



LESSON

# The MongoDB Query Language (MQL)

Google slide deck available [here](#)

This work is licensed under the [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](#)  
(CC BY-NC-SA 3.0)

## Overview



### Learning Objectives

At the end of this lesson, learners will be able to:

- Describe the use cases for the MongoDB Query Language (MQL).
- Perform basic CRUD operations using MQL.
- Execute queries using MQL find() and delete().
- Use the most common MQL query operators to optimize searches.

### Suggested Uses

- A whole lecture or spread out across multiple lecture periods
- Handouts / asynchronous learning
- Supplemental reading material - read on your own / not part of formal teaching
- Complement to University courses [Introduction to MongoDB](#) and [MongoDB for SQL Pros](#)

This lesson is a part of the courses [Querying in Non-Relational Databases](#) and [Introduction to Modern Databases with MongoDB](#).

### At a Glance



Length:  
70 minutes



Level:  
Intermediate



Prerequisites:  
[Querying in Relational and Non-Relational Databases](#)

This work is licensed under the [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](#)  
(CC BY-NC-SA 3.0)

Share your feedback: We hope these curriculum materials will be a valuable resource for you and your learners. Let us know how the materials work for you, what we can improve on, and how MongoDB for Academia can support you via our brief [feedback form](#).

MongoDB for Academia: MongoDB for Academia offers resources for educators and students to support teaching and learning MongoDB. Check out our [educator resources](#) and join the Educator Community. Students can receive \$50 in Atlas credits and free certification through the [GitHub Student Developer Pack](#).

Last Update: March 2025

# This lesson includes exercises



Follow along using these tools

## Create a Database

- [MongoDB Atlas](#) (cloud)
- [MongoDB Community](#) (local install)

## Connect to Your Database

- [MongoDB Shell](#) (open source)

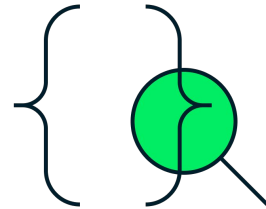
For more instructions on how to use MongoDB Atlas with your students, see [Atlas for Educators](#)

These exercises were designed to work in the MongoDB Shell. If you prefer to use a GUI to interact with your data, please use [MongoDB Compass](#) or the [MongoDB Atlas UI](#).

Simple syntax

Designed to query documents

Only queries a single collection



## MongoDB Query Language

The MongoDB Query Language or MQL, was designed to have a simple syntax as well as being explicitly designed for querying documents.

It only queries a single collection; when you need to query multiple collections or perform more complex data processing the MongoDB Aggregation Framework should be used.



# MongoDB Query Language

MQL is designed for **single collection queries** and it is typically used for creating, reading, updating, or deletion (CRUD) operations.

## MQL query operators:

- Comparison
- Logical
- Element
- Array
- Evaluation
- Bitwise
- Comment
- Geospatial
- Projection, Update and Update Modifiers

MQL is designed for single collection queries and it is typically used for creating, reading, updating, or deletion (CRUD) operations. In this lesson, we'll focus on introducing these CRUD operations.

MQL's query operators help enhance the MQL CRUD operations.

MQL operators vary from comparison to logical to array, as well as specific purpose operators like bitwise or geospatial. These allow for complex operations to be performed on the documents. We'll explore these operators later in the lesson as well.

The background is a dark green rectangle. On the right side, there is a lighter green shape that curves around the edge. In the top right corner of this lighter green area, there is a small, stylized leaf icon.

# MQL Find()



# MQL Find()

**db.<collection>.find()**

Query filter document

```
db.collection.find({ <field1>: <value1>, ... })
```

Specifying query operators

```
db.<collection>.find({ <field1>: { <operator1>: <value1> }, ... })
```

To begin to learn MQL, we will start with going over the find() command.

The MQL find() specifies the collection as part of the syntax of the method. As noted earlier, MQL and the functions like find() only work on a single collection.



# MQL Find()

```
db.<collection>.find()
```

Query filter document

```
db.collection.find({ <field1>: <value1>, ... })
```

Specifying query operators

```
db.<collection>.find({ <field1>: { <operator1>: <value1> }, ... })
```

A query filter document is used to explicitly define the expressions that limit the query to the subset of results you want to return within the collection.

The query filter document is similar to standard JSON and consists of field-value/key-value expressions.

If you use an empty query filter document `{}` then the `find()` will return all of the documents in the collection.





# MQL Find()

`db.<collection>.find()`

Query filter document

`db.collection.find({ <field1>: <value1>, ... })`

Specifying query operators

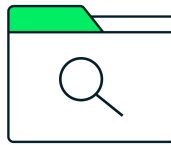
`db.<collection>.find({ <field1>: { <operator1>: <value1> }, ... })`

In the query filter document you can also use query operators to specify specific conditions like defining a range or exact match for the value(s) you want to limit your query to return.



## MQL Find(): Important Note

The collection is implicit in MQL based on the query's criteria.



In terms of MQL's find, it is important to recap that the specific collection is implicit in the query criteria in the syntax of the find().

# Quiz





# Quiz

**Which of the following is true for MongoDB MQL find()?**

- ☐ A. The collection to be queried is part of the query filter document
- ☐ B. Query operators can be used to specify conditions in the query filter document
- ☐ C. Multiple collections can be queried within a single MQL find()



# Quiz

Which of the following is true for MongoDB MQL find()?

- ☐ A. The collection to be queried is part of the query filter document
- ☒ B. Query operators can be used to specify conditions in the query filter document
- ☐ C. Multiple collections can be queried within a single MQL find()

INCORRECT: The collection to be queried is part of the query filter document - The collection is explicitly specified as part of the find syntax, specifically `db.collection.find()` rather than as part of the query filter document.

CORRECT: Query operators can be used to specify conditions in the query filter document - This is correct.

INCORRECT: Multiple collections can be queried within a single MQL find() - The syntax of MQL find() limits the query to a single specified collection



# Quiz

Which of the following is true for MongoDB MQL find()?

- ☐ A. The collection to be queried is part of the query filter document
- ☒ B. Query operators can be used to specify conditions in the query filter document
- ☐ C. Multiple collections can be queried within a single MQL find()

*This is incorrect. The collection is explicitly specified as part of the find syntax, specifically `db.collection.find()` rather than as part of the query filter document.*

INCORRECT: The collection to be queried is part of the query filter document - This is incorrect. The collection is explicitly specified as part of the find syntax, specifically `db.collection.find()` rather than as part of the query filter document.



# Quiz

Which of the following is true for MongoDB MQL find()?

- ☐ A. The collection to be queried is part of the query filter document
- ☒ B. Query operators can be used to specify conditions in the query filter document
- ☐ C. Multiple collections can be queried within a single MQL find()

*This is correct. Query operators can be used to specify conditions within the query filter document.*

CORRECT: Query operators can be used to specify conditions in the query filter document - This is correct. Query operators can be used to specify conditions within the query filter document.



# Quiz

Which of the following is true for MongoDB MQL find()?

- ☐ A. The collection to be queried is part of the query filter document
- ☒ B. Query operators can be used to specify conditions in the query filter document
- ☐ C. Multiple collections can be queried within a single MQL find()

*This incorrect. The syntax of MQL find() limits the query to a single specified collection.*

INCORRECT: Multiple collections can be queried within a single MQL find() - This is incorrect. The syntax of MQL find() limits the query to a single specified collection



The background of the slide is a dark green color. On the right side, there is a lighter green abstract shape that resembles a stylized leaf or a drop. In the top right corner of this lighter green shape, there is a small, dark green leaf icon.

# MQL Query Exercise



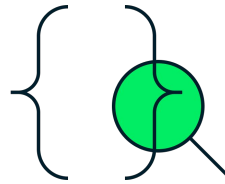
# Using MQL Exercise

Let's use MQL to find people whose age is below 30.

We'll walk through:

- Syntax.
- Creating the data and then querying the database to get the results.

We'll do these via the MongoDB Shell.



Let's take a few minutes to explore MQL find with a hands-on session, we'll walk through the syntax, create the data and then query the database to get the results.

Follow along using the [MongoDB Shell](#). If you prefer to use a GUI to interact with your data, please use [MongoDB Compass](#) or the [MongoDB Atlas UI](#).



```
db.people.find(  
  {  
    "age": { $lt: 30 }  
  }  
)
```

Let's find the  
fields in our  
documents

The first two aspect of the MQL find syntax are to identify the collection we'll use, in this case 'people' and the find function itself.

```
db.people.find(  
  { age: { $lt: 30 } }  
)
```



```
db.people.find(  
  {  
    "age": { $lt: 30 }  
  }  
)
```

With the age  
field value being  
\$lt (less than) 30

We will use a query filter document with the find function to limit the query to documents where the age field has a value less than 30.

```
db.people.find(  
  { age: { $lt: 30 } }  
)
```



# MQL: Exercise

Let's insert some real data on people!

```
>>> db.people.insertMany([{"user_id": "Eoin", "age": 29,
"Status": "A"}, {"user_id": "Daniel", "age": 25, "Status":
"A", "Country": "USA" }])

...
{
  acknowledged : true,
  insertedIds : [
    ObjectId(5f11c32ae89fad25d8875c1c),
    ObjectId(5f11c32ae89fad25d8875c1d)
  ]
}
```

You should cut and paste the following command directly from the slide or from these notes into the prompt (indicated by >>>). Once they have been inserted you will see the following output on the screen. The result should be similar with the exception that the ObjectIds will differ.

```
db.people.insertMany([{"user_id": "Eoin", "age": 29,
"Status": "A"}, {"user_id": "Daniel", "age": 25,
"Status": "A", "Country": "USA" }])
```

See:

<https://docs.mongodb.com/manual/reference/method/db.collection.insertMany/>



# MQL: Exercise

Let's find people whose age is below 30

```
>>> db.people.find({"age":{"$lt":30}})

{ "_id":
  ObjectId(5f11c32ae89fad25d8875c1c),
  "user_id" : "Eoin", "age" : 29, "Status"
  : "A"}

{ "_id":
  ObjectId(5f11c32ae89fad25d8875c1d),
  "user_id" : "Daniel", "age" : 25,
  "Status" : "A", "Country" : "USA" }
```

Now to use the MQL find() to query the data we've just added to the database. You can copy it from the slide or from the notes here

```
db.people.find({"age":{"$lt":30}})
```

See: <https://docs.mongodb.com/manual/reference/method/db.collection.find/>



# MQL: Exercise

Find the people whose age is great than (\$gt) 25.

Using the same window, change **<a>** to the operator for greater than.

Change **<b>** to the value necessary to find all people whose age is greater than 25.

```
>>>
db.people.find({"age":{"<a>":"<b>}})

{ "_id":
  ObjectId(5f11c32ae89fad25d8875c1c),
  "user_id" : "Eoin", "age" : 29, "Status"
: "A"}
```

It's your turn to find people whose age is greater than (\$gt) 25, you will need to change <a> to the operator for greater than and you will need to change <b> to the value necessary to get all the people whose age is greater than 25.

The result in the code block is what you should see if you are successful.

See: <https://docs.mongodb.com/manual/reference/method/db.collection.find/>



Create Read  
Update Delete  
(CRUD)





**insertOne():** Insert one document into a collection.

**insertMany():** Insert an array of documents into a collection.

**writeConcern:** Sets the level of acknowledgment requested from MongoDB for write operations.

**ordered:** For insertMany() there is an additional option for controlling whether the documents are inserted in ordered or unordered fashion.

# MQL Create

In this section, we'll explore a little more around the CRUD functions in MQL.

We'll start with the Create functions of `insertOne()` and `insertMany()`. In our previous hands-on example, we've already use `insertMany()` to populate the data into our collection.

The main difference is `insertMany()` takes an array of documents whilst `insertOne()` only takes a single document.

Both functions can take an optional `writeConcern` parameter, whilst `insertMany()` can also optionally take an `ordered` parameter which determines if the documents must be inserted in the order they are present in the array, the default is to insert documents ordered in the way they are present in the array.

# Example: MQL Create



```
>> db.cows.insertOne({name: "daisy", milk: 8}, {writeConcern: {w: "majority"}})
{
  "acknowledged" : true,
  "insertedId" : ObjectId("5f4e0c5b2d4b45b7f11b6d50")
}
>>> db.cows.insertMany([{name: "buttercup", milk: 9}, {name: "rose", milk: 7}],
{writeConcern: {w: "majority"}, ordered: false})
{
  "acknowledged" : true,
  "insertedIds" : [
    ObjectId("5f4e0ce52d4b45b7f11b6d51"),
    ObjectId("5f4e0ce52d4b45b7f11b6d52")
  ]
}
```

Here's an example of firstly using insertOne and then using insertMany.



**find():** Selects documents and returns cursor.

**findOne():** Returns first document that satisfies criteria.

**findAndModify():** Modifies and returns a single document.

**findOneAndDelete():** Deletes & returns the deleted document.

**findOneAndUpdate():** Updates a single document.

**findOneAndReplace():** Replaces a single document.

# MQL Read

We have already covered the R/Read with the find() function, however there are several other find functions that are also worth flagging here.

find(): returns a cursor to the results for the query document. The query document holds the criteria you want to filter the documents on and the projection document further refines the criteria by selecting specific fields to be returned.

findAndModify(): finds and then updates atomically a single document.

findOne() finds and returns a single document.

findOneAndXXX(): includes findOneAndDelete(), findOneAndUpdate(), findOneAndReplace(). These perform the related action again on a single document returning the original document (whether deleted, updated, or replaced). In the case of findOneAndUpdate() and findOneAndReplace() you can specify `returnNewDocument: true` to return the updated document or the replaced document (the new document).

findOneAndUpdate() and findOneAndReplace() perform a read and an atomic write.

# Example: MQL Read



```
>> db.cows.find({name: "daisy", milk: 8})

{ "_id" : ObjectId("5f4e0c5b2d4b45b7f11b6d50"), "name" : "daisy", "milk" : 8 }

>>> db.cows.findAndModify({query: {name: "daisy", milk: 8}, update: { $set: {milk:
12} }})

{
  "_id" : ObjectId("5f4e0c5b2d4b45b7f11b6d50"),
  "name" : "daisy",
  "milk" : 8
}

>>> db.cows.find({name: "daisy", milk: 12})

{ "_id" : ObjectId("5f4e0c5b2d4b45b7f11b6d50"), "name" : "daisy", "milk" : 12 }
```

On the previous example for “MQL Create” we added some documents into the ‘cows’ collection. Let’s find one of these documents where the cow’s name is ‘daisy’ and the milk value is ‘8’.

Now let’s modify that specific document to increase the value for the ‘milk’ field to ‘12’. In the findAndModify() return we see the old document was returned.

To verify the changes, let’s query for the updated document for the cow ‘daisy’ and see the changes applied to it.



`updateOne()`: Update one document into a collection

`updateMany()`: Update an array of documents into a collection.

## MQL Update

Update in similar fashion to Create has two functions, `updateOne` and `updateMany`, which operate on a single document and on an array of documents respectively.

Both functions take a filter document which specifies which documents to be updated. If the filter document is empty the first document in the collection is updated. These functions also take a second document as an argument which contains the various options that can be configured.

The default behaviour is not to insert a new document if the update cannot find a suitable document or documents that match the filter document. It is possible using the `upsert: True` parameter in the second document supplied to the function (the options document) to set that if a document isn't found that a new one will be inserted (hence the term `upsert` - update and insert).



# Example: MQL Update

```
>>> db.cows.updateOne({name: "daisy", milk: 12},{ $set: {milk: 8} })

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

>>> db.cows.updateMany({}, {$inc: {milk: 1}})

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }

>>> db.cows.find({})

{ "_id" : ObjectId("5f4e0c5b2d4b45b7f11b6d50"), "name" : "daisy", "milk" : 9 }

{ "_id" : ObjectId("5f4e0ce52d4b45b7f11b6d51"), "name" : "buttercup", "milk" : 10
}

{ "_id" : ObjectId("5f4e0ce52d4b45b7f11b6d52"), "name" : "rose", "milk" : 8 }
```

Let's revert the change we made in the MQL Read slide where we set the milk to 12 from 8 for the cow called daisy. We can see in the output of the updateOne, that it both found and updated 1 document.

Let's update all of the documents in the cows collection and increment/add their milk fields by 1.

Let's then use a find to look at the result for a subset of the documents, we can see that for cow named daisy, we'd set the milk field to 8 and then this was incremented by 1 so it's now showing 9.



`deleteOne()`: Deletes one document from a collection.

`deleteMany()`: Deletes many documents from a collection.

`writeConcern`: Sets the level of acknowledgment requested from MongoDB for write operations.

## MQL Delete

In terms of the D/Delete in Crud, both functions take a filter document which specifies which documents to be delete, if the filter document is empty the first document in the collection is updated. These functions also take a second document as an argument which contains the various options that can be configured.



# Example: MQL Delete

```
>>> db.cows.deleteOne({milk: 9})  
  
{ "acknowledged" : true, "deletedCount" : 1 }  
  
>>> db.cows.deleteMany({}, {writeConcern: {w:  
"majority"}})  
  
{ "acknowledged" : true, "deletedCount" : 2 }
```

Let's again use the data on cows that we have entered and test the `deleteOne` and `deleteMany()` functions. We can see in the `deletedCount` field for both functions how many documents they deleted for the respective operation.





Create: `insertMany()`

Update: `updateMany()`

Delete: `deleteMany()`

The Many variants apply to multiple records, however, `insertMany()` takes an Array of documents.

MQL: `xxxMany ( )`

We have covered `insertMany` in Create, `updateMany` in Update, and `deleteMany` in Delete.

The main point to flag here is that the `manyXXX()` variants of the functions apply that function to many documents. The only difference with the `insertMany` to these is that it takes an array of documents rather than applies the functions to the documents identified by the filter document.

There are many options and behaviors to these functions that are covered in more depth in their corresponding page in the MongoDB documentation. For instance, if you encounter an error whilst inserting with `insertMany()` it will stop on this error (the first error encountered) rather than continuing.

# CRUD: MQL Exercise



```
>>> cowCol = db.getCollection("cow")
Test.cow
>>> cowCol.drop()
>>> for(c=0;c<10;c++) {
cowCol.insertOne({ name: "daisy", milk: c} )
}

{
  acknowledged : true,
  insertedIds  : ObjectId(5f2aefa8fde88235b959f0b1e),
}
```

Let's create the cow collection, dropping it if it's already there (`cowCol.drop()`) and starting fresh.

The for loop inserts 10 documents each with the name field equal to "daisy" and a varying value for the milk field.

We can use the following code to input some data into our database.

```
cowCol = db.getCollection("cow")
cowCol.drop()
for(c=0;c<10;c++) {
cowCol.insertOne({ name: "daisy", milk: c} )
}
```

# CRUD: MQL Exercise



Using the shell window, change **<A>** to update the first document with the milk field value of 6 as the document we want to update.

Change **<B>** to the highlighted field name shown in the find() query below.

```
>>> cowCol.findAndModify(({ query: { milk: { $gt: <A> } }, sort: { milk: 1 },
update: { $set: { <B>: true } } }))

{ "_id" : ObjectId("5f4e50da2d4b45b7f11b6d76"), "name" : "daisy", "milk" : 6,
  "expected_milk" : 5 }

>> cowCol.find({"sell" : true})

{ "_id" : ObjectId("5f4e50da2d4b45b7f11b6d76"), "name" : "daisy", "milk" : 6,
  "expected_milk" : 5, "sell": true }
```

In this exercise we will use findAndModify ( ) to update a document.

In the same window, you should replace **<A>** to update the first document with the milk field value of 6 as the document we want to update

You should change **<B>** to the field which represents milk in the document.

The result should be similar (the ObjectIds will differ) will firstly be the original document as findAndModify needs the option new: true to return the updated document rather than the original document.

The second find output is the updated document and it has the additional field “sell” that we used the \$set operator in the findAndModify query to update the document to add the field with a value of true.

```
cowCol.findAndModify({ query: { milk: { $gt: 5 } }, sort: {
milk: 1 }, update: { $set: { sell: true } } })
cowCol.find({"sell" : true})
```

See:

<https://docs.mongodb.com/manual/reference/method/db.collection.findAndModify/>

# CRUD: MQL Exercise



Using the shell window again, change **<A>** to the greater than operator

Change **<B>** to the operator we will use to set the value of the field in the documents we will update.

As we already updated one of the eligible documents, only 3 are modified.

```
>>> cowCol.updateMany({ milk: { <A>: 5 } }, { <B>: { sell: true } })  
  
{ "acknowledged" : true, "matchedCount" : 4, "modifiedCount" : 3 }
```

Now we will use `updateMany ( )` to update 3 documents

In the same window, you should replace **<A>** to the greater than operator.

You should change **<B>** to the operator we will use to set the value of the field in the documents we will update.

The reason only 3 of the 4 matched documents are modified is because we modified one in our last slide with the `findAndModify()` operation.

```
cowCol.updateMany({ milk: { <A>: 5 } }, { <B>: { sell: true }  
})
```

See:

<https://docs.mongodb.com/manual/reference/method/db.collection.findAndModify/>

# Quiz





# Quiz

Which of the following are true for MongoDB MQL?

- ☐ A. All the CRUD functions in MQL take a filter document
- ☐ B. findAndModify() is an atomic operation
- ☐ C. findOneAndUpdate() and findOneAndReplace() perform a read and an atomic write
- ☐ D. find() is the only CRUD function which returns a cursor



# Quiz

Which of the following are true for MongoDB MQL?

- ☒ A. All the CRUD functions in MQL take a filter document
- ☒ B. findAndModify() is an atomic operation
- ☒ C. findOneAndUpdate() and findOneAndReplace() perform a read and an atomic write
- ☒ D. find() is the only CRUD function which returns a cursor

INCORRECT: All the CRUD functions in MQL take a filter document - insertOne and insertMany do not take a filter document

CORRECT: findAndModify() is an atomic operation

CORRECT: findOneAndUpdate() and findOneAndReplace() perform a read and an atomic write

CORRECT: find() is the only CRUD function which returns a cursor - true, all the others return a document rather than a cursor

# Quiz



Which of the following are true for MongoDB MQL?

- ☒ A. All the CRUD functions in MQL take a filter document
- ☒ B. findAndModify() is an atomic operation
- ☒ C. findOneAndUpdate() and findOneAndReplace() perform a read and an atomic write
- ☒ D. find() is the only CRUD function which returns a cursor

*This incorrect. The insertOne and insertMany functions do not take a filter document.*

INCORRECT: All the CRUD functions in MQL take a filter document - This is incorrect. The insertOne and insertMany functions do not take a filter document



# Quiz



Which of the following are true for MongoDB MQL?

- ☒ A. All the CRUD functions in MQL take a filter document
- ☒ B. findAndModify() is an atomic operation
- ☒ C. findOneAndUpdate() and findOneAndReplace() perform a read and an atomic write
- ☒ D. find() is the only CRUD function which returns a cursor

*This is correct. The findAndModify function is atomic in terms of the modifications to the single document it operates upon.*

CORRECT: findAndModify() is an atomic operation. This is correct. The findAndModify function is atomic in terms of the modifications to the single document it operates upon.

# Quiz



Which of the following are true for MongoDB MQL?

- ☒ A. All the CRUD functions in MQL take a filter document
- ☒ B. findAndModify() is an atomic operation
- ☒ C. findOneAndUpdate() and findOneAndReplace() perform a read and an atomic write
- ☒ D. find() is the only CRUD function which returns a cursor

*This is correct. This is how these functions operate, firstly the read and then the atomic write.*

CORRECT: findOneAndUpdate() and findOneAndReplace() perform a read and an atomic write. This is correct. This is how these functions operate, firstly the read and then the atomic write.

# Quiz



Which of the following are true for MongoDB MQL?

- ☒ A. All the CRUD functions in MQL take a filter document
- ☒ B. findAndModify() is an atomic operation
- ☒ C. findOneAndUpdate() and findOneAndReplace() perform a read and an atomic write
- ☒ D. find() is the only CRUD function which returns a cursor

*This is correct. All the other functions return a document rather than a cursor.*

CORRECT: findOneAndUpdate() and findOneAndReplace() perform a read and an atomic write. This is correct. This is how these functions operate, firstly the read and then the atomic write.

# MQL Delete



# MQL Delete

Specifically the **db.collection.deleteOne()** and **db.collection.deleteMany()** methods but also **db.collection.drop()**.

To delete all documents from a collection, pass an empty filter document {} to the db.collection.deleteMany() method

Additional methods include db.collection.findOneAndDelete() and db.collection.findAndModify(), both offer a sort option. Deletes are also possible via the db.collection.bulkWrite() method.

We covered the deleteOne and deleteMany methods, these will be used and sufficient for the majority of use. However, if you need to delete all of the documents in a collection then you should look at db.collection.drop(). It has a further advantage that it removes all of the associated indexes related to that collection as well as the documents.



# MQL Delete

Specifically the **db.collection.deleteOne()** and **db.collection.deleteMany()** methods but also **db.collection.drop()**.

To delete all documents from a collection, pass an empty filter document {} to the **db.collection.deleteMany()** method

Additional methods include **db.collection.findOneAndDelete()** and **db.collection.findAndModify()**, both offer a sort option. Deletes are also possible via the **db.collection.bulkWrite()** method.

It is possible to delete all the documents by simply passing an empty filter document {}, however as mentioned it may be more performant to use drop the collection rather than delete all the documents.



# MQL Delete

Specifically the **db.collection.deleteOne()** and **db.collection.deleteMany()** methods but also **db.collection.drop()**.

To delete all documents from a collection, pass an empty filter document {} to the **db.collection.deleteMany()** method

Additional methods include **db.collection.findOneAndDelete()** and **db.collection.findAndModify()**, both offer a sort option. Deletes are also possible via the **db.collection.bulkWrite()** method.

In cases where you want to delete the documents in a sorted order you can use **findOneAndDelete()** or **findAndModify()**.

It is also possible to delete documents using the **bulkWrite()** method, the function takes an array of **bulkWrite** operations where you can control the order of execution if necessary.

The bulk API allows for mixing operations which returns a single result. An example might be a daily processing job where a set of insert, update, and deletions should occur and in that sequence.

# Quiz







# Quiz

**Which of the following are true for MongoDB MQL?**

- A. A non-empty filter document must be given for the deleteXXX() methods
- B. Deletes cannot be sorted
- C. Deletes are atomic operations



# Quiz

**Which of the following are true for MongoDB MQL?**

- ☐ A. A non-empty filter document must be given for the deleteXXX() methods
- ☐ B. Deletes cannot be sorted
- ☒ C. Deletes are atomic operations

INCORRECT: A non-empty filter document must be given for the deleteXXX() methods - {} or an empty document will delete all documents in the collection

INCORRECT: Deletes cannot be sorted - It is possible with db.collection.findOneAndDelete() or with db.collection.findAndModify(), both offer the ability to sort order of the documents for deletion.

CORRECT: Deletes are atomic operations - All write operations in MongoDB are atomic, a delete is a write operation.



# Quiz

Which of the following are true for MongoDB MQL?

✗ A. A non-empty filter document must be given for the deleteXXX() methods

✗ B. Deletes cannot be sorted

✓ C. Deletes are atomic operations

*This incorrect. Using {} or an empty document will delete all documents in the collection.*

INCORRECT: A non-empty filter document must be given for the deleteXXX() methods - {} or an empty document will delete all documents in the collection



# Quiz

Which of the following are true for MongoDB MQL?

✗ A. A non-empty filter document must be given for the deleteXXX() methods

✗ B. Deletes cannot be sorted

✓ C. Deletes are atomic operations

*This incorrect. It is possible with `db.collection.findOneAndDelete()` or with `db.collection.findAndModify()`, both offer the ability to sort order of the documents for deletion.*

INCORRECT: Deletes cannot be sorted - This is incorrect. It is possible with `db.collection.findOneAndDelete()` or with `db.collection.findAndModify()`, both offer the ability to sort order of the documents for deletion.



# Quiz

Which of the following are true for MongoDB MQL?

✗ A. A non-empty filter document must be given for the deleteXXX() methods

✗ B. Deletes cannot be sorted

✓ C. Deletes are atomic operations

*This is correct. All write operations in MongoDB are atomic, a delete is a write operation.*

CORRECT: Deletes are atomic operations - This is correct. All write operations in MongoDB are atomic, a delete is a write operation.



# MQL Query Operators: Comparison

MongoDB's Query Language has a wide set of query operators that help locate and modify data.



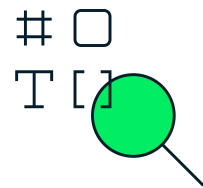
# MQL Query Operators

Comparison

Logical

Element, Array, Evaluation, Bitwise, Comment, Geospatial, and  
Projection

Update and Update Modifiers



In the next section of this lesson, we'll cover MongoDB's MQL query operators in more depth starting firstly with the comparison operators.

MQL's comparison operators provide functionality to compare different BSON types and values.



**\$lt** Exists and less than

**\$lte**: Exists and less than or equal to

**\$gt**: Exists and greater than

**\$gte**: Exists and greater than or equal to

**\$ne**: Does not exist or does but not equal to

**\$eq**: Exists and equal to

**\$in**: Exists and in a set

**\$nin**: Does not exist or not in a set

## MQL Query Operators: Comparison

Looking at MQL's comparison operators in more detail there is a less than, less than equal, greater than, greater than equal, not exist, exists and equals, in, and not in as a subset of the available query operators.

An important aspect of query operators is that you specify an Object with a key that's a comparison and a value.



# MQL Operators Comparison: Example



```
>>> cowCol = db.getCollection("cow")
test.cow
>>> for(c=0;c<10;c++) {
cowCol.insertOne({ name: "daisy", milk: c} )
}
{
  "acknowledged" : true,
  "insertedId"   : ObjectId("5f2aefa8fde88235b959f0b1e")
}
>>> cowCol.find({milk:{$gt:6}})
{"_id": ObjectId("5f2aefa8fde88235b959f0b1c"), "name": "daisy", milk: 7}
{"_id": ObjectId("5f2aefa8fde88235b959f0b1d"), "name": "daisy", milk: 8}
{"_id": ObjectId("5f2aefa8fde88235b959f0b1e"), "name": "daisy", milk: 9}
```

In the example, we see the \$gt (greater than) operator being used with the key 'milk' to return documents whose milk field has a value greater than 6.

# MQL Operator Comparison: Exercise



Let's insert some real data on cows!

```
>>> cowCol = db.getCollection("cow")
test.cow
>>> for(c=0;c<10;c++) {
cowCol.insertOne({ name: "daisy", milk: c} )
}

{
  acknowledged : true,
  insertedIds : ObjectId(5f2aefa8fde88235b959f0b1e),
}
```

Let's create the cow collection, dropping it if it's already there (cowCol.drop()) and starting fresh.

The for loop inserts 10 documents each with the name field equal to "daisy" and a varying value for the milk field.

We can use the following code to input some data into our database.

```
cowCol = db.getCollection("cow")
cowCol.drop()
for(c=0;c<10;c++) {
cowCol.insertOne({ name: "daisy", milk: c} )
}
```

See: <https://docs.mongodb.com/manual/reference/method/db.collection.insertOne/>

# MQL Operator Comparison: Exercise



Find the docs with the field “milk” greater than 6.

```
>>> cowCol.find({milk:{$gt:6}})

{ "_id": ObjectId("5f2aefa8fde88235b959f0b1c"), "name" : "daisy",
  "milk" : 7 }

{ "_id": ObjectId("5f2aefa8fde88235b959f0b1d"), "name" : "daisy",
  "milk" : 8 }

{ "_id": ObjectId("5f2aefa8fde88235b959f0b1e"), "name" : "daisy",
  "milk" : 9 }
```

We can use the `db.collection.find()` method to explore the data we’ve just added to the database.

See: <https://docs.mongodb.com/manual/reference/method/db.collection.insertOne/>

# MQL Operator Comparison: Exercise



Using `updateMany()` to update 3 documents. Using the same window, change `<a>` to the operator for less than or equal. Change `<b>` to the value necessary to find all the documents whose milk is less than or equal to 3.

```
>>> cowCol.find({milk:{<a>:<b>}})
```

```
{ "_id" : ObjectId("5f4e41d12d4b45b7f11b6d67"), "name" : "daisy", "milk" : 0 }  
{ "_id" : ObjectId("5f4e41d12d4b45b7f11b6d67"), "name" : "daisy", "milk" : 1 }  
{ "_id" : ObjectId("5f4e41d12d4b45b7f11b6d67"), "name" : "daisy", "milk" : 2 }  
{ "_id" : ObjectId("5f4e41d12d4b45b7f11b6d67"), "name" : "daisy", "milk" : 3 }
```

You should replace `<A>` with the operator for less than or equal.

You should change `<B>` to the value necessary to find all the documents whose 'milk' field is less than or equal to 3.

See: <https://docs.mongodb.com/manual/reference/method/db.collection.find/>

See: <https://docs.mongodb.com/manual/reference/method/db.collection.find/>



# MQL Query Operators Cont.



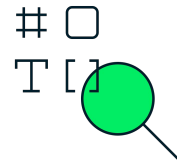
# MQL Query Operators

Comparison

Logical

Element, Array, Evaluation, Bitwise, Comment, Geospatial, and Projection

Update and Update Modifiers



Diverse Search Parameters

We'll move on to look at the second type of MQL Query Operators, the logical operators.

The MQL logical operators allow for combinations of other query operators to create more complex conditions.



**\$or** Match either of two or more values

**\$not** Used with other operators to negate

**\$nor** Match neither of two or more values

**\$and** Match both of two or more values

## MQL Query Operators: Logical

MQL offers various logical query operators including \$or, \$not, \$nor, as well as \$and. \$or matches two or more values, \$not can be used with other operators for negation, \$nor can match neither of two or more values, lastly \$and can match both of two or more values.



# Logical Operator: Example One

```
>>> cowCol.find({$and: [ {milk: { $gt: 6 } },  
  {milk: { $lt: 9 } } ] })  
  
{"_id": ObjectId("5f4e41d12d4b45b7f11b6d6e"),  
  "name": "daisy", "milk": 7 }  
  
{"_id": ObjectId("5f4e41d12d4b45b7f11b6d6f"),  
  "name": "daisy", "milk": 8 }
```





# Logical Operator: Example Two

```
>>> cowCol.find({milk:{$not:{$gt:6}}})

{"_id": ObjectId("5f4e41d12d4b45b7f11b6d67"), "name": "daisy", "milk": 0 }
{"_id": ObjectId("5f4e41d12d4b45b7f11b6d68"), "name": "daisy", "milk": 1 }
{"_id": ObjectId("5f4e41d12d4b45b7f11b6d69"), "name": "daisy", "milk": 2 }
{"_id": ObjectId("5f4e41d12d4b45b7f11b6d6a"), "name": "daisy", "milk": 3 }
{"_id": ObjectId("5f4e41d12d4b45b7f11b6d6b"), "name": "daisy", "milk": 4 }
{"_id": ObjectId("5f4e41d12d4b45b7f11b6d6c"), "name": "daisy", "milk": 5 }
{"_id": ObjectId("5f4e41d12d4b45b7f11b6d6d"), "name": "daisy", "milk": 6 }
```



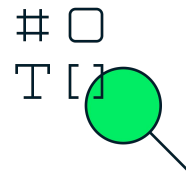
# MQL Query Operators

Comparison

Logical

Element, Array, Evaluation, Bitwise, Comment, Geospatial, and Projection

Update and Update Modifiers



Diverse Search Parameters

Thirdly, let's look at a wide range of operators from element to array operators but there are also geospatial and projection operators amongst the wide range of functionality available in MQL.



# MQL Query Operators: Various categories

**\$exists:** Match documents that have the specific field

**\$type:** Selects documents if a field is of the specified type

**\$elemMatch:** Selects documents if element in the array field matches all the specified \$elemMatch conditions. Limits the contents of an <array> field from the query results to contain only the first element matching the conditions.

**\$comment:** Adds a comment to a query predicate

**\$expr:** Allows use of aggregation expressions within the query language

**\$geoWithin:** Selects geometries within a bounding GeoJSON geometry, requires either a 2dsphere or a 2d index

**\$:** Selects and returns first match in array that meets condition

**\$slice:** Limits the number of elements projected from an array

Here's a subset of the specific operators available from the Element, Array, Evaluation, Bitwise, Comment, Geospatial, and Projection categories. The documentation site for MongoDB contains the full list along with any caveats for their usage.

\$exists is a useful operator that ensures a specific field exists, \$type selects documents based on whether or not the field is of a specific type, \$elemMatch selects documents if an element in the array field matches it's specified conditions, \$comment is another useful operator as you can add a comment to your query which is logged in the profile log.

Looking at the set of expressions start at the top with \$expr which provides the ability to use aggregation expressions within MQL queries. There are several geospatial operators but we'll just look at \$geoWithin which allows for queries bound by a GeoJSON geometry. The \$ operator returns the first match in an array that satisfies the condition. Lastly, \$slice limits the number of elements that are projected from an array for the results.

These operators span a wide set of functionality. A useful operator is the \$comment which sends the comment to the profile log when the MQL statement is executed. This can be helpful when debugging or tracing the performance of your queries.

\$expr can built help build more complex aggregation expressions within MQL.



# MQL Query Operators: \$expr

**\$expr** Example comparing two fields in the same collection

In this example, we again use cows and milk, however we add the **'expected\_milk'** field. This field is set one above or below the 'milk' field, above if 'milk' is an odd number or below if 'milk' is even.

```
>>> cowCol = db.getCollection("cow")
Test.cow
>>> cowCol.drop()
>>> for(c=1;c<10;c++) {
    a = 0
    if(c%2==0) {
        a = c-1
    } else {
        a = c+1
    }
    cowCol.insertOne({ name: "daisy",
milk: c, expected_milk: a } )
}
>>> cowCol.find( { $expr: { $gt: [ "$milk"
, "$expected_milk" ] } } ).count()
```

Let's take a deeper look at the \$expr operator as it allows for a huge number of Aggregation Framework operators to be utilized by MQL. Specifically, it allows for two fields to be compared with the \$gt operator.

In the example on the slide, we again use cows and milk, however we add the 'expected\_milk' field. This field is set one above or below the 'milk' field, above if 'milk' is an odd number or below if 'milk' is even.

```
cowCol = db.getCollection("cow")
cowCol.drop()
for(c=1;c<10;c++) {
    a = 0
    if(c%2==0) {
        a = c-1
    } else {
        a = c+1
    }
    cowCol.insertOne({ name: "daisy", milk: c, expected_milk:
a } )
}
cowCol.find( { $expr: { $gt: [ "$milk" , "$expected_milk" ] } }
).count()
```



## Here's the full result for our previous example...

```
>>> cowCol.find({ $expr: { $gt: [ "$milk" , "$expected_milk" ] } } )

{ "_id": ObjectId(5f351f10dcb672556f30f5fa), "name" : "daisy", "milk" : 2,
  "expected_milk" : 1 }

{ "_id": ObjectId(5f351f10dcb672556f30f5fc), "name" : "daisy", "milk" : 4,
  "expected_milk" : 3 }

{ "_id": ObjectId(5f351f10dcb672556f30f5fe), "name" : "daisy", "milk" : 6,
  "expected_milk" : 5 }

{ "_id": ObjectId(5f351f10dcb672556f30f600), "name" : "daisy", "milk" : 8,
  "expected_milk" : 7 }
```

Here's the full result of our previous example using \$expr with another operator (\$gt) to compare two fields within the same document. Specifically, it returns all the documents where the field milk is greater than the field expected\_milk. You can try this yourself in the shell.



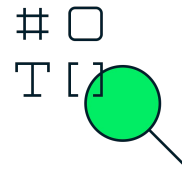
# MQL Query Operators

Comparison

Logical

Element, Array, Evaluation, Bitwise, Comment, Geospatial, and Projection

Update and Update Modifiers



The final set of query operators will explore in this lesson are the update and update modifiers. These are available for any of the update operations in MQL.



# MQL Query Operators: Various Categories

**\$inc:** Increments the specific field by the specified amount

**\$currentDate:** Sets the field to the current date, either as a Date or as a Timestamp

**\$set:** Sets the value of the field to the specified value

**\$setOnInsert:** Similar to \$set but only performs this when there is an insert of a new document, it won't update existing documents if present

**\$rename:** Changes a field's name to the specified name

**\$max:** Only updates the field if the value specified is greater than the existing value

**\$addToSet:** Selects and returns first match in array that meets condition

**\$push:** Adds an item to an array

**\$each:** Modifies the \$push and \$addToSet operators to append multiple items for array updates

In this section we'll look at a selection of the available MQL query operators for update and update modifiers. These operators are available for use in update operations, e.g. in [db.collection.update\(\)](#) and [db.collection.findAndModify\(\)](#).

Firstly, \$inc which provides a useful ability to increment fields by a specified amount. \$currentDate can be used to set a field to the current data. \$set is a very useful operator which sets the field to the particular value. \$setOnInsert is similar to \$set but only performs the 'set' part when there is an insert of a new document, if there is an existing document it will not update it.

# Query Operators: Exercise



```
>>> cowCol = db.getCollection("cow")
Test.cow

>>> cowCol.drop()

>>> for(c=0;c<10;c++) {cowCol.insertOne({ name: "daisy", milk: c})
}
{
  acknowledged : true,
  insertedIds :   ObjectId(5f2aefa8fde88235b959f0b1e),
}
```

Let's use the MongoDB Shell to try out this example.

Let's create the cow collection, dropping it if it's already there (cowCol.drop()) and starting fresh.

The for loop inserts 10 documents each with the name field equal to "daisy" and a varying value for the milk field.

We can use the following code to input some data into our database.

```
cowCol = db.getCollection("cow")
cowCol.drop()
for(c=0;c<10;c++) {cowCol.insertOne({ name: "daisy", milk: c} )
}
```

See: <https://docs.mongodb.com/manual/reference/method/db.collection.insertOne/>





```
>>> cowCol.find({milk:{$gt:8}})

{ "_id":
  ObjectId(5f2aefa8fde88235b959f0b1e),
  "name" : "daisy", "milk" : 9 }
```

```
>>> cowCol.update({ milk: 9 }, {
  $set: { "name" : "rose" },
  $setOnInsert: { milk: 10 }}, {
  upsert: true } );

>>> cowCol.find({milk:{$gt:8}})

{ "_id":
  ObjectId(5f2aefa8fde88235b959f0b1e),
  "name" : "rose", "milk" : 9 }
```

## Find the docs where milk is greater than 8

Let's query the data we just inputted to find the cow with the highest milk value

```
cowCol.find({milk:{$gt:8}})
```

Let's update this cow's record to change the cow's name to 'rose'.

```
cowCol.update({ milk: 9 }, { $set: { "name" : "rose" },
  $setOnInsert: { milk: 10 }}, { upsert: true } );
```

Let's do the same search again to see the update to the document.

```
cowCol.find({milk:{$gt:8}})
```

See: <https://docs.mongodb.com/manual/reference/method/db.collection.update>



## Using updateMany() to update 3 documents

Using the same window, change **<a>** to the operator for less than or equal to.

Change **<b>** to the value necessary to find all the documents whose milk is less than or equal to 3.

```
>>> cowCol.find({milk:{<a>:<b>}})

{ "_id" :
  ObjectId("5f4e41d12d4b45b7f11b6d67"),
  "name" : "daisy", "milk" : 0 }

{ "_id" :
  ObjectId("5f4e41d12d4b45b7f11b6d68"),
  "name" : "daisy", "milk" : 1 }

{ "_id" :
  ObjectId("5f4e41d12d4b45b7f11b6d69"),
  "name" : "daisy", "milk" : 2 }

{ "_id" :
  ObjectId("5f4e41d12d4b45b7f11b6d6a"),
  "name" : "daisy", "milk" : 3 }
```

In this exercise and in the same window, you should replace **<A>** with the operator for less than or equal.

You should change **<B>** to the value necessary to find all the documents whose 'milk' field is less than or equal to 3.

See: <https://docs.mongodb.com/manual/reference/method/db.collection.update>

# Quiz





# Quiz

**Which of the following are true for MongoDB MQL?**

- A. \$in and \$nin both use sets to determine membership or not
- B. \$gt and \$gte will both check of documents with fields exceeding or greater than the specified criteria
- C. \$eq and \$ne check the value is equal or not equal to the specified criteria



# Quiz

**Which of the following are true for MongoDB MQL?**

- ✓ A. \$in and \$nin both use sets to determine membership or not
- ✓ B. \$gt and \$gte will both check of documents with fields exceeding or greater than the specified criteria
- ✗ C. \$eq and \$ne check the value is equal or not equal to the specified criteria

CORRECT: \$in and \$nin both use sets to determine membership or not

CORRECT: \$gt and \$gte will both check of documents with fields exceeding or greater than the specified criteria

INCORRECT: \$eq and \$ne check the value is equal or not equal to the specified criteria - Both \$eq and \$ne further check for the exists or not of the specified criteria



# Quiz

Which of the following are true for MongoDB MQL?

- ✓ A. \$in and \$nin both use sets to determine membership or not
- ✓ B. \$gt and \$gte will both check of documents with fields exceeding or greater than the specified criteria
- ✗ C. \$eq and \$ne check the value is equal or not equal to the specified criteria

*This is correct. Both the \$in and the \$nin functions in MQL use a set to determine the membership or not for a particular piece of data.*

CORRECT: \$in and \$nin both use sets to determine membership or not



# Quiz

Which of the following are true for MongoDB MQL?

- ✓ A. \$in and \$nin both use sets to determine membership or not
- ✓ B. \$gt and \$gte will both check of documents with fields exceeding or greater than the specified criteria
- ✗ C. \$eq and \$ne check the value is equal or not equal to the specified criteria

*This is correct. The logic checks in both the \$gt and the \$gte will verify whether fields exceed (are greater than) the specified criteria.*

CORRECT: \$gt and \$gte will both check of documents with fields exceeding or greater than the specified criteria - This is correct. The logic checks in both the \$gt and the \$gte will verify whether fields exceed (are greater than) the specified criteria.



# Quiz

Which of the following are true for MongoDB MQL?

- ✓ A. \$in and \$nin both use sets to determine membership or not
- ✓ B. \$gt and \$gte will both check of documents with fields exceeding or greater than the specified criteria
- ✗ C. \$eq and \$ne check the value is equal or not equal to the specified criteria

*This incorrect. Both \$eq and \$ne further check for the exists or not of the specified criteria.*

INCORRECT: \$eq and \$ne check the value is equal or not equal to the specified criteria - This is incorrect. Both \$eq and \$ne further check for the exists or not of the specified criteria





# Quiz

**Which of the following are true for MongoDB MQL?**

- ☐ A. \$and works with multiple (more than two) operators.
- ☐ B. \$and works with multiple (more than two) fields.
- ☐ C. \$or can be nested.
- ☐ D. \$set will not create new fields when you specify multiple field-value pairs.



# Quiz

Which of the following are true for MongoDB MQL?

- ✓ A. \$and works with multiple (more than two) operators.
- ✓ B. \$and works with multiple (more than two) fields.
- ✓ C. \$or can be nested.
- ✗ D. \$set will not create new fields when you specify multiple field-value.

*This is correct. The \$and operator allows you to use two or more operators together.*

CORRECT: \$and works with multiple (more than two) operators - This is correct. The \$and operator allows you to use two or more operators together.



# Quiz

Which of the following are true for MongoDB MQL?

- ✓ A. \$and works with multiple (more than two) operators.
- ✓ B. \$and works with multiple (more than two) fields.
- ✓ C. \$or can be nested.
- ✗ D. \$set will not create new fields when you specify multiple field-value pairs.

*This is correct. The \$and operator allows you to work with multiple fields in a MQL query.*

CORRECT: \$and works with multiple (more than two) fields - This is correct. The \$and operator allows you to work with multiple fields in a MQL query.



# Quiz

Which of the following are true for MongoDB MQL?

- ✓ A. \$and works with multiple (more than two) operators.
- ✓ B. \$and works with multiple (more than two) fields.
- ✓ C. \$or can be nested.
- ✗ D. \$set will not create new fields when you specify multiple field-value pairs.

*This is correct. It can be nested to provide more complex logical conditions.*

CORRECT: \$or can be nested - This is correct. It can be nested to provide more complex logical conditions



# Quiz

Which of the following are true for MongoDB MQL?

- ✓ A. \$and works with multiple (more than two) operators.
- ✓ B. \$and works with multiple (more than two) fields.
- ✓ C. \$or can be nested.
- ✗ D. \$set will not create new fields when you specify multiple field-value pairs.

*This incorrect. \$set will create new fields in single or multiple field-value pairs if they do not already exist.*

INCORRECT: \$set will not create new fields when you specify multiple field-value pairs - This is incorrect. \$set will create new fields in single or multiple field-value pairs if they do not already exist

## Continue Learning!



[MongoDB University](#) has free self-paced courses and labs ranging from beginner to advanced levels.

## GitHub Student Developer Pack



Sign up for the [MongoDB Student Pack](#) to receive \$50 in Atlas credits and free certification!

This concludes the material for this lesson. However, there are many more ways to learn about MongoDB and non-relational databases, and they are all free! Check out [MongoDB's University](#) page to find free courses that go into more depth about everything MongoDB and non-relational. For students and educators alike, MongoDB for Academia is here to offer support in many forms. Check out our [educator resources](#) and join the Educator Community. Students can receive \$50 in Atlas credits and free certification through the [GitHub Student Developer Pack](#).