Introduction to Computer Science I
COMP 2406A – Winter 2020

Intro to MongoDB

Dave McKenney
david.mckenney@carleton.ca

Learning Outcomes

by the End of this Lecture, Students that have Completed the Reading Assignment and Review Questions should be Able to:

Understand what a database is

Use MongoDB and the MongoDB shell to perform CRUD operations on a document database

The Course So Far

Throughout the course, we have been improving our ability to design scalable, robust, dynamic web applications

We started with basic client-side DOM manipulation, considered simple HTTP servers, and have now used Express to organize a large amount of data

Database Definition

Database: at its core, the term database refers to a structured set of data

In a way, we have worked with databases already (cards tutorials, restaurants assignments)

Databases

Working with data stored in files, though, has a number of shortcomings…

Databases

1. Asynchronously reading files requires extra book-keeping (how to tell if all files are done?)
2. Asynchronously writing files needs proper management to avoid data corruption
3. Synchronous file operations are a terribly bad practice
4. If we wish to query the data, we have to write our own query parsing code

Databases

So our current file-based solutions are a bit like the simple HTTP server we first covered – we COULD do it this way, but there are tools to make life easier (and create better code)

Database Management Systems

Database Management System (DBMS): a DBMS is software that is used to interact with a database.

Often, the term database refers to both the DBMS and the database(s) the DBMS works with

Database Management Systems

DBMSs are designed from the ground up with the intention of storing, manipulating, and querying databases

In a way, Express is to web app development as DBMSs are to data storage and querying

Some things that DBMSs solve for us:

1. They handle concurrent access
2. They provide query languages
3. They automatically perform optimizations
4. They allow for scaling and redundancy

These lead to better overall app quality for us

There are two main classes of DBMS:

1. Relational databases – organize data into tables of rows/columns, typically use SQL (COMP 3005)
2. NoSQL databases – non-SQL or not only SQL, use something other than relational design (e.g., key/value, graph, document databases)

NoSQL databases have surged in popularity over the last ~15 years

Ultimately, there is no single best solution - the best choice depends on the data/queries/access patterns

Polyglot persistence deals with using multiple database types to optimize performance
(not a part of this course)

MongoDB

MongoDB is a document-based database

Mongo stores ‘collections’ of ‘documents’

Generally, a document represents a single entity or object within the system

MongoDB

Documents in Mongo are stored as JSON (actually, as BSON – binary JSON)

This is an idea we are already familiar with – instead of reading/writing objects to/from a file or HTTP connection, we will read/write them from a database

MongoDB

Has support for many programming languages (including Javascript and Node.js)

Is also cross-platform and used by many large groups (Google, Facebook, City of Chicago)

Supports distribution/sharding for scalability

And it’s free to use (community edition)

Getting Started with MongoDB

We will first look at MongoDB separate of any programming language

That is, we will use the Mongo shell to work with databases in a command-line environment

To get started, download and install the Community Edition of Mongo
https://docs.mongodb.com/manual/administration/install-community/

Getting Started with MongoDB

To start playing with MongoDB, you will need to run two separate programs: the MongoDB daemon and the MongoDB shell

Getting Started with MongoDB

To start the Mongo daemon:

1. Open a command line terminal
2. Navigate to the directory you want your database to be contained in (e.g., directory of your server)
3. Create a directory to store the database
4. Run: mongod --dbpath=dirName (dirName is the directory you created in #2)

To run the Mongo shell:

1. **Run the Mongo daemon**
2. **Open a second command line terminal**
3. **Run: mongo**

Later, we will use Node.js instead of the mongo shell, but using the shell will still be an important skill

If you are running the Mongo daemon remotely or using a non-default port, specify the address/port:

```
mongo --port 99999
mongodb://some.machine.com:28015
```

An important note: MongoDB does not have authentication enabled by default

So anybody that can connect to your machine can connect to your database

Ensure you enable authentication before making something publicly available

Getting Started with MongoDB

MongoDB uses a hierarchical structure to organize the data:

Database → Collection(s) → Document(s)

So you can have many databases, each of which can have many collections, each of which can have many documents

Getting Started with MongoDB

So we can have a database for our web app

Within the database, we can have a collection for each collection of items we have (products, users, cards, restaurants, etc.)

Within each collection, we can have documents to represent the specific resources (card, user, restaurant, etc.)

Example Database

We will work through some examples using a fabricated products dataset

We will have a database called ‘store’

The database will have a collection called ‘products’

Each ‘product’ document will have: name (string) price (number), stock (number), dimensions (object)

Example Database

We will walk through some examples of performing the common operations we have in our apps:

Creating documents

Reading and querying documents

Updating documents

Deleting documents

We will also look at some basic database management commands

Navigating Mongo Shell

To work with databases in Mongo shell:

See your current database: db

List all available databases: show dbs

Navigating Mongo Shell

To work with databases in Mongo shell:

To use a database: use *dbName*
(where dbName is the database name)

To remove a database: db.dropDatabase( )
(db in this case is the current database being 'used')

Note: you can 'use' a database that does not exist and Mongo will create it when you add a document

Example Database

Start by using a database called store

The first thing we will want to do is add some documents to the ‘products’ collection

In Mongo, you can add a single document at a time or add many documents at once…

Example Database

db.*collectionName*.insertOne({ name: “pants”, price: 50, dimensions: { x:10, y:30, z:2}, stock: 10 })

Inserts the given document into *collectionName*

Returns an object containing:
acknowledged: true if write occurred, false if not
insertedId: the _id value of inserted document

Note: Mongo will automatically generate a unique ID

Example Database

You can use the _id field created by Mongo as a unique ID in your app – we will see an example when we look at adding MongoDB to a Node app

Also, if the collection you specify does not exist, it will be created

Example Database

db.*collectionName*.insertMany([{...}, {...}, ...])

Inserts each object in array into *collectionName*

Returns an object containing:
acknowledged: true if write occurred, false if not
insertedIds: array with _id values of inserted documents

Example Database

Insert some data from store-data.txt into the ‘products’ collection within the ‘store’ database

Example Database

Now that we have documents in a database, the next thing we will consider is reading/querying

The most basic query we can perform is to find all documents inside a collection:

db.*collectionName*.find( )

Example Database

In some cases, you do want to find all documents. But in many cases, you want to search for specific documents

The find method accepts an optional ‘query filter’ document

This allows you to specify advanced queries

Example Database

The most straightforward query filter document performs equality matching

For example, if we wanted to find all out of stock products:

```
db.products.find( { stock : 0} )
```

Example Database

More advanced queries can be formed using the supported ‘query operators’

These allow you to define queries such as less than X, greater than X, not equal to X, etc.

Example Database

The general form of a query document using a query operator would be:

db.*colName*.find({attribute : {$operator : value})

For example, to get all products with price greater than $300, we could use:

db.products.find({ price : {$gt : 300 } })

Example Database

The following query operators exist for comparing:

$eq – equality
$ne – not equal to
$gt – greater than
$gte – greater than equal to
$lt – less than
$lte – less than equal to
$in – is the value in the given array
$nin – is the value NOT in the given array

Example Database

So for example, if we had an array of product names we were trying to match:

```
db.products.find( {name : {$in : ["Tasty Cotton Chair",
"Practical Steel Pizza", "Intelligent Metal Mouse"] } } )
```

Example Database

To start to build more complex queries, we can add multiple conditions to the query document

This performs an implicit AND operation – matching documents that match all conditions

Example Database

So to find products with price greater than 250 AND a stock of more than 20:

```
db.products.find( {price : { $gt : 250 },
stock : { $gt : 20 } } )
```

Note: this does not work to specify two conditions for the same field (e.g., 250 < price < 500)

Example Database

You can add multiple operators to the same field…

```
db.products.find( {price : { $gt : 250, $lt: 500} } )
```

Products with 250 < price < 500

Example Database

There are also query operators to perform AND, OR, NOT, and NOR operations

For example:

{ $and: [{ <expression1> }, { <expression2> } , ... , { <expressionN> }] }

{ $or: [{ <expression1> }, { <expression2> }, ... , { <expressionN> }] }

This allows you to build very complex queries

Example Database

As an example, we can find products with a price in the range $200-$500 that have less than 25 stock:

```
db.products.find(
 {
  $and: [
          {price: {$gt:300, $lt:500}},
          {stock : {$lt:25}}
  ]
 }
)
```

Example Database

You can further combine multiple and/or operations

To find products with the price range 200-500 that also have more than 40 units in stock:

```
db.products.find( {$and : [
   {$and: [{price: {$gte: 200}}, {price: {$lte: 500}}]},
   {stock: {$gt: 40}}
] } )
```

There are more query operators available

For a full list, see:

https://docs.mongodb.com/manual/reference/operator/query/#query-selectors

Example Database

In some cases, we have nested/embedded objects

For example, the dimensions property in our dataset contains x/y/z sub-properties

You can access sub-properties using dot notation

Example Database

For example, to get all products that have all of their dimensions less than 10:

```
db.products.find( {$and : [
        {"dimensions.x" : {$lt: 10}},
        {"dimensions.y" : {$lt: 10}},
        {"dimensions.z" : {$lt: 10}},
]})
```

Note quotations around field

Example Database

If you do not NEED all of the data in a document, it is possible to limit the fields that are returned

This is especially important if distribution is involved and the query results are being transferred over the network

You can specify a 'projection document' to indicate the fields you require

Example Database

For example, to get only the name of products that are out of stock:

```
db.products.find(
   {stock : 0},  ← The query filter document
   {name: 1, _id: 0} ← The projection document
)
```

Example Database

The projection document should include a 1 as a value for each field you want returned

The _id field is returned by default, but can be disabled by supplying _id: 0

Alternatively, you can specify a 0 for all fields you DON'T want, and all others will be included

Example Database

Some other useful things you can do with find:

db.products.find().limit(X) returns only X results

db.products.find().skip(X) skips the first X results

db.products.find().skip(X).limit(Y) skips the first X results and returns only Y results

db.products.count() returns number of documents

These are all useful for pagination. They can all also be given query filter documents.

Example Database

And one last thing – you can also find only a single matching document:

db.*collectionName*.findOne(*filter*)

So if you know only a single document should match, or only need a single document, you can save processing time

Example Database

The next thing we will consider is how to update documents. There are several methods:

db.*collectionName*.updateOne(*filter*, *update*)

db.*collectionName*.updateMany(*filter*, *update*)

db.*collectionName*.replaceOne(*filter*, *replacement*)

Filter to select documents is same as finding

Example Database

An important note – all modifications in MongoDB are guaranteed to be atomic

This means that no two modifications will be performed simultaneously

This allows us to avoid issues like writing to the same file twice and corrupting the data

Example Database

So we can replace a single document like this:

```
db.products.replaceOne(
    {_id : ObjectId("5dc0e248db9ed905d06d3a3b")},
    {name : "Old Hat", price : 3.50, stock : 28}
)
```

We could specify a different filter and the first match would be replaced. But a common replace use case is using the _id field (i.e., PUT operations).

Example Database

Replacing is very straightforward – we take the old document and replace it with a new document

Updating involves changing only a subset of the fields

Example Database

Update operators are used within the update document to specify what fields you want to change and how you want to change them

Like the query filter operators, there are a number of update operators: https://docs.mongodb.com/manual/reference/operator/update/#id1

Example Database

So to decrease the stock of an item with the name "Tasty Cotton Chair" by 1:

```
db.products.updateOne(
        {name: "Tasty Cotton Chair"},
        {$inc: { stock: -1 }}
)
```

Example Database

To change the price of that object to 300 and add 10 units of stock:

```
db.products.updateOne(
        {name: "Tasty Cotton Chair"},
        {$set: { price: 300 }, $inc: {stock: 10}}
)
```

Example Database

You can include multiple fields for each update operator:

```
db.products.updateOne(
        {name: "Tasty Cotton Chair"},
        {$set: { price: 500, stock: 100 }}
)
```

Example Database

Nested values use dot notation, as with querying:

```
db.products.updateOne(
        {name: "Tasty Cotton Chair"},
        {$set: { "dimensions.x": 45 }}
)
```

Example Database

The db.collection.updateMany method works in a similar way, except all matching documents will be updated

Example Database

Each of the replace/update methods returns an object with a modifiedCount key

The value associated with this key indicates how many items were updated

Example Database

The last important operation we will consider is deleting/removing

To delete an entire collection:

db.*collectionName*.drop()

Useful if you make a mistake and need to start over...

Example Database

To remove a single document:

db.*collectionName*.deleteOne(*filter*)

Again, filter works just like when querying

Example Database

To remove all matching documents:

db.*collectionName*.deleteMany(*filter)*

Returns an object with 'deletedCount' key indicating the number of documents that were deleted

Summary

This has been a summary of the basics of MongoDB

There are MANY operations that you can perform

Do not worry about memorizing them all

Summary

You should, however, get comfortable with using the documentation to build/execute queries to achieve specific goals

Summary

Next, we will look at incorporating MongoDB into a Node.js web application

After that, we will look at Mongoose

Questions

Questions?