MongoDB Certification Program Guide 2023

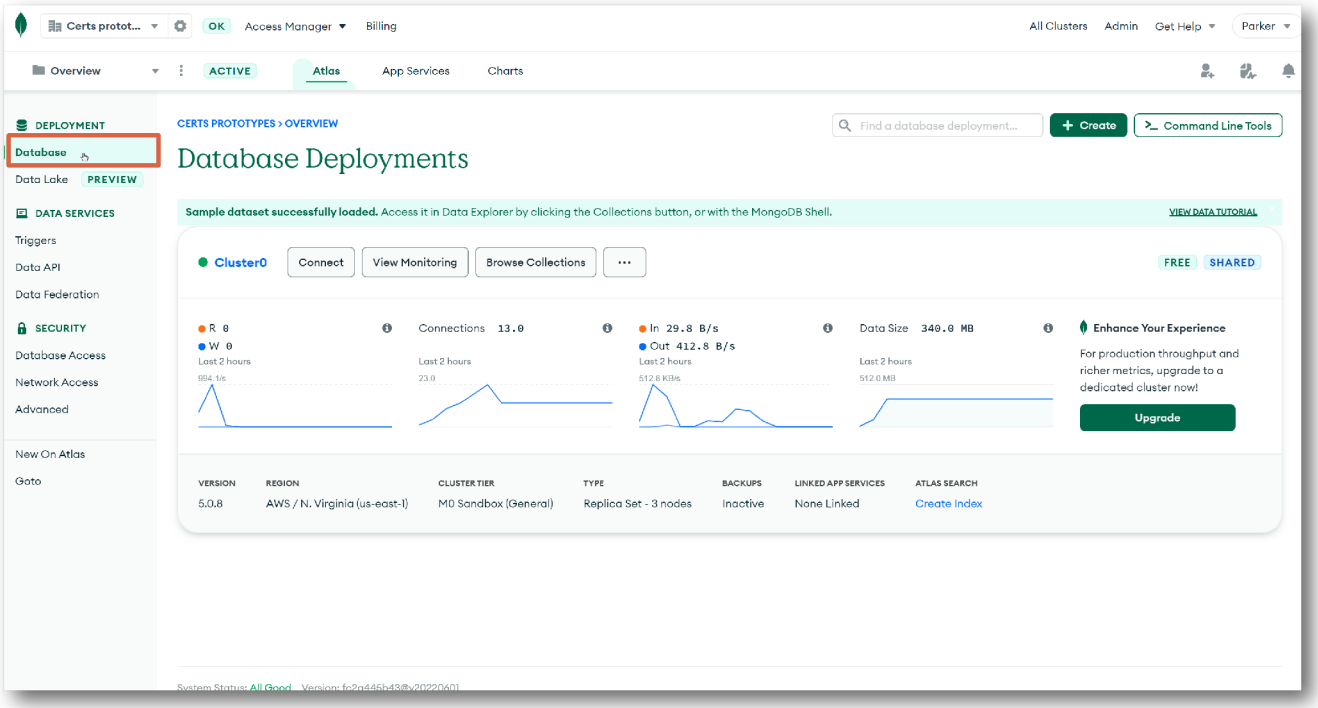
Table of Contents


```
show dbs;                                  // show the all database
use databaseName;                          // select the collection
db.dropDatabase();                         // drop the database
show collections;                          // show the list of all collections
db.createCollection("collectionName");     // create the new collections
```

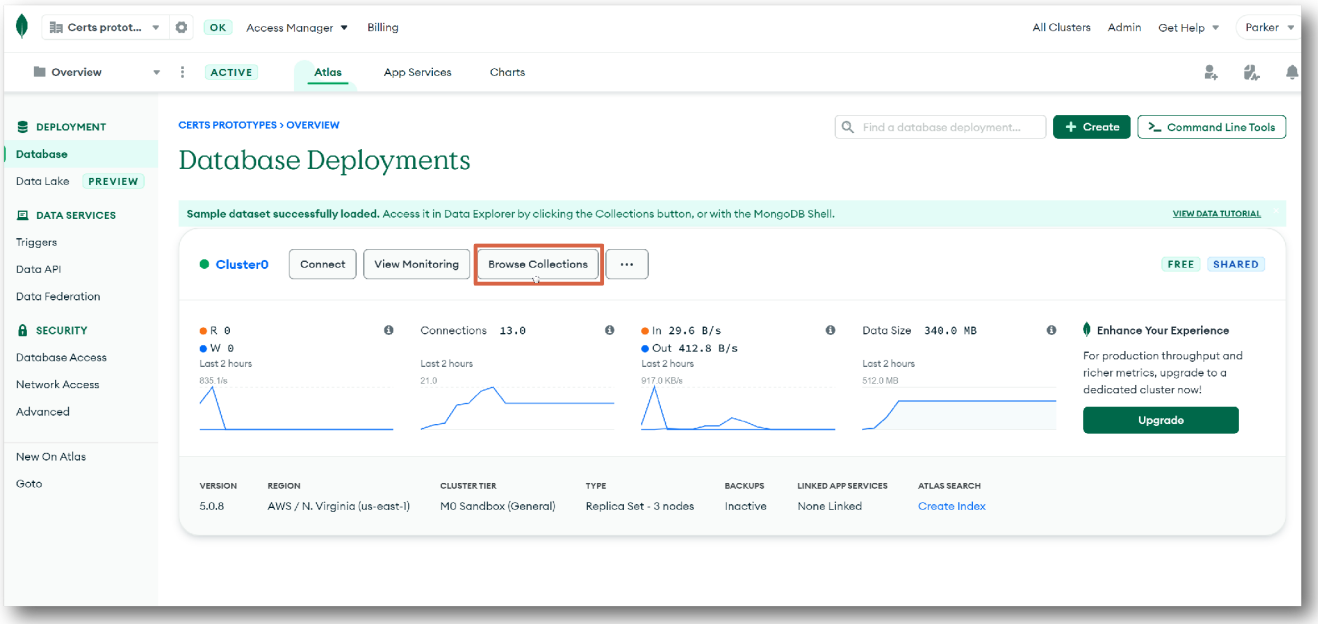
https://www.mongodb.com/languages/express-mongodb-rest-api-tutorial

Managing Databases, Collections, and Documents in Atlas Data Explorer

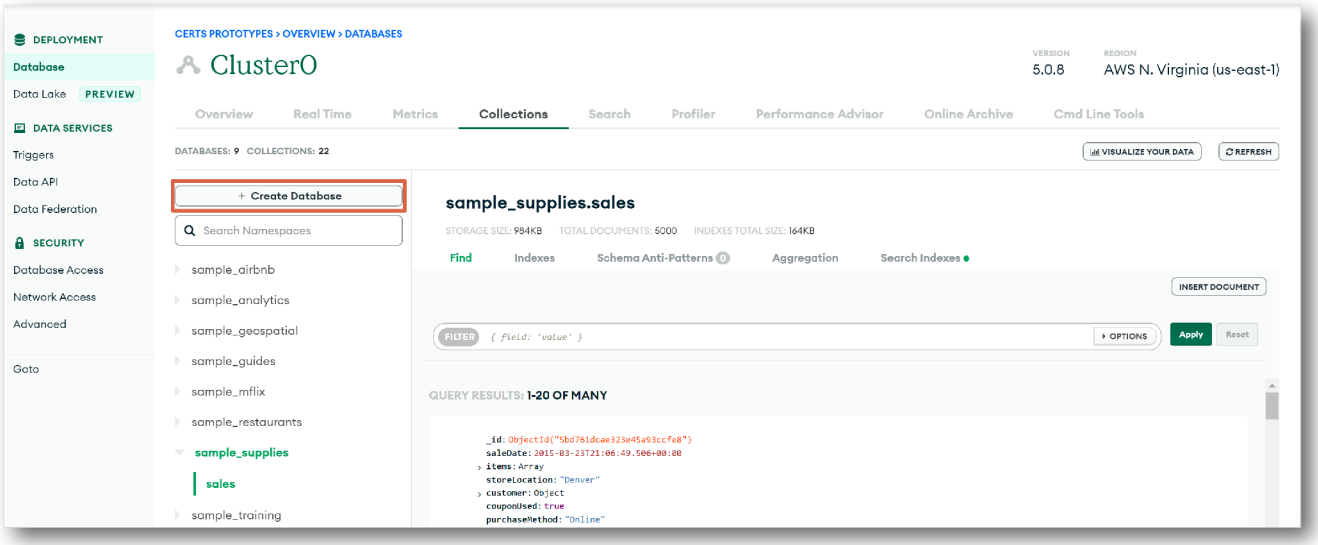
1. From the left panel in Atlas Data Explorer, click "Databases."



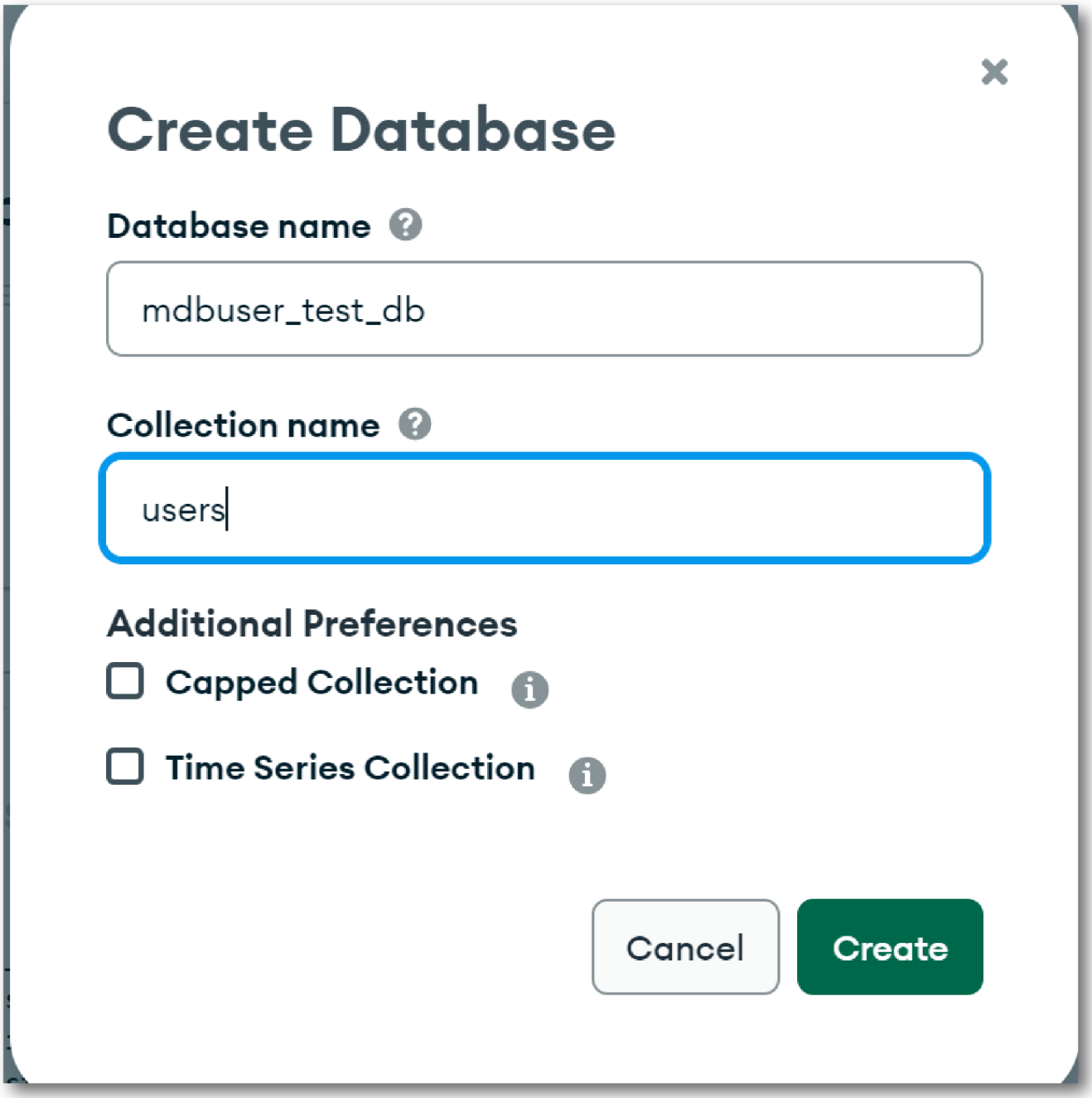
2. Click "Browse Collections."



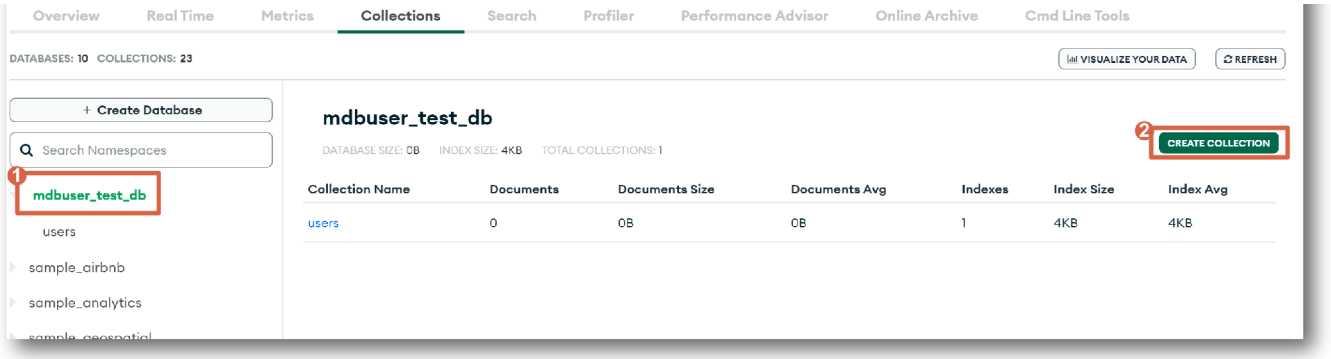
3. Click "Create Database."



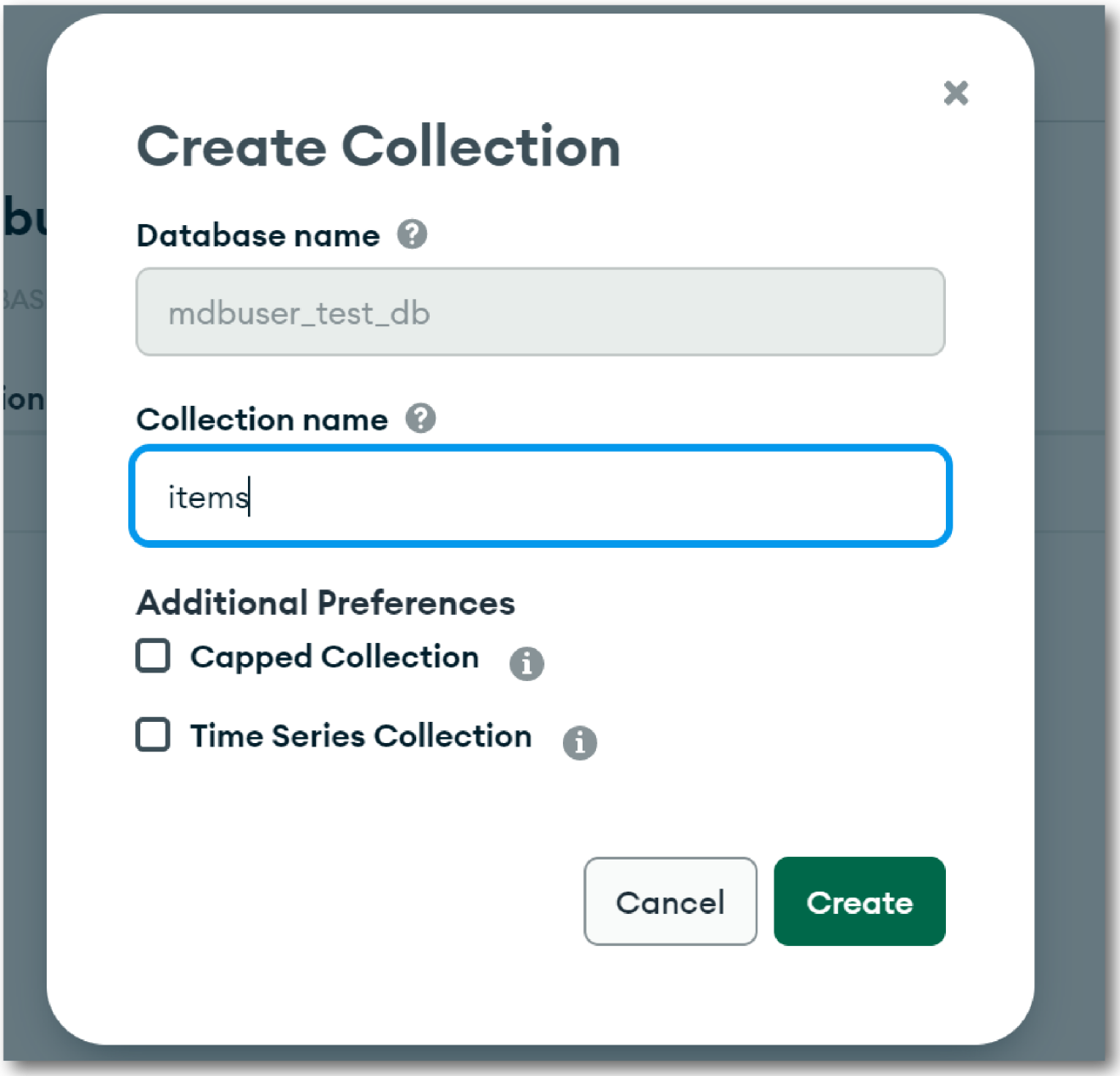
4. Enter a database name and a collection name, and then click the Create button.



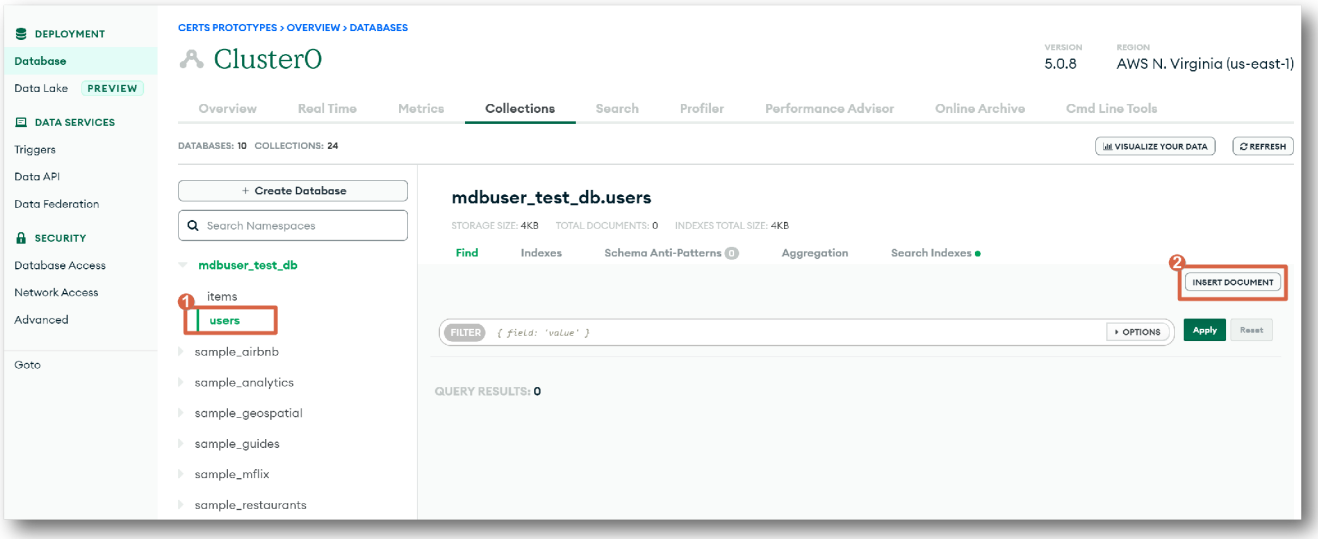
5. Confirm that your new database is listed in the left panel. To add another collection to your database, click "Create Collection."



6. Enter a new collection name, and then click the Create button.



7. To insert a document, select the collection where you want to insert it and then click "Insert Document."



8. Enter your document, and then click the Insert button.

The MongoDB Document Model

Document Structure

The values in a document can be any data type, including strings, objects, arrays, booleans, nulls, dates, ObjectIds, and more. Here's the syntax for a MongoDB document, followed by an example:

Syntax:

```
{
"key": value,
"key": value,
"key" : value
}
```

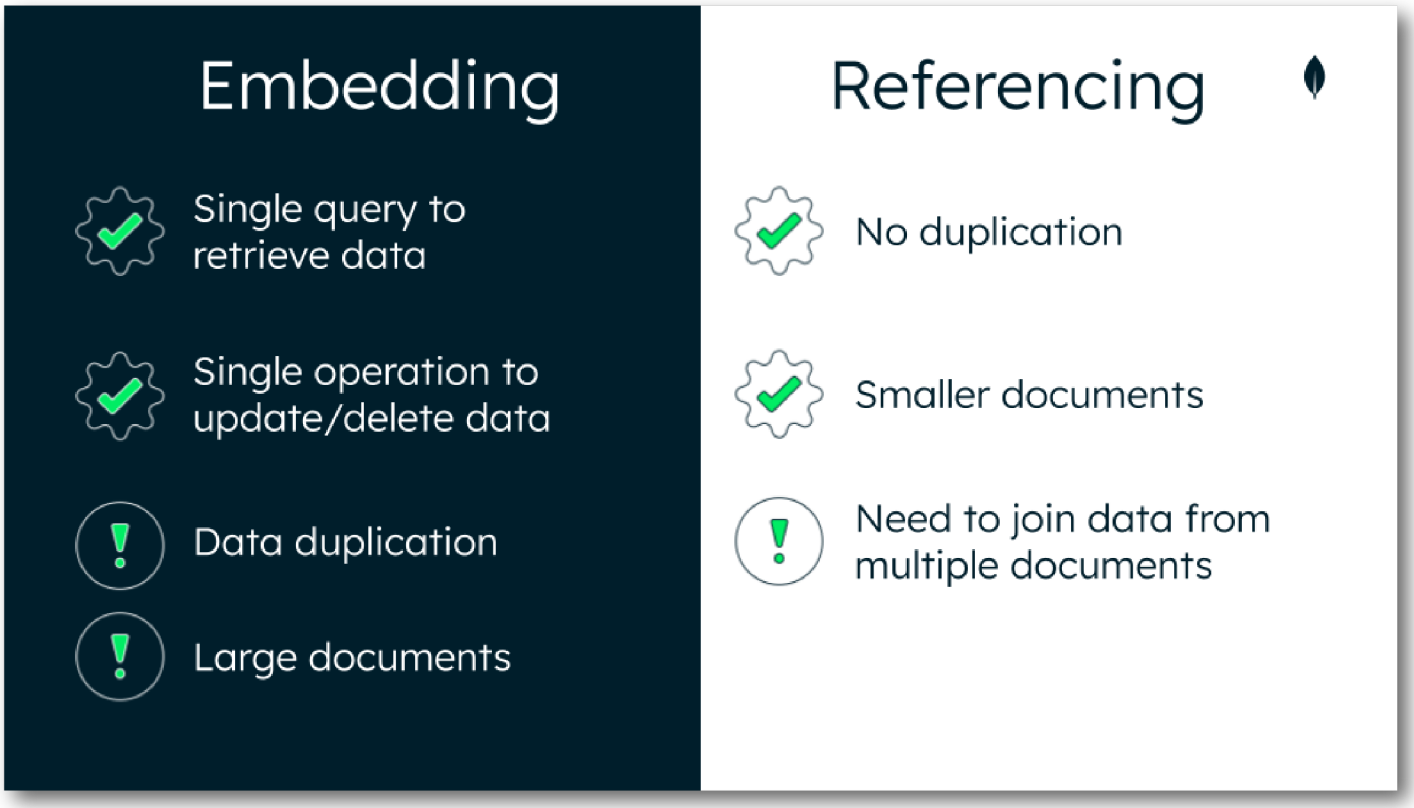
Example:

```
{
"_id": 1,
"name": "AC3 Phone",
"colors" : ["black", "silver"],
"price" : 200,
"available" : true
}
```

Lesson 5: Referencing Data in Documents / Learn

Embedding vs. Referencing

Here's a quick summary of the pro's and con's of embedding vs. referencing in MongoDB:



Introduction to MongoDB Data Modeling

In this unit, you learned how to:

- Explain the purpose of data modeling.
- Identify the types of data relationships (one to one, one to many, many to many).
- Model data relationships.
- Identify the differences between embedded and referenced data models.
- Scale a data model.
- Use Atlas Tools for schema help.

Resources

Use the following resources to learn more about the basics of data modeling:

Lesson 01: Introduction to Data Modeling

- Data Modeling Introduction
- Separating Data That is Accessed Together

Lesson 02: Types of Data Relationships

- Data Model Design
- Model Relationships Between Documents
- Embedding MongoDB
- MongoDB Schema Design Best Practices

Lesson 03: Modeling Data Relationships

- Data Model Design
- Model Relationships Between Documents

Lesson 04: Embedding Data in Documents

- Embedding MongoDB
- Model One-to-One Relationships with Embedded Documents
- Model One-to-Many Relationships with Embedded Documents

Lesson 05: Referencing Data in Documents

- Normalized Data Models
- Model One-to-Many Relationships with Document References

Lesson 06: Scaling a Data Model

- Operational Factors and Data Models
- Performance Best Practices: MongoDB Data Modeling and Memory Sizing

Lesson 07: Using Atlas Tools for Schema Help

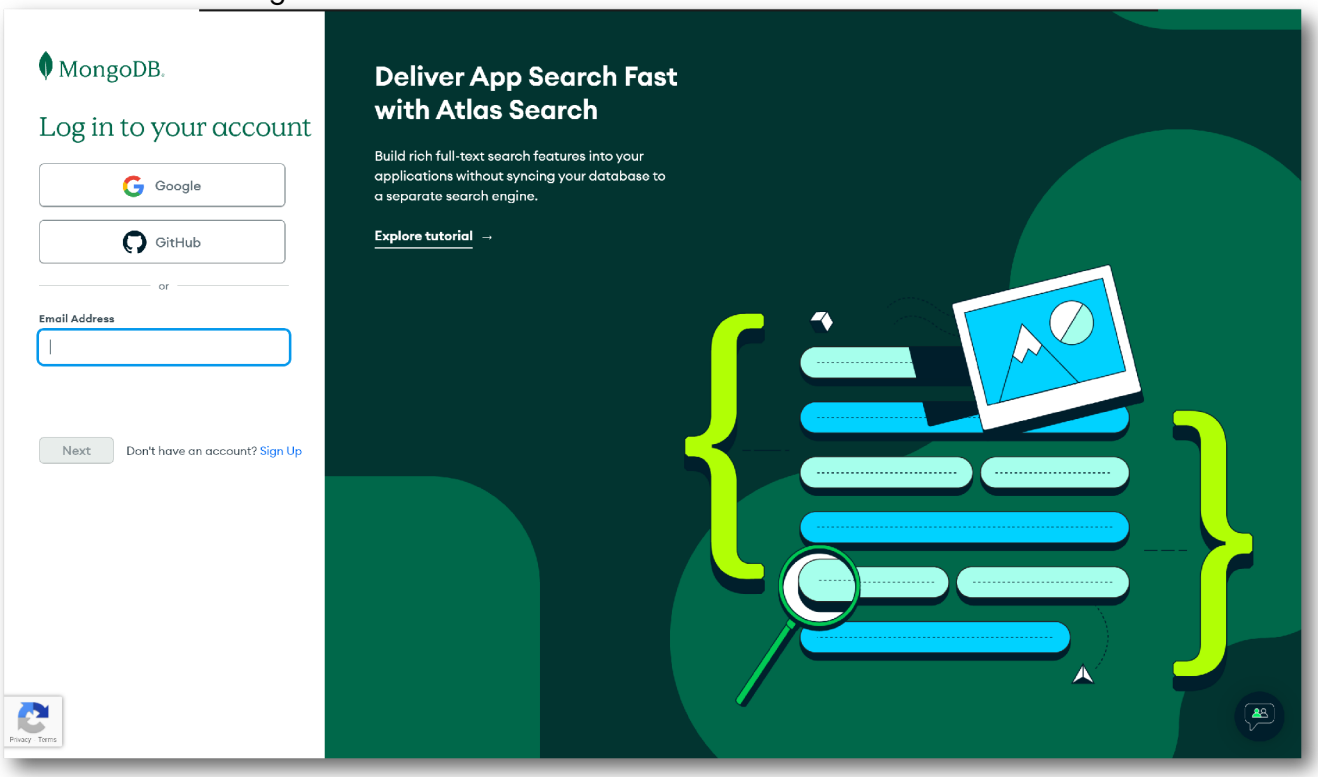
- A Summary of Schema Design Anti-Patterns and How to Spot Them

Locate the MongoDB Connection String

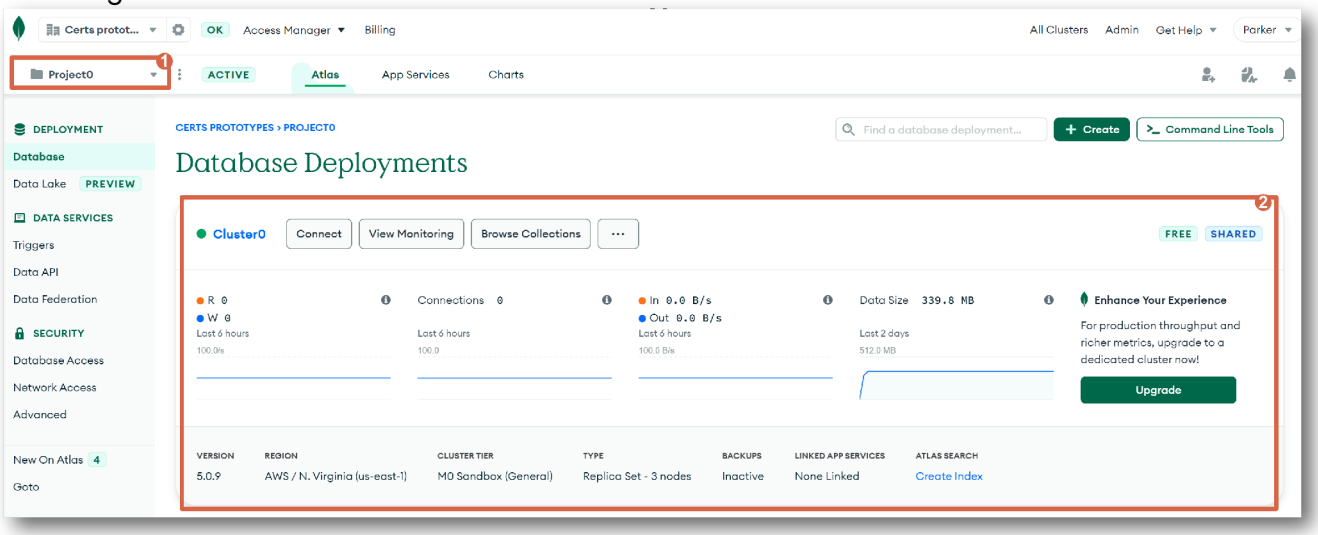
In this lesson, you will go through the steps to locate the MongoDB connection string within the Atlas dashboard.

Instructions

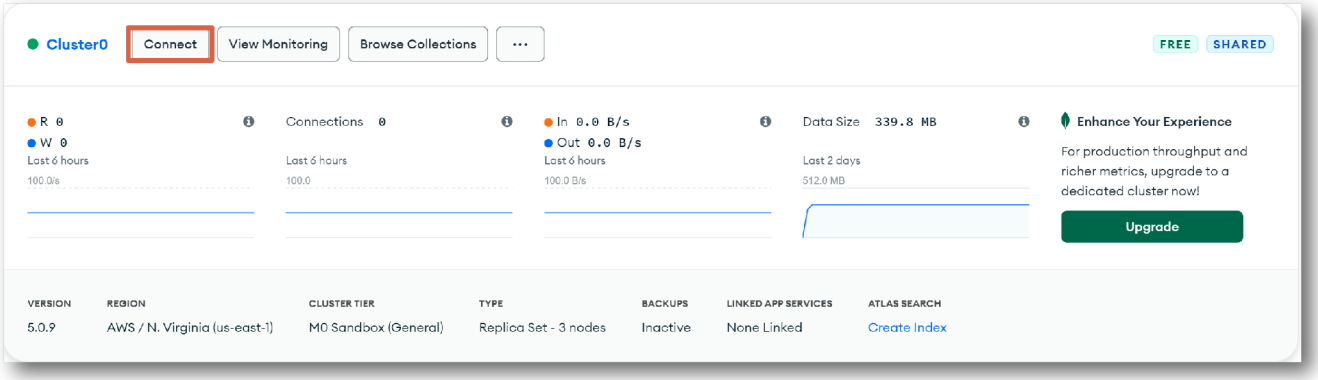
1. Go to https://account.mongodb.com/account/login and log in to Atlas. The login page should look like the following:



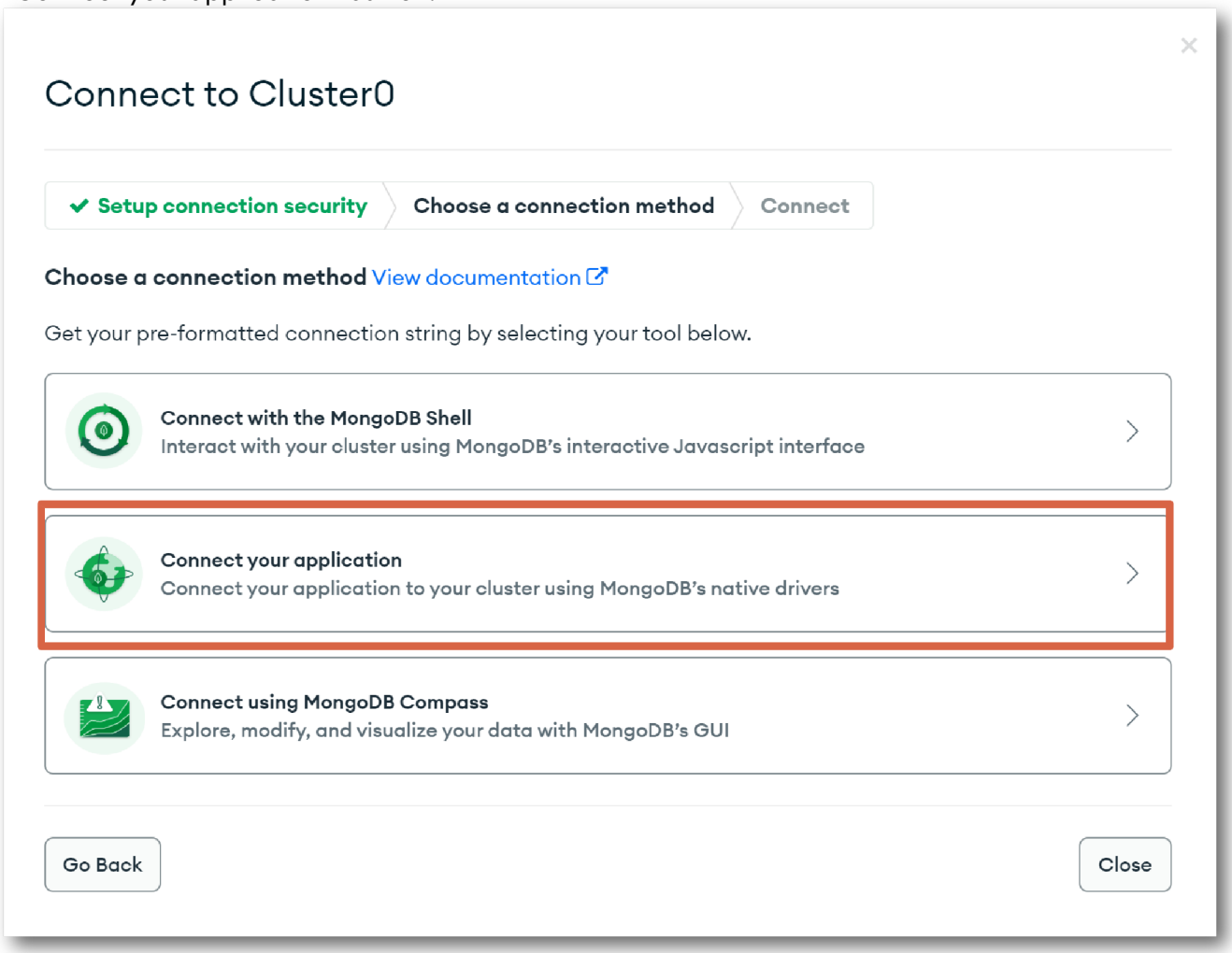
2. Once logged in, you will be taken to the Atlas dashboard for your current project, as shown in the following screenshot:



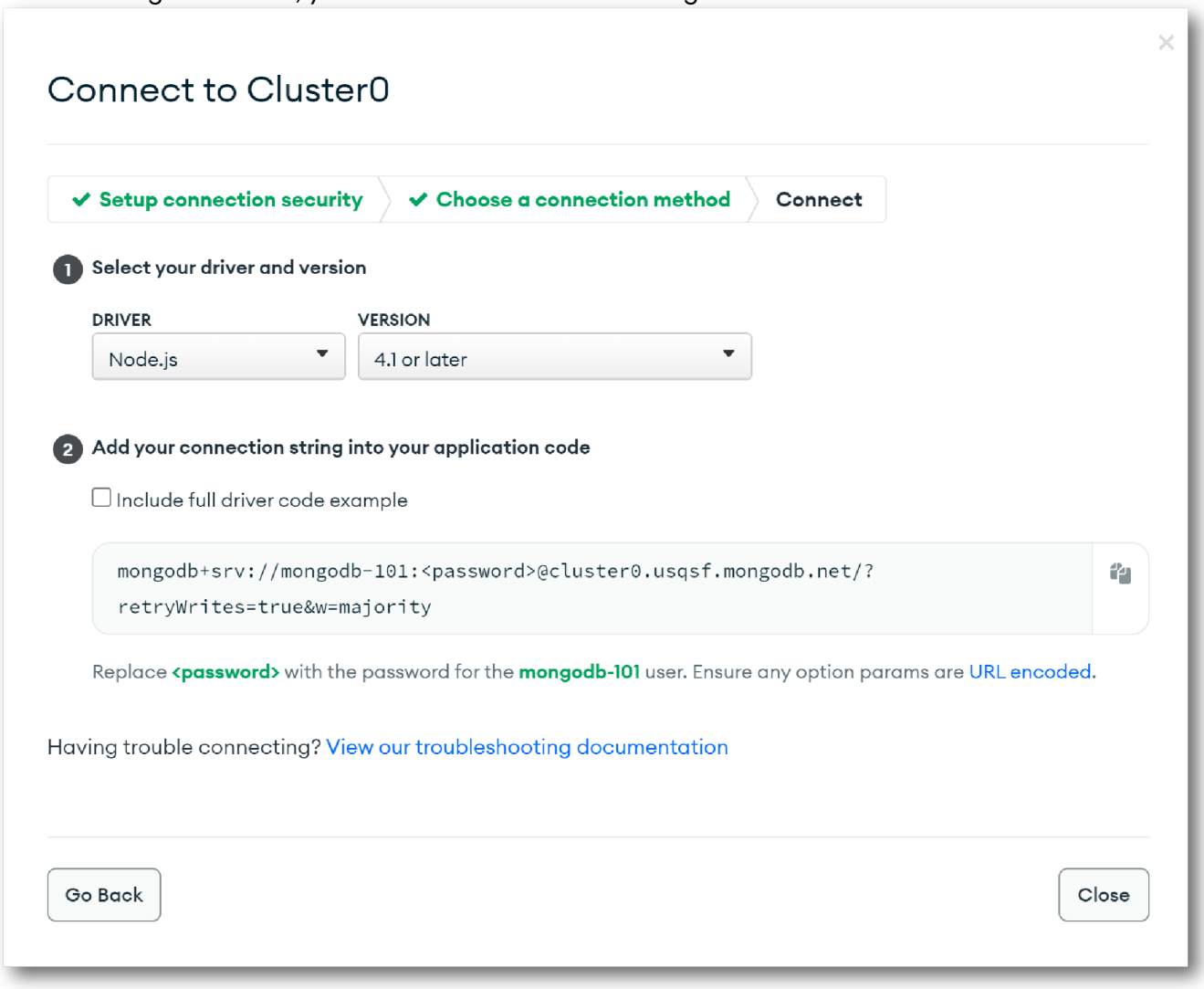
3. Click the Connect button.



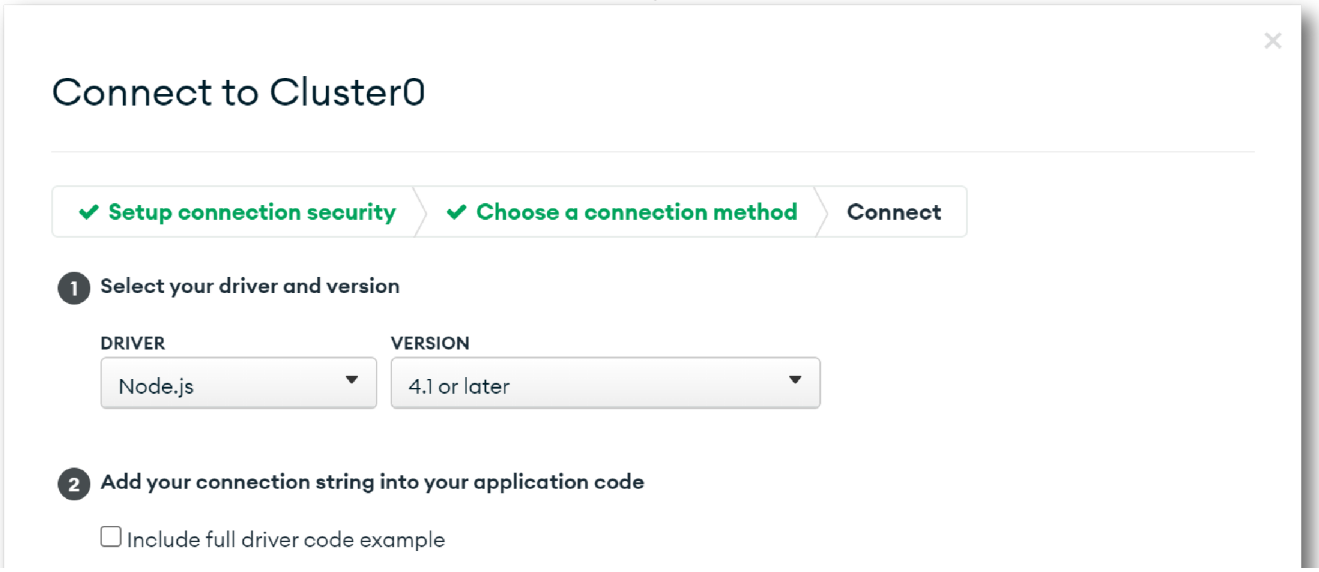
4. This should open a new window, which shows options for connecting to your cluster. Click the "Connect your application" button.

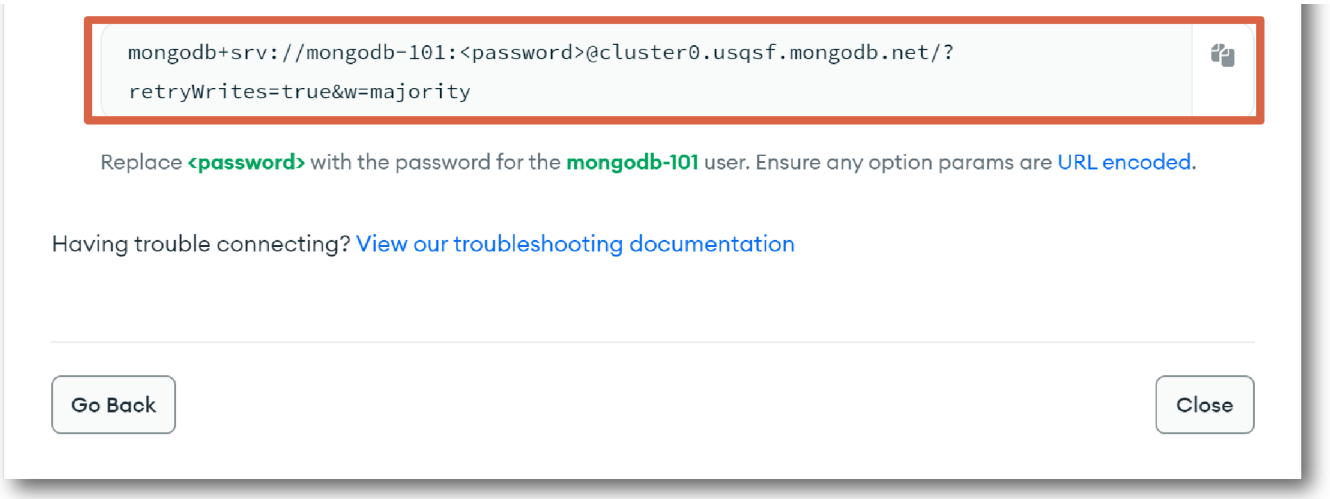


5. After clicking the button, you will be taken to the following window:



6. Click the copy icon to copy your connection string.





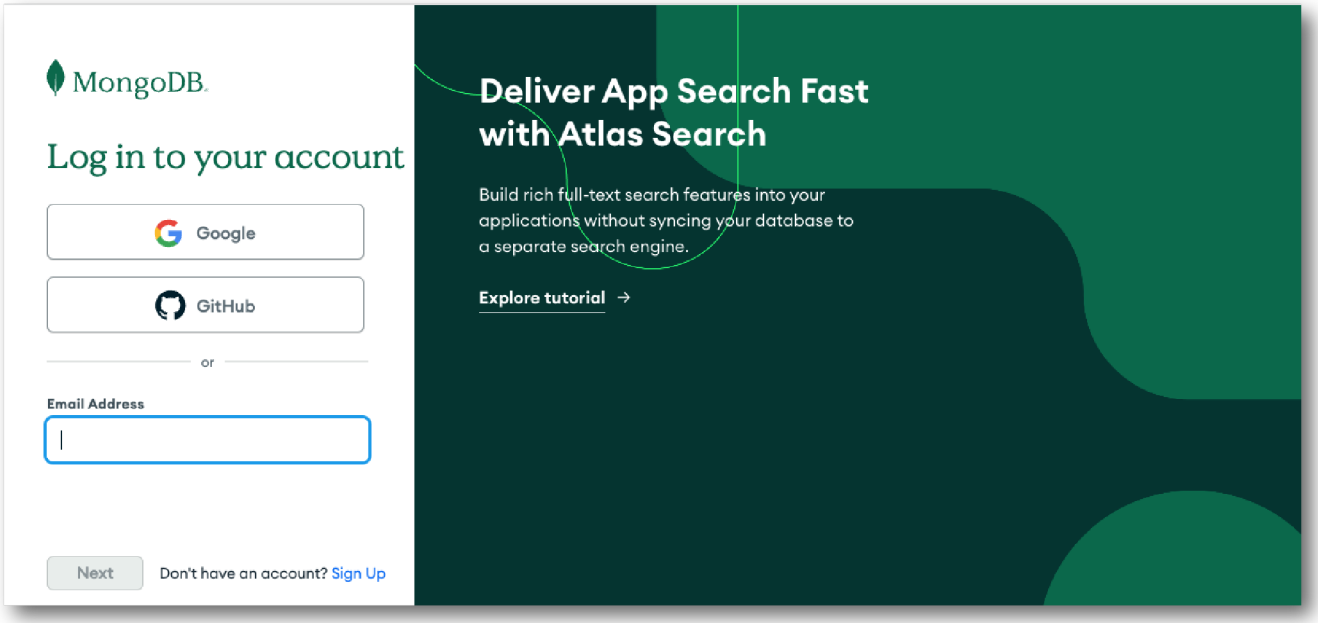
Now you can use your connection string to connect to your Atlas cluster!

Troubleshooting a MongoDB Connection in Node.js Applications

In this lesson, you learned how to update your network access configuration by adding your current IP address to the access list.

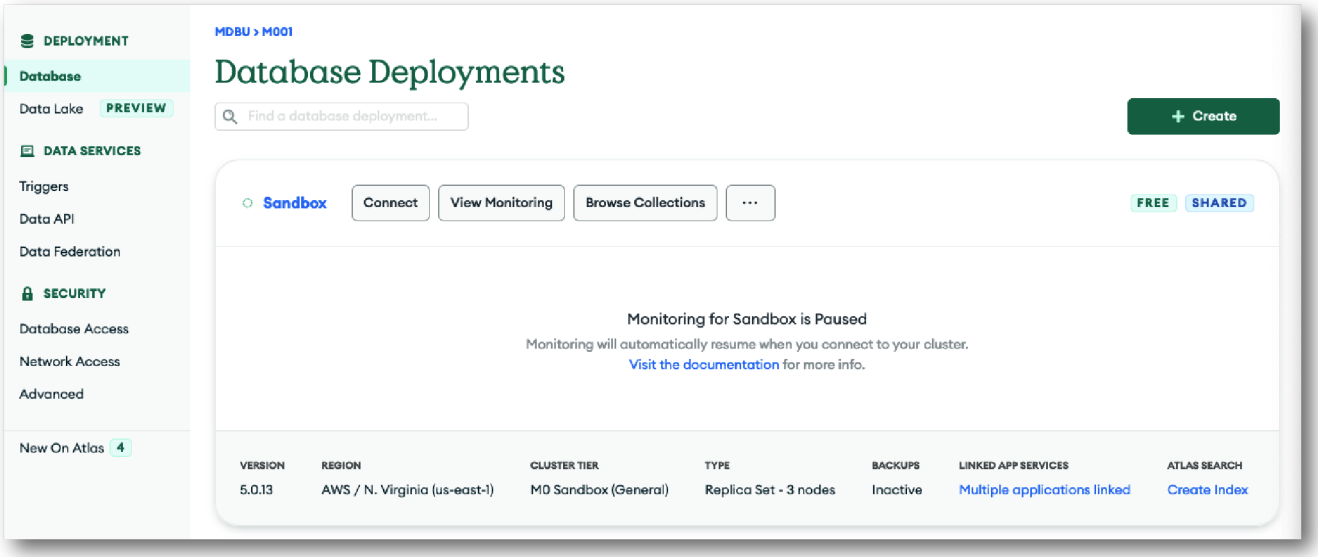
Instructions

1. Go to https://account.mongodb.com/account/login and log in into Atlas. The login page looks like the following:

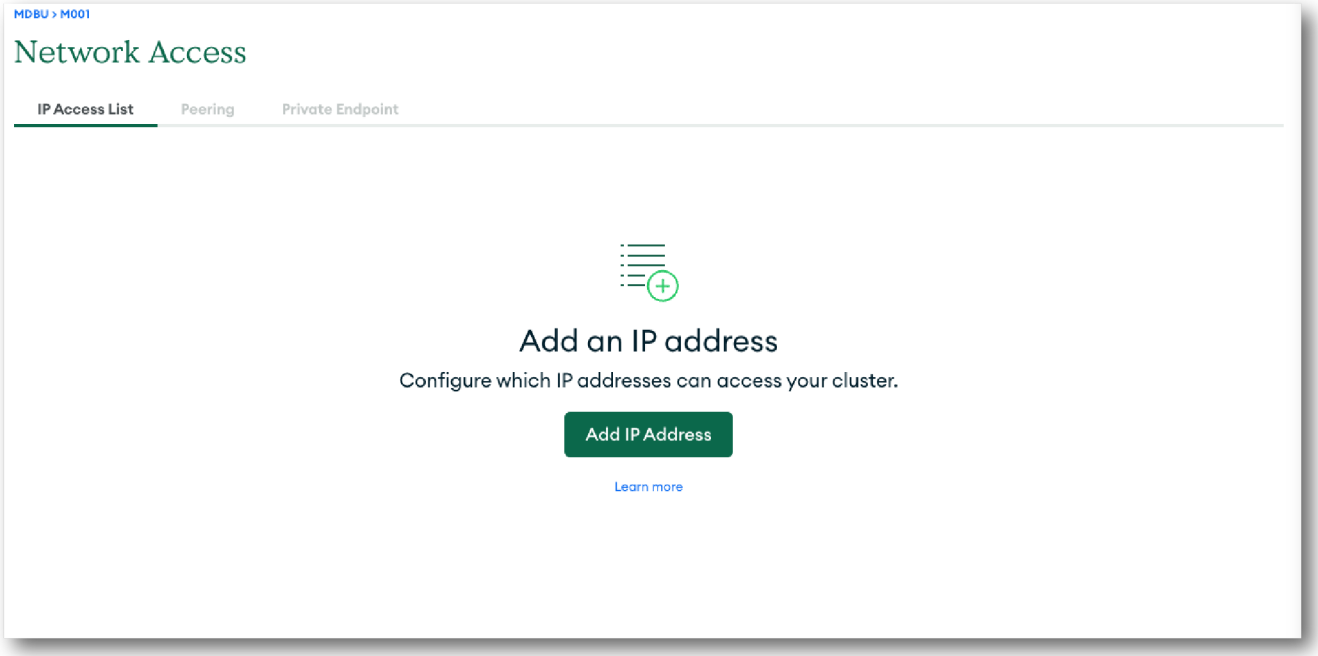


2. Once you're logged in, you should be taken to the Atlas dashboard for your current project.

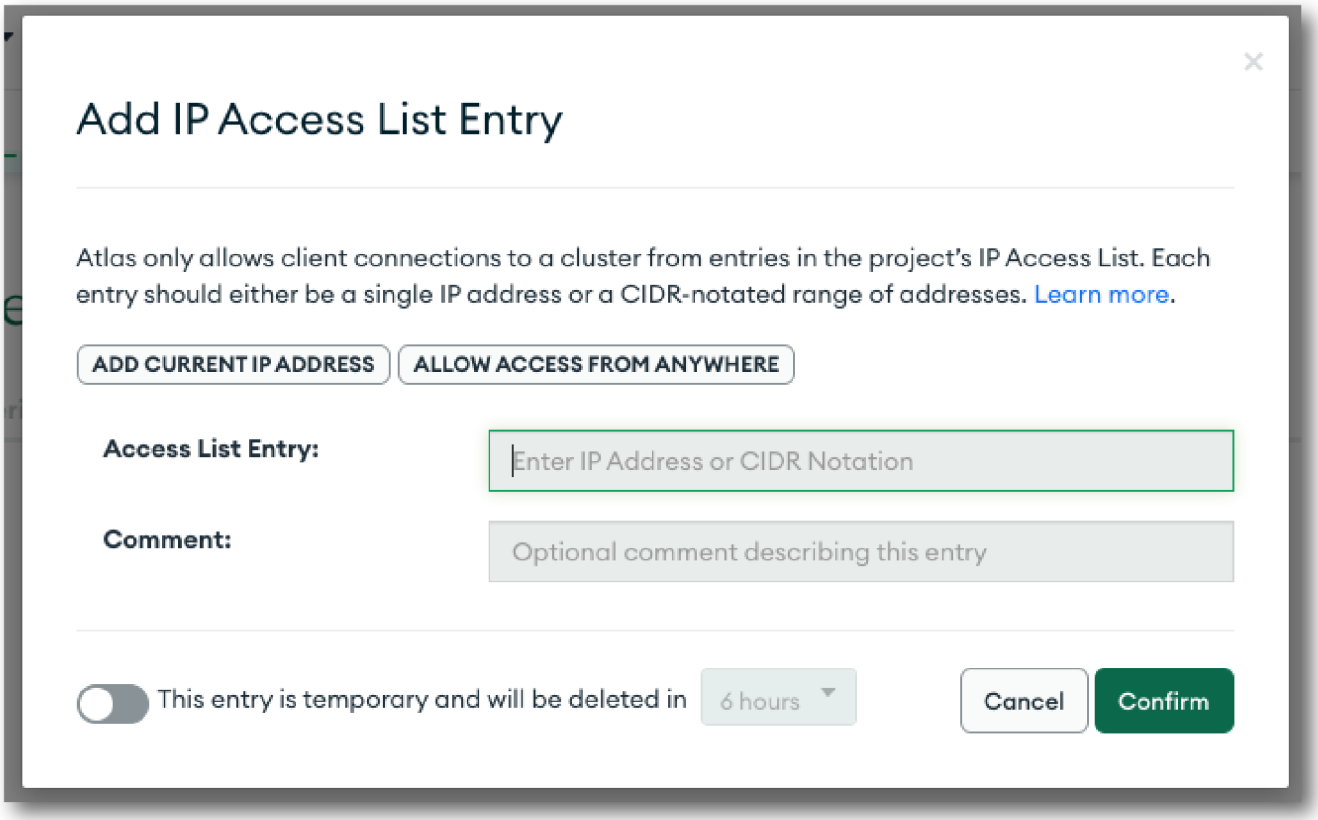
3. Go to Network Access.



4. Click the “Add IP Address” button.



5. Select the "Add Current IP Address" option.



6. Click the Confirm button.

Connecting to MongoDB in Node.js

In this unit, you learned how to:

- Install the Node.js MongoDB driver with Yarn.
- Connect to your Atlas cluster from a Node.js application.
- Use the MongoClient instance.
- Troubleshoot common connection issues when using the MongoDB driver.

Resources

Use the following resources to learn more about using the MongoDB driver to connect your Node.js application to MongoDB:

Lesson 01: Using MongoDB Node.js Client Libraries

- MongoDB Drivers
- MongoDB Node.js Driver Docs

Lesson 02: Connecting to an Atlas Cluster in Node.js Applications

- MongoDB Docs: Get Connection String
- MongoDB Docs: Connection String Format
- MongoDB Node.js Connection to Atlas

Lesson 03: Troubleshooting a MongoDB Connection in Node.js Applications

- MongoDB Docs: Troubleshoot Connection Issues

Using MongoDB with Node.js Course

By completing this unit, you've taken a step toward completing the Using MongoDB with Node.js course.

If you're interested in continuing, your next step is to review the following unit:

- Unit 02: MongoDB CRUD Operations in Node.js

Inserting Documents in a MongoDB Collection

Insert a Single Document

Use `insertOne()` to insert a document into a collection. Within the parentheses of `insertOne()`, include an object that contains the document data. Here's an example:

```
db.grades.insertOne({
  student_id: 654321,
  products: [
    {
      type: "exam",
      score: 90,
    },
    {
      type: "homework",
      score: 59,
    },
    {
      type: "quiz",
      score: 75,
    },
    {
      type: "homework",
      score: 88,
    },
  ],
  class_id: 550,
})
```

Insert Multiple Documents

Use `insertMany()` to insert multiple documents at once. Within `insertMany()`, include the documents within an array. Each document should be separated by a comma. Here's an example:

```
db.grades.insertMany([
  {
    student_id: 546789,
    products: [
      {
        type: "quiz",
        score: 50,
      },
      {
        type: "homework",
        score: 70,
      },
      {
        type: "quiz",
        score: 66,
      },
      {
        type: "exam",
        score: 70,
      },
    ],
    class_id: 551,
  },
  {
    student_id: 777777,
    products: [
      {
        type: "exam",
        score: 83,
      },
      {
        type: "quiz",
        score: 59,
      },
      {
        type: "quiz",
        score: 72,
```

```
    },
    {
      type: "quiz",
      score: 67,
    },
  ],
  class_id: 550,
},
{
  student_id: 223344,
  products: [
    {
      type: "exam",
      score: 45,
    },
    {
      type: "homework",
      score: 39,
    },
    {
      type: "quiz",
      score: 40,
    },
    {
      type: "homework",
      score: 88,
    },
  ],
  class_id: 551,
},
])
```

Finding Documents in a MongoDB Collection

Review the following code, which demonstrates how to query documents in MongoDB.

Find a Document with Equality

When given equality with an `_id` field, the `find()` command will return the specified document that matches the `_id`. Here's an example:

```
db.zips.find({ _id: ObjectId("5c8eccc1caa187d17ca6ed16") })
```

Find a Document by Using the `$in` Operator

Use the `$in` operator to select documents where the value of a field equals any value in the specified array. Here's an example:

```
db.zips.find({ city: { $in: ["PHOENIX", "CHICAGO"] } })
```

Finding Documents by Using Comparison Operators

Review the following comparison operators: `$gt`, `$lt`, `$lte`, and `$gte`.

$gt

Use the `$gt` operator to match documents with a field **greater than** the given value. For example:

```
db.sales.find({ "items.price": { $gt: 50}})
```

$lt

Use the `$lt` operator to match documents with a field **less than** the given value. For example:

```
db.sales.find({ "items.price": { $lt: 50}})
```

$lte

Use the `$lte` operator to match documents with a field **less than or equal to** the given value. For example:

```
db.sales.find({ "customer.age": { $lte: 65}})
```

$gte

Use the `$gte` operator to match documents with a field **greater than or equal to** the given value. For example:

```
db.sales.find({ "customer.age": { $gte: 65}})
```

Lesson 4: Querying on Array Elements in MongoDB / Learn

Querying on Array Elements in MongoDB

Review the following code, which demonstrates how to query array elements in MongoDB.

Find Documents with an Array That Contains a Specified Value

In the following example, "InvestmentFund" is not enclosed in square brackets, so MongoDB returns all documents within the `products` array that contain the specified value.

```
db.accounts.find({ products: "InvestmentFund"})
```

Find a Document by Using the `$elemMatch` Operator

Use the `$elemMatch` operator to find all documents that contain the specified subdocument. For example:

```
db.sales.find({
  items: {
    $elemMatch: { name: "laptop", price: { $gt: 800 }, quantity: { $gte: 1 } },
  },
})
```

Finding Documents by Using Logical Operators

Review the following logical operators: implicit `$and`, `$or`, and `$and`.

Find a Document by Using Implicit `$and`

Use implicit `$and` to select documents that match multiple expressions. For example:

```
db.routes.find({ "airline.name": "Southwest Airlines", stops: { $gte: 1 } })
```

Find a Document by Using the `$or` Operator

Use the `$or` operator to select documents that match at least one of the included expressions. For example:

```
db.routes.find({
  $or: [{ dst_airport: "SEA" }, { src_airport: "SEA" }],
})
```

Find a Document by Using the `$and` Operator

Use the `$and` operator to use multiple `$or` expressions in your query.

```
db.routes.find({
  $and: [
    { $or: [{ dst_airport: "SEA" }, { src_airport: "SEA" }] },
    { $or: [{ "airline.name": "American Airlines" }, { airplane: 320 }] },
  ]
})
```

MongoDB CRUD Operations: Insert and Find Documents

In this unit, you learned how to insert and find documents in a MongoDB collection. You built queries by using the following comparison operators:

- `$gt` (greater than)
- `$lt` (less than)
- `$lte` (less than or equal to)
- `$gte` (greater than or equal to)

You also used the following logical operators:

- `$and`
- `$or`

Finally, you learned how to query elements in an array and how to use the `$elemMatch` operator.

Resources

Use the following resources to learn more about inserting and finding documents in MongoDB:

Lesson 01: Inserting Documents in a MongoDB Collection

- MongoDB Docs: insertOne()
- MongoDB Docs: insertMany()

Lesson 02: Finding Documents in a MongoDB Collection

- MongoDB Docs: find()
- MongoDB Docs: $in

Lesson 03: Finding Documents by Using Comparison Operators

- MongoDB Docs: Comparison Operators

Lesson 04: Querying on Array Elements in MongoDB

- MongoDB Docs: $elemMatch
- MongoDB Docs: Querying Arrays

Lesson 05: Finding Documents by Using Logical Operators

- MongoDB Docs: Logical Operators

Lesson 1: Replacing a Document in MongoDB / Learn

Replacing a Document in MongoDB

To replace documents in MongoDB, we use the `replaceOne()` method. The `replaceOne()` method takes the following parameters:

- `filter`: A query that matches the document to replace.
- `replacement`: The new document to replace the old one with.
- `options`: An object that specifies options for the update.

In the previous video, we use the `_id` field to filter the document. In our replacement document, we provide the entire document that should be inserted in its place. Here's the example code from the video:

```
db.books.replaceOne(
  {
    _id: ObjectId("6282afeb441a74a98dbbec4e"),
  },
  {
    title: "Data Science Fundamentals for Python and MongoDB",
    isbn: "1484235967",
    publishedDate: new Date("2018-5-10"),
    thumbnailUrl:
      "https://m.media-amazon.com/images/I/71opmUBc2wL._AC_UY218_.jpg",
    authors: ["David Paper"],
    categories: ["Data Science"],
  }
)
```

Updating MongoDB Documents by Using `updateOne()`

The `updateOne()` method accepts a filter document, an update document, and an optional options object. MongoDB provides update operators and options to help you update documents. In this section, we'll cover three of them: `$set`, `upsert`, and `$push`.

`$set`

The `$set` operator replaces the value of a field with the specified value, as shown in the following code:

```
db.podcasts.updateOne(
  {
    _id: ObjectId("5e8f8f8f8f8f8f8f8f8f8f8"),
  },

  {
    $set: {
      subscribers: 98562,
    },
  }
)
```

`upsert`

The `upsert` option creates a new document if no documents match the filtered criteria. Here's an example:

```
db.podcasts.updateOne(
  { title: "The Developer Hub" },
  { $set: { topics: ["databases", "MongoDB"] } },
  { upsert: true }
)
```

`$push`

The `$push` operator adds a new value to the `hosts` array field. Here's an example:

```
db.podcasts.updateOne(
  { _id: ObjectId("5e8f8f8f8f8f8f8f8f8f8f8") },
  { $push: { hosts: "Nic Raboy" } }
)
```

Lesson 3: Updating MongoDB Documents by Using findAndModify() / Learn

Updating MongoDB Documents by Using `findAndModify()`

The `findAndModify()` method is used to find and replace a single document in MongoDB. It accepts a filter document, a replacement document, and an optional options object. The following code shows an example:

```
db.podcasts.findAndModify({
  query: { _id: ObjectId("6261a92dfee1ff300dc80bf1") },
  update: { $inc: { subscribers: 1 } },
  new: true,
})
```

Lesson 4: Updating MongoDB Documents by Using updateMany() / Learn

Updating MongoDB Documents by Using `updateMany()`

To update multiple documents, use the `updateMany()` method. This method accepts a filter document, an update document, and an optional options object. The following code shows an example:

```
db.books.updateMany(
  { publishedDate: { $lt: new Date("2019-01-01") } },
  { $set: { status: "LEGACY" } }
)
```

MongoDB CRUD Operations: Replace and Delete Documents

In this unit, you learned how to modify query results with MongoDB. Specifically, you:

- Replaced a single document by using `db.collection.replaceOne()`.
- Updated a field value by using the `$set` update operator in `db.collection.updateOne()`.
- Added a value to an array by using the `$push` update operator in `db.collection.updateOne()`.
- Added a new field value to a document by using the `upsert` option in `db.collection.updateOne()`.
- Found and modified a document by using `db.collection.findAndModify()`.
- Updated multiple documents by using `db.collection.updateMany()`.
- Deleted a document by using `db.collection.deleteOne()`.

Resources

Use the following resources to learn more about modifying query results in MongoDB:

Lesson 01: Replacing a Document in MongoDB

- MongoDB Docs: replaceOne()

Lesson 02: Updating MongoDB Documents by Using `updateOne()`

- MongoDB Docs: Update Operators
- MongoDB Docs: $set
- MongoDB Docs: $push
- MongoDB Docs: upsert

Lesson 03: Updating MongoDB Documents by Using `findAndModify()`

- MongoDB Docs: findAndModify()

Lesson 04: Updating MongoDB Documents by Using `findAndModify()`

- MongoDB Docs: updateMany()

Lesson 05: Deleting Documents in MongoDB

- MongoDB Docs: deleteOne()
- MongoDB Docs: deleteMany()

Lesson 5: Deleting Documents in MongoDB / Learn

Deleting Documents in MongoDB

To delete documents, use the `deleteOne()` or `deleteMany()` methods. Both methods accept a filter document and an options object.

Delete One Document

The following code shows an example of the `deleteOne()` method:

```
db.podcasts.deleteOne({ _id: Objectid("6282c9862acb966e76bbf20a") })
```

Delete Many Documents

The following code shows an example of the `deleteMany()` method:

```
db.podcasts.deleteMany({category: “crime”})
```

Sorting and Limiting Query Results in MongoDB

Review the following code, which demonstrates how to sort and limit query results.

Sorting Results

Use `cursor.sort()` to return query results in a specified order. Within the parentheses of `sort()`, include an object that specifies the field(s) to sort by and the order of the sort. Use 1 for ascending order, and -1 for descending order.

Syntax:

```
db.collection.find(<query>).sort(<sort>)
```

Example:

```
// Return data on all music companies, sorted alphabetically from A to Z.
db.companies.find({ category_code: "music" }).sort({ name: 1 });
```

To ensure documents are returned in a consistent order, include a field that contains unique values in the sort. An easy way to do this is to include the `_id` field in the sort. Here's an example:

```
// Return data on all music companies, sorted alphabetically from A to Z. Ensure consisten
db.companies.find({ category_code: "music" }).sort({ name: 1, _id: 1 });
```

Limiting Results

Use `cursor.limit()` to return query results in a specified order. Within the parentheses of `limit()`, specify the maximum number of documents to return.

Syntax:

```
db.companies.find(<query>).limit(<number>)
```

Example:

```
// Return the three music companies with the highest number of employees. Ensure consisten
db.companies
  .find({ category_code: "music" })
  .sort({ number_of_employees: -1, _id: 1 })
  .limit(3);
```

Returning Specific Data from a Query in MongoDB

Add a Projection Document

To specify fields to include or exclude in the result set, add a projection document as the second parameter in the call to `db.collection.find()`.

Syntax:

```
db.collection.find( <query>, <projection> )
```

Include a Field

To include a field, set its value to 1 in the projection document.

Syntax:

```
db.collection.find( <query>, { <field> : 1 })
```

Example:

```
// Return all restaurant inspections - business name, result, and _id fields only
db.inspections.find(
  { sector: "Restaurant - 818" },
  { business_name: 1, result: 1 }
)
```

Exclude a Field

To exclude a field, set its value to 0 in the projection document.

Syntax:

```
db.collection.find(query, { <field> : 0, <field>: 0 })
```

Example:

```
// Return all inspections with result of "Pass" or "Warning" - exclude date and zip code
db.inspections.find(
  { result: { $in: ["Pass", "Warning"] } },
  { date: 0, "address.zip": 0 }
)
```

While the `_id` field is included by default, it can be suppressed by setting its value to 0 in any projection.

```
// Return all restaurant inspections - business name and result fields only
db.inspections.find(
  { sector: "Restaurant - 818" },
  { business_name: 1, result: 1, _id: 0 }
)
```

Counting Documents in a MongoDB Collection

Review the following code, which demonstrates how to count the number of documents that match a query.

Count Documents

Use `db.collection.countDocuments()` to count the number of documents that match a query. `countDocuments()` takes two parameters: a query document and an options document.

Syntax:

```
db.collection.countDocuments( <query>, <options> )
```

The query selects the documents to be counted.

Examples:

```
// Count number of docs in trip collection
db.trips.countDocuments({})
```

```
// Count number of trips over 120 minutes by subscribers
db.trips.countDocuments({ tripduration: { $gt: 120 }, usertype: "Subscriber" })
```

MongoDB CRUD Operations: Modifying Query Results

In this unit, you learned how to modify query results with MongoDB. Specifically, you learned how to:

- Return query results in a specified order by using `cursor.sort()`.
- Constrained the number of results returned by using `cursor.limit()`.
- Specified fields to return by adding a projection document parameter in calls to `db.collection.find()`.
- Counted the number of documents that match a query by using `db.collection.countDocuments()`.

Resources

Use the following resources to learn more about modifying query results in MongoDB:

Lesson 01: Sorting and Limiting Query Results in MongoDB

- MongoDB Docs: cursor.sort()
- MongoDB Docs: cursor.limit()

Lesson 02: Returning Specific Data from a Query in MongoDB

- MongoDB Docs: Project Fields to Return from Query
- MongoDB Docs: Projection Restrictions

Lesson 03: Counting Documents in a MongoDB Collection

- MongoDB Docs: db.collection.countDocuments()

MongoDB CRUD Operations in Node.js

In this unit, you learned how to:

- Express documents in Node.js
- Insert documents by using `insertOne()` and `insertMany()`.
- Query documents by using `findOne()` and `find()`.
- Delete documents by using `deleteOne()` and `deleteMany()`.
- Create a multi-document transaction.

Resources

Use the following resources to learn more about performing basic CRUD with Node.js:

Lesson 01: Working with MongoDB Documents in Node.js

- Storing Documents in MongoDB
- Overview of BSON

Lesson 02: Inserting a Document in Node.js Applications

- Insert Documents by Using Node.js

Lesson 03: Querying a MongoDB Collection in Node.js Applications

- Retrieve Data by Using Node.js

Lesson 04: Updating Documents in Node.js Applications

- Update Documents by Using Node.js

Lesson 05: Deleting Documents in Node.js Applications

- Deleting Documents using Node.js

Lesson 06: Creating MongoDB Transactions in Node.js Applications

- Transactions in Node.js

Using MongoDB with Node.js Course

By completing this unit, you've taken a step toward completing the Using MongoDB with Node.js course. If you're interested in continuing, your next step is to review the following unit:

- Unit 03: MongoDB Aggregation with Node.js

Inserting a Document in Node.js Applications

Insert a Document

To insert a single document into a collection, append `insertOne()` to the collection variable. The `insertOne()` method accepts a document as an argument and returns a promise. In this example, the document that's being inserted is stored in a variable called `sampleAccount`, which is declared just above the `main()` function.

```
const dbname = "bank"
const collection_name = "accounts"

const accountsCollection = client.db(dbname).collection(collection_name)

const sampleAccount = {
 account_holder: "Linus Torvalds",
 account_id: "MDB829001337",
 account_type: "checking",
 balance: 50352434,
}

const main = async () => {
 try {
   await connectToDatabase()
   // insertOne method is used here to insert the sampleAccount document
   let result = await accountsCollection.insertOne(sampleAccount)
   console.log(`Inserted document: ${result.insertedId}`)
 } catch (err) {
   console.error(`Error inserting document: ${err}`)
 } finally {
   await client.close()
 }
}

main()
```

Insert Many Documents

To insert more than one document, append the `insertMany()` method to the `collection` object, and then pass an array of documents to the `insertMany()` method. The `insertMany()` method returns a promise. We await the promise to get the result of the operation, which we then use to log the number of documents that are inserted to the console. In this example, the accounts to be inserted are stored in an array variable called `sampleAccounts`. This variable is defined just above the `main()` function.

```
const dbname = "bank"
const collection_name = "accounts"

const accountsCollection = client.db(dbname).collection(collection_name)

const sampleAccounts = [
 {
   account_id: "MDB011235813",
   account_holder: "Ada Lovelace",
   account_type: "checking",
   balance: 60218,
 },
 {
   account_id: "MDB829000001",
   account_holder: "Muhammad ibn Musa al-Khwarizmi",
   account_type: "savings",
   balance: 267914296,
 },
]

const main = async () => {
 try {
   await connectToDatabase()
   let result = await accountsCollection.insertMany(sampleAccounts)
   console.log(`Inserted ${result.insertedCount} documents`)
   console.log(result)
 } catch (err) {
   console.error(`Error inserting documents: ${err}`)
 } finally {
   await client.close()
 }
}

main()
```

Querying a MongoDB Collection in Node.js Applications

Using find()

The `find()` method is a read operation that returns a cursor to the documents that match the query. The `find()` method takes a query or filter document as an argument. If you do not specify a query document, the `find()` method returns all documents in the collection.

In this example, we find all accounts that have a balance greater than or equal to 4700. The `find()` method accepts a query filter, which we assign to a variable called `documentsToFind`. We process each document that's returned from the `find()` method by iterating the cursor, which is assigned to the variable `result`.

```
const dbname = "bank"
const collection_name = "accounts"

const accountsCollection = client.db(dbname).collection(collection_name)

// Document used as a filter for the find() method
const documentsToFind = { balance: { $gt: 4700 } }

const main = async () => {
 try {
   await connectToDatabase()
   // find() method is used here to find documents that match the filter
   let result = accountsCollection.find(documentsToFind)
   let docCount = accountsCollection.countDocuments(documentsToFind)
   await result.forEach((doc) => console.log(doc))
   console.log(`Found ${await docCount} documents`)
 } catch (err) {
   console.error(`Error finding documents: ${err}`)
 } finally {
   await client.close()
 }
}

main()
```

Using findOne()

In this example, we return a single document from a query, which is assigned to a variable called `documentToFind`. We use the `findOne()` method on the collection object to return the first document that matches the filter criteria, which are defined in the `documentToFind` variable.

```
const dbname = "bank"
const collection_name = "accounts"

const accountsCollection = client.db(dbname).collection(collection_name)

// Document used as a filter for the findOne() method
const documentToFind = { _id: ObjectId("62a3638521a9ad028fdf77a3") }

const main = async () => {
 try {
   await connectToDatabase()
   // findOne() method is used here to find a the first document that matches the filter
   let result = await accountsCollection.findOne(documentToFind)
   console.log(`Found one document`)
   console.log(result)
 } catch (err) {
   console.error(`Error finding document: ${err}`)
 } finally {
   await client.close()
 }
}

main()
```

Updating Documents in Node.js Applications

Using updateOne()

In this example, we use the `updateOne()` to update the value of an existing field in a document. Append the `updateOne()` method to the collection object to update a single document that matches the filter criteria, which are stored in the `documentToUpdate` variable. The update document contains the changes to be made and is stored in the `update` variable.

```
const dbname = "bank"
const collection_name = "accounts"

const accountsCollection = client.db(dbname).collection(collection_name)

const documentToUpdate = { _id: ObjectId("62d6e04ecab6d8e130497482") }

const update = { $inc: { balance: 100 } }


const main = async () => {
  try {
    await connectToDatabase()
    let result = await accountsCollection.updateOne(documentToUpdate, update)
    result.modifiedCount === 1
      ? console.log("Updated one document")
      : console.log("No documents updated")
  } catch (err) {
    console.error(`Error updating document: ${err}`)
  } finally {
    await client.close()
  }
}

main()
```

Using updateMany()

In this example, we update many documents by adding a value to the `transfers_complete` array of all checking account documents. The `updateMany()` method is appended to the collection object. The method accepts a filter that matches the document(s) that we want to update and an update statement that instructs the driver how to change the matching document. Both the filter and the update documents are stored in variables. The `updateMany()` method updates all the documents in the collection that match the filter.

```
const database = client.db(dbname);
const bank = database.collection(collection_name);

const documentsToUpdate = { account_type: "checking" };

const update = { $push: { transfers_complete: "TR413308000" } }

const main = async () => {
  try {
    await connectToDatabase()
    let result = await accountsCollection.updateMany(documentsToUpdate, update)
    result.modifiedCount > 0
      ? console.log(`Updated ${result.modifiedCount} documents`)
      : console.log("No documents updated")
  } catch (err) {
    console.error(`Error updating documents: ${err}`)
  } finally {
    await client.close()
  }
}

main()
```

Deleting Documents in Node.js Applications

Using deleteOne()

To delete a single document from a collection, use the `deleteOne()` method on a collection object. This method accepts a query filter that matches the document that you want to delete. If you do not specify a filter, MongoDB matches and deletes the first document in the collection. Here's an example:

```
const dbname = "bank"
const collection_name = "accounts"

const accountsCollection = client.db(dbname).collection(collection_name)

const documentToDelete = { _id: ObjectId("62d6e04ecab6d8e13049749c") }

const main = async () => {
  try {
    await connectToDatabase()
    let result = await accountsCollection.deleteOne(documentToDelete)
    result.deletedCount === 1
      ? console.log("Deleted one document")
      : console.log("No documents deleted")
  } catch (err) {
    console.error(`Error deleting documents: ${err}`)
  } finally {
    await client.close()
  }
}

main()
```

Using deleteMany()

You can delete multiple documents from a collection in a single operation by calling the `deleteMany()` method on a collection object. To specify which documents to delete, pass a query filter that matches the documents that you want to delete. If you provide an empty document, MongoDB matches all documents in the collection and deletes them. In the following example, we delete all accounts with a balance of less than 500 by using a query filter. Then, we print the total number of deleted documents.

```
const dbname = "bank"
const collection_name = "accounts"

const accountsCollection = client.db(dbname).collection(collection_name)

const documentsToDelete = { balance: { $lt: 500 } }

const main = async () => {
 try {
   await connectToDatabase()
   let result = await accountsCollection.deleteMany(documentsToDelete)
   result.deletedCount > 0
     ? console.log(`Deleted ${result.deletedCount} documents`)
     : console.log("No documents deleted")
 } catch (err) {
   console.error(`Error deleting documents: ${err}`)
 } finally {
   await client.close()
 }
}

main()
```

Creating MongoDB Transactions in Node.js Applications

Creating a Transaction

In this section, we'll go through the code to create a transaction step by step. We start the transaction by using the session's `withTransaction()` method. We then define the sequence of operations to perform inside the transactions, passing the `session` object to each operation in the transactions.

1. Create variables used in the transaction.

```
// Collections
const accounts = client.db("bank").collection("accounts")
const transfers = client.db("bank").collection("transfers")

// Account information
let account_id_sender = "MDB574189300"
let account_id_receiver = "MDB343652528"
let transaction_amount = 100
```

2. Start a new session.

```
const session = client.startSession()
```

3. Begin a transaction with the `WithTransaction()` method on the session.

```
const transactionResults = await session.withTransaction(async () => {
  // Operations will go here
})
```

4. Update the `balance` field of the sender's account by decrementing the `transaction_amount` from the `balance` field.

```
const senderUpdate = await accounts.updateOne(
  { account_id: account_id_sender },
  { $inc: { balance: -transaction_amount } },
  { session }
)
```

5. Update the `balance` field of the receiver's account by incrementing the `transaction_amount` to the `balance` field.

```
const receiverUpdate = await accounts.updateOne(
  { account_id: account_id_receiver },
  { $inc: { balance: transaction_amount } },
  { session }
)
```

6. Create a transfer document and insert it into the `transfers` collection.

```
const transfer = {
  transfer_id: "TR21872187",
  amount: 100,
  from_account: account_id_sender,
  to_account: account_id_receiver,
}

const insertTransferResults = await transfers.insertOne(transfer, { session })
```

7. Update the `transfers_complete` array of the sender's account by adding the `transfer_id` to the array.

```
const updateSenderTransferResults = await accounts.updateOne(
  { account_id: account_id_sender },
  { $push: { transfers_complete: transfer.transfer_id } },
  { session }
)
```

8. Update the `transfers_complete` array of the receiver's account by adding the `transfer_id` to the array.

```
const updateReceiverTransferResults = await accounts.updateOne(
  { account_id: account_id_receiver },
  { $push: { transfers_complete: transfer.transfer_id } },
  { session }
)
```

9. Log a message regarding the success or failure of the transaction.

```
if (transactionResults) {
  console.log("Transaction completed successfully.")
} else {
  console.log("Transaction failed.")
}
```

10. Catch any errors and close the session.

```
} catch (err) {
  console.error(`Transaction aborted: ${err}`)
  process.exit(1)
} finally {
  await session.endSession()
  await client.close()
}
```

Lesson 1: Introduction to MongoDB Aggregation / Learn

Introduction to MongoDB Aggregation

This section contains key definitions for this lesson, as well as the code for an aggregation pipeline.

Definitions

- **Aggregation:** Collection and summary of data
- **Stage**: One of the built-in methods that can be completed on the data, but does not permanently alter it
- **Aggregation pipeline:** A series of stages completed on the data in order

Structure of an Aggregation Pipeline

```
db.collection.aggregate([
    {
        $stage1: {
            { expression1 },
            { expression2 }...
        },
        $stage2: {
            { expression1 }...
        }
    }
])
```

Using `$match` and `$group` Stages in a MongoDB Aggregation Pipeline

`$match`

The `$match` stage filters for documents that match specified conditions. Here's the code for `$match`:

```
{
  $match: {
     "field_name": "value"
  }
}
```

`$group`

The `$group` stage groups documents by a group key.

```
{
  $group:
    {
      _id: <expression>, // Group key
      <field>: { <accumulator> : <expression> }
    }
 }
```

`$match` and `$group` in an Aggregation Pipeline

The following aggregation pipeline finds the documents with a field named "state" that matches a value "CA" and then groups those documents by the group key "$city" and shows the total number of zip codes in the state of California.

```
db.zips.aggregate([
{
   $match: {
      state: "CA"
    }
},
{
   $group: {
      _id: "$city",
      totalZips: { $count : { } }
   }
}
])
```

Using `$sort` and `$limit` Stages in a MongoDB Aggregation Pipeline

Review the following sections, which show the code for the `$sort` and `$limit` aggregation stages.

`$sort`

The `$sort` stage sorts all input documents and returns them to the pipeline in sorted order. We use 1 to represent ascending order, and -1 to represent descending order.

```
{
    $sort: {
        "field_name": 1
    }
}
```

`$limit`

The `$limit` stage returns only a specified number of records.

```
{
  $limit: 5
}
```

`$sort` and `$limit` in an Aggregation Pipeline

The following aggregation pipeline sorts the documents in descending order, so the documents with the greatest `pop` value appear first, and limits the output to only the first five documents after sorting.

```
db.zips.aggregate([
{
  $sort: {
    pop: -1
  }
},
{
  $limit:  5
}
])
```

Using `$project`, `$count`, and `$set` Stages in a MongoDB Aggregation Pipeline

Review the following sections, which show the code for the `$project`, `$set`, and `$count` aggregation stages.

`$project`

The `$project` stage specifies the fields of the output documents. 1 means that the field should be included, and 0 means that the field should be supressed. The field can also be assigned a new value.

```
{
    $project: {
        state:1,
        zip:1,
        population:"$pop",
        _id:0
    }
}
```

`$set`

The `$set` stage creates new fields or changes the value of existing fields, and then outputs the documents with the new fields.

```
{
    $set: {
        place: {
            $concat:["$city",",","$state"]
        },
        pop:10000
     }
  }
```

`$count`

The `$count` stage creates a new document, with the number of documents at that stage in the aggregation pipeline assigned to the specified field name.

```
{
  $count: "total_zips"
}
```

MongoDB Aggregation

In this unit, you learned how to use aggregation in MongoDB and create an aggregation pipeline. You also learned how to use several of the most common aggregation stages, including:

- `$match`
- `$group`
- `$sort`
- `$limit`
- `$project`
- `$count`
- `$set`
- `$out`

Resources

Use the following resources to learn more about inserting and finding documents in MongoDB:

Lesson 01: Introduction to MongoDB Aggregation

- MongoDB Docs: Aggregation Operations
- MongoDB Docs: Aggregation Pipelines

Lesson 02: Using `$match` and `$group` Stages in a MongoDB Aggregation Pipeline

- MongoDB Docs: `$match`
- MongoDB Docs: `$group`

Lesson 03: Using `$sort` and `$limit` Stages in a MongoDB Aggregation Pipeline

- MongoDB Docs: `$sort`
- MongoDB Docs: `$limit`

Lesson 04: Using `$project`, `$count`, and `$set` Stages in a MongoDB Aggregation Pipeline

- MongoDB Docs: `$project`
- MongoDB Docs: `$count`
- MongoDB Docs: `$set`

Lesson 05: Using `$out` Stage in a MongoDB Aggregation Pipeline

- MongoDB Docs: `$out`

Using MongoDB Aggregation Stages with Node.js: $match and $group

Review the following code, which demonstrates how to build the `$match` and `$group` stages of an aggregation pipeline in MongoDB with Node.js.

Using $match

Aggregation gives you a way to transform data from your collection by passing documents from one stage to another. These stages can consist of operators that transform or organize your data in a specific way. In this lesson, we used `$match` and `$group`.

The `$match` stage filters documents by using a simple equality match, like `$match: { author: "Dave"}`, or aggregation expressions using comparison operators, like `$match: { likes: { $gt: 100 } }`. This operator accepts a query document and passes the resulting documents to the next stage. `$match` should be placed early in your pipeline to reduce the number of documents to process.

Using $group

The `$group` stage separates documents according to a group key and returns one document for every unique group key. The group key is usually a field in the document, but it can also be an expression that resolves to a field. The `$group` stage can be used with aggregation expressions to perform calculations on the grouped documents. An example of this is adding up the total number of movie tickets sold by using the `$sum` operator:

`$group: { _id: "$movie", totalTickets: { $sum: "$tickets" } }`

The `"$movie"` is the group key, and the `totalTickets` field is the result of the `$sum` operator.

In the following code, we assign our collection name to a variable for convenience. First, we declare some variables to hold the database connection and the collection we'll use:

```
const client = new MongoClient(uri)
const dbname = "bank";
const collection_name = "accounts";
const accountsCollection = client.db(dbname).collection(collection_name);
```

Next, we build an aggregation pipeline that uses `$match` and `$group` and that will find accounts with a balance of less than $1,000. Then we group the results by the `account_type`, and calculate the `total_balance` and `avg_balance` for each type.

```
const pipeline = [
  // Stage 1: match the accounts with a balance greater than $1,000
  { $match: { balance: { $lt: 1000 } } },
  // Stage 2: Calculate average balance and total balance
  {
    $group: {
      _id: "$account_type",
      total_balance: { $sum: "$balance" },
      avg_balance: { $avg: "$balance" },
    },
  },
]
```

To run an aggregation pipeline, we append the aggregate method to the collection. The aggregate method takes an array of stages as an argument, which is stored in a variable. The aggregate method returns a cursor that we can iterate over to get the results.

```
const main = async () => {
  try {
    await client.connect()
    console.log(`Connected to the database 🌎. \nFull connection string: ${safeURI}`)
    let result = await accountsCollection.aggregate(pipeline)
    for await (const doc of result) {
      console.log(doc)
    }
  } catch (err) {
    console.error(`Error connecting to the database: ${err}`)
  } finally {
    await client.close()
  }
}

main()
```

Using MongoDB Aggregation Stages with Node.js: `$sort` and `$project`

Review the following code, which demonstrates how to build the `$sort` and `$project` stages of an aggregation pipeline in MongoDB with Node.js.

`$sort`

Aggregation is a powerful tool that gives us the ability to compute and transform our data. In this lesson, we focused on the `$sort` and `$project` stages.

The `$sort` stage takes all the input documents and sorts them in a specific order. The documents can be sorted in numerical, alphabetical, ascending, or descending order.

The `$sort` stage accepts a sort key that specifies the field to sort on. The sort key can be 1 for ascending order or -1 for descending order. For example:

- `{ $sort: { balance: 1 } }` sorts the documents in ascending order by the `balance` field.
- `{ $sort: { balance: -1 } }` sorts the documents in descending order by the `balance` field.

`$project`

The `$project` stage takes all the input documents and passes along only a subset of the fields in those documents by specifying the fields to include or exclude.

For example, if we want our resulting documents to include only the `account_id`, we write `{ $project: { _id: 0, account_id: 1 } }`. The `_id` field is excluded by setting it to 0, and the `account_id` field is included by setting it to 1.

The `$project` stage can also create new computed fields based on data from the original documents. An example of this is creating a projected field that contains someone's full name, where only the first and last names are stored in the original document.

In the following example, we build an aggregation pipeline that uses `$match`, `$sort`, and `$project`, and that will find checking accounts with a balance of greater than or equal to $1,500. Then, we sort the results by the balance in descending order and return only the `account_id`, `account_type`, `balance`, and a new computed field named `gbp_balance`, which stands for Great British Pounds (GBP) balance.

```
const pipeline = [
  // Stage 1: $match - filter the documents (checking, balance >= 1500)
  { $match: { account_type: "checking", balance: { $gte: 1500 } } },

  // Stage 2: $sort - sorts the documents in descending order (balance)
  { $sort: { balance: -1 } },

  // Stage 3: $project - project only the requested fields and one computed field
 (account_type, account_id, balance, gbp_balance)
  {
    $project: {
      _id: 0,
      account_id: 1,
      account_type: 1,
      balance: 1,
      // GBP stands for Great British Pound
      gbp_balance: { $divide: ["$balance", 1.3] },
    },
  },
]
```

To run an aggregation pipeline, we append the aggregate method to the collection. The aggregate method takes an array of stages as an argument, which is stored here as a variable. The aggregate method returns a cursor that we can iterate over to get the results.

```
const main = async () => {
  try {
    await client.connect()
    console.log(`Connected to the database 🌍\n ${uri}`)
    let accounts = client.db("bank").collection("accounts")
    let result = await accounts.aggregate(pipeline)
    for await (const doc of result) {
      console.log(doc)
    }
  } catch (err) {
    console.error(`Error connecting to the database: ${err}`)
  } finally {
    await client.close()
  }
}

main()
```

MongoDB Aggregation with Node.js

In this unit, you learned how to:

- Define an aggregation pipeline and its stages and operators.
- Build the `$match` and `$group` stages of a pipeline in Node.js.
- Build the `$sort` and `$project` stages of a pipeline in Node.js.

Resources

Use the following resources to learn more about performing basic CRUD with Node.js:

Lesson 01: Building a MongoDB Aggregation Pipeline in Node.js Applications

- Aggregation
- Aggregation Pipeline
- Aggregation Stages

Lesson 02: Using MongoDB Aggregation Stages with Node.js: $match and $group

- `$match`
- `$group`

Lesson 03: Using MongoDB Aggregation Stages with Node.js: $sort and $project

- `$sort`
- `$project`

MongoDB Indexes

In this unit, we learned what indexes are and how they improve performance. We reviewed and built different indexes:

- Single-field (one field)
- Compound (2 to 32 fields)

We worked with index properties like unique and understood that Multikey indexes are indexes that include one array field.

We used the following commands in the collection to create or delete indexes:

- createIndex()
- dropIndex()

Finally, we learned how to view the indexes being used in a collection with the `getIndexes()` command and how to check if the index is being used in a query by executing the `explain()` command.

Resources

Use the following resources to learn more about indexes in MongoDB:

Lesson 1 - Using MongoDB Indexes in Collections

- MongoDB Docs: Indexes
- MongoDB Docs: Indexes Reference

Lesson 2 - Creating a Single Field Index in MongoDB

- MongoDB Docs: createIndex()
- MongoDB Docs: Unique Indexes
- MongoDB Docs: Measure Index Use
- MongoDB Docs: getIndexes()

Lesson 3 - Creating a Multikey Index in MongoDB

- MongoDB Docs: Multikey Indexes

Lesson 4 - Working with Compound Indexes in MongoDB

- MongoDB Docs: Compound Indexes
- MongoDB Docs: Indexing Strategies

Lesson 5 - Deleting MongoDB Indexes

- MongoDB Docs: dropIndex()
- MongoDB Docs: dropIndexes()

Creating a Single Field Index

Review the code below, which demonstrates how to create a single field index in a collection.

Create a Single Field Index

Use `createIndex()` to create a new index in a collection. Within the parentheses of `createIndex()`, include an object that contains the field and sort order.

```
db.customers.createIndex({
  birthdate: 1
})
```

Create a Unique Single Field Index

Add `{unique:true}` as a second, optional, parameter in `createIndex()` to force uniqueness in the index field values. Once the unique index is created, any inserts or updates including duplicated values in the collection for the index field/s will fail.

```
db.customers.createIndex({
  email: 1
},
{
  unique:true
})
```

MongoDB only creates the unique index if there is no duplication in the field values for the index field/s.

View the Indexes used in a Collection

Use `getIndexes()` to see all the indexes created in a collection.

```
db.customers.getIndexes()
```

Check if an index is being used on a query

Use `explain()` in a collection when running a query to see the Execution plan. This plan provides the details of the execution stages (IXSCAN , COLLSCAN, FETCH, SORT, etc.).

- The `IXSCAN` stage indicates the query is using an index and what index is being selected.
- The `COLLSCAN` stage indicates a collection scan is perform, not using any indexes.
- The `FETCH` stage indicates documents are being read from the collection.
- The `SORT` stage indicates documents are being sorted in memory.

```
db.customers.explain().find({
  birthdate: {
    $gt:ISODate("1995-08-01")
    }
  })
```

```
db.customers.explain().find({
  birthdate: {
    $gt:ISODate("1995-08-01")
    }
  }).sort({
    email:1
    })
```

Lesson 3: Creating a Multikey Index in MongoDB / Learn

Understanding Multikey Indexes

Review the code below, which demonstrates how multikey indexes work. If a single field or compound index includes an array field, then the index is a multikey index.

Create a Single field Multikey Index

Use `createIndex()` to create a new index in a collection. Include an object as parameter that contains the array field and sort order. In this example `accounts` is an array field.

```
db.customers.createIndex({
  accounts: 1
})
```

View the Indexes used in a Collection

Use `getIndexes()` to see all the indexes created in a collection.

```
db.customers.getIndexes()
```

Check if an index is being used on a query

Use `explain()` in a collection when running a query to see the Execution plan. This plan provides the details of the execution stages (IXSCAN , COLLSCAN, FETCH, SORT, etc.).

- The `IXSCAN` stage indicates the query is using an index and what index is being selected.
- The `COLLSCAN` stage indicates a collection scan is perform, not using any indexes.
- The `FETCH` stage indicates documents are being read from the collection.
- The `SORT` stage indicates documents are being sorted in memory.

```
db.customers.explain().find({
  accounts: 627788
  })
```

Working with Compound Indexes

Create a Compound Index

Use `createIndex()` to create a new index in a collection. Within the parentheses of `createIndex()`, include an object that contains two or more fields and their sort order.

```
db.customers.createIndex({
  active:1,
  birthdate:-1,
  name:1
})
```

Order of Fields in a Compound Index

The order of the fields matters when creating the index and the sort order. It is recommended to list the fields in the following order: Equality, Sort, and Range.

- Equality: field/s that matches on a single field value in a query
- Sort: field/s that orders the results by in a query
- Range: field/s that the query filter in a range of valid values

The following query includes an equality match on the active field, a sort on birthday (descending) and name (ascending), and a range query on birthday too.

```
db.customers.find({
  birthdate: {
    $gte:ISODate("1977-01-01")
    },
    active:true
    }).sort({
      birthdate:-1,
      name:1
      })
```

Here's an example of an efficient index for this query:

```
db.customers.createIndex({
  active:1,
  birthdate:-1,
  name:1
})
```

View the Indexes used in a Collection

Use `getIndexes()` to see all the indexes created in a collection.

```
db.customers.getIndexes()
```

Check if an index is being used on a query

Use `explain()` in a collection when running a query to see the Execution plan. This plan provides the details of the execution stages (IXSCAN , COLLSCAN, FETCH, SORT, etc.). Some of these are:

- The `IXSCAN` stage indicates the query is using an index and what index is being selected.
- The `COLLSCAN` stage indicates a collection scan is perform, not using any indexes.
- The `FETCH` stage indicates documents are being read from the collection.
- The `SORT` stage indicates documents are being sorted in memory.

```
db.customers.explain().find({
  birthdate: {
    $gte:ISODate("1977-01-01")
    },
  active:true
  }).sort({
    birthdate:-1,
    name:1
    })
```

Cover a query by the Index

An Index covers a query when MongoDB does not need to fetch the data from memory since all the required data is already returned by the index.

In most cases, we can use projections to return only the required fields and cover the query. Make sure those fields in the projection are in the index.

By adding the projection `{name:1,birthdate:1,_id:0}` in the previous query, we can limit the returned fields to only `name` and `birthdate`. These fields are part of the index and when we run the `explain()` command, the execution plan shows only two stages:

- IXSCAN - Index scan using the compound index
- PROJECTION_COVERED - All the information needed is returned by the index, no need to fetch from memory

```
db.customers.explain().find({
  birthdate: {
    $gte:ISODate("1977-01-01")
    },
  active:true
  },
  {name:1,
    birthdate:1,
    _id:0
  }).sort({
    birthdate:-1,
    name:1
    })
```

Deleting an Index

Review the code below, which demonstrates how to delete indexes in a collection.

View the Indexes used in a Collection

Use `getIndexes()` to see all the indexes created in a collection. There is always a default index in every collection on _id field. This index is used by MongoDB internally and cannot be deleted.

```
db.customers.getIndexes()
```

Delete an Index

Use `dropIndex()` to delete an existing index from a collection. Within the parentheses of `dropIndex()`, include an object representing the index key or provide the index name as a string.

Delete index by name:

```
db.customers.dropIndex(
  'active_1_birthdate_-1_name_1'
)
```

Delete index by key:

```
db.customers.dropIndex({
  active:1,
  birthdate:-1,
  name:1
})
```

Delete Indexes

Use `dropIndexes()` to delete all the indexes from a collection, with the exception of the default index on _id.

```
db.customers.dropIndexes()
```

The `dropIndexes()` command also can accept an array of index names as a parameter to delete a specific list of indexes.

```
db.collection.dropIndexes([
  'index1name', 'index2name', 'index3name'
  ])
```

MongoDB Atlas Search

Congratulations on learning about Atlas Search, an incredibly powerful tool for adding search functionality to your apps. In this unit, we covered the basics of Atlas Search, including the following:

- What a search index is.
- How to use a dynamically mapped search index to find results from any field in your data.
- How to use a statically mapped search index to find results from the most relevant parts of your data.
- How to use the aggregation pipeline to complete a `$search` operation.
- How to use a compound operator to search based on multiple operators.
- How to use a compound operator to give more weight to certain fields and give the user more relevant results.
- How to use `$searchMeta` and `$facet` to categorize search results to help your app's users find what they need more quickly

Resources

You can use the following resources to learn more about adding Atlas Search to your apps:

Lesson 01: Using Relevance-Based Search and Search Indexes

- MongoDB Docs: Atlas Search Overview

Lesson 02: Creating a Search Index with Dynamic Field Mapping

- MongoDB Docs: Create an Atlas Search Index

Lesson 03: Creating a Search Index with Static Field Mapping

- MongoDB Docs: `$search`

Lesson 04: Using $search and Compound Operators

- MongoDB Docs: Compound Operators

Lesson 05: Grouping Search Results by Using Facets

- MongoDB Docs: Facet

Using $search and Compound Operators

The compound operator within the $search aggregation stage allows us to give weight to different field and also filter our results without having to create additional aggregation stages. The four options for the compound operator are "must", "mustNot, "should", and "filter".

"must" will exclude records that do not meet the criteria. "mustNot" will exclude results that do meet the criteria. "should" will allow you to give weight to results that do meet the criteria so that they appear first. "filter" will remove results that do not meet the criteria.

```
$search {
  "compound": {
    "must": [{
      "text": {
        "query": "field",
        "path": "habitat"
      }
    }],
    "should": [{
      "range": {
        "gte": 45,
        "path": "wingspan_cm",
        "score": {"constant": {"value": 5}}
      }
    }]
  }
}
```

Grouping Search Results by Using Facets

$searchMeta and $facet

$searchMeta is an aggregation stage for Atlas Search where the metadata related to the search is shown. This means that if our search results are broken into buckets, using $facet, we can see that in the $searchMeta stage, because those buckets are information about how the search results are formatted.

```
$searchMeta: {
    "facet": {
        "operator": {
            "text": {
            "query": ["Northern Cardinal"],
            "path": "common_name"
            }
        },
        "facets": {
            "sightingWeekFacet": {
                "type": "date",
                "path": "sighting",
                "boundaries": [ISODate("2022-01-01"),
                    ISODate("2022-01-08"),
                    ISODate("2022-01-15"),
                    ISODate("2022-01-22")],
                "default" : "other"
            }
        }
    }
}
```

"facet" is an operator within $searchMeta. "operator" refers to the search operator - the query itself. "facets" operator is where we put the definition of the buckets for the facets.

```
{
    "name": "sample_supplies-sales-facets",
    "searchAnalyzer": "lucene.standard",
    "analyzer": "lucene.standard",
    "collectionName": "sales",
    "database": "sample_supplies",
    "mappings": {
      "dynamic": true,
      "fields": {
        "purchaseMethod": [
          {
            "dynamic": true,
            "type": "document"
          },
          {
            "type": "string"
          }
        ],
        "storeLocation": [
          {
            "dynamic": true,
            "type": "document"
          },
          {
            "type": "stringFacet"
          }
        ]
      }
    }
}
```

MongoDB Transactions

In this unit, you learned that ACID transactions ensure that database operations, such as transferring funds from one account to another, happen together or not at all. You also explored how ACID transactions work with the document model in MongoDB. Finally, you learned how to create and use multi-document transactions by using the `startTransaction()` and `commitTransaction()` commands, and how to cancel multi-document transactions by using the `abortTransaction()` command.

Resources

Use the following resources to learn more about ACID transactions in MongoDB:

Lesson 01: Introduction to ACID Transactions

- What are ACID Properties in Database Management Systems?

Lesson 02: ACID Transactions in MongoDB

- What are ACID Properties in Database Management Systems?

MongoDB Acid Transections

- A transaction has a maximum runtime of less than one minute after the first write.
- **MongoServerError:**
  Transaction 1 has been aborted

Multi-Document Transactions

ACID transactions in MongoDB are typically used only by applications where values are exchanged between different parties, such as banking or business applications. If you find yourself in a scenario where a multi-document transaction is required, it's very likely that you will complete a transaction with one of MongoDB's drivers. For now, let's focus on completing and canceling multi-document transactions in the shell to become familiar with the steps.

Using a Transaction: Video Code

Here is a recap of the code that's used to complete a multi-document transaction:

```
const session = db.getMongo().startSession()
session.startTransaction()

const account = session.getDatabase('< add database name here>')
.getCollection('<add collection name here>')
//Add database operations like .updateOne() here
session.commitTransaction()
```

Aborting a Transaction

If you find yourself in a scenario that requires you to roll back database operations before a transaction is completed, you can abort the transaction. Doing so will roll back the database to its original state, before the transaction was initiated.

Aborting a Transaction: Video Code

Here is a recap of the code that's used to cancel a transaction before it completes:

```
const session = db.getMongo().startSession()

session.startTransaction()

const account = session.getDatabase('< add database name here>').getCollection(
'<add collection name here>')

//Add database operations like .updateOne() here

session.abortTransaction()
```

Example :

```
const session = db.getMongo().startSession()

session.startTransaction()

const account = session.getDatabase('<bank>').getCollection(
'accounts')

//Add database operations like .updateOne() here
"account.updateOne({account_id:"MDB74036066"},{$inc:{balance:-30}});"
"account.updateOne({account_id:"MDB74036067"},{$inc:{balance:30}});



session.abortTransaction()
```