## Menzin Cheat Sheet for Getting Going with MongoDB ©Copyright by Margaret Menzin 2000—2023

**This assumes that you have already installed Mongo and created the folders**

**C:\data and C:\data\db and that you are working on the VM (or other Windows machine)**

1. **Firing up Mongo**
2. **Working with Databases**
3. **Working with Collections**
   1. **Getting Data into and out of a Collection**
   2. **Finding Data which Is Already in a Collection**
   3. **The Query Parameters**
   4. **Modifying Data**
4. **Working with Arrays**
5. **Working with Sub-documents**
6. **The Aggregation Pipeline**
7. **Indexes**
8. **References and $lookup**
9. **Using JavaScript in the Mongo Shell**
10. **Comments on cursors and on forEach()**

**1.Firing up Mongo**

* **You will need two command line windows or a command line window and a PowerShell window, or two PowerShell windows.  
    
  Both of them need to be in the directory   
  C:\Program Files\MongoDB\Server\6.0\bin**

There are several ways to accomplish this.

* For PowerShell windows, go to the Start menu on the Desktop and open the PowerShell Window program.  Then type:  
  **cd C:\”Program Files”\MongoDB\Server\4.0\bin**
* For a Command Line window you can migrate over to   
  C:\Program Files\MongoDB\Server\4.0\binusing the usual Windows Explorer (folder icon) and then **shift + right click and choose the “open command line window” in the pop up box.**
* For a Command Line Window you can open the window directly from the Desktop (either use the icon with the black screen,  or hold down the window key and type r, then type cmd in the dialog box).   
  Then you can migrate over to the folder you want by typing **cd C:\Program Files\MongoDB\Server\4.0\bin**
* NOTE: The cd command needs quote marks around Program Files when you are in the PowerShell b/c the blank space.
* **Start mongod**
  + In the PowerShell type **.\mongod** or
  + Or in the Command Line Window type **mongo**
  + NOTE: In the PowerShell the cli commands are prefixed with **.\**
* **Start mongosh**
  + In the PowerShell type **.\mongosh** or
  + Or in the Command Line Window type **mongosh**
* **The mongo shell is a javascript shell and therefore:**
  + Capitalization matters – i.e. myDB is not the same as mydb.
  + Extra blank spaces are ignored --- so you can put blank spaces around plus signs, parentheses for parameters, etc. for readability
  + You have variables!     
    Instead of typing   
             db.someLongCollectionName.count()   
             db.someLongCollectionName.find()  
    you can say  
             var myDB = db.someLongCollectionName  
             myDB.count()  
             myDB.find()  
    Note: db is actually a variable, which refers to the current database!
  + You can save javascript files and run them within the mongo shell.

1. **Working with Databases  
   Recall that a database is made up of a bunch of collections and that the collections are a bunch of documents.**
   1. **show dbs** displays all non-empty databases
   2. **db** displays the current database; db is a global variable
   3. **use myDB**  switches to the database myDB, creating it if it doesn’t yet exist
   4. **db.dropDatabase()** drops whatever database you are in
   5. **show collections** shows the collections for whatever database you are in
   6. **db.myCollection.drop()**  drops myCollection

1. **Working with Collections**
   1. **General commands**
      1. **show collections** shows the collections for whatever database you are in
      2. **db. myCollection .countDocuments()** counts the number of documents in **myCollection**
   2. **Getting Data into and out of a Collection**
      1. **NOTE: To copy & paste json from another file you use ctl+c to copy it and then right click in the command line or powershell window to paste it.**It is possible to do this by typing- for multiple documents  
             **db.myCollection.insertMany( [**   *then do the paste****] )***
      2. **db.myCollection.insertOne( {** *json for one document* **})** inserts one document
      3. **db.myCollection.insertMany( [   
            {** *json for one document* **}, ….  
            {***json for last document***)  
         ] )** inserts a bunch of documents listed in an array.
      4. **using mongoimport** to upload a whole file of json or a csv  
         **The mongoimport command is given from the command line in the bin folder – that is you must exit the mongo shell first.  Note the addition of the = in version 4.4 of MongoDB and later.**Using the Command Line window (or prefixing this with ./ if you are in the PowerShell window) type:  
         **mongoimport –db= *name\_of\_db\_you\_are\_importing\_into*  --collection= *name\_of\_collection\_you\_are\_creating***

**-- headerline --type =csv   
          --file= /example/data.csv   
          –ignoreBlanks**will import a csv file which is stored at /example/data.csv    
You may need to put the path file /example…..csv inside “  “s.  
The **–headerline** option tells Mongo to use the first row of the csv file as the field names.  
There is also a **–drop** option if you wish to replace an existing collection.  
On Moodle I have a separate page on other ways to use mongoimport with references.  
  
Various sites warn that some versions of Mongo will miss the last row of a csv file.  I have not had that problem with version 3.6 or 4.0 , but if you encounter that problem, drop the collection, add a row of garbage (with a unique value in one of the fields), import the whole collection again, and then use update() to get rid of the row of garbage (if it even appears.)

**v. db.myCollection.deleteOne( {***optional query}* **)** removes  the first    
                                      document which  matches the query             
                                      **db.myCollection.deleteMany( {***optional query}* **)** removes   
                                        all the first documents which matches the query.If there is no query then all  the documents in   
                                        the collection are removed.  
                                        Note: these functions replace the older, deprecated, **remove()** function.

1. **Finding Data which Is Already in a Collection**
   1. **db.myCollection.find()**  displays all the documents in **myCollection**
   2. **db.myCollection.findOne()**  displays the  first document in **myCollection**
   3. **db.myCollection .find({***some query parameters****} )*** displays all the documents in **myCollection** which meet the query parameters  (**See below for the query parameters)  
      db.myCollection .findOne({***some query parameters****} )*** to get the first document which matches the query.
   4. **db.myCollection .find({***some query parameters****} ).limit(n)*** returns the first n matching documents and
   5. **db.myCollection .find({***some query parameters****} ).skip(n)*** omits the first n
   6. **db.myCollection .find({***some query parameters****}, {****some field list* ***} )*** to get all documents which match the query and where the field list is of the form field1:1, field2:1, …fieldN:1  and lists the fields you want to display.  The \_id field is displayed unless you put  \_id:0 in your field list.
2. **The Query Parameters**
   1. The basic format is **{field1:value1, …, fieldN:valueN}** and the conditions are and-ed together.
   2. Operators (such as $gt for greater than)  are written as   
      **fieldX: {$*operator*: value}**  
      For example {age: {$gt: 18} } refers to documents with age > 18.
   3. All operators begin with $.  A short list is:  
        
      $eq   =            $gt   >             $gte  ≥          $lt   <      $lte    ≤       $ne  ≠  
      $and             $or                    $not      $in  for checking in an array.  
      A complete list is at <https://docs.mongodb.com/manual/reference/operator/query/> and  
      there are many examples at <https://docs.mongodb.com/manual/tutorial/query-documents/>

1. **Modifying Data**
   1. **The basic command is update, but you may use it to modify either on field or an entire document.**
   2. **Mongo has operations insert, update , and also both updateOneand updateMany with an upsert option.    
      By default, update() will update only the first document which matches the query.  
      If you want to update more thann one document use updateMany**The insert and update operations are what you would expect.  
      The **upsert** option (on an update() or replace() will update a document or field if it already exists and insert a new document/field if there is no document/field matching the query parameters.
   3. **db.myCollection .updateOne({***some query parameters****},   
                                                     {****a new document}* ***)***replaces the first document which matches the query parameter with the new document.
   4. **db.myCollection .update(One{***some query parameters****},   
                                                    {****a new document},   
                                                    {upsert:true}* ***)*** same as above but if the (old) document hasn’t been found then a new one is inserted.  We have set the upsert option to true.
   5. **db.myCollection .updateOne({***some query parameters****},  
                                       {$set:   {****field1:value1, …{fieldn:valuen} }}* ***)***replaces the values in the named fields.
   6. **db.myCollection .updateOne( *)*** also has a $incr operator and the ability to update multiple documents.  See <https://docs.mongodb.com/manual/reference/method/db.collection.update/index.html>  for details and examples.
   7. **There are updateMany() versions for all the updateOne possibilities. db.myCollection.updateMany( {},   
                                      *{$set:   {****field1:value1, …{fieldn:valuen} }}* ***)***will update all  the documents in myCollection.   That is, if you use the empty query parameter, then all the documents are updated.
2. **Deleting data**
   1. **db.dropDatabase()** deletes whatever database you are in
   2. **db.myCollection .deleteMany({***some query parameters****} )*** removes all documents in myCollection which match the parameters.
   3. **db.myCollection .deleteOne({***some query parameters****}, 1 )*** will remove the first document in myCollection which match the parameters.  
      Note: The second parameter in remove is the “justOne” value and here we have set it to 1 or True (not  to the number of documents we  are removing.)
3. **Working with Arrays**
   1. **Inserting arrays and elementsOne( {***some query parameters****},* {$set:  {***newArray***: [ ]** **}  } )**will insert an empty array with the field name newArray.
      1. **db.myCollection.updateOne( {***some query parameters****},* {$push  {***someArray***:** *someValue* **}  } )**will push (append to the end) someValue onto *someArray*
      2. ***db.myCollection.updateOne( {****some query parameters****},   
                                                      {$push  {****someArray****:*** *[anotherArray]* ***}  } )****will push (append to the end) the whole array anotherArray (as one element)  onto someArray.   
          That is if someArray was [1, 2, 3] and you push onto it  [4, 5]* ***{$push {someArray: [4, 5] }  }****then someArray will be [1, 2, 3, [4, 5]]*
      3. **db.myCollection.updateOne( {***some query parameters****},* {$push  {***someArray***: {$each:***anotherArray***}** **}  } )**will push (append to the end) someValue onto *someArray each of the values* in anotherArray.  That is if someArray was [1, 2, 3] and you use the $each operator to  push each of the values in [4, 5]   
         ***{$push* *{someArray: {$all:  [4, 5] }  } }***  
         then someArray will be  [1, 2, 3, 4, 5].
      4. **There are updateMany() versions of the updateOne90 function.**
   2. **Removing** **array elements**
      1. ***db.myCollection.updateOne( {****some query parameters****},   
                                                      {$pop  {****someArray****:*** *-1]* ***}  } )***will remove the first element of the array.
      2. ***db.myCollection.updateOne( {****some query parameters****},   
                                                      {$pop  {****someArray****:*** *1]* ***}  } )***will remove the last element of the array.
      3. Other array operators are described at <https://docs.mongodb.com/manual/reference/operator/update/push/index.html>
      4. **Aain, there are updateMany() version for all these statements.**
   3. **Queries on arrays**
      1. ***db.myCollection.find( {****someArrayField:  someValue* **})** will find all documents which have someValue appearing in someArrayField.
      2. ***db.myCollection.find( {****someArrayField:  someOtherArray* **})** will find all documents which have someArrayField being an exact match to someOtherArray.
      3. ***db.myCollection.find( {****someArrayField:* ***{$all:*** *someOtherArray* **})** will find all documents which have all the values in someOtherArray appearing in someArrayField.  (Order doesn’t matter.)
      4. ***db.myCollection.find( {****someArrayField:* ***{$in:*** *someOtherArray* **})** will find all documents which at least one element of someArrayField is in someOtherArray.  
           
         *Notice that $all gives us and's ; $in gives us or's.*
      5. ***db.myCollection.find( {****someArrayField:* ***{$size:*** *N****}}****)  will find all documents where someArrayField has exactly N elements.  That is ,   
         $size is the length of the array in question.*
4. **Working with Sub-documents**

**The dot notation is used to get you to sub-documents, just like in other programming languages you are familiar with.**

1. **The Aggregation Pipeline  
   The format for using aggregation is   
     
    *db.myCollection.aggregate( [   an array of stages, each in {  }* ])**

**Each stage has a collection of documents as input and also outputs a collection of documents.  
  
The most common stages are $group, $match, $project, $out, $limit, $count, $addField.  
  
Complete documentation on all stages is at** <https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/index.html>This cheatsheet provides only very basic information.  More information is also at the set of videos which begins at [**https://www.youtube.com/watch?v=uNwktK0qNXM&list=PLWkguCWKqN9OwcbdYm4nUIXnA2IoXX0LI&index=3**](https://www.youtube.com/watch?v=uNwktK0qNXM&list=PLWkguCWKqN9OwcbdYm4nUIXnA2IoXX0LI&index=3)

1. **{$limit: N}** returns only the first N documents from the input set.
2. **{$count: someName}**  counts the number of documents in the input set, and returns one document of the form  {someName: *the count*}.  
   Because this ends with such a summary it is  the last stage, when it is used. And there are other ways to get this information.
3. **{$out: “myNewCollection”}** takes all the documents at this stage and uses them to form a new collection, named myNewCollection.  Again, this is always a last stage in an aggregation pipeline.
4. **{$addField:  {newFieldName: someValue } }** will add the newFieldName with the value someValue --- but someValue may itself be a mongo object which has a calculuation.   For example, if our documents have fields hoursWorked and hourlyWage then we might say:  
       {$addField:  {grossPay:  {$multiply:  [“$hoursWorked”,” $hourlyWage”] } } }  
   Notes:   
         Any field referenced from the input documents is prefaced with $ and then enclosed in double quotes.  
        Please notice that here we are using **accumulators.**  
         The most useful operators for calculations are: $add,  $subtract, $multiply,  $divide, $avg, Ssum, $min, $max.  Details for all are at <https://docs.mongodb.com/manual/reference/operator/aggregation/multiply/>For all of arithmetic operators ($add, $subtract, $multiply, $divide) the operands are in an array – as in the example above.    
           $sum may also be used for the sum of other operations – see <https://docs.mongodb.com/manual/reference/operator/aggregation/sum/> and as a counter, typically with a $group.  If you have a collection of documents with ages in them then $sum: 1 may be used to count the number of people in each age group (see below.)
5. **{$project:  {field1:1, field2:1, …fieldN:1} }** returns a set of documents with only field1, …fieldN and \_id.  
   It is also possible to use the $project stage to restructure the documents, as in <https://www.youtube.com/watch?v=GG6Ax1B1Ag8&list=PLWkguCWKqN9OwcbdYm4nUIXnA2IoXX0LI&index=25>   
   Think of project as the verb in “I’m projecting from 3-dimensional space onto the x-y plane”.
6. **{$match:  {** *some query parameters***} }** returns those documents which match the query.  This is just like a find() but inside the aggregation pipeline.
7. **{$group: {\_id:** *uniqueGroupingExpression,  
                            field1: {accumulator object1}, …  
                           fieldN: {accumulator object N}* ***}*** *returns one document for each value of the uniqueGroupingExpression and also returns the calculated field1, ….fieldN.*If the uniqueGroupingExpression is null, then only one document is returned and its calculated field1, etc. will refer to the whole collection that was input to this stage.    
     
   Often the uniqueGroupingExpression is of the form “$fieldName”  where fieldName is one of the keys (field names) in the input collection.  But we can also have uniqueGroupingExpression be a document with several field names.    
     
   If you are grouping by values in only one field, the format is:  
            **{$group: {\_id:** *”$theOneFieldWeAreGrouping On”,  
                            field1: {accumulator object1}, …  
                           fieldN: {accumulator object N}* ***}***   
     
   If you are grouping on more than one field then you need to put them in a object.  For example, to group on the values in fields (from the input documents) fieldA and fieldB, you would code:

**{$group: {\_id:** *{fieldA: “$fieldA”, fieldB: “$fieldB”}  
                         field1: {accumulator object1}, …  
                        fieldN: {accumulator object N}* ***}*** *returns one document for each combination of the values in fieldA and fieldB  and also returns the calculated field1, ….fieldN.*

It is important to remember that when Mongo puts together *groups* the details from individual documents are not kept and that all you get are the summary data on the various groups – i.e. what’s in the accumulator objects (counts, mins and maxes etc.).  There will be *one document output for each group.*  
This is discussed  in <https://www.youtube.com/watch?v=dQIXLhKcv5c&list=PLWkguCWKqN9OwcbdYm4nUIXnA2IoXX0LI&index=9>   In that case you will need to use dot notation to get to the various subfields in \_id, as in \_id.fieldA and \_id.fieldB. <https://www.youtube.com/watch?v=RTjxSQaJcb8&list=PLWkguCWKqN9OwcbdYm4nUIXnA2IoXX0LI&index=15> also discusses this.  
  
The field1,  etc values (their accumulator objects) are typically things like **$add, $sum, $avg, $max, e**tc.  
Example – suppose we have a collection Students  of documents each of which refers to a student with fields *name, age, major, creditsEarned*.  Then we can have the following example:  
db.Students.aggregate([  
        {$group:   {\_id: “$major”,   
                             numberInMajor: {$sum:1},   
                             avCredits: {$avg: “$creditsEarned”}  
                           }   ,  
       ($out: allMajors }  
       ])  
*groups the students by major, adds 1 for each document in the group  
      and returns one document for each major (with the name of the major and   
     then numberInMajor and the average number of credits earned for all   
     students in that major.  
     Note that we used the values in the major field of the input documents by  
    asking for “$major”.   
    The $out stage writes those documents to a new collection allMajors.*     Please see other examples at   
      <https://docs.mongodb.com/manual/reference/operator/aggregation/group/>      especially the group by day, month, year examples.

1. **Accumulators and other operators for the aggregation pipeline**A detailed list and description may be found at <https://docs.mongodb.com/manual/reference/operator/aggregation/>     
   Please note that **$sum** may be used to add the values in an array.  See <https://docs.mongodb.com/manual/reference/operator/aggregation/sum/index.html> and  <https://www.youtube.com/watch?v=HkDNWiSUQCA&list=PLWkguCWKqN9OwcbdYm4nUIXnA2IoXX0LI&index=34> and [5](https://www.youtube.com/watch?v=HkDNWiSUQCA&list=PLWkguCWKqN9OwcbdYm4nUIXnA2IoXX0LI&index=34)for examples.

1. **Indexes**Indexes in mongo, as in MySQL, speed up frequently made queries, but involve overhead when documents are created and dropped and sometimes when they are updated.  
     
   Mongo automatically creates a document on the \_id field, and so that is frequently used in making references (establishing 1:1 and 1”many relationships – see next item.)  
     
   1. **db.myCollection.createIndex( {name:1, age:-1})** will create an index on ascending order of name and within that on descending order of age.  
      You may add an optional second parameter to name the index:  
      **db.myCollection.createIndex( {name:1, age:-1}, {name:myIndex})**
   2. **db.myCollection.dropIndex( “myIndex”)** does the obvious.
   3. **db.myCollection.getIndexes( )** does the obvious.
   4. **More information about managing indexes is at** <https://docs.mongodb.com/manual/tutorial/manage-indexes/>

1. **References and $lookup**
   1. **References**The easy way to link documents is by making the \_id field for one document a named field in another.  
        
      Suppose we have two collections, parent and child.   Each document in the parent collection has an \_id.   Then we can have a field in the child collection with key parent\_id and put there the value of the parent’s \_id.    
        
      We can use this method multiple times – e.g. have fields mother\_id and father\_id in the child document.

You could also link documents in the other direction – i.e. in the parent document have an array  myKids[ ]  which would hold the \_id field of all the parent’s children.  
  
In principle, one could use any field to link documents, but b/c mongo has an index on \_id fields it is most common to use that field.  
  
You can use these references in a query.  
  
<https://docs.mongodb.com/manual/tutorial/model-referenced-one-to-many-relationships-between-documents/index.html> discusses how to model with references and the previous paragraphs discuss how to model with embedded documents.  
<https://docs.mongodb.com/manual/reference/database-references/index.html#manual-references> has examples.

1. **$lookup** allows us to do joins on collections.  
   It is a stage in the aggregation pipeline.    
   <https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/index.html> has the syntax and examples are at <https://riptutorial.com/mongodb/example/28736/left-outer-join-with-aggregation----lookup->  and [https://www.stackchief.com/tutorials/%24lookup%20Examples%20|%20MongoDB](https://www.stackchief.com/tutorials/%24lookup%20Examples%20%7C%20MongoDB)     and explanations are at  <https://www.stackchief.com/tutorials/The%20MongoDB%20Aggregation%20Pipeline%20in%205%20Minutes>  and <https://www.youtube.com/watch?v=5ZZ5jWoqK6M>

The syntax for a $lookup is :

db. **collectionWhichWillHoldTheEmbeddedDocs**.aggregate([

      {$lookup:

  {

         from: " **collectionWhichWillGetEmbedded**",

         localField: " **fieldNameInTheCollectionWhichWillHoldTheEnbeddedDocs**",

         foreignField: " **fieldNameInTheCollectionWhichWillGetEmbedded**",

         as: " **nameOfFieldThatWillHoldTheEmbeddedDocs**"

       }

    },

    { **$out:** " **nameOfNewCollection**"}

 ])

**Warning- without the $out (or some other stage) you throw away the results of the embedding!**

**The 2 'fields' (localField and foreignField)  are what you are joining on.**

1. **Using JavaScript in the Mongo Shell  
   Please see “A little JavaScript to help you with the project”.**
2. **Comments on cursors and on forEach()**The find() method in mongo returns a **cursor** – that is a pointer which will iterate (step through) all the documents which are returned.  
   Cursors have many methods associated with them – including count(), limit() etc. You can find a description of all these methods at <https://docs.mongodb.com/manual/reference/method/js-cursor/>The .explain() and .explain(“executionStats”) are useful and discussed at <https://docs.mongodb.com/manual/reference/method/cursor.explain/> and in the videos on the aggregation pipeline.     
     
   For complicated code it may be easier to do the following:  
     
   var startTime = new Date().getTime();    //gets current time in milliseconds

            //the code you wish to time  
var endTime = newDate.getTime();  
var elapsedTime = endTime – startTime;

One of the most useful cursor methods is **forEach().** The syntax is:  
  
  
  
**db.myCollection .find({***some query parameters****} ).forEach( function(aDoc) { some code} )****where aDoc is the document which the cursor is pointing at as it iterates through whatever the* ***find()*** *returns.*That is:  forEach expects one parameter, which is a function.  And that function will itself have one parameter, namely, what the cursor is pointing to.

You may give any name to that parameter – here I chose aDoc – but as you will sometimes have nested functions you should name your parameter in a way that will help you follow your code (e.g. thisStudent, thisCustomer, etc.)  
  
The function  ***function(aDoc) { some code}*** is an anonymous function because we haven’t named it.  If it makes you happier, it is okay to write ***function myFunc(aDoc) { some code},*** *but**that is atypical and you need to be able to read code with anonymous functions.  
  
It is also okay to define your function separately and then use it in the forEach:* ***function myFunc (aDoc) { some code}***

**db.myCollection .find({***some query parameters****} ).forEach( myFunc )***

Either way, the function which is passed to forEach is a JavaScript function, with all the usual features of JavaScript plus all the Mongo statements.  (Please see “A Little JavaScript…”) and in particular variables inside it which are defined with a var are local to that  function.

Here is a pattern which is quite common with important comments below:  
  
**db.myCollection.find({**some query**}).forEach( function(thisDoc) {** *do stuff with thisDoc;*

*if you need to change values in thisDoc then do the following:* ***db.myCollection.update({\_id: thisDoc.\_id},   
                                                                 {*** *usual $set or $push***}  
                                                                 , {upsert:true}** //if upsert desired  
 **)** //ends the update

*do more stuff if desired***})**//The } ends the code for the function; the ) ends forEach.  
  
Please notice:

* The use of **thisDoc** to make update() work on the correct document.
* Having **thisDoc** gives us a way to refer to the specific document we are working on.
* Also, inside the anonymous function and also inside the update(), *any references to fields in the document should be of the form* ***thisDoc.fieldName****.*If you refer simply to fieldName, then mongo will look for a global variable fieldName.