MongoDB and Node.js

Accessing MongoDB using Node.js

MongoDB Structure

Project - Many clusters

Cluster - A set of a data base

Database - A set of collections (directories)

Collection - A table (a directory with files)

Document - A row in a table (a file)

MongoDB Cluster Structure

A MongoDB cluster is a group of MongoDB servers working together as a single system.

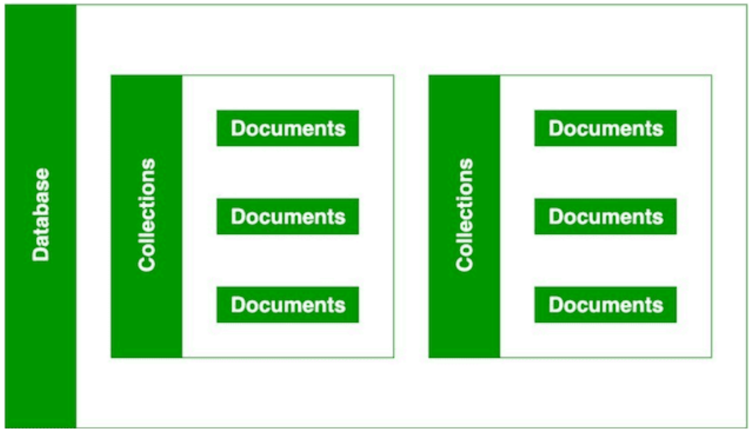
- You access it with a connection string like:

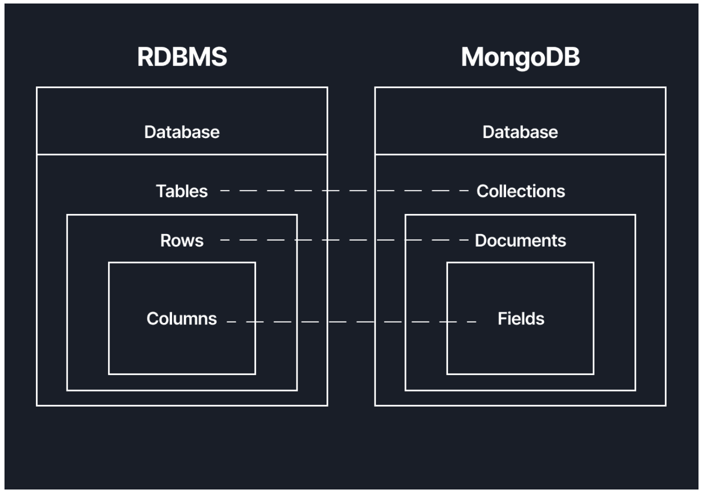
```
mongodb+srv://user:pass@cluster0.xxxxx.mongodb.net
```

A MongoDB Cluster provides reliability, safety, scalability, and performance that a single machine cannot deliver.

MongoDB Database

One Database has multiple collections that have multiple documents.





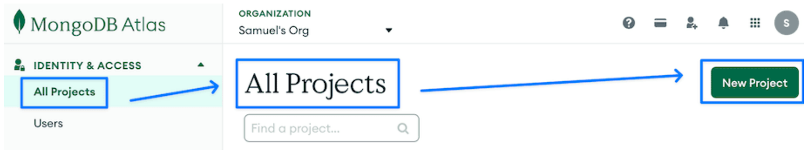
- One RDBMS table is equivalent to Collection (directory).
- A row in a table is to Document (one file).

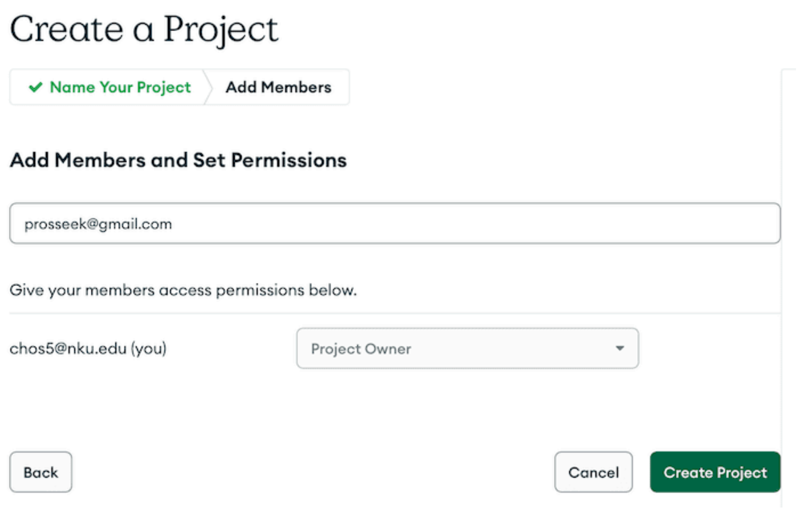
Using MongoDB Atlas

The simplest way to use MongoDB is to use Atlas and Compass.

1. Register MongoDB Atlas:
   https://www.mongodb.com/products/platform
2. Use Compass to access MongoDB from standalone Desktop app:
   https://www.mongodb.com/products/tools/compass

Create Project





Create a Project

Add Members and Set Permissions

prosseek@gmail.com

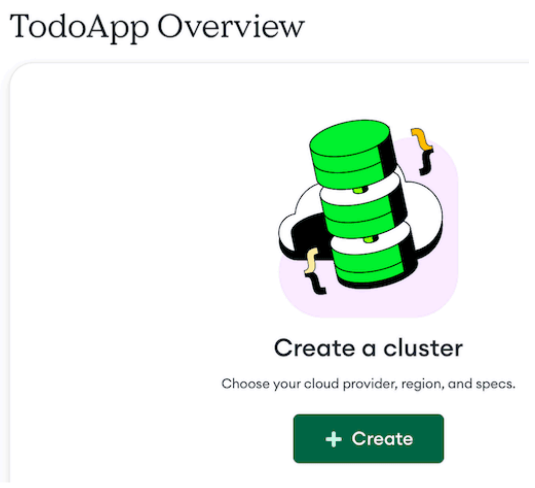
Give your members access permissions below.

chos5@nku.edu (you) — Project Owner

Let's say we make `TodoApp` project.

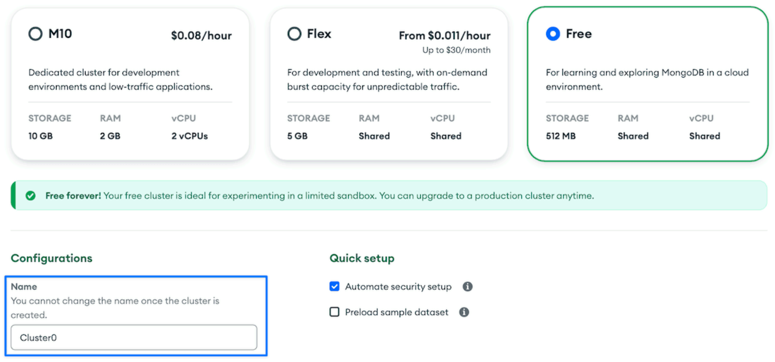
Create Cluster

In the project, we can create cluster; choose free version.


TodoApp Overview

Create a cluster

Choose your cloud provider, region, and specs.


M10	$0.08/hour		
Dedicated cluster for development environments and low-traffic applications.			
STORAGE	RAM	vCPU	
10 GB	2 GB	2 vCPUs	

Flex	From $0.011/hour Up to $30/month		
For development and testing, with on-demand burst capacity for unpredictable traffic.			
STORAGE	RAM	vCPU	
5 GB	Shared	Shared	

[✓ Free]			
For learning and exploring MongoDB in a cloud environment.			
STORAGE	RAM	vCPU	
512 MB	Shared	Shared	

Free forever! Your free cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

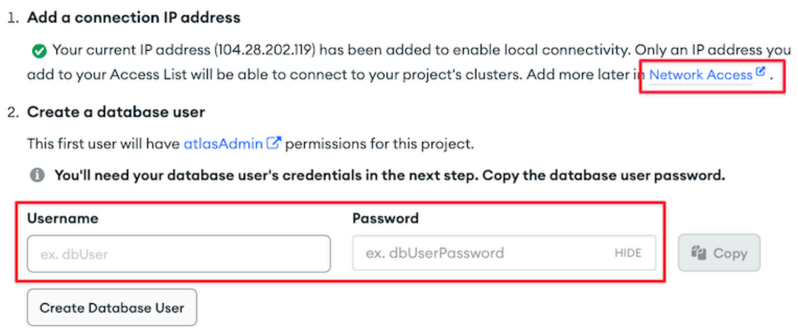
Configurations

Name
You cannot change the name once the cluster is created.
Cluster0

Quick setup

☑ Automate security setup
☐ Preload sample dataset

Create user/password for the cluster: chos5/1234hello (example)



1. **Add a connection IP address**

   Your current IP address (104.28.202.119) has been added to enable local connectivity. Only an IP address you add to your Access List will be able to connect to your project's clusters. Add more later in Network Access .

2. **Create a database user**

   This first user will have atlasAdmin permissions for this project.

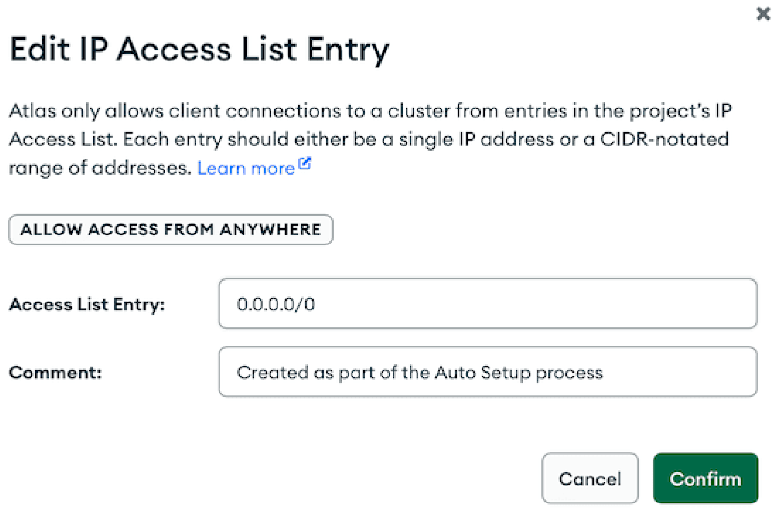
   You'll need your database user's credentials in the next step. Copy the database user password.

Username	Password
ex. dbUser	ex. dbUserPassword HIDE

Copy

Create Database User

Edit to allow the access from anywhere for easy development.



Edit IP Access List Entry

Atlas only allows client connections to a cluster from entries in the project's IP Access List. Each entry should either be a single IP address or a CIDR-notated range of addresses. Learn more

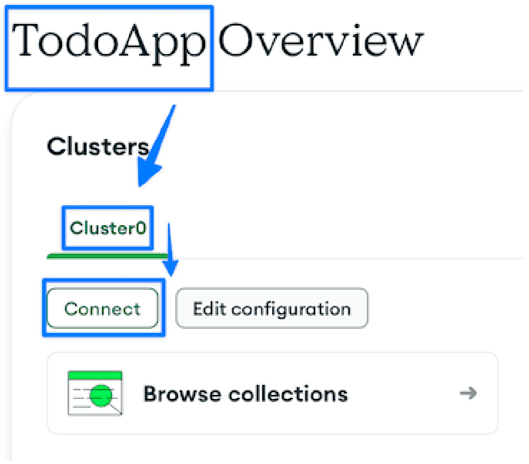
ALLOW ACCESS FROM ANYWHERE

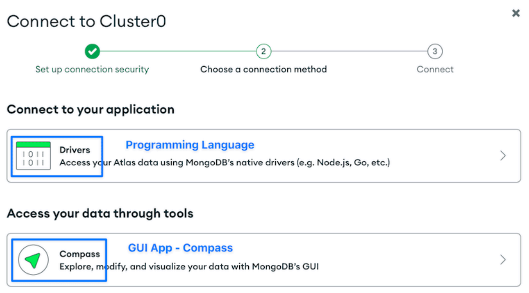
Access List Entry:	0.0.0.0/0
Comment:	Created as part of the Auto Setup process

Cancel Confirm

Connect to Cluster0

We assume that we use only cluster0 in the project; if you use other cluster name, adjust accordingly.




Connect to Cluster0

Set up connection security — Choose a connection method — Connect

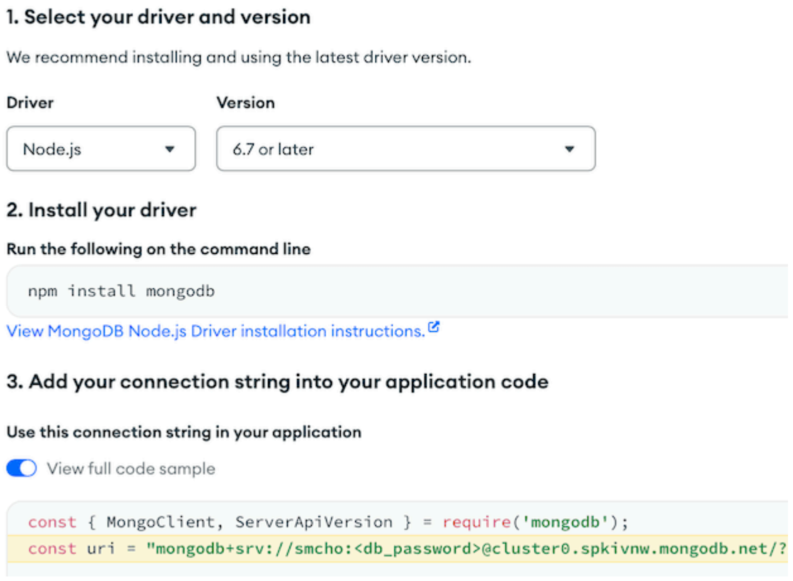
Connect to your application

Drivers — Programming Language
Access your Atlas data using MongoDB's native drivers (e.g. Node.js, Go, etc.)

Access your data through tools

Compass — GUI App - Compass
Explore, modify, and visualize your data with MongoDB's GUI

Choose `Node.js` to get the information to access the MongoDB.



1. Select your driver and version

We recommend installing and using the latest driver version.

Driver: Node.js

Version: 6.7 or later

2. Install your driver

Run the following on the command line

```
npm install mongodb
```

View MongoDB Node.js Driver installation instructions.

3. Add your connection string into your application code

Use this connection string in your application

View full code sample

```
const { MongoClient, ServerApiVersion } = require('mongodb');
const uri = "mongodb+srv://smcho:<db_password>@cluster0.spkivnw.mongodb.net/?
```

For `chos5/1234hello`, we can access the mongoDB Cluster (`spkinwx`) as:

```
mongodb+srv://chos5:1234hello@cluster0.spkinwx.mongodb.net/
```

The `appName` parameter in a MongoDB connection string is optional and is used to uniquely identify the application connecting to the database; It can be useful for monitoring, logging, and debugging purposes on the server side, as it allows MongoDB to see which application is making requests.

```
mongodb+srv://chos5:1234hello@cluster0.spkinwx.mongodb.net/appName=Cluster0
```

.env

We store `user/password` information in the `.env` file.

```
MONGO_USER=YOUR_USER
MONGO_PASSWORD=YOUR_PASSWORD
MONGO_CLUSTER=YOUR_CLUSTER
```

Use the env template:

1. Copy `env` to `.env` and replace your user/password/cluster with yours.
2. Be sure that `.gitignore` contains `.env` so it is not uploaded to GitHub.

You can add more information such as:

```
MONGO_DATABASE=YOUR_DATABASE
```

We use `dotenv` package to access the variables in the `.env` file.

```
require('dotenv').config({ quiet: true });
```

If the .env is not in the root directory (of your project), we need to specify the locaiton.

```
require('dotenv').config({ path: './path/to/your/.env' });
```

`uri.js` Utility module

We use `uri.js` to access the environment variables and create the uri string for MongoDB access.

```
const dotenv = require("dotenv");

// Load environment variables
dotenv.config({quiet: true, path: "../.env",});

// Validate required variables
const user = process.env.MONGO_USER;
if (!user) throw new Error("MONGO_USER is not defined");
const password = process.env.MONGO_PASSWORD;
if (!password) throw new Error("MONGO_PASSWORD is not defined");
const cluster = process.env.MONGO_CLUSTER;
if (!cluster) throw new Error("MONGO_CLUSTER is not defined");
// Database name
const databasename = process.env.MONGO_DATABASE || "todoapp";

// Construct the MongoDB connection string
const uri = `mongodb+srv://${user}:${password}@cluster0.${cluster}.mongodb.net/`;

// Export (CommonJS style)
module.exports = {uri, databasename,};
```

CommonJS Module Style

In this example, we use CommonJS module style:

1. It uses `require` for module import.

```
const dotenv = require("dotenv");
```

2. It uses `modules.exports` for module export.

```
module.exports = {
  uri,
  databasename,
};
```

MongoClient object client

With the `uri`, we can create the MongoDB client object to access the MongoDB database.

```
const { MongoClient } = require('mongodb');
const uri = require('./uri.js')
const client = new MongoClient(uri);
```

To ensures your application only uses commands and behaviors defined in the specified stable API version, we may use `ServerApiVersion`.

```
const { MongoClient, ServerApiVersion } = require('mongodb');
const uri = require('./uri.js')
const client = new MongoClient(uri, {
  serverApi: {
    version: ServerApiVersion.v1,
    strict: true,
    deprecationErrors: true,
  }
});
```

Connection Layer and Database

This is native driver mental model with the official mongodb driver:

```
import { MongoClient } from "mongodb";

const client = new MongoClient(uri); // connection pool (not yet connected)

let db = null;

export async function connect(databaseName = DATABASE) {
  if (db) return db;           // already connected & using this DB

  await client.connect();      // open TCP connections, auth, etc.
  db = client.db(databaseName) // lightweight DB handle on that connection
  return db;
}
```

In this model:

- client = connection pool object (one per app is typical)
- db = logical database (a thin wrapper over client)

```
client  ---- sockets, pool, auth
   |
   +-- db("todoapp")
   +-- db("logs")
```

To close DB connection, we use client:

```
client.close()
```

`db.js` Utility function

We export client object and getDB function

```
// db.js
// Create a single MongoClient instance
const client = new MongoClient(uri); // connection

// Create and reuse a single database connection
async function getDB(database = DATABASE) {
  if (db) return db;          // reuse existing connection

  await client.connect();     // connect once
  db = client.db(database);   // store db globally
  return db;
}

module.exports = {
  client,
  getDB,
  ...
};
```

Singleton Design Pattern

> In the code, we reuse `db` if it is assigned a Database: `if (db) return db;`.
> In this way, we use only one Database object.
> We call this design pattern "Singleton Design Pattern."

Collections

We need to make collections, so in the `db.js` module, we make utility functions to return the collections.

```
const POSTS = 'posts';
const COUNTER = 'counter';
```

```
// Return the "posts" collection
export function getPostsCollection() {
  return db.collection(POSTS);
}

// Return the "counter" collection
export function getCounterCollection() {
  return db.collection(COUNTER);
}
```

Example1: ping.js

We can use the ping command to check if we can access the MongoDB.

```
const {client, getDB} = require('./util/db.js')

async function run() {
  try {
      const db = await getDB("admin"); // get DB
    await db.command({ ping: 1 }); // ping
    console.log("Pinged your deployment. You successfully connected to MongoDB!");
  } finally {
    // Ensures that the client will close when you finish/error
    await client.close();
  }
}
run().catch(console.dir);
```

We use the `db.js` client object:

```
const db = require('./util/db.js')
let client;
client = await db.client;
```

In the CommonJS, we cannot use top-level async/await, so we should put the async/await code in a function and run it.

```
async function run() {
  try { ... }
  finally { await client.close(); }
}
run().catch(console.dir);
```

Example 2: database.db: Create DB & Collection

MongoDB will:

- Create database nku_demo
- Create collection students
- Insert one document (thus materializing both)

```
// 1. Create (or use) a database
const db = await getDB("nku_demo"); // reates only when data is added
console.log("Connected!");
// 2 Create (or use) a collection
const students = db.collection("students");
// 3. Optionally, define indexes or insert ata
await students.insertOne({ name: "Alice", age: 22, major: "ASE" });
```

Optional: Explicitly create collection

If you want to explicitly create a collection (e.g., with schema validation, capped size, etc.) before inserting, use:

```
await db.createCollection("students", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "age"],
      properties: {
        name: { bsonType: "string" },
        age: { bsonType: "int", minimum: 0 }
      }
    }
  }
});
console.log("Collection 'students' created with schema!");
```

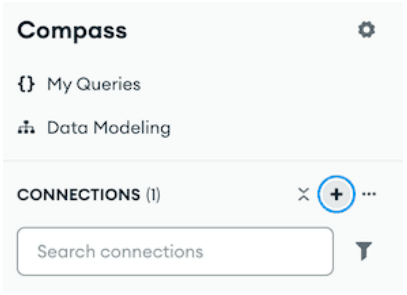
The Need for Mongoose

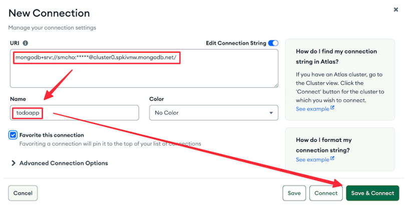
> For MongoDB, schema is an option, and it may sometimes cause an issue.
>
> Mogoose is the solution to enforce schema or Data model to resolve this issue.
>
> We will discuss the idea of Mongoose in the next section.

Compass as the Desktop MongoDB Tool

We use the the uri information to use Compass (and many applications).

```
(Example, use your uri)
mongodb+srv://chos5:1234hello@cluster0.spkinwx.mongodb.net/
```




New Connection

Manage your connection settings

URI: mongodb+srv://smcho:*****@cluster0.spkivnw.mongodb.net/

Edit Connection String

Name: todoapp

Color: No Color

[✓] Favorite this connection

Favoriting a connection will pin it to the top of your list of connections

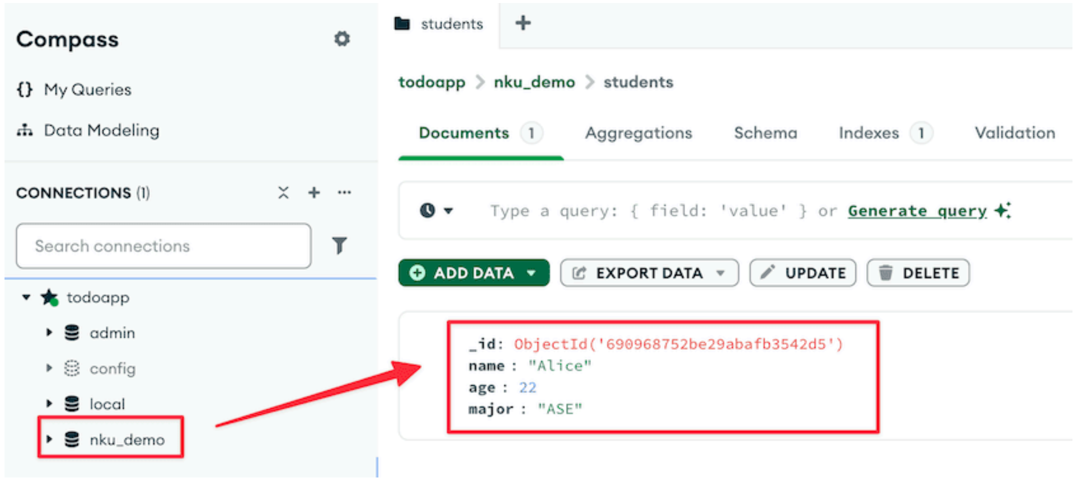
How do I find my connection string in Atlas?

If you have an Atlas cluster, go to the Cluster view. Click the 'Connect' button for the cluster to which you wish to connect.

How do I format my connection string?

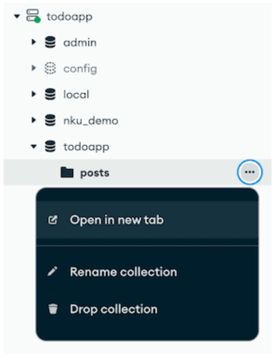
Verify your database & collection

You check the results:



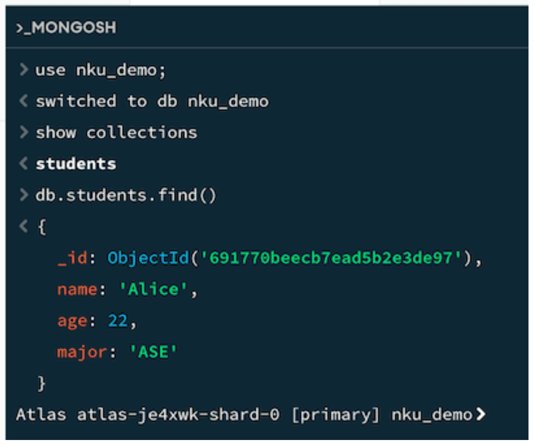
```
_id: ObjectId('690968752be29abafb3542d5')
name : "Alice"
age : 22
major : "ASE"
```

To update the screen, use "Open in new tab" function



You can also check it from the MongoDB shell in the Compass:

```
show dbs
use nku_demo
show collections
db.students.find()
```



```
> use nku_demo;
< switched to db nku_demo
> show collections
< students
> db.students.find()
< {
    _id: ObjectId('691770beecb7ead5b2e3de97'),
    name: 'Alice',
    age: 22,
    major: 'ASE'
  }
Atlas atlas-je4xwk-shard-0 [primary] nku_demo>
```

Undersatnding MongoDB

Lazy Creation

In MongoDB, databases and collections are created lazily — they don’t exist until you actually insert data into them: So, when you run something like:

```
const db = client.db("mydb");
const coll = db.collection("users");
await coll.insertOne({ name: "Alice" });
```

MongoDB automatically creates both mydb and users if they don’t already exist.

The _id field

Every MongoDB document has a special field _id that uniquely identifies it: By default, MongoDB creates it as an ObjectId — a 12-byte binary value (not a string).

Example when you insert a document:

```
{
  _id: ObjectId("672746cbe3e9d79cfafca932"),
  title: "Hello",
  date: "2025-11-03"
}
```

convert string → ObjectId

When data comes from HTTP (forms, JSON, etc.)

In Express, everything you receive from a request — body, params, query — is a string.

Example:

```
console.log(req.body._id)  // "672746cbe3e9d79cfafca932"  ← string, not ObjectId
```

MongoDB cannot match a string against a stored ObjectId:

```
await db.collection('posts').deleteOne({ _id: "672746cbe3e9d79cfafca932" })
// ❌ No document will be deleted, because the types don't match
```

Solution: ObjectId

That’s why we do:

```
const { ObjectId } = require('mongodb');
await db.collection('posts').deleteOne({ _id: new ObjectId(_id) });
```

Now the query compares ObjectId to ObjectId, and MongoDB finds the right document.

When you don’t need it

You only need ObjectId() if your _id values are ObjectIds (MongoDB’s default).

If you define _id as a plain number or string when inserting, then no conversion is needed:

```
await posts.insertOne({ _id: 1, title: 'A' });
await posts.deleteOne({ _id: 1 });  // works fine
```