Table of Contents

[USEFUL LINKS 1](#_Toc494561343)

[THINGS TO KEEP IN MIND/REMEMBER 1](#_Toc494561344)

[SETTING UP YOUR MONGOOSE.JS FILE - FILE FORMAT 2](#_Toc494561345)

[MONGODB SPECIFIC TERMINAL COMMANDS 3](#_Toc494561346)

[BASIC QUERIES 4](#_Toc494561347)

[RELATIONSHIPS AND QUERIES 6](#_Toc494561348)

[ADVANCED QUERIES 7](#_Toc494561349)

[AGGREGATE AND AGGREGATE OPERATIONS 7](#_Toc494561350)

# USEFUL LINKS

* Getting started with mongodb - <https://docs.mongodb.com/getting-started/shell/>
* MongoDB tutorial - <https://www.tutorialspoint.com/mongodb/>
* Installing MongoDB - <https://www.tutorialspoint.com/mongodb/mongodb_environment.htm>
* mongoDB column datatypes - <https://www.tutorialspoint.com/mongodb/mongodb_datatype.htm>
* Aggregation quick reference - <https://docs.mongodb.com/manual/meta/aggregation-quick-reference/>
* Aggregation, $group - <https://docs.mongodb.com/manual/reference/operator/aggregation/group/#pipe._S_group>
* Basics of MongoDB - <https://www.codeproject.com/Articles/828392/Basics-of-MongoDB>

# THINGS TO KEEP IN MIND/REMEMBER

* To have access to mongoDB in your terminal or apps, you have to have mongod running, open a terminal, navigate to the root directy with **cd**, then type **sudo mongod**. Leave this tab open and running, BUT be sure to exit mongod with **ctrl + c** before you close the terminal or turn off your computer, or it will keep running, which doesn’t necessarily hurt your computer, but will just be another thing your computer has to do. Google how to close it in this instance
* MongoDB saves things in JSON format, records or instances are documents, tables are called collections, the id column is “\_id” and is a 12 bytes hexadecimal number which assures the uniqueness of every document.
* When you’re using the terminal and mongo, you refer to all tables with “db.” infront of them. For example, once I go into the “Friends” database, if there is a “User” model created from the mean project, it will be saved as “users” in mongodb, but I would have to type “ **db.users.find({})** ” to find all the users, with db. before the plural table name
* When you create a model in mongoose, you create them singular and uppercase (User), and refer to them like that in mongoose. But in mongo, they are saved as and you refer to them as plural and lowercase, ex: “users”
* when it comes to many to one relationships, the many has a foreign key column where it saves the \_id of each one object in an array, ex: ( “comments”: [“582334ecaa2f0e1f1b8cb76d”, “58233524aa2f0e1f1b8cb76e”] ). But the one saves just the id of the many model, ex: ( “user\_id” : “id\_here” )

# SETTING UP YOUR MONGOOSE.JS FILE - FILE FORMAT

* The below format is how you setup your mongoose.js file, the file that connects you to mongodb. This is already included in the full mean template, the only thing you need to change is the “DBNAME\_HERE” part, same for this format:

var mongoose = require('mongoose'),

fs = require('fs'),

path = require('path'),

models\_path = path.join( \_\_dirname, "../models"),

reg = new RegExp( ".js$", "i" ),

dbURI = 'mongodb://localhost/ DBNAME\_HERE';

mongoose.Promise = global.Promise;

mongoose.connect( dbURI, {

useMongoClient: true

});

mongoose.connection.on( 'connected', function () {

console.log( `Mongoose default connection open to ${ dbURI }` );

});

mongoose.connection.on( 'error', function ( err ) {

console.error( `Mongoose default connection error: ${ err }` );

});

mongoose.connection.on( 'disconnected', function () {

console.log( 'Mongoose default connection disconnected' );

});

process.on( 'SIGINT', function() {

mongoose.connection.close( function () {

console.log( 'Mongoose default connection disconnected through app termination' );

process.exit( 0 );

});

});

fs.readdirSync( models\_path ).forEach( function( file ) {

if( reg.test( file ) ) {

require( path.join( models\_path, file ) );

}

});

# MONGODB SPECIFIC TERMINAL COMMANDS

* Once you have mongod running, open a terminal and type **mongo** to enter the database, type **exit** to exit mongo.
* **NOTE**: You never have to put column names in quote marks, even though when you query a document, all column names are in quotes.
* **show dbs** - shows all databases, which are usually created by your projects
* **use ‘db\_name’** - This makes you go into a certain dabatase, which you need to do to query that db. Example, if you have a db named ‘meanWall’, you would type **use meanWall**
* **show tables -** Use this command after you enter into a database to see all the tables within that database.
* **db.stats():** type this after entering into a database and it will show you everything about that database, number of tables (collections), total number of records (objects), etc.
* **INSERT:** insert is used to create a new document for a table, you type db.Table\_name\_plural.insert() and inside the parenthesis, you put your object in the correct format, ex: **db.users.insert( {name: “Jack”, age: 20} )**, if you insert into a collection that doesn’t exist, it will be created
* **DROP:** This is used to either delete a collection/table, or delete an entire database. ex: to remove the users table and all of its documents, type **db.users.drop()**. To delete an entire database, go into the database with “use db\_name” and type **db.dropDatabase()**, this will delete the database your currently in and all collections and documents.
* **CREATING A COLLECTION/TABLE:** To create a collection or table in mongodb, the format is as follows: **db.createCollection(name, options)**, the options are not the columns and data types, the options you can use are (capped, autoIndexId, size, and max), <https://www.tutorialspoint.com/mongodb/mongodb_create_collection.htm> see to get more info. The options are optional, to create a new users table I could type **db.createCollection(“user”)**
* **PRETTY():** The pretty method is used to format the results of your quries (usually find() ) in a “prettier” or easier to read way. For example, if I type **db.users.find()**, it would grab all of the users and look like:

{ "\_id" : ObjectId("582102b6f55c36191a378b47"), "name" : "Jerrod", "age" : 23, "\_\_v" : 0 }

{ "\_id" : ObjectId("58210eaba8098f19ce9d6d7d"), "name" : "Ryan", "age" : 40, "\_\_v" : 0 }

* **STILL PRETTY:** but if I typed **db.users.find().pretty()** it would look like:

{

"\_id" : ObjectId("582102b6f55c36191a378b47"),

"name" : "Jerrod",

"age" : 23,

"\_\_v" : 0

}

{

"\_id" : ObjectId("58210eaba8098f19ce9d6d7d"),

"name" : "Ryan",

"age" : 40,

"\_\_v" : 0

}

# BASIC QUERIES

* **FIND:** To find all the documents in a collection, type **db.users.find({})** in mongo or **User.find({})** in mongoose, or you could leave out the parenthesis and type **db.users.find()** in either mongo or mongoose. If you’re using mongoose, find() will return an array of objects, even if there is only one, or a blank array if there is nothing. There’s a lot you can add to this. There is the findOne, you could add {} to act as a where clause, multiple where’s etc. You can give multiple conditions by separating them with commas, but they differently than you’d expect. for example, in the user model, if I have one record with a name of “Jerrod” and age 23, and another record with name “Ryan” and age 40, if I typed **db.users.find( {name: “Jerrod”}, {age: 40})** in mongo, it only gets the Jerrod record, even though that only passes on conditional and the ryan record passes the age conditional, you’re better off using the AND/OR operaters detailed below.
* **INCLUDING/EXCLUDING DIFFERENT COLUMNS:** You can specify which columns show up with an option object, **{key: 1/0}**. After the initial {}, you make another object, and type a key/column name and 1 if you want it to show or 0 if you don’t. for example: **User.find({}, {name: 1, \_id: 0})**, this would find all the users but wouldn’t show the \_id column. It would show the name column, but any columns that aren’t specified through this method will show up anyways, so putting “name:1” is actually irrelevant, if you don’t include them in this method and they will be shown. You’re better off using this method to only list columns you don’t want shown.
* **$IN:** You can use $in to match more than one condition, for example: **db.users.find({home\_state: {$in: [‘Texas’, ‘California’]} })**, this query grabs all students whose home state is texas or California. The $in is followed by an array, and anything in that array can match.
* **FINDONE:** This is used to find a specific document, it only grabs one. You pute the conditions in a {} format, it will return the first object that it finds that matches the criteria. Ex: **db.users.findOne( {name: “Jerrod” })** in mongo or **User.findOne({name: “Jerrod”})** in mongoose, in mongoose, this returns a single object, not in an array line find() would.
* **SORT:** to sort by results you can use “sort()”, and put 1 for ascending or -1 for descending. For example, to sort all users by age from oldest to youngest, I would type **User.find({}).sort( {age: -1} )**
* **UPDATE:** the update() method updates single document by default, the format is “update( {conditional}, {$set: {column: new\_value}} )”, for example, if I typed **User.update( {name: “Josh”}, {$set: {age:34} } )**, it would find the first document it could find that had a name of “Josh” then set its age to be 34. To update multiple records, you add {multi:true} at the end, ex: **User.update( {name: “Jerrod}, {$set: {age:22}}, {multi: true})**. May not be necessary, but in the event that update() doesn’t update the lastmodified column by default, you could update it with **$currentDate: {}**, ex: **User.update( {age: 20}, {$set: {name: “Hobbo”}, $currentDate:{lastModified: true}})**, IMPORTANT NOTE: update() doesn’t activate the validations on mongoose models by default, using save() does however. If you want to see how to trigger the validations, see the next bullet point.
* **UPDATE - IMPORTANTE NOTE WITH VALIDATORS:** As, stated above, by default the update query doesn’t trigger your model validations. For example, if I had a model with a column like so: “**first\_name: {type: String, required: true, minLength: 4}**”, then I wouldn’t be able to create or save any document without it having a first\_name that is at least 4 characters long. And if I tried to update the record using the .save() method, I would still have to abide by these rules. However, using update(), I would be able to change the first\_name to less than 4 characters or even blank. In order to trigger the validations, you simply add “**{runValidators: true}**” after $set, for example: “**User.update({\_id: id\_here}, {$set: req.body}, {runValidators: true})**”
* **UPDATEMANY:** This is just like update() except it won’t’ stop at one document, ex: **User.updateMany({age: 21}, {$set {age:25}} })**, this would update all users that are 21 years old.
* **SAVE:** Use this method to create a new record (similar to insert), or even update a record in either mongo or mongoose. This is mainly used in mongoose, so I will work with those examples. If I wanted to create a new user and the columns were name and age, I could use “**var user = new User()** ” and it would be equal to a blank user record (but doesn’t save it), it would look like {name: ‘’, age: }, or if I submitted a form with the name and age, I could put that into the new user with “**var user = new User(req.body)** ” assuming the format/values are correct. Now it would look like **{ name: ‘Sam’, age: 25, \_id: ….}**, if there were other columns, they would be blank. Now I could manipulate it, I could change the name with “user.name = ‘Jose’ “, or set the value of a blank column, create a new column the same way, etc. Once you have the user how you want it, you just type (in mongoose) **user.save()**, and that saves/creates it with the values you’ve given it. However, you need to write an error handler in mongoose, either a .then() and .catch() chain via promises or but a callback function in the save(), ex: **user.save( function(err){ if(err){ do this…..}})** IMPORTANT NOTE: unlike update(), the save() query activates mongoose model validations
* **REMOVE:** to remove/delete a certain document. you put the conditions or what it looks for in an object format, for example, to delete a user: type **db.users.remove( {name: “Jack”} )** in mongo or **User.remove({name: “Jack”})** in mongoose, you could also add an option “justOne”, which stops after removing one record, just add it inside of the remove() after the condition, ex: inside of the remove parenthesis type: **( {name: “Jack”}, {justOne: true} )**, but using the \_id to find the record you want to remove is much safer and efficient than using “justOne”, it will only match one.
* **$push:** This is used to push a new value into an existing array. For example, if I had a user record as such: **{name: “Jerrod”, age: 24, interests: [‘Coding’, ‘Reading’, “Eating”]** **}**, then I could add a value to that array with $push, ex: **User.update({name: “Jerrod”}, {$push: {interests: “VideoGames”}} )**, this would push “Videogames” into the interests array. $push takes an object as an argument, with the key being the name of the column that you want to push into, and the value being the new value that is pushed.
* **$pop:** Almost the exact same as $push but it takes a record out. Ex: in the record shown in the $push bullet has (interests: [“Reading”, “Coding”]), then running **User.update({name: “Jerrod”}, {$pop: {interests: 1}} )**, this evidently takes an index as an argument, though it may also take a string, not sure
* **LIMIT:** used to limit the max number of records your query can find or grab. It only accepts one number type argument. ex: **User.find({}).limit(15)**
* **SKIP:** Similar to limit, but it is how many documents that your query will skip, for example: **User.find({}).limit(1).skip(1)** will only show the second document, since it skips the first one and stops after hitting its limit, one.
* **$currentDate:** This can be used many ways but it is equal to as it sounds, the current date. For example, if I wanted to add an updated\_on column, then I could run **User.update({\_id: …..}, {$currentDate: {updated\_on: true}} )**, this would add an ‘updated\_on’ column and set its value equal to the current date

# RELATIONSHIPS AND QUERIES

* **NOTE:** This section is just for mongoose
* **POPULATE:** This is used to “populate” the array in the many of a many to one. For example, if I have a Message and Comment model with many-to-one relationship, one message can have many comments. In the Message model, it has a column title “comments” with an array as the value. It is empty at first, but saves the \_id of any comments that are related to, meaning it looks like: ( **comments: [ 582334ecaa2f0e1f1b8cb76d, 58233524aa2f0e1f1b8cb76e, 59c147c5d8c0c018b1194985 ]** ). So it is an array of \_id’s, and if I queries the message model that is how it would appear, ex: **Message.find({})**, however, with the populate() method, mongoose replaces the \_id’s with the actual comment instance/object. For example, if I typed **Message.find({}).populate(‘comments’).exec()**, the comments column would now look like: **comments: [ [Object], [Object], [Object] ]**, so now we have direct access to the comments. In its basic form, you type the name of the column that you want to be populated, in this case the “comments” column in the Message model. You can also use “.populate()” on a query to find the one reference, meaning I could do “**Comment.find({}).populate(“\_message”).exec(function(){})**” to include the message that each comment belongs to.
* **USING POPULATE FOR MULTIPLE ASSOCIATIONS AND POPULATE INSIDE OF A POPULATE:** For example, let’s take the same case as above, we have a message that was created by a user, and has multiple comments, but those comments were also created by a user and have a ‘user\_id’ column. We could include the comments with a “.populate(‘comment’)”, but the ‘user\_id’ for each comment would literally be the \_id, what if we wanted to actually see the user? Then we do a sort of nested populate, for example: “**.populate( {path: ‘comments’, populate: {path: ‘user\_id’} } )**” Basically, this populates the comments column for a message, so it includes the comments associated with it, but then goes into those comments and populates their user\_id field, so we now have access to each user that wrote that comment. Basically, we’re using populate within a populate, the normal format is: “**.populate(‘column\_name’)**” but with this, it is: “**.populate( {path: ‘column\_name’, populate:{path: ‘column\_name’} })**”, a full example of this, where we want to include the user who created the message, all of its comments, and the users who created those comments is: “**Topic.findOne({\_id: req.params.id}).populate('user\_id').populate({path: 'comments', populate: {path: 'user\_id'} }).exec()** ”
* **POPULATE IMPORTANT NOTE:** Normally, when executing queries in mongoose, you use a callback (User.find({}, function….), or a promise ( User.find({}).exec() ). However, with populate(), you have to put the .exec() method after it, since the callback has to be at the end, and you can’t put it inside of the .populate() since you put in the column. This doesn’t change the promise format, it looks like (**Message.find({}).populate().exec()**), but the callback format changes, you now put a exec after the populate, and put the callback function in there. ex: **Message.find({}).populate(“comments”).exec(function(err, messages){…})**

# ADVANCED QUERIES

* **FIND - OPERATORS (GREATER THAN, LESS THAN, ETC.):** There’s a lot you can add onto the regular find({}), you can basically add on a where to it by just putting something in the curly brackets. Ex: **User.find({name: “Jerrod”})** would find and return in array format, all the documents with a name of “Jerrod”. If you use operators ( greater than, less than), you include them in their own {}. For example, greater than would look like **User.find({ age: {$gt:20} })**, this would find all users who are older than 20, the less than 20 would be **User.find( {age: {$lt:20} } )**, greater than or equal is **$gte**, less than or equal to is **$lte**, not equal to ( != ) is **$ne**,
* **FIND - AND/OR:** To use multiple conditionals you can use and/or, and makes sure that all conditionals must pass, you use the “$and:” operator, and you put all conditions inside of an array, with each condition as it’s own object separated by commas, ex: **User.find( {$and: [ {name: “Jerrod”}, {age: 40} ] })**. If I wanted to use the OR operator to make just one conditional pass, I would use the exact same format but replace “$and:” with “$or:”, ex: **User.find( {$or: [ {name: “Jerrod”}, {age: 40} ] })**. A slightly complicated example of using both, is if I wanted to find a user that is older than 20, and is either named “Ryan” or “Jerrod”. So they have to be older than 20 no matter what, but only one of the names has to pass, but at the same time at least one name has to pass. You don’t actually need to use the $and, it would look like: **User.find( { age: {$gt :20}, $or:[ {name: “Jerrod”}, {name: “Ryan”}] })**, basically the $or would be in the same curly braces that opened at the beginning, before “age”.

# AGGREGATE AND AGGREGATE OPERATIONS

* **LINKS:** For basics and explanation - <https://www.tutorialspoint.com/mongodb/mongodb_aggregation.htm>. For a quick reference and examples - <https://docs.mongodb.com/manual/meta/aggregation-quick-reference/>, aggregate and $group - <https://docs.mongodb.com/manual/reference/operator/aggregation/group/#pipe._S_group>,
* **AGGREGATE:** Aggregate operations process data records in some way and return computed results. For example, in SQL, the count(\*) followed with a “group by” is the equivalent of a mongodb aggregate. aggregate() accepts an array of stages as its argument, and it can be complex. A list of operations is shown in the link above, but include $group, $sort, $lookup, etc. $group is a operator that has its own “accumulators” such as $sum, $avg, etc.
* **GROUP AND ITS ACCUMILATORS:** $group, like it sounds, groups documents by some specified expression and outputs to the next stage a document for each distinct grouping. the format is: **{$group: {\_id: <expression>, <field1>: {<accumulator 1>: <expression1>}….}}**, the \_id isn’t referring to the column, it is which key the documents are grouped by, for example: **\_id: “$author”**  means that the documents will be grouped by authors, so all documents with the same author will be grouped together. When referencing a column name, either with the\_id or an accumulator, you always put the column name in quotes with a $ sign before it, ex: (**\_id: “$age”, or $avg: “$price”**). \_id can also be “null” in which case, it basically groups all the documents together. the accumulators are optional, for example if I had 3 users, with two of them having an age of 23, and 1 having an age of 40, and I typed **db.users.aggregate([ {$group: {\_id: “$age”}} ])** in mongo, then the results would be:

{ "\_id" : 40 }

{ "\_id" : 23 }

* **STILL GROUP:** it grouped the 3 users together, with the two users of the same age collapsing together. Now, for the next few examples, they work with this data set in the “sales” collection/table

{ "\_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-03-01T08:00:00Z") }

{ "\_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-03-01T09:00:00Z") }

{ "\_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-03-15T09:00:00Z") }

{ "\_id" : 4, "item" : "xyz", "price" : 5, "quantity" : 20, "date" : ISODate("2014-04-04T11:21:39.736Z") }

{ "\_id" : 5, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-04-04T21:23:13.331Z") }

* **STILL GROUP USING THE SALES COLLECTION:** Now, for a relatively simple example, lets say I wanted to group all of the documents together, and see how many documents there are, what is the average quantity, and the total price for the entire collection (which would be the sum or adding up of the price\*quantity for all documents). So I would set \_id to null, so that all the documents will be grouped together into one, then the $sum to actually count how many documents there are, the $avg accumulator to get the average quantity, and a mix of the $sum and $multiply to find the total price. See the query below followed by its result in green

**db.sales.aggregate([ {$group:**

**{**

**\_id: null,**

**totalPrice: {$sum: {$multiply: [“$price”, “$quantity”]}},**

**averageQuantity: {$avg: “$quantity”},**

**count: {$sum: 1}**

**}**

**} ])**

Result: { "\_id" : null, "totalPrice" : 290, "averageQuantity" : 8.6, "count" : 5 }

* **STILL GROUP USING THE SALES COLLECTION:** so the text before the actual accumulator (totalPrice, averageQuantity, and count) are actually the names of key/column where the result for that accumulator will be displayed. $sum: 1, as shown earlier just counts how many documents are grouped together. $avg finds the average of the column you give it, “quantity” in this case. And the $multiply accumulator multiplies the two columns you give it, “price” and “quantity” in this case. And when it is contained within the $sum accumulator, the $multiply multiplies the columns that give it together, then the $sum adds each of the result from $multiply. Meaning, $multiply, based on the collection we’re working with, would get (20, 20, 50, 100, and 100), but it wouldn’t multiply those results, it stops after multiplying the columns. Then, the $sum adds together those results according to which columns are grouped together. Since, they’re all grouped together in this instance, $sum adds them all together.
* **STILL GROUP USING THE SALES COLLECTION:** Now, a more complicated example, but similar, the only difference form this one and the one above is the \_id key. This example will group the results together by date ( “date”: ISODate("2014-03-01T08:00:00Z") ), but we want to group them together if they are in on the same MM/DD/YYYY format, in ISODate, it includes time as well, which could cause some documents to not be grouped together even though they are on the same date. However, you can specify multiple keys for the \_id field. Look at the example below, and results in green:

**db.sales.aggregate([ {$group:**

**{**

**\_id: {month: {$month: “$date”}, day: {$dayOfMonth: “$date”}, year: {$year: “$date”} },**

**totalPrice: {$sum: {$multiply: [“$price”, “$quantity”]}},**

**averageQuantity: {$avg: “$quantity”},**

**count: {$sum: 1}**

**}**

**} ])**

**RESULT BELOW:**

{ "\_id" : { "month" : 3, "day" : 15, "year" : 2014 }, "totalPrice" : 50, "averageQuantity" : 10, "count" : 1 }

{ "\_id" : { "month" : 4, "day" : 4, "year" : 2014 }, "totalPrice" : 200, "averageQuantity" : 15, "count" : 2 }

{ "\_id" : { "month" : 3, "day" : 1, "year" : 2014 }, "totalPrice" : 40, "averageQuantity" : 1.5, "count" : 2 }

* **STILL GROUP USING THE SALES COLLECTION:** So, I grouped the documents together if they had the same month, day of month, and year, taking it from the ISODate in the “date” column. We’re able to grab the specific parts of the date through the $month, $dayOfMonth, and $year operators, by giving them a date (may have to be an isodate format). If you compare the result to the sales collection above, you’ll notice that there were 2 document on 4/4/14, that had different times in the ISODate, using the method above in the query ignores the time and groups them by just date.
* **$MATCH AND $COUNT:**For example, let’s say I had a collection/table called scores, with a subject column and score (as a number) column. So, if I wanted to see how many documents in that collection scored over 80, I could first use the $match stage/aggregate\_operation, which works like it sounds, basically a “where” statement, then I could use the $count stage, which as it sounds, just counts how many records the query finds and puts it in a field/column that you name. The query would look like **db.scores.aggregate( [ {$match: {score: {$gt:80}}}, {$count: “passing\_scores”} ])**, so this would find all of the documents that have a score over 80, and count how many there were and would show it under a column titled “passing\_scores”. the $count stage takes only one value as a string, which is the name of the column that shows the count result. If there were only 4 records that had scores over 80, the result would be **{“passing\_scores” : 4}** A better way to see the query used above is shown below, each stage or operator is in its own set of curly braces:

**db.scores.aggregate( [**

**{$match: { score:{$gt: 80}} },**

**{$count: “passing\_scores”}**

**])**

END