MongoDB Command Line Tutorial

Introduction

Like MySQL, MongoDB requires a server to run in order for users to work with MongoDB databases. Mongo refers to this as a service.

Starting the MongoDB Service

Start the service via Homebrew:

```
brew services start mongodb-community@8.0
```

or

```
mongod --config /usr/local/etc/mongod.conf
```

Stopping the Service

```
brew services stop mongodb-community@8.0
```

or

```
CTRL + C
```

Restarting the Service

```
brew services restart mongodb-community
```

Checking the Service is Running

```
brew services list
```

Logging In

After starting the service, logging in is a simple matter of:

```
mongosh
```

Exiting

Type `CTRL + C` or `exit` to get out.

Clearing the Screen

```
CTRL + L
```

or

```
cls
```

Show Databases

Like MySQL, you can use `show databases;` to list your databases:

```
show databases
```

Note: A semicolon is *not* required to terminate the MongoDB command.

Show Databases

You can also do:

```
show dbs
```

Show the Database I’m Currently Focused On

To see which database Mongo is currently in, type:

```
db
```

Show the Database I'm Currently Focused On

You can also list the current database with a longer command:

```
db.getName()
```

Note: If you're focused on a database that has no data, the database won't show up when you type `show dbs`.

Get Help

Mongo provides general help:

```
db.help()
```

Get Help

You can also get help specific to your database. For example, let's look at the help files associated with the `test` database:

```
db.test.help()
```

Get Help

You can get a list of commands:

```
db.listCommands()
```

And, `db.` + `TAB` will provide a listing of autocomplete options.

Create a Database

Creating a new database is as simple as saying, `use <DATABASE>`, where `<DATABASE>` is the database you'd like to create. Running the `use` command will also switch into that database; that is, it will place focus on the database. Let's create a database called `music`:

```
use music
```

Drop a Database

To drop a database, we first need to place focus on the database we wish to delete, then use the `dropDatabase()` method on the `db` object. Let’s drop the `music` database we just created:

```
use music
db.dropDatabase()
```

Create the Music Database Again

```
use music
```

Create a New Collection

Use the `createCollection` function to create a collection. Let’s create an `artist` collection in our `music` database:

```
db.createCollection(`artist`)
```

Note: You may use dashes in the name of a collection, but you’ll need a slightly different syntax to work with the collection. This will be discussed further in a later section.

Create a New Collection

If a collection with the same name already exists, you'll be presented with something akin to the following

```
{
   "ok" : 0,
   "errmsg" : "a collection 'music.artist' already exists",
   "code" : 48,
   "codeName" : "NamespaceExists"
}
```

Verify Collection Creation

Verify that the collection was built:

```
show collections
```

Insert a New Record Into a Collection

To insert a new record into our `artist` collection, we create a JSON object and reference the collection in the `db` method:

```
db.artist.insertOne({"artist_name": "Mogwai"})
```

Important Note About Dashes in MongoDB

Before we continue, let's discuss how dashes are dealt with by Mongo. If, instead of naming our collection `artist` we had named it `the-artists`, then the dash would cause problems when carrying out inserts using traditional JS notation.

Important Note About Dashes in MongoDB

Let's create a collection called `the-artists`:

```
db.createCollection(`the-artists`)
```

Important Note About Dashes in MongoDB

Now, let's try to insert a new record into the `the-artists` collection using the syntax we used in slide 22.

```
db.the-artists.insert({"artist_name": "Interpol"})
```

Important Note About Dashes in MongoDB

Mongo responds with an error. This is because we cannot reference the collection using that syntax.

Important Note About Dashes in MongoDB

When using dashes in collection names, you'll need to refer to the collection using bracket syntax.

For example, to insert a new record into `the-artists` collection, we require either of the following syntaxes:

```
db['the-artists'].insert({"artist_name": "Interpol"})
db["the-artists"].insert({"artist_name": "Interpol"})
db[`the-artists`].insert({"artist_name": "Interpol"})
```

Important Note About Dashes in MongoDB

To avoid these dash-related problems, avoid dashes and use camel case instead. For example, use `theArtists` instead of `the-artists`:

```
db.theArtists.insert({"artist_name": "Interpol"})
```

In this manner, you'll be able to avoid using bracket syntax.

Remove a Collection

We use `drop` to remove a collection. For instance, let’s drop the collections created in the previous slides.

```
db.artist.drop()
```

and

```
db["the-artists"].drop()
```

Remove a Collection

In both cases, Mongo should have responded with `true`.

If you try to remove a collection that has already been removed, or one that never existed, Mongo will response with `false` to your `drop` statement.

Recreate the `artist` Collection

Let’s create the `artist` collection:

```
use music
db.createCollection(`artist`)
```

Insert a New Nested Record Into a Collection

When populating Mongo, it's crucial to format, organize, and validate your JSON *before* inserting any records into your collection.

Insert a New Nested Record Into a Collection

In the following example, I’m introducing two albums and two artists into our collection.

Insert a New Nested Record Into a Collection

```
db.artist.insertOne(
  {
    "name": "Mogwai",
    "albums": [{
      "Young Team": [
        {"track": ["Yes! I Am a Long Way from Home", 357]},
        {"track": ["Katrien", 324]}]
    },
    {
      "Every Country’s Sun": [{
        "Coolverine": 377,
        "Don’t Believe the Fife": 384
      }]
    }]
  }
)
```

Insert a New Nested Record Into a Collection

```
db.artist.insertOne(
  {
    "name": "Interpol",
    "albums": [{
      "Turn on the Bright Lights": [
        {"track": ["Untitled", 237]},
        {"track": ["Obstacle 1", 251]}]
    },
    {
      "Marauder": [{
        "The Rover": 218,
        "It Probably Matters": 248
      }]
    }]
  }
)
```

Retrieve All Rows in a Collection

```
db.artist.find()
```

Retrieve All Rows in a Collection Using `pretty`

```
db.artist.find().pretty()
```

Delete a Single Document in a Collection

```
db.artist.deleteOne({"name": "Mogwai"})
```

Delete Everything in a Collection

```
db.artist.deleteMany({})
```

Exporting JSON Using `mongoexport`

We use the bash-level command `mongoexport` to export our collections as JSON:

```
mongoexport \
  --collection=artist \
  --db=music \
  --out=music.json
```

Note: The back slashes invoke bash's line-folding feature. If your CLI doesn't support this feature, write the entire command on one line, sans slashes.

Exporting JSON Using `mongoexport`

The command is self-explanatory, but it’s worth noting that the value to `out` is the final JSON file we want.

```
mongoexport \
  --collection=artist \
  --db=music \
  --out=music.json
```

Note: If you want to fully replicate a database, use `mongodump` to export the database, and `mongorestore` to restore the dump.

Exporting CSV Using `mongoexport`

You can also export collections as a CSV file. You'll need to use the `type` *and* `fields` flags:

```
mongoexport \
  --collection=artist \
  --db=music \
  --type=csv \
  --fields=artist_name \
  --out=music.csv
```

All the collections in your database won't automatically be exported. You'll need to specify the fields you want as a comma-separated list of values to the `field` flag.

Importing JSON Using `mongoimport`

Use `mongoimport` to import a JSON file:

```
mongoimport \
  --db=music \
  --collection=artist \
  --file=music.json
```

Note: If your JSON file has records with the same `ObjectID`, Mongo will reject the import.