How to Install MongoDB Using npm

Learn how to use MongoDB's Node.js driver with npm to connect with the database and perform CRUD operations.

MongoDB is a modern, general-purpose document-oriented data platform that has been widely paired with Node.js in popular tech stacks such as the MEAN stack (MongoDB, Express.js, AngularJS, and Node.js) and the MERN stack (MongoDB, Express.js, React.js, and Node.js).

npm, Node.js's package manager, is the bridge that allows the MongoDB Node.js driver to be installed, making it easier for developers to work with MongoDB from inside a Node.js application.

Prerequisites

This blog post walks you through the steps needed to connect to a MongoDB database with the Node.js driver which is distributed as an npm package. To follow along, you will need the following tools:

- Node.js and npm

  Node.js is a JavaScript runtime for building fast and scalable network applications, and it comes with npm installed. You can download an installer for your platform from the Node.js website.

- MongoDB Node.js driver

  To connect your application with a database, you need a software component, also known as a driver. MongoDB Node.js driver supports database connection, authentication, CRUD operations, and observations, among other features.

- MongoDB Atlas

  You can create a database either locally or in the cloud by using MongoDB Atlas, a fully managed cloud database service.

First, make sure you have the compatible version of Node.js. You can do that by running the following command in your terminal:

```
node --version
```

If you see an error, head over to the Node.js documentation and install the correct Node version.

The Node.js installer also installs npm, the Node package manager. To make sure you have npm installed, execute the following command in your terminal:

```
npm --version
```

If you're facing an issue with this or any other npm commands, refer to the npm documentation to help resolve them.

Project repository

You can explore the finished code for this project in its GitHub repository. Alternatively, you can follow along and build the project from scratch.

Initialize the Project

First, create a project directory called "`node-mongoDB`". Open your terminal and execute the following command:

```
mkdir node-mongoDB && cd node-mongoDB
```

Once inside the directory, create a "`package.json`" file by running the following command:

```
npm init -y
```

The "y" flag generates an npm package that uses the default values.

In this project, we'll use ECMAScript modules, which are the standardized way to package JavaScript code. To enable ECMAScript modules, add a new field to the "package.json" file:

```
"type": "module"
```

This will instruct Node.js to treat your JavaScript code as ECMAScript modules instead of the default CommonJS modules. Enabling ECMAScript modules will also allow us to use top-level await. In short, this means that we can use the await keyword outside of async functions.

Add MongoDB as a Dependency

Next, install the MongoDB driver along with its dependencies by running the following command:

```
npm install mongodb
```

The execution of the `npm install` command will take a few seconds to a minute depending on your network connection. The command downloads the `mongodb` package from the npm public registry into the `node_modules` directory of your project.

Create a MongoDB Atlas cluster

For this tutorial, you will need a MongoDB database instance. If you don't have one at hand, you can create a free cluster in MongoDB Atlas. Follow these steps to create and connect to a cluster:

- Create an Atlas account
- Deploy a Free Tier cluster
- Add your connection IP address to your IP access list
- Create a database user for your cluster
- Connect to your cluster

Be sure to save the connection string and database user's credentials for connecting to the database.

Connect to Your Cluster

Now that we have a cluster, we can connect to it. It's considered a good practice to keep your connection string and credentials separate from your code. You can do that by creating a configuration file that's not checked into revision history. Create a "`.env`" file at the root of your project and add your cluster connection string as a variable:

```
DB_URI=mongodb+srv://<username>:<password>@<clustername>.mongodb.net/
```

To load that configuration into our application, we'll use the `dotenv` package. Install it by executing the following npm command in your terminal:

```
npm install dotenv
```

Next, create a new directory "`src`" in your project and a new "`index.js`" file in it. To do that, you can execute the following command in your terminal:

```
mkdir src && touch src/index.js
```

Open the project in your favorite code editor and then, start editing the "`src/index.js`" file.

At the top of the file, import the `dotenv` package. Because we're using ECMAScript modules, we should use the `import/export` syntax instead of the CommonJS `require()` function.

```
import { config } from 'dotenv';
```

Then, we need to invoke the `config()` function to load the variables from the "`.env`" file into `process.env`. Once we've done that, we can log the connection URI to make sure the configuration is loaded correctly.

```
import { config } from 'dotenv';

config();
console.log(process.env.DB_URI);
```

Execute the script in your terminal:

```
node src/index.js
```

You should see the connection string. Now that we have the URI loaded, we can use the MongoDB Node.js driver to connect to our cluster and query data. Let's separate that logic in a new "`src/studentsCrud.js`" file. Create the file and open it in your editor.

Start by importing the `MongoClient` object from the `mongodb npm` package. `MongoClient` is an object used to initiate the connection with the database.

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

After that, create a new async function called `connectToCluster` with the following implementation:

```
export async function connectToCluster(uri) {
   let mongoClient;

   try {
       mongoClient = new MongoClient(uri);
       console.log('Connecting to MongoDB Atlas cluster...');
       await mongoClient.connect();
       console.log('Successfully connected to MongoDB Atlas!');

       return mongoClient;
   } catch (error) {
       console.error('Connection to MongoDB Atlas failed!', error);
       process.exit();
   }
}
```

The function accepts a parameter – `uri` – and instantiates a `MongoClient`. Then, it connects by invoking the asynchronous `connect()` method. We use the `await` keyword to wait for the connection to be established before executing the following statements. The whole implementation is wrapped in a `try/catch` statement. In case of an error, we're logging a message on the standard output and we're terminating the execution with `process.exit()`.

In the following sections, we'll implement simple CRUD (Create, Read, Update, Delete) operations on a collection of "`students`" in our database. Let's lay the ground by creating a new function called `executeStudentCrudOperations()` in the "`src/studentsCrud.js`" file. From that function, we'll connect to the database using the extracted URI and the `connectToCluster()` function. We'll also wrap the connection call in a `try/finally` statement. In the `finally` block, we'll ensure the connection is closed when the script finishes execution or if any error occurs.

```
export async function executeStudentCrudOperations() {
   const uri = process.env.DB_URI;
   let mongoClient;

   try {
       mongoClient = await connectToCluster(uri);
   } finally {
       await mongoClient.close();
   }
}
```

Let's use the function to connect to our cluster! Import it in the "`src/index.js`" file and invoke it. Make sure you use the `await` keyword. Without it, the script will terminate before the function finishes its execution. You can also remove the line that logs the connection string.

The final version of the "`src/index.js`" file should look like this:

```
import { config } from 'dotenv';
import { executeStudentCrudOperations } from './studentsCrud.js';

config();
await executeStudentCrudOperations();
```

Execute the script in your terminal:

```
node src/index.js
```

You should see the following messages logged:

```
Connecting to MongoDB Atlas cluster...
Successfully connected to MongoDB Atlas!
```

Great job! In the following sections, we'll implement simple CRUD (Create, Read, Update, Delete) operations on a new "`students`" collection in our database.

CRUD Operations

Now, you can select a database by using the `db()` method. If the database doesn't exist yet, it will be created.

Our example will create a database called "`school`", which will have a collection of "`students`".

Update the `executeStudentCrudOperations()` function from the previous section:

```
export async function executeStudentCrudOperations() {
   const uri = process.env.DB_URI;
   let mongoClient;

   try {
       mongoClient = await connectToCluster(uri);
       const db = mongoClient.db('school');
       const collection = db.collection('students');
   } finally {
       await mongoClient.close();
   }
}
```

Create

Now that you have a database and a collection set up, you can create a new document by using `insertOne()`. To create a *document*:

- Create an asynchronous function called `createStudentDocument`, and pass `collection` as a parameter.
- Inside the function, create an object called `studentDocument`, and store the student's information in it.
- Using `await`, pass the newly created object as a parameter to `insertOne()`.

Copy and paste the following function in your "`src/studentsCrud.js`" file, right after the connectToCluster function export.

```
export async function createStudentDocument(collection) {
   const studentDocument = {
       name: 'John Smith',
       birthdate: new Date(2000, 11, 20),
       address: { street: 'Pike Lane', city: 'Los Angeles', state: 'CA' },
   };

   await collection.insertOne(studentDocument);
}
```

Now that you've written the function, you have to call it. Inside the `try` block in `executeStudentCrudOperations()`, after you've defined the collection, call `createStudentDocument()`. Make sure to use await so that the script waits for execution of the function before exiting.

```
export async function executeStudentCrudOperations() {
   const uri = process.env.DB_URI;
   let mongoClient;

   try {
       mongoClient = await connectToCluster(uri);
       const db = mongoClient.db('school');
       const collection = db.collection('students');

       console.log('CREATE Student');
       await createStudentDocument(collection);
   } finally {
       await mongoClient.close();
   }
}
```

Execute the "`src/index.js`" script in your terminal:

```
node src/index.js
```

You should see the message "CREATE Student" logged on your terminal. To verify that the document was created, navigate to your MongoDB Atlas instance, and select *Browse Collections*. Once selected, you should see your newly created database, collection, and document.

Even though you haven't provided the `_id` field, notice there's an *id field present for your document. This is because if you don't provide the* id field, the database will automatically create a unique id for the document.

Find

You can use the find() method to retrieve documents from the database. This requires you to pass a query object with optional properties to filter the database. Similar to the `createStudentDocument()` function, you will create a `findStudentsByName()` function.

Copy the following function and add it after the `createStudentDocument()` function.

```
export async function findStudentsByName(collection, name) {
   return collection.find({ name }).toArray();
}
```

Call this function inside the `try` block and persist the retrieved document to a new variable. Use `console.log()` to print the variable. Make sure to comment out the call to the

`createStudentDocument()` function. Otherwise, the script will create a new document with the same information and a new id.

```
export async function executeStudentCrudOperations() {
    const uri = process.env.DB_URI;
    let mongoClient;

    try {
        mongoClient = await connectToCluster(uri);
        const db = mongoClient.db('school');
        const collection = db.collection('students');

        // console.log('CREATE Student');
        // await createStudentDocument(collection);
        console.log(await findStudentsByName(collection, 'John Smith'));
    } finally {
        await mongoClient.close();
    }
}
```

Execute the script in your terminal:

```
node src/index.js
```

You should see the `studentDocument` object logged on your terminal.

To play around, try updating the name to *John* and notice what happens.

Update

Use the `updateOne()` method to update documents. This method has two required parameters:

- The query selector object that indicates which documents need to be updated
- An object indicating what changes need to be applied

Let's create a function that updates the specified fields of a document with a specified name:

```
export async function updateStudentsByName(collection, name, updatedFields) {
    await collection.updateMany(
        { name },
        { $set: updatedFields }
    );
}
```

Call this function and then `findStudentsByName()` once again to verify that the document was updated.

```
export async function executeStudentCrudOperations() {
    const uri = process.env.DB_URI;
    let mongoClient;

    try {
        mongoClient = await connectToCluster(uri);
        const db = mongoClient.db('school');
        const collection = db.collection('students');

        // console.log('CREATE Student');
        // await createStudentDocument(collection);
        console.log(await findStudentsByName(collection, 'John Smith'));

        console.log('UPDATE Student\'s Birthdate');
        await updateStudentsByName(collection, 'John Smith', { birthdate: new Date(2001, 5, 5)
        console.log(await findStudentsByName(collection, 'John Smith'));
    } finally {
        await mongoClient.close();
    }
}
```

Execute the script once again:

```
node src/index.js
```

You can compare the logged documents – before and after the update. If you refresh your MongoDB Atlas page, you should also see the updated document.

Delete

To remove documents from the database, use the deleteMany() method. This method takes in an object specifying which documents to delete as its first parameter.

Let's create a function that deletes all documents with a specified name right after the `updateStudentsByName()` function:

```
export async function deleteStudentsByName(collection, name) {
   await collection.deleteMany({ name });
}
```

Call the function inside the `try` block. You can uncomment the previous function calls, too.

```
export async function executeStudentCrudOperations() {
   const uri = process.env.DB_URI;
   let mongoClient;

   try {
       mongoClient = await connectToCluster(uri);
       const db = mongoClient.db('school');
       const collection = db.collection('students');

       console.log('CREATE Student');
       await createStudentDocument(collection);
       console.log(await findStudentsByName(collection, 'John Smith'));

       console.log('UPDATE Student\'s Birthdate');
       await updateStudentsByName(collection, 'John Smith', { birthdate: new Date(2001, 5, 5)
       console.log(await findStudentsByName(collection, 'John Smith'));

       console.log('DELETE Student');
       await deleteStudentsByName(collection, 'John Smith');
       console.log(await findStudentsByName(collection, 'John Smith'));
   } finally {
       await mongoClient.close();
   }
}
```

Execute the script one final time:

```
node src/index.js
```

You should see that no documents with the specified name are found after the `deleteStudentsByName()` function is executed. Refresh the MongoDB Atlas page. You'll see the documents have been removed.

Conclusion

This tutorial scratches the surface of the MongoDB Node.js driver and Atlas. The MongoDB Node.js driver offers various methods, including but not limited to methods for aggregating, inserting, and replacing documents. You can also use Atlas to distribute and secure your data, build for optimization, and perform other functions.

FAQs

How do you install MongoDB using npm?

How can you initialize the MongoDB app npm?

How do you install MongoDB in Windows using npm?

How do you install MongoDB on Mac using npm?

Can you use npm MongoDB without downloading from the MongoDB website?

How do you run MongoDB with Node.js?

What is Node.js?

How do I start MongoDB in Node.js?

What does npm stand for?

What is npm update?

What is the difference between npm and Node?

Are npm packages free?

Can npm update itself?

Is npm the same as npm install?