**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/>
* 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” )

**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 in mongo you could leave out the parenthesis and type **db.users.find()**, if you’re using mongoose, this will return an array of objects. There’s a lot you can add to this. You can also 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 irrelvenat, 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. 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.
* **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}})**
* **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…..}})**
* **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.
* **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.

RELATIONSHIPS AND QUERIES

* **POPULATE:**

**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:** 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 $sum, $avg, $min, $max, etc.
* **$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