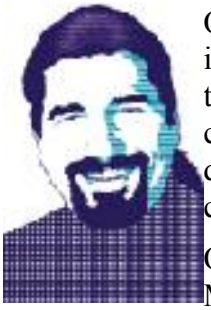
The Working Programmer

Going NoSQL with MongoDB

Ted Neward

Download the Code Sample



Over the past decade or so, since the announcement of the Microsoft .NET Framework in 2000 and its first release in 2002, .NET developers have struggled to keep up with all the new things Microsoft has thrown at them. And as if that wasn't enough, "the community"—meaning both developers who use .NET on a daily basis and those who don't—has gone off and created a few more things to fill in holes that Microsoft doesn't cover—or just to create chaos and confusion (you pick).

One of those "new" things to emerge from the community from outside of the Microsoft *aegis* is the NoSQL movement, a group of developers who openly challenge the idea that all data is/will/must be stored in a relational database system of some form. Tables, rows, columns, primary keys, foreign key constraints, and arguments over nulls and whether a primary key should be a natural or unnatural one … is nothing sacred?

In this article and its successors, I'll examine one of the principal tools advocated by those in the NoSQL movement: MongoDB, whose name comes from "hu*mongo*us," according to the MongoDB Web site (and no, I'm not making that up). Most everything MongoDB-ish will be covered: installing, exploring and working with it from the .NET Framework, including the LINQ support offered; using it from other environments (desktop apps and Web apps and services); and how to set it up so the production Windows admins don't burn you in effigy.

Problem (or, Why Do I Care, Again?)

Before getting too deep into the details of MongoDB, it's fair to ask why any .NET Framework developers should sacrifice the next half-hour or so of their lives reading this article and following along on their laptops. After all, SQL Server comes in a free and redistributable Express edition that provides a lighter-weight data storage option than the traditional enterprise- or datacenter-bound relational database, and there are certainly plenty of tools and libraries available to provide easier access to it, including Microsoft's own LINQ and Entity Framework.

The problem is that the strength of the relational model—the relational model itself—is also its greatest weakness. Most developers, whether .NET, Java or something else entirely, can—after only a few years' experience—describe in painful detail how everything doesn't fit nicely into a tables/rows/columns "square" model. Trying to model hierarchical data can drive even the most experienced developer completely bonkers, so much so that Joe Celko wrote a book—"SQL for Smarties, Third Edition," (Morgan-Kaufmann, 2005)—entirely about the concept of modeling hierarchical data in a relational model. And if you add to this the basic "given" that relational databases assume an inflexible structure to the data—the database schema—trying to support ad hoc "additionals" to the data becomes awkward. (Quick, show of hands: How many of you out there work with databases that have a Notes column, or even better, Note1, Note2, Note3 …?)

Nobody within the NoSQL movement is going to suggest that the relational model doesn't have its

strengths or that the relational database is going to go away, but a basic fact of developer life in the past two decades is that developers have frequently stored data in relational databases that isn't inherently (or sometimes even remotely) relational in nature.

The document-oriented database stores "documents" (tightly knit collections of data that are generally not connected to other data elements in the system) instead of "relations." For example, blog entries in a blog system are entirely unconnected to one another, and even when one does reference another, most often the connection is through a hyperlink that is intended to be dereferenced by the user's browser, not internally. Comments on that blog entry are entirely scoped to that blog entry, and rarely do users ever want to see the aggregation of all comments, regardless of the entry they comment on.

Moreover, document-oriented databases tend to excel in high-performance or high-concurrency environments; MongoDB is particularly geared toward high performance, whereas a close cousin of it, CouchDB, aims more at high-concurrency scenarios. Both forgo any sort of multi-object transaction support, meaning that although they support concurrent modification of a single object in a database, any attempt to modify more than one at a time leaves a small window of time where those modifications can be seen "in passing." Documents are updated atomically, but there's no concept of a transaction that spans multiple-document updates. This doesn't mean that MongoDB doesn't have any durability—it just means that the MongoDB instance isn't going to survive a power failure as well as a SQL Server instance does. Systems requiring full atomicity, consistency, isolation and durability (ACID) semantics are better off with traditional relational database systems, so mission-critical data most likely won't be seeing the inside of a MongoDB instance any time soon, except perhaps as replicated or cached data living on a Web server.

In general, MongoDB will work well for applications and components that need to store data that can be accessed quickly and is used often. Web site analytics, user preferences and settings—and any sort of system in which the data isn't fully structured or needs to be structurally flexible—are natural candidates for MongoDB. This doesn't mean that MongoDB isn't fully prepared to be a primary data store for operational data; it just means that MongoDB works well in areas that the traditional RDBMS doesn't, as well as a number of areas that could be served by either.

Getting Started

As mentioned earlier, MongoDB is an open-source software package easily downloaded from the MongoDB Web site, mongodb.com. Opening the Web site in a browser should be sufficient to find the links to the Windows downloadable binary bundle; look at the right-hand side of the page for the Downloads link. Or, if you prefer direct links, use mongodb.org/display/DOCS/Downloads. As of this writing, the stable version is the 1.2.4 release. It's nothing more than a .zip file bundle, so installing it is, comparatively speaking, ridiculously easy: just unzip the contents anywhere desired.

Seriously. That's it.

The .zip file explodes into three directories: bin, include and lib. The only directory of interest is bin, which contains eight executables. No other binary (or runtime) dependencies are necessary, and in fact, only two of those executables are of interest at the moment. These are mongod.exe, the MongoDB database process itself, and mongo.exe, the command-line shell client, which is typically used in the same manner as the old isql.exe SQL Server command-line shell client—to make sure things are installed correctly and working; browse the data directly; and perform administrative tasks.

Verifying that everything installed correctly is as easy as firing up mongod from a command-line client. By default, MongoDB wants to store data in the default file system path, c:\data\db, but this is configurable with a text file passed by name on the command line via --config. Assuming a

subdirectory named db exists wherever mongod will be launched, verifying that everything is kosher is as easy as what you see in **Figure 1**.

Figure 1 **Firing up Mongod.exe to Verify Successful Installation**

If the directory doesn't exist, MongoDB will not create it. Note that on my Windows 7 box, when MongoDB is launched, the usual "This application wants to open a port" dialog box pops up. Make sure the port (27017 by default) is accessible, or connecting to it will be … awkward, at best. (More on this in a subsequent article, when I discuss putting MongoDB into a production environment.)

Once the server is running, connecting to it with the shell is just as trivial—the mongo.exe application launches a command-line environment that allows direct interaction with the server, as shown in **Figure 2**.

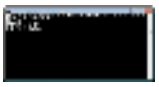


Figure 2 **Mongo.exe Launches a Command-Line Environment that Allows Direct Interaction with the Server**

By default, the shell connects to the "test" database. Because the goal here is just to verify that everything is working, test is fine. Of course, from here it's fairly easy to create some sample data to work with MongoDB, such as a quick object that describes a person. It's a quick glimpse into how MongoDB views data to boot, as we see in **Figure 3**.

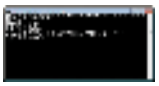


Figure 3 **Creating Sample Data**

Essentially, MongoDB uses JavaScript Object Notation (JSON) as its data notation, which explains both its flexibility and the manner in which clients will interact with it. Internally, MongoDB stores things in BSON, a binary superset of JSON, for easier storage and indexing. JSON remains MongoDB's preferred input/output format, however, and is usually the documented format used across the MongoDB Web site and wiki. If you're not familiar with JSON, it's a good idea to brush up on it before getting heavily into MongoDB. Meanwhile, just for grins, peer into the directory in which mongod is storing data and you'll see that a couple of "test"-named files have shown up.

Enough playing—time to write some code. Quitting the shell is as easy as typing "exit," and shutting the server down requires only a Ctrl+C in the window or closing it; the server captures the close signal and shuts everything down properly before exiting the process.

MongoDB's server (and the shell, though it's not as much of an issue) is written as a native C++ application—remember those?—so accessing it requires some kind of .NET Framework driver that knows how to connect over the open socket to feed it commands and data. The MongoDB distribution doesn't have a .NET Framework driver bundled with it, but fortunately the community has provided one, where "the community" in this case is a developer by the name of Sam Corder, who has built a .NET Framework driver and LINQ support for accessing MongoDB. His work is available in both source and binary form, from github.com/samus/mongodb-csharp. Download either the binaries on that page (look in the upper-right corner) or the sources and build it. Either way, the result is two assemblies: MongoDB.Driver.dll and MongoDB.Linq.dll. A quick Add Reference to the References node of the project, and the .NET Framework is ready to rock.

Writing Code

Fundamentally, opening a connection to a running MongoDB server is not much different from opening a connection to any other database, as shown in **Figure 4**.

Figure 4 **Opening a Connection to a MongoDB Server**

```
using System;
using MongoDB.Driver;

namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
      Mongo db = new Mongo();
      db.Connect(); //Connect to localhost on the default port
      db.Disconnect();
    }
  }
}
```

Discovering the object created earlier isn't hard, just … different … from what .NET Framework developers have used before (see **Figure 5**).

Figure 5 **Discovering a Created Mongo Object**

```
using System;
using MongoDB.Driver;

namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
      Mongo db = new Mongo();
      db.Connect(); //Connect to localhost on the default port.
      Database test = db.getDB("test");
      IMongoCollection things = test.GetCollection("things");
      Document queryDoc = new Document();
      queryDoc.Append("lastname", "Neward");
      Document resultDoc = things.FindOne(queryDoc);
      Console.WriteLine(resultDoc);
      db.Disconnect();
    }
  }
}
```

If this looks a bit overwhelming, relax—it's written out "the long way" because MongoDB stores things differently than traditional databases.

For starters, remember that the data inserted earlier had three fields on it—firstname, lastname and age, and any of these are elements by which the data can be retrieved. But more importantly, the line that stored them, tossed off rather cavalierly, was "test.things.save()"—which implies that the data is being stored in something called "things." In MongoDB terminology, "things" is a collection, and implicitly

all data is stored in a collection. Collections in turn hold documents, which hold key/value pairs where the values can be additional collections. In this case, “things” is a collection stored inside of a database, which as mentioned earlier is the test database.

As a result, fetching the data means connecting first to the MongoDB server, then to the test database, then finding the collection “things.” This is what the first four lines in **Figure 5** do—create a Mongo object that represents the connection, connects to the server, connects to the test database and then obtains the “things” collection.

Once the collection is returned, the code can issue a query to find a single document via the FindOne call. But as with all databases, the client doesn’t want to fetch every document in the collection and then find the one it’s interested in—somehow, the query needs to be constrained. In MongoDB, this is done by creating a Document that contains the fields and the data to search for in those fields, a concept known as query by example, or QBE for short. Because the goal is to find the document containing a lastname field whose value is set to “Neward,” a Document containing one lastname field and its value is created and passed in as the parameter to FindOne. If the query is successful, it returns another Document containing all the data in question (plus one more field); otherwise it returns null.

By the way, the short version of this description can be as terse as:

```
Document anotherResult =
         db["test"]["things"].FindOne(
           new Document().Append("lastname", "Neward"));
       Console.WriteLine(anotherResult);
```

When run, not only do the original values sent in show up, but a new one appears as well, an _id field that contains an ObjectId object. This is the unique identifier for the object, and it was silently inserted by the database when the new data was stored. Any attempt to modify this object must preserve that field or the database will assume it’s a new object being sent in. Typically, this is done by modifying the Document that was returned by the query:

```
anotherResult["age"] = 39;
       things.Update(resultDoc);
       Console.WriteLine(
         db["test"]["things"].FindOne(
           new Document().Append("lastname", "Neward")));
```

However, it’s always possible to create a new Document instance and manually fill out the _id field to match the ObjectId, if that makes more sense:

```
Document ted = new Document();
       ted["_id"] = new MongoDB.Driver.Oid("4b61494aff75000000002e77");
       ted["firstname"] = "Ted";
       ted["lastname"] = "Neward";
       ted["age"] = 40;
       things.Update(ted);
       Console.WriteLine(
         db["test"]["things"].FindOne(
           new Document().Append("lastname", "Neward")));
```

Of course, if the _id is already known, that can be used as the query criteria, as well.

Notice that the Document is effectively untyped—almost anything can be stored in a field by any name, including some core .NET Framework value types, such as DateTime. Technically, as mentioned, MongoDB stores BSON data, which includes some extensions to traditional JSON types

(string, integer, Boolean, double and null—though nulls are only allowed on objects, not in collections) such as the aforementioned ObjectId, binary data, regular expressions and embedded JavaScript code. For the moment, we'll leave the latter two alone—the fact that BSON can store binary data means that anything that can be reduced to a byte array can be stored, which effectively means that MongoDB can store anything, though it might not be able to query into that binary blob.

Not Dead (or Done) Yet!

There's much more to discuss about MongoDB, including LINQ support; doing more complex server-side queries that exceed the simple QBE-style query capabilities shown so far; and getting MongoDB to live happily in a production server farm. But for now, this article and careful examination of IntelliSense should be enough to get the working programmer started.

By the way, if there's a particular topic you'd like to see explored, don't hesitate to drop me a note. In a very real way, it's your column, after all. Happy coding!

Ted Neward *is a principal with Neward & Associates, an independent firm specializing in enterprise .NET Framework and Java platform systems. He has written more than 100 articles, is a C# MVP, INETA speaker and has authored and coauthored a dozen books, including the forthcoming "Professional F# 2.0" (Wrox). He consults and mentors regularly—reach him at ted@tedneward.com or read his blog at blogs.tedneward.com.*

Thanks to the following technical experts for reviewing this article: Kyle Banker and Sam Corder