Agenda

- Importing, Exporting Data
- Querying Data in MongoDB
- Creating and Manipulating Documents
- Advanced CRUD Operations

Importing and Exporting Data

Now that we have spoken about how data is stored, we can start working with it. As a first step, let's learn how to import and export data.

Let's say we have some data stored on our Atlas cluster and we wanted to export this data to a local machine or a different system entirely, is th possible? Absolutely-- but first, let's decide what format we're going to use. As we've already learned, data in MongoDB is stored in `BSON` but is viewe in `JSON` . `BSON` is great but isn't really human-readable. If I'm looking to just store this data and then maybe transfer it to a different system or cluster, would be best to export the data as `BSON` since it's lighter and faster. However, if I plan on viewing this data and reading through it locally after I expo it, then a human-readable `JSON` is a better choice.

In this lesson, we'll look at four commands, two that have to do with importing data and exporting data in `JSON` -- `mongoimport` and `mongoexport` and two that have to do with importing and exporting data in `BSON` -- `mongorestore` and `mongodump` .

`mongodump`

`mongodump` works as a utility to take the contents of a database and create a binary export. Running `mongodump` allows the user to export data fro a standalone, replica, set, and shared cluster deployments. The tool serves as a backup strategy. For IT professionals looking to schedule backups databases on a daily basis, this one of the methods for them to back up and restore databases (collections).

`mongodump` can save everything in a single file, while `mongorestore` can later be used to completely restore the database.

Syntax

You can run the `mongodump` command from the system command line, not the mongo shell.

This is the general `mongodump` command structure:

```
mongodump <options> <connection-string>
```

The user can connect to a mongo database using the `--uri` and a correctly formatted string or flag options like `--user` , `--db` , and `--password` The user isn't allowed to combine the two into a single command.

Using `mongodump` to back up a collection

While using the localhost, `mongodump` is able to dump a collection called `almabetter` with the following command while using a URI format and th following user information:

- Database name: `almabetter`
- Username: `user`
- Password: `password`

```
mongodump --uri="mongodb://user:password@localhost:27107/almabetter?ssl=false&authSource=admin"
```

Another example `mongodump` command using the standard flags would look like this:

```
mongodump --user=user --db=almabetter --password=password --authenticationDatabase=admin
```

It is also possible to run the database backup to an archive file. This is in contrast to dumping the files into a directory. These options are meant f transferring data between hosts or switching servers.

The `--archive` flag makes it possible to specify the name of the archive. The option creates one file that can be used to re-import the databas with `mongorestore` .

The standard `mongodump` process involves dumping the entire database into a single `dump` director which is named `dump` by default.This workir directory will be placed in the working directory that your ran the command in. The directory has a sub-folder that is named after the database.

In the previous example, this would be `almabetter` so the new structure looks like `./dump/almabetter` . Two different files for the collection in th database will be in the specific folder. This includes a BSON file, and a JSON file.

Following the same pattern, the `<collection>.metadate.json` file will contain the metadata like `options` , `indexes` , and `ns` to correspor with the namespace for the collection. The BSON file contains the `<collection>.bson` will hold the data in the collection. In `mongodump` the specif

behavior of the output can be changed by the user. The `dump` directory can use flags like `--out` that specify the name of the directory where yo want the database dumped. For instance, the name of the dump directory could be `dumbbase` instead of dump. The command would look like this:

```
mongodump --user=user --db=almabetter --password=password --authenticationDatabase=admin --out=dumbbase
```

All the collections are dumped into the output folder by default. The name of the folder will be included with the database. The user is able to furth control the utility by to only back up one collection at a time. Using the `-collection` flag allows the user to say which collection needs to be dumped.

If the only collection called `employees` should to be dumped, then an example `mongodump` command would look like:

```
mongodump --user=user --db=almabetter --password=password --authenticationDatabase=admin --out=dumbbase --collection=emp
```

The following folder structure would also be created with the command:

```
.
 |_dumbbase
   |_almabetter
     |_employees.metadata.json
     |_employees.bson
```

Using that command, it's possible to back up one collection at a time, as many times as the user desires. These commands won't overwrite any conten for the output folder.

Adding the `older` collection to the out dump folder would look like:

```
mongodump --user=user --db=almabetter --password=password --authenticationDatabase=admin --out=employees --collection=ol
```

That command would spit out the `database/almabetter` folder with the `older.metadata.json` and `older.bson` files added, making a structu that looks like the following:

```
.
 |_employees
   |_almabetter
     |_employees.metadata.json
     |_employees.bson
     |_older.metadata.json
     |_older.bson
```

****Using `mongodump` to dump all databases**

It's also possible to run the backup and have all files in an archive. This is in contrast to dumping everything into a dump directory. When transferring file between hosts or sending backup files between servers is when this option works best.

It uses the `--archive` flag so that the user can specify the name of the archive file. This option creates a single file that can be used to re-import th database with `mongorestore` . The user isn't allowed to use both the `--archive` and `--out` flags in tandem because of this.

The following `mongodump` command example below, will dump all databases (collections):

```
mongodump --db=almabetter --username=user --password=password --authenticationDatabase=admin --archive=almabetter.archiv
```

`mongorestore`

The opposite of the `mongodump` utility is the `mongorestore` , which allows users to restore the database. The program loads data fro the `mongodump` utility or any binary database dump.

The program differs from `mongoimport` in that `mongorestore` will only insert data. The program can't overwrite documents in the database th already exist. This includes updates. If the id for the document already exists, then the document won't be overwritten. Otherwise `mongorestore` ca create a new database or add data to an existing one.

When executing `mongorestore` , the only requirement is to have the path to the dump directory, the following `mongorestore` example can be used:

```
mongorestore dump/
```

If localhost is used as the host, and the names of the databases created have the same names of the sub-folders in the `dump` directory. The commar is only slightly more complicated when using a remote host.

The user will have to specify the `--uri` flag or include all the standard connection flags like:

```
--host
--db
--username
--port
--password
```

The program also doesn't require that the entire database get restored. It is possible to restore only a specific collection or list of collections. The us has the option to specify the `--collection` flag, the `--db` flag and include the path to the BSON file. In this case, `--collection` means th name of the collection in the database:

```
mongorestore --db=newdb --collection=comic_books dump/mydb/product.bson
```

`mongoexport`

As discussed earlier, `mongoexport` commands works in a similar manner as the `mongodump` , but it exports the data in JSON format which is huma readable.

Syntax

The `mongoexport` command has the following syntax:

```
mongoexport --collection=<coll> <options> <connection-string>
```

If you connect to the `localhost` MongoDB instance running on port `27017` , you don't need to specify the host and port.

For example, the following command exports the `books` collection from the `bookdb` database to the `books.json` file from the local MongoD instance running on port `27017` :

```
mongoexport --collection=books --db=bookdb --out=books.json
```

In this command:

- First, specify the `books` collection in the `--collection`
- Second, specify the database in the `--db` .
- Third, provide the path to the output file `books.json` in the `--out` .

If you want to export data from a remote MongoDB instance, you need to specify the host and port in the `--uri` connection string like this:

```
mongoexport --uri="mongodb://mongodb0.example.com:27017/reporting"  --collection=events  --out=events.json
```

This command exports the `books` collection of the `bookdb` database from the MongoDB instance locate at `mongodb://mongodb0.mongodbtutorial.org` running on port `27017` :

Alternatively, you can specify the hostname and port in the `--host` and `--port` like this:

```
mongoexport --host="mongodb0.example.com" --port=27017 --collection=events --db=reporting --out=events.json
```

For more information on the options available, see Options.

`mongoimport`

Let's take for example that we have a `books.json` file which contains data about thousands of books and we want to import this data to our MongoD server. To do so, we use the `mongoimport` tool. We can also use `mongoimport` to import data from CSV, or TSV formats.

The `mongoimport` command has the following form:

```
mongoimport <options> <connection-string> <file>
```

However, before importing data, you must first ensure to connect `mongoimport` utility to your MongoDB instance. While there are several ways connect `mongoimport` to your MongoDB database, it is recommended to use the `--uri` option, like this:

```
mongoimport --uri 'mongodb+srv://mycluster-ABCDE.azure.mongodb.net/test?retryWrites=true&w=majority'
   --username='MYUSERNAME'
   --password='SECRETPASSWORD'
```

Importing JSON files

Now, we can use the following command to import `books.json` into the MongoDB database server:

```
mongoimport --db bookdb --collection books --type=json --file c:\data\books.json
```

In a more concise way, this command can also be written as :

```
mongoimport c:\data\books.json -d bookdb -c books --drop
```

To summarise,

- First, start with the `mongoimport` program name.
- Next, specify the path to the `books.json` data file. In this example, it is `c:\data\books.json` .
- Then, use `-c bookdb` to specify the target database, which is `bookdb` in this example.
- After that, use `-c books` to specify the target collection, which is `books` in this case.
- Finally, use the `--drop` flag to drop the collection if it exists before importing the data.

Importing CSV files

The `mongoimport` command to import CSV files into a collection using the `--headerline` option. Headerline option notifie the `mongoimport` command of the first line; to be not imported as a document since it contains field names instead of data.

To import a collection from a CSV file, use the code as mentioned below:

```
mongoimport --db DB_Name --collection Collection_Name --type=csv --headerline --file=Name-of-file-to-import
```

Where,

- `DB_Name` represents the name of the database that contains the collection `Collection_Name` .
- `--type` specifies the file type CSV (Optional field).
- `--headerline` details the `mongoimport` command to take the 1st record of CSV file(s) as field names.
- `Name-of-file-to-import` represents the name and path of the CSV file to be imported/restored.

Importing CSV Files Lacking Header

In the event your CSV file doesn't have a header row, then you must notify the `mongoimport` command of the names of each column included in yo file. There are two ways of doing so:

- Specify Field Types
- Use a Field File

Specify Field Types :

One way is to write field names on the command-line using the `--fields` ****option. This can prove cumbersome if your CSV file contains lots columns, as can be seen from the command below:

```
mongoimport
   --collection=Collection_Name
   --file=Name-of-file-to-import
   --type=csv
   --fields="username,","identifier","one time password","recovery password","recovery account","one time code","recover
```

Using a Field file: The other method is to notify the `mongoimport` command on field columns using a separate .txt file. Here we've created a file calle " `Field_file.txt` " containing all column names present in our CSV file.

```
username
identifier
one time password
recovery password
recovery account
one time code
recovery code
```

```
user id
first name
last name
birth year
gender
department
company
country
```

After producing such a file, you can run the following command which will use your .txt file (in our example `Field_file.txt`) to extract information c these column names as field names in MongoDB:

```
mongoimport
   --collection= Collection_Name
   --file=Name-of-file-to-import
   --type=csv
   --fieldFile=Field_file.txt
```

Importing TSV files

TSV files are conceptually the same when compared with CSV file formats. Whether you use `mongoimport` Windows utility or another one, as a resu you may import TSV files using the same technique as described for CSV files.

There's only one minor change to keep in mind- instead of using the `--type=csv`, you can change it and use the `--type=tsv` option tell `mongoimport` of the new format.

Querying Data in MongoDB

In MongoDB, the `db.collection.find()` method is used to retrieve documents from a collection. This method returns a cursor to the retrieve documents.

The `db.collection.find()` method reads operations in mongoDB shell and retrieves documents containing all their fields.

Syntax

```
db.COLLECTION_NAME.find({})
```

Let's prepare a sample database to run the `find()` command on. On a fresh connection, the MongoDB shell will automatically connect the `test` database by default. You can safely use this database to experiment with MongoDB and the MongoDB shell.

To understand how MongoDB filters documents with multiple fields, nested documents and arrays, you'll need sample data complex enough to allo exploring different types of queries. As mentioned previously, this guide uses a sample collection of the five highest mountains in the world.

The documents in this collection will follow this format. This example document describes Mount Everest:

```
{
    "name": "Everest",
    "height": 8848,
    "location": ["Nepal", "China"],
    "ascents": {
        "first": {
            "year": 1953,
        },
        "first_winter": {
            "year": 1980,
        },
        "total": 5656,
    }
}
```

This document contains the following fields and values:

- `name` : the peak's name
- `height` : the peak's elevation, in meters
- `location` : the countries in which the mountain is located. This field stores values as an array to allow for mountains located in more than one country

- `ascents` : this field's value is another document. When one document is stored within another document like this, it's known as an *embedded* or *nested* document. Each `ascents` document describes successful ascents of the given mountain. Specifically, each `ascents` document contains a `total` field that lists the total number of successful ascents of each given peak. Additionally, each of these nested documents contain two fields whose values are also nested documents:

- `first` : this field's value is a nested document that contains one field, `year` , which describes the year of the first overall successful ascent
- `first_winter` : this field's value is a nested document that also contains a `year` field, the value of which represents the year of the first successful winter ascent of the given mountain

The reason why the first ascents are represented as nested documents even though only the year is included now is to make it easier to expand th ascent details with more fields in the future, such as the summiters' names or the expedition details.

Run the following `insertMany()` method in the MongoDB shell to simultaneously create a collection named `peaks` and insert five samp documents into it. These documents describe the five tallest mountain peaks in the world:

```
db.peaks.insertMany([
    {
        "name": "Everest",
        "height": 8848,
        "location": ["Nepal", "China"],
        "ascents": {
            "first": {
                "year": 1953
            },
            "first_winter": {
                "year": 1980
            },
            "total": 5656
        }
    },
    {
        "name": "K2",
        "height": 8611,
        "location": ["Pakistan", "China"],
        "ascents": {
            "first": {
                "year": 1954
            },
            "first_winter": {
                "year": 1921
            },
            "total": 306
        }
    },
    {
        "name": "Kangchenjunga",
        "height": 8586,
        "location": ["Nepal", "India"],
        "ascents": {
            "first": {
                "year": 1955
            },
            "first_winter": {
                "year": 1986
            },
            "total": 283
        }
    },
    {
        "name": "Lhotse",
        "height": 8516,
        "location": ["Nepal", "China"],
        "ascents": {
```

```
            "first": {
                "year": 1956
            },
            "first_winter": {
                "year": 1988
            },
            "total": 461
        }
    },
    {
        "name": "Makalu",
        "height": 8485,
        "location": ["China", "Nepal"],
        "ascents": {
            "first": {
                "year": 1955
            },
            "first_winter": {
                "year": 2009
            },
            "total": 361
        }
    }
])
```

The output will contain a list of object identifiers assigned to the newly-inserted objects.

You can verify that the documents were properly inserted by running the `find()` method with no arguments, which will retrieve all the documents yc just added:

```
db.peaks.find()
// output
{ "_id" : ObjectId("610c23828a94efbbf0cf6004"), "name" : "Everest", "height" : 8848, "location" : [ "Nepal", "China" ],
...
```

Querying individual fields

At the end of the previous step, you used MongoDB's `find()` method to return every document from the `peaks` collection. A query like this won't b very useful in practice, though, as it doesn't filter any documents and always returns the same result set.

You can filter query results in MongoDB by defining a specific condition that documents must adhere to in order to be included in a result set.

As an example, run the following query which returns any documents whose `name` value is equal to `Everest` :

```
db.peaks.find(
    { "name": "Everest" }
)
```

The second line — `{ "name": "Everest" }` — is the *query filter document*, a JSON object specifying the filters to apply when searching th collection in order to find documents that satisfy the condition. This example operation tells MongoDB to retrieve any documents in the `peaks` collectic whose `name` value matches the string `Everest` .

`.count()`

Appending `.count()` at the end of a `find()` query returns the number of documents that match the given query. For example :

```
> db.peaks.find(
    { "name": "Everest" }
).count()
//OUTPUT
1

> db.peaks.find().count()
//OUTPUT
5
```

`.pretty()`

Now, if we want to be able to view the data returned by a `find()` query in a nice, more readable way, we can do so by using `pretty()`. Here's a example:

```
> db.peaks.find(
    { "name": "Everest" }
)

//OUTPUT
{ "_id" : ObjectId("610c23828a94efbbf0cf6004"), "name" : "Everest", "height" : 8848, "location" : [ "Nepal", "China" ],

> db.peaks.find(
    { "name": "Everest" }
).pretty()

//OUTPUT
{
    "name": "Everest",
    "height": 8848,
    "location": ["Nepal", "China"],
    "ascents": {
        "first": {
            "year": 1953,
        },
        "first_winter": {
            "year": 1980,
        },
        "total": 5656,
    }
}
```

Creating and Manipulating Documents

In order to have data that you can practice reading, updating, and deleting in the later steps of this guide, this step focuses on how to create da documents in MongoDB.

Imagine that you're using MongoDB to build and manage a directory of famous historical monuments from around the world. This directory will sto information like each monument's name, country, city, and geographical location.

The documents in this directory will follow a format similar to this example, which represents **The Pyramids of Giza**:

```
{
    "name": "The Pyramids of Giza",
    "city": "Giza",
    "country": "Egypt",
    "gps": {
        "lat": 29.976480,
        "lng": 31.131302
    }
}
```

This document consists of four fields. First is the name of the monument, followed by the city and the country. All three of these fields contain strings. Th last field, called `gps`, is a nested document which details the monument's GPS location. This location is made up of a pair of latitude and longitud coordinates, represented by the `lat` and `lng` fields respectively, each of which hold floating point values.

Insert this document into a new collection called `monuments` using the `insertOne` method. As its name implies, `insertOne` is used to crea individual documents, as opposed to creating multiple documents at once.

In the MongoDB shell, run the following operation:

```
db.monuments.insertOne(
  {
    "name": "The Pyramids of Giza",
    "city": "Giza",
    "country": "Egypt",
```

```
    "gps": {
      "lat": 29.976480,
      "lng": 31.131302
    }
  }
)
```

Notice that you haven't explicitly created the `monuments` collection before executing this `insertOne` method. MongoDB allows you to run commanc on non-existent collections freely, and the missing collections only get created when the first object is inserted. By executing th example `insertOne()` method, not only will it insert the document into the collection but it will also create the collection automatically.

MongoDB will execute the `insertOne` method and insert the requested document representing the Pyramids of Giza. The operation's output w inform you that it executed successfully, and also provides the `ObjectId` which it generated automatically for the new document:

```
{
  "acknowledged" : true,
  "insertedId" : ObjectId("6105752352e6d1ebb7072647")
}
```

In MongoDB, each document within a collection must have a unique `_id` field which acts as a primary key. You can include the `_id` field and provic it with a value of your own choosing, as long as you ensure each document's `_id` field will be unique. However, if a new document omi the `_id` field, MongoDB will automatically generate an object identifier (in the form of an `ObjectId` object) as the value for the `_id` field.

Inserting documents one by one like this would quickly become tedious if you wanted to create multiple documents. MongoDB provide the `insertMany` method which you can use to insert multiple documents in a single operation.

Run the following example command, which uses the `insertMany` method to insert six additional famous monuments into the `monuments` collectior

```
db.monuments.insertMany([
  {"name": "The Valley of the Kings", "city": "Luxor", "country": "Egypt", "gps": { "lat": 25.746424, "lng": 32.605309 }
  {"name": "Arc de Triomphe", "city": "Paris", "country": "France", "gps": { "lat": 48.873756, "lng": 2.294946 }},
  {"name": "The Eiffel Tower", "city": "Paris", "country": "France", "gps": { "lat": 48.858093, "lng": 2.294694 }},
  {"name": "Acropolis", "city": "Athens", "country": "Greece", "gps": { "lat": 37.970833, "lng": 23.726110 }},
  {"name": "The Great Wall of China", "city": "Huairou", "country": "China", "gps": { "lat": 40.431908, "lng": 116.57037
  {"name": "The Statue of Liberty", "city": "New York", "country": "USA", "gps": { "lat": 40.689247, "lng": -74.044502 }
])
```

Notice the square brackets (`[` and `]`) surrounding the six documents. These brackets signify an *array* of documents. Within square brackets, multip objects can appear one after another, delimited by commas. In cases where the MongoDB method requires more than one object, you can provide a li of objects in the form of an array like this one.

MongoDB will respond with several object identifiers, one for each of the newly inserted objects:

```
{
  "acknowledged" : true,
  "insertedIds" : [
    ObjectId("6105770952e6d1ebb7072648"),
    ObjectId("6105770952e6d1ebb7072649"),
    ObjectId("6105770952e6d1ebb707264a"),
    ObjectId("6105770952e6d1ebb707264b"),
    ObjectId("6105770952e6d1ebb707264c"),
    ObjectId("6105770952e6d1ebb707264d")
  ]
}
```

You can verify that the documents were inserted by checking the object count in the `monuments` collection:

```
db.monuments.count()
// OUTPUT
7
```

Updating Documents

Similar to the `insertOne()` and `insertMany()` methods, MongoDB provides methods that allow you to update either a single document or multip documents at once. An important difference with these update methods is that, when creating new documents, you only need to pass the document da

as method arguments. To update an existing document in the collection, you must also pass an argument that specifies which document you want update.

To allow users to do this, MongoDB uses the same query filter document mechanism in update methods as the one we used previously wi the `find()` method. Any query filter document that can be used to retrieve documents can also be used to specify documents to update.

Try changing the name of **Arc de Triomphe** to the full name of **Arc de Triomphe de l'Étoile**. To do so, use the `updateOne()` method which updates single document:

```
db.monuments.updateOne(
  { "name": "Arc de Triomphe" },
  {
    $set: { "name": "Arc de Triomphe de l'Étoile" }
  }
)
```

The first argument of the `updateOne` method is the query filter document with a single equality condition, as covered in the previous step. In th example, `{ "name": "Arc de Triomphe" }` finds documents with `name` key holding the value of `Arc de Triomphe` . Any valid query filt document can be used here.

The second argument is the update document, specifying what changes should be applied during the update. The update document consists of upda operators as keys, and parameters for each of the operator as values. In this example, the update operator used is `$set` . It is responsible for settir document fields to new values and requires a JSON object with new field values. Here, `set: { "name": "Arc de Triomphe de l'Étoile" }` tel MongoDB to set the value of field `name` to `Arc de Triomphe de l'Étoile` .

The method will return a result telling you that one object was found by the query filter document, and also one object was successfully updated.

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

To check whether the update worked, try retrieving all the monuments related to `France` :

```
db.monuments.find({"country": "France"}).pretty()
//OUTPUT
{
  "_id" : ObjectId("6105770952e6d1ebb7072649"),
  "name" : "Arc de Triomphe de l'Étoile",
  "city" : "Paris",
  "country" : "France",
  "gps" : {
    "lat" : 48.873756,
    "lng" : 2.294946
  }
}
```

To modify more than one document, you can instead use the `updateMany()` method.

As an example, say you notice there is no information about who created the entry and you'd like to credit the author who added each monument to th database. To do this, you'll add a new `editor` field to each document in the `monuments` collection.

The following example includes an empty query filter document. By including an empty query document, this operation will match every document in th collection and the `updateMany()` method will affect each of them . The update document adds a new `editor` field to each document, and assigns a value of `AlmaBetter` :

```
db.monuments.updateMany(
  { },
  {
    $set: { "editor": "AlmaBetter" }
  }
)

//OUTPUT
{ "acknowledged" : true, "matchedCount" : 7, "modifiedCount" : 7 }
```

This output informs you that seven documents were matched and seven were also modified.

Confirm that the changes were applied:

```
db.monuments.find().pretty()
//OUTPUT
{
  "_id" : ObjectId("6105752352e6d1ebb7072647"),
  "name" : "The Pyramids of Giza",
  "city" : "Giza",
  "country" : "Egypt",
  "gps" : {
    "lat" : 29.97648,
    "lng" : 31.131302
  },
  "editor" : "AlmaBetter"
}
{
  "_id" : ObjectId("6105770952e6d1ebb7072648"),
  "name" : "The Valley of the Kings",
  "city" : "Luxor",
  "country" : "Egypt",
  "gps" : {
    "lat" : 25.746424,
    "lng" : 32.605309
  },
  "editor" : "AlmaBetter"
}
. . .
```

All the returned documents now have a new field called `editor` set to `AlmaBetter`. By providing a non-existing field name to the `$set` upda operator, the update operation will create missing fields in all matched documents and properly set the new value.

Although you'll likely use `$set` most often, many other update operators are available in MongoDB, allowing you to make complex alterations to yo documents' data and structure. You can learn more about these update operators in MongoDB's official documentation on the subject.

Deleting Documents

As with Mongo's update and insertion operations, there is a `deleteOne()` method, which removes only the *first* document matched by the query filt document, and `deleteMany()`, which deletes multiple objects at once.

To practice using these methods, begin by trying to remove the **Arc de Triomphe de l'Étoile** monument you modified previously:

```
db.monuments.deleteOne(
    { "name": "Arc de Triomphe de l'Étoile" }
)
```

Notice that this method includes a query filter document like the previous update and retrieval examples. As before, you can use any valid query specify what documents will be deleted.

MongoDB will return the following result:

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

Here, the result tells you how many documents were deleted in the process.

To illustrate removing multiple documents at once, remove all the monument documents for which `AlmaBetter` was the editor. This will empty th collection, as you've previously designated `AlmaBetter` as the editor for every monument:

```
db.monuments.deleteMany(
  { "editor": "AlmaBetter" }
)
//OUTPUT
{ "acknowledged" : true, "deletedCount" : 6 }
```

This time, MongoDB lets you know that this method removed six documents. You can verify that the `monuments` collection is now empty by countir the number of documents within it:

```
db.monuments.count()
//OUTPUT
0
```

Since you've just removed all documents from the collection, this command returns the expected output of `0` .

Thank You !