02/01/2019 • 11 minutes to read

In this article

July 2016

Volume 31 Number 7

[The Working Programmer]

How To Be MEAN: Let's Be DEAN

By Ted Neward | July 2016

Welcome back again, MEANers. Or, rather, for this month, "DEAN"ers.

One of the things that makes conversations and architecture around the MEAN stack so compelling is that the MEAN stack is intrinsically flexible—you can replace components of the stack with other, more-or-less-equivalent parts and create a new stack that can address corporate/business needs without surrendering the essence of the architecture. As a demonstration of that concept, in this column I'm going to experiment with replacing MongoDB with Microsoft Azure DocumentDB (hence, the "D" in place of the "M").

MongoDB vs. DocumentDB

For those who haven't spent much time getting familiar with DocumentDB, I highly recommend looking at some of the existing resources available, including Julie Lerman's June 2015 Data Points column (msdn.com/magazine/mt147238). Lerman gives a great overview of DocumentDB, including one of its most interesting features, serverside code execution (yes, as in stored procedures, written in JavaScript). If you've never glanced at DocumentDB, or need a quick intro, do a quick read before continuing.

On the surface, MongoDB and DocumentDB are similar creatures. They're both document-oriented databases, using JSON as the principal document representation format. They each extend JSON to incorporate some additional data types. They each think about data in terms of collections of documents, and use that as the principal aggregation scheme (instead of tables). They're intrinsically "non-relational" in that the database doesn't do any verification of document identifiers used to link a document to another document. For that matter, they're also both "schema-free," in that a document's contents are entirely in the hands of the developer, and the same document structure isn't enforced across all the elements of a collection.

Therefore, this should be a simple swap.

However, aside from some deeper differences you'll discover in a bit, right away there's one principal difference: While MongoDB can be downloaded and run locally off a laptop, DocumentDB is only available through an Azure subscription, so the first step toward DEAN is the simple task of creating a DocumentDB instance on an Azure account. This is described in several other places in detail (including in the DocumentDB documentation online), so I won't repeat the process here except to summarize. Go to the Azure Management Portal, create a new Azure DocumentDb resource, give it a unique name (I called mine dean-db) and punch the big blue Create button. The portal will churn for a few minutes and Azure will create your DocumentDB instance in the cloud.

Before you're done with the portal, there are a couple pieces of information you're going to need later in the code. So let's take a second and jot them down now. In particular, you'll need the URL to connect to, and the authorization key to ensure it's you. In the Azure Management Portal, this is called a PRIMARY KEY (there's a second one called SECONDARY KEY), and as of this writing, it's displayed in the All Settings tab, under Keys. These keys are shared secrets, so make sure not to give it out to anybody—in particular, if the code is going into a public source code repository (a la GitHub), make sure the key isn't part of the checked-in source code. (Conventionally, Node.js applications configure this as an environment variable and use the application entry point code to pick this up as part of its startup; you've done similar to this in previous iterations of this code base, so this shouldn't be a surprise.) If a key is compromised in any way (as in, you discover that the key somehow made it out into the public), make sure to regenerate the keys and replace them. In fact, it's probably a good idea to cycle them every 30 or 60 days just on principle.

This means that the config.js in the existing code base now looks like **Figure 1**.

Figure 1 Config.js in the Existing Code Base

JavaScript Copy

```javascript
module.exports = {
  mongoServer : "localhost",
  mongoPort : "27017",
  docdbServer : "https://dean-db.documents.azure.com:443/",
  docdbKey :
    "gzk030R7xC9629Cm1OAUirYg8n2sqLF3O0xtrVl8JT
      ANNM1zV1KLl4VEShJyB70jEtdmwnUUc4nRYyHhxsjQjQ=="
      };
if (process.env["ENV"] === "prod") {
  module.exports.mongoServer = "ds054308.mongolab.com";
  module.exports.mongoPort = "54308";
  module.exports.docdbServer = process.env["DOCDB_HOST"];
  module.exports.docdbKey = process.env["DOCDB_KEY"];
}
```

(By the way, I've already cycled the keys on my repository, so displaying the key here is just for show, to give you an idea of what it'll look like.)

Hello, from Node.js

The next step is predictable—you need the Node.js module for accessing the DocumentDB instance, and you do so using "npm install --save documentdb." Naturally, you'll need to "require" it in the app.js code, like so:

JavaScript Copy

```javascript
var express = require('express'),
  bodyParser = require('body-parser'),
  debug = require('debug')('app'),
  edge = require('edge'),
  documentdb = require('documentdb'),
  ...;
// Go get our configuration settings
var config = require('./config.js');
debug("Mongo is available at ",config.mongoServer,":",config.mongoPort);
debug("DocDB is available at ",config.docdbServer);
```

From here, opening a connection is simply a matter of constructing a DocumentClient object using the server and authorization key, like so:

JavaScript Copy

```javascript
// Connect to DocumentDB
var docDB = documentdb.DocumentClient(config.docdbServer, {
  masterKey: config.docdbKey
});
```

Now, of course, the bigger question is: What do you do with it, once you have it open?

Using DocumentDB

Like MongoDB, DocumentDB uses a JSON-based format for a schema, and collections of documents, so storing data there (such as "presentations") involves the same kinds of operations as you've seen with MongoDB in the past. However, the API is a little different, probably owing to some cultural differences between how Microsoft likes to design things and how things sort of evolve out of the open source world.

(By the way, the best DocumentDB/Node.js reference I've found thus far is the set of samples on the Azure Documentation page [bit.ly/1TkqXaP]. It's a collection of direct links to Microsoft-authored sample projects stored on GitHub, as well as links to the documentation generated out of the DocumentDB Node.js API source code, and in lieu of anything more official, I've been using it as my go-to reference for DocumentDB APIs.)

For starters, once the database client object is constructed, you need to connect to the database in question using a databaseId; this is the actual database, in much the same way that a MongoDB server can host multiple databases. However, unlike MongoDB, with DocumentDB this database must be created ahead of time. This can be done either programmatically (which is great for demos and DevOps scenarios) via the createDatabase API call, or via the Azure Management Portal (which is probably going to be the more common approach, given that this is usually a one-off operation) using the Add Database tab under the DocumentDB resource page. For simplicity's sake, the code here assumes a database of conferencedb already exists, presumably created in the portal earlier. Assuming the database exists, Node.js can connect to it by calling queryDatabases, which brings us to the next big difference in approach between MongoDB and DocumentDB, as shown in **Figure 2**.

Figure 2 Querying DocumentDB for a List of Databases

JavaScript Copy

```javascript
docClient.queryDatabases({
  query: 'SELECT * FROM root r WHERE r.id = @id',
  parameters: [
    {
      name: '@id',
      value: 'conferencedb'
    }
  ]}).toArray(function (err, results) {
  if (err) {
    handleError(err);
  }
  if (results.length === 0) {
      // No error occured, but there were no results returned
      // indicating no database exists matching the query
      // so, explictly return null
      debug("No results found");
    } else {
      // Found a database, so return it
      debug('Found a database:', results[0]);
      var docDB = results[0];
    }
  });
```

First, notice how the DocumentDB API uses an explicit "query specification," even for fetching the list of databases, complete with parameters. This is a marked difference from Mongo's "query by example" approach. Granted, you clearly don't have to use this —you could just do inline population of the arguments via string concatenation—but as any developer who's ever been the victim of a SQL injection attack will testify, doing parameterized queries like this is much, much safer.

The other major difference is, of course, the query language—hello, SQL, my old friend. To be fair, it's not exactly SQL, because you don't have tables, columns and such, but Microsoft has gone to great lengths to adapt the familiar query language over to the document-oriented world. Whether this is a "bug" or "feature" is likely to depend on the developer's love-or-hate relationship with the relational database world, but by these (and other) decisions, Microsoft is clearly marking DocumentDB as the compromise choice between SQL Server and MongoDB (or other document databases) and that's not a bad place to be.

Once a database is found, you need to obtain the collection, again by identifier. Like databases, collections are "formal" things, meaning they need to be either explicitly created using the createCollection API call or via the Azure Management Portal. Given how DocumentDB thinks of collections as tables, this is not surprising. Again, the simplest thing to do is create one via the portal, by (as of this writing) clicking on the database name in the databases tile in the DocumentDB resource tile. This will bring up a new tab, in which you can add a new collection, presentations, and then find it via that identifier, like in **Figure 3**.

Figure 3 Found a Database? Now Find the Collections

JavaScript Copy

```
// We found a database
debug('Found a database:', results[0]);
docDB = results[0];
debug('Looking for collections:');
docClient.readCollections('dbs/conferencedb').
toArray(function(err, colls) {
if (err) {
  debug(err);
  }
  else {
    if (colls.length === 0) {
      debug("No collections found");
    }
    else {
      for (var c in colls) {
        debug("Found collection",colls[c]);
        if (colls[c].id === 'presentations')
          presentationColl = colls[c];
        }
      }
    }
  });
```

Take note of the first parameter to the readCollections API call; if it looks like it's specifying some kind of URI/URL-like path to the database in question, it should. Each call to the Node.js DocumentDB API uses these kinds of REST-ish identifiers to make it easy to drill directly to the collection (and, ultimately, document) desired without having to navigate a complex hierarchy.

(One important note that's very different from MongoDB: Azure charges for its services on a per-collection basis, so where a MongoDB user will think about collections from a pure modeling standpoint, with DocumentDB, the decision to create a new collection will have a direct impact on the monthly fees you end up paying.)

Finally, to find any particular documents in that collection, you use the queryDocuments API. Again, you pass in the collection identifier (which, because you know the collection you're going after, you can just embed directly in the code), and take the results to hand it directly back as the JSON body, just like I did a few columns back for MongoDB, as shown in **Figure 4**.

Figure 4 List All the Documents in a Collection

JavaScript Copy

```javascript
var getAllPresentations = function(req, res) {
  debug("Getting all presentations from DocumentDB:");
  docClient.queryDocuments("dbs/conferencedb/colls/presentations",
  {
    query: "SELECT * FROM presentations p"
  }).toArray(function (err, results) {
    if (err) res.status(500).jsonp(err);
    else res.status(200).jsonp(results);
  });
};
// ...
app.get('/presentations', getAllPresentations);
```

Naturally, if you want to restrict the presentations to a speaker, that would be a query parameter as part of the route configuration and would become a parameter in the query specification, and so on.

Wrapping Up

When I began this article, I cited all the ways that MongoDB and DocumentDB are similar; however, by now you can start to sense that there are actually a number of differences between the two, by way of philosophy behind their construction, if nothing else. Microsoft is clearly looking to put some "enterprise" into the document-oriented database world with DocumentDB, and given its customer base, that's a solid move. However, one thing that's clearly lacking (for now) is any kind of "object-ish" wrapper around the DocumentDB API, akin to what Mongoose provides to the Node.js MongoDB API. This isn't a complicated thing for developers to build on their own, but it does represent more code that said developers would need to write and maintain over time. That is at least until the open source community either ports Mongoose, adapts it to use either MongoDB or DocumentDB (not likely), or until someone builds something similar to it but in a more specialized way than which DocumentDB approaches life.

More important, however, is to realize that the acronym "MEAN" is just that—an acronym. DocumentDB represents only one of a dozen or so kinds of persistence tools that could be the back end of an "*EAN"-ish stack, including good ol' SQL Server. Where "MEAN" doesn't fit with your corporate standards, operations staff or business goals, feel free to replace the offending piece with something that's friendlier. At the end of the day, following an architecture to the letter (literally) that creates problems for you as a developer and/or to the organization as a whole is just silly. I'm out of space for now … happy coding!

Ted Neward *is a Seattle-based polytechnology consultant, speaker and mentor. He has written more than 100 articles, is an F# MVP and has authored and coauthored a dozen books. Reach him at ted@tedneward.com if you're interested in having him come work with your team, or read his blog at blogs.tedneward.com .*

Thanks to the following technical expert for reviewing this article: Shawn Wildermuth

---

Discuss this article in the MSDN Magazine forum