Storing (small) Images in MongoDB

24 MARCH 2015

The question of *if and how to store images in MongoDB* keeps popping up on StackOverflow and newsgroups. The two standard answers are *"Yeah, use GridFS..."* or *"No, don't store Images in the Database"*. But GridFS is cracking a nut with a sledgehammer and storing a large number of files in the file system doesn't (usually) come with replication, backup or versioning. Let's explore a simple alternative.

"Images belong in the file system"

Raw image data is neither structured nor searchable anyway, so flat files will do just as well as a database, with less overhead. That approach still requires the database to store the path to the image, or a convention on how database items and filenames are related. The advantage of this approach is that the static file can be delivered by the web server directly, bypassing your server-side code, database connections and what not. But we'll get back to that.

However, your typical IaaS-Server usually doesn't come with a pre-configured distributed file system that supports **replication and centralized backup** - features that you *hopefully* have in place for your MongoDB already. Also, storing images in the file system isn't trivial at all, because different file systems respond very differently to folders with a very large number of files. We often need to introduce a convention-based directory structure such as `/a4/bf/a4bfafcd831cbf1325.jpg` to keep individual folders from growing too large.

> *Replication, backup, concurrency and limitations for folder sizes and file names aren't fun to deal with*

Thread Hogs?

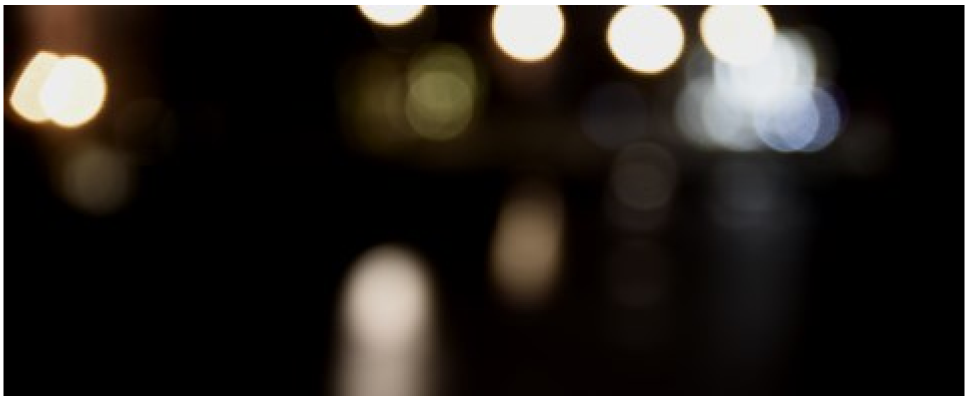
Another argument is you'll want to avoid running through your entire backend stack for the delivery of static files which hogs up your precious web server threads for operations that are I/O bound. Then again, since the advent of node.js, where a single thread through **asynchronous I/O** can serve a lot of clients concurrently with comparatively little resources, so this is no longer a much a concern.

Other platforms such as .NET 4.5 now come with aysnchronous I/O, a set of frameworks that implement it and syntactic sugar on top, too. The server model is still multi-threaded and very different from node.js unlike node.js, but I/O bound async operations don't force threads to idle anymore.

Sure, there's still overhead, and this would certainly not work for the likes of facebook, but compared to setting up and maintaining another technology for replication and backup, this is often a good deal. Also, if the images can be cached by proxies, good caching headers and ETags with a **CDN or reverse proxy** will probably **eliminate most of the requests anyway** which highlights the importance of better and easier management vs. low-overhead delivery.

"Use GridFS"

That's another standard answer, and it usually comes with little knowledge of GridFS. First of all, GridFS is *not* some kind of fancy MongoDB feature that was built into the core of Mongo to replace file systems. **GridFS is a mere data model convention** which makes it possible to supply some simple tools to help interact with data stored according to that convention. But let's look at this in detail. We'll store a slightly larger version of this image:



The larger version is called `large.jpg` and is about 380kB in size. Let's upload this to MongoDB using the `mongofiles` tool and look at the DB:

```
C:\MongoDB\bin>mongofiles put large.jpg
2015-03-06T01:34:26.841+0100    connected to: localhost
added file: large.jpg

C:\MongoDB\bin>mongo
connecting to: test
> show collections
fs.chunks
fs.files
system.indexes
```

Ok, so we now see two collections, called `fs.files` and `fs.chunks`. Again, these are just regular collections. Peeking inside, we find:

```
> db.fs.files.find().pretty();
{
        "_id" : ObjectId("54f8f61296fd73166000001"),
        "chunkSize" : 261120,
        "uploadDate" : ISODate("2015-03-
06T00:34:26.949Z"),
        "length" : 380744,
        "md5" : "02fb7cae77c617132f149dfd82ed91c1",
        "filename" : "large.jpg"
}
```

Nothing fancy. But where's the data? Now, if we do `db.fs.chunks.find().pretty()`, we basically see pages and pages of this:

```
fdoFFFAxaD0oopdQY3/GnyUUU0AidKB0/GiigA70dqKKnoAi/d
AtDdqKKBLqH+NN/ioopvcrsL2pKKKEIB3pB/SiigQL3oX71FFA
UUUUyQooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACi
ooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAoooo
CiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooA//ooAKF
```

Ok, that looks like the actual data. Let's exclude that field so we can see the rest of the document clearly:

```
> db.fs.chunks.find({}, {"data": 0}).pretty();
{
        "_id" : ObjectId("54f8f61296fd73166000002"),
        "files_id" :
ObjectId("54f8f61296fd73166000001"),
        "n" : 0
}
{
        "_id" : ObjectId("54f8f61296fd73166000003"),
        "files_id" :
ObjectId("54f8f61296fd73166000001"),
        "n" : 1
}
```

So we see we have two documents here, both pointing to the same `fs.files` through their `files_id`. The value of `n` specifies the ordering of the chunks so the data doesn't get messed up - that's the GridFS convention, nothing more.

Chunking

Besides working around MongoDB's document size limit of 16MB, the idea of chunking is to allow *streaming*, i.e. allow users to download (or stream) a file without having the server to store the whole thing in RAM at any point in time. Imagine the file is a 2.6GB HD video where that'd be really pointless. The default chunk size of GridFS is 256kB which is supposedly a good compromise of overhead (more queries to the database) and little memory use, but it can be configured.

But what does that mean for small images? At only 380kB, streaming hardly makes sense, and delivering our image will require **three round-trips to the database** instead of one: One to find the `fs.files` document, and two to get the chunks. Even if we increased the chunk size, we'd still need two round-trips before we can even *start* to deliver the file.

A Simple Alternative

As we have seen, there's nothing special about GridFS, so for small files, we can basically just merge the contents of the `fs.files` and `fs.chunks` collections, like so:

```
{
    "_id" : ObjectId("54f8f61296fd731660000001"),
    "data" : BinData(2, "AAFC342..."),
    "length" : 380744,
    "md5" : "02fb7cae77c617132f149dfd82ed91c1"
}
```

This eliminates two of the three round-trips to the database and increases the amount of required RAM only marginally, as long as images are reasonably small. This might not be the best approach for storing **large** images, but for your typical user thumbnail, it's certainly a viable alternative.

Pros

- Centralized: only one backup / replication strategy
- Can be used for images that need authorization / tracking / statistics
- Asynchronous I/O, e.g. via node.js or .NET 4.5 frees you from thread hogs

Cons

- Extra overhead - file systems *are* fast
- Might incur overhead because of the server stack