

MongoDB with Express

Topic:

- How do we connect to our database server

We might be using atlas or local database, the practice will be same, only the URL would change. When we click on '**CONNECT**', it was ask us to add the IP address. Either the IP address that we are using or another IP address. But we will click on **ALLOW ACCESS FROM ANYWHERE**, so that everyone will be able to access. Everyone as in users that we add to the **DATABASE USER**. We can save a user in our database. **Database users**' are actually credentials using which the database can be accessed the database directly. So, the database users is about the user which is going to be used as credentials to connect the application with the database.

As our application will be connecting to this database, for that we need create a user. Then we need a database admin, database backup expert and database designer. We can add all these users in this list. We can define the security rules as well. Now that user is created we will use the user credentials to connect to this database. We will not be using Mongo Shell, but we will put the URL in the MongoDB Compass. But we actually want to connecte the database to our application. So click on **Connect Your Appilcation**. IN this we will get a URL which we need to put inside the application.

So, we need to get the URL or the string. When we click on connect, it would open Connect to Cluster() window and we are asked to add our Current IP Address. It would auto prefill to 0.0.0.0 so click on autofill.

In a database there will be multiple data points. Database user → a credential which can access the user database directly. We need to access the google, then it will not work.

We will create a user so; we will give username as expressApp and password as 1234. Now the user is created, we will user the user credentials to connect to the data base.

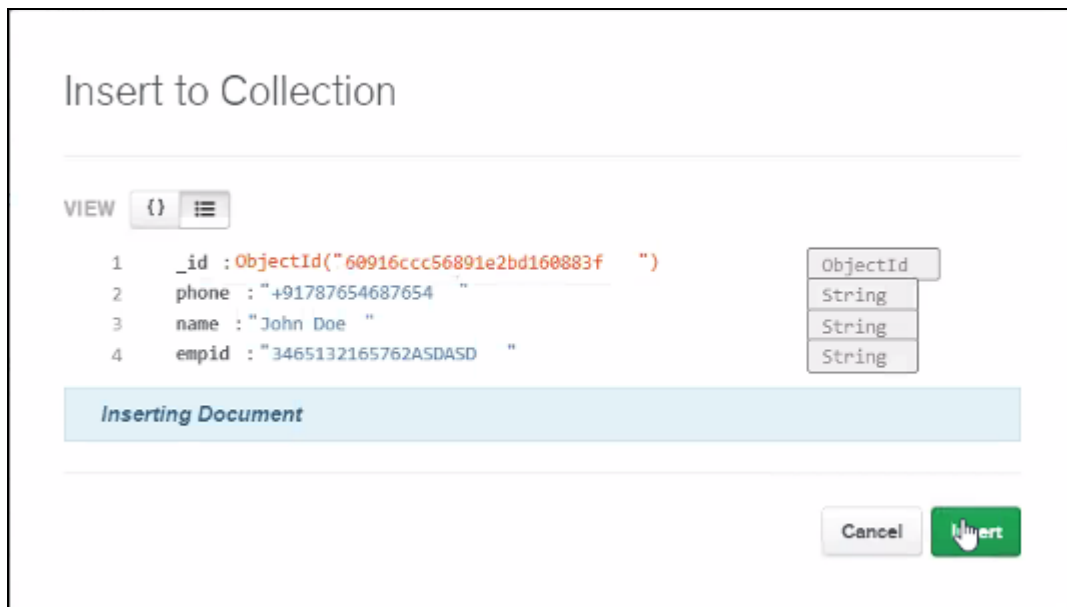
This URL we will be copied inside the application so that it can be connected. We have the ability to create server using express js. In the same way we have a mongodb npm package we can use to connect to this particular package.

Finally, when we connect it then we can see it as well. Once we got the URL, we get the mongoDB npm package → **MongoDB NodeJS Driver**

We will use this package to connect to the database. We also have to initialize the NodeJS project.

npm i mongodb

We have created a **database** as “**maruti**” and **cars**, **employees** and **customers** as **collections**. In each collection, we will add some sample data. We will add some random data using INSERT DOCUMENT.



Insert to Collection

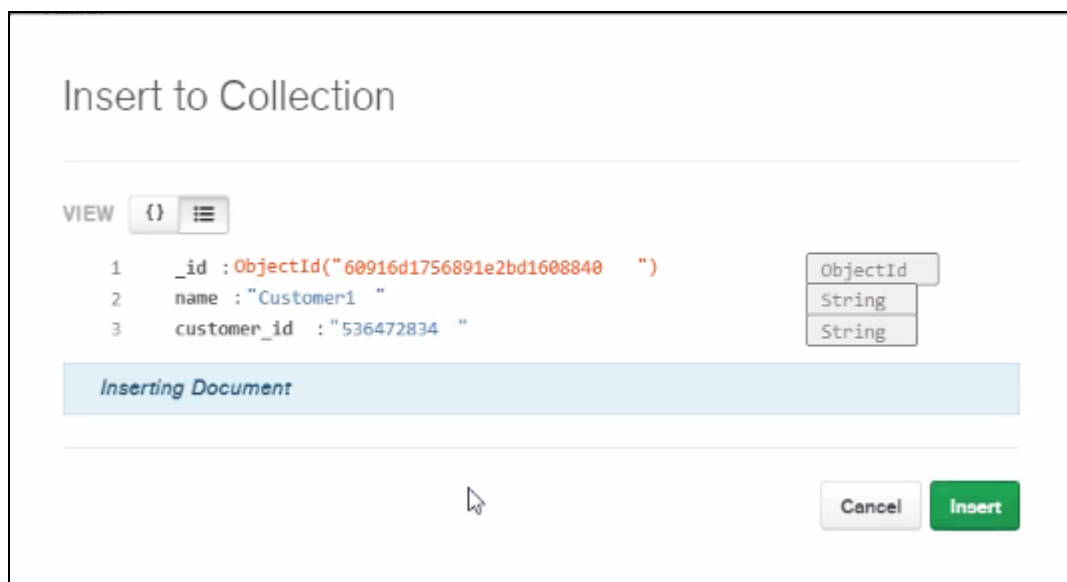
VIEW {} ≡

```
1  _id : ObjectId("60916ccc56891e2bd160883f")
2  phone : "+91787654687654"
3  name : "John Doe"
4  empid : "3465132165762ASDASD"
```

Inserting Document

Cancel Insert

We will add a customer ID as well,



Insert to Collection

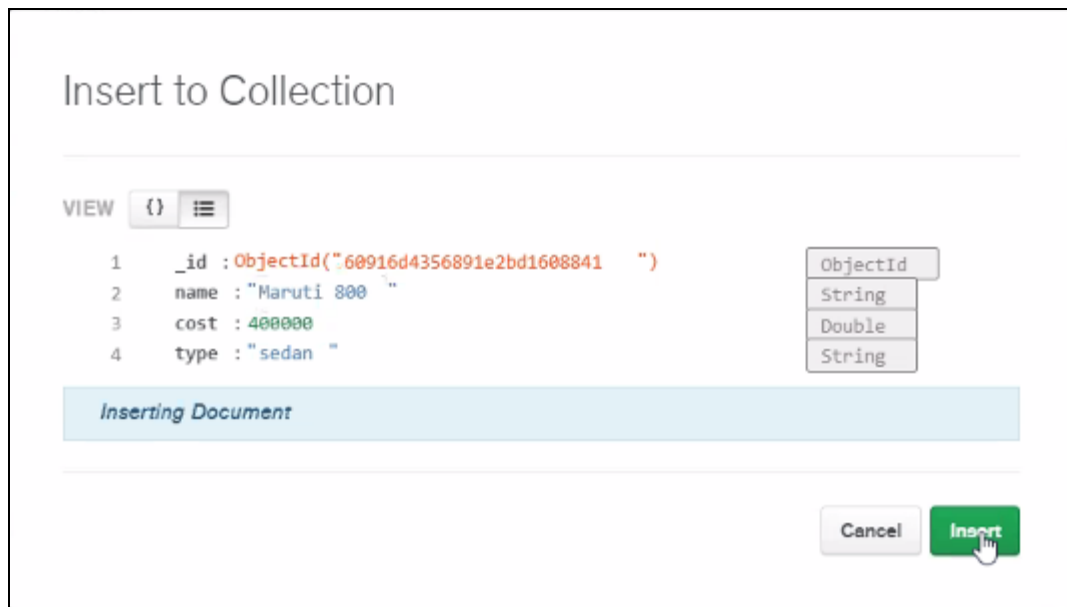
VIEW {} ≡

```
1  _id : ObjectId("60916d1756891e2bd1608840")
2  name : "Customer1"
3  customer_id : "536472834"
```

Inserting Document

Cancel Insert

We are adding all the details manually, but later on, we will use the script.



We have three collections with one document in each.

We have to require mongodb and inside it, we need MongoClient, which will help me to connect to the database. In our javascript file,

```
const MongoClient = require('mongodb').MongoClient;

const dbURL = 'mongodb+srv://express:1234@cluster0.69m9q.mongodb.net/myFirstDatabase?retryWrites=true&w=majority'
```

We will have to pass a callback. If we are on localhost, then the URL can be changed to,

```
const MongoClient = require('mongodb').MongoClient;

const dbURL = 'mongodb+://express:1234/localhost:3030/'
```

In production level we don't store directly in the application. Now we need to create an instance of MongoClient. We can do it directly. While creating an instance first and then calling connect, in such case, the first parameter will become the callback function itself. We will have to pass a callback function `() => {}`. **This** function gets executed the moment MongoDB gets connected.

```
MongoClient.connect(dbURL, () => {

  console.log('MongoDB Connected')

})
```

What if we have an error? ➔

Just as in express, the call back has 3 parameters req, res and next; in here the first parameter would be the error object. For example, if internet is down or server is down, in such case this object will be truthy or non-null object and its value will be undefined or null. The second parameter will be actual instance of client which will be connected to my database. If an error is there, while connecting to the database, it would throw error and parallelly, we will log the client as well.

```
MongoClient.connect(dbURL, (err, client) => {  
  
  if(err) throw new Error('Failed to connect')  
  
  console.log('MongoDB is Connected')  
  console.log(client)  
})
```

As it is getting called asynchronously, so it is taking time to get the result.

MongoDB is Connected.

In the clusters if we have a persistent connection, then in the CONNECTIONS, it will show us how many connections we have.

Now, that we have the client's instance, the database, "maruti". In our cluster we can have multiple databases. The client is having all the collection and reference to the hoisted database.

```
const db = client.db('maruti')
```

Now the client is configured to use "maruti's" database. It returns a **db** instance which you can call to collections or get reference to the collections. In db, maruti's database reference is being stored. Now, if are storing the db instance, then to get the reference to the collection() and then the collection name should be either cars, employees or customers. For now, let's take **cars** and we'll store the cars collection reference in **carCollection**.

In the cluster we are saying **client.db("maruti")** ➔ "maruti" database reference is being stored in the **const db**. MongoDB does not know if it is a car or a user. It knows it will be a collection.

```
const carsCollection = db.collection('cars')
```

This will store reference to the cars collection. For now, we will call **find** and put an empty object as we need everything. If we mention any search query, it will only get that detail. After this, we will call the function `toArray()` to get the output as an array. This `toArray()` returns a promise, so we can await for it and so we can,

```
const MongoClient = require('mongodb').MongoClient;

const dbURL = 'mongodb+srv://express:1234@cluster0.69m9q.mongodb.net/myFirstDatabase?retryWrites=true&w=majority'

MongoClient.connect(dbURL, async (err, client) => {

  if(err) throw new Error('Failed to connect')

  console.log('MongoDB Connected')
  // console.log(client)

  const db = client.db('maruti')
  const carsCollection = db.collection('cars')
  const cars = await carsCollection.find({}).toArray()
  console.log(cars)
})
```

If everything works right, we should get the details in an array. `MongoClient.connect` initiates the connection to the database. Once it is done it will invoke the callback function. If the connect is successful, variable **client**, being an object, but it will have reference and connection it needs to get the data from the database.

The database name that we have given is “maruti”. Since we want to data from a document in the cars collection from “maruti” database, so we are saying in our code that we want reference to database of “maruti”. Once we have the reference, we want the data from cars as the document is inside it. So, we have to tell the script to get the details from the cars collection. Once that reference is there, we want to list all the data. We have multiple functions to get all the data in an array form. Till `carsCollection.find({}).toArray()`, the client goes to the cloud server fetches all the data from it and once it is done, it converts the data into an array saves the data in variable **cars**.

Let’s add one more document to the **cars** collection.

Now, it will have an array of all the data. Let us add one more document in the cars collection.

Insert to Collection

VIEW

{}

≡

1

`_id : ObjectId("6091753f56891e2bd1608842")`

2

`name : "Zen"`

3

`cost : 500000`

ObjectId

String

Double

Inserting Document

Cancel

Insert

It inserted the data, even though the type property is undefined. If this was an SQL based data, then irrespective of us providing data, the type property would be mentioned. We can dynamically store any sort of data and that field can be variable. We don't have to change the structure of that entire object.

INSERT DOCUMENT

FILTER

{ "filter": "example" }

Find

Reset

QUERY RESULTS 1-2 OF 2

`_id: ObjectId("60916d4356891e2bd1608841")`
`name: "Maruti 800"`
`cost: 400000`
`type: "sedan"`

`_id: ObjectId("6091753f56891e2bd1608842")`
`name: "Zen"`
`cost: 500000`

Since we have added data, if we call this particular script again, we will be displaying the data of both the documents. Now we see that the process is still running because I need to call some called as `connectInstance.close()` do this most of the times, just a good practice. This line of code terminates the process.

In the over view tab, we can see the connections:2 because previously we did not give the `close()` function. In the real time we can see how many users are connected.

This is a basic connection.

insertOne()

Now we are able to read the data, let us see, how to insert data. We can translate this to other collections (customers & employees) as well. If we want to insert in the cars collection, then we need reference to the cars collection.

```
const carsCollection = db.collection('cars')
```

We want to insert a data into this collection. We will create the new data first that we want to insert to the cars collection.

```
const newCar = {  
  name: 'Alto',  
  price: 800000,  
  mfg: 'Maruti',  
  bhp: 120  
}
```

Even though we do not have mfg and bhp options, still we will be able to insert the data as it is flexible.

```
const asd = await carsCollection.insertOne(newCar)
```

Based on the user case, we might want to insert one or more document at once. So we will be called insertOne(). We have the data and now using insertOne(), will be adding the newCar data to our database.

```
const MongoClient = require('mongodb').MongoClient;  
  
const dbURL = 'mongodb+srv://expressApp:1234@cluster0.36m9q.mongodb.net/  
myFirstDatabase?retryWrite=true'  
  
MongoClient.connect(dbURL, async (err, client) => {  
  
  if(err) throw new Error('Failed to connect')  
  const carsCollection = db.collection('cars')  
  
  const newCar = {  
    name: 'Alto',  
    price: 800000,  
    mfg: 'Maruti',  
    bhp: 120  
  }  
  const asd = await carsCollection.insertOne(newCar)  
})
```

In the output at the end, we can see

insertedCount: 1,
insertedID:

On the MongoDB cloud if we refresh, we can see the data being added to the cars collection. The insertedID that is available on the terminal is also attached to the newly added data to the database. This id is a unique id across database so that it keeps track of database.

deleteOne()

Let us how to use deleteOne(). To deleteOne() we need someway to identify it. This can be done using the unique id.

```
const deleteResult = await carsCollection.deleteOne({name: 'Maruti 800'})

console.log(deleteResult.deletedCount)

client.close()
```

If we refresh, we can see that the Maruti 800 document has been removed. Now, if we want to delete a car with a particular ID, then we see it is not being deleted. This is because we have to import the ObjectId.

```
const ObjectId = require('mongodb').ObjectId

const deleteResult = await carsCollection.deleteOne({id: ObjectId("654a
sd6a5sa6sd8a6sd38s6df")})

console.log(deleteResult.deletedCount)

client.close()
```

The updateOne() will update a particular document and the remaining details would not be changed.

We want to update the document in the cars collection. So, we will call the public object. The first parameter would be a filter object.

```
const updates = {
  cost: 100000
}
```



```
const updateResult = await carsCollection.updateOne({_id: ObjectId("654asd6a5sa6sd8a6sd38s6df")}, updates)

console.log(updateResult)
```

This is not working but if we use `replaceOne()` instead of `updateOne()` function, then it would remove the **name** and will change the cost value from previous to the given value.

```
const updateResult = await carsCollection.replaceOne({_id: ObjectId("654asd6a5sa6sd8a6sd38s6df")}, updates)

console.log(updateResult)
client.close()
```