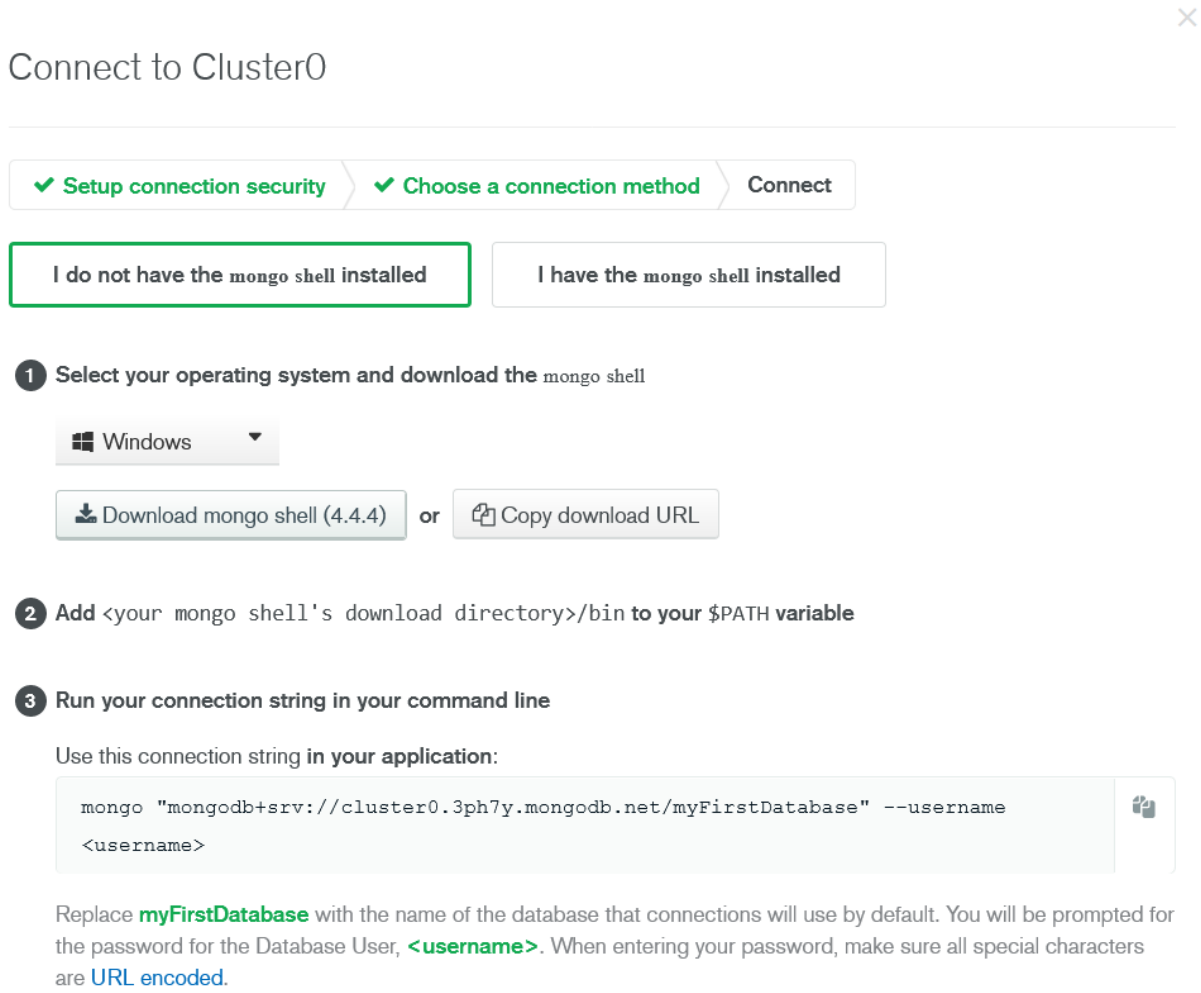
MongoDB Access through CLI

After creating a Cluster follow these steps to access MongoDB through CLI

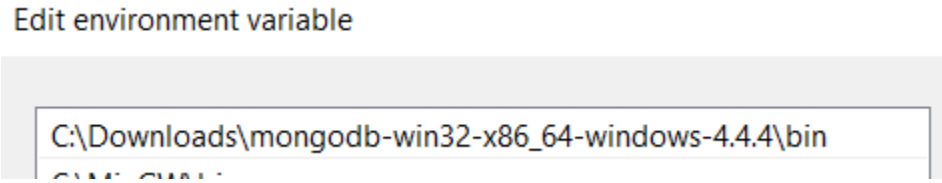
1) Connecting with Mongo Shell

- If you do not have mongo shell previously installed follow the instructions under **Connect -> Connect with mongo shell**



1. Download the mongo shell from here

2. Add <your mongo shell's download directory>/bin to your $PATH variable (User variable)



3. Copy the connection string as shown in the image in **step-3** and paste in your command line and press enter. Replace <username> with user-id of the DATABASE USER

4. Type the password of the DATABASE USER and press enter

NOTE : All special characters of your password should be URL encoded

On Successful setup you will see something like this on your command line

```
MongoDB Enterprise atlas-kdeov2-shard-0:PRIMARY>
```

2.) CRUD operations - Create , **R**ead , **U**pdate , **D**elete.

CREATE:

1. We can create a collection by the following command on command prompt

```
db.createCollection('student')
```

Insert the following data into the collection

```
db.student.insert({
```

```
        student_id: "25",
        name: "Mike",
        course: {
          courseName: "NoSql",
          duration: "4 months"
        }
    })
```

You can see the output under the collection tab in MongoDB Atlas.
Notice the creation of the **_id** field for each inserted document.

```
{
        "_id" : ObjectId("6050f67cc3fd5daeba4a3d2f"),
        "student_id" : "25",
        "name" : "Mike",
        "course" : {
                "courseName" : "NoSql",
                "duration" : "4 months"
        }
}
```

show collections is another useful command to view all the collections created.

db.myCollection.insertMany() , db.myCollection.insertOne() can be used to insert multiple,single documents.

READ:

Read operations retrieves documents from a collection; i.e. queries a collection for documents. Mongo provides **db.myCollection.find().**
This is equivalent to **select * from myCollection**
Use **.pretty()** to get the output in indented form

The following command will give all the students with the **name "Mike"**

```
db.student.find(
 { name : "Mike" }
).pretty()
```

We can also put conditions while searching
The following command will give all the students with **student_id > 25**

```
db.student.find(
 { student_id : {$gt : 25} }
).pretty()
```

Mongo also supports other operators like `$in,$lt,$ne` The full list can be found here

In order to see all the data inside the collection (student) type the following command

```
db.student.find().pretty()
```

Update:

Update operations are provided by the **db.collections.updateOne()** and **db.collection.updateMany()** functions

The following command will update **name** of all the students having **name "Mike" to "Albert"**

```
db.student.updateMany(
     { name : "Mike" },
     { $set : { name : "Albert" } }
)
```

In order to update only a single student you can use **db.student.updateOne()**

Delete:

Two functions are there : **db.collections.deleteOne() or deleteMany()**
One has to specify the conditions through query operator for different fields to selectively delete those documents that satisfy the conditions

The following command will delete **all the students** with the name “Mike”

```
db.student.deleteMany(
     { name : “Mike” }
)
```

We can use **db.student.deleteOne()** to delete a single student.

Aggregation:

MongoDB has options for aggregation modelled via a data processing pipeline called the aggregation pipeline

Aggregation Pipeline

The Aggregation pipeline has stages each of which are represented by an aggregation operator. The **`match`** operator filters the documents on which aggregation is applied and the **`group`** operator groups documents to form the resultant document

We make another collection **orders** and insert the following data in it

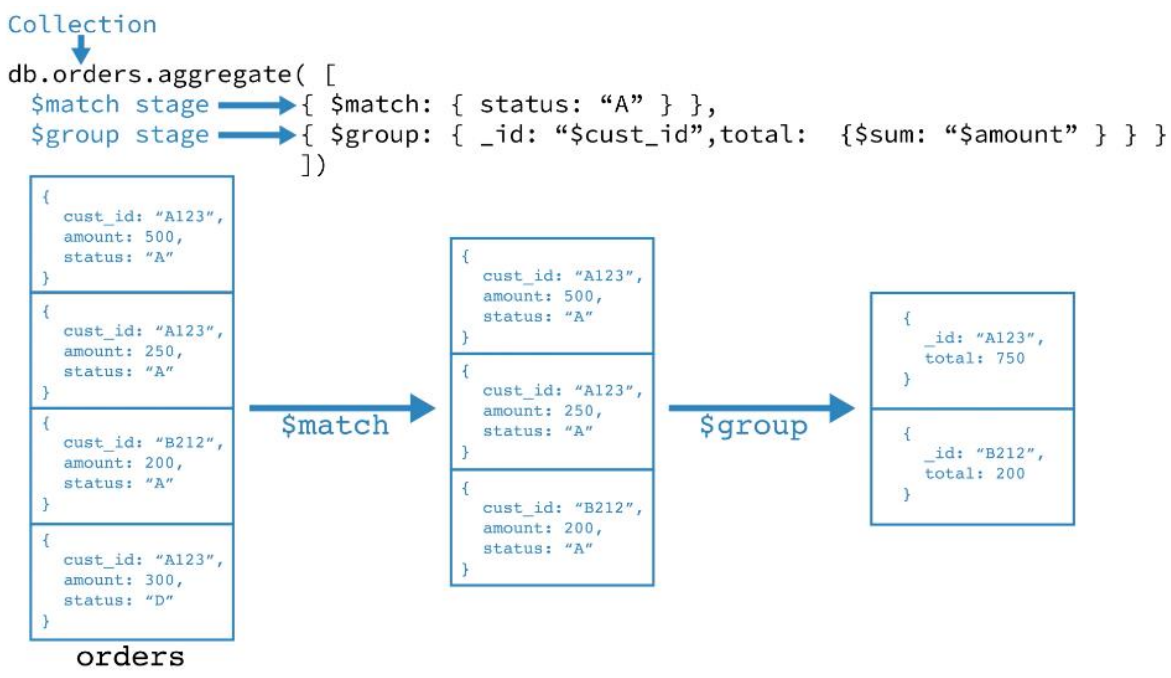
```
db.createCollection('orders')

db.orders.insertMany([
     {
         cust_id :"A123",
          amount : 500,
          status : "A"
     },
     {
         cust_id :"A123",
          amount : 250,
          status : "A"
     },
     {
         cust_id :"B123",
```

```
                amount : 200,
                status : "A"
            },
            {
                cust_id :"A123" ,
                amount : 300,
                status : "D"
            }
        ])
```

Now we will perform the following aggregate operation on the **orders** collection

```
db.orders.aggregate([
   { $match: { status: "A" } },
   { $group: { _id: "$cust_id", total: { $sum:
"$amount" } } }
])
```



First Stage: The $match stage filters the documents by the status field and passes to the next stage those documents that have status equal to "A".

Second Stage: The $group stage groups the documents by the cust_id field to calculate the sum of the amount for each unique cust_id.

The match stage can be written in the same way the *update filter* and *delete filters* are provided. A generic group stage looks like this

```
{
    $group:
    {
        _id: <expression>, // Group By Expression
        <field1>: { <accumulator1> : <expression1> },...
    }
}
```