|  |  |
| --- | --- |
| -  Sindy Saintclair  Friday, January 28, 2022  Lesson 3: Updating Documents | |
| **Learning Objectives and Questions** | **Notes and Answers** |
| **OVERVIEW** | Update operations modify existing documents within a collection. So far, you have learned how to Create and Read documents in a Mongo database. This lesson explores how to update documents within your database. |
| **Update a Single Document** | The updateOne( ) method is the simplest of the update methods. Below is the syntax:  db.[collection].updateOne({filter}, {update}, {options})  The above query will find the first document that matches the filter. It will then update the data to whatever has been set in the query. This will be using the $set keyword. Lastly, there are particular options that you can use at the end of your query. Below is an example of an updateOne() method using your appusers database:  db.appusers.updateOne({"\_id" : 1}, { $set : { "middleName" : "Gertrude"}}, { upsert : true})  Above, you are using the updateOne  command to change the middle name of your user with an \\_idvalue of 1 to Gertrude. In this case, your filter criterion is that the \\_id is equal to one, and then you want to get the middleName value to Gertrude. So, you have your *Filter,* you have your *Update,* and you are using the *Option* upsert, which means “if found then update, otherewise insert.” You will learn more about upsert within this lesson.  You should get a result from MongoDB that gives you the normal ‘acknowledge’ message of true, followed by a ‘matchedCount’ of 1 and a modifiedCount of 1 as well. It told you that it did find and modify exactly one row, changing the middle name to Gertrude. That was expected, as your filter condition was on the \\_id, and you know that value to be unique in your collection. Take a moment now to execute a find() command and examine the document to verify that it did make the change you requested.  ***Update Many Documents***  The updateMany() method is very similar to updateOne(). The difference is that instead of finding the first document that matches the filter in the query, it will find and update all documents that match the filter. See below:  db.appusers.updateMany({"firstName" : "Tommy"}, { $set : { "middleName" : "Henry"}}, { upsert : true})  The above query will set all documents with a first name of "Tommy" to have a middle name of “Henry”. If you noticed, there is something that says upsert : true.  ***Using update( )***  The update() command has all of the functionality available to it that both the update() and updateMany() methods have when combined. In the last {} block in the statement, you previously include upsert : true, which is available to all three of the update() variations (updateOne, updateMany, update). Unique to the generic update() command is another option called multi, a Boolean value. Setting this to true makes your update() command function as an updateMany(), while setting it to false makes the update() act as if it was an updateOne command. So it can do both, but you need to specify which you are intending. An example of the syntax is below:  db.[collection].update( {filter}, {update}, {upsert : true/false, multi: true/false})  Now you know how to query a basic update method, it’s now time to explore the query’s parameters more in-depth. You have learned how to filter in previous lessons, and those query selectors are all applicable here. Now you’ll look at the *Options* available to you for the update methods. |
| **Update Options** | Below are the options that are available when using any update query. First, take a look at the syntax of where these options will go:  db.[collection].updateOne({filter}, {update}, {options})  ***upsert***  The value of upsert is a Boolean, and if set to true, it will create a new document if no document matches the criteria. If upsert is not defined, the default value is false and will NOT create a new document when no match is found. upsert is optional.  Now take a look at an example. Consider the following query:  db.appusers.updateOne({"firstName" : "Tommy"}, { $set : { "middleName" : "Henry"}}, { upsert : true})  If your filter above had instead consisted of {"firstName" : "Tommy"}, then MongoDB would have found the first document that had a firstName  of “Tommy”, and changed the middleName on that document to “Henry”.  But what if the value does not exist? That’s where the upsert option comes into play. If a middleName does not exist, it will change the update into an insert, and add a middleName field for that document. Now, take a look at your user “Coderboy”. Time to find “Coderboy” by his first name and add a middle name, “Cole”. The Syntax for this upsert becomes:  db.appusers.updateOne({"firstName" : "Coderboy"}, { $set : { "middleName" : "Cole"}}, { upsert : true})  You should receive from MongoDB the “acknowledged/true”, “matchedcount/1”, and “modifiedcount/1” that you did with the previous update. Use your find() command to locate the “Coderboy” document in the collection. You will see that the field of middleName has been added to your document.  ***Multi***  The multi value is only used when an update() is queried. If it is set to true, it will update multiple documents that meet the query filter. The default value is *false* which will update only one document. So when using update() and you want to update many documents, you must include the multi option in your query.  Below is the syntax:  db.[collection].update( {filter}, {update}, {upsert : true/false, multi: true/false})  ***writeConcern***  A writeConcern is an option available to you when running queries. It describes the level of acknowledgement requested from MongoDB for write operations. You won’t dig deep into writeConcern but is good to know it exists.  Great work so far! Now that you have a basic understanding of the syntax when updating, it is time to dive into what operators you can use. |
| **Update Operators** | Below is a list of common update operators.  ***$currentDate***  Will set the value of a field to the current date. It can be set either as Date or a Timestamp. If neither is specified, the default is Date. Below is the syntax:  { $currentDate : { <field> : <typeSpecification>, ... } }  Above, <field>will be whatever field to which you are updating the date.  The <typeSpecification> can be either one of two things:   * A Boolean value, when set to true, will set the current date as a Date. * A document that explicitly defines the type: { $type : "timestamp" } or { $type : "date"}. Keep in mind that this operator is case-sensitive and both “timestamp” and “date” must be lowercase.   Now you will update one of your appusers to have a signedUp field to indicate when they have signed up. Now you will look at using a Boolean value:  db.appusers.updateOne({ \_id: 1 }, { $currentDate: { signedUp: true } });  If you run the above query, you will get the acknowledged true message that the user with an \_id of 1 has been updated. If you run a find() query or look in Atlas, you will see now that your user has a field of signedUp and the date is today’s date.  ***$inc***  The $inc operator increments a field by a specified value.  { $inc: { <field1>: <amount1>, <field2>: <amount2>, ... } }  Above, the field that you want to be incremented is defined, and then the amount for it to be incremented by is specified.  The $inc operator accepts positive and negative values. If the field defined does not exist, it will create the field and set it to the specified value. If you use the $inc operator on a field with a *null* value, you will get an error.  Take a look at an example of this. Consider the following collection called products:  {  \_id: 1,  quantity: 10,  metrics: {  orders: 2,  ratings: 3.5  }  }  Above, you have one product with an \_id of 1, quantity of 10, an embedded document called metrics with an orders field of 2, and ratings of 3.5. Below, you are going to update the document with an \_id of 1:  db.products.update({ \_id: 1 }, { $inc: { quantity: -2, 'metrics.orders': 1 } });  Above, the update() operation uses the $inc operator to decrease the quantity field by 2 (i.e., increase by -2) and increase the metrics.orders field by 1. Because the metrics field is an embedded document, you need to use dot notation to say where the orders field is located. If you run the query below, you will create an orders field and set it to 1 outside the metrics field. Do not run this query!  db.products.update({ \_id: 1 }, { $inc: { quantity: -2, orders: 1 } });  Remember to use dot notation when updating embedded documents!  ***$min***  The $min operator will update the value to a specified value if it is less than the field’s current value.  { $min: { <field1>: <value1>, ...} }  If the field does not exist, the $min operator will set the field to the specified value.  Consider the following collection named scores:  { \_id: 1, highScore: 800, lowScore: 200 }  The lowScore is currently 200. The following query uses $min to compare 200 to the specified value of 150. If 150 is, in fact, less than 200, it will update the lowScore field to 150.  db.scores.update({ \_id: 1 }, { $min: { lowScore: 150 } });  Because 150 is less than 200, the field will be updated to 150. If you run a find() query, the document will now be:  { \_id: 1, highScore: 800, lowScore: 150 }  If you run the query below, nothing would happen because 150 is less than 250, not the other way around:  db.scores.update({ \_id: 1 }, { $min: { lowScore: 250 } });  ***$max***  The $max operator updates the value to a specified value if it is *greater than*  the field’s current value.  { $max: { <field1>: <value1>, ...} }  If the field does not exist, the $max operator will create and set the field to the specified value. Continuing with the scores collection, the query below will update the highScore field because 950 is greater than 800:  db.scores.update({ \_id: 1 }, { $max: { highScore: 950 } });  The scores collection now looks like below:  { \_id: 1, highScore: 950, lowScore: 200 }  ***$mul***  The $mul operator will multiply the current value by the specified value. It will then set the value of the field to the product of the multiplication operation.  { $mul: { field: <number> } }  If the field does not exist in a document, $mul creates the field and sets the value to zero.  Time to move back to the inventory collection you have in your database.  Say you just updated your inventory and doubled all products that have a status of “A”. You could update your database by using the $mul operator. See below:  db.inventory.updateMany({ status: 'A' }, { $mul: { qty: 2 } });  You are filtering through your documents that have a status of “A”. You are then defining that you want to multiply your documents qty  field by 2. If you run the query above, your products with a status of “A” will now have twice as many products.  ***$set***  The $set operator replaces the current value of a field with the specified value.  { $set: { <field1>: <value1>, ... } }  If the field does not exist, the $set operator will create and set the field to the specified value. This operator is used very frequently when updating documents. You saw an example of this during the updateOne() description at the beginning of this lesson. Below is another example of using $set:  db.inventory.update({ item: 'journal' }, { $set: { status: 'B' } });  Above, you are looking for the first document where the item field is journal. You are then setting the status of that document to “B”.  ***$unset***  The $unset operator will delete a particular field.  { $unset: { <field1>: "", ... } }  By defining the field name and using “”, it will delte that field. If the field already does not exist, $unset will do nothing.  Now time to delete the status field of any documents where the item field is journal: db.inventory.update({ item: 'journal' }, { $unset: { status: '' } });  If you run a find( ) query or look in Atlas, you can no longer see a status field for the journal  item.  ***$rename***  The $rename operator updates the name of a field.  {$rename: { <field1>: <newName1>, <field2>: <newName2>, ... } }  The new name must be different from the existing name the $rename operator uses, and $unset of both the old and new names and then performs a $set operation with the new name. If you choose to rename a field, the order of the document will not be preserved, and the renamed field may move within the document.  If the field you are trying to rename does not exist within the document, $rename will do nothing.  Now you can rename the field qty to quantity for every document in your inventory collection:  db.inventory.updateMany({}, { $rename: { qty: 'quantity' } })  Above, you are using the updateMany() method. You do not need any filter specifications because you want to update every document, so that is why there are empty curly braces { }. Last, you are renaming the field qty to quantity. If you run the query above, you will notice that “qty” is now “quantity”. |
| **More Update Queries** | The methods you have learned in this lesson are often used, but is good to be aware of the other methods available to you. The below methods accomplish a lot of the same things you have learned so far, but just in a slightly different way.  ***Replacing Documents***  If you want to replace an entire document, you can use the replaceOne() method. This method will replace the entire contents of your document with a new document that you will specify in a command. Most often used with a filter that is the \_id of the document you wish to replace, it has the same structure and is available to be used exactly like an update would be. The structure of a replaceOne() is: db.[collection].replaceOne( {filter}, {replacement document}, {options})  Time for you to try it!  In your appusers collection, update your user with an \_id of 2. This document currently has the data below:  {  "\_id": 2,  "lastName": "Wilson",  "firstName": "Coderboy",  "age": 18,  "favoriteColor": "blue"  }  This user has just informed you that he has completely changed his name, is one year older, and no longer has the favorite color of blue; it is now green .Using replaceOne(), you can update all of his info, so he keeps the same \_id.  db.appusers.replaceOne(  { \_id: 2 },  { lastName: 'Beck', firstName: 'Rupert', age: 19, favoriteColor: 'green' }  );  After running the above query, you have replaced all of the fields with the new info. This could have been accomplished using any update query but is good to know that you can also replace documents. This document now looks like this:  {  "\_id": 2,  "lastName": "Beck",  "firstName": "Rupert",  "age": 19,  "favoriteColor": "green"  }  ***Find and Replace***  You can also use the findOneAndReplace() method. This is used if you would like to return the original document before updating it. Time to insert a new collection:  db.scores.insertMany( [ { "\_id" : 1521, "team" : "Fearful Mallards", "score" : 25000 },  { "\_id" : 2231, "team" : "Tactful Mooses", "score" : 23500 },  { "\_id" : 4511, "team" : "Aquatic Ponies", "score" : 19250 },  { "\_id" : 5331, "team" : "Cuddly Zebras", "score" : 15235 },  { "\_id" : 3412, "team" : "Garrulous Bears", "score" : 22300 } ] )  Now, run the following query:  db.scores.findOneAndReplace(  { score: { $lt: 20000 } },  { team: 'Observant Badgers', score: 20000 }  );  When you run the above query, you should see that the Playground Results returns the documents before it is replaced with the new data, as shown below:  Find and replace. Search string db.scores.findOneAndReplace.  Figure 3-3: NEW Find and Replace  If you run a find() query for the team with a name of Observant Badgers, you will see the new data.  ***Find and Update***  The findOneAndUpdate query works the same way as findOneAndReplace. All you are doing is updating data instead of replacing it. This query will also return the current document before updating it.  **db.collection.findOneAndUpdate( filter, update, options )** |