20 19 18 17 16

www.dbooks.org

foreword

Thank you for downloading *RavenDB in Action*. While we will not be releasing *RavenDB in Action* as a finished book, we're pleased to offer the complete manuscript to you! It's still a great way to learn RavenDB; with Itamar's skills and wealth of experience really shining through; it just lacks our usual finishing touches!

This book is aimed mainly for C# developers who are tired of the awful mess that comes with database work, and are looking for a better, seamless way to persist data. While it is designed to teach RavenDB from scratch even to complete newbies, we're sure that experienced developers can benefit from the book as well, by incorporating lessons learned from user stories and RavenDB workshops and consultations.

We really hope you find this book useful. Thanks once again, and enjoy.

brief contents

PART 1: HELLO RAVENDB

1 *A 2nd generation document database*

2 *Your first RavenDB application*

PART 2: UNDERSTANDING RAVENDB

3 *RavenDB Indexes*

4 *Working with indexes*

5 *Document-oriented Data Modeling*

PART 3: ADVANCED RAVENDB

6 *Full-text, geo-spatial and reporting queries*

7 *Scalling out*

8 *Extending RavenDB*

9 *Advanced capabilities*

10 *Getting ready for production*

www.dbooks.org

1

A second-generation document database

This chapter covers

- Some history of database technologies
- What is RavenDB?
- Getting started with RavenDB

Data persistence is at the heart of every technology project. Whether it's to remember users and their preferences, gather statistics and numbers for later crunching, sell products out of a catalog, save logs, or count Facebook "likes," data has to be stored. Although data can be safely stored even on tapes, most of the time we need it accessible for quick retrieval to display it or for further processing, and to do that we need the ability to query.

For several decades relational databases, or SQL databases, have been used to store data in a retrievable way, and they still are. They are reliable and quite easy to get data out of when needed, and that's how they became the industry standard for data persistence.

But they aren't the perfect solution. As the world progresses, the amounts of data that have to be dealt with grow and can reach massive scales. That's when the very design of relational databases breaks, and we are left with a choice: we can either spend a lot of effort coming up with workarounds, or we can look for alternative solutions.

RavenDB is an evolving database technology written in .NET, a non-relational one that aims at providing a reliable storage engine that scales well and fits the shape of data used in modern applications. It's also a very opinionated piece of software that will do quite a lot to help you code better and faster and prevent you from shooting yourself in the foot.

There are several alternatives to relational databases, but RavenDB has some unique design decisions built into it. Those design decisions are about maximizing productivity and

efficiency of data retrieval, and they are the reason why RavenDB prides itself in being a second-generation document database. You're going to learn about some of them later in the chapter, and the rest of them you'll get to know throughout the book.

Although the main RavenDB API is actually a REST API over HTTP, this book focuses on usage through the .NET Client API, which ships with the server itself. You'll write a lot of C# code and not see much of HTTP, even though there's a lot of it going on under the hood. The samples in this book were written with Visual Studio 2012 and .NET 4.0. This book assumes C# and LINQ knowledge and makes heavy use of C# Lambda expressions. For your convenience, Appendix A describes the REST API and gives an example of how to use it.

We start this chapter with a short overview of the current state of the database world and where RavenDB fits in it. Shortly after that, and after introducing a few important core concepts, we dive right in and show you how to start writing your first RavenDB-backed software.

1.1 The database chronicles

Even today, 40 years after they were invented, relational databases are still the first choice for data persistence in many new projects. It is the safest bet any project manager can make: it is highly tested software with great tooling, all developers know how to work with it, and no work is required to convince management that it can be trusted. Everybody wins.

Or do they?

As you're about to see, the very design of relational databases makes them unsuitable for many modern applications, where non-relational database alternatives are going to do a much better job.

1.1.1 Relational databases and the modern world

Take the following famous website for example. What would it take to build it on top of SQL Server?

www.dbooks.org

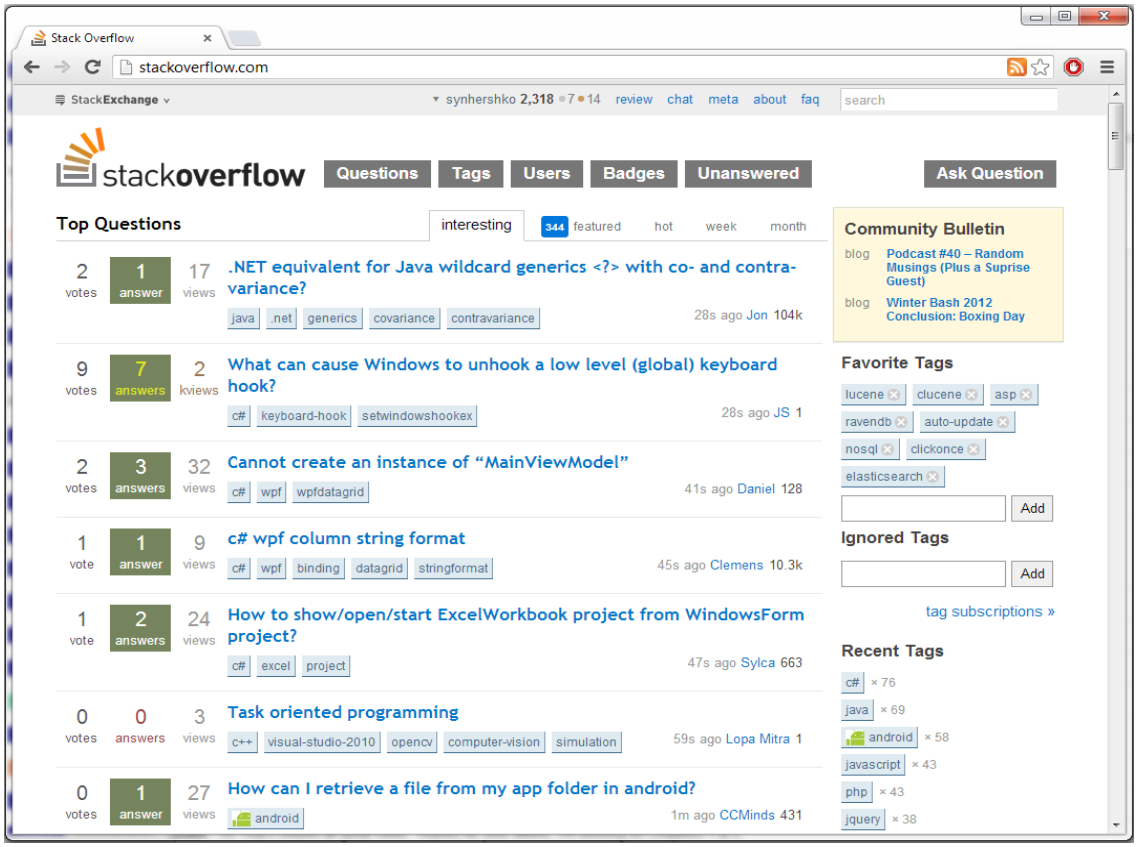


Figure 1.1: The homepage of the famous StackOverflow website. It's known to be built on SQL Server—can you guesstimate the number of tables and queries involved in generating this page?

A simple relational model for the StackOverflow website would require a database table for each of the displayed entities—Questions, Answers, Tags, and Users. Because users can comment on questions and answers alike, at least one additional table for storing them is required as well. Now think about the amount of work required to generate a single page—the homepage or a question page. How many queries and joins are involved?

As this example shows, modern applications usually deal with complex object graphs, which are not easy to store in a relational model and require quite a bit of work to get them in and out of relational storage. This is commonly referred to as the *object/relational impedance mismatch*, and it's one thing that makes relational databases not a good fit for many modern applications.

In an effort to ease the mismatch, software tools like NHibernate and Entity Framework were created, referred to as *object/relational mappers*. But in the process of simplifying, they

hide even bigger problems under the rug—like the infamous SELECT N+1 problem where what looks like one simple query in code translates in reality to many. Ted Neward put it beautifully:

> ORM is the Vietnam of Computer Science. It represents a quagmire which starts well, gets more complicated as time passes, and before long entraps its users in a commitment that has no clear demarcation point, no clear win conditions, and no clear exit strategy.

Not only do relational databases require you to spend a lot of time conforming to database concerns instead of putting all that time into building software that solves business problems, but they're also limited in handling large scales of data, and this really was the straw that broke the camel's back, as you'll now see.

Throwing more hardware on your single DB server to accommodate more data, also known as *scaling up*, has its limits, from both the perspective of available hardware and the amount of traffic it can handle. When you have too much data for one machine and are spreading it across multiple nodes, things get really complicated.

Any system that's spread across more than one machine—also known as a distributed system—is going to experience network issues or have nodes failing every now and then. A mathematically proven theorem called the *CAP theorem* states that in those scenarios where partition tolerance is desired, only one of consistency or availability can be guaranteed across the system. In other words, if you want your system to be able to operate when nodes go down (and that's bound to happen at some point), you get to choose whether you want the data in your nodes to be consistent across all nodes all the time or to guarantee availability so no request is ever left hanging. It's been mathematically proven that you can't have it both ways.

In order to maintain relations, primary keys, foreign keys, and the whole relational story, a relational database system must be consistent at all times. Since most distributed environments would want to have both high availability and partition tolerance, you are left with no option but to relax the consistency requirement. This in turn means stop being relational.

SQL databases were designed to work as a single-node system and never in a cluster of more than one node. While you can certainly continue working with relational databases spread across multiple nodes, it becomes quite inconvenient. You are no longer treating all nodes as one database, but rather each node holds one piece of the data. This setup is called *sharding*, and most benefits of the relational model aren't available to you. For example, cross-node joins are not supported. This is the where non-relational databases become popular.

1.1.2 NoSQL

NoSQL—Not-SQL or Not-only-SQL—is a nickname for a family of non-relational databases. Different NoSQL solutions have different design decisions, usually around the CAP theorem we just mentioned. Most NoSQL databases relax the requirement for consistency to guarantee

www.dbooks.org

both high availability and partition tolerance, and almost all of them don't require you to define a schema upfront.

There are many NoSQL databases available today, and many of them are listed at http://nosql-database.org/. Their schema-less nature and ability to scale much easier than relational databases are what makes them very popular. Interestingly enough, the majority of them are also open source.

Although there are many NoSQL databases, they can be roughly broken into three categories: graph databases, key/value stores, and document databases.

GRAPH DATABASES

Graph databases like Neo4j persist graphs—nodes and arcs—and provide a rich set of functionalities around querying on them. In graph databases the relationships between records are as important as the records themselves, and as such querying usually involves those as well.

Graph databases are useful for things like social networks, recommendation engines, and many optimization algorithms. Every type of connected data that can be thought of in terms of graphs can naturally fit within a graph database.

KEY/VALUE STORES

A key/value store is a database engine which is very efficient at storing arbitrary data under a unique key. Because only one piece of data can be stored under one key, it's easy to scale out with because it's easy to decide on a simple rule that will determine which node should have which key and the other way around. It's essentially a hash map that's persisted to disk and is scalable to multiple nodes in a cluster of computers.

A database of this type has no knowledge whatsoever about the data that was put to it. It can be a number, a complex document, or a binary blob. The database doesn't know or care.

With a standard key/value store the only way to get data out of it is by its ID. You can't search for data in any other way, because the key/value store engine doesn't have any knowledge of it. More sophisticated key/value stores allow having additional metadata for values.

Other key/value stores do assume something about the value stored in them. For example, column-family databases like Apache Cassandra and Amazon SimpleDB are a variation of a key/value store where the value is known to be a list of rows.

DOCUMENT DATABASES

Document databases are an evolved version of a key/value store. At its core a document database is a standard key/value store, but the data that goes into store has to be a document, that is, of a specific format the database is familiar with. Documents are usually represented in a string, in human-readable form, and can contain hierarchical structure.

The reason for this requirement is fundamental to the most important feature of document databases: their ability to perform additional processing on the data stored to them and by that allow documents to also be searchable on their data and not only their ID.

RavenDB is a document database, and it has some unique design characteristics of its own.

1.2 Introducing RavenDB

RavenDB is a document database, made for persisting and retrieving documents. As with all document-oriented databases, *documents* are identified by a unique key and can be searched on their attributes as well.

In this section we're going to look at the benefits of RavenDB and basic concepts required for working with it. At the end of this section there's a quick LINQ primer for those who don't feel comfortable enough with LINQ yet.

The most important fact to remember is RavenDB is not a silver bullet. Its origins are from the understanding that one size does not fit all, as many developers thought relational databases to be. RavenDB will shine when used as a document database, using the right concepts and methodologies. But it can do a pretty bad job when used for things it wasn't meant to do, for example, when trying to enforce relational methodologies on it.

1.2.1 Benefits of RavenDB

What makes RavenDB so attractive is several design decisions that were made early on in the development process. The result of those design decisions is RavenDB's unique way of working, and it's those design decisions that RavenDB prides itself on in being a second-generation document database.

Apart from being extensible all around and very easy to scale out in complex systems, the following are the important benefits you will notice rather quickly when starting to work with RavenDB.

NO SCHEMA

One important benefit of RavenDB being a database that works with documents is that it has no schema to maintain, resulting in faster and easier development, and software that's easier to maintain.

Being schema-free also means reads are a lot faster than relational databases, because usually most read operations concentrate on one document. Reading one document is tremendously faster than doing a relational join over two or more tables. We'll demonstrate this when we discuss document-oriented modeling in chapter 5.

Having no schema doesn't mean the querying abilities are limited. In chapters 3 and 4 when we discuss RavenDB's indexes, we will show how flexible they can be and how that empowers the querying abilities. Then in chapter 6 we will look at some of the advanced querying capabilities that RavenDB offers.

READS ARE ALWAYS FAST

Most modern applications deal with a lot of data or a lot of users, or both, but are usually read-heavy. It is less common to find a website that deals with more writes to its database than it needs in relation to server reads; the ratio between reads and writes is usually hundreds of reads per write, if not more than that.

This fact is at the heart of several design decisions behind RavenDB, which promises that reads will always be fast. When you need data, you want to have it fast and to never have to

www.dbooks.org

wait for it. To this end RavenDB has several caching mechanisms built in, and it's also what guided the design of its indexing process, as you'll see in chapter 3.

AUTO-TUNING

The database itself knows best about usage patterns of the applications talking to it, and RavenDB makes use of this to self-optimize and to minimize the administration operations required in normal operation.

Among other things, RavenDB creates and manages indexes automatically and performs operations in bulk, with varying bulk sizes depending on usage. All of this is done in order to make the database more responsive and to require considerably less maintenance.

DIFFERENT DEPLOYMENT STORIES

RavenDB has different deployment stories. The most common usage is to run it in a client/server architecture, where the RavenDB server is listening on the network and responding to incoming requests from one or more clients. It can also be embedded in any .NET application, making it usable on disconnected or occasionally connected devices.

RavenDB can also be used from everything. The RavenDB server exposes a REST API over HTTP, which means you can do pretty much everything with it using a browser or tools for generating HTTP requests like Fiddler. This makes RavenDB accessible for practically every language on every environment.

For .NET developers, RavenDB ships with a well-thought-out .NET Client API with a first-citizen LINQ integration, both of which hide a lot of complexity and make working with RavenDB from any .NET application a breeze.

SAFE BY DEFAULT

RavenDB prides itself in being safe by default, meaning that while keeping your data safe, it also protects from misuse. Seeing how easy it was to misuse relational databases and ORMs, the developers put in place mechanisms to prevent similar cases of misuse for RavenDB.

There are several important safe-by-default features to note and remember:

- Your data is always safe.

  RavenDB is fully transactional, and once you get a confirmation that your data has been put into the database, it's there to stay. You're guaranteed that no power outage or network failure will cause data loss. This also holds true for distributed environments, as you'll see in later chapters when we discuss database replication.

- Limiting the number of database calls that can be made in a single session.

  Being too chatty with the server is dangerous. Many applications died because of SELECT N+1 issues, where you asked the database for a set of rows of data and then another row or more for each of the rows in the result set. In essence this made one simple-looking query very expensive to perform. The RavenDB client API will perform auto-batching automatically where possible and throw an exception if you make too many calls to the RavenDB server. By default the amount of allowed calls is 30 per session.

- It protects against unbounded results sets.

  By default any query to RavenDB will return only the top 128 results, and the server will actively refuse to send more than 1024 results even when asked to do so. Sending back unbounded results sets and processing them is another application killer, and RavenDB actively fights against that. If you need more than the default amount of results provided, you should page through the results. Never again do a SELECT * type of query.

These safeguards against session abuse and database misuse are extremely helpful in making sure any RavenDB-backed application is built right from the ground up. You may find it too opinionated and bothersome at times, but eventually you'll see how it forces you to write better code and approach problems with better solutions.

1.2.2 Basic concepts

Before we start working with RavenDB, there are several key concepts you need to grasp. The first concept we're going to look at is a document—what it is, what it looks like, and how it's being used from the .NET Client API. Then we're going to explore how we refer to groups of similar documents and how automatic key generation works for documents.

A DOCUMENT

Being a document database, RavenDB deals with documents. A document in RavenDB is any data serialized as a JSON-formatted string and is the most basic piece of data RavenDB recognizes and can work on.

In case you're not yet familiar with it, JSON—shorthand for JavaScript Object Notation—is a convenient way of representing data in a human-readable string. It's pretty much like XML but without all the noise of tags and attributes. You'll see an example of a JSON document in just a second.

Using the .NET Client API, any class in your application can be serialized into JSON and stored as a RavenDB document. Let's take the class in figure 1.2, for example, and let's assume we created an instance of this class with some data. Once saved to RavenDB using the .NET Client API, it will look like what's shown in the right side of the figure.

www.dbooks.org

.NET class

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }
    public double Price { get; set; }
    public int YearPublished { get; set; }
    public bool InStock { get; set; }
}
```

JSON representation

```
{
  "Title": "Some book",
  "Author": "John Doe",
  "Price": 25.99,
  "YearPublished": 2012,
  "InStock": true
}
```

Figure 1.2 A .NET class and a JSON representation of an object of this class. This JSON document is exactly how a class of this object would look like once stored into RavenDB.

Any .NET class—often referred to as a Plain Old CLR Object (POCO)—can be serialized into JSON and stored into RavenDB. The .NET Client API does that for you, as you'll see in the next section. The class was serialized as is without any help from our side—no special attributes are required nor a schema definition of some sort.

Any type of data is supported, ranging from native types, such as strings and numeric types, to dates and objects of other classes. And more importantly, unlike what you may be used to from previous experience you had with OR/Ms, nothing happens beyond the serialization to JSON. The complete object graph is saved into RavenDB as one JSON document. Each document has a unique identifier, so it can be uniquely identified and pulled from the store.

A COLLECTION

Because documents in RavenDB are not grouped in any way—they're all stored in the same storage space with nothing really to tell them apart—there needs to be some way of grouping similar documents together and talking about groups of them.

Think of it as going to a carpenter shop and looking at all the wood products made there. Some will look like tables, some like chairs, and other like closets. Instead of speaking of wood products, you'd be interested in speaking about all chairs, all tables, or all closets. Because they look different, the distinction is easily done, even if all tables look completely different from one another.

This is exactly the case with documents in RavenDB. Documents holding user data are different from documents holding product information, and we want to talk about groups of them, or collections of documents.

To that end, RavenDB allows for a document to be stamped with a string value that will be evidence of its type. We usually prefer that value to be as descriptive as possible, like "Users" and "Products."

When using the .NET Client API this is done automatically for you. When saving an object to RavenDB, the type stamp that will be saved along with the document is inferred from the object class name. The convention is to use the plural form of the class name, so a User object

will be serialized into a JSON document and then stamped with the “users” entity type, and a Product object will have a “products” type stamp on it.

This in turn helps RavenDB to decide what collections are in the database—by looking up all the unique type stamps in the database. A *collection* in RavenDB is just a way to talk about all documents that share the same type stamp, just like you’d refer to wood products looking like chairs as “chairs.” The Products collection, for instance, is essentially a request from the database to show all the documents with the “products” type stamp. The number of *collections* in a RavenDB instance is equal to the number of unique values of those type stamps.

It is important to remember that a *collection* in RavenDB is just a logical way of talking about all documents that share the same type. It is a completely virtual concept and completely different from a SQL table.

UNIQUE IDENTIFIERS

In its heart RavenDB is just a key/value store that handles documents. A document is stored with a unique key to identify it, which is also used to pull it back from storage later. In RavenDB those keys, also called document IDs, are simple strings. Any string can be used as a document key, and that string is unique across all documents in the system, no matter what collection they are part of (remember: a *collection* is purely a logical concept!).

This also means, and this bit is important, that whenever you store a new document using a key that already points to a document, that document will get overridden with the new document you’re storing. Think of it like a path to a file—if you save a new file to an existing path, the previous file will be overridden by the new file.

RavenDB uses a default convention for document IDs, where the collection name is leading and following it is a unique value for the document within that collection, as shown in figure 1.3.

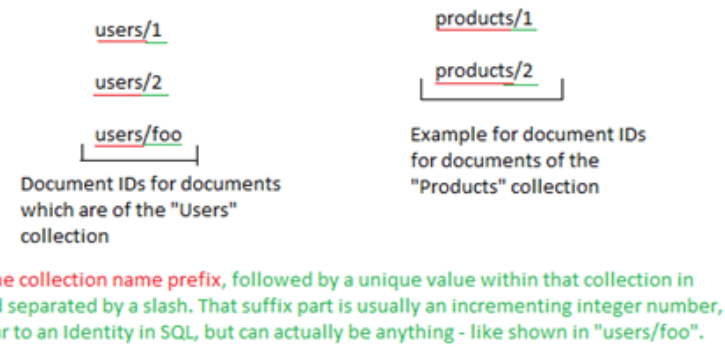


Figure 1.3 The RavenDB default convention for document IDs

Using this convention, writing a document with ID users/1 will override the document from the Users collection, and writing a document with ID products/1 will override the document from

www.dbooks.org

the Products collection. Writing a document with an ID of just 1 will create a new document with a string ID value of 1—or overwrite it if it existed before.

With this convention, document IDs are human readable and also ensure that no documents are overwritten within the same *collection* or in the general datastore. But this is merely a convention, and obviously you can use whatever string you want as an ID for your document.

RavenDB relieves you from the duty of keeping track of unique IDs within the same *collection*. By using the .NET Client API, whenever you ask to store a new object, RavenDB will assign it a unique ID by generating the prefix and will negotiate a unique untaken integer ID within that *collection*. This is done efficiently by using a variation of the HiLo algorithm, which you can read more about by Googling if you want.

As a general rule, you should let RavenDB manage IDs for you, because it's done efficiently and the default convention is also helpful with document-oriented modeling, which you'll learn more about later in the book. You can always change the defaults or assign IDs on your own, though; just make sure not to overwrite documents accidently.

1.2.3 LINQ

RavenDB uses LINQ extensively; therefore this book relies on you being comfortable using LINQ as well. For your convenience, following is a quick primer to LINQ. By no means a complete or a thorough guide, just something to get you started if you are new to this wonderful query language. If you need a refresher a good resource is the book *LINQ in Action* by Manning or "101 LINQ Samples" at http://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b. If you've already used LINQ a few times, you can safely skip this.

What is LINQ? LINQ is a collection of extension methods that act on lists, where a list is any .NET data structure implementing the IEnumerable interface. These methods filter lists, sort lists, and in many other ways help manipulate the individual items in a list.

For example, if you had a list of strings and wanted to keep only elements longer than five characters, you could do it with the following simple LINQ statement:

```
List<String> list = new { "foo", "bar", "test", "long string" };
var result = list.Where(x => x.Length > 5).ToList();
```

And this would leave you with only one item (`"long string"`) in the result list.

Because LINQ is just a set of extension methods, any list can be "LINQed." Furthermore, not only can the extension methods be overloaded to work on local collections such as lists and arrays, but they can also act on an endless number of more conceptual collections. So anyone can create a LINQ provider and allow developers to interact with their collections through this common set of methods. This is exactly what the RavenDB client API does: it implements the extension methods in a way that make querying RavenDB using LINQ possible.

Although LINQ is just a set of extension methods for IEnumerable, there is also the LINQ query syntax, which is just a syntactic sugar on top of the basic LINQ, which is also known as the method syntax. If you've worked with SQL before, that syntax might feel a little familiar.

But everything you write in LINQ can either be written using this LINQ query syntax or written using extension methods. The two syntaxes are completely identical—in the end it all boils down to your personal preferences.

These are equivalent statements that filter a given list of states for the ones that start with the letter *a*:

```
var States = new [] { new{ Name= "Alabama"}, new { Name= "Florida"}, new {
Name= "California" }}; #A

var list1 = from s in States where s.Name.StartsWith("A") select s; #B

var list2 = States.Where(s => s.Name.StartsWith("A")); #C
```

#A – The list of State objects to filter
#B - LINQ query syntax
#C – LINQ extension method syntax

This ends our quick LINQ primer. If you feel you need more practice with LINQ, check out the resources I mentioned at the beginning of the primer. It's important to be comfortable with LINQ before starting to work with RavenDB.

Starting with the next section we'll dive right in into working with RavenDB. After a quick installation and a tour of the server package we'll start the real action by writing some code to talk to the RavenDB server from a simple .NET console application.

1.3 Installing RavenDB

So far you've seen some theory on NoSQL in general and dove into specific bits on document databases and RavenDB. You came to know basic concepts in RavenDB, and now you're ready to get a first taste of what it's like to build an actual application with RavenDB.

In this section you'll get to know the building blocks every RavenDB-based application has and how to use them. By the end of this chapter you should feel comfortable with the basics, and in the next chapter you'll build a full web application from the ground up, demonstrating this knowledge and learning new things along the way.

To start working with RavenDB, you first need to download it. You can obtain a build package from the RavenDB website at http://ravendb.net/download. For the samples in this book we used version 3XXX, but any version from the 2.5 line of RavenDB will work just the same.

In most books this is where they tell you to flip to an appendix where the rigorous installation process is explained, so the flow of reading isn't disturbed. But because in RavenDB there's hardly any setup work involved, we're going to leave this part here.

1.3.1 Running the server

After the download is finished, you will have a zip file of the RavenDB distribution, as shown in figure 1.4. The distribution package contains the server binaries, various clients (the standard

www.dbooks.org

.NET 4 client, the Embedded client version, and a client for Silverlight), and also binaries for various bundles and tools.

To start, copy the /Server folder from the zip file you downloaded into some folder on your hard drive.

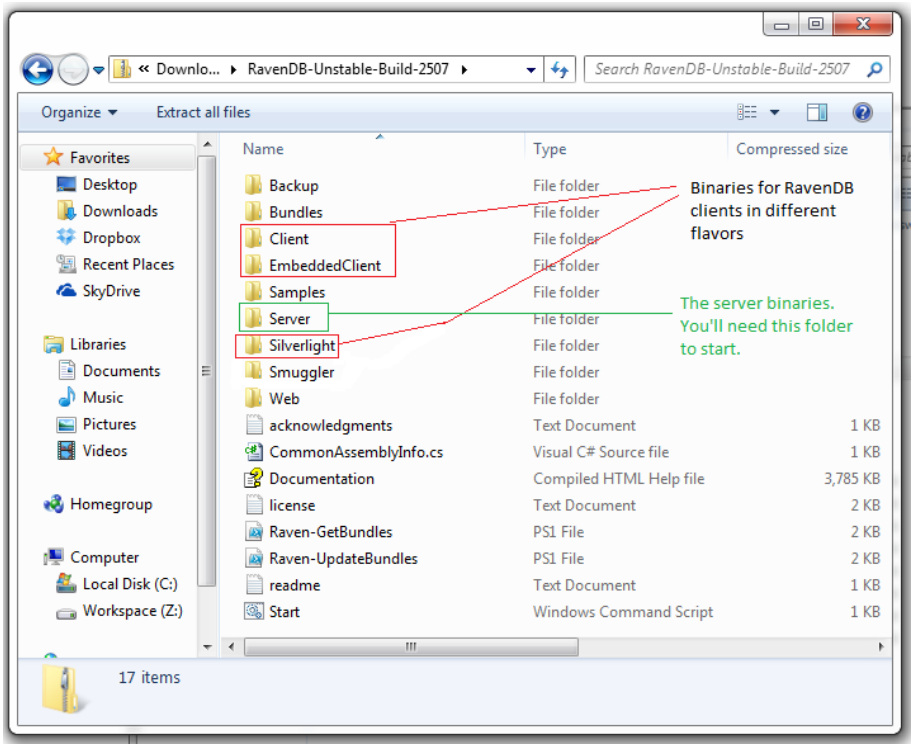


Figure 1.4 The RavenDB distribution package

Once you've done that, run Raven.Server.exe. This will start the RavenDB server in a debug console mode. The console will initially show basic information on the environment it runs in, as shown in figure 1.5, and as new requests come in, it will log them to the window.

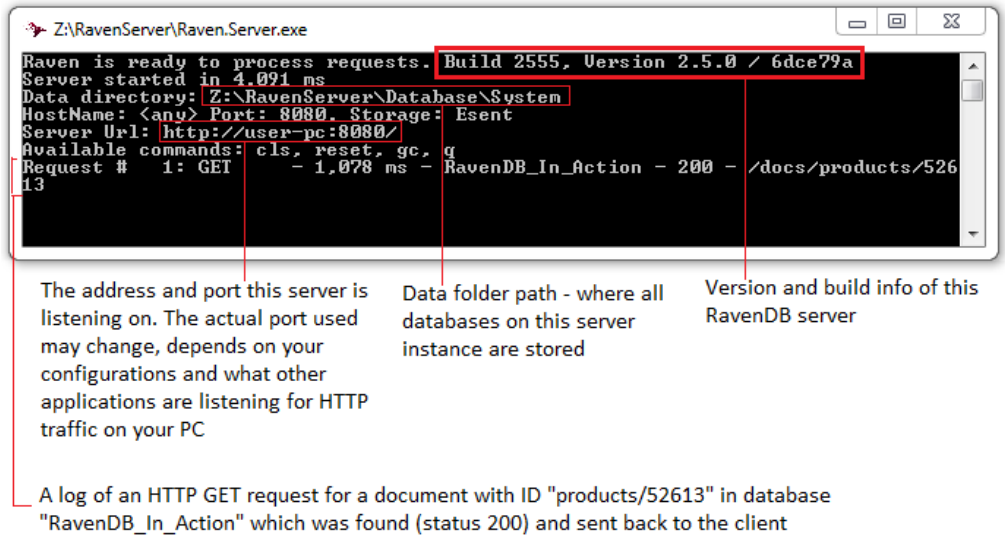


Figure 1.5 The RavenDB server running in debug console mode

This is all you have to do to have RavenDB up and running—there are no installations or pre-configurations required. The RavenDB server is now listening on HTTP port 8080, as indicated by the fourth and fifth lines in the console window. The actual port you see may vary, depending on the other applications that you have running locally.

As expected, if you shut the server down and bring it up later, all data will be persisted and won't go away. This is why we have databases, after all.

But during development or learning you might want to be able to wipe your database clean in an instant. Fortunately, RavenDB allows you to run it in memory, meaning the data storage becomes temporary and gets erased whenever you restart the server or type `reset` in the console.

To do this, you need to tell RavenDB to run in memory. This can be easily done by passing a `/ram` parameter to RavenDB.Server.exe. When run this way, RavenDB will report that it holds the data in memory instead of on disk.

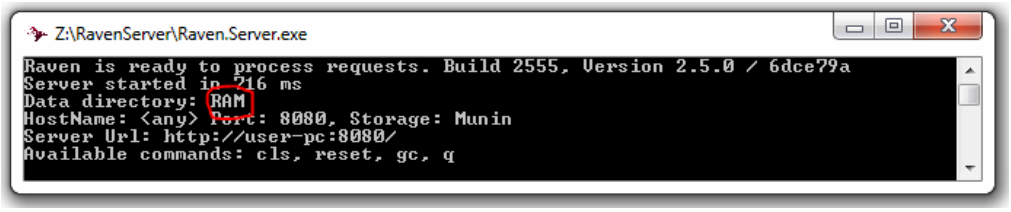


Figure 1.6 Running `Raven.Server.exe /ram` will make the RavenDB server run entirely in memory, without persisting anything to disk.

www.dbooks.org

This is a handy feature, and I strongly encourage you to make use of it as much as possible during your learning experience with RavenDB. You may find it so useful that you will continue using it even later, for quickly prototyping real-world RavenDB applications.

1.3.2 The Management Studio

To easily view and manage the documents, collections, and indexes in a RavenDB instance, you can access the RavenDB Management Studio bundled with the server. The Management Studio is a Silverlight application served by the RavenDB server instance when accessed using a browser. Let's take a short tour.

To start, hit the URL RavenDB is listening on using your favorite browser—it's the URL that appears in the debug console and usually is http://localhost:8080/. You'll be served with the Management Studio, which is an in-browser Silverlight application. If you don't have Silverlight installed, you will be asked to install it.

You now have a RavenDB server running locally and have an easy way to manage it, but it's completely blank and contains no databases at all. The first thing you need to do then is to create a new database, as demonstrated in figure 1.7. Note that you have to be an administrator on the machine, and you might be asked to provide credentials.

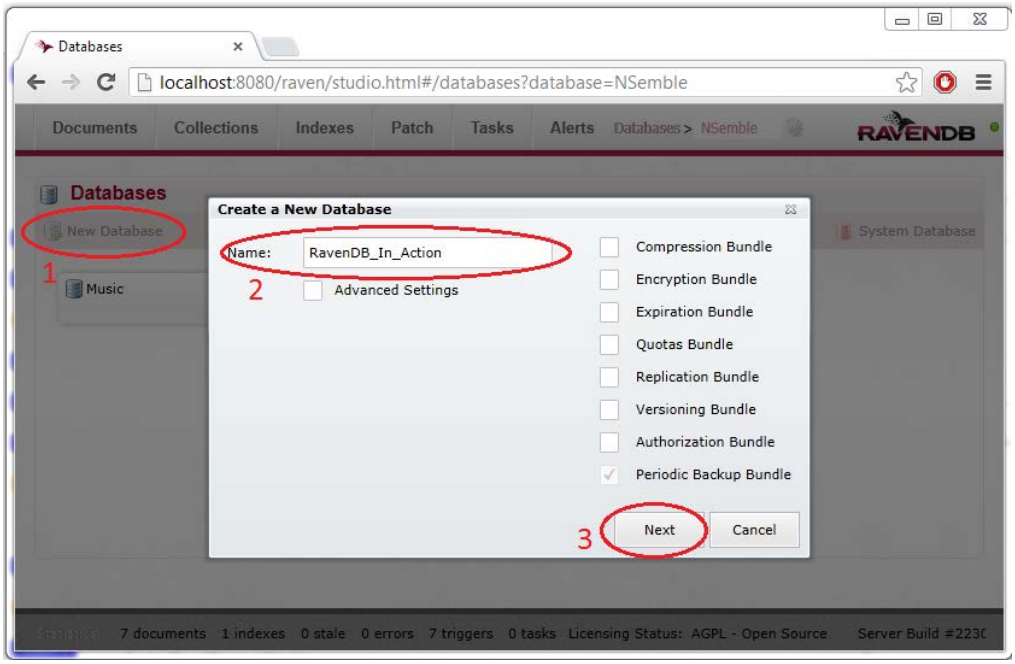


Figure 1.7: Creating a new database on a RavenDB server through the Management Studio

Double-clicking the database icon will allow you to browse the data in it, but because it was just created it's empty and there's nothing to see. If you go to Tasks -> Create Sample Data and click the Create Sample Data button, RavenDB will generate some sample data for you, RavenDB's take on the Northwind sample database. Do that now so that you can take a short tour of the database through the glasses of the Management Studio.

Figure 1.8 shows the Documents tab, which displays all the documents in the system. Note how documents from different collections are colored differently, and how the document itself holds data in JSON format, in a schema-less way

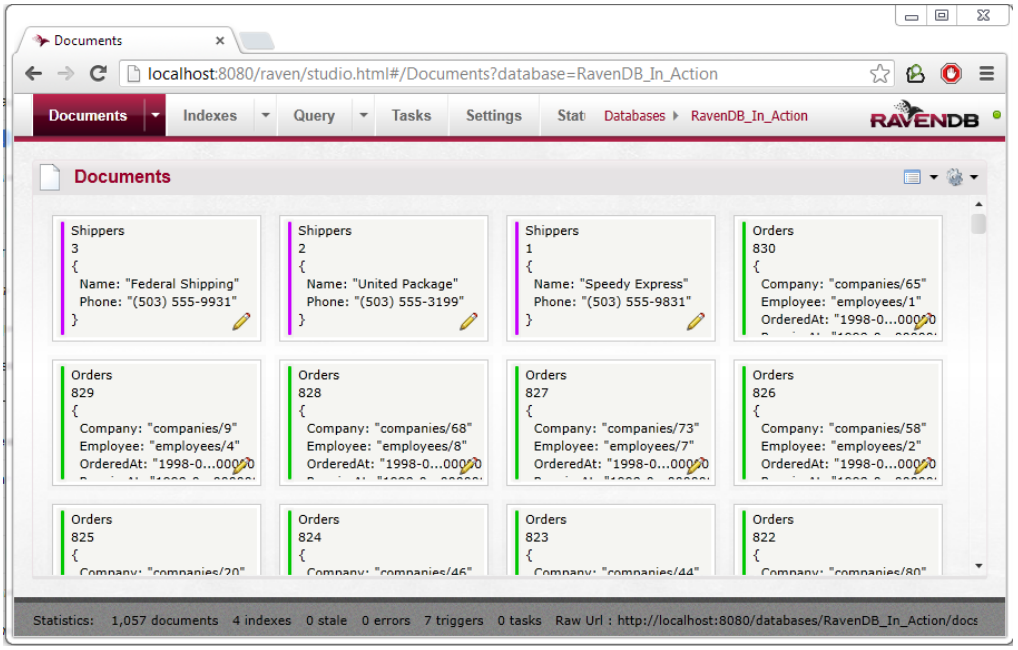


Figure 1.8 The Documents tab in RavenDB's Management Studio, showing all documents in the system

Double-clicking any of the documents will open it for editing, as shown in figure 1.9. Notice how the document ID is following the RavenDB convention for document IDs and the schema-less nature of the document, which contains complex objects and arrays of objects.

www.dbooks.org

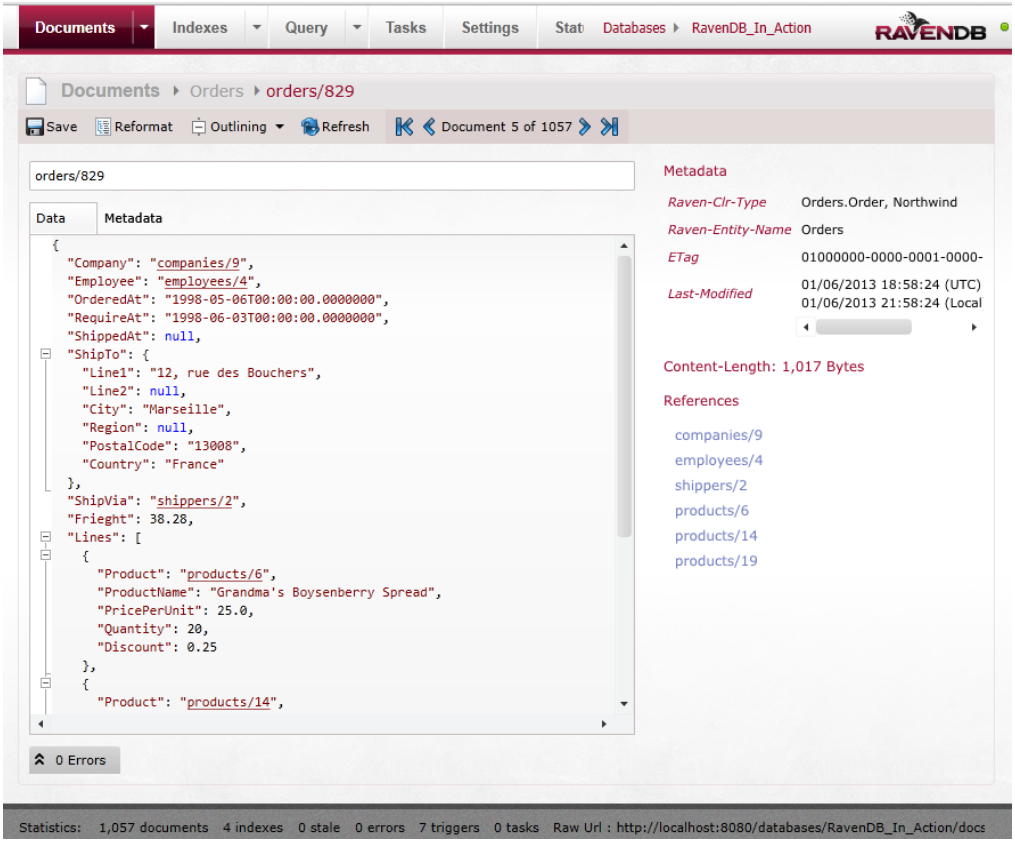


Figure 1.9 Editing a document through the RavenDB Management Studio

The Collections tab will show all documents in the system, browsable by their collection name. In our case you can see that all documents in the database are grouped into eight collections, among them are Orders and Products, and by clicking a collection name you can view all the documents that are under it, as demonstrated in figure 1.10.

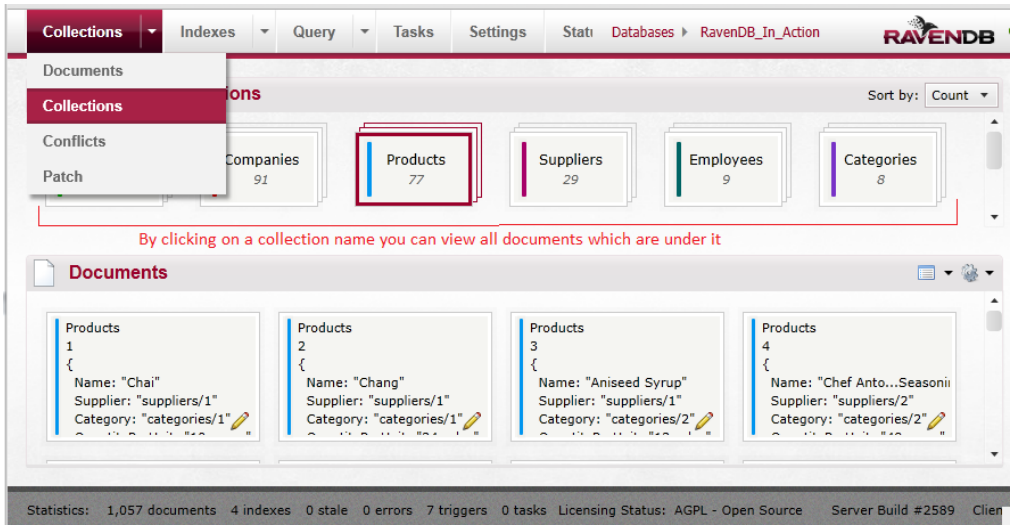


Figure 1.10 The Collections tab in the RavenDB Management Studio shows all the collections in the current database. Clicking a collection will display the documents that belong to it.

The other tabs are for managing indexes, logs, and other advanced operations like patching and administrative tasks. We're going to look at them in depth later in the book.

1.3.3 Adding RavenDB to your project

All throughout this book we'll use RavenDB in a client/server architecture, meaning we'll have a server instance running and the applications we write will use a RavenDB client to connect to the server and perform operations against it. When creating a new Visual Studio project of any type, you'll need to add the RavenDB client to it in order to talk to any RavenDB server.

The easiest way to add RavenDB to a project is by installing it from the NuGet gallery, the popular package manager for .NET. If you're not familiar with NuGet or don't have it installed, the easiest way to get started is by visiting their website: http://nuget.org/ .

To add the RavenDB client to your project using NuGet, search for the RavenDB.Client package, or type `Install-Package RavenDB.Client` in the Package Manager console.

You can also reference the client assemblies from the distribution package you downloaded earlier—just copy the /Client folder to your project and reference all the DLLs in it from your project.

It's important to know that there are several modes the RavenDB server can run in, as well as other ways to connect to it, and we'll review them later in the book. For now you'll use the standard RavenDB client to connect to the RavenDB server you have running locally in the debug console mode.

www.dbooks.org

1.4 Starting to work with RavenDB

After the initial setup is done, you're ready to start writing your first RavenDB-backed application. Before you start writing a full example application, you need to get familiar with the basic building blocks required by any RavenDB-backed application.

Any type of .NET application can use RavenDB as its backend store, ranging from websites, desktop applications, WPF, and WinForms, to thin clients using Silverlight..NET and non-.NET applications alike can use RavenDB using the server's REST API by issuing simple HTTP instructions. But as stated earlier, this book handles only the .NET Client API because it provides much better experience and hides a lot of technicalities.

To establish a connection with the RavenDB server from a .NET client, create a DocumentStore object and pass it the URL of the server or a name of a connection string. Using this DocumentStore object you can create sessions, which allow you to work with objects directly against the store. In this section you'll walk through this process.

You can follow the samples in this section by creating a simple console application, typing the code in as you go. Don't forget to add references to the RavenDB client assemblies and to have a RavenDB server instance running!

1.4.1 Instantiating a DocumentStore

Communication between client and server in RavenDB is done over HTTP. The DocumentStore object is responsible for handling the entire communication, as well as providing additional services like caching. Under the hood, that's the pipeline between your application and the database server.

Instantiating a DocumentStore on your client is quite simple: you're going to initialize a DocumentStore that's configured to talk to your local RavenDB server that you ran as a console and use the RavenDB_In_Action database you created earlier:

```
var docStore = new DocumentStore
        {
          Url = "http://localhost:8080", #A
          DefaultDatabase = "RavenDB_In_Action", #B
        };
docStore.Initialize(); #C
```

#A The URL the server is listening on
#B The database to use on the server
#C Don't forget to initialize the DocStore instance!

Remember to add a using directive to the RavenDB namespace, like so: `using Raven.Client.Document;`

Now that you have a DocumentStore instance initialized, you can use it to create DocumentSession objects, simply called sessions. Sessions are your main workhorse for most client operations.

Instantiating a DocumentStore isn't a cheap operation, and therefore an application should have only one DocumentStore object instantiated and reuse it for creating sessions. It's thread-safe, so multiple threads can access it to open new sessions. Therefore, the singleton

pattern definitely applies here: the general practice is to have a single global instance of the DocumentStore object, which is accessed from every interested part in the application.

1.4.2 The DocumentSession

After initializing a DocumentStore object, the connection between your application and the RavenDB server is established. You now want to start talking with the server and perform read or write operations against it. Whereas the DocumentStore is the communication pipe, a session object is the work context in which the developer—you—will be working to perform most operations against RavenDB. There are only a few things you'll need to access the DocumentStore directly for, and we're going to visit them later in the book.

A session object is very cheap to create and very easy to use. To create a session, you simply call `DocumentStore.OpenSession()`. Also make sure to dispose of it when finished (in the example here, the `using` will do that for you):

```
using (var session = docStore.OpenSession())
   {
        // Do work with the session here
   }
```

The Session object is responsible for performing serialization and deserialization of objects, maintaining transactions, and tracking changes made to loaded objects. Perhaps the most beautiful thing about the session object is how simple it is to use. Figure 1.11 shows the options it offers.

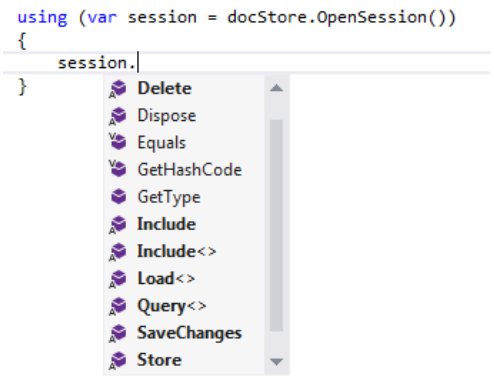


Figure 1.11 The methods available from the session object—to store new objects to the database or query it for existing ones

As shown in the figure, all common operations are easily accessible, and as you're about to see, they're also very easy to use. You're going to look at each of those more closely now, as you start putting data in RavenDB and then pulling it out in various ways.

www.dbooks.org

1.4.3 Storing a new document

As we mentioned, any .NET object can be a document in RavenDB. The only thing that needs to happen for an object to become a document is serialization—to transform it from an object to a string representation in JSON format. The session object handles all this seamlessly for you, and all you need to do is pass the object you wish to persist to the `Store()` method of your session object and commit it by calling `SaveChanges()`:

```
using (var session = docStore.OpenSession())
{
session.Store(new Book #A
{
Title = "Harry Potter",
Author = "J.K. Rowling",
Price = 32.99,
YearPublished = 2001,
});

session.SaveChanges();
}
```

#A – This is the Book class we've seen earlier on page X

This is all that's required to add a new document to a RavenDB server—assuming `docStore` was initialized previously, obviously.

Looking at Management Studio you can verify that the document has been added. Figure 1.12 shows the Collections screen in Management Studio after adding the new document. You can see how a new collection was added; when you click it the new document will be shown in the lower pane.

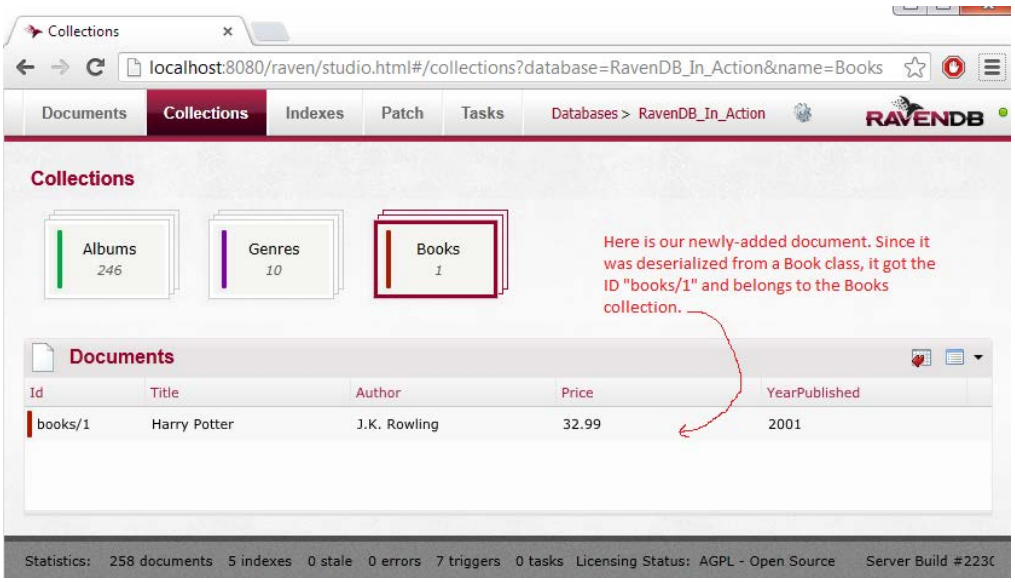


Figure 1.12: The Collections screen in Management Studio after storing a first Book object to our database

The call to `Store()` performs serialization and also assigns the newly created document an ID. That document ID is guaranteed to be unique no matter how many sessions were opened and how many clients are connected to the server. The generated ID is in line with the RavenDB convention we discussed before. In our example, the Book object was serialized to a document and got the `books/1` unique ID—`books` is the plural form of the class name, and `1` is the next integer that was available for that collection.

One other thing to note in the code snippet shown previously is the call to `session.SaveChanges()`after the call to `Store()`. Without calling `SaveChanges()` the operation wouldn't go through. Opening a session and performing operations on it won't commit anything to the database, because everything happens in memory in the session object itself. Until you call `SaveChanges()`, all you're doing is building a transaction, and only by calling `SaveChanges()` when you've finished do you tell RavenDB to commit it by sending it as commands to the server.

1.4.4 Loading and editing a document

Loading a document is done by ID, when you want to get a specific document you know the ID for. Load the document you just saved. As always, you start by opening a session and then call `Load()`, passing in the string ID you know your document has:

```
using (var session = docStore.OpenSession())
{
var book = session.Load<Book>("books/1");
Console.WriteLine(book.Title);
```

www.dbooks.org

```
}
```

The output to the console is going to be “Harry Potter”—the title of the book you saved moments ago. Deserialization from a JSON document to an object is also done seamlessly.

RavenDB also supports value types in the call to `Load`, so instead of passing the actual string ID, you can pass it the unique identifier within the collection itself, and it will build the full string ID on its own, using the convention:

```
using (var session = docStore.OpenSession())
{
var book = session.Load<Book>(1);
Console.WriteLine(book.Title);
}
```

And here comes something cool. The session object implements the unit of work pattern, so once you load a document through it, the session will be set to track changes for that object. Any object that is loaded from the database using the session object is tracked for changes until the moment the session is disposed of (in this case, the end of the `using(){}` block).

Just like with storing new objects, changes won’t be persisted until you call `SaveChanges()` on the session. Change the price for the book you just loaded, and save those changes back to the database:

```
using (var session = docStore.OpenSession())
{
var book = session.Load<Book>("books/1");
book.Price = 10.99; #A
session.SaveChanges(); #B
}
```

#A Price is now updated in memory and not in the database.
#B Only now the price change was saved to the database.

1.4.5 Deleting documents

Surprisingly or not, deleting a document is just as simple as all the other operations you’ve seen so far. If you have the object loaded via a session object, you can just call a Delete on it, and don’t forget to call `SaveChanges()` afterwards:

```
using (var session = docStore.OpenSession())
{
var book = session.Load<Book>("books/1");
session.Delete(book);
session.SaveChanges();
}
```

#A Marks the document to be deleted. This is done completely in memory on the client side, without anything taking effect on the server.
#B Performs the actual deletion by sending a command to the server.

To delete a document in one go without loading it first, you can use the `Defer` command available from the Advanced session operations (you will need to add another using directive to make this work: `using Raven.Abstractions.Commands;`):

```
using (var session = docStore.OpenSession())
{
session.Advanced.Defer(new DeleteCommandData { Key = "posts/1234" });
session.SaveChanges(); #A
}
```

#A Performs the actual deletion

With both approaches, deleting the document is fully transactional. If there are other operations participating in the transaction and one of them fails, the deletion will not happen.

1.4.6 Querying using LINQ

The most interesting feature of a database is probably its querying abilities, and RavenDB will not disappoint you here. By using LINQ as its querying language in the Client API, RavenDB gets slick and powerful querying abilities. Almost the entire LINQ grammar is available, making strongly typed queries easy to write and read, yet keeping them powerful.

```
using (var session = docStore.OpenSession())
{
var allBooks = session.Query<Book>().ToList(); #B

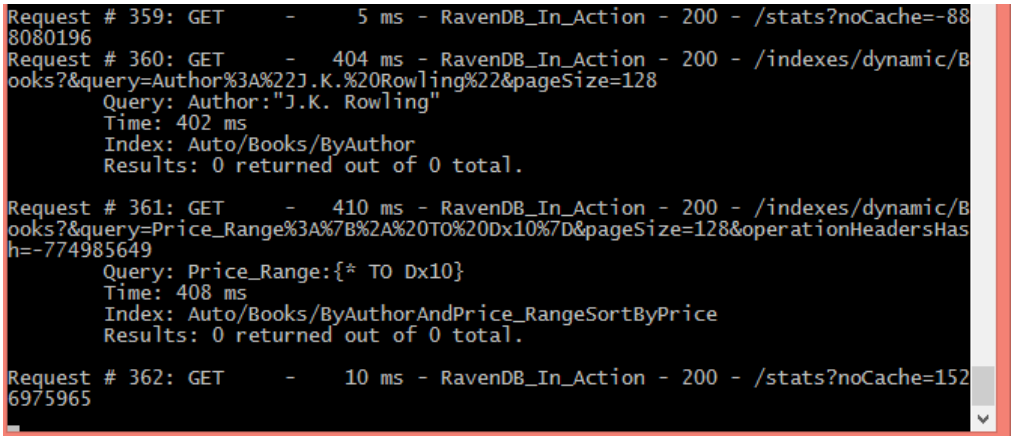
var jkRowlingBooks = session.Query<Book>()
.Where(x => x.Author.Equals("J.K. Rowling"))
.ToList();

var cheapBooks = session.Query<Book>()
.Where(x => x.Price < 10)
.ToList(); #A
}
```

#A Important: without starting to iterate through the results or calling `ToList()` or `ToArray()` on the query, the query won't get executed. Sometimes we tend to forget LINQ's deferred query execution.
#B Being safe by default, RavenDB will actively limit the amount of results to 128 by default, so this will never provide more than 128 results, even if there are a dozen times that amount of results satisfying the query. If you need more than 128 results, you should use paging, which we discuss in the next chapter.

If you try running this code, make sure you have some Book documents available on your RavenDB database, otherwise you will get no results. Either way, running a query is also a great opportunity for us to have a look at the RavenDB server console, which will show us the incoming requests:

www.dbooks.org



Similarly to loading documents by ID, query results will also get tracked for changes by the session objects. Just like you saw there, changes are going to be discarded unless SaveChanges is called, in which case all uncommitted changes to all tracked objects are going to be committed in the RavenDB server.

One type of query isn't supported. By design, queries that perform computations are not allowed, because this contradicts RavenDB's view of "reads should be fast." Although aggregation operations like max/min, group-by, and join are not supported in queries, you're going to learn how they can be performed later in the book, when we discuss indexes.

Congratulations! You now have all the building blocks you'll need to continue with the next chapter, where you'll build an actual web application using RavenDB as its storage. In the chapters that follow you'll learn more about the internals of RavenDB and what's going on under the hood. This knowledge is important in order to build applications that take advantage of all of RavenDB's nice features.

1.5 Summary

Databases are a crucial part of almost every software. They're needed in most projects, and they need to be reliable. In modern days—the BigData era—databases also have to be able to deal with a lot of data and scale out easily. RavenDB was started with all that in mind, but it also set out on a journey to make sure developers' lives are made easier even though RavenDB's task is pretty heavy.

We started this chapter by looking at the various problems in the world of data persistence. We explained where relational databases came from and how the assumptions behind them have become less and less relevant today. You then saw how various NoSQL-type databases offer data persistence solutions that fit better the requirements of modern software projects.

RavenDB is such a NoSQL database, more specifically a document database—one that deals with documents. By providing a very easy to use and intuitive Client API for .NET developers, the hope is more efficient applications can be built more quickly and easily and the scale of data will become a non-issue.

Once you've referenced the RavenDB client assemblies from your application and you have a server instance running, all that's left for you to do is instantiate a communication pipeline, the DocumentStore, which is also your session factory.

Because all operations made within the context of a session are transactional, it's important to remember to call `SaveChanges()`, which will commit the transaction. Without doing so, all operations that were made in memory will be completely discarded.

ASSUMPTIONS

We are going to assume the following in these first few chapters because it's easier to start learning RavenDB with these simplifying assumptions in mind:

- There are no references between documents.

  A document is self-contained. One document means one .NET object (a POCO). There are never relations between documents, and no references are made between documents.

- Querying just works.

  RavenDB uses some sort of black magic to be able to satisfy any type of query you throw at it and do it fast.

- RavenDB is ACID.

  Once you put data into RavenDB, you can immediately get it back.

Actually, none of those assumptions is completely true, but during your first steps with RavenDB it's perfectly safe to assume they are. We're going to revisit and untie each of them when the time comes.

IMPORTANT LINKS

RavenDB homepage—with documentation and downloads: http://ravendb.net

RavenDB's mailing list: http://groups.google.com/group/ravendb

Source code: http://github.com/ravendb/ravendb

www.dbooks.org

2

Your first RavenDB application

In Chapter 1 we introduced RavenDB and the basic concepts required for working with it. We also learned about the basic building blocks required for writing a .NET application that uses RavenDB as its backing store – the `DocumentStore` and `DocumentSession` objects.

Now it is time to take off. In this chapter we are going to build our first RavenDB-driven application: an ASP.NET MVC 4 bookstore. It is going to be a rather simplistic version of a bookstore, which means we will only be implementing some basic screens for viewing and editing the store catalog. Although simplistic, it is going to be a fully working application which we can use to demonstrate how to build an application using RavenDB.

> **NO ASP.NET MVC KNOWLEDGE REQUIRED!** You should be able to follow this chapter also if you are not familiar with ASP.NET MVC at all. While we will not be stumbling upon MVC concepts, or learn how to use it, and I will assume that you have some familiarity with ASP.NET MVC – this chapter intends to teach actual usage of RavenDB and just uses ASP.NET MVC as a context.
>
> Only because ASP.NET MVC is the most commonly used technology for building web applications with .NET, we are going to use it in this chapter; however, you should be able understand the RavenDB concepts taught in the chapter without any MVC knowledge.

We will start by defining our data model, and then setup our project and look at how the RavenDB sessions should be managed in a web application. After we have done that, we can build basic CRUD (Create-Read-Update-Delete) screens to enable us to add books to the store's catalog, and then look into querying it.

Once the basic screens are written and working, we will look at how to write tests using RavenDB's in-memory option, and what happens when we need to change our data model.

It is recommended that you try and follow this chapter hands-on, either by building your own ASP.NET MVC web application following this chapter, or writing whatever other application to experiment with the RavenDB client API your way. If at any point you feel stuck, make sure to check the source code for this chapter in the source code that accompanies this book (this is also where you will find some missing parts, like the Views and so on).

The goal of this chapter is to give you all the tools necessary for building RavenDB applications, and more importantly – the confidence in RavenDB. By the end of this chapter you should have all that is needed to be able to start building RavenDB-backed applications yourself. The rest of the book will explore RavenDB in more detail.

2.1 The Raven Bookstore

The sample application we are going to build in this chapter is an online bookstore, and for us to sell books to customers we need to have a catalogue of titles. Our customers need to have a way to browse the catalogue, and we need a way to manage it – adding and removing titles as the stock changes.

To allow for that we will create screens to add, edit and delete books, and a screen for browsing the catalogue by various criteria, including a screen to actually view a book.

The RavenDB Bookstore we are going to build will be an ASP.NET MVC web application. To start, open up Visual Studio and create a new ASP.NET MVC 4 web application. When asked what type of ASP.NET MVC project to create, select the Basic template with a Razor view engine and check the box to include the accompanying test project, like shown in figure 2.x.

www.dbooks.org

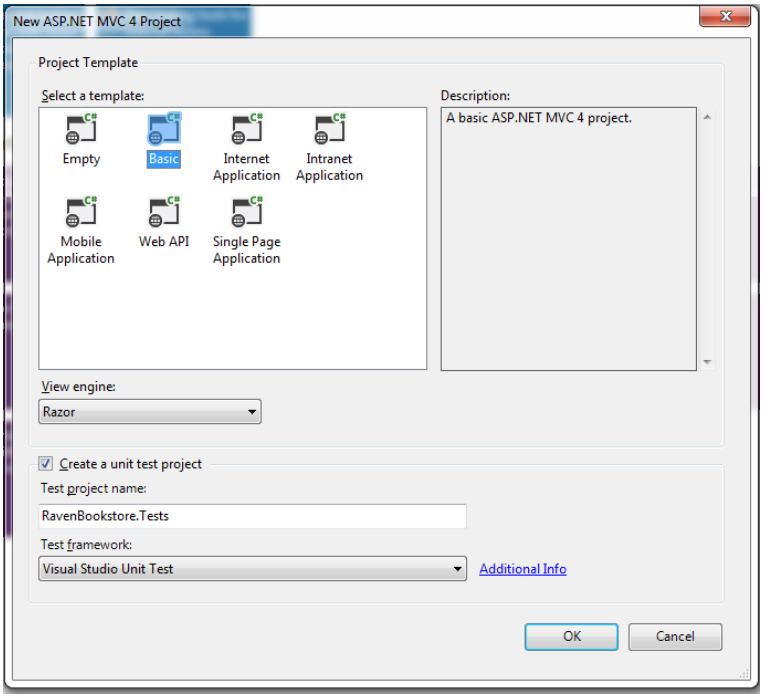


Figure 2.x: Creating a new ASP.NET MVC 4 project for our Bookstore sample web application

Now that we have a new project open and ready, in this section we are going to create the data model we need for the Bookstore, and then setup our project to use RavenDB. After we have a proper data model in our project and it is talking to a RavenDB server, we can start adding the various screens to the website.

2.1.1 Planning the Model

Before adding RavenDB to our website and actually starting to store and load data, we need to have a *Model* to work with. Creating the Model should really be the very first step of development of any data-driven application. The Model – the M of MVC – is just a set of classes representing the entities we are going to store.

The first entity in our Model is a Book. Each of our entities should contain fields to hold all data we wish to hold on it. So a standard Book class in our Model could look like this:

```
public class Book
   {
        public string Id { get; set; }
        public string Title { get; set; }
        public string Author { get; set; }
        public string Description { get; set; }
        public double Price { get; set; }
```

```
        public int YearPublished { get; set; }
        public List<string> Departments { get; set; }
    }
```

We want our bookstore to have the ability of showing customer reviews on books, and also ratings on books purchased. For that end, let's create an appropriate CustomerReview class:

```
    public class CustomerReview
    {
        public int Rating { get; set; } // number of stars out of 5
        public string Review { get; set; }
        public string CustomerName { get; set; }
        public string CustomerEmail { get; set; }
}
```

We now need to add a field to our Book class to hold the review. Since a document in RavenDB is a complete object graph, we can store objects and lists of objects – it will just work and require no further action from our side. Here is the complete Book class:

```
    public class Book
    {
        public string Id { get; set; }
        public string Title { get; set; }
        public string Author { get; set; }
        public string Description { get; set; }
        public double Price { get; set; }
        public int YearPublished { get; set; }
        public List<string> Departments { get; set; }
        public List<CustomerReview> CustomerReviews { get; set; }
}
```

Since implementing user login is out of scope for this application, for us a user ID is his email.

Our model is quite simple, a Book class with some properties and an array of CustomerReview objects. Each book is going to be stored as one document, along with all the reviews written for it. This has both advantages and disadvantages that we'll get to in Chapter 5, but for now it is easier to think of a Book in the system as one document containing everything there is to know about that book.

2.1.2 Setting up

For our web application to be able to store data in RavenDB, we need to have a server instance running somewhere, and then connect to it using the RavenDB .NET client assemblies. We did exactly that in the end of Chapter 1. Let's quickly reiterate the steps here as well:

1. **Get the distribution package** by going to http://ravendb.net/download.
2. **Have a server running:** Extract the /Server folder from the distribution package to your hardrive, and run Raven.Server.exe. For better development experience, use the /ram flag so the server run entirely in memory so you can easily reset it at any time.

www.dbooks.org

3. **Add RavenDB to your application:** Using the NuGet Package Manager install the RavenDB.Client package, or type Install-Package RavenDB.Client in the Console Manager console.
   Alternatively, reference the assemblies from the /Client folder in the distribution package from your project.

The last thing we need to do before we start building the actual site is take care of session management – the lifecycle for the RavenDB session. We need the session object to talk with the RavenDB server, and with proper session management we can make the session object easily accessible and managed automatically.

2.1.3 Session Management

Now that we have put all the pieces together, it is time to start adding the code to connect RavenDB to our application.

Recall how In chapter 1 we learned about the `DocumentStore` and the `DocumentSession` objects – the `DocumentStore` object is in charge of the communication with the server, and the `DocumentSession` object is what we actually use to perform most of the operations against RavenDB. We use the `DocumentStore` to create new `DocumentSession` objects.

We also said there should always be only one `DocumentStore` object per database connection application-wide, mainly because it is costly to create. The single instance we create and keep around can then be used to create `DocumentSessions` whenever necessary.

This also applies to web applications. Only one connection to the RavenDB server should be created when the application starts, and then it should be kept alive throughout the life of the application. In ASP.NET MVC the best place to initialize the `DocumentStore` is global.asax, in the `Application_OnStart` method. We will also make the `DocumentStore` static and publicly accessible, so we can access it later from other places in our application – mainly for creating new session objects:

Listing 2.1: Initializing the DocumentStore object in Global.asax.cs

```
public class MvcApplication : System.Web.HttpApplication
{
    public static IDocumentStore TheDocumentStore { get; private set; }

    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();

        WebApiConfig.Register(GlobalConfiguration.Configuration);
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);

        TheDocumentStore = new DocumentStore
                           {
                               Url = "http://localhost:8080",
                               DefaultDatabase = "BookWebstore"
                           };
```

```
            TheDocumentStore.Initialize();
        }
    }
```

Note how we specified the server URL explicitly in code. Usually you would want to avoid hardcoding such values and rather specify it in a configurable place, like the web.config file, so they can be changed easily. To do that, you can use a Connection String. You define the connection string in the configuration part of your web.config:

```
<configuration>
  <connectionStrings>
    <add name="RavenDB"
connectionString="Url=http://localhost:8080;DefaultDatabase=BookWebstore" />
  </connectionStrings>
…
```

The connection string can be then passed by name when initializing the document store:

```
documentStore = new DocumentStore
                                {
                                    ConnectionStringName = "RavenDB"
                                };
            documentStore.Initialize();
```

Once we got a `DocumentStore` object created and the communication with the database server established, the next step is to streamline the process of creating new `DocumentSession` objects and their disposal so we don't have to manually handle this for every operation we want to make against the database.

A session represents a work context by the user, so in a web application it makes the most sense to have one session per HTTP request. If we open a session when a new HTTP request comes in and close it when the response is ready to be sent back, we wrapped all the database work required by this request in a single database session.

The easiest way to streamline session management in an MVC web application is by using a custom Controller base class. The Controller base class will take care of creating and closing the session object, so that we don't have to do this manually every time we need a session – it will just be around for us, ready for use. Deriving from this class will then let you have a RavenDB session open and ready to work with.

This is much easier then opening a session manually every time, and also means all changes made within the same HTTP request are committed in one transaction – so either all go through or all fail. Since sessions are cheap to create, even if some Action methods are not going to use the session object, the cost of having it there is negligible.

Let's create a new Controller class called `RavenController` which will take care of managing the RavenDB session lifecycle:

www.dbooks.org

Listing 2.2: RavenController – a controller base class to manage RavenDB session lifecycle:

```
    public abstract class RavenController : Controller
    {
        public IDocumentSession RavenSession { get; set; }

        protected override void OnActionExecuting(ActionExecutingContext
filterContext)
        {
            RavenSession = MvcApplication.TheDocumentStore.OpenSession(); #A
        }

        protected override void OnActionExecuted(ActionExecutedContext
filterContext)
        {
            if (filterContext.IsChildAction)
                return;

            using (RavenSession)
            {
                if (filterContext.Exception == null) #B
                    RavenSession.SaveChanges();
            }
        }
    }
```

#A – This method is executed before any Controller Action is invoked, exactly where we want to open the session so it is available for the Controller Action to use.
#B – We save all changes made unless an exception was thrown somewhere in the process

Since the RavenDB session object performs all operations in memory, storing, changing and even deleting documents will have no effect on the database until `SaveChanges()` is called. In our implementation this will only happen when the page processing is done, and if no unhandled exception was thrown in the process. Only then the call to `SaveChanges()` commits all the changes to the server, in one transaction.

Using your favorite Inversion-of-Control container

Many developers today like using IoC or DI containers to inject object instances to a context, instead of passing them manually (and in quite a clumsy way) like we just did here.

RavenDB's DocumentStore and DocumentSession are entirely capable of that. Use your favorite container – Castle Windsor, AutoMapper, Ninject, StructureMap or whatever else – to inject `IDocumentStore` with a single instance of a `DocumentStore` created as a singleton, and `IDocumentSession` with a `DocumentSession` instance per call that is marked as requires disposal, and was created using the singleton instance of `IDocumentStore` you just registered with the container.

Now, by deriving our Controller classes from RavenController we can access the RavenSession object to perform data access operations. This is exactly what we are going to do starting in the next section. To visually understand the steps we've just gone through to get our application setup to use RavenDB, take a look at figure 2.x:

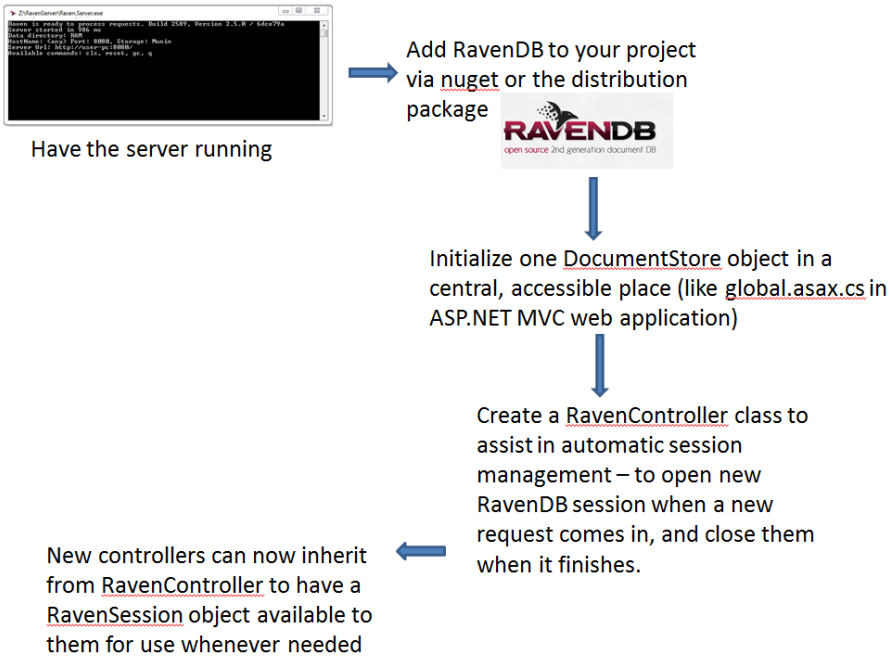


Figure 2.x: The process of adding RavenDB connectivity to an ASP.NET MVC web application

2.2 Basic operations

By now we have a RavenDB server running, a brand new MVC web application with a Model defined and the ability to connect and talk to our database server. Now it is time to create some screens and perform actual work against the server – adding books, viewing them, editing their details and deleting books. In the next section we will add the ability to browse the book catalogue using filtering, sorts sorting and paging.

The RavenDB .NET Client API goes a long way to provide a smooth and easy working experience with the database server. Perhaps the most important feature is the strongly-typed API for making changes to the database and loading documents. Since the session object deserializes everything automatically, you never deal with documents – always with objects. Using Linq as the strongly-typed querying language for the client enhances this capability so you can also express queries using the classes in your model, without learning a new query language.

www.dbooks.org

During the following pages, as we build our first screens using the RavenDB client, pay attention to the flow and how easy it is to work with. Getting used to the simplicity and seeing how it "Just Works" is an important step in the process of embracing RavenDB. The simplest solution is usually the best one when it comes to working with RavenDB.

While some actions we perform in this section are things that an admin of the website will do and normal user should not have access to them, for simplicity we will just use one Controller for them all and not handle security. Let's create a new Controller and call it BooksController. By making it inherit from RavenController like shown in listing 2.3 we get an access to RavenDB session object, and then have easy access the RavenDB server.

2.2.1 Listing all books in the system

The newly created `BooksController` class is empty, but by now it should inherit from `RavenController` so we know it has access to the RavenDB session object. Now we are going to see how easy this makes writing MVC Controller Actions.

Before adding any new content to our web-store, we want to make sure we have a way of viewing its books catalogue. The first thing we are going to do is add a way for us to look at all the books in the system, by simply querying RavenDB for all the books in the system and displaying them.

On MVC, Views and ViewModels

MVC applications are in their heart a separation of 3 core concepts – the data Model, the Views the data is displayed in, and the Controllers which coordinate user actions coming from the Views and various APIs, perform the required actions and return some result back, usually other Views. The idea behind this separation is to allow each part of the software to have one purpose and not to mix business logic with application logic.

For MVC applications to really work as intended there has to be complete separation between the 3 concepts – Model, Views and Controllers. They may know of each other, but should really be kept completely separate.

Since the Views usually don't just display one class of the Model as-is, but require some additional Model classes and often times also a mutated version of them, a common way of passing information between Views and Controllers is the ViewModel. ViewModel classes are essentially a copy of the Model classes but structured in a way that fits for each View. This in turn usually gives a class per View, bloating the amount of code to write and maintain one again.

For this reason and many more, in real life achieving a real and rigorous separation is hard work, and many people just give up. To avoid such complexity ourselves, in the following examples we will not use real Views. Instead, whenever possible we will have JSON strings as a response, which you can easily view in your browser. Whenever we will use actual Views they will be available in the source code examples accompanying this book.

For an even easier experience, I advise you to use the Chrome browser with the JSONView add-on. This will display the JSON output in a very easy to read format.

As we've seen in chapter 1, querying RavenDB is easy and only requires writing a Linq expression using the `session.Query<T>()` method. A call to the `Query<Book>()` method without narrowing it down at all will just ask RavenDB for all the Book documents in the system. RavenDB, in turn, will send the first 128 documents and not the potentially huge result set, as part of it being "Safe by Default".

Let's add an Action method to our `BooksController` class to show all the books in the system:

```
using System.Linq;
using System.Web.Mvc;
using Chapter02Bookstore.Models;
using Raven.Client;

namespace Chapter02Bookstore.Controllers
{
    public class BooksController : RavenController #0
    {
        public ActionResult ListAll()
        {
            var books = RavenSession.Query<Book>().ToList();
                                        #A    #B       #C

            return View("List", books); #D
        }
    }
}
```

#0 – By inheriting from `RavenController` we gain access to the `RavenSession` object, which is now initialized and ready for us to work with
#A – Starts to construct a query
#B – Asks for documents of the Books collection
#C – Executes the query
#D – Passes the resulting List<Book> to the View for rendering the page to the user

Now that we have means to inspect our books catalog, we can go ahead and add new books to the system. We will be working with more queries in the next section.

2.2.2 Adding new books

What we want to do now is be able to add new books, so our catalogue actually has something to show. In order to do that, we need to create a new Controller Action method and name it Create. We need two variations – one for the Get HTTP verb which will present the user with an input form, and the other handling POST HTTP requests, to catch form submissions.

The GET method isn't really much interest to us, but the POST is since its where all the action happens. After we get the submitted form data in the POST action method and have an instance of a Book class, we can go ahead and persist it by calling `RavenSession.Store()`.

Let's add the Create method to our `BooksController`:

www.dbooks.org

```
        [HttpGet]
        public ActionResult Create()
        {
            return View(new Book());
        }

        [HttpPost]
        public ActionResult Create(Book book)
        {
            if (!ModelState.IsValid)
            {
                return View(book); #A
            }

            RavenSession.Store(book); #B

            return Json("Book was successfully added and was assigned the ID
" + book.Id, JsonRequestBehavior.AllowGet); #C
        }
```

#A – checking the Model state from the submitted form. If it's invalid we will return to show an error to the user.
#B – Storing the new Book object we got from the form in the current RavenDB session. This will not store the object in the database just yet, just serialize it and assign it an ID.
#C – Once the action method will return, SaveChanges() will get invoked in our base controller RavenController, and the new book object will be sent and persisted on the server. Our new book object has been assigned an ID by the client API on the call to Store(), so calling book.Id isn't meaningless.

REMINDER Remember it is not the call to Store that saves the data, but the call to SaveChanges in RavenController just before the session is disposed that actually commits the change. It is crucially important to remember that, and I'm going to remind you of this a few more times.

Let's compile and run the application to add a new book using this form, and verify it was saved to the database. If it indeed was, it means we connected to the server correctly, have the session management setup correctly, and ASP.NET MVC hasn't failed us. We can verify it was created by pointing our browser to the server, and looking at the Collections tab in the "BookWebstore" database in figure 2.x:

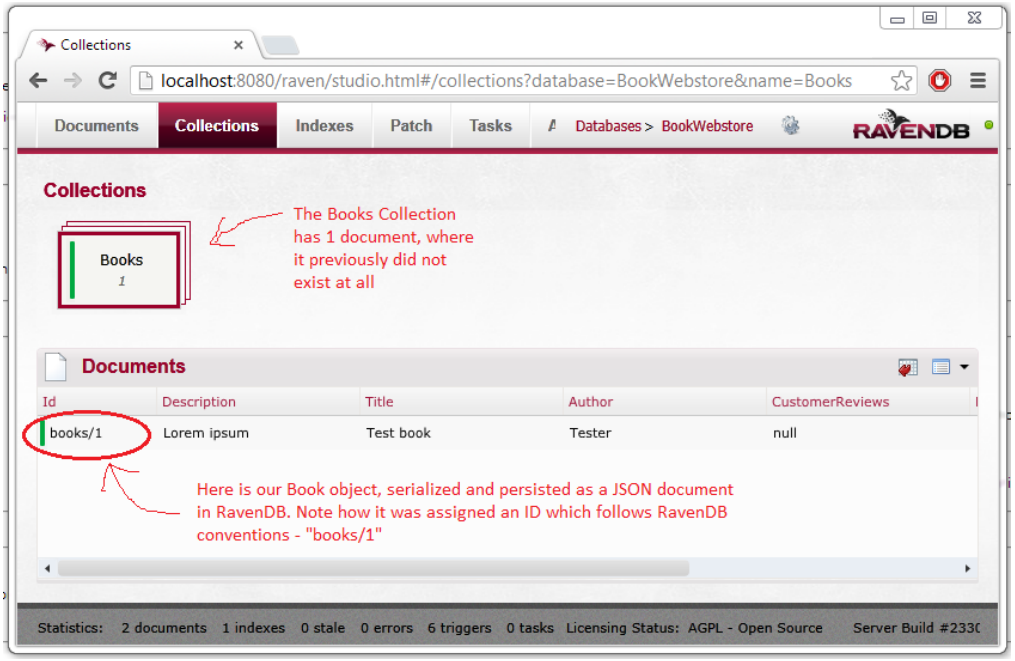


Figure 2.x: After storing a Book object in the database, we can use the Management Studio to view it as a document.

At this point, we've created basic screens to show all the books in the catalogue, and to add new books. Our next step is to be able to show the user the book he or she is interested. This way we can give them complete details about a title. Let's create this screen now.

2.2.3 Viewing an existing book

Imagine that a user wants to know more about a book before they buy it. For them to get the information they need they decide to look up the title by ID, by clicking on the book title in the book listing page. They expect to get to a screen where all the book details are shown - this is going to be the next screen we will write.

Viewing a book by ID can be easily done by calling `RavenSession.Load<>()` with the document ID. The session object will then get the Book document bearing the requested ID from the server, deserialize it into a Book object and then it can be passed to the View for display.

There is, however, one small thing to note. If you'd recall, all IDs in RavenDB are strings and the default convention contains a "/" character, so all the Book documents in the system end up having IDs like `books/1`. This poses some difficulty when working with MVC routes expecting IDs – like the default routes used for viewing or editing entities, since they do not expect to find slashes in the IDs.

www.dbooks.org

To avoid clashes with the MVC routing system we are going to use the value-type version of `Load<T>()`. As we've seen earlier, the RavenDB client API is smart enough to transform an integer ID to the full-blown string ID. Therefore, we need to add a Show Action to the `BooksController` class, which will look like this:

```
        public ActionResult Show(int id)
        {
            var book = RavenSession.Load<Book>(id);
            if (book == null)
                return HttpNotFound("The requested book wasn't found in the
system");

            return Json(book, JsonRequestBehavior.AllowGet);
    }
```

It is indeed that easy. The call to Load will both communicate with the server and perform the deserialization from a JSON document to a .NET object. If the document couldn't be found, it will simply return a null.

Key Takeaway: MVC-friendly Load method

Document IDs in RavenDB are always strings, but sometimes it is useful to use C# value-types like integers for loading documents. This is extremely useful when using MVC frameworks like ASP.NET MVC, where using this method is necessary to be able to work with the default routes defined in those types of frameworks.

Luckily, we can rely on the RavenDB Client API to use the conventions to figure out the string ID, given the value-type ID and the requested object type. To do this, RavenDB has a value-type version of the `session.Load<T>()` method; this method will figure out the correct string ID and use it to load the document from RavenDB.

We are going to use this method in the rest of the controller actions we will be writing in this section as well

We have just loaded a book to display, so now we have a very basic book catalogue working – we can add books, list all of them and show an individual book once selected. Now we move on to more advanced screens – we start by building screens for managing existing books.

2.2.4 Editing existing books

Let's say we detected an error in the details of one of the books, and we need to correct it. In order to do that, we need to have a way to edit the book, and this is what we are going to do next. Luckily, RavenDB makes it amazingly easy to do. It is just a matter of loading the object and making whatever changes we want to it. RavenDB takes care of all the rest for us.

The RavenDB session implements the Unit-of-Work pattern, which means the session will track all changes to objects retrieved from RavenDB. As long as the session is open every

change made to objects loaded using it will be committed to the server on the first call to `SaveChanges()`. Until then all changes are kept entirely in memory. Not calling to `SaveChanges()` before closing the session means the changes won't be committed, and get discarded.

In our application we are always making this call to `SaveChanges()` in the `RavenController` base class, just before closing the session. This means all we have to do in order to implement editing is load the object, assign it the values we got from the form fields, and by the time we return the view to the user the changes have been committed to the database:

Listing 2.x: Action methods for editing a book in the system

```
        [HttpGet]
        public ActionResult Edit(int id)
        {
            var book = RavenSession.Load<Book>(id);
            if (book == null)
                return HttpNotFound("Requested book wasn't found in the
system");

            return View(book);
        }

        [HttpPost]
        public ActionResult Edit(int id, Book input)
        {
            var book = RavenSession.Load<Book>(id);
            if (book == null)
                return HttpNotFound("Requested book wasn't found in the
system");

            book.Title = input.Title;
            book.Author = input.Author;
            book.Description = input.Description;
            book.Price = input.Price;
            book.YearPublished = input.YearPublished;
            // And so on...

            return Json("Book was edited successfully",
JsonRequestBehavior.AllowGet);
        }
```

Just like before, it is easy to verify that the desired changes were made, by logging in to the Management Studio and opening the document for editing. We can also navigate to the Show page to see the document through our own UI.

Key Takeaway: The Unit-of-Work pattern

Any object loaded from RavenDB using the session object is going to be tracked for changes. Any change made to the object will be committed to RavenDB once SaveChanges() is called

www.dbooks.org

for this session. Calling to SaveChanges() is the equivalent of committing a transaction in other databases.

Just like when loading by ID, the Book objects returned by a call to Query will be tracked for changes as well.

Remember changes to objects you load from RavenDB are tracked for changes automatically so you don't make changes inadvertently, and don't forget to call SaveChanges() after you've made changes to objects otherwise they won't be persisted.

Next we will see how we can delete a book from the catalogue, should this action be necessary.

2.2.5 Deleting a book

Now we have screens for adding books, we've edited one, and now we also want to allow the store admin to remove books from the catalogue – for example as means for correcting an error.

Deleting a book is pretty much like editing it. After we load it we mark it for deletion by calling `RavenSession.Delete()`, and when the session's `SaveChanges()` method is called it will get deleted.

```
        [HttpPost]
        public ActionResult DeleteConfirmed(int id)
        {
            var book = RavenSession.Load<Book>(id);
            if (book == null)
                return HttpNotFound("The requested book wasn't found in the
system");
            RavenSession.Delete(book);

            return Json("Book was deleted successfully",
JsonRequestBehavior.AllowGet);
    }
```

Now that we have the code for all the basic operations laid out, we want to allow our users to browse the books catalog. This is essentially querying RavenDB for all books by various criteria, and is exactly what we are going to cover next.

2.3 Querying

If we stopped working on the application right now, there would be no way for our clients to find anything to buy from the online bookstore, because they'd have no way to actually browse the books catalogue. While we did provide them a way to see all the books in the system, it is not so helpful when there are many books in the catalogue to go through.

To the rescue here are queries. Queries in RavenDB are strongly-typed, which means you get to work directly with your Model classes, and they use Linq as the querying language. If

you are not familiar with Linq, I suggest you check the primer in chapter 1. It is important you feel comfortable with Linq before going any further in the book.

One important aspect of queries in RavenDB is that they do not allow for computations. Queries are in fact made on a pre-calculated dataset on the server, which gets updated whenever a change is made to the data. These are called indexes, and we are going to learn more about this behavior in chapter 3, but I just wanted to get your attention to the fact that all the queries you are about to see in this section are simple. This is by design, and does not at all mean the querying abilities are limited.

Now let's see how querying actually works in RavenDB and make it so that users can browse our book catalogue.

2.3.1 Paging

Earlier in the chapter, we added a way to show all the books in the system. Our controller action looked like this:

```
public ActionResult ListAll()
{
    var books = RavenSession.Query<Book>().ToList();

    return View(books);
}
```

If you'd recall, by default RavenDB will only serve 128 results per query. The number of results returned for the query can be controlled using the Take() operator. The default page-size for queries is 128, so the above example is equivalent of having this:

```
public ActionResult ListAll()
{
    var books = RavenSession.Query<Book>().Take(128).ToList();
    return View(books);
}
```

Any query will return a maximum of 128 documents by default. While the exact number of results returned for a query can be changed by explicitly specifying a value in the `Take()` operator, it will only respect values up to 1024. Getting more than 1024 results in one query requires a change to the server configurations – part of RavenDB's Safe-by-Default principles.

As a matter of fact, there isn't real value in sending more than one or two dozen results for a single query. Think about it – when searching in Google, or browsing a listing in a website, how often do you really look at the 10th item in the page? For use-cases requiring a snapshot of the data for reporting or exporting, there is a dedicated API to do that which we will look at in chapter xx.

The easiest way to browse through query results is with paging. Using Linq's `Skip()` and Take() operators, and by defining a page size and knowing the current page number, it is easy to implement paging for any query:

```
private const int PageSize = 10;
```

www.dbooks.org

```
public ActionResult ListAll(int page)
{
    var books = RavenSession.Query<Book>()
        .OrderByDescending(x => x.YearPublished)
        .Skip(PageSize * page)
        .Take(PageSize)
        .ToList();

    return View("List", books);
}
```

Using paging we can allow our users to go through many results easily by presenting them only a handful at results at time. Next we will look at sorting the results using various criteria, and filtering them using user queries.

2.3.2 Sorting results

What if our users wanted to sort books by various criteria? For example to have more recent books first, or cheapest books at the top of the list? We are now going to add several other versions of the ListAll() controller action method which will support paging as well.

Unless a sorting criterion was specified in the query, query results are returned in an arbitrary order. We can ask the RavenDB server to sort query results on the server before sending them back, by using Linq's OrderBy and OrderByDescending operators.

Let's change our ListAll() method to sort results by the year they were published so new books appear first:

```
public ActionResult ListAll()
{
    var books = RavenSession.Query<Book>()
        .OrderByDescending(x => x.YearPublished)
        .ToList();

    return View("List", books);
}
```

Alternatively, we can show cheaper books first by specifying a different OrderBy clause:

```
public ActionResult ListAll()
{
    var books = RavenSession.Query<Book>()
        .OrderBy(x => x.Price)
        .ToList();

    return View("List", books);
}
```

So now we know how to list all books in the system, and how to easily page through them and sort the results by various criteria. There is one thing missing however before we can say we have complete paging support – the ability to tell the user how many books in total there are in the system. This is what we will do next.

2.3.3 Query statistics

To give our users better perspective on what's going on when paging, we want to show them some numbers – how many results they have to go through, which page they are on, etc. For any query, RavenDB can report some statistics that may often come in handy. The most commonly used item provided in the query statistics is the total number of results in the database satisfying the query.

To get the query statistics object alongside the query results, add the Statistics operator right after the call to Query(). Let's use it to report the number of Books found by a query:

```
public ActionResult ListAll(int page)
{
    RavenQueryStatistics stats;
    var books = RavenSession.Query<Book>()
        .Statistics(out stats)
        .OrderByDescending(x => x.YearPublished)
        .Skip(PageSize * page)
        .Take(PageSize)
        .ToList();

    ViewBag.Title = stats.TotalResults + " books found";

    return View("List", books);
}
```

This, in combination of paging, can give the ultimate browsing experience to our visitors, as it gives them a complete picture on the books in the system.

Next we show how to filter the results, so we only get books we know we are interested in instead of paging and sorting through tons of mostly irrelevant results.

2.3.4 Filtering

What happens when we already know of some criteria we can filter books by? For example when we only look for books published in a certain year, or when we have a limited budget and only want to list books which are within our budget?

As we've seen in Chapter 1, querying RavenDB is easily done by using Linq through the `Query<T>()` method of the session object. We already issued queries using this method several times, but now we are going to add actual filtering operators to the query.

Querying with LINQ is really simple and easy to work with. Let's add a few more methods of querying for books, to enrich the experience of browsing the books catalog. These are going to be some common queries when looking for books – for example filtering books by year, by department or by price:

Listing 2.x: Common queries for books with filtering

```
public ActionResult ListByYear(int year)
{
    var books = RavenSession.Query<Book>()
        .Where(x => x.YearPublished == year) #A
        .ToList();
```

www.dbooks.org

```
            return View("List", books);
        }

        public ActionResult ListByDepartment(string department)
        {
            var books = RavenSession.Query<Book>()
                .Where(x => x.Departments.Any(y => y.Equals(department))) #B
                .ToList();

            return View("List", books);
        }

        public ActionResult ListByPriceLimit(double price)
        {
            var books = RavenSession.Query<Book>()
                .Where(x => x.Price <= price) #C
                .ToList();

            return View("List", books);
        }
```

#A – Filters Book documents so only Books from the given year are returned
#B – Returns only books containing the specified department in their departments array
#C – Returns Books with price lower or equals to the price specified

This is the last of the screens for us to write – we now have all basic functionality done and ready. We can add books, browse the catalogue efficiently using paging and sorting, and also filter books from the listing. Our website admins can also go and edit or delete a book if they need to.

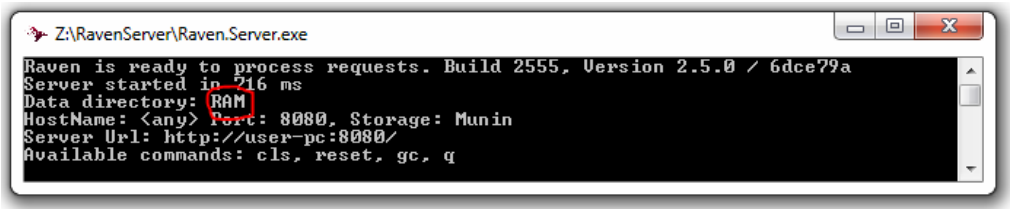
2.4 Deployment and Testing

Congratulations! You have successfully built your first RavenDB application. Now, to complete the picture, we will briefly touch on a couple of topics to wrap up the experience: Deployment options and testing. Both of these topics we will dive into later in the book, but we wanted to give you a quick introduction here so you can see a full application from start to finish.

Our online bookstore is now complete. We have screens to add and manage data, and screens to show it to the user in a way that is easy to understand and navigate. So now what? Well, to make sure everything is working we need to test it and then get it into the world by deploying it somewhere.

2.4.1 Deployment options

The RavenDB server can run in several modes. Where in Chapter 1 when we wanted to run a local RavenDB server and use it for prototyping and debugging purposes you saw how to run RavenDB in the Debug Console mode, where all requests are displayed in an interactive mode, and we had the option of running it completely in-memory. It looked like this:



For running our online bookstore in production environment this may not be the best option since we need the server to run in a more robust way. We also want to make sure it absolutely does not run in-memory but stores everything on disk.

Take a look at table 2.x for a list of deployment options. From this we can see that the best options for our online bookstore are running as a service or through IIS, because then it is properly managed as a service and being brought back up if the server goes down for some reason.

Following is a list of all possible ways to run the RavenDB server in, and how to use them. All server options require .NET 4.0 to be installed on the machine.

Mode	Description	How to run
Debug Console	Runs as a Console window, showing all incoming requests. Useful for debugging or running ad-hoc. Supports the /ram flag to run in-memory.	Extract the /Server folder from the distribution package, and run Raven.Server.exe
Windows service	Runs as an automatic Windows service, listening to HTTP traffic on the designated port. Suitable for installing on servers. Doesn't support HTTPS.	Extract the /Server folder from the distribution package, and run Raven.Server.exe /install. This will install RavenDB as a Windows service. To uninstall run Raven.Server.exe /uninstall.
IIS	Uses IIS for managing HTTP traffic, so HTTPS is supported.	Extract the /Web folder from the distribution package, and create a new website in IIS pointing to this folder.
Embedded	Used for embedding RavenDB in a .NET application. HTTP traffic can be enabled. See Embedded under "clients" below.	
Cloud	RavenDB database instance on the cloud for hire	http://ravenhq.com

www.dbooks.org

By running RavenDB in our production environment as a Windows service or hosted within IIS we make sure it is run robustly and properly taken care of in case of server failures. This would be our way to go for running a RavenDB server to be used by our online websore.

RavenDB also has multiple clients targeting different environments and use cases. Until now we used the lightweight .NET Client API, which can be used by any .NET application. There are several more options for clients worth mentioning, in case you wanted to work with it using other technologies like Silverlight:

Client	Description	How to use
.NET Client API	The standard client API, supports .NET 4.0 onwards. It is a lightweight client, implementing communication with an external RavenDB server. Can be used from any .NET application, and also from Mono.	Nuget: Install-Package RavenDB.Client Alternatively, reference the DLLs from /Client in the distribution package.
Silverlight	Silverlight client, allows for accessing RavenDB from Silverlight applications.	Reference the DLLs from the /Silverlight folder in the distribution package (for Silverlight 5; use the /Silverlight4 folder for Silverlight 4 applications).
Embedded	The Embedded client is a complete RavenDB implementation which could be embedded in any .NET application to provide data persistence within the application process without requiring an external server. Useful for disconnected systems and testing.	Nuget: Install-Package RavenDB.Embedded Alternatively, reference the DLLs from /EmbeddedClient in the distribution package.

Now that we have our deployment environment set, let's setup a few tests to demonstrate the testing capabilities of RavenDB. In real-world we will probably write and run tests during the entire development process, but for our purpose in this chapter we only want to discuss RavenDB's capabilities in that area.

2.4.2 Testing

RavenDB's ability to run completely in-memory makes applications built with it extremely easy to test. An `EmbeddableDocumentStore`, which can be created using the Embedded RavenDB client version, can be set to run completely in memory and by that making the whole process of populating it with test data, running a test and clearing it to make it ready for the next test very easy.

To try this yourself with the application we've just built, install The Embedded client from the Tests project that was created along with the web application project. The easiest way to do that is using nuget, by installing the RavenDB.Embedded package.

The next steps are to initialize a new `EmbeddableDocumentStore` instance, populate it with some test data and then run the tests. We show how to do that in the example in listing 2.x, of a simple test showing how to test the ListByPriceLimit method.

To run this test, go to the Tests project in the solution you created for the online bookstore we just built, and create a new class called BooksControllerTests. In it add the method below, and run it by right clicking in the editor in Visual Studio and clicking Run Tests from the context menu.

Listing 2.x: An example of test method

```
[TestMethod]
public void ReturnsBooksByPriceLimit()
{
using (var docStore = new EmbeddableDocumentStore { RunInMemory = true } #A
.Initialize() #B
                )
{
                using (var session = docStore.OpenSession()) #C
                {
                    // Store test data
                    session.Store(new Book { Title = "Test book",
YearPublished = 2013, Price = 12.99 });
                    session.SaveChanges();
                }

                var controller = new BooksController { RavenSession =
docStore.OpenSession() }; #D

                var viewResult = (ViewResult)controller.ListByPriceLimit(15);
#E
                var result = viewResult.ViewData.Model as List<Book>;
                Assert.IsNotNull(result);
                Assert.AreEqual(1, result.Count);

                viewResult = (ViewResult)controller.ListByPriceLimit(10); #F
                result = viewResult.ViewData.Model as List<Book>;
                Assert.IsNotNull(result);
                Assert.AreEqual(0, result.Count);

                controller.RavenSession.Dispose(); #G
} #H
}
```

#A – Creating a new embedded document store, and specifying it should run completely in memory.
#B – Initializing the document store. Don't forget this or nothing would work!
#C – Creating a session to store test data, storing the test data and calling SaveChanges() in the end to persist it
#D – Creating an instance of the Controller we are going to test, and injecting it with a DocumentSession object created using our in-memory database.
#E – First test, to verify results are returned with a high price limit
#F – Second test, to verify results are not returned by a low price limit

www.dbooks.org

#G – Dispose of the session once we are done with it
#H – The in-memory document store is going to be disposed of here, in the end of this using block.

This is testing RavenDB's capabilities more than the application logic. Real tests would grow more complex and have more test data in them, but this feature of RavenDB will really make the experience of writing tests for your apps much more pleasant, because instead of mocking and doing various ninja-moves to make our tests work, we can just run RavenDB in its whole in-memory.

2.4.3 Handling Model changes

Our online store is ready to go but it seems our client forgot to mention he wants to have the cover image for the book, and we already went live with the website. He is on the phone and it doesn't seem we can get away by saying we will add it in the next version. They want it in ASAP.

As you may have already experienced yourself, this one of the things that always happen. The requirement for changing the data model will always pop up last minute, or even after going live. No matter the amount of efforts you put to make sure this doesn't happen it will surely happen.

Fortunately for us we are using RavenDB. Since RavenDB doesn't have a schema we don't have to maintain one, and thanks to the .NET client API this is a fairly easy change to make. All we have to do is add a property to the Book class to store the URL for the book image, and update the relevant screens accordingly. The RavenDB client API will do the rest for us – new documents will be saved using the new structure, and existing documents will get updated with the new property as they are loaded over time.

```
    public class Book
    {
        public string Id { get; set; }
        public string Title { get; set; }
        public string Author { get; set; }
        public string Description { get; set; }
        public double Price { get; set; }
        public int YearPublished { get; set; }
        public List<string> Departments { get; set; }
        public List<CustomerReview> CustomerReviews { get; set; }
        public string CoverImageURL { get; set; }
}
```

This migration operation was a fairly easy one. New documents will gradually get the new document property we added, although it will be a null value which you might need to account for.

Due to the nature of the database, many of the common migrations are like this. However, in some cases we will need to do a bit more work. We will cover migrations in more details in chapter 12.

2.5 Summary

In this chapter we took our first shot at writing a RavenDB-backed application. We practiced setting up RavenDB and then doing actual persistence work against it. We learned about how to do session management in ASP.NET MVC web applications, and finished by looking at how testing can be done easily.

By now you should be confident enough with how to work with RavenDB, and appreciate the amount of work done by the session object – by serialization and deserialization, automatic change tracking and the comprehensive querying language.

The practices we learned in this chapter are not at all MVC specific. Everything we learned could be applied to any .NET application, web-based or desktop. The basic concepts of session management are just the same – have only one global `DocumentStore` instance available, and one `DocumentSession` object for each work context.

This chapter concludes the introduction part of the book. Starting with the next chapter we are going to dive deeper and deeper into RavenDB and learn more and more on what's happening under the hood, and how to make good use of it.

www.dbooks.org

3

RavenDB indexes

This chapter covers

- The indexing process
- Creating an index and querying it
- Map-only and Map/Reduce indexes

So far we've been operating on the assumption that querying with RavenDB "just works." Although this still holds true, starting with this chapter we're going to explain exactly how it works.

We don't need to go far to come up with an example to demonstrate what a crucial part indexes play in answering queries. This is exactly like going to a library with millions of books and trying to find one book on the thousands of shelves. And when you find that book, you must find the one page out of the hundreds of pages it contains that mentions the topic you're interested in. If the library has a decent catalog, this will take a matter of minutes. Otherwise, you may spend a whole weekend there before getting to the right book.

Without indexes it's impossible to find data on anything other than the document ID in RavenDB, and therefore RavenDB becomes just a bloated key/value store. Indexes are the piece of the puzzle that allows rich queries for data to be satisfied efficiently.

Luckily, RavenDB automatically creates and manages indexes for most common queries, but for some cases you'll have to create indexes yourself. Understanding the query life cycle, when creating indexes manually is required and how those indexes are structured, is going to help you with building complex software with RavenDB.

You'll start by becoming familiar with the Index Store, which is where the indexes are persisted, and with the indexing process. Then you'll learn how and when indexes are created and how they play their role in answering user queries.

3.1 The Index Store

Being a document database, RavenDB has dedicated storage for documents—where documents are stored and pulled from when accessed. This is the heart of RavenDB and what we call the Documents Store. When you stored and updated documents in the previous chapter, you were working directly against the Documents Store.

The Documents Store has one important feature: it's very efficient in pulling documents out by their ID. But this is also its only feature and the only way it can find documents. It can have only one key for a document, and that key is always the document ID; documents can't be retrieved based on any other criteria.

When you need to pull documents out of the Documents Store based on some search criteria other than their ID, the Documents Store itself is useless. To be able to retrieve documents using some other properties they have, you need indexes. Those indexes are stored separately from the documents themselves—in what we call the Index Store.

Starting with the following sections and on into the next chapter, you're going to learn everything there is to know about indexes in RavenDB: how and when they're created, what they look like, how to create them yourself, and the times you want to do that and not rely on automatic index creation.

In this section you'll learn about indexes and the indexing process in RavenDB. It's important to understand what it is and why it's needed before actually using it.

3.1.1 The RavenDB indexing process

Let's assume for one moment all you have in your database is the Documents Store. In it are a couple million documents, and now you get a user query you need to answer. The Documents Store by itself can't help you, because the query doesn't contain the document IDs. What do you do now?

One option is to go through all the documents in the system and check them to see if they match the query. This will work, sure, if the user who issued the query is kind enough to wait for a few hours in a large system. But no user is. In order to efficiently satisfy user queries you need to have your data indexed. By using indexes the software can perform searches much more efficiently and complete queries much faster.

Before we look at those indexes, let's consider for a moment when are they going to be built or updated with the new documents that come in. If you calculate them when the user issues the query, you delay returning the results. This is going to be much less substantial than going over all the documents, but you still incur a performance hit to the user for every query they make.

Another perhaps more sensible option is to update the indexes when the user puts in the new documents. This makes more sense at first, but then when you start to consider what it would take to update several complex indexes on every put, it becomes much less attractive. In real systems, this means writes would take quite a lot of time, because now not only is the document being written, but all indexes have to be updated as well. There's also the question

www.dbooks.org

of transactions—what happens when a failure occurs while the indexes are being updated; should it fail a transaction? And how do you handle concurrent writes?

With RavenDB, a conscious design decision was made to not cause any wait due to indexing. There should be no wait at all—never when you ask for data and never during other operations, like adding new documents to the store.

So when *are* indexes updated? RavenDB has a background process that's handed new documents and document updates as they come in, right after they were stored in the Documents Store, and it passes them in batches through all the indexes in the system. For write operations, the user gets an immediate confirmation on their transaction—even before the indexing process starts processing these updates—without waiting for indexing, but being 100% certain the changes were recorded in the database. Queries don't wait for indexing either; they just use the indexes that exist at the time the query is issued. This ensures both smooth operation on all fronts and that no documents are left behind. This is shown in figure 3.1.

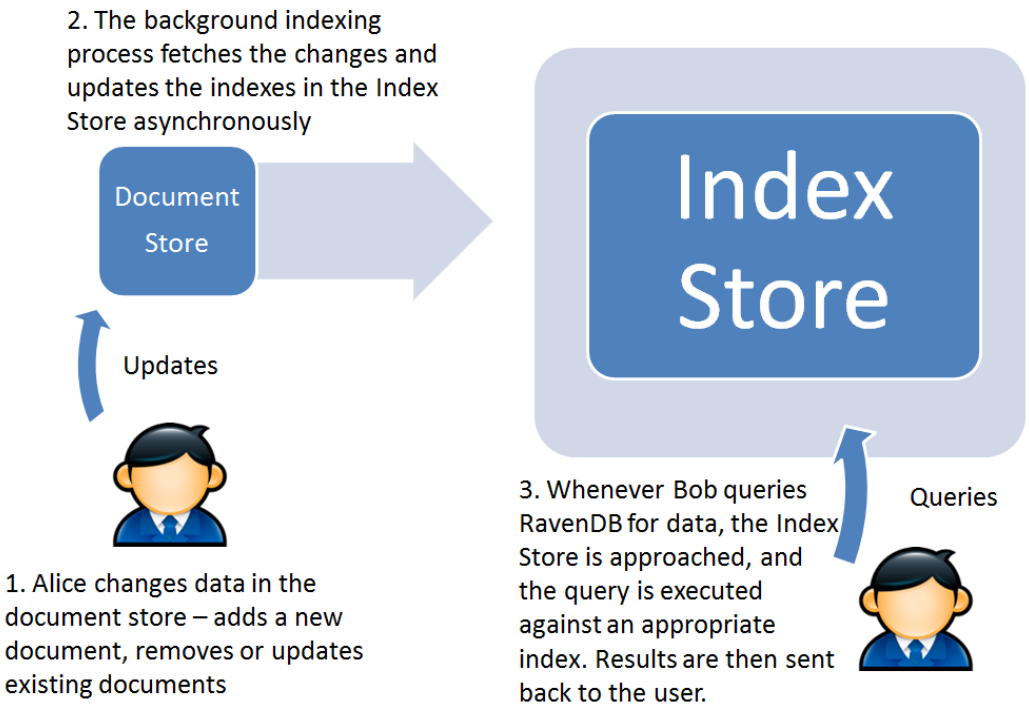
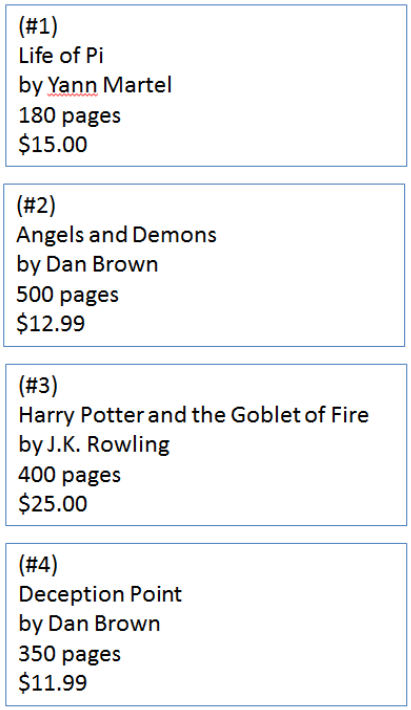


Figure 3.1 RavenDB's background indexing process doesn't affect response time for either updates or queries.

It all sounds suspiciously good, doesn't it? Obviously, there's a catch. Because indexing is done in the background, when enough data comes in, that process can take a while to complete. This means it may take a while for new documents to appear in query results. Although RavenDB is highly optimized to minimize such cases, it can still happen, and when this happens we say the index results are *stale*. This is by design, and we discuss the implications of that at the end of this section.

3.1.2 What is an index?

Consider the list of books shown in figure 3.2.



If I asked you what was the price of the book written by J.K. Rowling or to name all the books with more than 600 pages in them, how would you find the answer to that? Obviously, going through the entire list is not too cumbersome when there are only 10 books in it, but it becomes a problem rather quickly as the list grows.

An index is just a way to help you answer such questions more quickly. It's all about making a list of all possible values grouped by their context and ordering it alphabetically. As a result, the previous list of books becomes the lists of values shown in figure 3.3, each value accompanied by the book number it was taken from.

www.dbooks.org

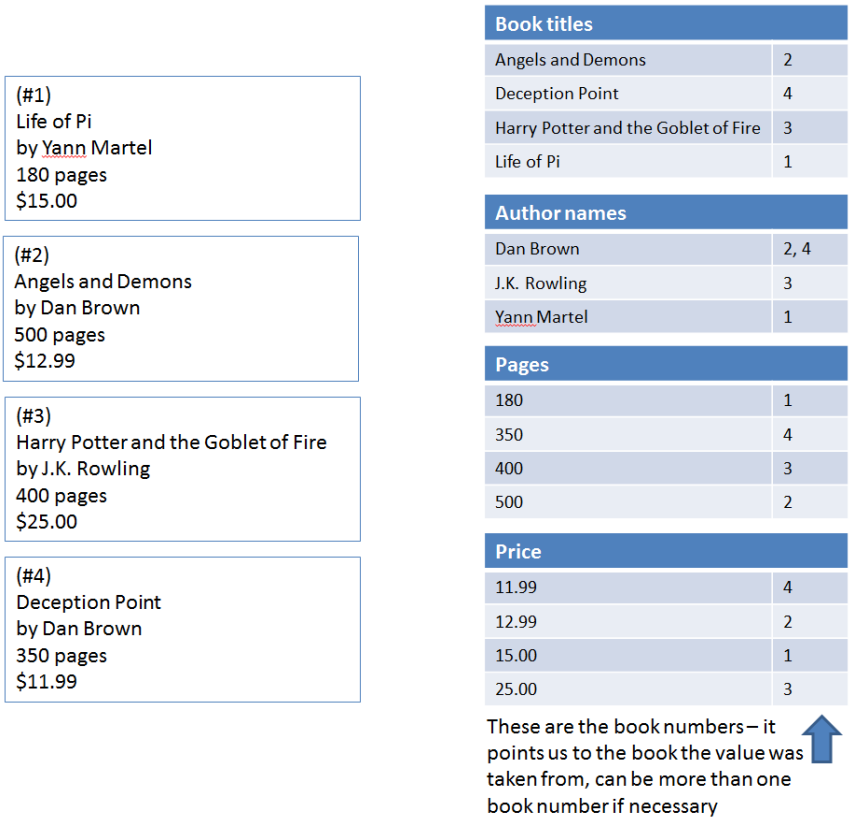


Figure 3.3 A list of books (left) and lists of all possible searchable values, grouped by context

Because the values are grouped by context (a title, an author name, and so on) and are sorted lexicographically, it's now rather easy to find a book by any of those values even if you have millions of them. You simply go to the appropriate list (say, Author Names) and look up the value; because the lists are lexicographically sorted, this can be done rather efficiently. Once the value is found in the list, the book number that's associated with it is returned and can be used to get the actual book if you need more information on it.

The process of creating an index like that is called *indexing*. RavenDB uses Lucene.NET as its indexing mechanism. Lucene.NET is the .NET port of the popular open-source search engine library Lucene. Originally written in Java and first released in 2000, Lucene is the leading open-source search engine library. It's being used by big sites like Twitter, LinkedIn, and more to make their content searchable and is constantly being improved to be made faster and better.

3.1.3 A Lucene index

Because RavenDB indexes are in fact Lucene indexes, before you go any deeper into RavenDB indexes you need to familiarize yourself with some Lucene concepts. This will help you understand how things work under the hood and allow you to work better with RavenDB indexes.

In Lucene, the base entity that's being indexed is called a *document*, and every search yields a list of matching documents. In this example you're searching for books, so each book would be a Lucene document. Just like books have a title, author, name, page count, and so on, oftentimes you need to be able to search for documents with more than one piece of data taken from each. To this end, every document in Lucene has the notion of *fields*, which are just a logical separation between different parts of the document you indexed. Each field in a Lucene document can contain different pieces of information about the document, which you can later use to search on.

To apply those concepts to our books example, in Lucene each book would be a document, and each such book document would have the fields Title, Author Name, Price, and Pages Count. Lucene will create an index with several lists of values, one for each field, just like shown in figure 3.4. To complete the picture, each value that's put in the index in one of those lists (for example, Dan Brown from two books in the Author Name field) is called a *term*.

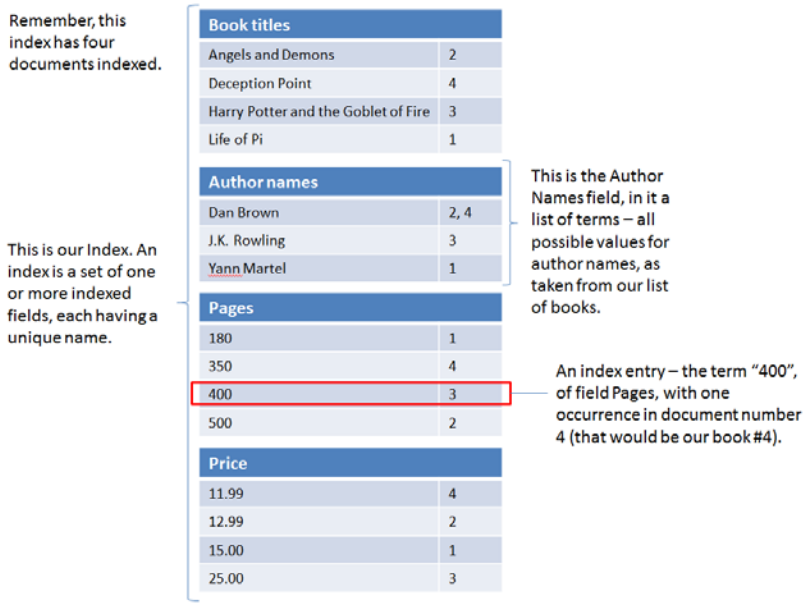


Figure 3.4 The structure of a typical Lucene index

www.dbooks.org

Searches are made with terms and field names to find matching documents, where a document is considered a match if it has the specified terms in the searched fields, like the following pseudo-query: all Book documents with the Author field having a value of Dan Brown. Lucene allows querying with multiple clauses on the same field or even on different fields, so queries like “all books with author Dan Brown or J.K. Rowling, and with price lower than 50 bucks” are fully supported.

An index in RavenDB is just a standard Lucene index. Every RavenDB document from the Documents Store can be indexed by creating a Lucene document from it. A field in that Lucene document is then going to be a searchable part of the RavenDB document you’re indexing—for example, the title of a blog post, the actual content, and the posting date will each be a field.

We’ll discuss the way Lucene works in more depth in chapter 7, but what you need to remember now is that queries are made against one index, on one field or more, using one term or more per field. In the next section you’ll see exactly how that falls into place.

The indexing process in which Lucene documents are created from documents stored in RavenDB—from raw structured JSON to a flat structure of documents and fields—uses two types of functions referred to as Map and Reduce, which are declared for every index (where the Reduce function is optional). Starting in the next section we’ll go through RavenDB’s Map/Reduce process and work our way to properly grokking it.

3.1.4 Eventual consistency

Before we have a look at actual indexes, allow me to pause for a minute or two to discuss the implications of indexing asynchronously. As I explained in the beginning of the chapter, indexing in RavenDB happens in the background, so on a busy system new data may take a while to appear in query results. Databases behaving this way are said to be *eventually consistent*, meaning at some point in time new data is guaranteed to appear in queries but it isn’t guaranteed to happen immediately.

At a first look getting stale results for queries doesn’t seem all that attractive. Why would you want to work with a database that doesn’t always answer queries correctly?

This is mostly because you’re used to databases that are implemented with the ACID properties in mind—atomicity, consistency, isolation, and durability. In particular, relational databases are ACID and always guarantee consistent results. In this context that means every query you send will always return with the most up-to-date results—they are *immediately consistent*. If a document exists in the database when the query is issued, you are guaranteed it will be returned in all matching queries. But is that really required?

> **IMMEDIATE OR EVENTUAL?** The Documents Store is immediately consistent; queries are eventually consistent. With RavenDB, eventual consistency is the case only when querying. Loading a document by ID is always immediately consistent and fully ACID compatible.

Even when results are known to be 100% accurate and never stale like they are in any SQL database, during the time it takes the data to get from the server to the user’s screen, plus the time it takes the user to read and process the data and then to act on it, data could have

changed on the server without the user ever knowing. When there is high network latency or caching involved, it's even more likely to be the case. And what if the user went to get coffee after asking for that page of data? In the real world, when it comes down to business, most query results are stale or should be thought of as such.

Although your first instinct is to resist the idea, when it actually happens to you, you don't fight it and usually even ignore it. Take Amazon, for example: having an item in your cart doesn't ensure you can buy it. It can still be out of stock by the time you check out. It can even be out of stock *after* you checked out and paid, in which case Amazon's customer relations department will be happy to refund your purchase and even give you a free gift card for the inconvenience.

Does that mean Amazon is irresponsible? No. Does that mean you were cheated? Definitely not. It's just about the only way they could efficiently track stock in such a huge system, and we as users almost never see this happen.

Now, think about your own system and how up to date your data should be on display. Could you accept query results showing data that is 100% accurate as of a few milliseconds ago? Probably so. What about half a second? one second? five seconds? one minute?

If consistency is really important, you wouldn't accept even the smallest gap, and if you can accept some staleness, you could probably live with some more. The more you think about it, the more you come to realize that it makes sense to embrace it rather than fight it.

At the end of the day no business will refuse a transaction even if it was made by email or fax, and meanwhile stock data have changed. Every customer counts, and the worst that could happen is an apology. And that's what stale query results are all about.

A lot of work has been done making sure the periods in which indexes are stale are as minimal as possible. Thanks to many optimizations, most stale indexes you'll see are new indexes that were created in a live system with a lot of data in it or when there are many indexes and a lot of new data keeps coming in consistently. In most installations indexes will be non-stale most of the time, and you can safely ignore the fact that they're still catching up when they are indeed stale. For when it's absolutely necessary to account for stale results, RavenDB provides a way to know when the results are stale and also to wait until a query returns non-stale results. We cover this at the end of chapter 4.

3.2 Simple Map indexes

In the previous section we discussed the Index Store and how the indexing process works. We explained what indexes are and the process in which they're built and updated. As you've seen, without an index, you can't pull documents on anything other than their ID.

Your goal in this section and throughout the rest of this chapter is to understand the actual constructs that make up an index and the way they operate on data. All indexes share the same basic structure; it doesn't matter if they were created manually or automatically, so what you're going to learn about indexes applies to both auto and manual indexes.

You start by realizing RavenDB is a schema-free database, and as such it has no knowledge of the structure of the documents it contains. When new data comes in, all existing indexes get

www.dbooks.org

notified of the new entries and are given the chance to process them. But how can hierarchical data in raw JSON format be crunched and transformed into a useful index entry?

This is the responsibility of the Map function, which is the most fundamental piece of an index. All indexes have a Map function, and it's the first operation to run on every document that is passed to the index.

3.2.1 The Map function

In every document there are different pieces of data—some important and some less so. In every document there are also the pieces of data you will be using to retrieve them by (that is, query) and the pieces you'll be interested in having stored but probably won't query with.

For example, if you were to build an online book store, for each book you would want to store data such as its price, title, author name, and so on. You may also want to store additional pieces of data to display on your website, like a short description and an image. The first group of properties is definitely going to be used quite often in queries, for example, to get all books by a certain author or within a price range. But you probably wouldn't want to allow queries on the book's description or on its image.

That's where the Map function comes in. Its main responsibility is to extract pieces of data you'll be willing to search on from a document. Those pieces of data, often referred to as indexed document attributes, are being saved to an index and are what makes the document queryable. Because this data-extraction process is being made with a Map function, we often refer to this process as mapping.

Using the bookstore example, mapping a document to an index entry would look something like figure 3.5. The result is just a Lucene index with four fields, exactly like the one you saw earlier in the chapter. Once you extract the values from the JSON document stored in RavenDB and put index entries for it in the respective fields, it becomes something you can query.

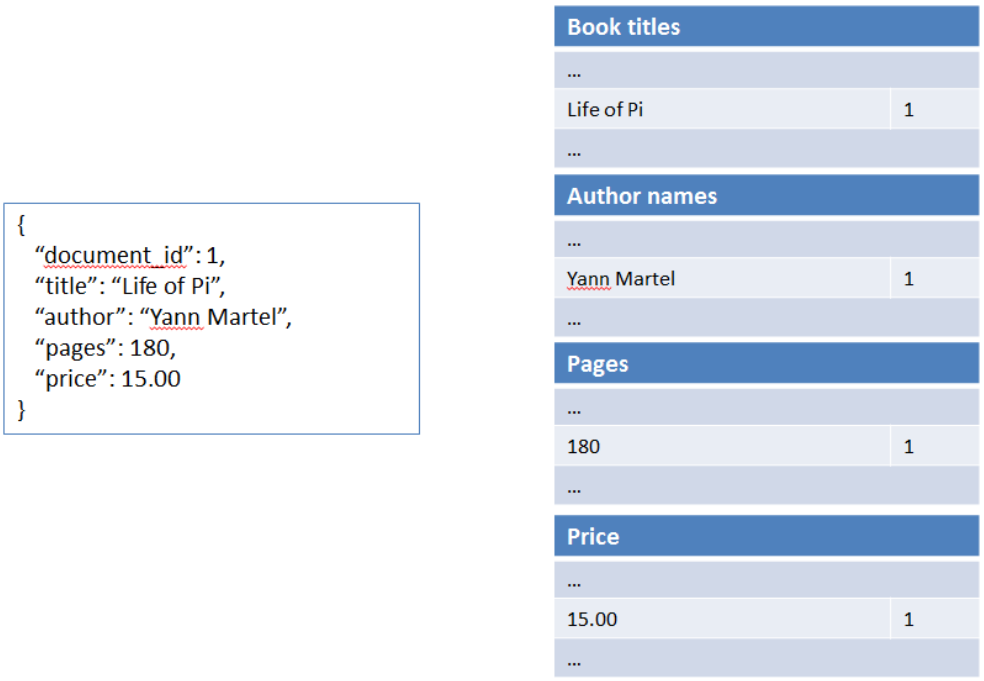


Figure 3.5 A RavenDB document (left) and the result of mapping it to a Lucene index (right)

In RavenDB, LINQ is used to express this data extraction, or mapping. This mapping for the bookstore scenario would produce results like those shown in figure 3.5. A Map function would look like this:

```
from doc in docs.Books #A
select new { Title = doc.Title, Author = doc.Author, Pages = doc.Pages, Price
= doc.Price } #B
```

#A `docs` represents the entire collection of documents stored in the database, and `docs.Books` tells RavenDB to look at only documents in the Books collection.
#B The Map function outputs an anonymous object with properties as defined within the curly brackets.

This tells RavenDB to look only at documents of the Books collection and to map the properties Title, Author, Pages, and Price from every document it's given to the index with those respective names. It's important to note that at this stage all RavenDB sees is a JSON document, and the Map function operates on it alone. You can think of it as a LINQ to JSON type of thing. If those documents were created using the .NET Client API by serializing an object of class `Book`, RavenDB doesn't have those types available while indexing, so you can't call any methods specific to them in the Map function.

www.dbooks.org

NOTE In LINQ, specifying the field name explicitly is redundant if the field name matches the document property.

The output of the Map function is a set of anonymous objects with a flat structure—in this case an object with fields Title, Author, Pages, and Price. Each of those objects is going to be put in the index, each of its fields in a dedicated field in Lucene, as you saw previously in Figure 3.5.

No matter the type or the format, the value that will be indexed matches exactly the original value. So if, for example, the book title contains spaces, when querying for it you will have to type it exactly as it appears in the original document, including spaces.

It does make sense when you think of it; it can't be RavenDB's responsibility to perform data processing from the way it appears in the document to the way it is indexed, since it has no knowledge of what that data actually is. Therefore, it doesn't make any assumptions and just stores the value as is. If a different behavior is desired, it's up to the user to specify that explicitly. In the next chapter we're going to look at index field options that allow doing just that, and in chapter 7 you'll learn how to enable full-text search on fields.

There is one exception, though; null values or missing document attributes are being handled gracefully. You don't have to worry about writing a Map function to a type that may not have data or an attribute you're selecting to a project. RavenDB will ignore missing attributes referred to in a Map function and treat null values as actual values, so no exception will be thrown.

As we mentioned already, all indexes have a mapping phase and at least one Map function. Some indexes have only a mapping phase, and some have additional phases, which you'll get familiar with later. RavenDB guarantees that all documents in the system will go through all indexes and by that ensures that all data can be made queryable, as long as there's an index to map it.

The mapping function doesn't only allow for filtering data; it can also help shape the data prior to indexing. We'll be looking at such techniques later in the book. But it's important to get the Map function right, to make sure your indexes are efficient and perform their duty well. Updating a Map function of an index will trigger an index rebuild, which may be very costly.

3.2.2 A simple example of an index

Simple map indexes, or map-only indexes, are as their name suggests indexes with only a Map function. Although this type of index is the simplest form, it's the most common to have: there will usually be at least one such index per collection mapping all queryable fields from documents in the collection.

We'll assume you have a collection of Book documents for all books in your bookstore, each with the same properties defined previously. Let's jump right in and define a simple Map index that will let you query for Book documents in your database.

Access Management Studio by going to the HTTP endpoint the server is listening on (the default for a local running server is http://localhost:8080). If you go to the Indexes tab and click Add New Index, you'll get a chance to add a new index.

All you need to do now is type in the Map function, give the index a proper name, and click Save, as shown in figure 3.6.

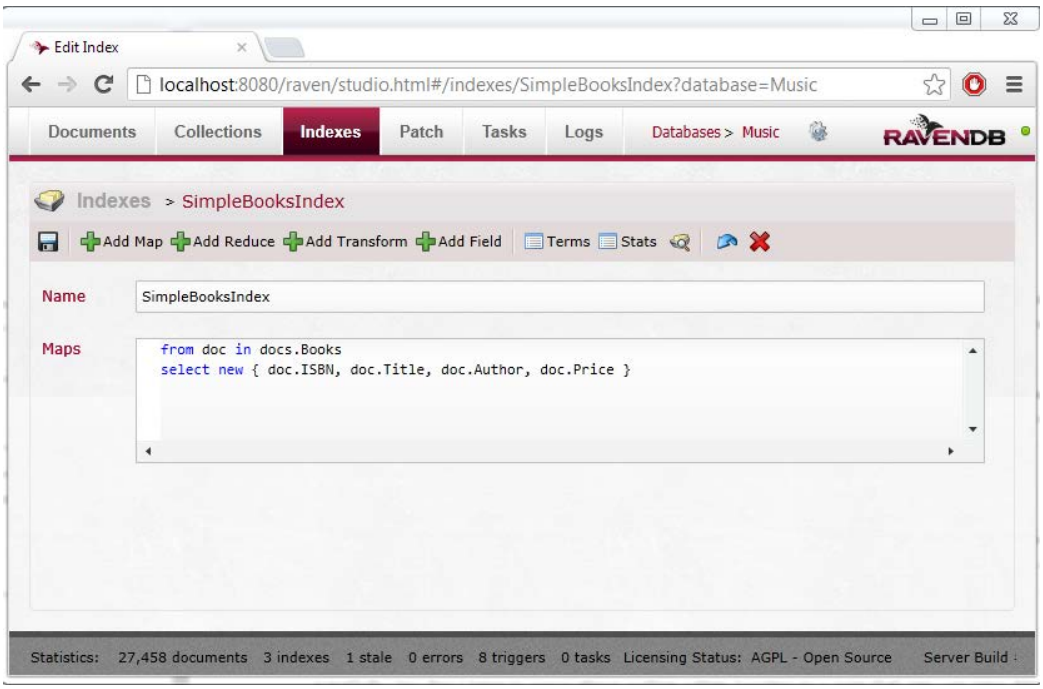


Figure 3.6 Creating a simple Map index

Once you click Save, the RavenDB database will have a new index created, at this point with no documents indexed in it. As you saw earlier in the chapter, the background indexing process will now feed the index all the documents stored in the database. The index will then filter out those that are not part of the Books collection. For documents that are of a Book type, the properties you marked for indexing using the Map function will be extracted from the document and saved into the Lucene index, ready to be queried on.

As you might have noticed, this is all pretty much straightforward. Simple Map indexes require little thought, other than what properties of the document need to be searchable. Most of the time you won't need to write simple Map indexes at all, thanks to RavenDB's capability to create indexes when needed.

3.2.3 Querying indexes

Because under the hood RavenDB indexes are Lucene indexes, the Lucene Query Language can be used to query them. This is exactly what RavenDB does for LINQ queries issued from the .NET Client API.

www.dbooks.org

In chapter 2 you used the client API to perform basic querying and pulled documents out of the RavenDB server using some search criteria. One simple query you issued looked like this:

```
public ActionResult ListByYear(int year)
{
    var books = RavenSession.Query<Book>()
        .Where(x => x.YearPublished == year) #A
        .ToList();

    return View("List", books);
}
```

Behind the scenes, the client API translated the LINQ expression you wrote as a query to a Lucene query syntax and sent it over the wire to the server, where an appropriate index to query was selected and then served with that query.

You can use the Lucene Query Language to query indexes directly via Management Studio, by going to the Indexes tab and clicking the index to query. Let's query the index you just created to try to find all books with price between 10 to 20 bucks, whose author is Dan Brown, as shown in figure 3.7.

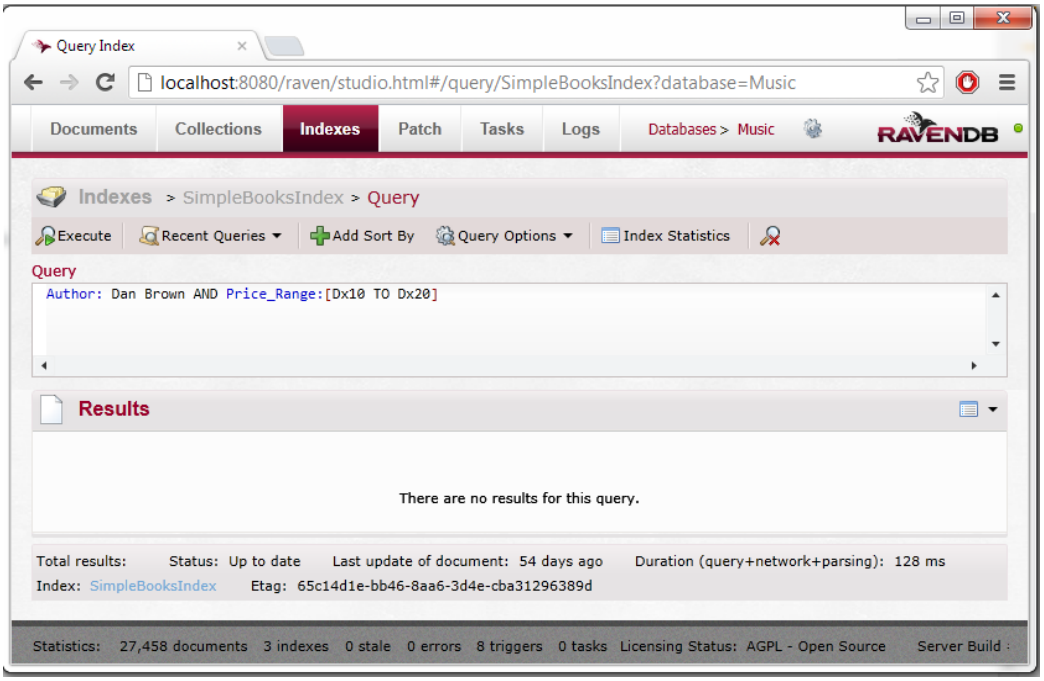


Figure 3.7 Querying an index with the Lucene query language using Management Studio

You may be new to the Lucene Query Language, but you should find this query syntax familiar from your daily usage of Google. The Lucene query syntax is simple and intuitive and can be summarized as shown in figure 3.8.

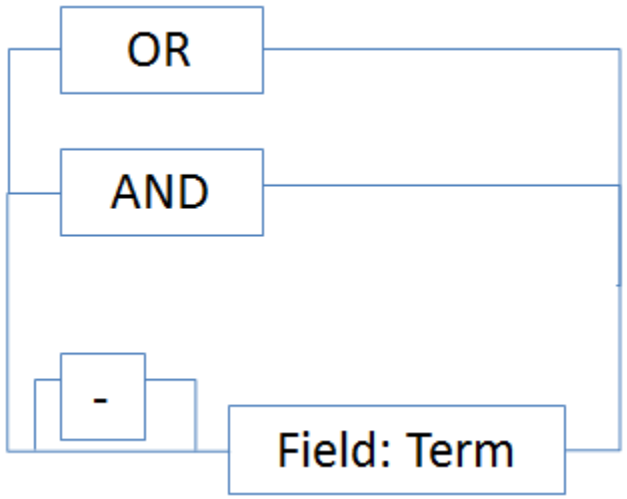


Figure 3.8 The structure of a Lucene query

Field is the name of the property from the anonymous object you returned from your Map function, and Term is the value or range of values you're trying to match in that field. You can have multiple clauses like this in one query and relate them with the OR/AND operators and negate a clause by prepending a minus (-). One clause can be created by several smaller clauses by grouping them with parentheses.

In this case you're querying for an Author name of Dan Brown and for a range on the Price field, which is known to be of type Double, so your query looks like the one in figure 3.9.

www.dbooks.org

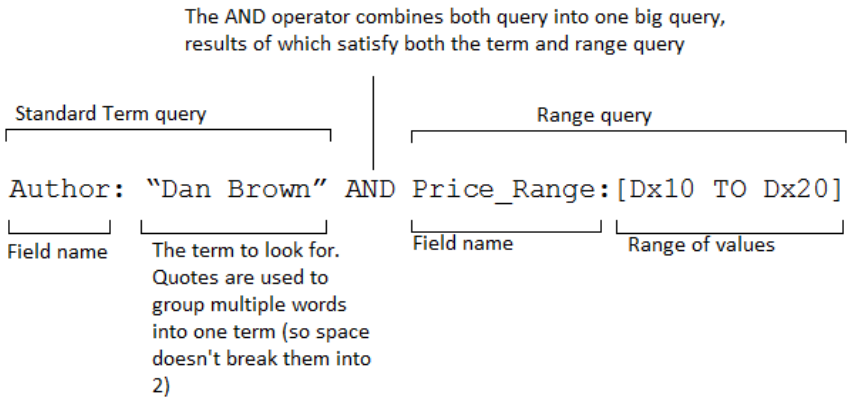


Figure 3.9 Querying for author Dan Brown and a price range

The Range clause is the only specialized case of a term in query. With it, the term is the range specification in the syntax showed in the figure—from what value to what value (NULL in either edge to specify no bound). Ranges can be made on any field type, but for numeric fields the field name has to have added a suffix of _Range, and the actual values in the range need to specify their type: Dx prefix for Double, Lx for Long, Ix for Integer, Fx for Float. Hexadecimal representation of integer values is supported as well and should be prefixed with 0x.

Once a query is issued, the results will appear below the query pane in Management Studio. In the next chapter we'll look at querying indexes from .NET applications.

3.2.4 Peeking into the index

Let's pause for a moment to have a look at what's happening on the index level. You just created an index with a Map function that maps four document properties to it: ISBN, Price, Title, and Author. If you query the index again, click the down arrow beside Query Options, and check Index Entries, you'll be provided with the actual data from the index instead of the documents themselves, as shown in figure 3.10.

As expected, the index contains all the fields you mapped to it. There are two additional things that you should notice. The first is the additional __document_id field—for each index entry, RavenDB will store the document ID from where that entry originated, even if it wasn't in the Map function. This is because you need to know what documents to load after a successful query.

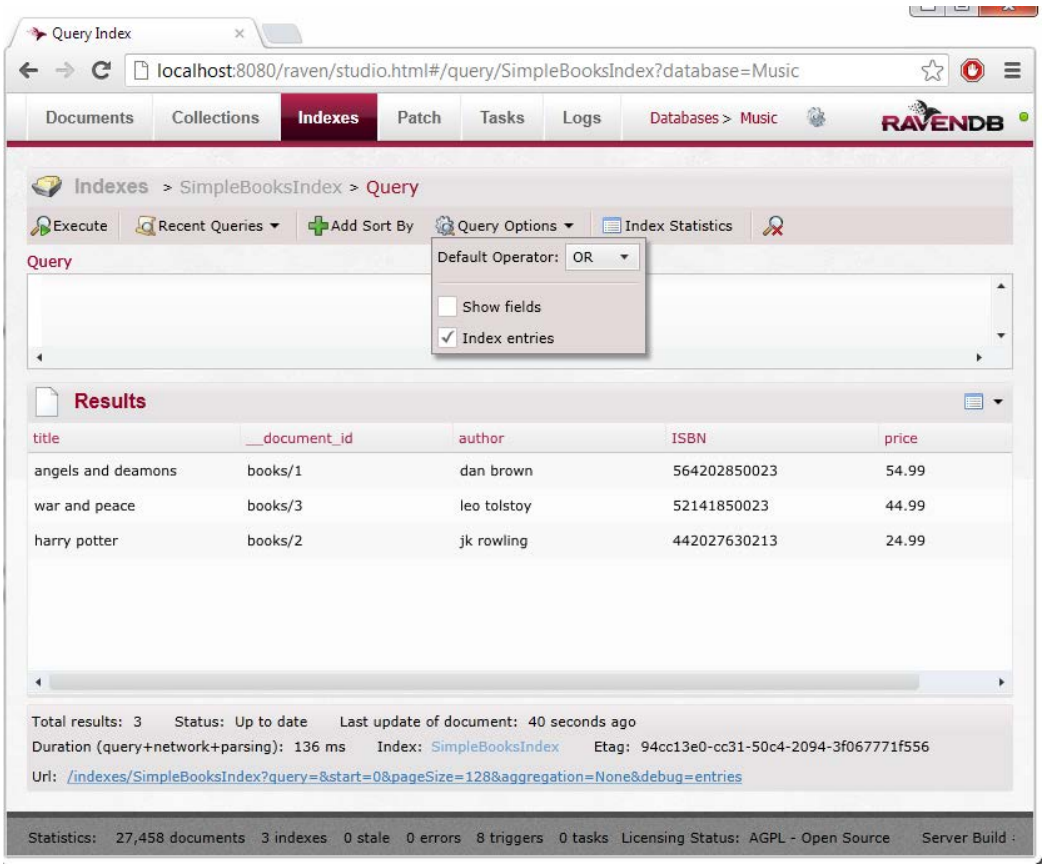


Figure 3.10 Issuing a query and asking to show index entries will return matching documents as they appear in the index.

The other thing you might notice is that all the string values have been lowercased. This is by design to make sure string queries are not case sensitive. This default behavior can be changed, and we will discuss this in the next chapter, under “Index fields options.”

This Index Entries feature is useful for debugging indexes when you need to see what was actually indexed and how it looks, so you know your indexes are working as expected.

3.2.5 Mapping collections

So far you’ve looked at fairly simple documents, with no lists or arrays, and only had to write fairly easy mappings. Let’s throw an array into the mix, to make things more interesting.

Next you’ll add a list of tags to your Book documents, so your customers can find new books more easily. A Book document of the new version would look like this:

www.dbooks.org

```
{
    "ISBN": "564202850023",
    "title": "Angels and Demons",
    "author": "Dan Brown",
    "price": 54.99,
    "tags": [
        "Romance",
        "Fiction",
        "Bestseller"
    ]
}
```

Now also enable querying for books by a tag associated with them. For example, to get all romance books in the collection you'd issue the following query:

```
Tag: Romance
```

There are two ways you can change your Map function to index a collection like this. One way is to have the Map function iterate through all tags and output a new index entry per tag, like this:

```
from doc in docs.Books
from tag in doc.Tags
select new { doc.ISBN, doc.Title, doc.Author, doc.Price, tag }
```

With this Map function, your index would look like the one in figure 3.11.

title	__document_id	author	ISBN	price
angels and deamons	books/1	dan brown	564202850023	54.99
angels and deamons	books/1	dan brown	564202850023	54.99
angels and deamons	books/1	dan brown	564202850023	54.99
war and peace	books/3	leo tolstoy	52141850023	44.99
war and peace	books/3	leo tolstoy	52141850023	44.99
harry potter	books/2	jk rowling	442027630213	24.99

Figure 3.11 Revised index

You have an index entry for each tag of each book, with all the book data replicated between index entries of the same book. This is the simple way of doing it and is quite expected.

There's another way to do this, which is more efficient in terms of index size, because it requires only one index entry per document:

```
from doc in docs.Books
select new { doc.ISBN, doc.Title, doc.Author, doc.Price, Tag = doc.Tags }
```

In this Map function you select the entire list of tags as a field. RavenDB is smart enough to handle this properly, resulting in the index structure shown in figure 3.12. Note how the Tags

field in the index has multiple terms associated with it, so querying for any of the tags a book has would yield a correct result.

title	__document_id	tags	author	ISBN
angels and deamons	books/1	["bestseller","fiction","romance"]	dan brown	564202850023
war and peace	books/3	["bestseller","novel"]	leo tolstoy	52141850023
harry potter	books/2	bestseller	jk rowling	442027630213

Figure 3.12 The Tags field has multiple terms.

Both approaches are valid, but it's common practice to prefer the latter approach, which, as we mentioned, results in a smaller index size and hence better performance.

3.3 Map/reduce indexes

So far you've been working with simple Map indexes, which help you get documents by querying on their attributes. They're extremely useful, but they can't answer some queries.

Suppose you wanted to look for data based on some computation, for example, finding all books from all authors who published more than five books or an author name for the author whose average book costs less than $50. Since you're no longer querying on values that exist in any of the fields of the documents but rather on the result of an aggregation operation made on them, simple Map indexes are pretty much useless.

> **Key takeaway: RavenDB queries never perform computations**
>
> Some databases will allow you to do that quite easily; for example, in SQL Server you could use the `COUNT` and `GROUP BY` operators. But such a query will perform computations when executed and RavenDB by design forbids such queries. RavenDB allows queries only on pre-indexed data, to ensure the queries always perform well.

To make this work you need to have the actual computation result in the index so you can query on it. To be able to answer a question like "all authors who have published five books or more," you need the index to contain the number of books published by each author, along with their names.

Map/Reduce indexes are the tool for the job. What they do is make the required computation—like count documents satisfying a certain condition—an extra indexing step on the mapped results before putting them into the index. The result of that computation is what's being put into the index, hence making it searchable.

Unlike simple Map indexes, Map/Reduce indexes are never created automatically and always require being created manually.

www.dbooks.org

3.3.1 What is Map/Reduce?

Map/Reduce is just a complex name for a simple two-phase operation for selecting and grouping data, which is defined by a pair of functions: a Map function and a Reduce function. This operation enables you to perform computations during indexing and then make the results accessible for searching on.

We'll dive right into the Map/Reduce process and follow up with a concrete example. The complete Map/Reduce process we're explaining now is shown in figure 3.13.

The first part of any Map/Reduce operation is the Map phase. It's exactly the same procedure as you saw in the previous section. Data is being extracted from documents using a Map function, but this time, instead of storing it in the index as you would do in a simple Map index, you make it go through an additional phase, called the Reduce phase.

The Reduce phase is where the aforementioned computation happens. It consists of grouping the data gathered by the Map phase into buckets based on some key, usually something involving data from the documents themselves, called the reduce key. Then, each bucket contains the documents matching the reduce key, and you can perform the aggregation operation on those buckets. The result is a result value for each bucket along with the value of bucket reduce key.

By writing the result of the Reduce phase to the index you can query on the precomputed results of that computation instead of computing it while querying.

Let's look at the example query "all authors who have published five books or more" again. Because you don't have the number of books written by each author in your book documents, or in any other document for that matter, you need to have a Map/Reduce index to answer this query. To do that, the index needs to contain all the author names along with the number of books they've written, so a query against the number of books will give you the author names.

The first step is to collect the data you have. You can use a Map function to iterate through all Book documents you have and list all the author names for which you have books. Because every selection of an author represents one book they've written, you'll assign that record an extra data count of 1:

```
from doc in docs.Books
select new { AuthorName = doc.Author, Count = 1 }
```

The result of the Map function won't be written to the index but will be stored as a mapped result in storage dedicated for this index. The output of the Map function in a Map/Reduce index is a smaller document, which contains a subset of the data from the original document, possibly with some additions—like the Count property in this case. The data in this subset either will be made searchable or will be required for the computation in the next step (or both).

The next step is to reduce all the records in the mapped-results storage for this index based on the reduce key. The reduce key is selected by the piece of data you want to group everything by. In this case, you want to count the occurrences of each author, so you group by the author name.

At this stage you have buckets containing the count of unique authors. Each bucket is marked with a unique value for the reduce key—basically an author name—and contains all the mapped results matching that value for the reduce key. You can now apply the computation on each bucket separately. Your computation would be to count the number of objects it has, which is done by aggregating the Count property of all mapped results in each bucket. Then you'd take the value for each bucket along with the computation result, and that would be the final result of your Map/Reduce index. The steps are illustrated in figure 3.13.

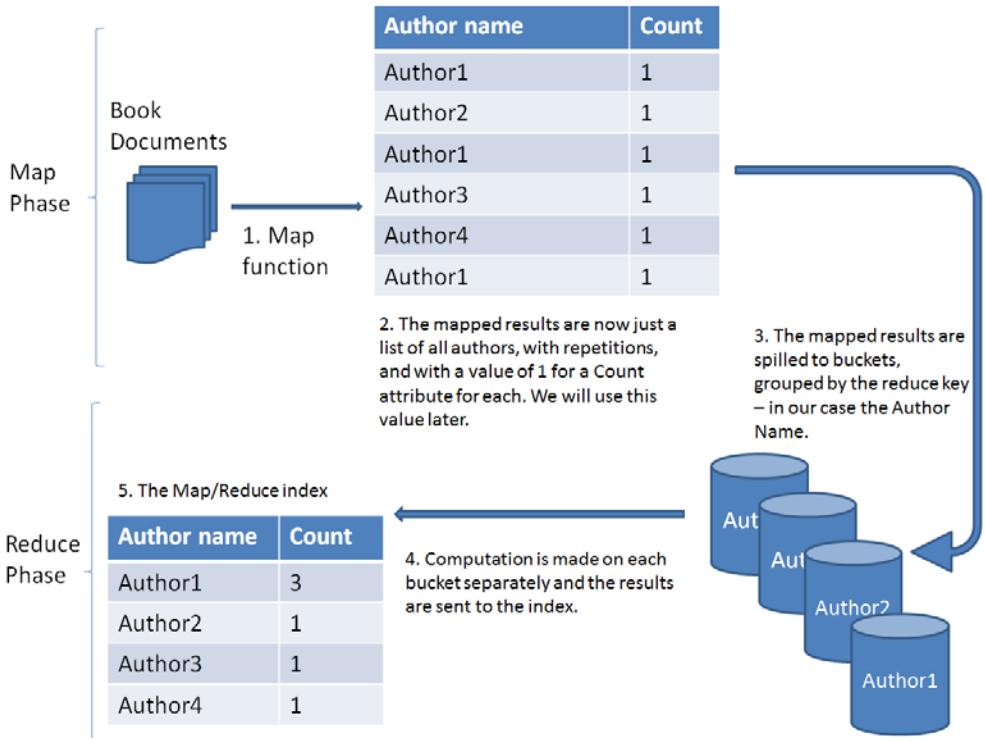


Figure 3.13 The Map/Reduce operation in RavenDB

The Reduce operation will always run after the Map phase of a Map/Reduce index. It's a two-step operation on the results of the Map function and is defined by the Reduce function.

3.3.2 The Reduce function

As its name suggests, the responsibility of the Reduce function is to take a dataset and reduce it. The dataset that serves as input to the Reduce function is the result of the Mapping phase, or the mapped results of this index. At the Reduce phase this is essentially a set of objects with a simple structure.

www.dbooks.org

The output of the Reduce function is always another set of objects having the exact same structure, but it's usually a much smaller set than the original set it got as input.

Reducing the mapped results is effectively done by performing two sequential operations, grouping and merging. First, you group all the objects in the mapped results by some value, and then you merge the documents within each group by applying an aggregation operation on it.

The Reduce function for the Books example looks like this:

```
from result in results #A
group result by result.AuthorName into g #B
select new { AuthorName = g.Key, Count = g.Sum(x => x.Count) } #C
```

#A Gather all mapped-results.
#B Put the result into a bucket (called g) using `AuthorName` as the reduce key.
#C Once you've finished grouping and have all buckets available, you can return a list of objects; each bucket will give you an object with `AuthorName` as its key and an sum of all `Count` properties in the documents it has (essentially, counting the documents) as its `Count`.

If you've worked with LINQ before, this should look familiar. Grouping can be done on essentially any object, but you usually use a field of an object from the result because it makes the most sense. Once you've grouped the results into buckets, you can use LINQ's aggregate operators `Sum`, `Min`, `Max`, and `Average` to produce a result value from the data in each bucket and save it in the result as a field in the objects you return. In the next section you'll practice these new concepts using a more involved example.

3.3.3 Map/Reduce by example

In order to become more comfortable with Map/Reduce indexes, you'll tackle a real-world problem and build some of the functionality of the StackOverflow website—a simple question-and-answer website.

First, add a few Question documents to your database. In each Question document you want to have a list of tags and the date and time the question was asked. Go ahead and add some question documents using Management Studio. Here are a few examples:

```
{ // questions/1
  "Title": "One question",
  "Content": "foo bar",
  "PublishedAt": "2013-04-16T17:00:00.000+00:00",
  "Tags": [
    "RavenDB",
    "NoSQL",
    "C#"
  ]
}

{ // questions/2
  "Title": "Another question",
  "Content": "foo bar",
  "PublishedAt": "2013-02-11T17:00:00.000+00:00",
  "Tags": [
    "Java",
```

```
    "NoSQL"
  ]
}

{ // questions/3
  "Title": "Some other question",
  "Content": "foo bar",
  "PublishedAt": "2012-02-11T17:00:00.000+00:00",
  "Tags": [
    "C#",
    "HTML",
    "jQuery"
  ]
}
```

Given this model, you want to be able to answer queries like what are the most popular tags, what is the most recently used tag, and so on. Because you don't store any information on tags, you need to use a Map/Reduce index to compute those values for you based on the data you store for questions bearing the tags.

The first step in writing a Map/Reduce index is to determine what data you need to generate using the Map function, so you have everything you need in the Reduce phase. Since you're interested in data on tags, you'll select all tags from all questions. Along with each tag you also want to map the date and time the question was asked and a Count of 1, which you'll use for counting at the Reduce phase:

```
from doc in docs.Questions #A
from tag in doc.Tags #B
select new { Tag = tag, Timestamp = doc.PublishedAt, Count = 1 } #C
```

#A Iterates through all documents of the Questions collection.
#B Iterates through all tags for each question. This will make the Map function output multiple anonymous objects for a question—the same number of tags it has.

Go ahead and create your first Map/Reduce index using Management Studio. Open Management Studio and from the Indexes tab select New Index. In the Edit Index screen, type in the name of the index and the Map function just shown; see figure 3.14.

www.dbooks.org



Figure 3.14 First step of adding a Map/Reduce index—creating the index and specifying the Map function

The Map function will generate mapped results with objects having three fields each. The first field is the tag name, the second is the date and time of the question it was taken from, and the third is a field called Count with a value of 1. Now you'll write the Reduce function, with the mapped results as its input.

Because you're interested in the data on tags, the first thing your Reduce function needs to do is group the mapped results on the tag name. As you saw previously, this will create a bucket for each unique tag name, which is now your reduce key, and in this bucket you'll have all the records from the mapped results sharing the tag name that was assigned for that bucket.

After you've grouped the results by the reduce key, you can look at the data inside the buckets and perform aggregation operations on it. You want to get the latest timestamp for each tag, which is the last date and time the tag was used, and the number of times it was used:

```
from result in results #A
group result by result.Tag into g #B
select new {
     Tag = g.Key, #C
     Timestamp = g.Max(x => x.Timestamp), #D
     Count = g.Sum(x => x.Count) #E
}
```

#A Iterating through all the records in the mapped results for this index

#B Grouping the results by the reduce key, `result.Tag`, into `g`
#C `g` is your bucket, and `g.Key` is the bucket's value of the reduce key.
#D Select the maximum `Timestamp` value from the bucket, which is the last time this tag was used in a question.
#E Count the records in the bucket—the number of times a tag was used in a question.

Add the Reduce function to the index definition by clicking Add Reduce in the Edit Index screen. This will add an input field for the Reduce function, where you can type it in and click Save, as shown in figure 3.15.

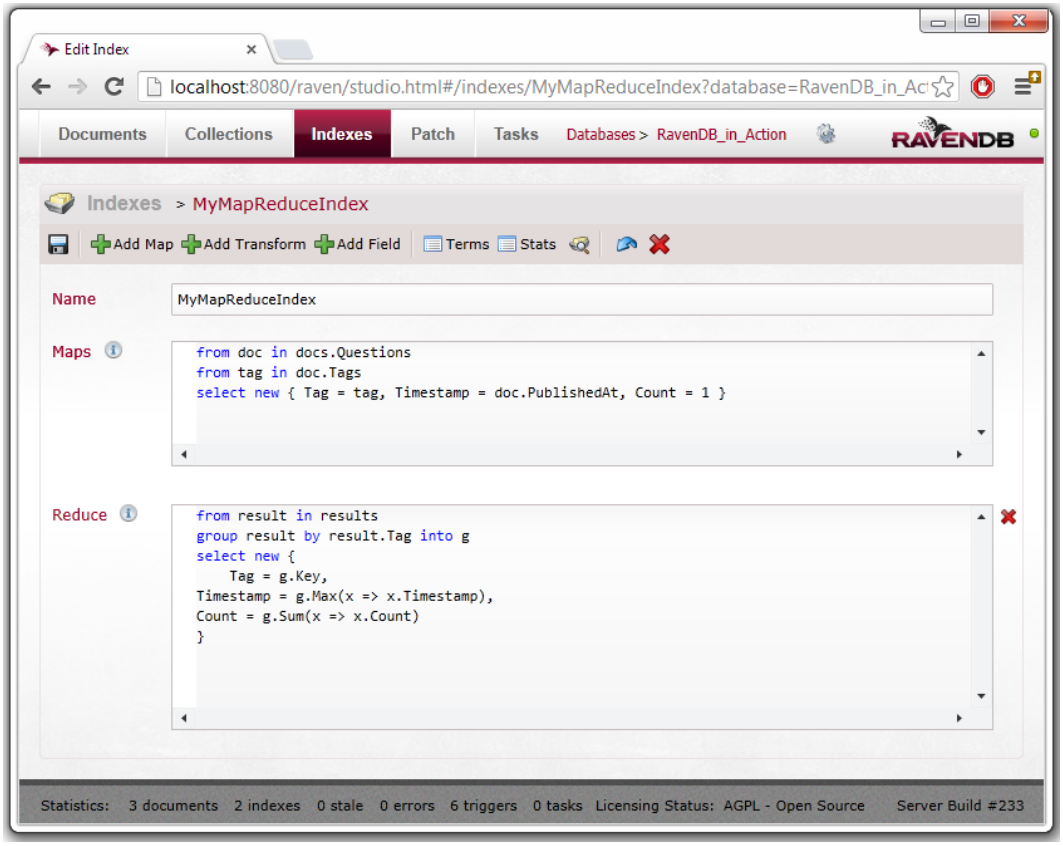


Figure 3.15 The complete Map/Reduce index definition

Note how the return type is the same on both the Map and Reduce functions. This is a requirement RavenDB enforces; failing to return the same structure of anonymous object from both the Map and Reduce functions will prevent the index from registering on the server.

www.dbooks.org

Once you save the new index, RavenDB starts pushing data through it so it can process it for indexing and make it available for querying. Because the amount of data you currently have is minimal, it will be ready instantly.

Issue a query on your new Map/Reduce index to see if it works. Go to the Indexes tab and select your newly created index to reach the Query Index screen. Because the output of your Map/Reduce index has the fields Count, Tag, and Timestamp, you can use any of these (or more than one) for querying. Issue a simple query to find all tags that appear exactly two times, as shown in figure 3.16.

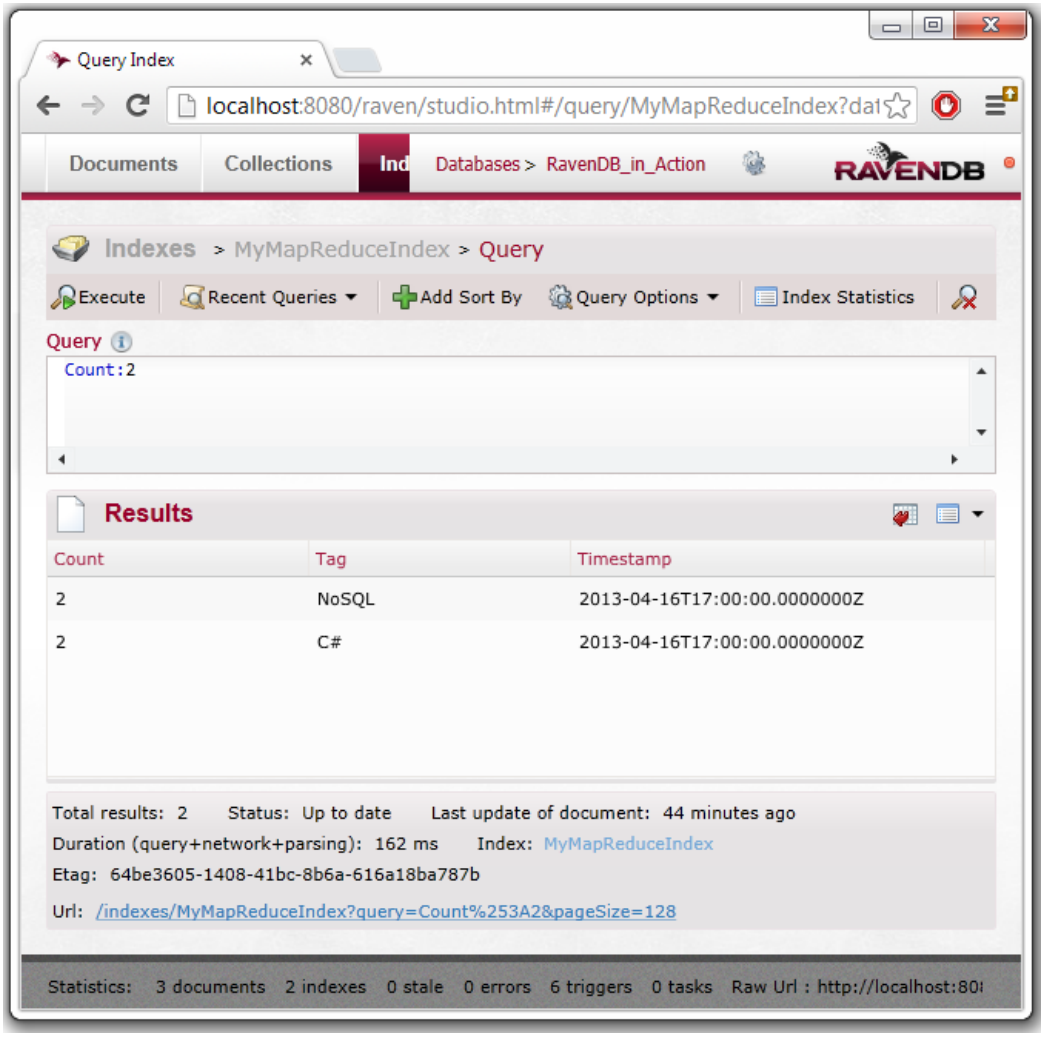


Figure 3.16 Querying a Map/Reduce index

Fortunately, it works as expected. You can also query on timestamps, ranges of values in Count or Timestamp, or on the tag itself to get the result of those computations for the specified tag.

3.4 Summary

Querying in RavenDB wouldn't be possible without indexes, and in this chapter we took the time to explain how they work. We introduced the way RavenDB performs indexing—the asynchronous indexing process—and talked about the possibility of getting stale results, although in most real-world cases you'll hardly notice that.

As we dove deeper into the subject you learned of the building blocks that make an index, and we demonstrated how they can be used to create various indexes.

The first and most fundamental building block of an index is the Map function. The Map function is responsible for defining the index fields of an index and also what data should be indexed to them using the documents in the database as input. Every index has to have a Map function; it's what defines the index by both defining the fields it can have and declaring the mapping between documents and the index fields.

Indexes containing only a Map function are called simple map indexes, and their responsibility is to make documents retrievable by querying their data. As you will see over time, most indexes in RavenDB are like that. Every *collection* needs to have one simple index defined for it; otherwise, it would be impossible to query for documents assigned to it.

But sometimes you don't need to query directly on the results of the Map function—the values that were written to the index—but instead on a result of a calculation made with them. A simple example would be a query to list all users who have made more than 10 orders. When querying just on the results of a simple Map function operating on all Order documents, you need to issue a query for each user to count how many orders they have made, which isn't efficient even with a small number of users.

This is where the Reduce function comes in. Instead of writing the results of the Map function to the index, this function takes that as its input, and the output of the Reduce function operating on the mapped results is written to the index instead. The Reduce function can then do various things with the results, and you've seen several examples of that. Indexes containing a Reduce function in addition to the Map function are called Map/Reduce indexes, and they allow RavenDB to answer more types of queries quickly and efficiently, because the results are precomputed during indexing time.

Both the Map and Reduce functions are written as LINQ expressions, which get a dataset as input and output a dataset. The Map function may output a dataset of different types, but the Reduce function must return a dataset of the same type.

In this chapter you created indexes manually and used Management Studio to create them, by typing in LINQ expressions. When you queried those indexes, you used Management Studio again and used the Lucene query syntax. In the next chapter we'll look more into index creation and visit automatic index creation and manual creation from code. We'll also look

www.dbooks.org

more closely at how to query indexes from code—queries that will be transformed into Lucene syntax queries before they will be executed against RavenDB.

4

Working with indexes

This chapter covers:

- Auto-indexes and their lifecycle
- Intervening in management of auto-indexes
- Manual index creation and management
- Advanced field options of indexes

Indexes play an important role in RavenDB. This is the part of RavenDB that allows it to answer user queries; without indexes RavenDB is just a simple key/value store. By offering fine-grained control over indexes and advanced features like automatically creating and managing them when possible, RavenDB shines as a document database.

In the previous chapter you learned about Simple Map indexes that index document properties and enable querying on them and also how Map/Reduce indexes can help with aggregation operations on that data. We discussed the index structure and mentioned several times how important indexes are to the querying process. What we haven't discussed yet is when indexes are created and how.

To ease the process of development RavenDB uses auto-indexes, which relieves you of the burden of declaring indexes upfront manually. Automatic management of indexes is part of RavenDB's auto-tuning feature, which enables you—the developer—to focus on what matters most and leave the technicalities to the database itself.

This chapter concludes our discussion of indexes by examining how RavenDB manages them. You'll learn when to rely on automatic indexes and how to manage them and when it's necessary to create indexes manually. For those times when you do need to create indexes manually, you'll learn how to do so easily from code using strongly typed language to get benefits like compile time checks and intellisense.

www.dbooks.org

Some advanced index features are also mentioned in this chapter, such as the ability to fine tune sorting by specifying a type or a string collation to an index field or enabling full-text search on a field. As you'll learn, those features can add power to indexes but also affect the way indexes are treated.

We wrap up our discussion of indexes by teaching you a few more querying skills, which apply to all types of indexes. These include making sure result caching is effective and asking for non-stale results explicitly.

4.1 The index lifecycle

In the previous chapter you learned about the indexing process and looked closely at the structure of indexes in RavenDB. You learned about the `Map` and `Reduce` functions and how RavenDB uses them to process documents so they're searchable through indexes.

Because the entire indexing process occurs in the background, no computations whatsoever happen when processing a query. The query sent by the user always works on precomputed data—the index—and will never do any computations on its own. This allows queries in RavenDB to return results quickly, because an answer to every possible query has already been worked out by the indexing process.

For this to work, the right indexes must be created prior to any queries coming in. By now you understand indexes and how they operate, but we have yet to cover when and how those indexes are created and how they're selected to be used for an incoming query.

In this section we're going to look at what happens when RavenDB gets queried for information, that is, when the Index Store receives a query. The first item we'll look at is the way RavenDB decides which index to use for answering a given query.

> **NOTE** It's important to remember the difference between querying for documents and loading documents by ID. If you know the document ID, you can go directly to the Document Store (from code it is done by calling `Load<T>()` on the session object). Queries to the Index Store are made only when trying to retrieve documents by attributes other than their IDs.

4.1.1 Automatic index selection

Because a RavenDB database usually has more than one index defined in it, all indexes are given names. You saw that in action in the previous chapter when you created indexes manually using Management Studio. By clicking New Index from the Indexes tab, you got to a screen like the one shown in figure 4.1, where you typed in the `Map` function and also had to give the index a name before saving.

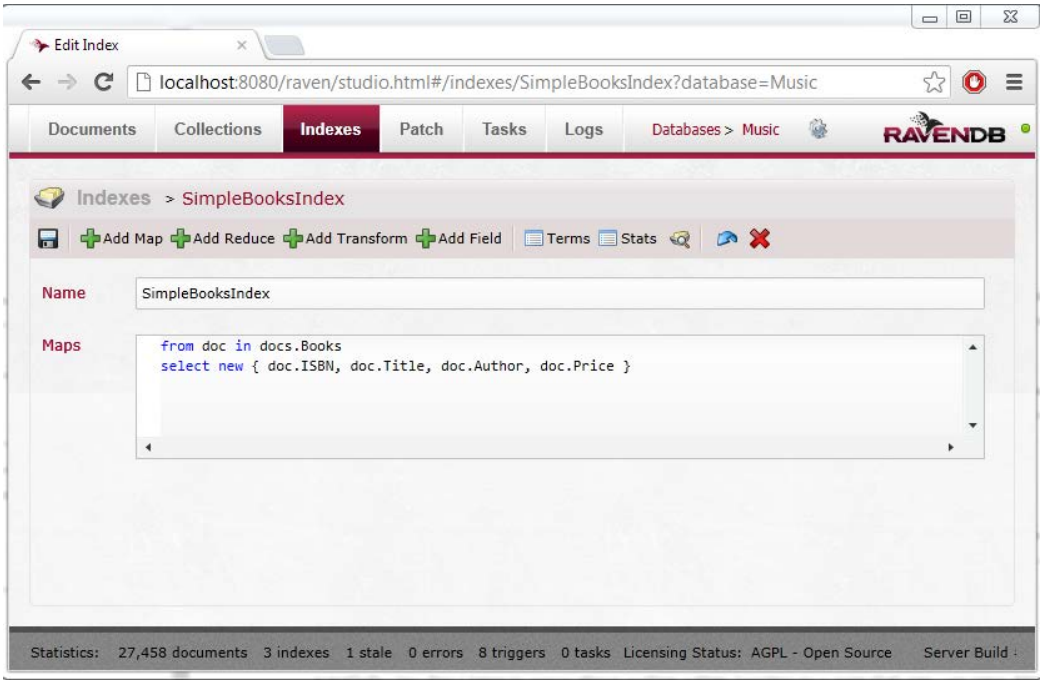


Figure 4.1 Creating a new index manually using Management Studio requires you to specify an index name.

When querying RavenDB for data, you're querying one particular index in RavenDB. The name of the index to query can be specified within the query itself, telling RavenDB that this query needs to be executed against the index bearing this name.

But specifying the name of the index in the query isn't always mandatory. For most common queries RavenDB is smart enough to figure out on its own what index can be used to answer them. If a query did specify the index name explicitly, RavenDB will use that specific index for answering the query; if it didn't, RavenDB will attempt to select an index automatically based on what's being queried.

Imagine you have an index on the Books collection, and you call it Books_Index. A query asking for `all books by Dan Brown` using `Books_Index` will be routed to Books_Index directly, because you specified the index name explicitly. The query is then going to be handled by that index, and results will be sent immediately as they return from the index.

Now consider the query `all books with 500 pages or more`—notice how it doesn't specify what index to use. Since all documents in the Books collection are indexed, and you also took care of indexing the page count data they contain into a field in Books_Index, you know that index can answer this query for you. This is exactly what RavenDB will do: by looking at the indexes you have in your database, it will detect that Books_Index has all Book documents indexed and also have their page count indexed.

www.dbooks.org

Try it yourself

To see automatic index selection in action, you can use the test dataset provided by RavenDB for experimenting with the features described in this section. You can create the test dataset only on an empty database. To do that, create a new RavenDB database for testing, and in Management Studio click Create Sample Data under the Tasks tab.

Once the sample data set has been created, go to the Documents tab and verify that there are about 1,000 documents grouped into eight collections, not counting the System Documents collection, by looking at the left pane.

Going to the Indexes tab in Management Studio, you can also see that four indexes were created. Looking at the Map function of the Orders/Totals index, you can see that the Employee property of all Order documents is indexed by this index.

You can use this screen to query it directly, but instead go back to the main screen of the Indexes tab and click the Dynamic Query button at the top. This will let you type a query against RavenDB without specifying the index. Just make sure you query on the right collection by setting it in the drop-down list at the top of the Dynamic Query screen where it reads Indexes > Dynamic/[Orders] > Query.

For example, querying on the Orders collection with the query `Employee:employees/5` will use the Orders/Totals index you just saw, without needing to specify it explicitly. You can verify that this happened by looking below the documents pane in the results, where it shows both the total number of results and the name of the index used to answer the query.

Automatic index selection works only for queries that can be satisfied by Simple Map indexes. The rule is simple: RavenDB can use its knowledge of the document structure to create an index on its properties so it can be searchable; it can't guess how to perform any additional work on that data, so it can't create Map/Reduce indexes automatically or set any special field options for that Simple Map index.

Now, what happens when no index was found to answer an unnamed query? Let's assume that aside from your huge list of books, you also have a lot of documents describing music CDs in your database. But you don't have them indexed in the system. When a query like `All CDs by Radiohead` comes in, RavenDB won't be able to find a matching index. This is where RavenDB's auto-index feature kicks in.

Key takeaway: Automatic index selection works only with queries that have no index name

Whenever a query gets executed, you get a chance to tell it which index to use. If you don't specify an index name, the automatic index-selection process will kick in, selecting the index for you.

This holds true for simple Map indexes only. Map/Reduce indexes and other variations of Map-only indexes (which we'll look at later in this chapter) will never get picked up automatically and always have to be queried by name.

4.1.2 Auto-indexes

RavenDB is capable of creating new indexes when needed. This will happen when the automatic index-selection process was completed with no good candidate. Just like the automatic index-selection process, it will create only Simple Map indexes for queries that can be answered by such indexes.

Automatic index creation will happen only when a query with no directions to use a specific index gets executed and only if RavenDB can't find an existing index to satisfy the query. In this case, RavenDB will create an appropriate index or extend an existing index so it can answer this type of query as well. Such indexes are called auto-indexes and are managed completely by RavenDB.

Map/Reduce indexes and some types of Map indexes that we'll cover later in this chapter will never be created automatically, mostly because RavenDB cannot figure out on its own how they should be created. When a query requires an index that is not just a Simple-Map index that index has to be created manually or RavenDB will fail to provide results. You always have to create those indexes manually and all queries to them must bear the index name.

Key takeaway: Auto-indexes are never complex

Auto-indexes are always Simple Map indexes. RavenDB creates them by automatically generating a Map function to map data from documents of a specific collection into index fields.

The Map function generated for an auto-index over a collection will map all document properties that have ever been queried onto index fields, to make them queryable.

Because RavenDB can't assume anything about the data type of a document property or guess what calculations to do, auto-indexes will always have a simple Map function.

The process of automatic index selection and automatic index creation is summed up in figure 4.2.

www.dbooks.org

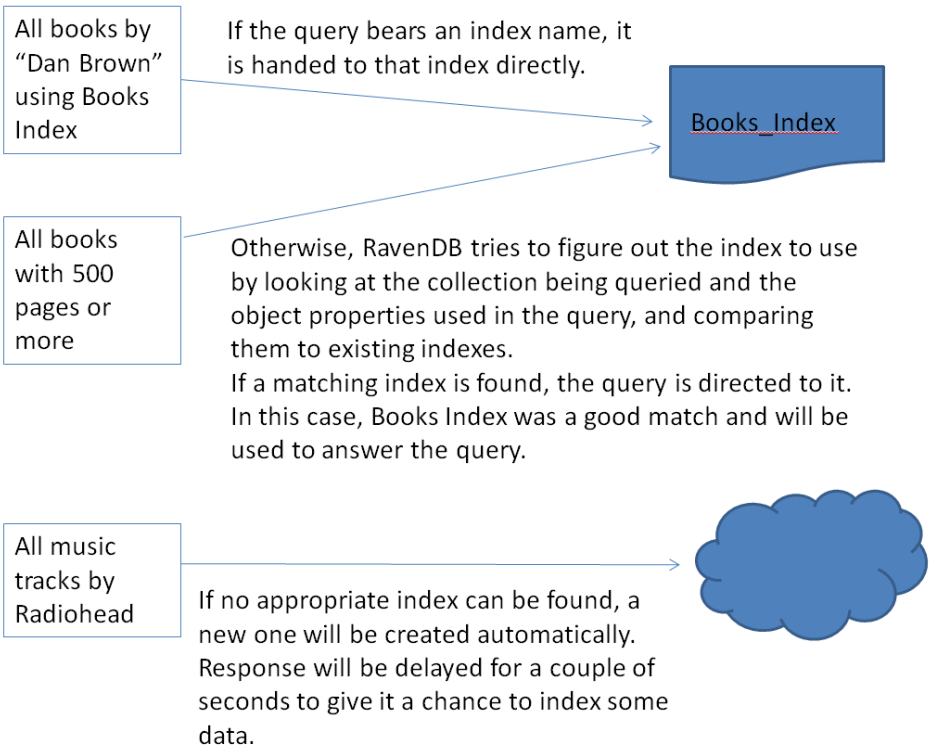


Figure 4.2 The index-selection process for an incoming query

As you'll see, auto-indexes are perfectly capable of handling all basic queries. RavenDB's goal is to manage indexes automatically as much as it can by minimizing the amount of index administration required.

Try it yourself

Continuing your adventure, issue a dynamic query on a property of a document in the system that wasn't indexed by any of the existing indexes. Alternatively, you can delete all the indexes manually from the Indexes tab in the Studio (by right-clicking the index and clicking Delete) and execute the same queries you ran before.

For example, a dynamic query on the Companies collection with the `ExternalId` property will create an index automatically to satisfy the query. Try executing the query `ExternalId:"ANTON"` and then check the Indexes tab again to see that a new index called Auto/Companies/ByExternalId was created automatically.

Now issue another query on another unindexed property of the Companies collection, say `Address.PostalCode:"05023"`. Instead of creating a new index just for this property,

RavenDB will create an index that indexes both queried properties, `ExternalId` and `Address.PostalCode`, even though you didn't use them in the same query.

This is called index merging—RavenDB is smart enough to detect similar indexes operating on the same collection and merge their definitions into one when automatically creating new indexes.

4.1.3 Index aging and prioritization

RavenDB tracks the age of auto-indexes along with the number of queries made to them. If an automatic index isn't used in a specified period, it will become idle and eventually be marked as abandoned. Idle indexes are processed only when the database is considered idle, and abandoned indexes are updated even less frequently.

Index prioritization plays an important part in reducing the stress on the server when new data comes in and in minimizing the chance of getting stale results for queries. By computing the more frequently used indexes first, you can bring them up to speed faster and reduce the time it takes for updates made to the server to appear in results to queries made to those indexes.

Key takeaway: Index priorities

Normal—The default priority for all auto-indexes, which means normal index operation. Whenever new data comes in, it gets processed by all auto-indexes in Normal state right away.

Idle—An index that hasn't been queried much will get flagged as Idle, making it process documents only when the database is idle (usually when about 10 minutes passed without writes). Once an Idle index starts being queried again, it will get promoted back to a Normal index.

Abandoned—An Idle index that hasn't been queried in 72 hours will be marked as Abandoned, making it process new documents only after the server has been idle for about 3 hours. Freshly created indexes that become marked as Abandoned will be considered temporary and get deleted automatically as they reach this state.

Index priorities are managed automatically for all auto-indexes by RavenDB, which has the ultimate knowledge of the usage made with them. This prioritization, made by looking at the query history, can guarantee fair demoting of indexes that aren't used frequently enough.

In some scenarios you might want to intervene and force a priority setting on an index, for example, to make sure indexes used for reporting aren't affecting the various indexes used by the website front end. You can manually set an index priority from the index management tab in Management Studio, as shown in figure 4.3.

www.dbooks.org



Figure 4.3 Changing index priority from Management Studio

Setting an index priority manually to anything other than Normal will force the index to stay at that priority and not be affected by the automatic prioritization process. Setting the priority to Normal manually will cause the index to return to automatic prioritization.

One special index priority can only be set manually—Disabled. Indexes set as Disabled will not participate in indexing at all and will practically be shut off. This is a useful feature when you expect to have a lot of data to come in, for example, during data imports, and you don't want to slow down the server or the import process itself. Like with the other priorities, you can bring the index online again by setting its priority to Normal.

Some indexes can't be created automatically by RavenDB and will require you to create them manually. For example, all Map/Reduce indexes have to be created manually, as do some types of simple Map indexes having special configurations. We'll look at such configurations later in the chapter in section 4.3, "Index field options."

4.2 Creating indexes manually

Most queries made to RavenDB can be satisfied with Simple Map indexes, so you can completely rely on RavenDB's auto-index feature for those types of queries. Some queries, however, require more evolved indexes—like Map/Reduce indexes or Map-only indexes with special configurations, such as the ones we'll discuss later in the chapter.

Creating an index manually means writing the `Map` and `Reduce` functions by hand and then passing the index definition to the server for registration. Once the index has been registered on the server, RavenDB will start indexing the entire data set into it.

Manual indexes don't participate in the automatic management process of auto-indexes, but their priority can still be set manually using the Studio, as shown in the previous section.

You registered an index manually in the previous chapter when you learned about Map and Map/Reduce indexes and defined indexes through Management Studio. Using the Studio is one way to create indexes manually with RavenDB; the second is from code.

In this section you'll learn about creating indexes manually from code—both Simple Map indexes and Map/Reduce indexes. By the end of the section you'll be able to define strongly

typed indexes from code, so indexes can become part of your code and not managed separately.

4.2.1 From code

As you saw in chapters 1 and 2, queries in RavenDB are easy to write with the .NET Client API, thanks to its LINQ support, which enables you to write queries in a strongly typed manner.

Writing indexes in code provides the same benefits, because it uses strongly typed index classes. This means you can define an index as a class in your code and write the `Map` and `Reduce` functions against the actual `Model` classes. This provides several important advantages, like intellisense support when writing the indexes and compile-time checks. For example, an index can fail compilation if a change was made to the model but the index wasn't updated accordingly. If you're working with a source control management system like Subversion or Git, you'll also enjoy the benefits of tracking changes to those classes, just like for any other piece of code in your project.

To write an index, create a new class and have it inherit from `AbstractIndexCreationTask<T>`, which is available under the namespace `Raven.Client.Indexes`. This is a generic class that expects a type, the type on which this index will operate. Only documents in the database under the collection generated for this type will pass through this index and get indexed.

Simple Map indexes are trivial to write. It's just about using LINQ to express the `Map` function and to define it in the class constructor. Here's an example for a strongly typed index:

```
public class SampleIndex : AbstractIndexCreationTask<Book> #A
{
    public SampleIndex() #B
    {
        Map = docs => #C
                      from doc in docs
                      select new #D
                                 {
                                     doc.Title,
                                     doc.Author,
                                     doc.Price,
                                     Year = doc.YearPublished,#E
                                 };
    }
}
```

#A Only documents from the Books collection will be indexed.
#B The entire index definition is always written in the default constructor of the index class.
#C The `Map` function is created as a strongly typed LINQ expression.
#D Each property of the anonymous object returned from the `Map` function is a searchable index field.
#E LINQ will use the same property name of the property passed to it to create the anonymous object, but it can also accept renames, as it did here.

As you'd expect, Map/Reduce indexes are a bit more involved. In addition to defining the type they work on, which gets translated to the collection name, for the strongly typed version to work correctly you need to define another class to represent the shape of the data in the

www.dbooks.org

Reduce phase. If you already have it handy in your Model domain, that's fantastic, and you can use it there; otherwise, you'll have to define it manually as demonstrated in the following listing. Which class you use has no influence on the index definition whatsoever; it's merely to enable the strongly typed version of the index definition.

Listing 4.1 A strongly-typed Map/Reduce index

```
public class MapReduceIndex : AbstractIndexCreationTask<Book,
MapReduceIndex.ReduceResult> #A
{
        public class ReduceResult #A
        {
            public int Year { get; set; }
            public int Count { get; set; }
        }

        public MapReduceIndex() #B
        {
            Map = books => from book in books
                           select new
                                      {
                                          Year = book.YearPublished,
                                          Count = 1,
                                      };

            Reduce = results => from r in results
                                group r by r.Year
                                into g
                                select new
                                           {
                                               Year = g.Key,
                                               Count = g.Sum(x => x.Count),
                                           };
        }
    }
```

#A The `ReduceResult` class is used to express the index result in a strongly typed fashion and is also defined in the class as a template type.
#B Just like Simple Map indexes, the `Map` and `Reduce` functions of strongly typed Map/Reduce indexes are defined in the class constructor.

Note how the index classes are named `IndexCreationTask`, meaning they represent only the creation task of an index. Indeed, RavenDB will translate that strongly typed representation of an index definition into a simple string representation as the one you'd use when typing in Management Studio and will send that for registration as an index on the server.

This means two things:

- You can't use methods defined on the Model object or similar in your `Map` and `Reduce` functions. They don't exist on the server, and the index won't be able to operate at all.
- You need to somehow trigger the registration of the index on the server; it won't happen on its own. We cover this next.

4.2.2 Registering indexes on the server

An index-creation task is, as we explained, just a strongly typed representation of an index definition. The index definition itself needs to be generated out of that class and sent to the RavenDB server.

Every index class is derived from `AbstractIndexCreationTask` and can be executed on the Index Store for registration directly:

```
new SampleIndex().Execute(documentStore);
```

But it's a lot easier to ask the client API to find all those classes automatically and send them altogether. This is usually done right after initializing the Document Store object, in the startup of the application, or in your web-application's global.asax.cs, by calling

```
IndexCreation.CreateIndexes(Assembly.GetExecutingAssembly(), documentStore);
```

Registering an index on RavenDB will have no effect if that index has already been registered on the server for that database, unless its definition has changed. If an index definition has changed, registering the new definition will reset the existing index and calculate the new index from scratch, which might take a while for large databases. The index will still be accessible, but until it catches up, the results to queries it answers will be stale.

When you need to update an index definition in production, you might want to register the new index under a different name to let it catch up before deploying the new version, which uses the new index definition.

Once the index has been registered on the server and indexing to it has started, it's immediately available to querying. If it's a Simple Map index, it will participate in the automatic index-selection process; otherwise, it will have to be queried directly by name. You already learned how to do that using the Studio, and we'll look at how to do this from code later in the chapter, in section 4.4.

4.2.3 Error handling

When registering an index manually you might want to watch for errors it may throw during creation or during normal operation. Index compilation errors will be caught when registering the index, so they're quite hard to miss. The Status tab in Management Studio provides an easy viewer for all index errors that are thrown during normal operation, as shown in figure 4.4.

www.dbooks.org

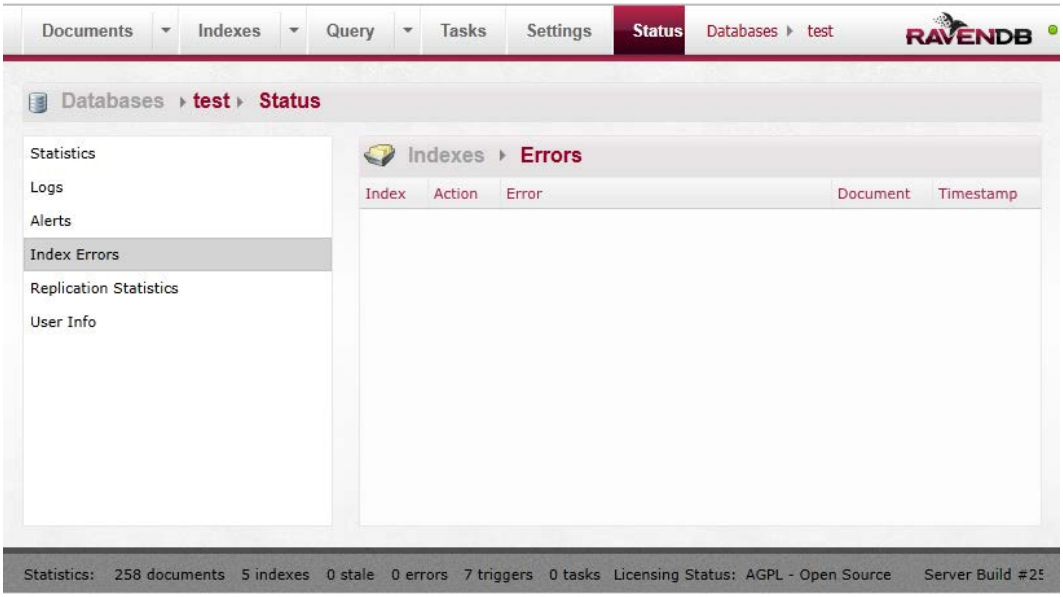


Figure 4.4 The Index Errors viewer

Given the detailed error description, you should be able to go back to the index you defined manually and fix it.

Try it yourself

Try to register an index that throws an error, spot the error using Management Studio, fix the index definition, and register it again, fixing the error.

As you've probably experienced more than once, the easiest error to re-create is dividing by zero. Create a `Map` function that assigns a value that's divided by zero to an index field (for example, `Price = 5 / 0`) and see what happens. Although the world probably won't come to an end, the Index Errors log will start reporting problems, and you'll see that this specific index will be shut down.

Another type of error is one that is caused by some of the documents but not all, for example, in a `Map` function that makes a calculation based on two fields in a document, such as division where the divisor in some of the documents is `0`. As you'll see, this will cause errors to be reported but won't completely halt indexing for that index.

Fixing the index is a matter of editing via the Studio—or if it was defined in code, just fixing its definition there—recompiling and running. The `IndexCreation.CreateIndexes` call you should have on startup will reset all indexes whose definition has changed.

Having an erroneous index in the system doesn't affect overall system performance. Indexes with severe issues will be disabled, and indexes with less-severe issues that affect only some documents will skip those documents and continue logging errors as they encounter issues.

It's important to occasionally check the Index Errors log, both during development and in production, to make sure your indexes play nicely with your data.

4.3 Index field options

In chapter 3 you learned about the structure of indexes in RavenDB and saw how values are mapped to fields and made available for searching on. Sometimes you need to give certain index fields special attention, for example, if they're numerics and you need to sort on them.

To enable this and more, RavenDB allows defining various options on individual fields. In this section we explore setting custom field options, which applies to both Simple Map and Map/Reduce indexes.

4.3.1 Setting a field option

Setting an index field option is done in the index definition. If your index is a manual index that's defined in code, you add an appropriate setting in code, as I'll demonstrate in the subsections that follow.

Index field options can also be set from the Studio, by editing the index definition and clicking Add Field, as shown in figure 4.5. In the field that's added, you type the field name and select the field options to apply to it.

www.dbooks.org

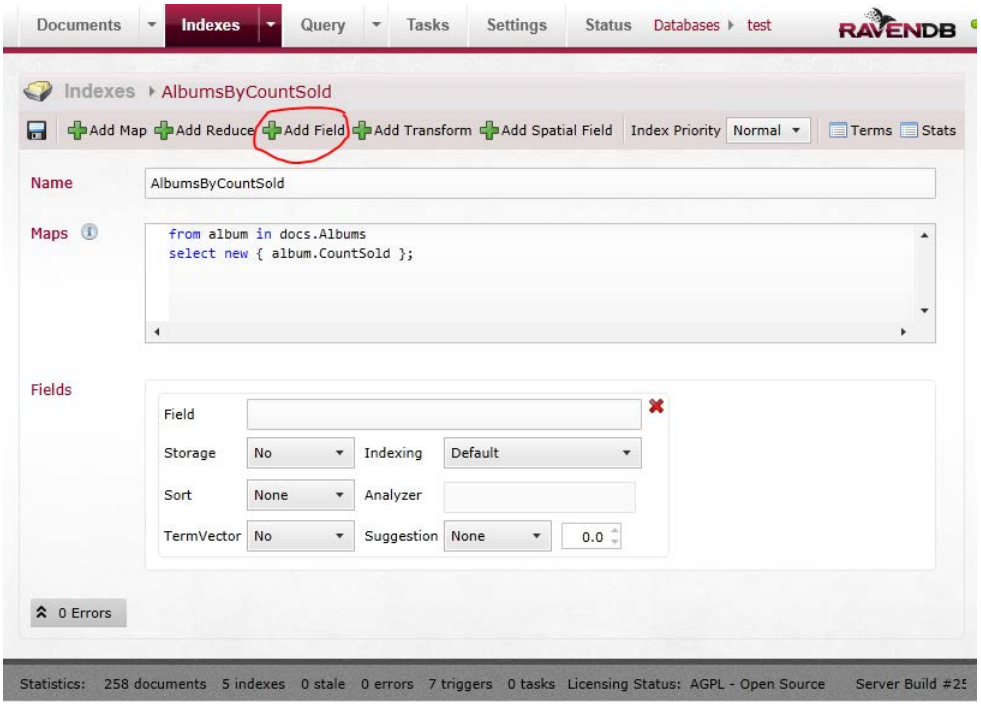


Figure 4.5 Adding field options to a field in an index definition by using the Studio

We'll now discuss the various field options that are available to you when editing an index definition.

4.3.2 Available field options

You can set three types of options on an index field. They all have to be applied on a field that's being mapped to the index as the product of the `Map` or `Reduce` function, and once they're set on that field, they influence the way it's being written to the index.

The first type is sort options; if a field isn't a string field and it needs to participate in sorting, RavenDB needs to be told of its type explicitly. By setting a sort type on the field, you tell the RavenDB indexing process how to interact with that field.

The second type is various options that can be set on string fields to make them full-text-search enabled, and they enable nice features like suggestions or highlighting of terms in the results. This includes the Indexing, Analyzer, TermVector and Suggestion configurations one can set for the selected field.

The last type of index field options, storage options, is responsible for the way the data is loaded back into the results of a query. If a field is set to be Stored, RavenDB will store the original field values in the index itself and load them from there when loading query results, instead of loading the data from the original document.

We'll now look more closely at these options and show how they can be used.

SORT OPTIONS

By default all values in RavenDB indexes are strings; therefore, when sorting results by an index field, the results are lexicographically sorted based on that field. For most strings it makes sense, but when sorting on numerical values this can yield some confusing results.

To be able to sort results by a non-string field, RavenDB must be told the type of that index field. In the Studio, select the data type of the field from the Sort drop-down menu. In `AbstractIndexDefinitionTask<T>` you can define a sort on a field by code, like this:

```
public class CustomersIndex : AbstractIndexCreationTask<Customer>
{
    public CustomersIndex ()
    {
        Map = users => from doc in users select new { doc.Name, doc.Age };

        Sort(x => x.Age, SortOptions.Int);
    }
}
```

Of all the non-string native types, date fields are an exception to the rule and don't require setting a custom sort, because they're being stored in the index using a lexicographically sortable representation.

ANALYZED FIELDS

By default all string values are stored as is in the index, which means no full-text-search support is enabled for them. For example, if you're looking for "Dan," you won't find "Dan Brown"; you have to query for "Dan Brown," the full string value, in order to find a document containing it. String queries in RavenDB are case insensitive, so "dan brown" will work too—but still the full string value has to be provided.

To enable full-text search on string fields so a query like dan would still yield documents containing "Dan Brown" in the results, you need to explicitly define that string field as Analyzed. This can be done by setting the Indexing field option to Analyzed or by specifying which analyzer to use explicitly. For Analyzed fields you can then set TermVector and Suggestion options to enable features like results highlighting and suggestions when typos are made in queries. We go into much greater depth for all those features in chapter 7, when we discuss full-text search in more detail.

STORED FIELDS

As you saw in the previous chapter, index fields are created based on the output of the `Map` function (or the `Reduce` function, in Map/Reduce indexes). The idea is simple: every field in the object that's returned from that function is being mapped to the index under a matching name, and by that it becomes searchable.

When RavenDB executes a query using an index, by default the index returns only a list of document IDs matching the query. RavenDB then uses those IDs to load the documents before

www.dbooks.org

sending them back as results to the query. In other words, by default the indexes serve only as means for search, and no data that's being stored in them is handed back to the user.

RavenDB allows you to define a field as Stored, and by that store the original value that was mapped to it, in addition to making it searchable. This in turn allows you to get the value mapped by the index function directly from the index.

This has several uses; one of the common ones is to use the `Map` function to perform some calculations on the data and save the result to the index. For example, when indexing Order documents, you can compute the total from within the `Map` function and dump it to the index, instead of having an explicit field in the document hold that value.

By saving the computed value to a Stored field, you can get values to display even if they're not present in the document itself. And that's in addition to the documents being searchable by those fields.

To set a field as Stored, select Yes from the Storage drop-down in the Studio, or if the index definition is in code, add this line:

```
Store(x => x.Age, FieldStorage.Yes);
```

To retrieve a Stored field you can use the `.ProjectFromIndexFieldsInto<T>` operator. Using the type passed to it as `T`, it will request from the index the fields matching the properties of that class. If the index fields are stored, they will be sent from the index; if they're not, then the values from the document itself will be used if they exist there.

Stored fields and stale query results

A more advanced use case for Stored fields, which is also quite common, is to avoid displaying confusing data due to stale query results.

We already discussed the possibility of queries returning stale results. Having stale results for a query usually means some documents weren't indexed yet by the indexing process, hence some queries could return fewer results than there really are. But in some scenarios this could also mean having false positives in query results, and this scenario may confuse or frustrate users.

Because the most recent version of the documents is being returned as search results, but the version that existed on the index when the search was executed could have been a bit older and different, there may be cases where you get results that don't belong there, for example, when you execute a query for all products under a certain price, and just at that moment someone else raises its price. Because there's always a race condition between writes and indexing, it may not have been caught up by indexing yet, and you may get that document with the new price as a result to a query that actually asked for all products with a lower price.

By retrieving the field value from the index itself and not using the value from the document, you can guarantee that the display always matches the expectations of the user executing the query. Obviously, as this example shows, there may be other considerations to take into account.

4.3.3 Custom field options and automatic index selection

Simple Map indexes with custom field settings will still participate in the automatic index-selection process discussed earlier in the chapter, but any Analyzed or Sorted fields they may have will be ignored.

For example, if you have an index on Product documents and the Description field of the Product document is set to Analyzed, to enable full-text search on it you'll need to specify the index name explicitly when querying using that field. Otherwise, RavenDB will create another index to match the query because it will deem the existing index unsuitable for answering the executed query.

This holds true whenever the name of the field queried matches the name of the mapped field that was set as Analyzed. If the field name was different, RavenDB would ignore it completely, look for the field with a name matching the one of the document, and add another field automatically to the index if it doesn't exist.

When needed, you can map a text field multiple times, once as Analyzed and once normally. This will make sure the index still participates fully in the automatic index-selection process and can also be queried as a full-text-search-enabled index. This applies to Sorted fields as well.

4.4 Querying

To conclude our discussion of indexes we now go back to querying, to show how this new knowledge affects your RavenDB querying abilities.

In chapter 2 we looked at the RavenQueryStatistics object, using it to get the total number of results a query yielded. The RavenQueryStatistics object contains more data that now will make sense to you, in particular the name of the index used and various data on the staleness of the index and the results it provided.

As a short reminder, here's how to get the Statistics object when querying:

```
RavenQueryStatistics stats;
var books = RavenSession.Query<Book>()
    .Statistics(out stats)
    .OrderByDescending(x => x.YearPublished)
    .ToList();
```

Some entries you might find useful are listed in table 4.1.

www.dbooks.org

Table 4.1 Commonly used entries of RavenQueryStatistics

Entry name	Purpose
`IsStale`	True if the query results are stale. This doesn't mean the results aren't precise; it means the index still has work to catch up with and the results might not be up to date with the latest changes.
`TotalResults`	The total number of results the index reported for the query.
`IndexName`	The name of the index used to answer the query.

This information can be useful in various cases, for example, when you want to show the user how many results there are to their query in total, or to give some indication that the results they're looking at may not be the latest ones because RavenDB indicated they may be stale.

In the next sections we're going to look at a few more options you have at your disposal when querying: how to query a specific index (required when not querying Simple Map indexes), how to properly take advantage of query results caching, and finally how to explicitly wait for non-stale results.

4.4.1 Specifying which index to query

All the queries we've issued from code so far didn't specify an index name to work on and instead relied heavily on RavenDB's automatic index-selection feature. To make a query to a specific index by name, all you need to do is pass the index name, or the index class type, to the query method as a parameter or a type parameter, relatively:

```
Query<Book>("BooksIndex")#A

Query<Book, BookIndex>()#B
```

#A A query for a Book document(s) made explicitly to an index named `BooksIndex`
#B Same query, to the same index, but using the index class inheriting from `AbstractIndexCreationTask`. This allows for more-readable and more-maintainable code.

As we mentioned earlier, when querying Map/Reduce indexes or Simple Map indexes with field options set on one field or more, specifying an index name in queries is a requirement.

Because Map/Reduce indexes don't return the same type as the collection they work on, you can take advantage of the `ReduceResult` class used earlier for creating a strongly typed index and use it as the return type of the query. As before, if you already have an appropriate class to use in your domain model, you can use that instead:

```
Query<MyMapReduceIndex.ReduceResult>("MyMapReduceIndex")

Query<MyMapReduceIndex.ReduceResult, MyMapReduceIndex>()
```

4.4.2 Results caching

Query results are cached by default, so if the query results haven't changed on the server (no new data came in), invoking the query will return a 304 HTTP status code instead of sending the results again and will help reduce network traffic.

The DocumentStore object keeps query results and objects loaded through it cached locally, so it can take advantage of this HTTP cache negotiation and provide the data itself when a 304 response is received.

To make sure this can be used to its full potential, write queries that are cacheable. For example, don't ever use `DateTime.Now` in a query, because by definition it can't be cached and will also cause the cache to fill up rather quickly. Instead, use a timestamp by the minute or even hour, to make the query cacheable.

4.4.3 Waiting for non-stale results

In some cases you may want to explicitly request non-stale results for queries, even if this means waiting until the index to be used catches up. RavenDB allows you to do this, but as you'll see, you need to do so cautiously and only in select scenarios.

By adding a `WaitForNonStaleResults` operator to your query as shown here, you're asking RavenDB to wait until the index catches up completely before executing the query:

```
session.Query<Book>().Customize(x => x.WaitForNonStaleResults())
```

You can also specify a timeout, to prevent RavenDB from waiting on query execution longer than desired. If the query doesn't return within the requested time span, a `TimeoutException` will be thrown:

```
session.Query<Book>().Customize(x =>
x.WaitForNonStaleResults(TimeSpan.FromSeconds(5))); #A
```

#A You usually expect queries to return results in terms of milliseconds, a couple hundred of milliseconds at most. Five seconds is a rather long wait.

What either call does is tell RavenDB to wait for the index to report that it's no longer stale before executing the query. The wait is up to the specified timeout or indefinite if no timeout was specified. It's important to realize that this includes new data that came in after the query was issued—if the index never stopped being stale, RavenDB will hold query execution until it does.

More relaxed versions of this method exist as well. They will execute the query once it's known that the index has caught up with all data that came in before a specified point in time. Because they ignore the current state of the index, the wait is guaranteed to come to an end, but the wait can still be long. Those versions accept an optional timeout value as well and will hold query execution until the queried index reports that it indexed all data that came in until the given date and time:

- `WaitForNonStaleResultsAsOf(DateTime)`
- `WaitForNonStaleResultsAsOfNow()`

www.dbooks.org

IMPORTANT Waiting for non-stale results is useful for several cases, most notably tests and administration jobs, but none of them should ever be used in a production environment, because they can and will slow down the system quite a bit.

4.4.4 Personal consistency

A usual eyebrow-raising moment with eventually-consistent systems like RavenDB occurs when an input screen redirects to a viewing screen after submission. The user then expects to see the changes they just made appear in the viewing screen, and imagine their surprise when they don't, as usually happens.

Common examples where this happens all the time include an Add Product screen of an e-commerce solution that redirects the user to a Product Listing page and a blog post page that refreshes after posting a comment. Because the redirection from the data input page to the data display page is instantaneous, and the latter is issuing a query, chances are the indexes will be stale and the new data won't display.

In those scenarios, waiting a bit for indexes to catch up with the latest changes you made to avoid confusion makes a lot of sense. To allow this, RavenDB provides a way to enforce *personal consistency* using the `WaitForNonStaleResultsAsOfLastWrite()` operator. Its sole purpose is to allow queries to wait for indexes to catch up with known changes made by the current user.

This method of querying your write works only on Map-only indexes. Map/Reduce indexes won't respond to this method, because an additional Reduce phase still needs to happen after the write has completed.

Even though production use of this method is considered safe, you may still want to use this only where it's absolutely necessary, to avoid unnecessarily slowing down page loads.

4.5 Summary

In this chapter you learned what happens when you query, how automatic index selection can pick the right index to be used, and how indexes can be created automatically when no matching index is found. We also looked at how auto-indexes are managed to see how they're prioritized and automatically demoted to allow more important indexes to use server power.

For queries that can't be answered using simple Map indexes, you have to create indexes manually. You already knew how to do this from chapter 3, but in this chapter you learned how to do this from code, making this process easier to write and maintain overtime.

We examined cases where creating indexes manually is needed—namely Map/Reduce indexes and Map-only indexes with special field options set.

We finished by revisiting the querying abilities of RavenDB and seeing how your new understanding of indexes is complemented by RavenDB's querying abilities, namely for querying specific indexes by name and waiting for non-stale results when necessary.

This chapter concludes your study of the basic parts of RavenDB. In the next chapter, which is the last of this part of the book, we'll discuss document-oriented modeling, and you'll learn how to take advantage of indexes to help with modeling decisions. In chapter 6 we will look at

more advanced query types that really leverage the capabilities of RavenDB indexes, and the various field types you came to know in this chapter.

www.dbooks.org

5

Document-oriented Data Modeling

In this chapter:

- What is document-oriented modeling
- How to approach modeling for RavenDB applications
- Working with document references
- Various considerations that influence modeling approaches

Communication is key, and language is at the heart of any communication. Two people not speaking the same language will not understand each other, and while in many scenarios that can be just fine, think about trying to organize a large event without having one language being spoken by all the organizers. That will not end up nicely.

This is pretty much the same with software. Running a software product in isolation is easy. Once it needs to communicate with the outside world, things get more complicated as there needs to be an agreement on protocols, language, formats and so on, otherwise it just won't work.

One important example of such commonly required communication is when an application needs to persist data. To be able to this correctly, it needs to talk with some database engine. This discussion needs to be made in a language both ends can fully understand. A glitch in understanding the communication from either side may result in data corruption, and this is something we don't want to see happening.

Equally important is the amount of chatter that happens between the application and the database. The less chatter that happens, the more to-the-point the discussion is, the more performant the application can be.

Data modeling is the process of forging the shape of communication between your application and your database. It is about finding a way to represent our application data in a way that can make sense to the underlying persistence engine, while still making it usable by our application.

Correct data modeling is an essential part of every application development process. This is the part of the development that will influence almost all aspects of your application. How performant, how flexible and easy to extend, and eventually how successful it will be. Getting the Model right can save you a lot of trouble moving forward, because it will mean both the database is able to perform various optimizations, and the application code isn't fighting its way to use the data loaded from the database.

In relational databases, the modeling process involves UML diagrams and normalization procedures to break things up to a flat structure that can be represented in tables and rows. While modern Object/Relational Mappers like NHibernate can do most of this automatically, this still has great effect on our thinking when designing the objects which will represent that data in code. And it still requires a lot of manual tuning to be made efficient.

RavenDB, being a document-oriented database, is good at storing and retrieving of documents. If a piece of data can be represented by a Document, it can be persisted in RavenDB. And once that's done, it can be loaded back directly by ID or come as a result for various queries, exactly like we've seen in the previous chapters.

As a matter of fact, everything can be represented in a Document form. The real challenge is to create a data model that allows us to leverage the features offered by a document database, while keeping to a representation that truly reflects our usage of it in the application. How do we do that? Document-oriented data modeling, which is the focus of this chapter.

Before we start, it is important to realize that the mindset required for correctly modeling a data model for a RavenDB application is quite different from modeling to any other database, especially from a relational database. For readers coming with a strong background in relational modeling this chapter may be a bit hard to digest at first, but please try to resist the urge of doing things the way you are used to. It won't take long for you to realize it's like trying to fit square pegs through round holes.

That being said, many people actually find document-oriented modeling goes much more naturally with their model than a relational model would. Although some models can become extremely complex and be influenced by many business rules, a document-oriented model usually represents the business logic like it is thought of in the real world.

We start this chapter with some discussion on document-oriented modeling, get some concepts defined and useful tools prepared, and then we will go deeper into learning about different considerations that play a part in almost every modeling adventure.

5.1 RavenDB's approach to document-oriented modeling

Document-oriented modeling is all about representing the data entities from your domain by documents. In some cases the transition is obvious – in a document management application like Google Docs for example, a *Document* in the business domain maps exactly to one document in the data model. In most scenarios, however, the mapping between data entities and documents may not be so straightforward.

In this section we discuss the method in which we can create a document-oriented model for a given business domain, and what characteristics define a good model. There is no

www.dbooks.org

algorithm to do that, but there are good guidelines and rules-of-thumb that we can use. It is a skill that's acquired with experience, and by trying and failing, so you should not be afraid of that.

Often times there is more than one good way of modeling things. This is when business considerations should play part, and there are also some modeling advice we give in the end of this chapter, regarding some design considerations that can come into play when modeling.

Since RavenDB is a document database, it uses many modeling approaches any other document database would as well, and a lot of what is being taught in this chapter can be applied elsewhere as well. RavenDB adds several additional semantics and tools that strengthens its ability to work with such models, and those are unique to RavenDB. In this section we start with document-oriented fundamentals, and starting with the next section we will see how those come into play as well.

5.1.1 Documents and Data Entities

In a Document-oriented model, each data entity can be represented either by a document, or as a property of another document (and that document, in turn, represents another data entity). In order to form a language both our application and the database understand, we need to start thinking of our data entities in terms of documents. Therefore the question becomes – what part of the domain should be a document in the model, and what should become a part of another document?

Let's start with a simple example – a simple data model of a Blog engine, where there are BlogPosts, Tags and Comments. Think for a moment, how would you model it so it fits into a document-oriented database?

By quickly sketching some code for classes to contain this model we get a very clear picture of what is a document and what is a property of another document:

```
public class Comment #A
{
    public string Content { get; set; }
    public DateTime PostedOn { get; set; }
}

public class BlogPost #B
{
    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime PublishedAt { get; set; }
    public List<string> Tags { get; set; } #D
    public List<Comment> Comments { get; set; } #C
}
```

#A – a class to represent a Comment business object in our data model
#B – a class to represent a BlogPost business object in our data model
#C – a BlogPost contains Comments, so we put Comments in an array within a BlogPost object
#D – a BlogPost can be tagged (in our case using simple string tags), so we put them within a BlogPost object

A document in a document-oriented model is usually created after an important piece of data in the business domain. Less important data entities will be contained within the more important ones.

In our case, a `BlogPost` object is clearly a document, since it is the most important piece of data in our model. The rest of the data "belongs" to it, as can be clearly seen by the arrays containing Comment and Tag objects within the BlogPost object.

Thanks to the schema-less nature of RavenDB, which lets it store objects as complete object graphs serialized into JSON, the entire object graph of every BlogPost object can be persisted to RavenDB as a Document including all the comments and tags. It is important to note that every object contained within an object we persist will be serialized as-is within the containing document.

This also means every change that needs to be made to the BlogPost – be it a content edit or an addition of a new comment – will involve changing only this one Document. Essentially, that single document now contains everything we need to know about that BlogPost – the actual content, the tags and all the comments made on it.

Essentially, we have identified a data entity in our domain that can handle all changes made to it all by itself, and does not need to propagate changes to other data entities in the model. We call such a data entity a **Unit of Change**.

Key takeaway: Identifying Units of Change is key for good document-oriented models

A Unit of Change is every data entity in your domain that can handle changes all by itself, and no other data entities in the domain will ever need to make changes to it.

Creating a document-oriented model is all about finding those entities, and building the model around them. In the end of the modeling process every model object instance of a Unit of Change entity can be represented by a single Document in our RavenDB database.

There are some exceptions to this rule, and those will be discussed in the next sections.

5.1.2 Unit of Change

In the Blog Model example we've just seen, all operations are performed on the BlogPost entity. To add a new comment, you just add a new Comment object to BlogPost; to edit the content BlogPost, well – you edit it; and when you want to delete the BlogPost you delete it and all the comments made on it will get deleted along with it, without any further action from your side, because they all exist in the same document. There is no change that may be required to a BlogPost document that will require you to go outside of the scope of that document. Every BlogPost object is a single **Unit of Change**.

A model consisting of **Units of Change** has several immediate benefits. The most important of all is the ability to load and save back an entire BlogPost object in one go. All the data we need for one blog post is contained within that object, and we are not required to make any further calls to the database.

www.dbooks.org

For reads, this means we can load full object graphs very quickly, and query them much more easily. Loading a page in a website is now just a matter of loading several of those objects. This also has a great benefit when writing data, since writing one document is much easier and faster to do, and does not require orchestrating several writes in one transaction or dealing with multiple locks.

Another important benefit is the ability to scale out easily. Since one document contains everything we need to know about the business entity it represents, it is easier to spread one database across multiple servers, as no server needs to know about the others. We will look up more closely on scaling out in Chapter 7.

Good Document-oriented Models are ones that have correctly identified the Units of Change within a domain. Other than the immediate benefits we just mentioned, doing so would also allow you to take advantage of more advanced features RavenDB provides more easily. Features like caching, “includes” and result transformers can bring a lot of additional performance and also boost the entire development process, by allowing you to focus on things that matter.

Let’s look at another simple example. Imagine a model of a store taking orders from people - be it a Point of Sale application or an on-line eCommerce solution. In the real world you will have an order, and within that order you will have a list of products the customer bought:

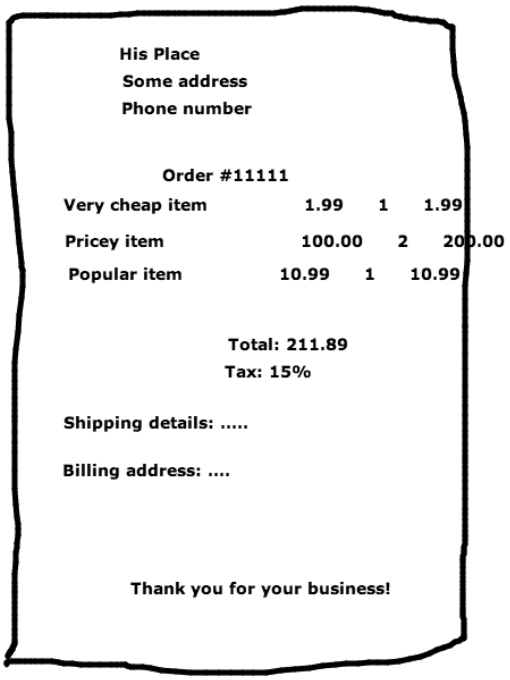


Figure 5.x: A real receipt. We would like to think of a receipt in computerized systems just in the same way

we see it in the real world.

If you were using a relational database you'll be using a model similar to this, breaking up this one piece of data into pieces:

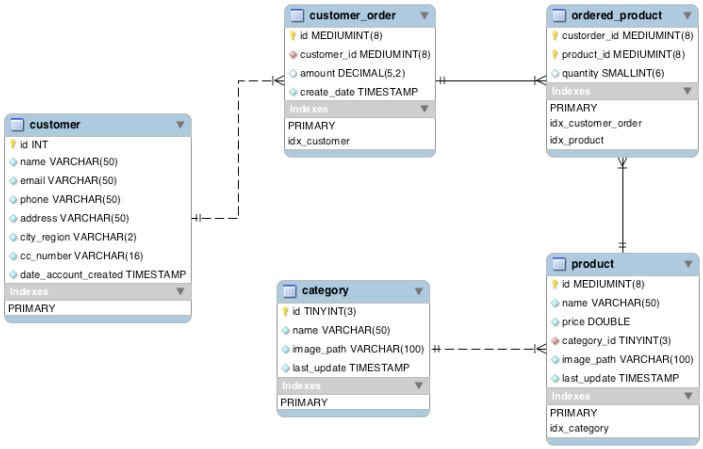


Figure 5.x: A typical relational model for an e-commerce system. At least two tables are needed to store a simple Order and all its details.

With a document oriented model an order will be represented exactly like it is thought of in the real world. An order will simply be a Document, and within the Order document we can have a list of all products bought, and it will look exactly like in figure 5.x above.

For simplicity sake let's ignore for a moment the requirement to record details on the customer, like payment methods, shipping and billing addresses etc, so classes modeling an Order would look like this:

```
    public class OrderLine #A
    {
        public string ProductName { get; set; }
        public string SKU { get; set; }
    public double UnitPrice { get; set; }
public int Quantity { get; set; }
    }

    public class Order
    {
        public List<OrderLine> OrderLines { get; set; }
        public double TotalPrice { get; set; }
public OrderStatus Status { get; set; }
        public DateTime CreatedAt { get; set; }
```

www.dbooks.org

```
}
```

#A An OrderLine object is a simple representation of a Product or service bought - their name, SKU, unit price and amount purchased.

An `OrderLine` instance doesn't have any meaning outside the scope of its containing Order. We are never interested in a single OrderLine object, only in the Order that contains it. If an `OrderLine` changes, the Order changes. This is how it is in the business logic of the domain, and exactly how our Model represents it now.

We represent the products bought using a simple array of OrderLine objects within the Order itself. The fact that RavenDB is schemaless and deals with JSON documents gives us the ability to store entire objects, even very complex ones, as one document – exactly their JSON representation, and gain all the aforementioned benefits.

An Order once stored in the database would look like this:

```
{
    "TotalPrice": 342.4,
    "Status": "Processed",
    "CreatedAt": "2013-11-23T19:00:51+00:00",
    "OrderLines": [ #A
        {
            "ProductName": "Some product",
            "SKU": "V4C3D5R2Z6",
            "UnitPrice": 150.0,
            "Quantity": 2
        },
        {
            "ProductName": "Another product",
            "SKU": "V4C3D5R2Z2",
            "UnitPrice": 42.4,
            "Quantity": 1
        }]
}
```

#A – The OrderLine objects are stored in an array with the Order document itself

The immediate benefit is quite obvious – getting an order is a very simple load operation. There is no need to merge data from additional sources – tables or documents. All you need to do is load the Order document, and there you have it – all the Order data in one place.

Similarly, write operations are very convenient as well – to make changes to an order you need to change only that one document. Using this model, there are no other documents involved; each order object is saved to exactly one document in the database.

Key takeaway: A Unit of Change can be represented by a single autonomous document

Since every Unit of Change is a data entity that can handle all changes made to it without propagating changes to other data entities in the data model, it can and should be represented by a single document.

Having each document completely autonomous and responsible for all changes to the data entity it represents helps greatly with performance and enables better scaling out story. That is proper document oriented modeling.

Every modeling process with RavenDB starts with one simple question – what are the Units of Change in my domain? Once you've answered that, you can start growing your model, dropping in further considerations. We will now look at a way of thinking that usually helps in identifying the Units of Change in a domain.

5.1.3 Transactional Boundaries

Identifying Units of Change of a data model is easy in some cases, but in others it may prove to be a real challenge. It is sometimes easier to identify them by isolating the scenarios in which data entities are being written to the database. Try to think about the cases in which you write this data to the database; what are the domain entities that are likely to share the same transaction? Those are the immediate suspects for becoming one document in a proper document-oriented model.

To demonstrate this, let us take the e-store domain again but this time with more data entities in it, and look at modeling it again. Previously we had Order and OrderLine entities, where OrderLine represented purchased goods, and an Order could contain several OrderLine objects. Let's now throw a Customer and a Product entities into the mix as well, and see how we can use Transactional Boundaries to guide us to the Units of Change in this domain.

A *Customer* entity will represent a customer in our system, including data like his address, phone number and so on to be used for shipping and billing. Every Order will be associated with one Customer object to record the customer who made the order, so our Order class can be changed to contain a Customer as well:

```
public class Order
    {
        public List<OrderLine> OrderLines { get; set; }
        public double TotalPrice { get; set; }
        public Customer Customer { get; set; }
        public OrderStatus Status { get; set; }
        public DateTime CreatedAt { get; set; }
    }
```

A *Product* entity will represent a Product in the system, being used for the catalogue. Every OrderLine is practically pointing at a Product from the catalog, so every OrderLine object is effectively pointing to one Product object. Let's reflect that in the OrderLine class we have:

www.dbooks.org

```
public class OrderLine
    {
        public Product Product { get; set; } #A
public int Quantity { get; set; }
    }
```

#A – The Product object will contain the product name, SKU, price and everything else we need to know about the product

Whenever we want to record a new Order, we always write the Order object along with a list of OrderLines, and when doing so we never make any change to the Customer entity or the actual Product objects. We just need to record them in our Order so we know who the customer is and what Products has he bought, but this doesn't need to affect the actual Customer or Product entities in our system.

This also holds true the other way around. Whenever a Customer changes his details, no Product object needs to change. Likewise, editing a Product details never affects any Customer in the system. So effectively we can identify 3 different Units of Change by looking at the transactional boundaries in our domain:

1. An Order, with all the accompanying information like billing and shipping info, and OrderLines containing the prices and SKUs for the products bought.
2. A Customer, representing our customer and optionally containing his default settings and information.
3. A Product, to be displayed in our system's catalogue, and to be used in various places in the system.

There is no case where a change to one of those entities requires a change to the other. By looking at the transactional boundaries in our domain we were able to identify the Units of Change in it, and create Model using them. The next step would be to represent each of those Units of Change as a Document in our RavenDB database.

Further reading: Domain Driven Design and Aggregate Roots

This approach to the data model design has strong ties with Domain Driven Design methodologies. The Unit of Change is effectively DDD's Aggregate Root, and Transactional Boundaries are also what defines Aggregate Roots in the context of DDD.

As mentioned before, acquiring good design skills are just a matter of practice and experience. Experience comes from implementing these methods in real projects, or learning from others. Reading materials on DDD definitely falls under this category.

A good recommended read on DDD is Eric Evans's book *Domain Driven Design* (Addison-Wesley). That book defines DDD and uses a lot of examples and project stories to guide you through very complex business domain designs.

5.1.4 Denormalization

Still using our Order entity as an example, let's see how such an Order document looks like in RavenDB:

```
{
    "OrderLines": [ #A
        {
            "Product": {
                "Name": "Some product",
                "Description": "...",
                "SKU": "V4C3D5R2Z6",
                "UnitPrice": 150.0
            },
            "Quantity": 2
        },
        {
            "Product": {
                "Name": "Another product",
                "Description": "...",
                "SKU": "V4C3D5R2Z2",
                "UnitPrice": 42.4
            },
            "Quantity": 1
        }],
    "TotalPrice": 342.4,
    "Status": "Processed",
    "CreatedAt": "2013-11-23T19:00:51+00:00",
    "Customer": { #B
        "Name": "Itamar Syn-Hershko",
        "Email": "itamar@domain.com",
        "RegisteredAt": "2012-05-25T11:00:00+00:00",
        "Address": "Some place quiet"
    }
}
```

#A – OrderLines contain the entire Product document within them, so it now includes a lot of additional information like Description, Image and so on.
#B – The Customer document is now embedded into the Order as well

Notice how objects like OrderLine, Product and Customer have been serialized in their entirely, so every property in those objects appears in the Order document as well. While we might want this for OrderLine, a Product and a Customer object actually do live outside the scope of the Order entity and also have their own ID in the system (exactly because they have meaning outside the context of the Order). Being saved like this within the order means if the original Customer document changes, the instance within the Order will **not** automatically update to reflect that.

This is called denormalization, and it is an important side-effect of the way serialization works in RavenDB. If you recall, RavenDB takes any object graph of any depth and just saves its JSON representation as a document, so whatever this object contains will be present in the JSON representation.

www.dbooks.org

There are times when you want that to happen and will actually accept denormalization with open arms, and times when you should really avoid it. We will now look at various scenarios in which denormalization can help or hurt you.

When to Denormalize, and when not to

Let's start with the obvious. Usually, you'd want to avoid denormalizing Units of Change. If you reference a Unit of Change, just use a reference to their ID and not denormalize. Those entities stand on their own right, have their own unique ID within the system, and can be simply referenced by that ID. This way you can always display correct information also if their data change. This also helps keeping your documents lighter, and prevents them from bloating when referencing many other entities.

While denormalization may result in duplication of data, sometimes it does make sense to do this. Our Order scenario is one great example. Imagine we allow our customers to view previous orders, and now a product price or its name change. If we don't denormalize by saving the details of the product as they appear at the time of the order placement, we might end up showing wrong data in the future. By denormalizing product information within the Order document we guarantee it will always show the Order information as it was at the time the order was placed and paid for.

This also applies for Customer info. If user information changes sometime in the future – for example if they moved to a different apartment, it doesn't change the fact that a past order was sent to their previous residence. Denormalizing that information in the Order document guarantees our data will remain historically accurate. We can see then that denormalization is very useful for snapshotting data, to provide point-in-time view of our data when necessary.

Updating Denormalized Data

While RavenDB provides tools to do that, it is not too easy to keep denormalized data up to date. The general rule for denormalization is to do it if you need the data there forever even if it changes in the referenced document (like the Shipping Address of an Order, which doesn't care if the customer moved at a later date), or when it helps greatly with indexing (less common, but can be the case when scaling out).

We will discuss the tools for updating denormalized data later in chapter 9 when we discuss server-side document updates, but for now it is important to realize this is something you should avoid from doing, unless you have a very good reason to.

As we've seen, such good reasons could include point-in-time view of data, or other scenarios where that data has to be handy with no further operations.

For the rest of the cases, you should work with normal document references. RavenDB has several techniques that make handling of referenced documents very easy and powerful, as if they were embedded within the referencing document itself, and this is what we will see next.

Key takeaway: Denormalization has a price, but it may well be worth it

Denormalizing data helps keeping your documents completely autonomous. This is your way of ensuring all data that is required to adequately represent a data entity exists in the context of that entity, even if it is taken from another entity.

This is helpful for display, for better and more efficient indexing, it enables proper scaling out story by easier sharding of data (as we will see in chapter 7), and most commonly used for capturing point-in-time view of data that is never going to change again.

Denormalization comes with a cost of somewhat larger documents, and a lot of duplicated data that sometimes you will need to maintain yourself as data changes in the original documents. As always, this is a trade-off that you will need to decide for yourself if it pays off or not.

Denormalization being a standard practice in RavenDB (and other document-oriented databases as well) highlights the fact that RavenDB does not try to adhere to normal forms like traditional relational databases do; Units of Change is the key concept in good data models that are well-suited to be used with RavenDB.

While RavenDB provides a very good support around referenced documents, as we will see in the next section, it cannot enforce referential integrity – meaning, it doesn't support cross-document integrity checks out of the box. There are some checks that could be added using various extension points, but the very requirement of them usually points to a bad document-oriented model, and exactly why denormalization is a very common practice.

Simply put, referential integrity is not something you should try with, or at all need, with RavenDB. With that in mind, let's see how to work with documents that do have some relationship between one another.

5.2 Working with Referenced documents

Up to this point we established that document-oriented modeling is about keeping all data related to an entity within one document. However, even in a good document-oriented model documents don't live in isolation, and they do sometime need to reference each other.

The most obvious place where referencing documents is necessary is to link between 2 Units of Change, for example to link an Order document to the document representing the Customer who created it.

Since a document ID in RavenDB is always a string, a reference between two documents is just a string property in one of them. For example, here is an Order document referencing the Customer document:

```
{
    "TotalPrice": 342.4,
    "Status": "Processed",
    "CreatedAt": "2013-11-23T19:00:51+00:00",
    "CustomerId": "customers/23492",      #A
    "OrderLines": [
        {
```

www.dbooks.org

```
            "ProductName": "Some product",
            "SKU": "V4C3D5R2Z6",
            "UnitPrice": 150.0,
            "Quantity": 2
        },
        {
            "ProductName": "Another product",
            "SKU": "V4C3D5R2Z2",
            "UnitPrice": 42.4,
            "Quantity": 1
        }]
}
```

#A – A reference to a customer whose details are saved in a document keyed customers/23492

Such a reference between 2 documents essentially provides a loose coupling of them, and is there first and foremost for the means of data persistence – for the sake of keeping the reference for your own business logic. But since both documents reside in the same database, RavenDB can provide several ways to bring the two together when needed.

For the purposes of fetching several related documents quickly and efficiently and save on database calls and network traffic, we can use the Includes feature or link them using Semantic IDs.

If you need to query on data that exists in the referenced documents, there is also a mechanism to load referenced documents from the Map functions, so one query can be made to answer questions made on data that is scattered across documents (in-place indexing).

In this section we will be looking closely at those as means to efficiently work with document references and relationships.

5.2.1 Includes

Includes are a tool RavenDB provides for retrieving referenced documents when the referencing documents are retrieved via a Load or Query operation.

It is quite a common scenario where you load an object from the database, and that entity references other entities stored in the database, and then you have to approach the database again to retrieve them too. Looking (once again) at our Order document referencing the Customer document, it would look like this:

```
using (var session = docStore.OpenSession())
{
     var order = session.Load<Order>(“orders/1”); #A
     var customer = session.Load<Customer>(order.CustomerId);
}
```

#A – Load the Order document
#B – Load the Customer document for the customer who made that order

The problem with this is the double roundtrip to the database server – first we ask for the Order, and then we go again to ask for the Customer. It gets much worse when loading more than one document referencing one or more documents each, in which case we may end up with potentially dozens or hundreds of database calls.

Includes are a way to tell RavenDB to load the referenced documents along with the document requested – during either a Load operation or a Query operation. This way the referenced documents are sent along with the actual response so to avoid the extra roundtrip. To use Includes just add the Include operator to your Query or Load operation:

```
using (var session = docStore.OpenSession())
{
     var orders = session.Include<Order>(x => x.CustomerId)#A
                         .Query(x => x.TotalPrice > 100)
                         .ToList();

     var order = session.Include<Order>(x => x.CustomerId) #B
                         .Load(“orders/1”);

}
```

#A – Include the documents referenced in the CustomerId property of the loaded Order object(s) along with the Order documents matching this query.
#B – This works just the same with Load operations, however although counterintuitive the call to Include has to come before Load, and specify the type to deserialize to (in this case, Order), because the call to Load will perform the actual operation immediately once called.

Includes work by telling RavenDB during a Query or a Load operation that a String property of a stored document is in fact a document ID, and should be treated as a reference to that document. RavenDB will go and grab those string values from all documents loaded as a result of that Load operation or a Query, and just before sending them back as results it will also load the documents with IDs matching the values in those string properties.

Since the included documents are being sent back with the original result set, there is no additional cost associated with Loading them later. So for example this:

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

     foreach (var order in orders) {
            var customer = session.Load<Customer>(order.CustomerId);
     }
```

Will only issue one database call – right after the call to .ToList() on the query. All other operations will be performed entirely in memory, as the other documents loaded have been already loaded by the currently open session.

You can point to any document property containing a string value, or values, to tell RavenDB these are document IDs that needs to be loaded and attached to the response. This means you can do secondary level includes (denormalized objects with properties containing an ID) and also use Includes to load documents from a collections containing a list of IDs:

www.dbooks.org

```
var order = session.Include<Order>(x => x.OrderLines.Select(li =>
li.ProductId))
    .Load("orders/1");
```

5.2.2 Semantic IDs

Using strings for document IDs makes a lot of things really simple. We already mentioned how handy it is to be able to tell the type of the document by its ID – *users/3251* being a User document, and *products/93421* a Product document. RavenDB's default conventions take care of that for us already, and make things like debugging or tracing really easy.

But this doesn't have to end here. A document ID can do a lot more than just denote the type of the document. Because it is a string, it can be virtually anything, and provide a lot more info on the document. If used correctly, it can help you avoid performing additional queries where they aren't really necessary.

The most trivial example for using Semantic IDs is having global configuration documents for your system. Since there is usually only one document used for storing system-wide configurations for a specific part of the system, we can give it a name representing exactly that. For example, *mysystem/shop/config* or *configs/mysystem/myshop*. Having it this way makes it very easy to load it, as you know exactly where to find it.

The RavenDB client API makes that very easy to do:

```
using (var session = docStore.OpenSession())
{
    session.Store(configuration, "configs/global"); #A
    session.SaveChanges(); #B
}

using (var session = docStore.OpenSession())
{
                                          #D
    var configuration = session.Load<dynamic>("configs/global"); #C
}
```

#A – Stores an existing configuration document under a predefined ID
#B – Commits the write to RavenDB
#C – Load the config document back. We can make its ID a constant somewhere so we don't repeat it all over again.
#D – As a shortcut we use a C# dynamic, so we don't have to specify an actual class to deserialize into

Another common scenario where semantic IDs are useful is where you need to break a Unit of Change down into smaller documents. This can provide better separation of data for security reasons for example, or to reduce the size of the main document (a scenario which we will handle later in the chapter). With Semantic IDs this can be much easier to do and maintain, since we can load multiple documents at once, and also load documents using ID prefixes:

```
using (var session = docStore.OpenSession())
{
    var objs = session.Load<dynamic>("blogposts/1", "blogposts/1/comments");
#A
```

```
    var orders =
session.Advanced.LoadStartingWith<Order>("customers/3234/orders/"); #B

}
```

#A – Loading multiple documents, even of different types, in one Load call.
#B – Using a document key prefix to load a series of documents in one go.

Key takeaway: Loosely coupled documents relationships

In RavenDB, there is no such thing as relationships between documents. Relationships may only exist in your business domain, and in turn can be reflected in your document model using RavenDB's string IDs.

One way of doing so is by referencing one document from another, by specifying it's ID. If the two are of completely different domain entities (a Product and a Customer, for instance), then it makes the most sense to just link to one document from another, using a simple string property in the referencing document. Once a document has been referenced, you can use Includes to make read operations more efficient. It can also be used from indexes, as we will see later.

Another way is to use sematic IDs – taking advantage of document IDs being simple strings and using this to our advantage. This is usually more applicable for entities that are more tightly linked, usually in cases where if not for other considerations you would have put them all within one document.

5.2.3 Querying on referenced data

Having a Customer document referenced from our Order document still doesn't allow us to query using the referenced data, for example to get all Orders made by a user with first name John. An index is usually created to index documents of one Collection, and the documents in the Order collection don't contain the data on a Customer since that data resides in a document which is in the Customers collection.

One way to make such queries work is by denormalizing the Customer data into the Order document, like we have done before earlier in the chapter:

```
{
    "OrderLines": [ #A
        {
            "Product": {
                "Name": "Some product",
                "Description": "...",
                "SKU": "V4C3D5R2Z6",
                "UnitPrice": 150.0
            },
            "Quantity": 2
        },
        {
            "Product": {
                "Name": "Another product",
                "Description": "...",
                "SKU": "V4C3D5R2Z2",
```

www.dbooks.org

```
                "UnitPrice": 42.4
            },
            "Quantity": 1
        }],
    "TotalPrice": 342.4,
    "Status": "Processed",
    "CreatedAt": "2013-11-23T19:00:51+00:00",
    "Customer": { #A
        "Name": "Itamar Syn-Hershko",
        "Email": "itamar@domain.com",
        "RegisteredAt": "2012-05-25T11:00:00+00:00",
        "Address": "Some place quiet"
    }
}
```

#A – denormalized references, that data will be available to indexes since it exists in the document itself

However, as we mentioned, this approach has its drawbacks, namely bloated documents and the challenge of updating the denormalized references once data changes. We need to have a way to have that data indexed even though it only exists in the referenced document.

RavenDB provides a better way to allow referenced data to be indexed without actually embedding it in the refencing document. Every index gets a chance to load referenced documents during the indexing process itself if it wishes to, and use their data during indexing. This is how the Map function of such an index looks like:

```
from order in orders
let customer = LoadDocument(order.CustomerId) #A
select new {
     order.TotalPrice,
     // more order stuff we want to be able to query on
     customer.Name #B
}
```

#A – Automatically load the Customer document so we can index data from it (#B)

Now the index that allows us to query on Order documents also has data from Customer documents indexed in it, for each Order using the Customer that created it, thus we can query on Orders based on that data as well.

Whenever data in a referenced document changes it will trigger an index update for the referencing documents automatically, so the index always contains the latest data of both the referencing and referenced documents. However, this also means those indexes are going to perform significantly more work, as indexing work for them is going to be triggered more often. As a general advice, avoid calling `LoadDocument` as much as you can.

Key takeaway: Denormalizing vs writing costly indexes

The easiest way to allow for querying on data that exists elsewhere is probably denormalization, because denormalizing means we now have the data locally without extra work. This does potentially mean there needs to be more work done on keeping this data up to date, and documents may grow significantly larger.

RavenDB does enable you to avoid denormalization completely by using the `LoadDocument` method within indexes. However, this does mean the index has to track more documents and it will be triggered to do its job more often, and has serious repercussions when used too much or in inappropriate places.

By now you should be familiar enough with approaching any business domain and to create a good document oriented model of it. In this section we learned how we could take advantage of some of RavenDB's features that make working with such models much easier.

We are now moving into the next step of document-oriented design. After we have learned about what makes a good model, and how to work with the various pieces efficiently, in the next section we will look at various considerations that can make a model even better.

5.3 Design considerations

In real-world scenarios choosing the domain entities that will make up your domain often times requires a lot more thought than just identifying the *Units of Change*. There are plenty of considerations that play part in identifying the Units of Change in a model, and then there are even more considerations that may play part in making you to store that Unit of Change in multiple parts, instead of in just one document like you'd expect.

In this section we will look at some of those considerations, like how the document size can influence the selection of a Unit of Change, and make you break one document into several smaller ones.

Every domain has its own challenges, and in this section I tried bringing some of the more common patterns I've seen in various projects. For example, how to handle models with large lists of data that bloat documents? How can we optimize queries that are performed very frequently? How to properly support concurrent updates to a document? How to achieve proper unique constraints functionality? We will see answers to those questions and more in this section.

This isn't meant to be an exhaustive list, but rather a list of common patterns ready for you to use, and to show you how rules can be bent for good reasons when necessary. You are more than likely to meet similar questions in any advanced project, and this is certainly a good list of practices to have on your belt.

5.3.1 Size considerations

While designing a document-oriented model it is easy to end up having bloated documents, since many Units of Change can become quite large. However, the size of a single document has a tremendous importance.

This can be seen even in a simple Blog model. Let's consider the amount of comments that can be made to one blog post, which is a Unit of Change in this model like we demonstrated earlier in the chapter. Some blogs are very popular, and can attract many commenters over time. If we store all comments made to one post within its respective BlogPost document, that document will get very bloated in very popular blogs.

www.dbooks.org

A BlogPost document that is large because of comments made to it is quite annoying, as many screens in a blog website don't even list those comments, or lazy-load them. Putting efforts on loading and sending the comments don't really make much sense when all you want is to read the blog post content.

Key takeaway: Keep your documents as thin as possible

It is really important to keep your documents as thin as possible, so they are easy to load by consumers, and also the indexing process doesn't work too hard.

However, beware of anorectic documents. Loading one middle-sized document is way better than loading many very small ones. Keep this in mind while working through modeling your domain with size considerations in mind. The ideal end-result should be self-contained and coherent documents, that aren't too large.

A possible workaround is to store all comments made to a post in a document separate from the BlogPost document – for instance using semantic IDs so for a BlogPost document saved under the ID *blogposts/1* all comments will be stored in a document with the ID *blogposts/1/comments*:

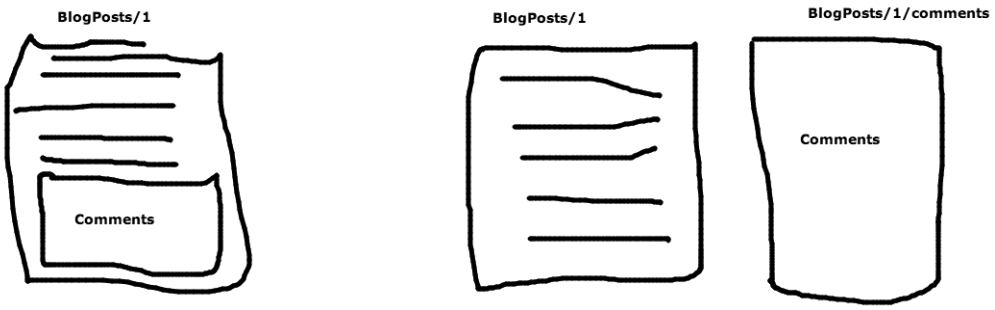


Figure 5.x: A standard BlogPost document (left) can be split into 2 separate documents (right) if the comments are expected to bloat it.

But we might still remain with the exact same problem as before. Indeed – it doesn't affect the BlogPost document anymore, but whenever we wanted to work with comments, we will be loading a potentially large document.

Scenarios in which the requirement for having one self-contained document representing one Unit of Change and the requirement for having small documents can't really line-up, are actually fairly common. Cases where if we wanted to have self-contained documents we will end up having large documents, and breaking them apart to multiple documents would mean breaking the Unit of Change pattern.

There are a couple of ways of dealing with this, and we will be looking at them in just a moment. But as a first step you should ask yourself what is the expected growth of the data, and whether it can be bounded or not. Some types of data are naturally bounded, and some are just highly unlikely to go overboard.

For example, the number of kids in a family is naturally bounded; sure, the actual number and perhaps even the range may change between societies, but there is a number that can be agreed on which can be declared as the maximum possible number of children. So the number of kids in a family is naturally bounded.

The number of products in a standard order is also bounded. The chances of receiving an order with thousands of order lines are very slim; and just to be on the safe side, you can always pose limits in the screens that the user creates an order from. Obviously there could be exceptions where this doesn't hold.

Some data types, though, cannot be bounded. A Facebook post that has gone viral and suddenly gets hundred times more comments than expected; an audit trail of a popular document that grows over time; or a list of favorite songs in a playlist website. While there can be some expectations on normal usage in those scenarios, you still need to allow them to grow well beyond that when necessary.

There are two roads for dealing with unbounded data sizes:

- Using capped documents, so all documents are guaranteed to stay small
- Saving some pieces of data separately in their own small documents and retrieving them by using queries.

Let's look at each, and then follow up with a discussion on how to deal with data whose size can be bounded.

Chains of Capped Documents

A Capped Document is a logical concept to describe a document that is never written to once it reaches a certain size. This is useful to enforce small sizes on documents; for example a document containing comments made to a blog post will stop being written to once it reaches its maximum capacity.

To accept more comments to the blog post, we can create another document and start adding comments to it. We can then chain instances together by referencing one another, or by leveraging semantic IDs for example.

Think of Capped Documents as a kind of a Linked List of documents, where each document contains a chunk of the data. When you want to display it all, you can just load them one by one using the chain that you created. For example, an audit trail for a product:

www.dbooks.org

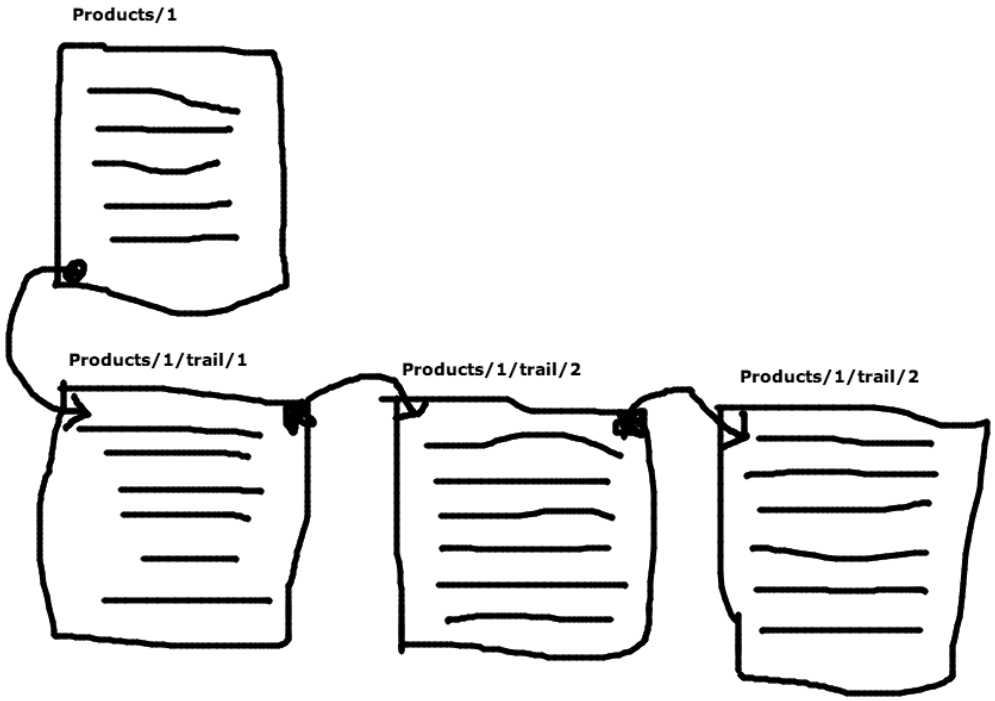


Figure 5.x: A product audit trail that is expected to grow large can be split out to a chain of capped documents

The first part of the chain – the one still with free slots for new data – is always easily accessible so you can add more items to it, and since the rest of the documents in the chain are "locked" by design, you can think of that first document as your "Unit of Change", although it isn't really the entire thing.

Deleting or updating data using this approach can be very complicated, so is indexing it. This is why it is recommended to use this approach for storing data that is used mainly for display, and is not expected to change in the future – like blog post comments, or an audit trail. For data that is more dynamic in nature, you may want to consider using a Sub-Collection.

Using a Sub-Collection

When you want to have a maintainable (read: updatable, deletable) large list of sub-documents – like a Comment document that lives in a BlogPost document – but that list is expected to grow too large to be contained within your Unit of Change, you should consider storing every document in this list as a standalone document. Because this will create a *Collection* out of those sub-documents, we call this a Sub-Collection.

There are several side-effects to this approach which are very important to keep in mind:

1. First, this breaks the “Unit of Change” pattern, since documents are not stored within the entity that logically contains them. Depending on the model, this may have some implications on data integrity considerations and so forth.
2. Another important side-effect for this approach is that accessing that data now has to be done mostly by using queries, which are Eventually Consistent. This means loading the list via an index query will not always retrieve the most recent list, and this needs to be accounted for in your business logic.

There are some cases in which it will be possible to leverage Semantic IDs prefixes and thus load all related sub-documents using a simple load-by-prefix like we’ve seen in the section about Semantic IDs. This is a possibility, but it is very scenario specific and will require a judgment call when designing the model.

BOUNDED DATA

Dealing with bounded sizes of data is something that is relatively easier to deal with. The decision making process of such cases would usually look something like this:

1. Is, or will, the list of data bloat my Unit of Change more than I want to? If not, it can just stay there; otherwise:
2. Let’s take that list and put it in another document, suffixed with some meaningful suffix, leveraging the Semantic IDs technique.
3. If that document could still become bigger than we want it to, we can result to using the Capped Documents or Sub-Collection approaches. Usually with bounded lists the Capped Documents approach would make more sense, since the number of chained documents will be relatively small.

Some boundaries, even though they are natural or very strict, are so high that you should consider them unbounded right from the start.

Key takeaway: Long lists of sub-documents can and should be broken apart

This will help keep your documents thin, and your day-to-day work with your Units of Change faster and easier. Whether you use Sub-Collections or Capped Documents to store those lists separately of your Unit of Change,

Maintaining those lists separately also come with the major benefit of being able to properly page through the list. A human user can’t really process large amounts of data at once, and any system should really try and give them data in small chunks. If the list was big enough that you had to take it out from your Unit of Change, you will most probably want to give your user the ability of paging through it – and its very easy to do with either approaches.

Using the Sub-Collection approach you also get all the benefits of a standard collection, like the ability to query on any value, sort results and so on.

www.dbooks.org

5.3.2 Concurrent Writes and Optimistic Concurrency

It is very common in modern systems for the same resource to be accessed by multiple users concurrently. Since a good document-oriented model revolves around documents representing *Units of Change*, concurrent changes to a domain entity means one document may be changed by different users at the same time. While this is a perfectly acceptable behavior, it could potentially lead to interesting challenges, ones that you need to plan ahead for.

As a result, some modeling decisions can be influenced by concurrency concerns, or the consistency level required by documents. This may sound a bit vague, so in this section we will look at a couple of examples, to give you some tools and so you know what to expect.

Let us start by looking at one scenario in particular where proper attention is required. If two different users read the same document version at the same time, and then each is making their changes to the document and write it back to the database, we get a conflict scenario:

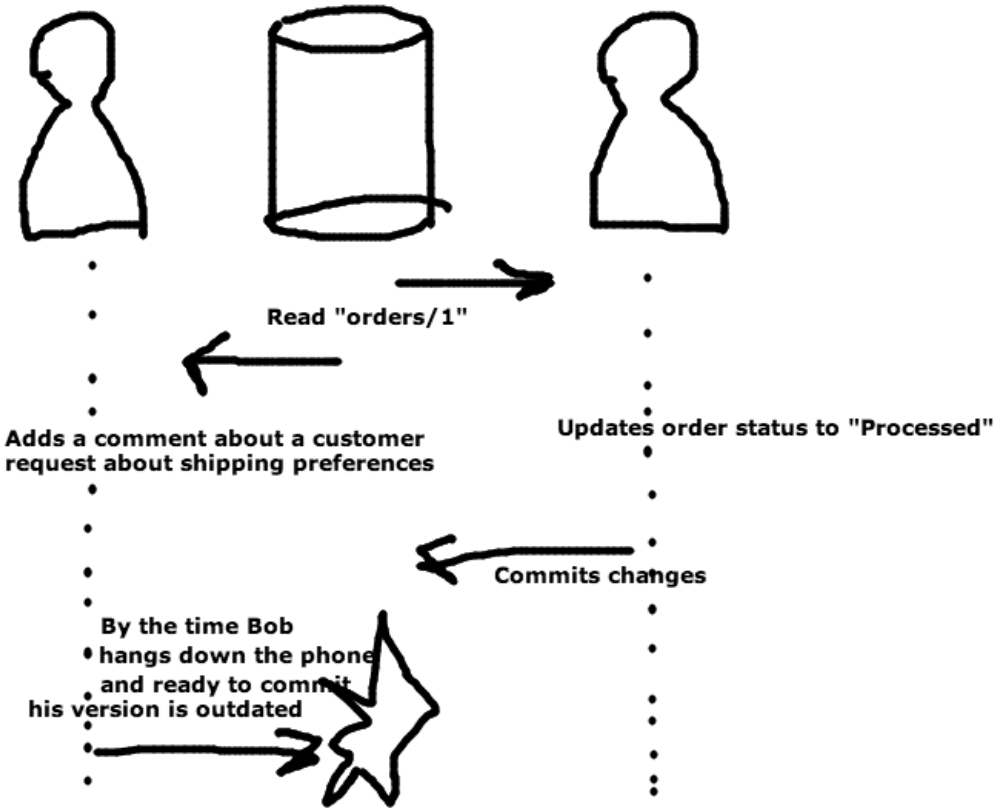


Figure 5.x: Order handling by multiple employees can create data conflict scenarios, timeline going from top to bottom

By default RavenDB uses a last-write-wins approach, so if two users concurrently load and change the same document, only the last change is persisted and the write by the first user discarded. This effectively means every change made by the first user is lost forever.

In cases where it is important to not lose data like this, like in our scenario in figure 5.x, you should be using Optimistic Concurrency. Enabling this on a session will verify the version the current session holds matches the current version on the server, and in case it does not it will throw `ConcurrencyException`. By catching the exception you can then retry the operation:

```
int retries = 5;
while (retries > 0) #A
{
   try
   {
        using (var session = docStore.OpenSession())
        {
                session.Advanced.UseOptimisticConcurrency = true; #B
                var order = session.Load<Order>("orders/1"); #B

                order.Status = "Processed"; #C
                order.SaveChanges(); #D
                break; #E
        }
   } catch (ConcurrencyException e) {
        retries--;
   }
}
```

#A – We want to retry the operation multiple times
#B – Loading the document after opting-in for Optimistic Concurrency. This is required on a session basis.
#C – Making the change to the document, with all required business logic
#D – Persisting the changes – this will throw a ConcurrencyException which we catch and retry
#E – The change was persisted successfully, so we can break out of the retry loop

Optimistic Concurrency guarantees a document is only written if we can guarantee we hold the latest version of it, otherwise it fails the transaction. Obviously we can add a retry logic as well, like shown above. This can be used as a tool when designing a model, and we will now see two examples of this.

Data Verification and Constraints

In recent years there is a growing number of social meetings that are arranged using on-line websites like Meetups.com or Facebook events. User group meet-ups or social engagements sometimes take place in small venues like coffee shops or conference rooms. Often times you want to allow for registration, but also want to make sure registration is closed once the venue capacity has been reached.

Using Optimistic Concurrency you can guarantee registration will only succeed if there is still free room, if you hold the entire registration status (registered users and free seats count) in one document. Since RavenDB guarantees strong consistency model when working with one document directly and not through indexes, you can guarantee parallel registrations. This can be done easily, by checking the number of registrations while saving the registration and

www.dbooks.org

enabling Optimistic Concurrency to fail the write transaction if the document has been changed on the server while we were adding a new registration locally.

Protip: Allowing for drifts

The difference between modeling for immediate consistency like this case for event registration versus eventual consistency which can be achieved by querying for registration records is just a matter of a business decision.

If registrations were counted using a query on a registrations collection, the number isn't guaranteed to be 100% up to date, but even on a fairly large system it will be right most of the time. The question is how critical it really is to have it correct all the time? In those times that it isn't, what's the worst thing that could happen?

When we discussed Eventual Consistency back in chapter 3, we mentioned real-world scenarios are eventual consistent by their nature. This can be seen here as well.

Anyone arranging events, or meet-ups, knows that large percentage of registrants don't show up. This is especially true for free events, but also for paid events. So how bad really is it to allow for drifts in the registration count? Worst case some people would have to stand…

UNIQUE CONSTRAINTS

Another use of optimistic concurrency is for honoring unique constraints. Let's say you have many users in your system, and you want to make sure an email address is only used once, and there are no multiple users using the same email address.

Since queries in RavenDB are Eventual Consistent, the only property of a RavenDB based system that you can rely on for consistency is document IDs. If you will use the user's email as part of the User document ID, you can use Optimistic Concurrency to guarantee such user doesn't exist yet:

```
try
{
   using (var session = docStore.OpenSession())
   {
        session.Advanced.UseOptimisticConcurrency = true;
        session.Store(user, "users/" + user.Email);
        session.SaveChanges();
   }
} catch (ConcurrencyException e) {
        #A
}
```

#A – This means there already is a user with this email address, so you should ask your new registrant to maybe file the "Forgot Password" feature

But using this approach marries the user in your system with the email he used when registering. While this does actually make sense, many times users need to change their email address and will be very unhappy to discover this will require them to lose their history with your system.

To get away with that, we can use marker documents, and use the fact that Optimistic Concurrency will fail an entire write transaction even if one of the documents in it is not up to date. This approach will look like this:

```
try
{
   using (var session = docStore.OpenSession())
   {
        session.Advanced.UseOptimisticConcurrency = true;
        session.Store(user); #A
        session.Store(new {}, "users_uc/" + user.Email); #B
        session.SaveChanges();
   }
} catch (ConcurrencyException e) {
}
```

#A – We store the User object normally, using RavenDB's automatic ID assignment mechanism
#B – We store a marker document (empty in this case) under an ID with his email address. As before, if this marker exists already, we will get a ConcurrencyException

In chapter 8 we will look at a way to automate this process, using the Unique Constraints Bundle as a RavenDB extension, but as you can see – it is quite straightforward setting this up yourself.

5.4 Summary

When using a technology, it is very important that you use it the way it was intended to be used. This way you make sure you can use the entire feature set it offers, and can squeeze every last bit of performance.

While relational databases are perfect for applications that need to work with tables and rows like financial applications, document-oriented databases are meant to store and retrieve documents. You can only take full advantage of the database system when you speak its language – in our case you can only take full advantage of RavenDB when your model is assembled of documents.

In this chapter we discussed what makes a document in a system. We've seen how it makes the most sense to find *Units of Change* within the domain and then have them represented in the database as documents. Since changes to data in RavenDB is expressed in documents (add a document, update a document, delete a document), the Unit of Change concept maps beautifully to operations on documents RavenDB is very familiar on how to do.

Identifying the *Units of Change* for a document-oriented data model is actually not that hard. If you have some Domain-Driven Design experience, you should already be familiar with the methodologies of the process of finding and defining the Aggregate Roots in a domain. But even if you don't, we demonstrated how we can identify transactional boundaries and use that to define a *Unit of Change*. This process becomes faster and easier as you gain experience with it, but most people agree it just requires common sense and in the end document-oriented models represent their business concerns much better than other models they used before with relational databases.

www.dbooks.org

Admittedly, sometimes the *Units of Change* in a model don't just pop-up from the domain. And even when they do, we still have other considerations in play which will change the "natural" model to be a bit more artificial in order to provide a better experience yet. In the chapter we looked at various such considerations, and how we can deal with them.

In the chapter we also discussed how to handle relationships between entities, or documents. How simple string references, which represent loose relationships, can still empower very strong capabilities in terms of cross-document querying and efficient loading of linked documents.

Like we mentioned several times, arriving at a strong and concise model is very important for powering healthy and strong applications. While the amount of variables in this process may be overwhelming at first, I can promise you all you need is a couple of exercise and then this will become much simpler.

Next up are discussions on more advanced querying capabilities in RavenDB, as well as discussions on scaling out and various other advanced use cases. We will not talk much about modeling in those chapters because we will focus at the topics at hand, but modeling still plays an important role there as well. The better the model is, the more powerful queries are, and the better scaling out story you will have. With that in mind, let's move on for more advanced capabilities of RavenDB.

6

Full-text, geo-spatial and reporting queries

In this chapter:

- Understanding full-text search
- Basic full-text search with RavenDB
- Additional capabilities provided by full-text search
- Reporting and ad-hoc queries
- Geo-spatial queries

One of Google's strengths is the quality and easiness of data retrieval. The ability to use a simple multi-word query and bring back results that are relevant is quite an art. The fact that they do this in milliseconds on unimaginable stocks of data is another amusement, but all agree Google redefined the way we think of search and data lookup by introducing this minimalistic search box close to 2 decades ago. Since then many followed, and nowadays users learned to expect a similar experience in every application or website they interact with.

Imagine you could provide such abilities to your users – just letting them type something and be able to act on that to provide them with what they were looking for. In fact, you should probably avoid at all costs not imagining that, unless you are targeting users who are not familiar with Google, Amazon, eBay and many other websites that do allow this interaction, and do that very well.

In this chapter we will deal with going the extra mile with queries. The ability to perform full-text search queries to allow users to find products based on even one word from their description or titles; enabling drilling-down in results, so to give the user better context of the search results without paging; suggesting the user better terms for search and enabling auto-

www.dbooks.org

complete for when he types his query; allowing for location-based searches; all those and more will no doubt enrich the usage experience for your users. All you have to do is learn about those capabilities, and use them wisely.

We start by looking at full-text searches, which is the bread and butter for Lucene – the library behind RavenDB's indexes. This is what it knows to do best, and at first we will learn about how to unleash its power. We will then look at some of the interesting options it opens up – like detecting typos and suggesting replacement terms or suggesting similar documents.

Next we will learn some about Facets, which can be used for giving a breakdown of query results. When used for display, this gives a lot of insights into the results, and can help users quickly refine their search.

This will be followed by a discussion on reporting queries – what are they and how to approach them. We will learn of the tools that RavenDB provides to perform such queries, and also when to delegate it to other tools not in RavenDB's realm.

We will wrap up by discussing geo-spatial search, which is a powerful feature that enables indexing of shapes on earth (a location, an area, and so forth) and then querying for them using spatial relations – an interaction between two shapes on the globe.

6.1 Full-text search queries

All query types we've seen up to this point were doing exact value matches. This is an important detail when dealing with String values, as the value lookup will always be exact, and will include spaces, punctuation marks and so on.

Let's demonstrate this using the bookshelf example from chapter 3:

(#1)
Life of Pi
by Yann Martel
180 pages
$15.00

(#2)
Angels and demons
by Dan Brown
500 pages
$12.99

(#3)
Harry Potter and the Goblet of Fire
by J.K. Rowling
400 pages
$25.00

(#4)
Deception Point
by Dan Brown
350 pages
$11.99

Book titles	
Angels and demons	2
Deception Point	4
Harry Potter and the Goblet of Fire	3
Life of Pi	1

Author names	
Dan Brown	2, 4
J.K. Rowling	3
Yann Martel	1

Pages	
180	1
350	4
400	3
500	2

Price	
11.99	4
12.99	2
15.00	1
25.00	3

These are the book numbers – it points us to the book the value was taken from, can be more than one book number if necessary

In this familiar example we can see the books to the left, and the resulted index to the right. The Author Names and Book Titles are each a multi-word String values which are being stored as one value, and that includes spaces and punctuation marks. If you wanted to query the index for them you'd have to use the exact value as it appears in the index. Querying for just "Rowling" will return nothing, because the index only knows about "J.K. Rowling", and querying for "Deseptoin Point" will also yield no results because of the spelling errors.

A common requirement with queries on String values is to look for only one word within it. Even for single word values, often you'd want to still get results even if you misspelled the term when querying.

This can become even more complex, for example non-English words containing diacritics like the French word *tréma* have to be typed in a query exactly the way they were indexed – and that can be a problem for non-French speaking people. Other than this issue, there may be

www.dbooks.org

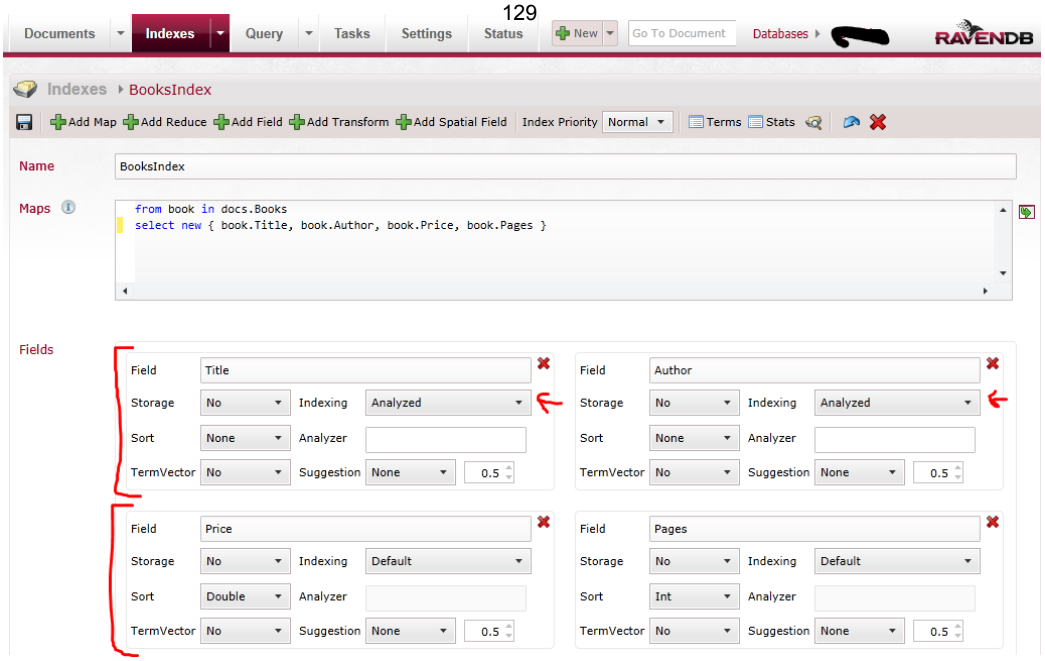


Figure 6.x: A simple-Map index on the Books collection, with 2 text fields that are defined as “Analyzed” to enable full-text search on them (up), and 2 numeric fields which are not analyzed and have some custom sort configurations enabled (bottom).

The same can be done with an index definition that exists in code:

```
public class BooksIndex : AbstractIndexCreationTask<Book>
{
    public BooksIndex()
    {
        Map = books => from book in books
                       select new {book.Title, book.Author, book.Price,
book.Pages};

        Index(x => x.Title, FieldIndexing.Analyzed); #A
        Index(x => x.Author, FieldIndexing.Analyzed); #A

        Sort(x => x.Price, SortOptions.Double);
        Sort(x => x.Pages, SortOptions.Int);
    }
}
```

#A – This tells RavenDB that the Title and Author index fields, which were mapped from document properties using the Map function above, are to be enabled for Full-Text Search.

In the next section we will see why it is called “analyzed”. But before we do, let’s look at the type of queries that this enables us to do.

Once an index has one analyzed field or more, you can use the Search operator to perform full-text searches on those fields:

```
var books = session.Query<Book, BooksIndex>()
                   .Search(x => x.Author, "Rowling") #A
```

```
                    .ToList();

books = session.Query<Book, BooksIndex>()
                    .Search(x => x.Author, "Rowling Brown") #B
                    .ToList();
```

#A – The Search operator takes a field name and a term to look for in it
#B – By default multiple terms will expand the search, and not limit it; this query will return all books with Rowling or Brown in their name.

Combining multiple Search clauses to execute searches on multiple fields is possible, but requires paying attention to the relation between them – is it an OR, an AND or NOT operation:

```
var books = session.Query<Book, BooksIndex>() #A
                    .Search(x => x.Author, "Brown")
                    .Search(x => x.Title, "angels", 1, SearchOptions.And)
                    .ToList();

books = session.Query<Book, BooksIndex>()#B
                    .Search(x => x.Author, "Brown")
                    .Search(x => x.Title, "angels", 1, SearchOptions.And |
SearchOptions.Not)
                    .ToList();
```

#A – Searches for books by author whose name contains "Brown" and title containing "angels". Using our sample bookshelf, output will be "Angels and Demons".
#B – Searches for books by author whose name contains "Brown" and title does NOT contain "angels". Using our sample bookshelf, output will be "Deception Point".

Analyzed fields allow for all of Lucene's great full-text search querying capabilities to be used on text stored in them. For example, using prefixes and wildcards we can query for documents without knowing the exact spelling of them:

```
var books = session.Query<Book, BooksIndex>()
                    .Search(x => x.Author, "Rowli*", 1, SearchOptions.Or,
            EscapeQueryOptions.AllowPostfixWildcard).ToList(); #A

books = session.Query<Book, BooksIndex>()
             .Search(x => x.Author, "bro?n", 1, SearchOptions.Or,
     EscapeQueryOptions.RawQuery).ToList(); #B
```

#A – Executes a Lucene prefix query on the field, so all documents with a word starting with "Rowli" will come up, in our example books by J.K. Rowling will be in the results.
#B – A wildcard query is executed, and all books by Dan Brown are returned.

Key takeaway: Analyzed text fields enable full-text search

An "analyzed" field in RavenDB means the text stream in that field is broken down to individual terms, and thus can be searched using full-text search queries.

However, not every text field has to be analyzed.

Full-text queries are for scenarios when you want to enable a search on text field that will involve one word or several, and when exact value matching against it is irrelevant. For example, a title of a book, or a description field for a product in an e-commerce website. If

www.dbooks.org

they will ever be searched, this will only be done using one or two words, never the entire string value.

Some text fields, like fields used to hold SKUs or phone numbers, do not need to be set as Analyzed. As a matter of fact, doing this will most likely make standard queries (the exact match type queries) stop working, as the index entries produced from the field will now be different, as we will explain later in this section.

Pay attention to what fields you define as Analyzed and what fields need to stay untouched. You can always index a string from a document into multiple fields when you need both query types to work – exact match AND full-text queries:

```
    public class BooksIndex : AbstractIndexCreationTask<Book>
    {
        public BooksIndex()
        {
            Map = books => from book in books
                           select new {book.Title, book.Author, AuthorForSearch
= book.Author};

            Index(x => x.Title, FieldIndexing.Analyzed);
            Index("AuthorForSearch", FieldIndexing.Analyzed);
        }
    }
```

The biggest challenge with full-text search is achieving good relevance, that is – making sure the returned results satisfy the user's query. While RavenDB is not a search server product, it does provide very good full-text search support and the ability to customize and fine-tune its search capabilities using Analyzers. Analyzers are the Lucene part that make full-text search possible, through what we call the *analysis process*, and this is what we are going to look at next.

6.1.2 Lucene's analysis chain

When Lucene receives from RavenDB a text to index into an Analyzed field, it passes it through a process that prepares it to be added to the index, so it can be queried. This process is called Analysis, and Analyzers are the part of Lucene that define the Analysis chain.

The *Analysis process* is being performed by Lucene in a 2-step process: *tokenization* and *filtering*. Tokenization is the process of splitting raw stream of text into individual tokens, and Filtering is an optional processing step which allows for removing tokens or altering them in different ways. This is demonstrated in figure 6.x:

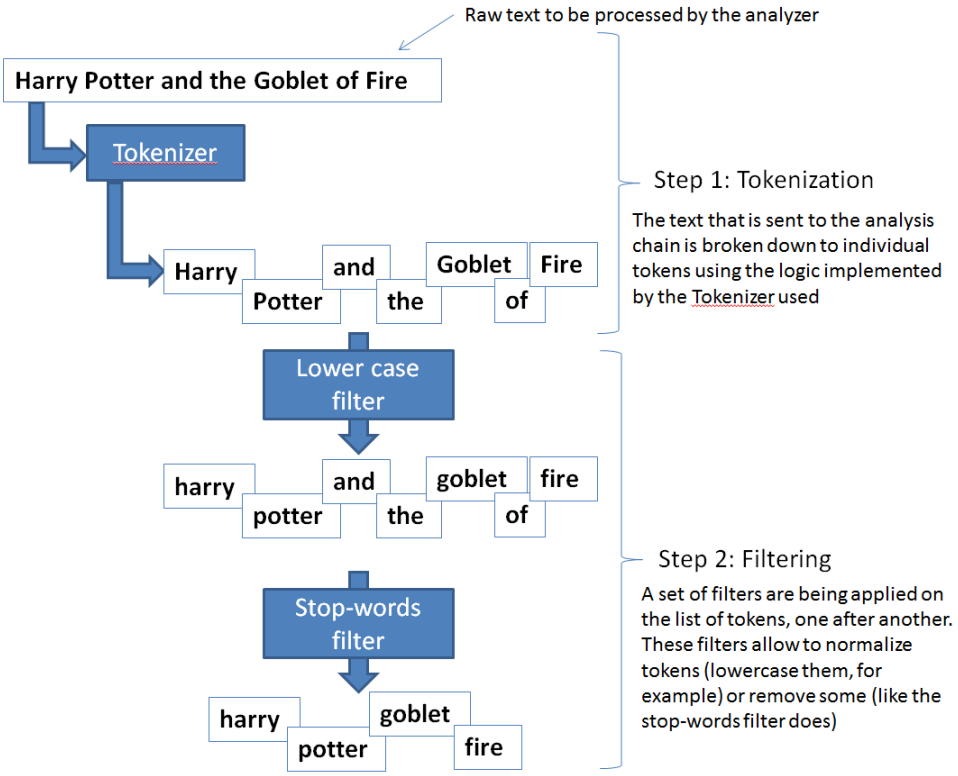


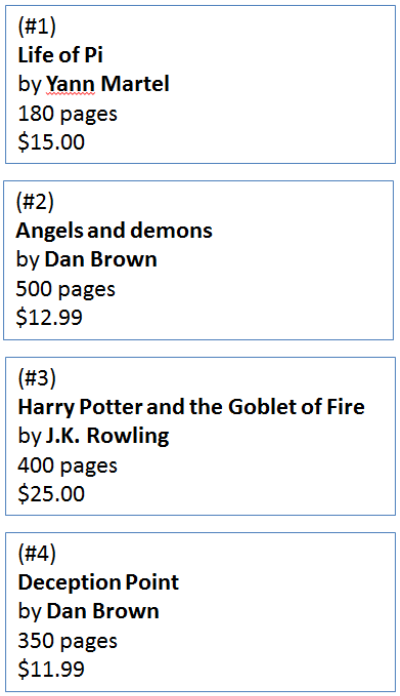
Figure 6.x: Demonstration of Lucene's analysis process in action, using a white-space Tokenizer and 2 filters, as they operate on a given sample text. The process may omit tokens, add new ones, or change existing. When it is done doing its job, the tokens are saved to the index.

The first step of any Analysis process is splitting the text into individual tokens. We say tokens and not words because a token doesn't have to be a word – it really all depends on the logic of the *Tokenizer* being used. This transforms a text field of a RavenDB document into a list of token.

After the stream of text has been tokenized, the array of produced tokens goes through a set of *Filters* that are in charge of normalizing them. Common normalization patterns include lowercasing, removal of diacritics, and removing tokens that may damage retrieval quality. This step is optional, but a good Analyzer will make smart use of it to improve search performance and quality of retrieval.

Once done going through this chain, the tokens are put in index field as terms, like shown in figure 6.x below:

www.dbooks.org



Book titles	
angels	2
deception	4
demons	2
fire	3
goblet	3
harry	3
life	1
pi	1
point	4
potter	3

Author names	
brown	2, 4
dan	2, 4
j	3
k	3
martel	1
rowling	3
yann	1

Figure 6.x: An illustration of the index fields for Book titles and author names once they've been defined as "Analyzed". Notice how the terms are now broken down to individual words, are lowercased, and stop-words like "and", "the" and "of" have been omitted.

The decision of which *Tokenizer* and chain of *Filters* to use really depends on the corpus, the language, the content and the expected queries. To ensure quality of query results is high it is recommended that you hand pick this chain to suit your requirements.

An Analyzer is a Lucene component that encapsulates the work done by a Tokenizer and subsequent filters (if any). This is where the name "analyzed field" is derived from. RavenDB comes bundled with several analyzers, each of which defines its own analysis chain that you can use out of the box. You will find those out-of-the-box *Analyzers* mostly sufficient for basic requirements, but once you start having more refined requirements for full-text search queries you'll most likely need to roll your own analyzer.

6.1.3 Customizing an Analyzer for a field

Lucene comes bundled with several *Analyzer* implementations out-of-the-box, and those are available directly from RavenDB as well. To tell RavenDB which Analyzer to use in an analyzed

field, just put the fully-qualified name of the *Analyzer* class in the matching box in the Management Studio when editing the index definition (Figure 6.x):

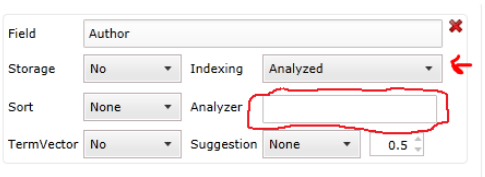


Figure 6.x: Specifying an analyzer name for a field in an index

Or from code, using the same index definition that we had before only we now hand-pck the analyzer to be used for the Author field:

```
    public class BooksIndex : AbstractIndexCreationTask<Book>
    {
        public BooksIndex()
        {
            Map = books => from book in books
                           select new {book.Title, book.Author, book.Price,
book.Pages};

            Index(x => x.Title, FieldIndexing.Analyzed); #A
            Analyze(x => x.Author, "SimpleAnalyzer"); #B

            Sort(x => x.Price, SortOptions.Double);
            Sort(x => x.Pages, SortOptions.Int);
        }
    }
```

#A – By default, RavenDB uses Lucene's StandardAnalyzer for all fields defined as Analyzed
#B – We specify a different analyzer for the Author field, in this case Lucene's SimpleAnalyzer, to customize processing of text coming from this field. More specifically, we are not interested in stop-word removal on this field.

Here's a list of notable analyzers that are available from Lucene. For those analyzers it's enough to put the class name alone for RavenDB to locate them:

```
The quick brown fox jumped over the lazy dogs, bob@hotmail.com 123432. #A

KeywordAnalyzer:
[The quick brown fox jumped over the lazy dogs, bob@hotmail.com 123432.]

WhitespaceAnalyzer:
[The] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dogs,]
[bob@hotmail.com] [123432.]

SimpleAnalyzer:
[the] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dogs] [bob]
[hotmail] [com]

StopAnalyzer:
```

www.dbooks.org

```
[quick] [brown] [fox] [jumped] [over] [lazy] [dogs] [bob] [hotmail] [com]

StandardAnalyzer:
[quick] [brown] [fox] [jumped] [over] [lazy] [dog] [bob@hotmail.com] [123432]
```

#A - given this sentence, the following demonstrates the list of tokens each analyzer generates

For fields that are not set as Analyzed, a lowercasing KeywordAnalyzer is used. This is why a non-analyzed field requires queries to exactly match the values that were indexed, except from case which is lowercased automatically to make standard querying a bit more straight-forward.

Rolling your own Analyzer is a fairly straight-forward task, but I haven't included a discussion about it in this book because it requires many steps that are just out of the scope of this book.

Know your analyzer: Searching for stop words as example

The analyzer you use will influence the query results you will get.

When there is no analyzer specified for an analyzed field, Lucene's StandardAnalyzer is used. StandardAnalyzer removes stop-words by default, and that is a common gotcha that people tend to get confused about. If searching on words like "and", "it" etc is important for you, make sure to use another analyzer.

6.1.4 Results highlighting

Full-text search queries are about finding documents using one or several words that appear in them. But getting the document doesn't show you where the word appeared in that document – the document is in the search results because it matched the query, but by default there is no indication of which word actually matched.

When you search Google, the results show up with highlighted snippets of text, where the search terms are highlighted in the snippet. This gives great sneak-peek of the results, so you know which to click and great feedback to the user on his query and whether he needs to refine it.

This is also possible to do with RavenDB. For every full-text search query you can add the Highlight operator to ask for highlighted snippets:

```
(code)
```

Highlighted words may not necessarily match the words in the query, depending on the analyzer used. This is also why it is preferred to highlighting manually on the result text.

Once query results have been returned, you can evaluate how the query performed, and having highlighted snippets can help a lot with that. If you got too few results, or completely irrelevant results, something may be wrong with the query – for example you may have queried for a term that does not exist in the index, or perhaps you had a typo in your query. In this case, RavenDB can help you detect that and find the correct term to use in your query.

6.1.5 Suggesting alternate spellings for search terms

Imagine the following scenario: You go on a website, and look for some info on an acquaintance. You search once, twice, but can't find it, so you give up. A few days later you discover Danny is actually spelled Danni, but it is already too late. The website you gave up on lost you – a potential user, or in some cases a paying customer.

Sounds familiar? This happens time and again with too many websites and applications. Trying to guess what the user was actually looking for and trying to provide him with meaningful alternatives is being considered by many developers overkill. By doing so, they don't realize the full potential of their application and lose both customers and business.

A very popular approach by search engines is to try and guess term suggestions when they detect search results may not be satisfactory. You are probably familiar with Google's "did you mean?" when you make an accidental typo. Well, it's not that Google is trying to mock you or anything; it's just that it was able to find a higher scoring term with a certain edit distance from the term you actually typed.

RavenDB provides a very easy and intuitive way of providing alternative terms for queries that returned little or no results, just like Google's "Did You Mean?". When a full-text query returns zero results, or when you have other indication of bad results being returned for a query, you can ask RavenDB to provide suggestions for the term or terms used in that query:

```
var query = session.Query<Book>("BooksIndex").Search(x => x.Author, "brwon");
var results = query.ToList(); #A

if (results.Count == 0) #B
{
     var suggestions = query.Suggest(); #C
     foreach (string suggestion in suggestions.Suggestions)
     {
            Console.WriteLine(suggestion); #D
     }
}
```

#A – Creating the query and issuing it to get results. We keep it aside so we can reuse it for suggestions if necessary.
#B – No results will be found for "brwon" in our data set, so we ask for suggestions
#C – Reusing the query to issue a suggestions query
#D – Suggestions will return a list of terms. Each of them can be used to notify the user, or even re-issue the query, all depends on what you see fit for your application.

6.1.6 Finding related documents using MoreLikeThis

Let's continue with the e-store model. Many big online stores - Amazon for instance - maximize their profits by showing users "related products" in product pages and during check-out. Similarly, websites like CNN can get users to spend more time on their website by showing links to related content at the bottom of an article. Unlike what you might think, this doesn't require a full editorial staff to through all your content. It can be done fairly easily by comparing data in one content entity to the rest of the content, and showing the highest ranking ones to the user. So the question remains – how can you do this easily and efficiently?

www.dbooks.org

RavenDB exposes Lucene's MoreLikeThis functionality, which creates a full-text search query from a document and uses that to find similar documents. The result is documents that are similar to the original document, based on the terms in the query document and their frequency. To do this you need to approach the RavenDB server with a document ID, and tell it which index to use for the comparison:

```
var list = session.Advanced.MoreLikeThis<Book>(
     "BooksIndex", #A
     new MoreLikeThisQuery
     {
            DocumentId = "books/2", #B
            Fields = new[] {"Title", "Author", "Description"}, #C
            MinimumWordLength = 2, #D
     }
);
```

The result of calling this method is an array of book objects RavenDB deems similar to the book used as a query. To get the most out of this feature, you want the lookup to be performed on text properties like title and description, and make sure they were indexed as Analyzed. Doing so will utilize RavenDB's full-text search capability behind the scenes, and will maximize relevance of the products considered relevant.

It is also possible to perform fine-tuning and adjustments, for example to hand-pick what properties to use for the actual comparison, and what is the minimum or maximum word length. All of those are passed as parameters to the MoreLikeThisQuery object that is passed to the method, like shown above.

(segue)

6.2 Taking advantage of the inverted index

The Lucene index at the heart of every RavenDB index is an inverted index. Like we demonstrated several times by now, it has all the possible values for every field, and each of those values point to all documents which have that value for that field. It is called an "inverted index" because instead of looking at a document to see all the terms inside it (what we humans call "reading"), we look at a term to find all documents that contain it.

The inverted index plays a crucial part in powering the query and search capabilities of RavenDB. The fact that it is inverted has a tremendous effect on performance, as efficient data-structures that allow for quick term lookups can be used. But as it turns out, it also contains a wealth of information we can use.

Since every inverted index is in fact an alphabetical list of terms that appear somewhere in some document in the system, grouped by the field they appear in, in effect it becomes a dictionary of sorts that can be used to offer possible values for a field. It is effectively the vocabulary for a field in a collection. And because a list of matching documents is attached to each of the terms in that list, we know how many documents has that term; or put differently, how many matches a query for that term would yield.

In this section we will look at various ways we can access the Lucene index directly and take advantage of all the information it contains to do a lot of interesting stuff. By enumerating all terms in a field we can create auto-complete features, and by additionally counting the documents which have those values, or ranges of values, we can allow for drill-down like functionality for queries.

There is a slight difference in the possible uses between analyzed fields and fields which are not set as analyzed. Accessing the list of terms and the data it provides for not-analyzed fields will effectively look at all the possible values of that field; if the value is in the list, a document exists somewhere in the system with that term as the entire value of that field.

For an analyzed field this list will contain all terms that appear in a field in a document, but it's not guaranteed the document containing that term will have only that term in the field.

This difference between analyzed and non-analyzed fields is important to remember, as it will often bite you when trying to do various things with the terms in the index.

6.2.1 Accessing the list of terms

Iterating through the list of terms in an index, in the lexicographical order in which they are stored, is easily done. By a simple call to the DocumentStore you can ask RavenDB to give you that list by specifying an index and a field name:

```
IEnumerable<string> terms = store.DatabaseCommands.GetTerms( #A
                                    "BooksIndex", #B
                                    "Author", #C
                                    null, #D
                                    128 #E
                                    );
```

#A – GetTerms() is available from the DocumentStore class, there is no need to open a session to access it. In case you only have access to a session object, the DocumentStore object is also available from the session through session.Advanced.DocumentStore.
#B – The name of the index to be used
#C – The name of the field in the index
#D – The term to start iterating from – used for paging.
#E – Amount of terms to return.

While this makes the most sense to be used on fields that are not analyzed, this works the same for analyzed and not-analyzed fields.

When iterating through indexes that contain large lists of terms it's advised to use the paging abilities, by providing the last term that was seen in the previous page, and using moderate page sizes.

Possible uses include menu listings and auto-complete features, but usage really depends on the way the data is represented in the index – which in turn depends on how it was indexed and analyzed.

www.dbooks.org

EFFICIENT AUTO-COMPLETE FEATURE

Let's look at how we can use the GetTerms() functionality to implement an efficient auto-complete feature. Given a user input, we want to display a list of possible values that user may be meaning to type.

Because we are accessing the terms list in the index directly, this is much cheaper to do than a full-blown query, and therefore it makes perfect sense to use for as-you-type suggestions like Google does, so do many other popular applications. For the same reasons, it is also a very basic approach and for many applications you will find it is not enough. However, it does work:

```
if (userInput.Length > 2) { #A
   IEnumerable<string> terms = store.DatabaseCommands.GetTerms(
                                    "BooksIndex",
                                    "Author",
                                    userInput, 5
                                    ); #B
   foreach (var term in terms)
   {
        if (!term.StartsWith(userInput)) #C
               continue;
        Console.WriteLine("\t" + term);
   }
}
```

#A – We only want to start suggesting when the user input is long enough to mean something; 2 characters is fine
#B – Querying the field that contains the terms we want to lookup – in this case authors
#C – We only display terms which actually start with the prefix the user typed – more on this in the gotchas below

There are 3 gotchas with this approach we should notice:

1. If the field we used this on is not analyzed, we will get complete values. If it was analyzed, and the value the user was looking for was multi-word, the list of terms we will get will only be assembled from the first word, because analyzed fields go through tokenization.
2. Speaking of analyzed fields, if your field is analyzed some terms may get normalized in a way that will make them look different. StandardAnalyzer, for example, makes sure not to tokenize e-mail addresses and therefore you'll have email addresses show up as terms. Other analyzers do not preserve e-mail addresses and therefore will not have them as terms. As always, pay attention to the analyzer used.
3. We ask for the list of terms starting with a prefix, but a term with that prefix may not exist in the index, or maybe it will exist but the term immediately preceding it will have a completely different prefix. The GetTerms() method asks the index for the terms by the order they appear in the index, it does not do any filtering whatsoever.

(mention other approaches? – using prefix queries for example)

(segue)

6.2.2 Faceted search

You know a good site search interface when you see one. Take eBay or Amazon for example, where shopping and drilling-down in results is so easy. You type a search query and then click your way through a list of options. This is what we call Faceted Search, as demonstrated here (Figure 6.x):



Figure 6.x: Faceted search in action **(we should recreate this image to avoid copyright issues)**

Faceted search is a way to efficiently and dynamically break query results down to categories and classifications and perform dynamic aggregation operations on results within those categories. We can then use that data to allow for visually drilling-down into result sets like demonstrated above, among other things. As a general rule, giving a birds-eye view of query results is much preferable to forcing them to page through them all.

RavenDB provides facilities to perform this type of query. The gist of Faceted Search with RavenDB is quite simple – you query, and tell RavenDB the criteria in which results should be faceted on. RavenDB in turn, after performing the query, will go through the results and create the facets in an efficient manner.

Asking for facets is as simple as creating a query as usual, and calling ToFacets() on it along with the definition of facets to be returned:

```
var facets = session.Query<Book>("BooksIndex") #A
                    .ToFacets(new List<Facet>
                    {
                            new Facet {Name = "Author"}, #B
                            new Facet
                            {
                                    Name = "Price_Range",
                                    Mode = FacetMode.Ranges, #C
                                    Ranges = new List<string>
                                    {
```

www.dbooks.org

```
                                        "[NULL TO Dx20.0]",
                                        "[Dx20.0 TO Dx50.0]",
                                        "[Dx50.0 TO Dx200.0]",
                                        "[Dx200.0 TO Dx400.0]",
                                        "[Dx400.0 TO NULL]",
                                    }
                                },
                            });

foreach (var facetResult in facets.Results) #D
{
     Console.WriteLine("* " + facetResult.Key);
     foreach (var facetValue in facetResult.Value.Values)
     {
            Console.WriteLine("\t" + facetValue.Range + ": " +
        facetValue.Hits);
     }
}
```

#A – Faceted search starts with a standard query, against an index. If we wanted we could add a Where or Search operator, but our data-set is small enough to ignore that for now
#B – A standard facet, will collect all possible values for that field and count occurences
#C – A range facet, will collect and count all values for this field in the results, splitting this into the ranges defined below. The ranges use the query syntax for numeric fields we covered in chapter 3.
#D – Printing out the results

Executing the above code on our sample data set will produce the following output:

```
* Author    #A
        Dan Brown: 2
        J.K. Rowling: 1 #B
        Yann Martel: 1

* Price_Range #C
        [NULL TO Dx20.0]: 3
        [Dx20.0 TO Dx50.0]: 1
        [Dx50.0 TO Dx200.0]: 0
        [Dx200.0 TO Dx400.0]: 0
        [Dx400.0 TO NULL]: 0
```

#A – The values for the facet on the Author field
#B – The Author field in this case was using Lucene's KeywordAnalyzer, which doesn't tokenize and also doesn't lowercase, this is why we are seeing the original values. The default RavenDB analyzer would have lowercased this, and any other full-text search analyzer would have produced tokenized results.
#C – The results for the range facet on the Price field. Notice how each range is displayed with the number of hits it has.

Faceted search is a very nice way to allow for drilling down in query results and get many insights on them. It is a nice segue to have into reporting queries – the ability to slice and dice data on the fly using changing criteria, which is our next topic.

6.3 Reporting queries

All queries in RavenDB work on pre-computed indexes. Back in chapters 3 and 4 when we were discussing RavenDB indexes and queries, we mentioned you cannot query RavenDB on data

that hasn't been indexed. For simple queries RavenDB is able to automatically create indexes on the fly, but ultimately this means no query can return results without a matching index.

We also mentioned there are generally two types of indexes: simple Map indexes, which just map document properties to index fields to make them searchable; and Map/Reduce indexes, which can take some document properties, perform an aggregation operation on them, and then have the result of that aggregation operation searchable.

But this doesn't cover all types of aggregation queries you might want to do in your system. Consider this list of orders (Table 6.x.):

Order ID	Customer ID	Date	# of items	Total price	Currency
orders/1	customers/5	05/12/2012	2	823.12	USD
orders/2	customers/10	02/01/2013	1	102.99	USD
orders/3	customers/1	12/03/2013	2	35.50	USD
orders/4	customers/2	01/04/2013	5	1052.00	EUR
orders/5	customers/9	20/04/2013	3	15.00	EUR
orders/6	customers/20	21/04/2013	1	14.60	USD
orders/7	customers/5	11/06/2013	4	86.99	EUR
orders/8	customers/8	08/07/2013	10	743.00	USD
orders/9	customers/5	12/07/2013	1	23.10	USD
orders/10	customers/9	01/08/2013	1	71.24	EUR
orders/11	customers/12	18/01/2014	2	75.98	USD

The table shows how the result of a simple Map function would look like on a collection of Order documents. That Map function will map the Order Id, Customer Id and all the rest of the fields in the table from the Order document where they exist as data in properties, into an index field. This data is then given to the indexing engine, and that would make the Orders searchable by that data. This enables simple queries like "give me the orders that were paid in USD", "all orders which were made in 2012", "orders with only 1 item and total price > 1000" and so on.

When we discussed Map/Reduce we explained Map-only indexes don't give us the ability to query on aggregations of the data, so queries like "the month with the highest number of products sold" or "the customers who spend the most money in our shop" are not possible to do using simple Map indexes. To be rather more exact, they are possible but they require handling the entire result set on the client side to perform the aggregation, and that is just to costly to do in most real-world scenarios.

www.dbooks.org

Instead, we've seen how Map/Reduce indexes can do those aggregation operations and have the results of those operations indexed so they could be queried on. To answer queries like the total of items bought and total order value per customer, we can create a Map/Reduce index that will group this data by customer ID and output the following data to the index (Table 6.x):

Customer ID	Orders count	Average # of items	Average price	Highest price	Lowest price
customers/1	1	2	35.50	35.50	35.50
customers/2	1	5	1052.00	1052.00	1052.00
customers/5	3	2	311.07	823.12	23.10
customers/8	1	10	102.99	102.99	102.99
customers/9	2	2	43.12	71.24	15.00
customers/10	1	1	102.99	102.99	102.99
customers/12	1	2	75.98	75.98	75.98
customers/20	1	1	14.60	14.60	14.60

As you can see, we can group the data once, and then have various aggregation operations made on it.

Try it yourself: Understanding the limitations of Map/Reduce indexes

In this section we return to our earlier discussion from Chapter 3 and talk in more depth about the limitations RavenDB indexes have. After we will be successful in explaining the limitations and the reasons for why this happens, we can look at tools to allow for the type of queries RavenDB indexes can't provide answers to.

Like in most complex topics, it's much easier to understand the material by practicing it first-hand. It is highly recommended that you try creating such an application yourself as we go along and play with querying, to see which queries require manually creating an index and which queries are just impossible to answer with the tools we've learned about so far.

If in any case you feel stuck, the complete source code for this example can be found in the sources that accompany the book.

As we progress in this section you will learn to identify what are Reporting Queries, and learn about tools which will help you answer them regardless.

The fact that Map/Reduce aggregations happen at the time indexing occurs means that once this data has been computed, it can be queried on quite efficiently. But it also means we

will need to have one Map/Reduce index for every type of aggregation query that we need to have – one index that groups by customer ID, another that groups by a currency and so on. Since Map/Reduce indexes never get created automatically, all those indexes will have to be added manually.

While we were able to reuse the above index for several queries, it only has the data to answer very specific queries. If we wanted for example to answer queries like breakdown of orders by month or currency (or both), we would have to create more Map/Reduce indexes explicitly for them. For this type of queries, which we will call Reporting Queries or Ad-hoc Queries, the aggregation capabilities Map/Reduce indexes provide just aren't enough.

Reporting queries are by-design not one of RavenDB's strengths, but it is still possible to do. Leveraging Facets, RavenDB can do quite a lot but with the expense of large memory requirements, since most of the work is done in-memory post-query. While this guarantees very fast results, this also requires large amounts of memory to be available to the server instance.

After we went in length on defining the problem, we are now ready to look at ways to work around it. In this section we will look at two different ways to perform reporting queries; one is using Dynamic Aggregations, a feature built-in with RavenDB that is capable of providing good results to those queries. The other approach will utilize tools from the SQL world, tools that have been built to deal with this type of queries, and is more suitable for reporting queries that need to operate on large sets of data.

6.3.1 Dynamic Aggregations with Facets

Dynamic aggregations, like the ones we just showcased, are possible to do with RavenDB using the `AggregateBy()` query operator. Those Aggregations are actually a wrapper around the Facets implementation we saw earlier in the chapter. Standard faceting only returns number of hits per term for every field we facet on, and doesn't really use the actual value in any way, except from when using ranges. Using the Aggregation query operators provides us a way to perform aggregation operations on the actual values, so its primary use is for numeric fields.

Here is an example for getting aggregations on the number of items in all orders, and on the total price of all orders:

```
var results = session.Query<Order, OrdersSimpleMapIndex>()
                        .AggregateBy(x => x.CustomerId)
                             .AverageOn(x => x.NumberOfItems)
                             .MaxOn(x => x.TotalPrice)
                             .MinOn(x => x.TotalPrice)
                             .SumOn(x => x.TotalPrice)
                        .AndAggregateOn(x => x.Currency)
                             .AverageOn(x => x.NumberOfItems)
                             .MaxOn(x => x.TotalPrice)
                             .MinOn(x => x.TotalPrice)
                             .SumOn(x => x.TotalPrice)
                        .ToList()
                    ;
```

www.dbooks.org

The nice thing about this is that we can change the query as much as we want by introducing `Where()` and `Search()` operators, and still keep those aggregations operators in place. This dynamic aggregation capability of RavenDB happens post-query on the server side, and as such it is not limited by the type of query submitted, or the indexes that you have defined. As long as RavenDB can answer the query without the aggregation operations, it can do the defined aggregation operations on that query.

It is important to understand that while Aggregations are dynamic and usually very fast to execute, Map/Reduce queries are much cheaper to execute. For queries that can use Map/Reduce indexes, it is recommended to use Map/Reduce. The memory requirements for dynamic aggregations are quite big, so this technique isn't recommended for frequent queries on large data sets.

For queries that can use either, prefer Map/Reduce. For high-demand queries that can't use Map/Reduce, you should consider using the excellent reporting tools that come bundled with any decent SQL database.

6.3.2 The SQL replication bundle

Earlier in the book we mentioned RavenDB doesn't try to be a silver bullet, and while it strives at being very good at what its good at, some things it just wasn't designed to do. Reporting queries is one of those things that RavenDB wasn't designed for, and admittedly not one of RavenDB's strengths.

While dynamic aggregations are supported, SQL databases are much better at executing ad-hoc queries that include aggregations. The field of producing reports out of raw data that can be represented by rows in tables is dozens of years old, and very mature. Highly optimized and tested algorithms have been developed over the course of decades to solve this problem, and those tools do a really good job with that.

RavenDB realizes this fact and while supporting this type of queries out of the box, for large data-sets and for intensive querying it is recommended to use the reporting tools SQL databases offer.

SQL replication works by defining a script on a Collection, and executing that script for every document in that collection whenever it is created, updated or deleted. Assuming we have a SQL database with the following tables created:

```
CREATE TABLE [dbo].[OrderLines]
(
    [Id] int identity primary key,
    [OrderId] [nvarchar](50) NOT NULL,
    [Qty] [int] NOT NULL,
    [Product] [nvarchar](255) NOT NULL,
    [Cost] [int] NOT NULL
)

CREATE TABLE [dbo].[Orders]
(
    [Id] [nvarchar](50) NOT NULL,
    [OrderLinesCount] [int] NOT NULL,
```

```
    [TotalCost] [int] NOT NULL
)
```

Enabling SQL replication to it is just a matter of putting a document with all the required settings to RavenDB:

```
session.Store(new SqlReplicationConfig
{
    Id = "Raven/SqlReplication/Configuration/OrdersAndLines",
    Name = "OrdersAndLines",
    ConnectionString = @"
        Data Source=.\SQLEXPRESS;
        Initial Catalog=ExampleDB;
        Integrated Security=SSPI;",
    FactoryName = @"System.Data.SqlClient",
    RavenEntityName = "Orders",
    SqlReplicationTables =
        {
            new SqlReplicationTable
            {
                TableName = "Orders", DocumentKeyColumn = "Id"
            },
            new SqlReplicationTable
            {
                TableName = "OrderLines", DocumentKeyColumn = "OrderId"
            },
        },
    Script = @"
        var orderData = {
            Id: documentId,
            OrderLinesCount: this.OrderLines.length,
            TotalCost: 0
        };

        replicateToOrders(orderData);

        for (var i = 0; i < this.OrderLines.length; i++) {
            var line = this.OrderLines[i];
            orderData.TotalCost += line.Cost;

            replicateToOrderLines({
                OrderId: documentId,
                Qty: line.Quantity,
                Product: line.Product,
                Cost: line.Cost
            });
        }"
});
```

The Script portion is where all the magic happens. It is javascript code which gets the document as a parameter, and performs the actual replication to SQL by creating objects with the data and calling methods `replicateTo[table_name]` to perform the actual data copying.

(code; segue)

www.dbooks.org

6.4 Geo-spatial search

Spatial search is about using shapes to search for shapes – using a point to find other points near it, a polygon to find intersecting polygons and so on. Being geo-spatial means all shapes are represented on a sphere, essentially a place or a region on Earth (and other planets as well, in case it comes to that!). Shapes can be stored in a document, thus associating a document with a place or an area on earth, or multiple of those.

To be able to search for a shape, it needs to be indexed. To represent a shape in an index, shapes need to be converted to an index-friendly representation. There are quite a few ways to do this; most commonly known approaches are prefix trees and bounding-box.

The QuadPrefixTree approach, for example, represents the earth with 4 grid squares at its first level of precision. The squares are labeled A, B, C and D. The next level of precision introduces another letter to the representation, so we get 16 grid squares - AA, AB, AC, AD, BA, ... and so on. By having this multiple layers of precision, we can create the most efficient representation of a shape which balances number of terms stored to the index vs precision. Another implementation called GeohashPrefixTree uses geohashes which have more grid squares per layer.

In this section we are going to learn how to perform geo-spatial queries. We will learn how to represent shapes in RavenDB documents, how to index them and then how to use shapes to query for documents having a relation with the query shape.

6.4.1 Point and Radius Searches

The most basic geo-spatial search operation is to have documents with a point on earth – represented by a pair of longitude and latitude values, and then send a query with a point and a radius. Such a query will return all documents that were marked to be in the given radius from the query point.

Let's say we have many events in our system, and we want to get a list of events that are close to us, in a 10-kilometer radius. An event can be represented using a class like this:

```
public class Event
{
    public string Title { get; set; }
    public double Latitude { get; set; }
    public double Longitude { get; set; }
}
```

An index with a spatial field will look like this when defined from code:

```
public class EventsIndex : AbstractIndexCreationTask<Event>
{
    public EventsIndex()
    {
        Map = docs => from doc in docs
select new
{doc.Title,
    _ = SpatialGenerate("Location", doc.Latitude, doc.Longitude)
};
```

```
} }
```

One can also create a spatial field from the studio in case his indexes are not maintained in code:

(image)

The spatial field that is being created using `SpatialGenerate` method or using the Studio is in fact going to create a Point shape whose coordinates are taken from the document, and then index it into a searchable format in the Lucene index.

Once an index with a spatial field was created on the event's location, querying is easy:

```
        var events = session.Query<Event>()
            .Customize(x => x.WithinRadiusOf("Location", 10, 32.456236,
54.234053))
            .ToList();
```

The `WithinRadiusOf` query issues a query with a circle shape, to find all documents with points (or other shapes) that are contained within this circle. This is a private case of the way geo-spatial queries work under the hood, which is what we will visit next.

6.4.2 WKT shapes

Since the point-and-radius search operation is so common, RavenDB hides some details to simplify usage. The underlying API used is still exposed, and next we will see how you can make use of it, too.

Internally, shapes are represented as strings using Well-Known Text (WKT) markup language. WKT is a simple, human readable format for representing a shape using text, for example: "`POLYGON ((30 10, 10 20, 20 40, 40 40, 30 10))`" or "`POINT (24.532341 54.352753)`". Every shape that can be represented using the WKT representation can be indexed or used for search as a query shape (some restrictions apply).

The point-and-radius scenario we showed above is equivalent to this bare implementation:

```
public class Event
{
    public string Title { get; set; }
    public string Location { get; set; } #A
}

public class EventsIndex : AbstractIndexCreationTask<Event>
{
    public EventsIndex()
    {
      Map = docs => from doc in docs
                  select new
                  {
                        doc.Title,
                        _ = SpatialGenerate(fieldName: "Location",
                        shapeWKT: doc.Location,
```

www.dbooks.org

```
                        strategy: SpatialSearchStrategy.GeohashPrefixTree,
                        maxTreeLevel: 12)
                };
        }
}
```

#A - WKT representation of a point on earth, ex. POINT (24.532341 54.352753)

And to query this we can issue the following:

```
var events = session.Query<Event>()
             .Customize(x => x.RelatesToShape(fieldName: "Location",
            shapeWKT: "Circle(32.456236 54.234053 d=10.000000)",
    SpatialRelation.Within))
.ToList();
```

So now we are passing a Point shape to the index explicitly, but note how the query is also a shape – a Circle with the origin at the query point with the given radius, which in our case is 10km. It is also important to note we could have indexed any shape (e.g. circle, a rectangle or a polygon) and the query would have worked; we could have also used any other shape to query.

What we haven't discussed yet is the last parameter to the query – the spatial relation. This is what we will cover next.

6.4.3 Spatial Relations: The 4 supported search types

There are 4 types of shape relationships that are supported by RavenDB:

1. **Intersects** - Querying for a shape which intersects a shape stored in a document within RavenDB will find those shapes which intersect with the given shape. Intersection occurs when the two shapes have at least one shared grid hash. Because of current limitations of the algorithm, very large indexed shapes are not deemed to intersect with very small query shapes. However, smaller indexed shapes will intersect with larger query shapes.
2. **Disjoint** - Finds those indexed shapes which are disjoint to the query shape. This means the indexed shapes and query shape must have no shared grid hashes.
3. **Within / Contains** - Finds those indexed shapes which are fully contained within the query shape. Unlike intersects, this means that all of the indexed shape must be present in the query shape. Any shapes that have additional area outside of the query shape are excluded.
4. **Nearby** – Does not perform any filtering, just boosts results by distance, so nearby locations will score higher in the results. This is useful when you want to use other search criteria to do the filtering, and use the shapes for relevance scoring only.

This method of indexing and querying opens up a ton of new opportunities for software developers. You can throw any shape, path or region into your RavenDB database, and use any type of shape for querying, using the available spatial relations.

There are quite a few places for customizations we haven't discussed, like the ability to specify the spatial strategy used for indexing and querying, or the precision to be used (the error percentage). The defaults used are quite decent, so unless you have a compelling reason not to use them you can safely ignore the various configuration options.

6.5 Summary

Queries in RavenDB can go way beyond simple value lookup, and in this chapter we went in length about more advanced query types that are supported by RavenDB.

Having Lucene as the underlying index mechanism opens up many great opportunities for full-text search type of queries. Finding a single word (or several) in a large body of text is so much easy to do with Lucene, and we have seen how RavenDB exposes that. By using Lucene's built-in analyzers, or by rolling your own, you can perform high-quality and high-performance full-text queries in any language and on any type of corpus.

The Lucene index can be of great assistance for other types of operations, like an efficient auto-complete feature for your website or providing suggestions for misspelled query terms. We have looked at several such examples.

Lucene is also great at indexing shapes, and we've learned about how shapes can be represented in text form, and how it can be indexed. Once a shape is indexed in a document, it can be queried using another shape and by specifying the requested spatial relation between the 2 shapes. We used this to demonstrate how a proximity search can be performed to find events by providing a location and a radius. Obviously, there are endless ways to use this, and the tools provided in this chapter should help you create your own neat geo-aware application.

Finally, we discussed the challenge with performing reporting queries efficiently. We talked in length about RavenDB's capabilities when it comes to this type of queries, and saw what can be done when the queries that need to be performed cannot be satisfied by standard RavenDB Simple-Map or Map/Reduce indexes.

www.dbooks.org

7

Scaling out

This chapter covers

- Adding replication for high availability and high read throughput
- Master-Master replication and write conflict resolution
- Sharding your data for better data distribution and high write throughput
- Multi-tenancy
- Mixing Sharding and Replication

Software system engineers are in the never ending pursuit for building stable software systems. The biggest challenge of every data-driven system is the requirement to be able to grow larger, and to be as fault tolerant as possible, without one taking a bite off the other.

Imagine Amazon, for example, limiting the amount of products in their catalogue because their servers have limited storage space. Or allowing for huge product catalogues but accepting some down time every now and then because one of their servers crashed. This isn't really acceptable, is it?

A fundamental player in this story is the database engine operating behind the scenes, and it usually breaks in one of two scenarios (or both combined!). The first case is when the data outgrows what a single database machine can hold. If your database server has 1TB of storage space available on disk, it obviously won't be able to store 2TBs of data. While adding more disk space is sometimes possible, other factors like available RAM also plays a part in querying large data sets. For any database system there is some absolute upper limit to which you can scale the server hosting it, and this is where it breaks.

The second case is where your one server database system can't hold the pressure anymore. This usually happens on high traffic websites, where database queries are very frequent and the database gets hit very often. This can also happen in scenarios where a high

write throughput is required. Just like before, there's some absolute upper limit to the amount of pressure one server can take.

Many businesses will feel the need to scale out in one point or another. You also don't always need to wait for things to break in order to make a decision to scale out. For example, you may decide you want to have more than one database server in your system to introduce redundancy for the quite-common cases of server failures or power-cuts.

Like we mentioned in chapter 1, this has been the biggest pain point of traditional database systems like SQL Server for quite some time. Those database systems are designed to be a single-machine database and therefore are very difficult to scale out. This is also in many cases why people go and look for alternatives.

Modern databases, often referred to as NoSQL, are usually better at this. Most of them are designed in a way that makes it possible drop new database servers into the system and for offering a better story around handling large amounts of data, to allow high-availability and redundancy in cases of failures.

While scaling up, read: adding more hardware to make an individual server better and stronger, has definite limits, scaling out (meaning: adding more servers to the system) has virtually no real limit. Scaling out is also many times preferable to scaling up as it allows to run on many commodity hardware servers as opposed to very few very costly servers.

Since RavenDB is a document database, scaling out in the context of RavenDB is the concept of adding more servers and split the data between them (also called *Sharding*), and optionally have multiple copies of a database on multiple servers for redundancy (also called *Replication*). In this chapter we will be discussing both in length.

It is important to note the two are completely orthogonal, and can be used independently if you only need one property and not the other, or combined together to provide a system that provides both high availability and high throughput.

The concepts behind scaling out are fairly simple, as you will come to see in this chapter. Mastering them is only a matter of experimenting and gaining experience. Therefore, it is recommended that you try each scenario yourself while reading this chapter. I've have made it easy to follow each new concept by giving practical guidance along the sections that you can follow.

7.1 Replication

Setting up replication on a database will tell RavenDB to clone this database to other RavenDB servers, and keeping all clones up-to-date as new changes come in. This is done by telling the database the addresses of those servers, and once replication is enabled RavenDB will perform the replication in the background.

Since in RavenDB we deal with documents, enabling replication for a database effectively means all documents from that database will be available from all RavenDB servers that have been configured to hold this database. Replication is an asynchronous process, meaning it runs in the background of the replicating server. Therefore, it may take new documents, or changes to existing documents, some time to propagate to all the servers holding the database.

www.dbooks.org

However, once they have propagated, every document in the database will be available from all those servers.

Replicating data across multiple machines can achieve 3 different goals, and is usually used to facilitate one or more of them:

1. **High availability** – having the same piece of data available on different machines and possibly datacenters enables access to the data also in cases of failures like power cuts, server failures, network issues etc.
2. **Increased performance** – once data is replicated and is available on more than one server, those extra servers can participate in queries and data reading operations to assist lower the pressure from the main servers.
3. **Live backups** – replicating data can also be used to perform live backups to remote locations, in cases of catastrophic failures like servers catching fire, data center flooding or any other kind of disaster.

Once replication has been setup for a database, that database is going to reside on multiple servers and therefore documents will be available via Load or Query operations even if (rather, when) a node fails. However, since only documents are being replicated if you rely on custom hand-written indexes you'd have to register them with every replica separately.

Another gotcha for using replicas to answer queries is the added latency. Back in chapter 3 we discussed the asynchronous indexing process and the fact the queries may return stale. When querying a secondary server (a replica) we may experience an additional potentially more severe staleness due to the latency of the replication process. Since data may have not been replicated yet, the index on the replica may not even be aware it is returning stale data. This is an important realization that you have to keep in mind when enabling fail-over of queries to secondaries.

A database wishing to have replicas has to be created with the Replication bundle enabled from the get go. Figure 7.x shows how to enable this when creating a new database. If you want to replicate a database that doesn't have this turned on, you will have to create a new database with the replication bundle enabled, and import all data to it. We will explain why it is so later in the chapter when we discuss conflict resolution.

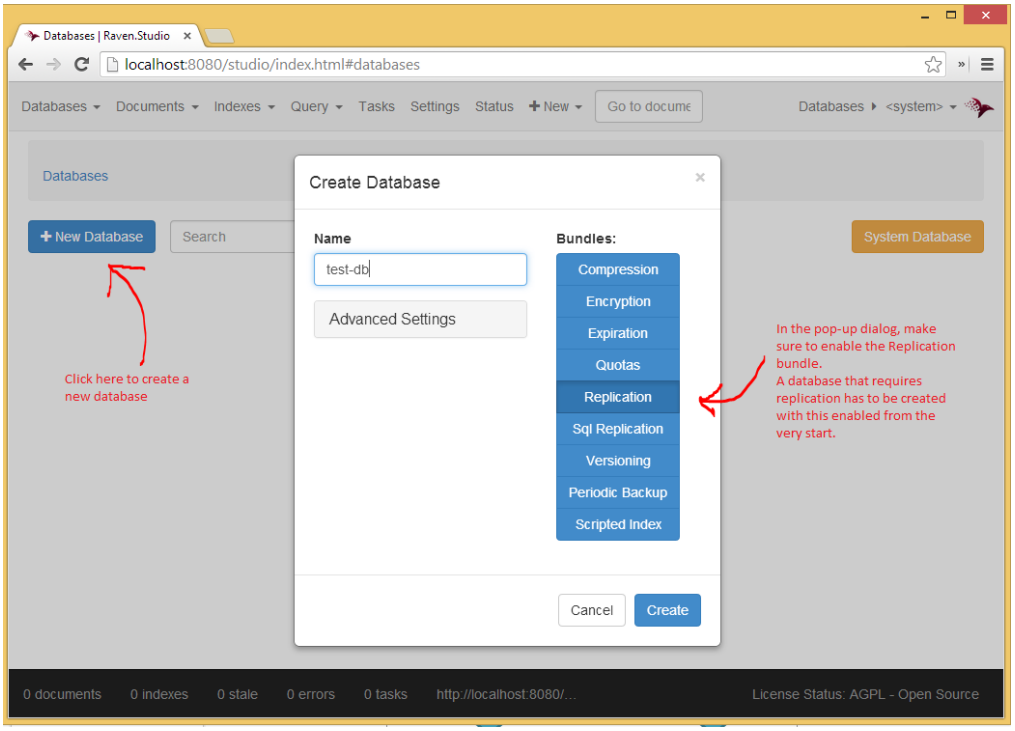


Figure 7.x: Enabling the replication bundle when creating a new database, so it supports data replication.

In this section we will look at the various configurations for database replication that are available, and on how to enable them. After replicating data around we will see how automatic failover helps with achieving high-availability, how read-striping helps with performance, and how to setup remote live backup machines.

7.1.1 Master-Slave replication

The most basic mode of replication is the Master-Slave setup. In this setup there is only one Master server defined, and that Master is the only one accepting writes and responsible for replicating the data written to it to all Slave servers.

A Slave server is a fully operational RavenDB server, holding a full copy of the database, except it only accepts read requests for the database they act as slaves for. This is depicted in figure 7.x.

www.dbooks.org

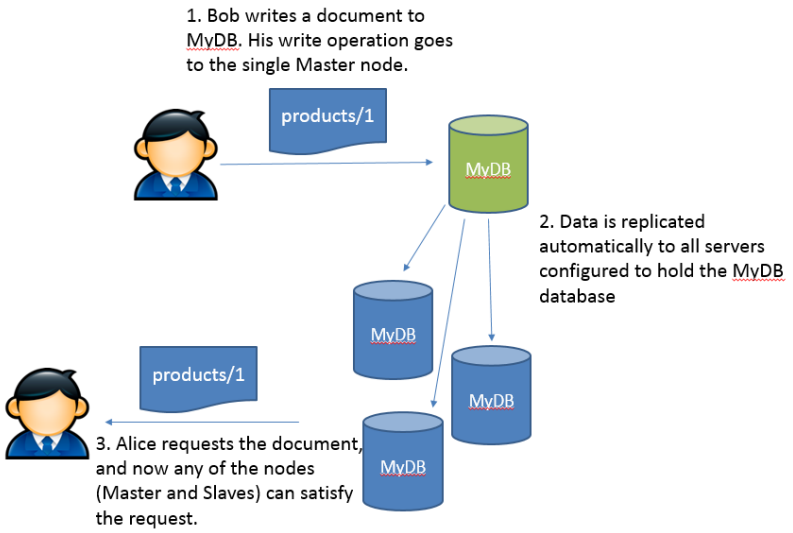


Figure 7.x: A Master-Slave replication setup

This mode allows providing high availability for reads, as documents from the database reside on multiple fully operational servers with the exception that there is only one pre-defined server that is able to accept write requests. If this Master server stops being available then the database becomes read-only until that node comes online again. We discuss the possibility of having several Master nodes later when we discuss the Master-Master replication setup.

SETTING UP MASTER-SLAVE REPLICATION

Enabling replication on a database is as easy as writing a document to that database defining the replication destinations. The replication settings document is always expected to have the ID of `Raven/Replication/Destinations`. In that document you list the URLs for the other servers, and once it is done RavenDB will do the rest.

For example, if we have 5 RavenDB servers running and we want `server1` to replicate to the rest of them, we will need to have the following replication settings document in our database on `server1` (assuming the same server naming convention, and the default port 8080 is used on all of them):

```
{
  "Destinations": [
    {
      "Url": "http://server2:8080/",
      "Database": "MyDB"
    },
    {
      "Url": "http://server3:8080/",
```

```
      "Database": "MyDB"
    },
    {
      "Url": "http://server4:8080/",
      "Database": "MyDB"
    },
    {
      "Url": "http://server5:8080/",
      "Database": "MyDB"
    }
  ]
}
```

This document can be added programmatically using the client API like so:

```
using (var session = documentStore.OpenSession())
{
     var replicationDocument = new ReplicationDocument #A
     {
            Destinations = new List<ReplicationDestination>
                           {
                                  new ReplicationDestination #B
                                  {
                                     Url = destinationUrl,
                                     Database = destinationDatabaseName
                                  }
                           };

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

#A – the ReplicationDocument and ReplicationDestination classes are convenient classes that are defined by the RavenDB client API
#B – Each ReplicationDestination is a Url and a Database name, each for every replication destination

The Management Studio provides a nice interface for managing the replication destinations, available under Settings -> Replication as shown in figure 7.x. In that screen you can add, change or remove replication destinations easily, and in many times you'll find it easier to work with. This is effectively a nice editor to the replication destinations document we were adding manually above.

www.dbooks.org

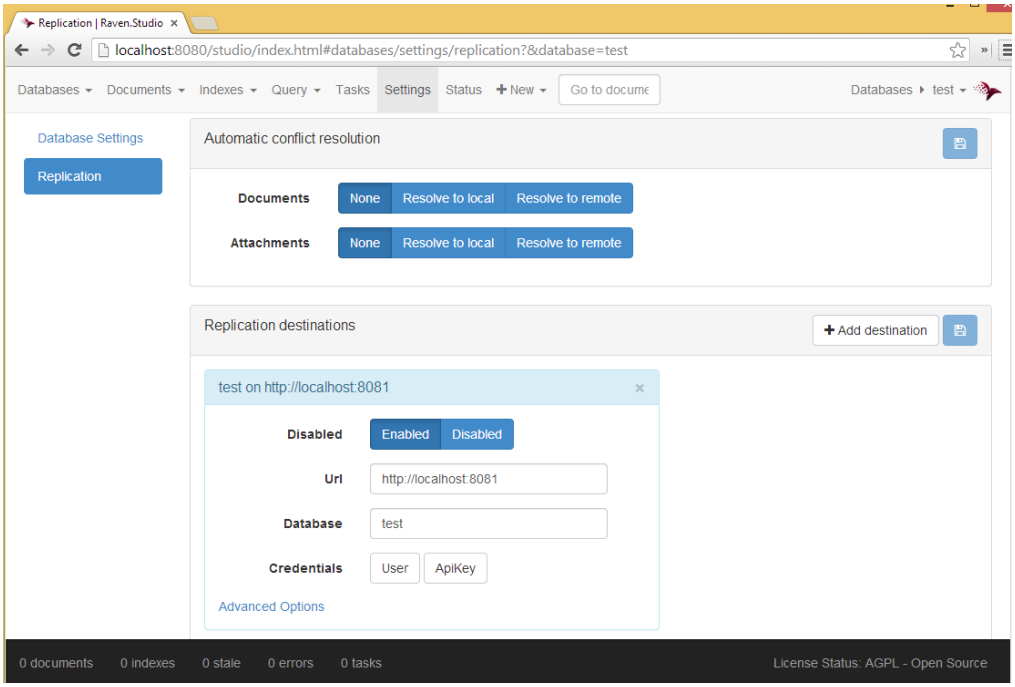


Figure 7.x: Managing Replication Destinations for a database using the Management Studio

Once replication destinations have been set by putting the `Raven/Replication/Destinations` document or by using the Management Studio, replication of the database contents will start to all database that are defined as replication destinations.

In the Master-Slave replication topology there is only one replicating server, and that server is then referred to as the Master server. The replication destinations that it is configured to replicate to are referred to as Slaves, as they only receive data from the Master and never participate in write operations by any of the clients.

For replication to work, all replication destinations have to have the database created in advance, and the replication bundle enabled on them as well as on the database on the replicating server. The steps for doing that are exactly the same as setting this up on the master server. The Replication Destinations document, however, only needs to be on the replicating server – in our case that is the Master server.

Try it yourself: Creating a Master-Slave setup

The easiest way to experiment with scaling out RavenDB databases is to launch multiple RavenDB server instances. Using the run-in-memory option you can have as many RavenDB databases as you wish running, each will grab a different port automatically and write nothing to disk.

To run multiple in-memory instances of RavenDB, get into a command shell (using PowerShell or by typing cmd in the Start Menu), and navigate to the RavenDB installation path. When there, type `Raven.Server.exe /ram -set=Raven/Port==8081` and hit enter to launch an in-memory database. Repeat this to create as many instances as you want, make sure to change the port (8081-9000 are usually available).

The next order of business it to create a replication-enabled database on all servers. By accessing the Management Studio you can create a test database. When creating the database, make sure to enable Replication by enabling the Replication bundle as shown above. Make sure to create this database on all the RavenDB server instances you launched.

Once the database has been created, add the replication settings document to one of the nodes, and in it list all the other RavenDB servers that you created. This node is now going to act as your Master node, replicating any new data to all other nodes, which are now Slave nodes for this database.

Create a new document on the Master node using the Management Studio, and watch the Management Studio of the other nodes as the document is being replicated to them and their Management Studio refreshes to show this document as well.

Adding a new replication destination to an operational system is as easy as bringing the server up and adding its URL to the Destinations document on the master. Once that document is changed, the Master server will start replicating to new server that have been added to that document, and stop replicating to servers that have been removed.

It is important to note replication will only clone documents to fully operational RavenDB servers, but no index definitions are going to be replicated. If you have created indexes manually on the master server, you will have to create them again on all the slave servers as well.

AUTOMATIC FAILOVER

One of the most common uses of replication in any database system is to enable high availability for data, meaning allowing access to data also in case of server or network failures. With RavenDB once a database is configured to be replicated, we get this behavior for free when using the Client API. It detects the replication destinations for a database automatically and remembers them, and then whenever a communication failure occurs it connects to the replicas instead and asks them for this data. We refer to this behavior as Automatic Failover.

> **FAIL-OVER IS PURELY CLIENT-SIDE BEHAVIOR** Automatic failover is done entirely by the client API. You will not see this behavior by using the REST API directly.

This feature highlights the fact that in a distributed environment you cannot assume availability and connectivity, as networks will fail and servers will crash. And indeed replication with RavenDB as a whole is designed to deal with failures. This is true for both the asynchronous replication process in replicating servers, and clients trying to connect to servers.

www.dbooks.org

Both are resilient to failures, and will try to fallback to other servers with occasional retries to detect servers that come back to life.

Once a failed node comes back online again, both the connected clients and the cluster of RavenDB servers will detect that and resume working with it: replicating servers will resume replicating data to it from the point in which replication previously stopped, and clients will continue performing reads and writes against it, depending of the configurations of databases on that server.

It is important to remember that in a Master-Slave setup, we can only read and never write if the Master is inaccessible.

Try this yourself: Crashing one server and still getting data

To experience Automatic Failover first-hand, have several RavenDB servers running and set up replication from one of the servers to the others, exactly like we did before.

Create an application that connects to the Master server, by creating a DocumentStore instance that points to it. Create a new document and see how it replicates to all the Slave nodes as well.

Now kill the Master slave. Since our code only knows about the Master slave, you would expect it to fail when trying to load the document again, right?

Wrong! Try loading the document by Id (or by querying for it) and see how RavenDB succeeds in loading it back. The delay that you will notice is RavenDB detecting the Master node has failed by waiting for some period before timing out.

If you want, you can follow the sample code in the Chapter07.Replication project in the code accompanying the book.

READ STRIPING

When replication is enabled the same data resides on multiple servers, and all Slaves are virtually equal in terms of the requests they are able to answer. Why not use them as well then, instead of having just one server take the heat of all read operations?

The Read Striping feature does exactly that. By enabling it you tell the RavenDB client API to perform a round-robin style querying, meaning every server will be approached once in a cycle, and each cycle will make sure to include all servers. This ensures even distribution of read operations between all available servers.

This feature can be enabled by setting `documentStore.Conventions.FailoverBehavior = ReadFromAllServers`. Once this is done RavenDB will read data from secondaries (meaning, replication destinations) of the database as normal operation. If one node fails, it will be skipped just like in normal automatic failover. This only applies for reads; writes will still go to the Master server only.

One important thing to note when enabling this feature is the consistency model used. Since secondaries may take a while to sync with latest changes made to the Master – both

because replication is asynchronous and because their indexes have to catch up independently of the indexes of the Master – reading data

This also means *optimistic concurrency* may not function properly, since it expects to check its version with the server it is going to write to (see chapter 5). To correctly use Optimistic Concurrency when Read Striping is enabled open the session with the flag `ForceReadFromMaster` set to true. This will ensure this session only uses the Master node, and therefore will not fail when using Optimistic Concurrency.

Hot Backup

Another possible use of replication is to continuously clone the live database to an off-site location, to protect the data from even the rarest disasters. This copy of the database often times isn't going to be accessed by clients at all, and only function as a backup strategy that does not involve nightly tasks.

The Master-Slave replication setup is perfect for this use case. It guarantees the backup server will always be up to date, and does not require heavy processing of backups at midnight or anything similar to that.

Recall indexes are not being replicated, only documents. This means you can avoid registering any indexes on the backup server, and as long as you don't enable read striping it will never be accessed unless there was a serious failure.

To summarize what we have seen up to this point: Master-Slave replication is perfect for having multiple copies of your data to protect from cases of failure, or to distribute the load of read operations. The idea behind it is to always have at least one server up and available to answer requests for data.

But the Master-Slave replication topology is focused around reads, and only has one Master server that can be written to. What this means is that if the Master server goes down, the system becomes read-only.

Some businesses will find this problematic. For example if you use RavenDB as the storage for a system taking orders from customers, if a server goes down you will still be able to view orders and handle them in the backend or allow access to the user to view his recent orders, because the data is still available from other servers. But if the Master server is the one that became unavailable, you will not be able to accept new orders, or to mark existing orders as "shipped" from your store backend.

If this is not an acceptable situation in your scenario, you can solve this by having more than one Master server in your system, so if one disappears the others can be used instead. This is called Master-Master replication, and is what we'll be discussing next.

Replicas Write Assurance

By default a RavenDB server will not wait for replication to be executed when writing to it, and will confirm the write immediately once the write was confirmed by the operating system. The data will then get replicated to the replication destinations asynchronously.

Since the acknowledgment received by the client writing the data only acknowledges the write was successfully done on the Master, the client has no way of knowing if the replicas now

www.dbooks.org

have this change or not. Since potentially the replication queue can be long, or the Master server could have crashed immediately after, replication is not guaranteed to happen immediately after a successful write.

In cases where it is important to know the data was indeed replicated, RavenDB provides a mechanism for write assurance. You'd have the following lines of code after saving data to RavenDB, and this will wait until it's confirmed the data was replicated successfully to the number of replicas defined, or until the timeout has passed:

```
var userEtag = session.Advanced.GetEtagFor(user);

var replicas = await ((DocumentStore)store).
    Replication.WaitAsync(etag: userEtag, timeout: TimeSpan.FromMinutes(1),
replicas: 1);
```

7.1.2 Master-Master replication

Master-Master replication enables having more than one Master server in the system. This provides high-availability also for writing new data. But this is not without a cost – you will have to account for possible conflicts of data, as changes to one document can be now made via two different servers simultaneously by different people.

A system set up with Master-Master replication has the exact same behavior and benefits as a system with Master-Slave replication setup, this is mostly because it has the exact same behavior: a server that has been written to will replicate the data written to it to all replication destinations that were defined to it.

There are two fundamental differences though – each server can now be approached for writing as well, and your system needs to account for possible data conflicts. Figure 7.x depicts a system with a RavenDB database with a Master-Master replication setup.

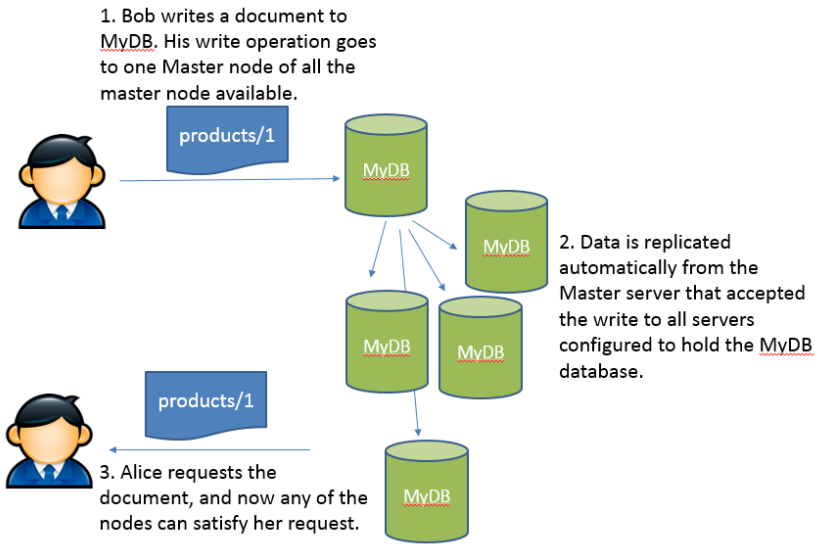


Figure 7.x: A database in a Master-Master replication setup. All server nodes are configured as Master and can be written to, and all nodes operate as secondaries for reads as well.

Let's first see how to create a Master-Master type system, and then look at how conflicts may happen, and how to resolve them.

Setting up Master-Master setup

Setting up Master-Master replication is almost identical to setting up Master-Slave. There are only two subtle differences.

First, the replication destinations document needs to be present on all servers. Unlike in the Master-Slave setup where we put it on only one server (the one we denoted as Master), here you need to create it on all servers to allow changes made on any of the servers to propagate to the others as well.

So assuming a 3-node setup, `server1` will have this document as `Raven/Replication/Destinations`:

```
{
  "Destinations": [
    {
      "Url": "http://server2:8080/",
      "Database": "MyDB"
    },
    {
      "Url": "http://server3:8080/",
      "Database": "MyDB"
    }
  ]
}
```

www.dbooks.org

And server2 this document:

```
{
  "Destinations": [
    {
      "Url": "http://server1:8080/",
      "Database": "MyDB"
    },
    {
      "Url": "http://server3:8080/",
      "Database": "MyDB"
    }
  ]
}
```

And you can guess what the 3rd server would have as its replication destinations. Every server will be configured to replicate to its fellow servers. Once this is done, every document written to the replicated database using any RavenDB server will be replicated to the rest of the servers holding this database. Go ahead and try this yourself now!

To enable writing to those Master servers using the Client API, you need to set the FailoverBehavior of the connection to failover writes as well:

```
documentStore.Conventions.FailoverBehavior =
FailoverBehavior.AllowReadsFromSecondariesAndWritesToSecondaries;
```

This setting will ensure that whenever the server we connected to isn't available, writes will fail-over to secondaries as well as reads.

Configuring the Master-Master setup is a 2 step process – first step is to enable replication between all servers so data written to any of them will be replicated to the rest of them; and the second step is to tell the Client API it can also write to replicas in case it can't write to the Master node.

Try this yourself: Setting up Master-Master replication

Let's now experience first-hand Master-Master replication setup. The process of setting this up is almost identical to setting up Master-Slave replication like we did a few pages ago – have multiple servers running locally on different ports, and write a simple application that writes some data to the document store.

This time we are going to setup replication destinations documents on all of the servers we have running. That will make all our servers replicate all data to each other, and then by specifying `AllowReadsFromSecondariesAndWritesToSecondaries` in your application code you effectively allow writes to all servers that have been configured as a replication destination to the Master server.

Once you run your application, crashing the Master server (that is, the server you originally connected to with your `DocumentStore` instance) will allow failing over reads, just like we've seen before.

But unlike what we did before, this time writes will also fail over to secondaries, making your application highly-available also for writes.

Data Conflicts and Conflict Resolution

RavenDB does its best to write to only one master, so clients will usually talk with only one Master always. However, when that one Master fails to respond and a client decides it went offline, the client will failover to one of the other available master nodes. This may lead to different clients writing to different servers, and if 2 clients were editing the same document – a data conflict will occur.

Figure 7.x shows how a conflict can occur during normal operation when there is more than one master server.

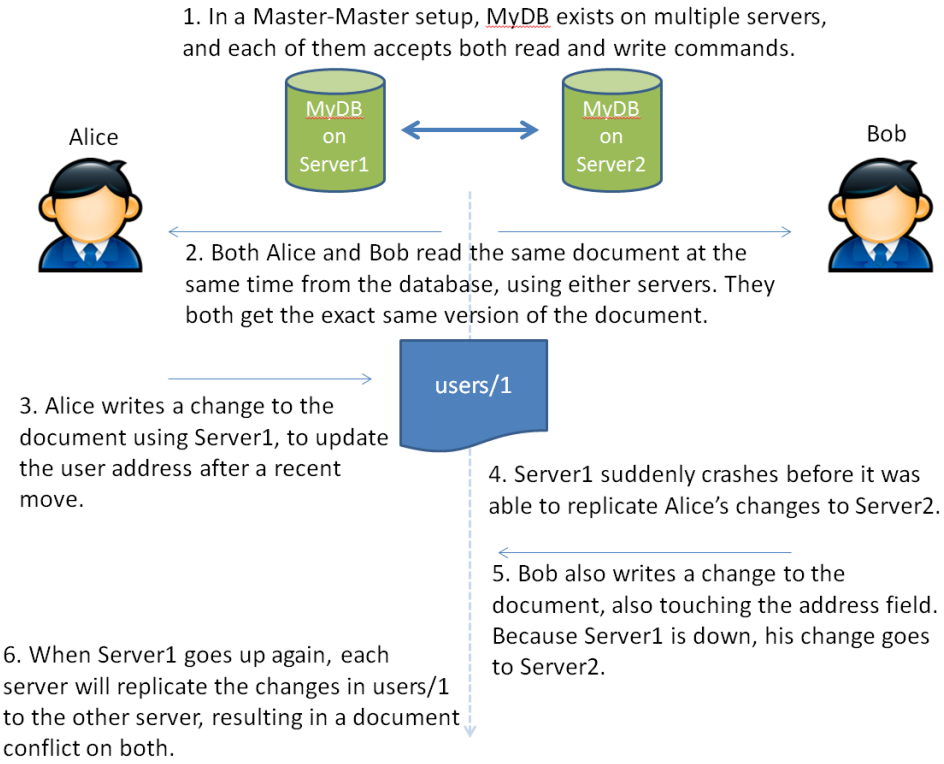


Figure 7.x: A document conflict can is likely to happen in a Master-Master replication setup if two conflicting writes are made to two different server concurrently.

When there is a document conflict in RavenDB, each conflicted document instead of containing its data will be marked as a conflict document (using some document metadata) and

www.dbooks.org

contain references to temporary documents of the different conflicting versions. Assuming we have a conflict on a document with ID of `users/1`, the document will now look like this:

```
{
    "Conflicts": [
        "users/1/conflicts/4429f05f-ddee-23d4-506b-dc6d39339bbf",
        "users/1/conflicts/34fc4bd3-d690-b63a-769e-99eac1cbbd11"
    ]
}
```

The `Conflicts` array is in-fact a list of document references containing the actual conflicting data, which replaces the actual data in a conflict document. The data is stored in the referenced documents, representing different versions coming from different sources:

```
{
    "Name": "Itamar"
} #A

{
    "Name": "Itamar Syn-Hershko"
} #B
```

#A – users/1/conflicts/34fc4bd3-d690-b63a-769e-99eac1cbbd11
#B – users/1/conflicts/4429f05f-ddee-23d4-506b-dc6d39339bbf

When trying to load the document using the client API (via direct load or querying), a `ConflictException` will be thrown, bringing the conflict to your attention and requiring you to resolve the conflict. Until the conflict has been resolved the document will be unreadable in its original form, and trying to load it using the client API will result in an exception, that is in order to require you to resolve the existing conflict.

Resolving a document conflict can be done manually, by overwriting the document itself. Since the conflicting versions are stored in the database itself, they are available for you to use for either mending or resolving directly to one of those original versions. Each project, each entity and each environment will have their own business rules for conflict resolution, and it is highly recommended to accurately understand when conflicts are expected, and how they should be resolved. Last-write-always-wins is often times not the right answer, even though it always appear to make sense at start.

```
using (var session = documentStore.OpenSession())
{
    try
    {
        var user = session.Load<User>("users/1"); #A
    }
    catch (ConflictException e) #B
    {
        var sources = new List<JsonDocument>();
        for (int i = 0; i < e.ConflictedVersionIds.Length; i++) #C
        {
            var doc =
documentStore.DatabaseCommands.Get(e.ConflictedVersionIds[i]);
            sources.Add(doc);
```

```
        }

        var resolved = sources[select]; #D

        documentStore.DatabaseCommands.Put("users/1", null,
resolved.DataAsJson, resolved.Metadata); #E
    }
}
```

#A – Loading a document, to do further processing or just to display it
#B – If `ConflictException` was thrown, we should catch it and attend to resolving the conflict
#C – We load the documents holding the sources so we can resolve it by either using one version as a whole (like shown here) or use various merging algorithms on text or via deserialization.
#D – Once we decided on a version to go with, we need to push it back to the database, overriding the current document.

RavenDB provides facilities to automatically resolve conflicts using general approaches that consider only the conflicting sources. It is possible to tell the RavenDB server to always prefer the Local version, or the Remote version, when a conflict occurs, and it easiest to do this via the Management Studio, as figure 7.x shows.

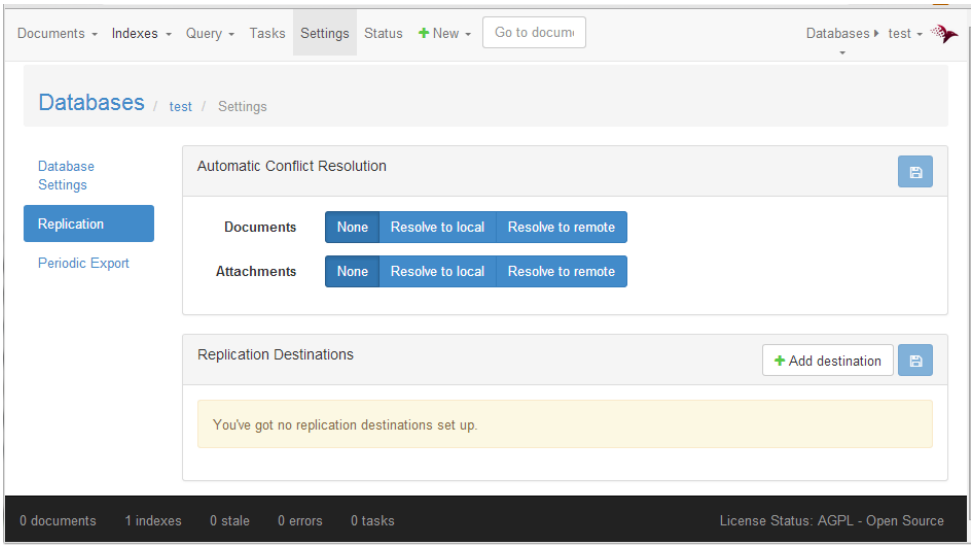


Figure 7.x: Automatic conflict resolution settings

However, as we just stated – automatic conflict resolution is often times not the right answer at all even though it does indeed sound appealing. As a general rule, you should first implement conflict resolution in your code, and only if this turns out to always prefer an entire one version over another, start looking at moving the logic to be done using the automatic resolution on the server side.

www.dbooks.org

Try this yourself: Resolving data conflicts

Using our Master-Master setup from before, you can easily create data conflicts and practice handling them.

To create a data conflict, create one document with the same ID – say “users/1” – on multiple servers before setting up the replication destinations documents. Make sure the document contents conflict – for example by having different values in the Name property.

Now setup Master-Master replication like you did before, and you will immediately notice the document is now marked as conflicted document.

If you enabled automatic conflict resolution, you will not see the conflict document and instead see the document after the conflict resolution. If you do see the conflict document, try loading it using a simple program by calling session.Load<dynamic>(“users/1”); and you will immediately have ConflictException thrown.

By catching this exception you can make the changes required to the document and commit the conflict resolution back to the database, so future read operations on this document will not throw.

7.2 Sharding

Sharding is the process of splitting one database into multiple parts, which in turn will be known as “shards”. There are multiple reasons for doing so, most of them originate from performance concerns.

One such scenario is when the database simply has too much data to be contained on one server. When there is no single server that can contain all data in the system, there is practically no other choice than to split it into multiple shards.

Sharding is also applicable in other scenarios where one database needs to be broken apart, like when geo-distribution of data is required. This can be for performance reasons – so North America clients have their data in data-centers that are close to them for faster responses, and European clients have their data close to them. Legal reasons can also play part here, like the European data protection act which prohibits storing data on European citizens on servers outside of the EU.

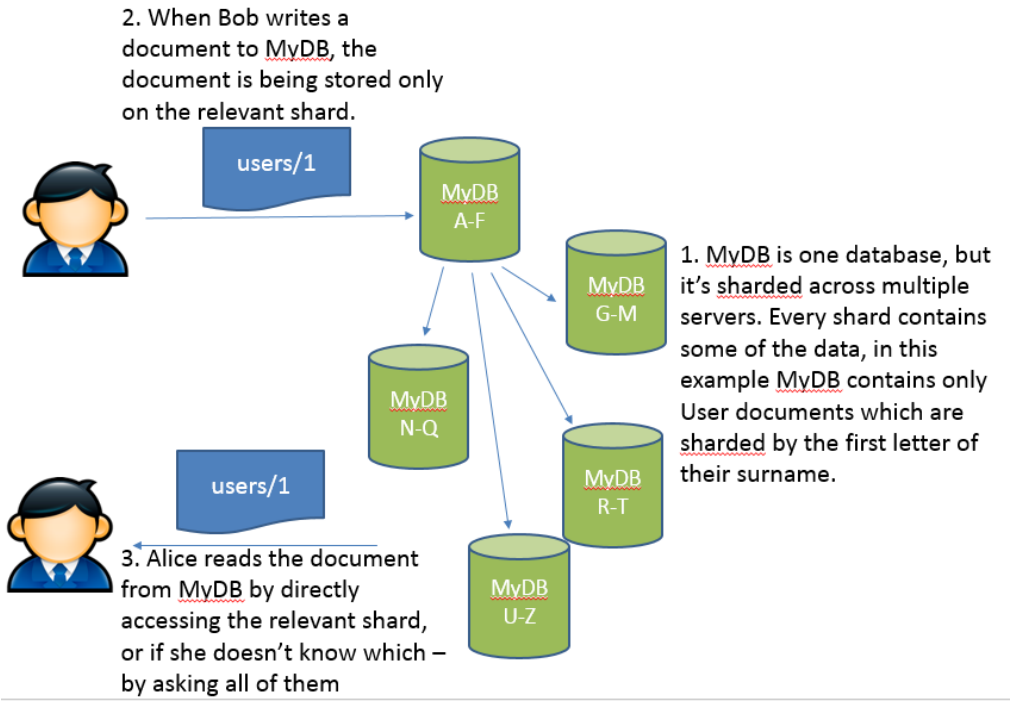


Figure 7.x: An example of a database containing User documents and sharded across multiple servers based on the user surname.

While in a replicated database every server contains the entire dataset, one sharded database in effect spans across multiple servers where every server holds only a piece of the entire database, without being even aware of the other shards. The challenge in a sharded environment is in spreading the data evenly across them, and then knowing which machines has which part of the data.

In charge of coordinating all of that is the Sharding Strategy defined on the client side. The Sharding Strategy dictates which database server each piece of data belongs to. This is important when storing new data, and also when retrieving documents via queries or load operations.

Sharding Strategies usually leverage deep knowledge of the shape of the data in your system to facilitate effective sharding and optimize shard access. This is why as a general rule you should provide your own Sharding Strategy, which is a simple implementation of an interface. If you don't provide one then a default, a best-effort implementation supplied by RavenDB will kick in. This mode is called Blind Sharding, as it will essentially facilitate data sharding without knowing anything about your data.

www.dbooks.org

In this section we will look at sharding with RavenDB and how to use it correctly. In order to get our feet wet we will start with Blind Sharding; we will quickly implement sharding without defining the sharding behavior ourselves, and see how RavenDB behaves.

After experiencing first-hand what sharding is and how it works, we will look at sharding that operates off your own supplied sharding strategy. We will call this data-driven sharding, as the sharding technique used can now make use of the data model of your system. We will finalize the discussion on sharding by showing how the Sharding Strategy can be changed in live systems to accommodate changes in original estimations, or simply as the system grows larger.

7.2.1 Blind Sharding

The Sharding Strategy is a way to tell RavenDB about your data model, and to allow it to optimize for it in various ways while sharding. When providing an empty Sharding Strategy RavenDB essentially will know nothing about the data model used, and therefore will have to default to working on best-effort basis for sharding. This is then called Blind Sharding.

Let's work on setting up sharding and see what effects it has on our system. Next up, in the next sub-section we can look at customizing the sharding strategy.

Since sharding means talking to multiple servers but considering them as hosting one database, connecting to a sharded database is essentially connecting to all the shards using the ShardedDocumentStore:

```
var shards = new Dictionary<string, IDocumentStore>
             {
          {"Asia", new DocumentStore {Url = "http://localhost:8080"}},
          {"Middle East", new DocumentStore {Url = "http://localhost:8081"}},
          {"America", new DocumentStore {Url = "http://localhost:8082"}},
             }; #A

var shardStrategy = new ShardingStrategy(shards); #B

using (var documentStore = new
ShardedDocumentStore(shardStrategy).Initialize()) { #C
}
```

#A – Creating the list of document stores, each of them is a named shard of our database.
#B – Creating a blank Sharding Strategy. This effectively implies Blind Sharding.
#C – A Sharded document store, once created, is just a normal DocumentStore we can create sessions with and so on. All the logic is handled inside.

Once that is done, let's write some documents to RavenDB. Running the following lines of code multiple times will generate several documents in RavenDB:

```
using (var session = documentStore.OpenSession())
{
     session.Store(new { Foo = "Bar" } );
     session.SaveChanges()
}
```

Notice how even though we are using a different type of `DocumentStore` (namely, the `ShardedDocumentStore`) the session usage is exactly the same. Once written, every document is going to reside in a different shard and you will notice RavenDB writes them in a round-robin fashion – meaning it will make the first write to the first shard, second to the second shard and so on, starting the cycle all over again when it has run out of shards.

IDs of Sharded Documents

It is perhaps more important to look at the documents stored on a shard. Once the above code is executed, accessing one of the shards using the Management Studio will reveal this (you can quickly write this yourself, or run the Chapter7.Sharding sample application from the sample code accompanying the book):

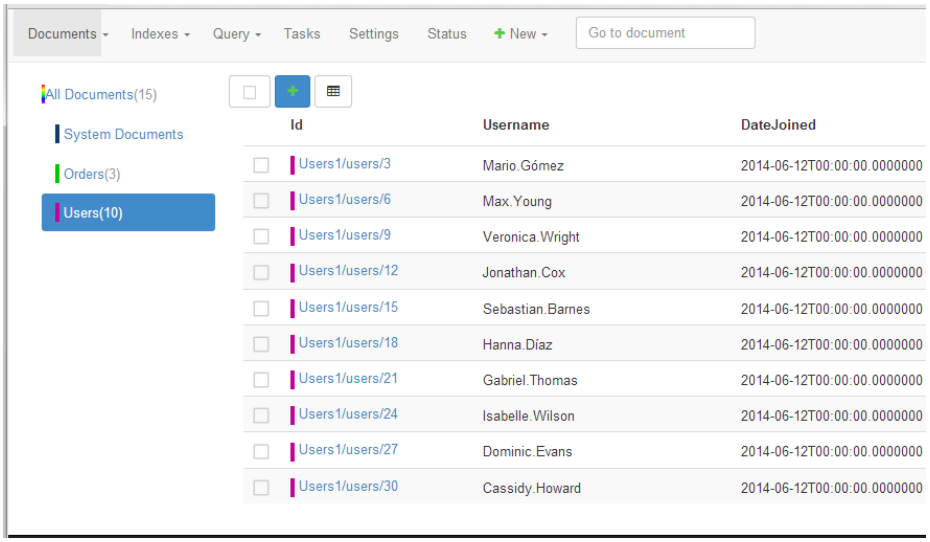


Figure 7.x: A look at documents added to a shard named Users1 of a database

Notice how the document IDs bear the shard ID (in this case Users1) with them? This is an important part of the way RavenDB sharding works. Those prefixes bearing the shard name as it was defined when configuring `ShardedDocumentStore` help access only the relevant shard when loading the document by ID. Since in that scenario the Sharding Strategy does not have enough knowledge of the document, the ID bearing the shard ID is extremely helpful in optimizing this fairly common case.

Blind Sharding and Consequences of Sharding

A Sharded RavenDB database is effectively assembled of numerous completely autonomous database instances running on different servers. No Shard knows about the other, and the

www.dbooks.org

sharding mechanism is done and managed on it's entirely by the client code through the Sharding Strategy.

Because the shards live in complete isolation from one another, any references made between documents in the system will be meaningless unless they reside on the same shard. For example, an Order document referencing a User document to indicate he is the one made the Order; If you want to be able to use Includes or load the User document during indexing you'll have to make sure all Order documents are on the same shard as their perspective User.

This is an important realization when working with a sharded database. You have to plan your sharding strategy properly in order to avoid cross-shard operation on core actions of your applications, and sometimes it can even affect the actual data modeling process – at least if sharding was a requirement at that stage of development.

Blind sharding, while works out of the box and is simple to get up and running with, lacks the capability of doing that. Since the Sharding Strategy is the default one and doesn't have any real knowledge of the data model, Blind Sharding is not really suitable for real-life deployments.

Another problem with Blind Sharding is many optimizations are not available when using it – again because we have no knowledge of the data model. This prevents RavenDB from optimizing shard access on queries, for example.

Blind sharding is a very easy way to get off the ground with, but once your data model has stabilized it's highly recommended that you provide a Sharding Strategy that's tailored to your model and your requirements. This then becomes data-driven sharding which we discuss next.

The difference between Blind Sharding and Data-Driven Sharding can be seen in action by running the Chapter7.Sharding sample in the source code accompanying the book.

Try this yourself: Setting up sharding

Try taking a data model you are familiar with, and setup a sharded environment where you can store it. Then write a small application that creates a ShardedDocumentStore and uses it to connect to the sharded database, and store and retrieve data from it.

Creating a sharded environment is actually pretty simple, and it's basically identical to what we did with replication when we've ran multiple RavenDB servers in memory.

Once you've done that, use the code shown in this section to connect to the servers and then perform data storage and queries using it.

You can use the Chapter7.Sharding sample accompanying this book as a starting point and a reference.

7.2.2 Data-driven sharding

Enabling data-driven sharding is done by providing a meaningful Sharding Strategy, one that gives RavenDB enough information about the way your data is structured. This will allow RavenDB to look at the data model when it tries to decide where to send a document to or how

to query for them. Creating a sharding strategy using your model can be done in a strongly typed way like so:

```
var shardStrategy = new ShardStrategy(shards)
     .ShardingOn<Company>(x => x.Region)
     .ShardingOn<Invoice>(x => x.CompanyId);
```

In the above example, Company documents will be sent to a shard bearing a name that matches the Region property of the Company instance. Once that is done, Invoice documents will then be directed to shards based on their CompanyId. This ensures locality of reference, which then enables proper use of various features like Includes, Result Transformers, calls to LoadDocument in indexes and so on – all of which we looked at earlier in the book.

The exact way of making the sharding engine aware of your data structure will vary based on the actual structure of your model, but the way to set it up is always going to be the same: create the ShardStrategy object, pass it the list of shards, and then for each entity in your model specify what it should be sharded on.

Let's look at the User and Order model we mentioned earlier. This model involves two simple classes:

```
    public class User
    {
        public string Id { get; set; }
        public string Username { get; set; }
        public DateTime DateJoined { get; set; }
}

    public class Order
    {
        public string Id { get; set; }
        public string UserId { get; set; }
        public double TotalPrice { get; set; }
}
```

And let's assume we want to shard this users & orders database across 3 servers, so we create our ShardedDocumentStore as follows:

```
var shards = new Dictionary<string, IDocumentStore>
{
    {"Users1", new DocumentStore{Url = "http://localhost:8080"}},
    {"Users2", new DocumentStore{Url = "http://localhost:8081"}},
    {"Users3", new DocumentStore{Url = "http://localhost:8082"}},
};
using (var docStore = new ShardedDocumentStore(new ShardStrategy(shards)))
{
    docStore.Initialize();
    PushDataToDocumentStore(docStore);
}
```

Once this code runs, when the `PushDataToDocumentStore` method will be called User and Order documents will be generated and stored in RavenDB. However, Blind Sharding is in effect

www.dbooks.org

since a naïve sharding strategy is used. Running the sample application we get a screen similar to this – red lines indicating Order documents that are saved to a server different than the one in which the User document they belong to was saved:

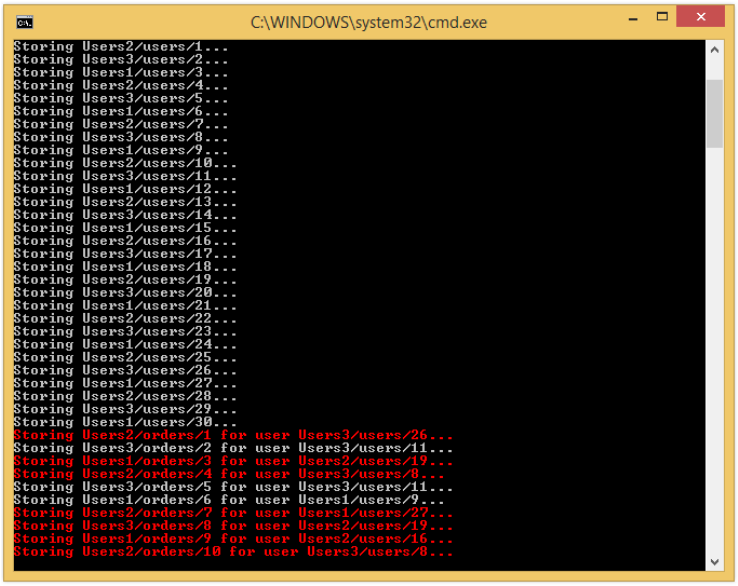


Figure 7.x: Blind sharding, not knowing anything about the data model used, will essentially store data in a way that doesn't allow for optimizing shard access (red lines show referenced Order and User documents not being stored on the same server).

When an Order document does not reside on the same server as its User document, fetching an Order document doesn't allow to include the User document by ID for example, or to index data from the user document into an index that operates on the Orders collection.

To resolve that, we need to tell RavenDB more about our model – on how we want to shard the User documents (default is round-robin, and does not rely on the actual data), and then that Order documents are referencing User documents.

We do this by explicitly specifying our sharding strategy like below. In this particular example, we shard the User documents based on the first letter of their username using a translator function to map between the data property to a shard name:

```
var shardingStrategy = new ShardStrategy(shards)
      .ShardingOn<User>(x => x.Username,
         username =>
         {
           username = username.ToLowerInvariant();
           if (username[0] >= 'a' && username[0] < 'm')
              return "Users1";
           if (username[0] >= 'm' && username[0] < 'q')
```

```
                return "Users2";
            return "Users3";
    })
.ShardingOn<Order>(x => x.UserId);
```

By passing this `shardingStrategy` object to the `ShardedDocumentStore` we initiate sharding of our data in a way that is aware of our data model.

Once a non-trivial sharding strategy has been set to handle one of the entities in the model, it has to be told how to handle all the entities in the model (at least those that are related to the entity that is used). An exception will be thrown if some parts of the data model weren't covered by the sharding strategy.

Expert: Getting more out of the sharding strategy

When defining a sharding strategy on your model, you are telling RavenDB to create 3 different strategies behind the scene. There are 3 interfaces that are implemented internally, and instances of those objects dictate the way RavenDB decides on which shards to communicate with in various scenarios.

`IShardAccesStrategy` – determine how we call to the servers.

`IShardSelectionStrategy` – determine how we select which server a new instance will go to, and what server an existing instance belongs on.

`IShardResolutionStrategy` – determine which servers we should query when we are querying for data (allow to optimize which servers we are actually hitting for particular queries).

RavenDB hides this implementation detail using the syntax shown in this section. In some cases you might want to implement your own custom logic, and those are the integration points you will be looking for.

Often times, after selecting a sharding strategy and running with it for a while, reality doesn't meet your estimations and it turns out some servers take more load than others. Another scenario is when you want to add new servers and want the sharding function to take those into account, or maybe to fill them up quickly to help spread the load from the entire cluster of servers.

Thanks to the fact that sharding is dictated by a sharding function that exists on the client (or clients), it is fairly easy to modify it to bias towards certain servers to help balance the load. This leaves you with 2 technical challenges – (1) is to actually figure out the new (biased) sharding strategy, and (2) to make sure all clients are aware of this change.

While the second challenge is something you need to figure out how to do with your infrastructure, it is fairly easy to write a biased sharding function. A biased sharding function is just a new translator function that maps to the new server in a biased way, or ignores certain servers for new writes. By looking at date properties of the data model entity for example, it is possible to direct more or less data for each shard in your system, according to your requirements.

www.dbooks.org

Using a data-model-aware sharding strategy helps optimizing shard access on reads, and this is why it is very important to get right. Only shards that are known to contain relevant data for a query will participate, even to Map/Reduce indexes. The Sharding Strategy is the way we entrust RavenDB with this info.

Well-defined sharding strategy also helps facilitating proper transactions on writes. Since referenced documents will end up on the same shard, there is no need for distributed transactions since by definition only one node will participate in a transaction. If data is being written to two different servers, then that is done across transactional boundaries of the model as well.

7.3 Summary

In this chapter we acquired the tools to manage a modern database, one that needs to have the ability to grow very large and withstand network and hardware failures, or just be performant no matter the pressure thrown at it.

We looked at Replication, and learned how it works and how to use it in order to provide failover mechanism for cases of server failure. We also saw how it can improve overall database performance by spreading the load on to multiple servers, and how to leverage it to get a live backup of your database.

The most common replication setup is the Master-Slave setup, and we saw how to set it up and work with it. We then looked at Master-Master setup, which in contrast to the Master-Slave setup also allows failovers for write operations, although with the additional cost of having to take care of data conflicts.

We then moved to look at sharding, and what it means to shard your data. Sharding is usually required for performance reasons, and consists of breaking data of one database into several autonomous servers, each containing only a piece of the data. While sharding with RavenDB is quite easy to setup, we have seen how it can be made much more effective by leveraging data-driven sharding, as opposed to blind sharding.

It is important to note sharding and replication are completely orthogonal features. You can have sharding enabled, and you can have replication enabled, and each will be completely unaware of the other. If you have both enabled in the same time, they will play nicely together and features of each of them can be controlled separately, exactly like we discussed in the previous sections.

While sharding and replication do not conflict, Sharding and Replication are each a all-or-nothing offering on the database level. If you have replication enabled on a database, the entire dataset is going to be replicated and to the exact same replication destinations. There is no way of performing partial replication, or control which Collections get replicated where. The same thing is with sharding – if it's enabled, it becomes effective to the entire database. There is no way of performing partial sharding.

Now you are ready to scale-out your database, to make it more performant and your applications more stable. In the next chapters we will look at more advanced topics, like

extending RavenDB and other advanced features it offers, and we will finish with discussion on RavenDB in staging and production.

www.dbooks.org

8

Extending RavenDB

This chapter covers

- How to extend RavenDB
- Server side and client side extension points
- Some common RavenDB bundles
- Writing your own extension

By now you should be comfortable enough with RavenDB to model and build pretty much every type of application. However, it is only when you start using a software in real-world scenarios when you learn first-hand its weaknesses and discover the places where it lacks.

As a general rule, no software should be considered a silver bullet. All products fit very well for several intended scenarios but are completely the wrong decisions for other albeit similar usages. RavenDB is no different; some stuff it is very good with, but in other cases you will either find it is lacking or just not up for the challenge.

For those cases in which RavenDB can actually fit very well but some extra stitching work is needed, you may find extending RavenDB a good solution. In fact, there may already be an extension lying around somewhere that you could find useful.

In this chapter we will look at the extension points RavenDB offers, and showcase some extensions (also called *bundles*) that you might find useful. We start with *server-side bundles*, these are the extensions that integrate with the RavenDB server itself. Then we will look at client-side integration points. Finally, we will talk about how you could write your own extension, or a bundle, to introduce additional functionality.

8.1 Server-side bundles

Server-side bundles are extensions to RavenDB which integrate with the server itself, as opposed to client-side extensions which extend the Client API (which we will visit later in the

chapter). Integrating with the server allow for interacting with the stored documents, as well as with the indexing capabilities.

Server side bundles can define background processes to be run on startup or periodically, respond to database events like documents being written or deleted, and expose new functionality by defining new REST endpoints to the server. Add that to the fact they are first citizens within the database itself, with access to the document store and indexing engine, and you can see how much options this can open up.

We start this section by looking at some bundles that are already provided with RavenDB. While they are tightly integrated with RavenDB and ship as part of the core product, they are still provided in the form of a bundle. This is true both historically, and in the way they are written internally and being picked up by RavenDB.

Next we will look at how to install and use external bundles, including a short showcase of popular ones. Later in the chapter we will describe how to actually write one of your own.

8.1.1 Internalized (provided) bundles

RavenDB comes bundled with several bundles that were used commonly enough for them to be integrated within the product itself. In fact, some of RavenDB's core features such as Replication we covered in the previous chapter, or SQL Replication we discussed in Chapter 5, and the *MoreLikeThis* feature we saw in chapter 6 – are all implemented as bundles. They are so useful and commonly used that they were just internalized within RavenDB so no installation is required for them.

Learning about internalized bundles as such is useful because it serves as a good intro to bundles in general. It will also give you a good idea of what's possible through a RavenDB extension.

Let's take a close look at two such bundles we haven't seen before. You might find them useful on their own, but even if you don't looking at them is definitely a good way to start learning about server-side bundles.

The Expiration Bundle

The expiration bundle takes care of automatically deleting expired documents. Documents are considered expired when the time defined for them in the `Raven-Expiration-Date` metadata entry has passed, and this value needs to be set explicitly by a client. This bundle runs a background process on the server to clean-up those documents without user intervention.

This is very useful when you have entities in your model that just are not relevant after a certain period of time. A good example for common usage of this bundle is authentication tokens that need to expire after a certain period of time. Using the expiration bundle lets you delegate the work of expiration to RavenDB itself, through the bundle.

To use the Expiration Bundle, you will need to enable it on a database. When creating a new database, select Expiration on the available bundles screen:

www.dbooks.org

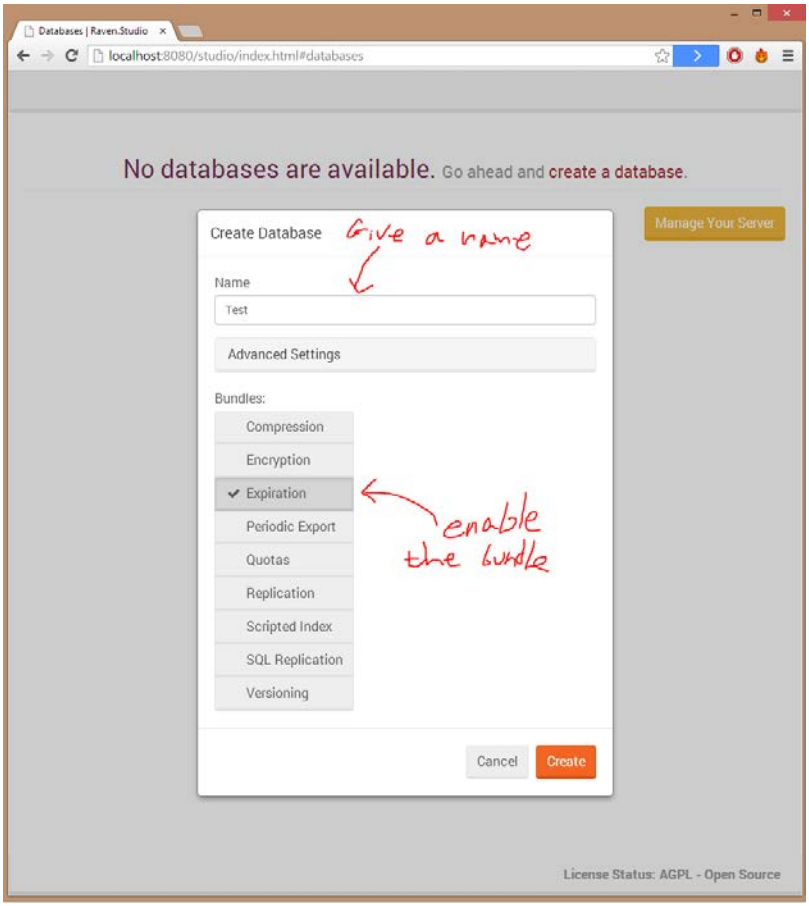


Figure 8.x: Creating a new database and enabling the expiration bundle

After the expiration bundle has been enabled on the server, the user can stamp documents with an expiration date and time to make them expirable:

```
var expiry = DateTime.UtcNow.AddMinutes(5); #A
using (var session = documentStore.OpenSession())
{
    session.Store(obj); #B
    session.Advanced.GetMetadataFor(obj)["Raven-Expiration-Date"] = new
RavenJValue(expiry); #C
    session.SaveChanges();
}
```

#A – Expiration date and time have to be in UTC
#B – Storing a new object; we could have loaded existing one as well
#C – Stamping the document with the expiry metadata

Now that you know of the expiration bundle and how to use it, let's have a quick look at how it all works. Under the hood, the expiration bundle has a background process which periodically queries for expired documents using an index it registers that indexes the `Raven-Expiration-Date` metadata, and if any are found they are then removed from the database.

Since this background process may not delete expired documents at the exact same time in which they were supposed to get expired, and also if it did RavenDB's asynchronous indexing process might make the deletion some time to propagate in all indexes, the bundle also explicitly blocks access to those documents (for write, reads and access from the indexing mechanism). This is done using *triggers*, which we will look at in more depth later in the chapter.

The Versioning Bundle

The versioning bundle is another example for a bundle that interacts with the document store, but instead of deleting documents it adds new ones constantly.

The idea behind this bundle is to painlessly allow you to have an audit-trail of documents by keeping track of each version stored for them. Every change made to a tracked document will cause a version document to be created. Viewing and browsing the audit trail is then just as easy as reading documents using semantic IDs.

The best way to understand what it does is to see it in action. Let's start by creating a new database and enabling the *versioning bundle*. Similarly to using the expiration bundle, the versioning bundle needs to be activated when creating the database:

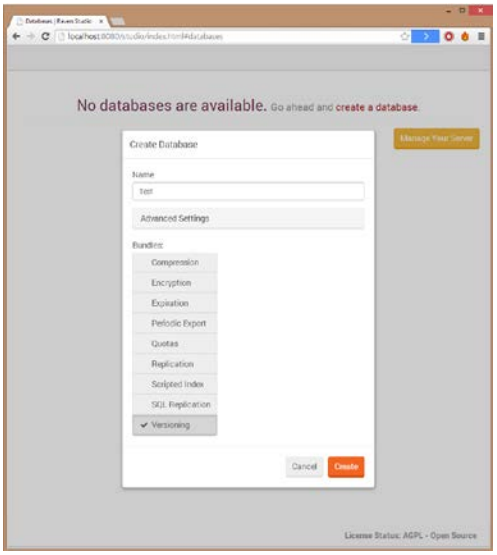


Figure 8.x: Creating a database and enabling the Versioning Bundle

www.dbooks.org

Immediately after creating the database from the Studio, you will be prompted to setup the Versioning Bundle settings. Just accept the defaults by clicking *Done*:

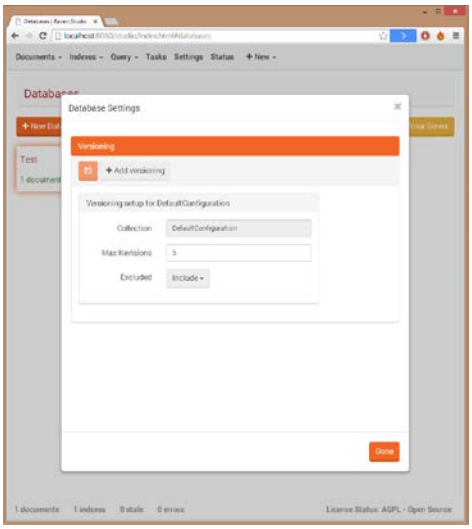


Figure 8.x: The versioning bundle settings prompt after creating a new database with Versioning Bundle enabled through the Management Studio.

Next, let's create a new document. Easily done by using the Management Studio, like we did so many times by now:

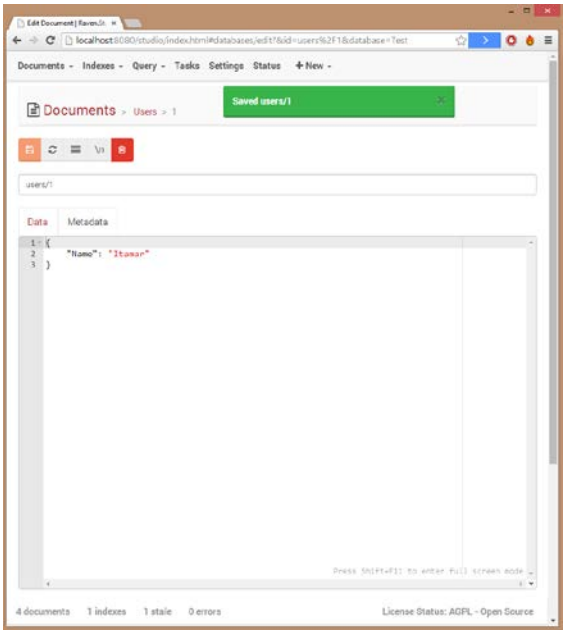


Figure 8.x: Creating a new document through the Studio

Now let's go and edit it by changing its content. Notice what happens once you save the changes you made to the document:

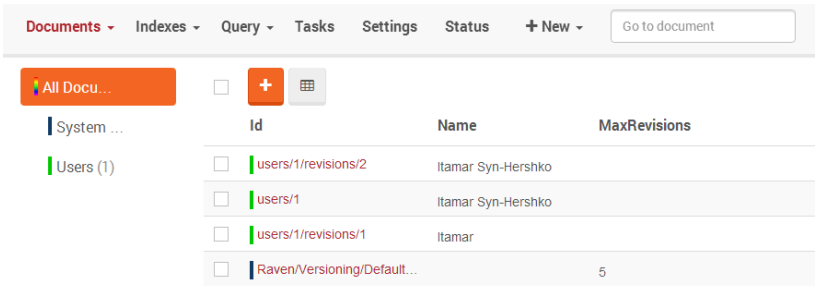


Figure 8.x: The versioning bundle keeps document revisions

As you can see, every change made to documents in a database when the versioning bundle is activated will be tracked, and older versions of documents will be persisted alongside the current version of that document. The revisions themselves use semantic IDs, so revisions for a document with ID `users/1` will be stored as `users/1/revisions/1`, `users/1/revisions/2` and so on, like shown in figure 8.x.

www.dbooks.org

By default all *collections* will be tracked for changes this way, and the last 5 revisions will be saved for each document. This can be changed by when setting up the database, in the bundle settings dialog that opens up.

The revisions documents are read-only – they can't be changed or deleted. RavenDB will refuse such requests, and this is the bundle's doing using *Triggers*. While we have mentioned Triggers before, there is one thing this bundle uses that we haven't seen before – client API integration.

To allow for easier interaction with the revision documents from the client side, the bundle also includes extension methods for getting the revision documents, `GetRevisionsFor<T>` and `GetRevisionIdsFor<T>`. Calling `session.Advanced.GetRevisionsFor<User>("users/1", 1, 5)` for example will fetch the last 5 revisions of this document, and allow you to display them to the user or do some processing over them (like computing a diff).

We will look at more client API integration points later in the chapter, but this is a good showcase of how a bundle can have extensions on the client side to complement what it is doing on the server side.

8.1.2 Using external bundles

By now we established that RavenDB can be extended, however we showcased extensions which are actually part of RavenDB itself. It is now time to look at extensions that are external to RavenDB. While the usage details is pretty much identical to that of the internalized bundles, it does require one additional step of installation:

- For a server-side bundle, put the compiled assemblies of the bundle and its dependencies under the *Plugins* folder under the installation folder of the RavenDB server. It is important not to forget to copy all the bundle dependencies as well, except of course from the RavenDB assemblies you may depend on.
- For a client-side bundle, add a reference to the compiled assemblies from your project.

Obviously, and like we have just seen, some bundles may have both server- and client- side extensions, and for them you will need to perform both steps. Once the bundle is installed, using it is no different than using the internalized bundles we've seen previously.

Let's have a look at an example of using a bundle, from installing it to using it.

THE UNIQUE CONSTRAINTS BUNDLE

The unique constraints bundle aims at solving a challenge with RavenDB we described back in chapter 5. If you recall, we explained how RavenDB has full ACID guarantees only when working with document keys. If you need to guarantee uniqueness of documents based on some data they contain, that data has to be represented in a document key.

We gave the example of users and guaranteeing uniqueness of their e-mail addresses. We said that while you *could* use their e-mail as their ID in your system, this will prevent you from allowing them to change it later. If this is a problem, you can always write marker documents in one transaction with the user document itself to guarantee uniqueness. That is, write the user document and another document with the current user e-mail in its ID (for example,

users/1 and users/1/me@mydomain.com) and by doing that and using optimistic concurrency you can effectively implement a way to enforce unique constraints on document properties.

This is exactly what the Unique Constraints bundle does, only it hides those implementation details behind a curtain and provides additional nice syntax and tooling with it.

To install the Unique Constraints bundle, take the Raven.Bundles.UniqueConstraints.dll file from the Bundles folder in the RavenDB download ZIP (which you can get from http://ravendb.net/download). Put this file under the Plugins folder of the RavenDB server, and restart RavenDB.

Next, install the client bundle. The easiest way to go about it is to use nuget: `Install-Package RavenDB.Client.UniqueConstraints`.

Once installation is done you can decorate your model classes with a `UniqueConstraint` attribute to mark it as such, and let the bundle do the heavy lifting for you:

```
class User
{
    [UniqueConstraint]
    public string Name { get; set; }

    [UniqueConstraint]
    public string Email { get; set; }

    public string FirstName { get; set; }

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

Additionally, the bundle lets you load documents by the constraints, or check if they are fulfilled before submitting a change to the database. You can read more about this bundle and it's usage in the RavenDB documentation: http://ravendb.net/docs/2.5/server/extending/bundles/unique-constraints.

8.1.3 Custom analyzers

Another, albeit different, type of server-side plugins are custom analyzers. In chapter 6 we saw you can specify a name of a Lucene analyzer to use for a textual field in an index, and explained the meaning of that feature, and what it does. While RavenDB comes bundled with quite a few Lucene analyzers, it is quite a common requirement to use custom analyzers – for example to handle HTML and XML string content more wisely, or to provide better support for search in various languages.

Writing a Lucene analyzer is not very difficult and actually pretty straight forward, but it's out of scope for this book. You can find a variety of guides on how to do this online.

Once you have created your own analyzer, or got your hands on an analyzer someone wrote and you want to use it from RavenDB, installing it is as easy as dropping the DLL (and dependencies, if any) to the `Analyzers` folder under the RavenDB server folder.

It will then get picked up automatically, without any configuration or work from your side, and you can start using it right away from your indexes. Use the instructions from chapter 6 for

www.dbooks.org

specifying the analyzer to use in your indexes – and don't forget the full name is expected (assembly name, namespace and class name).

8.2 Writing a custom bundle

There's no good rule of thumb when you need to write your own bundle. It will usually be required when you feel some pain with RavenDB and that pain is clearly not an indication you are using the wrong tool, just that what you are trying to achieve is a bit more delicate or just is not efficient enough when implemented using the standard tools.

The currently available bundles are a great example of where a bundle is needed:

- When the server needs to be aware of document changes, keep track of them and prevent deletions of this generated history. The best place to implement this logic is on a central place that is the server. This is the versioning bundle.
- Background or periodic tasks that deal with indexes and documents and potentially many of them. This is for example the expiration bundle, the cascade deletes bundle, and the periodic export bundle.
- Custom or complex index queries, like done in the MoreLikeThis bundle. There just isn't a way to implement such a query efficiently from the client.
- Index customization – like the encryption and compression bundles that add encryption and compression to RavenDB indexes.

The list could probably go on, but the gist is that bundles were meant to be used for implementing logic that would otherwise be impossible or very inefficient to do.

In this section we will cover the building blocks that are available for plugins, and we will conclude with a concrete example of writing a (rather naïve) plugin.

8.2.1 Building blocks

Any server-side plugin for RavenDB can implement *triggers*, *request responders*, or *tasks* (any combination of those, of course). Those are the building blocks of every plugin. Let's see what each of them provides, while keeping in mind we leave the fine grained details to RavenDB's documentation as those details may change any time.

Each of the building blocks – be it triggers, request responders or tasks – has complete access to the database underneath. The document store, the indexes and the rest of the infrastructure are all totally accessible and ready to be used.

TRIGGERS

Virtually every internal operation in RavenDB has a trigger which is fired when that event occurs. Those operations span from addition of a new document to the indexing thread asking for data. Each of those available triggers can be implemented by the bundle, and perform additional operations or completely block it.

Here is a list of the common triggers available:

- PUT triggers are available by implementing `AbstractPutTrigger`. By doing this you can override methods like `AllowPut`, `OnPut` and `AfterCommit` which will let you veto writes

amend the document before it is being written, or perform operations (like writing additional documents) after it has been written. Each of those methods is provided with the full details on the document it is being called for.

- DELETE triggers are available by implementing `AbstractDeleteTrigger`. As with PUT triggers, you can veto a delete (conditionally or altogether) by overriding `AllowDelete`, or perform additional operations on requested deletes by overriding `OnDelete`, `AfterDelete` and `AfterCommit`.
- Read triggers allow to control access to documents or manipulate them on the fly once they are being read. By implementing `AbstractReadTrigger` you can override `AllowRead` and `OnRead` to add your logic.
- Additional query processing can be made by implementing `AbstractIndexQueryTrigger` and overriding `ProcessQuery`. It is quite a low-level API, but you get a chance to change the Lucene query before RavenDB executes it on the index.

One naïve example of a trigger implementation would be the following Read trigger, which will stamp every document with the date and time in which the server delivered it to the client as a response to its request:

```
public class StampDocument : AbstractReadTrigger
{
    public override void OnRead(string key, RavenJObject document,
RavenJObject metadata, ReadOperation operation, TransactionInformation
transactionInformation)
    {
        metadata[“Raven-Requested-At”] = DateTime.UtcNow.ToString();
    }
}
```

Request Responders

RavenDB operates as a web-server, accepting HTTP requests and providing responses, effectively exposing a REST API over HTTP. This means it has HTTP endpoints it exposes, and using bundles you can create your own such endpoint.

Adding an endpoint may be useful when you want to allow for a custom query type for example. You can then implement the query runner and expose its results using a dedicated HTTP endpoint. To create this new endpoint you would implement your own *request responder*.

A request responder consists of a URL pattern, the supported verbs (HTTP verbs, such as GET and POST), and a Respond method. You define the URL pattern and HTTP verbs you want to expose the new endpoint as, and then implement the response logic.

A naïve responder implementation would be this nonsense hello-world sample:

```
using System;
using Raven.Database.Extensions;
using Raven.Database.Server;
using Raven.Database.Server.Abstractions;

namespace Raven.Database.Bundles.MoreLikeThis
{
```

www.dbooks.org

```
	public class MoreLikeThisResponder : AbstractRequestResponder
	{
		public override string UrlPattern
		{
			get { return "^/hello/?(.+)"; }
		}

		public override string[] SupportedVerbs
		{
			get { return new[] { "GET" }; }
		}

		public override void Respond(IHttpContext context)
		{
			var name = context.Request.QueryString.Get("name");
		    if (string.IsNullOrWhiteSpace(name))
		        name = "world";

		    context.WriteJson(new { Message = "Hello " + name + "!", SentAt =
DateTime.UtcNow });
		}
	}
}
```

Startup and Background Tasks

You could also initialize and execute background tasks, to be run on startup or periodically, by implementing `IStartupTask` or `AbstractBackgroundTask`. The latter is simply a shallow implementation of the former, which makes it easy to schedule periodic tasks. Both are executed for the first time when the server starts up.

A one-time or periodic background task can be used for many things. You can use it to cleanup data (like the expiration bundle does) or you can use it for monitoring. A one-time startup task can be used to notify you whenever the server goes up, so you know it went through a restart, for example.

Implementing a task is rather simple. Here is an example of a task which will send an email every hour (or when the server starts up), just to notify the IT manager the server is alive. I would advise you to get better monitoring tools, but here it goes:

```
public class HourlyMailerTask : AbstractBackgroundTask
{
    protected override bool HandleWork()
    {
       var message = new MailMessage("ravendb@company.com","it@company.com")
            {
                Subject = "RavenDB server working.",
                Body = "Current time: " + DateTime.UtcNow.ToString()
            };

        using (var smtpClient = new SmtpClient("mail.company.com"))
        {
            smtpClient.Send(message);
        }

        return true;
```

```
        }

        protected override TimeSpan TimeoutForNextWork()
        {
            return TimeSpan.FromHours(1);
        }
    }
```

8.2.2 Client-side extension points

Some extensions you may want to have some client API integration to complement it with, or maybe your idea of extension is just client-side based, with no server integration. That is completely legit.

The client API has quite a few extension points. In fact, it is probably impossible to list all the possible integration points available to you on the client API, so I will list some of the important ones to be familiar with.

Extension Methods

One of the nicest features of C# is the ability to define a method on an object without subclassing it, and this is done using *extension methods*. If you are not familiar with the concept I strongly recommend you read about it (here for example: http://msdn.microsoft.com/en-us/library/bb383977.aspx or in chapter 10 of *C# in Depth* by Jon Skeet).

Using an extension method you can define additional methods on main RavenDB objects, like the DocumentSession and DocumentStore. This is a very neat way of exposing new functionality in an easy to use and easy to access way.

For example, this is how the versioning bundle exposes the GetRevisionsFor<T> method from session.Advanced:

```
public static T[] GetRevisionsFor<T>(this ISyncAdvancedSessionOperation
session, string id, int start, int pageSize)
{
    var inMemoryDocumentSessionOperations =
((InMemoryDocumentSessionOperations)session);
    var jsonDocuments =
((DocumentSession)session).DatabaseCommands.StartsWith(id + "/revisions/",
null, start, pageSize);

    return jsonDocuments
.Select(inMemoryDocumentSessionOperations.TrackEntity<T>)
            .ToArray();
}
```

Listeners

Client-side listeners are a way for the client API to notify you of events that are happening as part of the communication with the server. For example, listeners can tell you whenever a document is going to be sent to the server to be stored, or when a query was issued. By

www.dbooks.org

implementing your own listener implementation, you can plug in your own logic to be performed at those events.

You can implement one of the following interfaces to implement your own listener logic:

- `IDocumentDeleteListener` provides a way to invoke code before a delete operation is sent to the server. For example you could fail a delete on the client side by overriding `BeforeDelete` and throwing an exception from within it.
- `IDocumentStoreListener` allows you to execute custom code before or after the Store command is executed on the server, or to amend it before it is sent there.
- Implementing `IDocumentQueryListener` lets you execute code before a query is being sent to get executed on the server. For instance, it gives you an opportunity to customize the query or its parameters.
- The conversion listeners will let you intervene with the serialization and deserialization process. By implementing `IDocumentConversionListener` and its methods `EntityToDocument` and `DocumentToEntity` you can take control of how CLR objects are serialized into a JSON document and vice versa. There are additional ways of interacting with this process, and they are discussed under *Conventions* in the next chapter.
- Additional conversion listeners available by implementing `IExtendedDocumentConversionListener` allow to perform additional operations on the client side whenever a conversion event occurs.

Another type of client-side listener is the conflict resolution listener. Like we seen in chapter 7, when master-master replication is enabled you should take into account document conflicts will happen and it is your responsibility to resolve them. Resolving a document conflict is quite a straight-forward operation as we've demonstrated there, but by implementing `IDocumentConflictListener` you can create a listener and off-load that conflict resolution there, instead of having this in your code. This is also a good way for creating a single conflict resolution strategy and synching it between different clients and applications.

As an example, here is how to always automatically resolve to the newest version of the document in a case of a conflict using a client-side listener. It is not advisable to use this strategy without fully understanding the consquences, but it is a simple one to give as a code example:

```
public class TakeNewestConflictResolutionListener : IDocumentConflictListener
{
    public bool TryResolveConflict(string key, JsonDocument[] conflictedDocs,
out JsonDocument resolvedDocument)
    {
        var maxDate = conflictedDocs.Max(x => x.LastModified);
        resolvedDocument = conflictedDocs.FirstOrDefault(x => x.LastModified
== maxDate);

        return resolvedDocument != null;
    }
}
```

Once you have implemented one or more listeners, you will need to register them manually by calling `RegisterListener` on your `DocumentStore` instance, e.g.:

```
documentStore.RegisterListener(new TakeNewestConflictResolutionListener());
```

8.2.3 Wrapping up

Now that we mapped the various integration points of RavenDB, and the building blocks for writing a bundle, you are pretty much set to go for writing your own.

If you indeed intend on writing a plugin on your own, the recommended way to go about it is to create a project for the server-side plugin, and a project for the client-side plugin. You can skip creating a project if you don't need either sides, of course.

Once you have created the projects, add references to Raven.Database from the server-side bundle, and to Raven.Client.Lightweight from the client-side bundle.

www.dbooks.org

9 Advanced capabilities

This chapter covers

- Getting more of your queries
- Documents metadata
- Being notified from the client when data changes on the server
- Streaming unlimited number of documents from RavenDB
- Server-side updates via commands and scripts

We are about to wrap-up our journey learning RavenDB. By now you should have everything you need to start building pretty much any type of application with RavenDB as its backing store.

There are only two bits remaining. The first is a discussion on how to deploy and manage a RavenDB server instance. This is our topic for the next chapter, the last one in the book.

The other thing that we still have to look at is a collection of functionalities and capabilities of RavenDB we haven't had the place or time to look at yet. This is what we are going to do in this chapter.

Each individual topic discussed here can be discussed in great length, much more than this chapter can allow for. So instead of dropping those topics completely I chose to discuss them in short, and group them in one "garage" chapter.

9.1 Advanced querying

In previous chapters we talked about RavenDB indexes at length. We saw how they operate on data, and how they use Lucene and Map/Reduce functions to map document properties and results of aggregation operations into searchable indices.

In this section we are going to use this knowledge of the underlying building blocks, and build on to making your queries more powerful and efficient yet, by teaching some more features and giving some advice.

We will start by explaining the DocumentQuery API, which maps closely with the Lucene query syntax that is used under the hood, and provides a very powerful way of constructing queries manually. We will then discuss ways of making queries more performant, by leveraging caching properly and also the concept of lazy queries. Lastly, we will look at the query intersection feature, which makes some more complex queries possible.

9.1.1 The Avanced.DocumentQuery API

Throughout this book we mostly used the Linq provider from the .NET Client API to query RavenDB. When we were working with the Management Studio, we used the Lucene syntax for querying and typed those query strings directly in the Management Studio. You may have wondered how the two relate.

As I've indicated previously in the book, the Linq provider used for querying with the .NET Client API in fact outputs a Lucene query that in turn is sent to the server. If you inspect the network traffic between your .NET application using the RavenDB Client API, you will see the queries produced by the Linq API are exactly the same as the ones you'd use to type in the Management Studio for querying.

Sometimes it is easier to build a query directly with the operators available through a Lucene query as opposed to Linq. A middle-tier API in the Client API allows you to construct the Lucene query directly, without using the Linq provider. It is accessible via the session.Advanced.DocumentQuery<T>() method, and can be used as follows:

```
session.Advanced.DocumentQuery<Book>()
                    .WhereEquals("Author", "Dan Brown")
                    .AndAlso()
                    .WhereLessThan("Price_Range", 50.0f)
```

Is equivalent to:

```
session.Query<Book>()
    .Where(x => x.Author.Equals("Dan Brown") && x.Price < 50.0)
```

Both boil down to this Lucene query:

```
Author:"Dan Brown" AND Price_Range:[* TO 50]
```

Thanks to a more explicit query definition, it is often easier to construct complex queries using this API, as opposed to expressing them as Linq. Some querying abilities are also only accessible using this low level API.

Listed below are some of the more common operators available through the DocumentQuery API and their intended usage:

- WhereEquals() – this should be used for comparison type queries, as opposed to using plain Where, as this method will apply several important string escaping routines.
- Search() – Just like the standard Linq provider, use this method to execute full-text

www.dbooks.org

searches on fields.

- OrderBy() / OrderByDescending() – Selecting a field to sort by. There's also the OrderByScore() variant to sort results based on the Lucene score, and SortByDistance() to force sorting the results by distance when executing geo-spatial query.
- OpenSubclause() / CloseSubclause() – The ability to group several conditions together.
- AndAlso() / OrElse() – define relations between query clauses or subclauses, whether it is an AND or an OR relation.
- Negate() – negates the operation (or subclause) following the call to this operator.
- WhereBetween() / WhereGreaterThan() / WhereLessThan() etc – all the various ways one could define a range query with.

9.1.2 Plan for effective client caching

One of the design decisions behind RavenDB is to make reads as fast as possible. Following this, queries were designed to always be made on pre-computed indexes and never do any calculations. We have discussed this in length back in Chapter 3.

Another optimization principle in computer science is the Principle of Locality, which states that the closer that a piece of data is located to the consumer, the faster it is able to process it. It is actually pretty trivial when you think of it.

Since locality matters, it is important to try and utilize RavenDB's client side caching as much as possible. By default RavenDB will cache query results as well as retrieved documents in the caller's DocumentStore object. If you read the same documents frequently, or issue the same query over and over again, caching will kick in and provide you with nice speed up.

However, sometimes it is easy to miss this opportunity without even noticing. Take this query for example:

```
session.Query<User>().Where(u => u.LastLoggedIn <= DateTime.UtcNow.AddDays(-
      1)).ToList();
```

This query will return exactly what you'd expect it to return – all users who haven't logged in since yesterday. But this query will never benefit from caching, because it will generate different Lucene queries every time it is called. For example, if called once a minute, it will look like this:

```
LastLoggedIn:[* TO 1421165580]
LastLoggedIn:[* TO 1421165640]
LastLoggedIn:[* TO 1421165700]
LastLoggedIn:[* TO 1421165760]
LastLoggedIn:[* TO 1421165820]
```

To make queries caching effective, consider rounding values that are prone to changes, forgoing precision maybe but getting much better performance altogether (and lower memory footprints on clients, too!).

9.1.3 Lazy operations

Another important optimization rules is to make as few calls to the database as possible. Indeed, the session object will throw an exception if you will use it to perform more than a sane amount of operations (30 by default). However in some cases you just can't get away of performing multiple read operations when rendering a single page. Maybe you are querying for products and user info and categories and latest news all to be displayed in one page, and you can't really join all of those into one.

To enjoy the benefits of the two worlds – both sane code paths and efficient querying performance from RavenDB, you can issue multiple read operations as one using Lazy operations. This will let you issue all of those in as fewest database calls as possible which will result in noticeable performance gains, especially if the database is on a remote server.

Both Query and Load operations can participate in a lazy batch, and it goes as follows:

```
User user = null;
IEnumerable<Order> orders = null;
IEnumerable<City> cities = null;

session.Advanced.Lazily.Load<User>("users/1", x => user = x); #A

session
      .Query<Order>()
      .Where(x => x.UserId == "users/1")
      .Lazily(x => orders = x);

session
      .Query<City>()
      .Where(x => x.Name == "New York")
      .Lazily(x => cities = x); #B

session.Advanced.Eagerly.ExecuteAllPendingLazyOperations(); #C
```

#A – Lazily define a load operation. This isn't being executed now.
#B – Neither are these queries.
#C – Execution happens here, all three read operations at once, in a single request.

9.1.4 Query intersection

Query intersection is a nice feature of RavenDB which allows for query results to be intersected on the server side. It is best explained using an example, so let's assume we are selling t-shirts and have the following data model to represent t-shirts and the possible color and size combinations each shirt type has:

```
public class TShirt
      {
            public String Id { get; set; }
            public String PrintDescription { get; set; }
            public int BarcodeNumber { get; set; }
            public List<TShirtType> Types { get; set; }
      }

      public class TShirtType
      {
            public String Colour { get; set; }
```

www.dbooks.org

```
            public String Size { get; set; }
      }
```

A typical index to enable looking up shirts will look something like this – note how we iterate through all shirts and all types, in order to index every shirt variant as a unique shirt:

```
from tshirt in tshirts
from type in tshirt.Types
select new
      {
               PrintDescription = tshirt.PrintDescription,
               Color = type.Color,
  Size = type.Size,
               ReleaseYear = tshirt.ReleaseYear
      };
```

Now, this index allows us to locate shirts that have Blue Small shirts, or Red Large ones. Since each variant has its own index entry, it is as easy as issuing the following query:

```
var shirts = session.Query<TShirt>("TShirtsIndex")
                    .OrderBy(x=>x.BarcodeNumber)
                    .Where(x => x.Types.Any(t => t.Colour == "Blue" && t.Size ==
      "Small"))
                    .ToList();
```

However, if we wanted to find all shirt types that have 2 or more specific variants, this isn't possible to do. The query language is just not strong enough to pronounce that. Instead, we can combine several queries like above and intersect the results. Instead of doing this on the client side, RavenDB supports query intersection on the server side using the `Intersect()` operator, like shown here:

```
var shirts = session.Query<TShirt>("TShirtsIndex")
             .OrderBy(x=>x.BarcodeNumber)
             .Intersect()
             .Where(x => x.Types.Any(t => t.Colour == "Blue" && t.Size == "Small"))
             .Intersect()
             .Where(x => x.Types.Any(t => t.Colour == "Red" && t.Size == "Large"))
             .ToList();
```

Query intersection will return only results that match all subqueries, and those subqueries have to be made against the same index.

9.2 Working with documents metadata

Each and every document in RavenDB has metadata attached to it, also stored in JSON format. Metadata is used by RavenDB internally to store some additional properties on documents, but it can also be used by you when needed – either using the internal metadata yourself or storing your metadata.

In general, prefer to store document related data in the document itself. Only use the metadata option when the data you are looking to store is indeed metadata on the document itself – like data pertaining to how it should be stored, when it should be expired and so on.

The metadata entries being set and then used by RavenDB are as following:

- `Raven-Clr-Type` - Records the CLR type, set and used by the JSON serializer and deserializer in the Client API.
- `Raven-Entity-Name` - Records the entity name, usually a plural form of the class name as seen by the client API. This is also being used to determine the name of the RavenDB collection this entity belongs to.
- `Non-Authoritive-Information` - This Boolean value will be set to true if the data received by the client has been modified by an uncommitted transaction.
- `Temp-Index-Score` - When querying RavenDB, this value is the Lucene score of the entity for the query that was executed.
- `Raven-Read-Only` - This document should be considered read only and not modified, respected internally by RavenDB in various places.
- `Last-Modified` - The last modified timestamp for the entity.
- `@etag` - The e-tag (entity tag) value the database was holding globally when this document was last. Being used internally to drive the indexing process, and also to power optimistic concurrency (which we discussed in chapter 5).
- @id - The entity id, as extracted from the entity itself.

Additionally, when replication is enabled additional metadata records are added to enable tracking the replication status for the document. Many bundles add their own metadata entries as well, for example the expiration bundle which by default adds and expiration timestamp to the document metadata.

As always, you don't have to use the metadata entries provided by RavenDB, and you can use your own keys for your own values. Just make sure not to invalidate metadata entries being used by RavenDB or its bundles internally, and to not to overuse this concept. Pretty much the standard disclaimer.

9.2.1 Reading document metadata

Document metadata is not being deserialized together with the document instance itself. Instead, it is available to be loaded for a given object which was deserialized from a document using the client API:

```
var product = session.Load<Product>(1);
RavenJObject metadata = session.Advanced.GetMetadataFor(product);
```

It is then retrieved as a RavenJObject object, which is a C# wrapper for a JSON object. It may be easier to think of it as a dictionary object, one you could get values out of using keys – or iterate through all keys and values associated with each.

9.2.2 Querying on metadata

To query on document metadata, it has to be indexed. Here is an example of a Map function of an index accessing metadata entries:

```
from doc in docs
```

www.dbooks.org

```
where doc["@metadata"]["Raven-Entity-Name"] != null
select new { Tag = doc["@metadata"]["Raven-Entity-Name"] };
```

This is, in fact, a Map function very similar to the one used to define collections in RavenDB. In essence, it finds all the Entity Names and then allows you to query based on that (and also to facet on it, as we have seen in chapter 6).

When indexing it you get the chance to give it the name in which it will be visible from the index, therefore querying on it is as simple as issuing a query on the index field name. This is usually when `DocumentQuery` comes in handy, because this index field name will most likely not be available from the type you are querying on using the Linq provider.

9.2.3 Manipulating metadata

Manipulating metadata is very easy to do, all that is required is to change the RavenJObject instance accordingly and persist it again:

```
var product = session.Load<Product>("someproductid");
RavenJObject metadata = session.Advanced.GetMetadataFor(product);

Metadata["Raven-Entity-Name"] = "MyEntityName";
Session.SaveChanges();
```

Explicitly manipulating the metadata by changing or adding keys of your own requires persisting it back by having a call to `session.SaveChanges()` when you are done.

Let me just remind you that as we have seen previously in chapter 8, you can also use Listeners (the client-side extension points) to manipulate metadata – or use metadata during document manipulation.

9.3 The DocumentStore conventions

RavenDB prefers Conventions over Configurations (http://en.wikipedia.org/wiki/Convention_over_configuration), and as such relies heavily on conventions being defined correctly. Almost every part of RavenDB uses those conventions one way or the other, and they dictate everything from how an entity is translated to JSON to how RavenDB servers are approached with queries.

Since conventions can be changed rather easily, it makes them a central place for customizing RavenDB's behavior. Usually you will find RavenDB's defaults quite reasonable, but still it does make sense to be familiar with what can be done.

The RavenDB Conventions are accessible per DocumentStore, in one central place that's called… wait for it… Conventions. There you will find many properties and methods with long names, and in this section we will go through some of them to see what they do and how they can be used.

9.3.1 The identity property locator

Once you are storing an object using the Client API, RavenDB needs to determine the property that's being used to store the object ID. This also holds true when querying RavenDB, when

the client API deserialized an object out of a document string, and needs to set this property accordingly with the document ID.

By default, and we have seen this already, RavenDB will treat properties named "Id" as the identity property, and this allows us to just add a `string Id` property and have the document ID accessible to us easily from the deserialized object.

However, if we wanted to change this behavior, for example by using a property called "Key" for this instead, all we have to do is change the `FindIdentityProperty` convention accordingly, and have it return true on when it sees a property named "Key". This can be done as follows:

```
store.Conventions.FindIdentityProperty = prop =>
{
    return prop.Name
            .Equals("Key",StringComparison.InvariantCultureIgnoreCase);
};
```

9.3.2 Collection name resolution

Another common example for using the conventions is to change the way Collections names are derived out of class names. By default RavenDB will pluralize a class name, and make that the entity-name of objects stored from that type. However, in some cases you may want more control over this convention. Meet `FindTypeTagName` and `FindDynamicTagName`.

`Conventions.FindDynamicTagName` is a way for you to provide RavenDB a different strategy that works better with your model, for instance if you wanted to keep the default behavior except from one class, for which you want to always give a custom type name (and therefor, a collection name), you can use the following convention override. Just make sure to define it in all clients accessing the same RavenDB server, and also to set it when the DocumentStore object initializes:

```
store.Conventions.FindTypeTagName = t =>
{
    if (t == typeof(MyClass)) return "MyCustomCollectionName";
    return Raven.Client.Util.Inflector.Pluralize(t.Name);
};
```

Using this method you can for instance put many object types deriving from one base type in your model in the same collection, and various other common use cases. `Conventions.FindDynamicTagName` plays the exact same role, but is called for dynamic types.

9.3.3 Custom document key generation

By default RavenDB uses the HiLo algorithm to assign unique IDs to new documents. Those IDs are guaranteed to be unique under a collection, and the general advice is to accept RavenDB's default and not change this.

However, sometimes it is required to change this to a custom implementation, or an implementation that does things a bit differently. For instance, if you wanted to ensure document IDs are always sequential, without gaps, you'd prefer the identity key generation

www.dbooks.org

strategy over the HiLo based one. If you wanted to set it to be RavenDB's default, you could change the conventions as follows:

```
store.Conventions.DocumentKeyGenerator = (dbname, commands, entity) =>
      store.Conventions.GetTypeTagName(entity.GetType()) + "/";
```

Taking advantage of the fact every document key which ends with a trailing / character will cause the RavenDB server to assign it the last free key, instead of negotiating unused keys prior to insertion with the clients.

9.3.4 Advanced serialization and deserialization

To serialize and deserialize JSON documents, RavenDB uses Newtonsoft's JSON.NET internally. This library was actually internalized so it doesn't conflict with other libraries you may be using and are dependent of different versions of JSON.NET. Yes, .NET is sometimes joyful like that.

Regardless, RavenDB still exposes functionality which lets you decide how you want your objects serialized or deserialized, and following are several examples of common things you may find yourself wanting to do.

For those amendments to work, you will need to have a using statement importing `Raven.Imports.Newtonsoft.Json` in your class.

IGNORING A PROPERTY

Assume you have a property in your model class, but for some reason you don't want it persisted. Maybe this is a property being used for display only, or for validation. Not a problem! Just add a `[JsonIgnore]` property on top of it and you are all set.

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

      [JsonIgnore]
      public string SecretText { get; set; }
}
```

SERIALIZING UNDER A DIFFERENT NAME

Or say you wanted this property persisted, but using a different name. Because you cursed your boss in your code and you don't want this to be reflected in your database for all to see, for example:

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

[JsonProperty(PropertyName = "ILoveMyBoss")]
      public string DreadfulBossName { get; set; }
}
```

MORE CUSTOMIZATIONS OPPORTUNITIES

You can get more control over the serializer and deserializer used by interacting with the Conventions when it prepares the serializer instance. The method registered under `Conventions.CustomizeJsonSerializer` is called in Just The Right Time™ for this, and if you are familiar with Newtonsoft's Json serializer you can find plenty of customization opportunities there.

Additionally, JSON.NET has a `ContractResolver` class which defines the serialization strategy and can be tweaked to behave differently when needed. For example by chaning the ContractResolver in use you can control the required visibility for class members in order for them to be serialized:

```
store.Conventions.JsonContractResolver = new
DefaultContractResolver(shareCache: true)

        {

                DefaultMembersSearchFlags =
                                BindingFlags.Public | BindingFlags.Instance

        };
```

9.4 The changes API

The Changes API is a way for a client to be notified of changes to the remote database server without polling it periodically. Using this API you subscribe to push notifications from a RavenDB client, and have events fire when those notifications come in.

There are several types of notifications that one can subscribe to:

- **Document changes** – once a document is being written to a notification is sent. This can be a specific document by id, all documents in the system, or documents by their collection, CLR type, or by an ID prefix.
- **Index changes** – notifies all subscribed clients whenever an index definition changes on the server. Can be set for a specific index, or all indexes.
- **Transformer changed** – likewise, notifies all subscribed clients when a result transformer definition is changed on the server.
- **Replication conflicts** – all subscribed clients will get notified when a replication conflict occurs. This allows to register "conflict resolver" clients, which will load the conflicting documents automatically and handle the conflict manually or using conflict listeners.
- **Bulk inserts** – send a notification whenever a bulk insert operation is initiated, is ended, or has an error thrown.

9.4.1 Subscribing and unsubscribing to notifications

`session.Advanced.Changes()` will provide you with the various notifications type to register to. Calling the desired variant will return a `subscription` object which implements `IDisposable`, meaning you need to make sure to call `Dispose()` on it when your application quits:

www.dbooks.org

```
IDisposable subscribtion = store
      .Changes()
      .ForDocumentsInCollection<Order>()
      .Subscribe(new OrdersObserver());
```

The OrdersObserver instance is an implementation of IObserver<DocumentChangeNotification>, which in turn is defined as:

```
public class OrderObserver : IObserver<DocumentChangeNotification>
{
    public void OnCompleted()
    {
    }

    public void OnError(Exception error)
    {
        #A
    }

    public void OnNext(DocumentChangeNotification documentChangeNotification)
    {
        if (documentChangeNotification.Type == DocumentChangeTypes.Put) #B
        {
            using (var session = MvcApplication.Store.OpenSession())
            {
                var order =
  session.Load<Order>(documentChangeNotification.Id);// get document by id
                    #C
            }
        }
    }
}
```

#A – handle exceptions here
#B – You get the chance of choosing which events to handle. In this case, we want to handle document writes.
#C – Do something with the Order, which we just loaded by ID. I.e., send an email notifying your garage to prepare the order for shipping.

MAKE THIS EASIER WITH RX If you are familiar with Reactive Extensions (Rx) you may find them very helpful here, especially for avoiding to explicitly implement IObservers. Read more about Rx here: https://rx.codeplex.com/.

Once you have subscribed to notifications, unsubscribing is simply a matter of disposing of the subscription object you got when subscribing. As long as that subscription object is around, you will continue receiving notifications.

9.5 Unbounded results set

Perhaps one of the most opinionated features of RavenDB is to short circuit query results, and make sure paging of result sets is enforced on the user (or, developer) rather than sending unbounded results.

While this makes sense for general operations, sometimes you do need to retrieve data from your RavenDB instance with stable paging, that is – without omitting any documents in the way, or the possibility of getting duplicates.

Such scenarios range from various data processing requirements or simply to back up the data using remote code (we will cover other, more structured ways of creating backups in the next chapter). For those scenarios RavenDB provides the streaming API, often times called the unbounded result set API; using this API you can stream documents matching any query out of RavenDB without paging and without any hard limit of the number of results you can get.

Key takeaway: Use the streaming API only for select use cases

With great power comes great responsibility. Don't use the streaming API to get unbounded result set as a way to bypass the limits RavenDB poses on the standard query API. This will be shooting yourself in the foot.

Only retrieve unbounded number of results using this API when you absolutely need the entire (or most of) the results of a given query. For example for performing backups, or performing various data processing operations.

Using the streaming API is as easy as executing the query using the `session.Advanced.Stream` API, like shown here:

```
var query = session.Query<Book>("BooksIndex").Where(x => x.IsListed); #A

QueryHeaderInformation queryHeaderInformation; #B
using (var enumerator = session.Advanced.Stream(query, out
       queryHeaderInformation))
{
       while (enumerator.MoveNext()) #C
       {
              Book book = enumerator.Current.Document; #D
       }
}
```

#A – Any query can be used here, also of type DocumentQuery, with or without Where clause, sorting order, etc. The index has to be already there though.
#B – This QueryHeaderInformation object can give us some insights to the query results, like total number of results or whether the results are stale.
#C – The actual streaming loop, which you break out of anytime
#D – An instance of the streamed, serialized object, ready for your consumption

The streaming API has variants for streaming documents directly without hitting the indexes layer, and also an async variant for executing the streaming asynchronously.

Streaming data has to be done in a single method, and will reuse one connection to the database. In turn it will guarantee stable results stream that isn't affected by concurrent changes to the database, and isn't too bothersome to handle like paging is.

www.dbooks.org

9.6 Performing server-side updates

Sometimes it doesn't make sense to update documents in client code. For example when you try to make a very small change to a very large query and you'd like to avoid the round trip, especially if that change happens frequently. Or if you have many documents to change and it just doesn't make sense to iterate through all of them on the client side.

In this section we will look at document patching, and the ability to execute patching and delete commands on a set of documents matching a given query. This is all the tooling we need in order to perform server-side updates to documents when needed.

9.6.1 Patching

A Patch is a way to ask RavenDB to patch a single document with a given set of instructions. Those set of instructions are set as a PatchRequest object, via the `DatabaseCommands.Patch` API. For example, this is how you'd add a Comment object to a BlogPost object entirely on the server side, without loading the BlogPost document on the client:

```
var comment = new BlogComment
{
      Title = "Foo",
      Content = "Bar"
}; #A

documentStore.DatabaseCommands.Patch( #B
      "blogposts/1234", #C
      new[]
            {
                  new PatchRequest
                        {
                              Type = PatchCommandType.Add, #D
                              Name = "Comments", #E
                              Value = RavenJObject.FromObject(comment) #F
                        }
            });
```

#A – Creating the Comment object, we will transform it to json in just a bit
#B – DatabaseCommands.Patch is the API we use to execute patch requests
#C – The document ID to patch
#D – There are several types of patch requests available. This one will add an object to an array or list
#E – The document property to operate on in the document
#F – The value to work with, in this example we add this to the end of the Comments array in the document.

The PatchRequest object supports several types of actions. In the example above we were using Add to add an item to the end of an array, but there are several more supported:

- Set – sets a value to the specified document property
- Unset – removes the specified document property
- Add – inserts a value to the end of an array
- Insert – inserts a value to an array at the specified position
- Remove – remove a value in the specified position from an array

- Modify – support for a nested patch operation
- Inc – increments (or decrements if a negative value is provided) a numeric document property.
- Copy – copies a value from the specified property into a new property in the same document
- Rename – renames the specified property, overriding the property if it already exists.

9.6.2 Scripted patching

Sometimes the `PatchRequest` object we just seen is not expressive enough for the change we are trying to do. For this end RavenDB supports `ScriptedPatchRequests`, which as their name suggest will execute a script (written in Javascript) against the document.

For example, the following PatchRequest will create a new property called “FullName” based on FirstName and LastName, and then delete latter two:

```
new ScriptedPatchRequest()
{
	Script = @"
		this.FullName = this.FirstName + ' ' + this.LastName;
		delete this.FirstName;
		delete this.LastName;
	"
}
```

Since writing scripts can be sometimes tricky, you can use the management studio to try out new scripts against existing documents, and preview the results before committing any change.

TODO

9.6.3 Set-based operations

Patch requests, as well as scripted patch requests and delete operations, can be executed against an index, so all matching documents matching the query sent will be updated based on the patching command submitted.

Set based operations (that is: executing an operation on a set of documents matching a query) are supported in RavenDB via the DatabaseCommands, for example to execute a deletion based on a query you will execute:

```
documentStore.DatabaseCommands.DeleteByIndex("IndexName",
new IndexQuery
		{
				Query = "Author:RavenDB"
				}, allowStale: false);
```

Queries are expressed in the Lucene query syntax directly. If you are unsure of the format, you can build a query using the Linq provider and execute ToString() on it and it will provide you with the Lucene query string.

www.dbooks.org

Another point to note is that by default set based operations will not run on stale index results. You can either specify allowStale: true when executing the query, or set a value to the Cutoff property in the IndexQuery object sent in the request.

As noted, not only deletions are supported in set based operations. You can use the PatchRequest and ScriptedPatchRequest objects we have just seen to update a set of query results by executing the patch against each and every query result:

```
store.DatabaseCommands.UpdateByIndex("Posts/ByUser",
    new IndexQuery{Query = "UserId:" + userId},
    new AdvancedPatchRequest
        {
            Script = @"
var user = LoadDocument(this.UserId);
this.UserName = user.Name;
"
        });
```

As you can see, we can even call the famous LoadDocument method we already seen in index definitions to empower patch scripts even more.

9.7 Summary

9.7.1 Leveraging aggressive caching

Any Query or Load operation made against RavenDB is cached by default by the DocumentStore object in the client API. Query or Load requests that have already been made before will result in a lightweight HTTP operation is subsequent calls. The RavenDB server will send back a 304 HTTP Not-Modified response and by that tell the client API to use the cached version, instead of sending the actual data back, thus saving on network traffic and latency times. This is of course unless changes have been made to the requested data, in which case the data will be sent in the response.

We already mentioned how important it is to make sure queries are cachable – for example by not using constantly changing values in them, like DateTime.Now for example. The cache implementation in the Client API plays an important part in making the operation against RavenDB fast.

But the default cache implementation still performs a check against the database for every request. If the data hasn't changed since the last similar request this communication would be short and lightweight, but it will still impose some overhead and latency. If on the other hand, the data changes frequently, this would mean the cache will hardly be ever used.

RavenDB offers you the ability to opt-out of this check against the server, and opt-in for Aggressive Caching. What Aggressive Caching does is cache the results of a query once, and only invalidate it after a given period of time, without ever checking whether the data have changed on the server. The biggest benefit is that those operations now happen entirely in memory – there is no network involved most of the time. The drawback is obviously of working with cached data, that might have changed in between.

This feature can be enabled on an opt-in basis. By wrapping any Load or Query call with AggressivelyCacheFor, you tell RavenDB to aggressively cache the results for the specified time span:

```
Using
(session.Advanced.DocumentStore.AggressivelyCacheFor(TimeSpan.FromMinutes(5)))
{
    var user = session.Load<User>("users/1");
}
```

Any other similar calls not wrapped like this by default will still go to the server on every call to check if there is a new version of it available.

RavenDB allows you to force a query or Load operation to go to the server, thus updating the cache with the latest version of a document or query results. To do that, simply wrap a Query or Load call with DisableAggressiveCaching, like so:

```
using (session.Advanced.DocumentStore.DisableAggressiveCaching())
{
    var user = session.Load<User>("users/1");
}
```

Applicable Scenarios

There are multiple scenarios in which it makes sense to use Aggressive Caching, taking into account the side effects of using a possibly out-of-date data.

1. Very common queries, which showing stale results for is acceptable. A web page listing blog posts is a good example. Aggressively caching such queries for a couple of minutes will have hardly any effect on the display, and will save on a lot of work on both client and server side.
2. Queries for data that changes very infrequently. Aggressively Caching
3. Caching of large documents, like configurations or large hierarchical structures like categories tree in an e-commerce website. Sometimes it isn't really possible to completely avoid large documents, thus aggressively caching them for standard use basically means having them locally, in memory for processing, just like standard large data structure.

The decision on how long to Aggressively Cache a document really depends on the situation, but can range from seconds (for very highly used queries on a very popular website, for example) to hours or even days (for a lot of other stuff, and using manual cache invalidation to refresh the data when necessary).

^ ^ ^ Cache auto expires when changes happen in the DB

www.dbooks.org

10

Getting ready for production

This chapter covers

- Deploying and maintaining RavenDB in production
- Configuration options
- Troubleshooting issues on production
- Creating data back-ups, and how to restore from them
- Performing data migrations on live databases

Welcome to the last chapter of the book!

So far we focused on getting to know RavenDB – the database – from the developer's perspective. We used code, or the management studio to create and edit documents, define indexes, and shape the structures of both to make RavenDB as performant and useful as possible.

In this chapter we are going to focus on RavenDB as a database product in production, and the various concerns that surround it. We will start by talking about various deployment options, including access control and the available configuration options.

We then move to discussing backup strategies – this is a very important topic for every database engine, and I made sure to give you as many tools as possible to make sure you can do this the right way. Just like backups, data migration operations are inevitable in production, and are going to discuss them too.

10.1 Deployment

In this section we will discuss various topics relating to the deployment process of a RavenDB server. Unless you have RavenDB embedded in your application, you are going to need to deploy a RavenDB server, configure it and maintain it so your application has a database endpoint it can communicate with. Embedded RavenDB is easier to maintain, as it ships with your own application.

As we will see, there are two flavors of installing RavenDB on a Windows server. Once installation is done, there are several configurations you might want to look at changing.

We start by discussing the installation procedure, and talk about configurations later.

10.1.1 Installing a RavenDB server

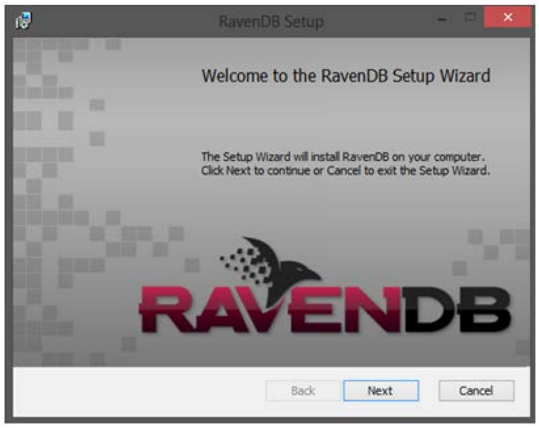
RavenDB can be installed on a Windows server under IIS, or set to be run as a Windows service. When installed as a service, RavenDB will create its own HTTP server and process all requests by its own. Hosted under IIS, HTTP handling is done all by IIS and requests are delegated to RavenDB for processing.

Installing as a Windows service is as simple as running the command `Raven.Server.exe /install` as an administrator. This exe file is the same one that you were using during development and throughout this book. Running this command will install it as a Windows service that automatically launches and listens of port 8080 (this can be changed via configurations, see later in this section).

By running RavenDB under IIS you get some additional benefits, like taking advantage of monitoring tools and HTTPS. Setting this up however can be more involved, depending on the IIS version you are running and how fluent you are with IIS – as some extra configurations may be needed.

Installing RavenDB as an IIS application is a bit more involved and depends on the version of IIS you are running, you should check RavenDB's official documentation for the guidance on IIS installation.

Installation using either approaches can be done using an installer software, downloadable from RavenDB's site.



www.dbooks.org

Installation via Chocolatey

Chocolatey is a package manager for your Windows machine. It allows installing and managing various software parts automatically, and is extremely useful when automating server creation and maintenance.

Unfortunately, at the time of this writing there is no RavenDB package that is officially released and supported by the team behind RavenDB. Hopefully by the time you read this, there will be one.

For now, an unofficial installation package can be found in the following link: https://chocolatey.org/packages/RavenDB

10.1.2 Database Authentication

Requests to a RavenDB server are made over HTTP, and by default the server doesn't check to see if they were made by an authenticated and authorized user. Those requests can be authenticated, however this feature requires a valid commercial license so may not work for you during testing before providing the license file to the server.

Authenticating requests can be done either using Windows authentication, or OAuth. By default, when authentication is enabled Windows authentication is used. This is also the case for many administrative endpoints that RavenDB supports.

With Windows authentication, by default all Windows users and groups have complete access to all databases. You can change this by writing a document with an ID `Raven/Authorization/WindowsSettings` to the database, with a structure similar to this:

```
{
      "RequiredGroups": [],
      "RequiredUsers": [
            {
                  "Name": "IIS AppPool\DefaultAppPool", #A
                  "Enabled": true,
                  "Databases": [
                        {
                              "Admin": false,     #B
                              "TenantId": "MyDB", #B
                              "ReadOnly": true    #B
                        }
                  ]
            }
      ]
}
```

#A – The username being allowed to access the database
#B – The permissions being granted – in this case a read-only access to the MyDB database

Authentication using the OAuth protocol is also supported, using API keys. This is a nice and easy way that allows one user that has write access to the database to enable other users access by writing documents to RavenDB which enable then access using a provided API key. It works like this:

1. An authenticated user writes a document prefixed with `Raven/ApiKeys/` to the `system` database on the server. That document specifies a secret, and a list of allowed resources:
   `store.DatabaseCommands`

```
    .Put(
          "Raven/ApiKeys/joey",
          null,
          RavenJObject.FromObject(new ApiKeyDefinition
          {
                  Name = "joey",
                  Secret = "ThisIsMySecret",
                  Enabled = true,
                  Databases = new List<ResourceAccess>
                          {
                                  new ResourceAccess {TenantId = "*"},
                                  new ResourceAccess {TenantId =
    Constants.SystemDatabase},
                          }
          }), new RavenJObject());
```

2. Now, Joey can access the databases he's been granted access to by providing the API key when connecting to the server:

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

3. To revoke Joey's access, all you need to do is delete the API key document.

Authorization and document level permissions

This section handled the question of authenticating users, and restricting access to server-level resources (databases). Restricting access to resources is often referred to as authorization, as opposed to authentication which just verifies the identity of the user.

Once a user is granted access to a database, he has full control over that database. He can create, edit and delete documents as he pleases.

Using the Authorization Bundle (see: http://ravendb.net/docs/article-page/3.0/csharp/server/bundles/authorization) you can add support for restricting access to documents on a database level – meaning, defining which users can touch which documents in every individual database.

While this sounds powerful and useful, it is infact an anti-pattern. If you need to do this it probably means you are allowing access to the RavenDB server from an external network, or the Web, and this is something that is highly discouraged.

Make sure you make permission decisions in your application, and not leak this responsibility over to the database engine.

www.dbooks.org

10.1.3 Configurations

RavenDB's configurations reside in the Raven.Server.exe.config file (or web.config if you are running under IIS or in Embedded mode). This is an XML configuration file, exactly like the one you are used to from many other .NET projects. It is going to look something like this:

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

There are many configurations options available, and all are listed in the official documentation on RavenDB's website. There are however several important configurations that are worth putting forward here as well, so you are aware of them:

- `Raven/DataDir` allows to specify the data folder in which RavenDB's data will be persisted.
- `Raven/IndexStoragePath` allows you to specify the folder in which indexes are stored and computed. Often times placing the indices on a separate hard-disk, especially if it's an SSD one, can provide a significant performance boosts.
- `Raven/RunInMemory` when set to true, will force RavenDB to run completely in memory, making the entire data volatile.
- `Raven/StorageTypeName` RavenDB supports several storage engines. While the default is using Microsoft's ESENT, this is where you can switch to alternatives.
- `Raven/HostName` and `Raven/Port` specify the hostname and port to which RavenDB binds itself to listen to incoming HTTP requests.

As I mentioned, there are many more useful configuration options for paths, memory settings, quotas, bundles and more. You can see them all here: http://ravendb.net/docs/article-page/3.0/csharp/server/configuration/configuration-options.

Encryption and compression support

A very frequently asked question is about encryption and/or compression support in RavenDB. People are usually interested in making sure their data is put securely on disk, and also to have it compressed – both on store and indexes.

The answer is short and simple: yes it's is all supported, but through bundles. You will have to install and configure them yourself, but they are maintained and supported by RavenDB's team.

10.2 Backup strategies

Database backups are probably the most important thing you should do in a system. Unfortunately, many database deployments don't have any backup strategy defined. Most system backup strategies are created and deployed after a catastrophic system failure.

Instead of getting there, one minute after it is too late, you should prioritize having a backup strategy in place before even going to production.

Your responsibility for creating a good backup doesn't end in creating one. You should also a clear and defined way for restoring full-service from backup – both of the application and the database backup. Companies like Netflix are known for doing this on a weekly basis to minimize the risk and the time it takes them to launch to recover from backup. While this does require certain resources, this is definitely a good practice to have. Either way, make sure the way to recover from a backup is clear to everybody in the team, and occasionally test and verify the backup files.

In this section we will discuss the various backup strategies that are available, and how to restore from them.

10.2.1 Enterprise backup solutions

Many Windows-based systems rely on Volume Shadow Copy Service (VSS) for creating backups. This is a Windows service which creates snapshots of data on disk at the block level of the file system. Those "shadow copies" can then be used for restoring the disk to a previous state when needed.

RavenDB supports VSS backups, all you need to do is point your backup tool at RavenDB's data folders. Like we mentioned in the previous section, RavenDB's data directory can be configured via the `Raven/DataDir` setting. If your indexes reside in a different folder (see the Configuration section above), you may want to look at including them as well, to avoid long wait for indexing when you restore from backup.

Since this backup strategy is done on the operating system level, the database operation left undisturbed. Setting up a VSS backup and restoring from it when needed is out of the scope of this book.

10.2.2 The Raven.Backup tool

RavenDB can create a snapshot of your database and store it on a folder local to the database server as backup. During this backup procedure the database remains completely operational,

www.dbooks.org

although writes that were made to the database after the backup process has started will not make it to that backup.

Executing a backup process can be done using code, or via the REST API. Either way, the caller has to have elevated permissions, and only one backup operation may run at any given time for a database.

If running in a client/server mode, you can use Raven.Backup.exe like shown below to perform manual or scheduled backups, or access the backup endpoint using HTTP directly. For Embedded RavenDB instances, using the backup tool externally is not supported, but you can create a backup by calling `DocumentDatabase.Maintenance.StartBackup()`. When running from IIS, make sure to enable Windows Authentication for RavenDB's IIS application.

Running the utility manually is done by executing the following command:

```
Raven.Backup.exe --url=http://localhost:8080/ --database=MyDB --dest=C:\Temp\Foo -
      -nowait –readkey.
```

Restoring from backups can be done by running the following command, note how the Raven.Server.exe executable is being used for this, and not the backup tool:

```
Raven.Server.exe --restore-source=C:\Temp\Foo --restore-
      database=http://localhost:8080/
```

It is important to note that using this method restoring from backup is only supported on the same OS used to create the backup, or a newer version of it.

10.2.3 Periodic exports

One of the problems with manual backups is that it is a manual process, which has overhead and only happens if the one responsible for triggering this doesn't forget to do that. Using the backup tool we discussed in the last subsection also doesn't protect from cases where the server itself disappears, which may happen on cloud environments, since the backup is still kept on the database server itself.

Instead of automating calling the backup process itself, RavenDB offers the Periodic Export feature through a bundle. Using this feature, you can both automate the backup, and have it uploaded for you to a cloud-storage of your choosing.

To use the periodic export feature, the bundle needs to be enabled when creating the database using the management studio, or by setting the `Raven/ActiveBundles` configuration setting to contain the value `PeriodicExport`.

Once the bundle was initialized, you can use the Management Studio for setting up the export destination. The exports can be saved either to the local file system, on Amazon cloud services (AWS S3 and Glacier) or on Microsoft Azure storage.

Restoring from this backup is done using the Smuggler tool. We will discuss this tool now.

10.2.4 The Smuggler tool

Smuggler is another tool that is bundled with RavenDB, with the sole purpose of supporting an import and export functionality for RavenDB.

As opposed to backups, which are optimized for quickly restoring from as the storage itself is snapshotted and copied over, Smuggler's output is the actual documents. Restoring to a live database from a Smuggler output may take a long time, since it effectively means adding and indexing all those documents from scratch.

The main use case for smuggler is transferring databases from, or creating backups in a readable form that is not RavenDB specific, since the output is just a bunch of JSON documents in block compression.

You can find the Smuggler tool under the /smuggler folder in the distribution download, or under the /tools folder if you got RavenDB via nuget.

To export your data from a database using Smuggler, simply run the following command:

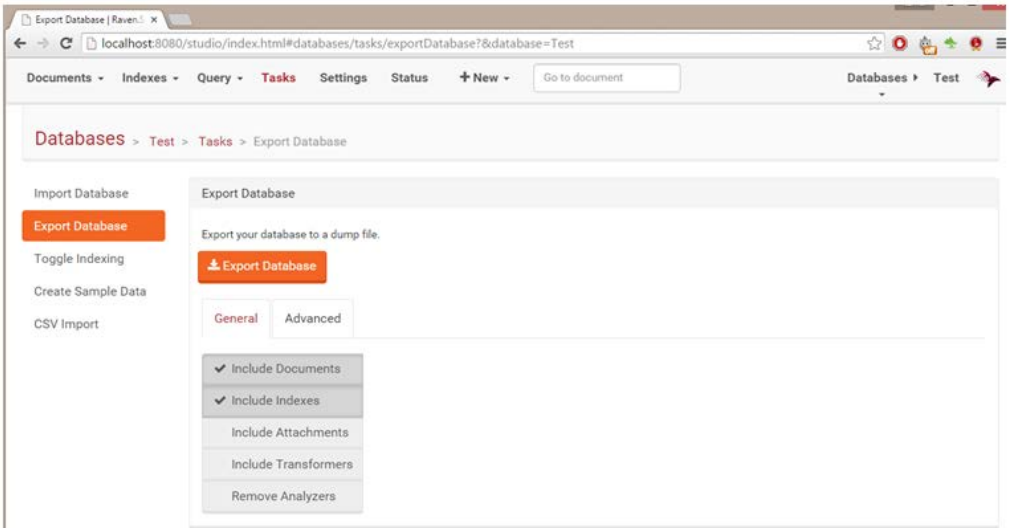
```
Raven.Smuggler out http://localhost:8080/ dump.ravendump --database=Northwind
```

Importing data back from a Smuggler dump file to a database is also a one-liner, just make sure you create the database beforehand as it won't do it for you:

```
Raven.Smuggler in http://localhost:8080 dump.ravendump --database=NewNorthwind
```

Smuggler also supports incremental experts and imports, use the `--incremental` flag with the tool, but Smuggler should auto-detect that automatically for you.

The Smuggler output can also be downloaded directly from the Management Studio, by asking it to Export your database from the Tasks tab.



10.3 Data migrations

While RavenDB does not require a schema to be declared up-front, it is still not entirely schema-free. Like many other similar databases, the schema is implicitly declared rather than

www.dbooks.org

explicitly. This essentially means RavenDB doesn't require a schema to be declared up-front, but once it accepted documents for storage they are implicitly defining a schema; therefore if their structure changes, some type of data migration process still needs to take place.

Since this book mainly focuses on using RavenDB from the .NET client API, a schema in our context is in-fact the structure of the POCOs in our model. The structure of those C# classes will translate to JSON documents with certain property names, with certain types (string, date/time, numeric, Boolean).

It is when the schema (read: those model POCOs in your project) changes that some form of data migration process may be required. We divide this into several scenarios, as follows:

Unobtrusive Changes

The simplest case is when a new document property has been added to the POCO – there is nothing special you need to do. This document property will be added gradually to all documents as they are being read by the client API and then saved back to the database. The indexing mechanism will silently ignore missing document properties even if there are indexes which are referencing them explicitly.

This is the same for the case in which a document property was removed and data contained in it doesn't need to be preserved. Those fields will gradually be removed from all documents over time and the indexes can be changed accordingly without the fear of breaking something.

Most changes to an evolving model are like that, requiring new data to be added to a class or a field removed.

Intrusive Changes

Often times you will need to have a document property renamed, or have its type changed. For example, when `ProductName` needs to become `ProductTitle` for some reason; or when `Price` needs to become `RetailPrice` to avoid confusion with `DiscountedPrice`.

In an ideal world, those changes will be caught during development, when it's safe to just nuke everything and start over again with a fresh database. Obviously this can and will happen after deployment to production, when you have a live database to maintain and can't just clear it and start fresh.

The challenges with doing such updates to a live system are big. You want to make sure nothing breaks and no data gets corrupted, while at the same time minimizing system down time.

Luckily, you already have the tools to deal with such changes. This can be handled in one of two ways:

1. Using client-side listeners to translate old-style properties to new ones whenever a document is loaded from the server.
   Using this approach you can ensure smooth transition between model versions, at the cost of some added code (and tests!). You will need to make sure all clients that use your model classes get updated at once so all clients can deal with both new and old versions at the same time.

2. Proactively making this change, using the patching and scripted patching APIs we discussed in the previous chapter. This allows you to contain the change and potentially avoids writing extra code over time. It is better to have zero listeners than maintain many of them over time. However, this will probably mean some down-time – full or partial – depending on the scenario.

MODEL CHANGES

A more intrusive type of change is an actual change to the model – where instead of just changing a property or two, you are also changing the entire class by renaming it or removing it completely in favor of a different representation of the data. And you want to make sure you seamlessly transfer all data to the new form.

For some simple cases this can still be accomplished using the same techniques we just described. As an example, if this is just a class name change, or a class name change done along with a property migration, the same methods as before still apply.

It is when there is a significant change to the model when it makes more sense to have an offline migration process. You can then use the result streaming API we discussed in chapter 9 to iterate through all documents and transform them manually, using a dedicated process. It is easy enough to write one as a console application to be used for a one time migration, after testing and dry runs of course.

You can also have this migration process duplicate the data, so the old model still exists side by side with the new one after the migration, and then make the switch when you are ready. Obviously, you will have to take into account changes that happened in between – or you can make your system read-only during that timeframe.

10.4 Summary

www.dbooks.org