UNIT III Advanced Nodejs and Database

NoSQL database

NoSQL databases are databases designed to be used across large distrusted systems. They are notably much more scalable and much faster at handling very large data loads than traditional relational databases. Unlike other databases, NoSQL databases do not use the standard tabular relationships the relational databases employ. Instead, NoSQL databases allow for the querying and storage of data by a variety of other means, depending on the specific software.

How Does a NoSQL (nonrelational) Database Work?

NoSQL databases use a variety of data models for accessing and managing data. These types of databases are optimized specifically for applications that require large data volume, low latency, and flexible data models, which are achieved by relaxing some of the data consistency restrictions of other databases.

Consider the example of modeling the schema for a simple book database:

In a relational database, a book record is often dissembled (or “normalized”) and stored in separate tables, and relationships are defined by primary and foreign key constraints. In this example, the Books table has columns for ISBN, Book Title, and Edition Number, the Authors table has columns for AuthorID and Author Name, and finally the Author-ISBN table has columns for AuthorID and ISBN. The relational model is designed to enable the database to enforce referential integrity between tables in the database, normalized to reduce the redundancy, and generally optimized for storage.

In a NoSQL database, a book record is usually stored as a [JSON](http://json.org/) document. For each book, the item, ISBN, Book Title, Edition Number, Author Name, and AuthorID are stored as attributes in a single document. In this model, data is optimized for intuitive development and horizontal scalability.

Why should you use a NoSQL database?

NoSQL databases are a great fit for many modern applications such as mobile, web, and gaming that require flexible, scalable, high-performance, and highly functional databases to provide great user experiences.

Flexibility: NoSQL databases generally provide flexible schemas that enable faster and more iterative development. The flexible data model makes NoSQL databases ideal for semi-structured and unstructured data.

Scalability: NoSQL databases are generally designed to scale out by using distributed clusters of hardware instead of scaling up by adding expensive and robust servers. Some cloud providers handle these operations behind-the-scenes as a fully managed service.

High-performance: NoSQL database are optimized for specific data models and access patterns that enable higher performance than trying to accomplish similar functionality with relational databases.

Highly functional: NoSQL databases provide highly functional APIs and data types that are purpose built for each of their respective data models.

Advantages of NoSQL:  
There are many advantages of working with NoSQL databases such as MongoDB and Cassandra. The main advantages are high scalability and high availability.

1. High scalability –  
   NoSQL database use sharding for horizontal scaling. Partitioning of data and placing it on multiple machines in such a way that the order of the data is preserved is sharding. Vertical scaling means adding more resources to the existing machine whereas horizontal scaling means adding more machines to handle the data. Vertical scaling is not that easy to implement but horizontal scaling is easy to implement. Examples of horizontal scaling databases are MongoDB, Cassandra etc. NoSQL can handle huge amount of data because of scalability, as the data grows NoSQL scale itself to handle that data in efficient manner.
2. High availability –  
   Auto replication feature in NoSQL databases makes it highly available because in case of any failure data replicates itself to the previous consistent state.

Disadvantages of NoSQL:  
NoSQL has the following disadvantages.

1. Narrow focus –  
   NoSQL databases have very narrow focus as it is mainly designed for storage but it provides very little functionality. Relational databases are a better choice in the field of Transaction Management than NoSQL.
2. Open-source –  
   NoSQL is open-source database. There is no reliable standard for NoSQL yet. In other words two database systems are likely to be unequal.
3. Management challenge –  
   The purpose of big data tools is to make management of a large amount of data as simple as possible. But it is not so easy. Data management in NoSQL is much more complex than a relational database. NoSQL, in particular, has a reputation for being challenging to install and even more hectic to manage on a daily basis.
4. GUI is not available –  
   GUI mode tools to access the database is not flexibly available in the market.
5. Backup –  
   Backup is a great weak point for some NoSQL databases like MongoDB. MongoDB has no approach for the backup of data in a consistent manner.
6. Large document size –  
   Some database systems like MongoDB and CouchDB store data in JSON format. Which means that documents are quite large (BigData, network bandwidth, speed), and having descriptive key names actually hurts, since they increase the document size.

Types of NoSQL database:  
Types of NoSQL databases and the name of the databases system that falls in that category are:

1. MongoDB falls in the category of NoSQL document based database.
2. Key value store: Memcached, Redis, Coherence
3. Tabular: Hbase, Big Table, Accumulo
4. Document based: MongoDB, CouchDB, Cloudant

When should NoSQL be used:

1. When huge amount of data need to be stored and retrieved .
2. The relationship between the data you store is not that important
3. The data changing over time and is not structured.
4. Support of Constraints and Joins is not required at database level
5. The data is growing continuously and you need to scale the database regular to handle the data.

MongoDB

Like any other database management language, MongoDB is based on a NoSQL database that is used for storing data in a key-value pair. Its working is based on the concept of document and collection. It is also an open-source, a document-oriented, cross-platform database system that is written using C++. In this chapter, you will learn more about MongoDB and its importance.

Mongo DB can be defined as a document-oriented database system that uses the concept of NoSQL. It also provides high availability, high performance, along with automatic scaling. This open-source product was developed by the company - 10gen in October 2007, and the company also maintains it. MongoDB exists under the General Public License (GPL) as a free database management tool as well as available under Commercial license as of the manufacturer. MongoDB was also intended to function with commodity servers. Companies of different sizes all over the world across all industries are using MongoDB as their database.

Here are some key terminologies that you must know to get into the in-depth of MongoDB:

### What is a Database?

In MongoDB, a database can be defined as a physical container for collections of data. Here, on the file system, every database has its collection of files residing. Usually, a MongoDB server contains numerous databases.

### What are Collections?

Collections can be defined as a cluster of MongoDB documents that exist within a single database. You can relate this to that of a table in a relational database management system. MongoDB collections do not implement the concept of schema. Documents that have collection usually contain different fields. Typically, all the documents residing within a collection are meant for a comparable or related purpose.

### What is a Document?

A document can be defined as a collection of key-value pairs that contain dynamic schema. Dynamic schema is something that documents of the equal collection do not require for having the same collection of fields or construction, and a common field is capable of holding various types of data.

Here is a table showing the relation between the terminologies used in RDBMS and MongoDB:

|  |  |
| --- | --- |
| RDBMS | MongoDB |
| Database | Database |
| Table | Collection |
| Tuple or Row | Document |
| Column | Field |
| Table Join | Embedded Documents |
| Primary Key | Primary key / Default key |
| Mysqld / Oracle | mongod |

Popular organizations that use MongoDB

Here is a list of some popular and multinational companies and organizations that are using MongoDB as their official database to perform and manage different business applications.

* Adobe
* McAfee
* LinkedIn
* FourSquare
* MetLife
* eBay
* SAP

MongoDB used

Beginners need to know the purpose and requirement of why to use MongoDB or what is the need of it in contrast to SQL and other database systems. In simple words, it can be said that every modern-day application involves the concept of big data, analyzing different forms of data, fast features improvement in handling data, deployment flexibility, which old database systems are not competent enough to handle. Hence, MongoDB is the next choice.

Use MongoDB

Some basic requirements are supported by this NoSQL database, which is lacking in other database systems. These collective reasons make MongoDB popular among other database systems:

* Document-Oriented data storage, i.e., data, is stored in a JSON style format, which increases the readability of data as well.
* Replication and high availability of data.
* MongoDB provides Auto-sharding.
* Ad hoc queries are supported by MongoDB, which helps in searching by range queries, field, or using regex terms.
* Indexing of values can be used to create and improve the overall search performance in MongoDB. MongoDB allows any field to be indexed within a document.
* MongoDB has a rich collection of queries.
* Updating of data can be done at a faster pace.
* It can be integrated with other popular programming languages also to handle structured as well as unstructured data within various types of applications.

Advantage of MongoDB

* It is easy to set up, i.e., install the MongoDB.
* Since MongoDB is a schema-less database, so there is no hassle of schema migration.
* Since it is a document-oriented language, document queries are used, which plays a vital role in supporting dynamic queries.
* Easily scalable.
* It is easy to have a performance tuning as compared to other relational databases.
* It helps in providing fast accessing of data because of its nature of implementing the internal memory to store the data.
* MongoDB is also used as a file system that can help in easy management of load balancing.
* MongoDB also supports the searching using the concept of regex (regular expression) as well as fields.
* Users can run MongoDB as a windows service also.
* It does not require any VM to run on different platforms.
* It also supports sharding of data.

Basic Queries In Mongodb Shell

What is the MongoDB Mongo shell

MongoDB Mongo shell is an interactive JavaScript interface that allows you to interact with MongoDB instances through the command line. The shell can be used for:

* Data manipulation
* Administrative operations such as maintenance of database instances

## Installing the mongo shell

The mongo shell gets installed when you install the MongoDB server. It is installed in the same location as the MongoDB server binary.

If you want to install it separately, you can visit the [MongoDB download center](https://www.mongodb.com/try/download/shell), from there select the version and package you need, download the archive, and copy it to a location in your file system.

Mongo shell is available for all main operating systems, including:

* Windows
* Linux
* Mac OS

## Connect to MongoDB database

Once you’ve [downloaded and installed MongoDB](https://www.bmc.com/blogs/how-to-install-mongodb-ubuntu-mac/), you can use the mongo shell to connect with a MongoDB server that is up and running.

net start MongoDB

Then type mongo command to run the shell.

Mongo

Now you are in the Mongo shell.

db



Run the **use** command to switch to a different database. If you don’t have a database, [learn how to create a new database](https://www.bmc.com/blogs/mongodb-create-database/).

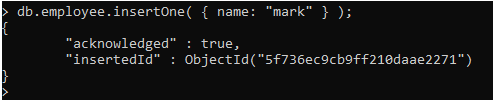
use company



You can create collections and insert data with the following command:

* db refers to the current database in use.
* employee is the collection name.
* insertOne is the method to insert a document to the collection.

db.employee.insertOne( { name: "mark" } );



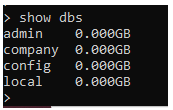
Use the **find** method to fetch data in a collection. The **forEach(printjson)** method will print them with JSON formatting

db.employee.find().forEach(printjson)



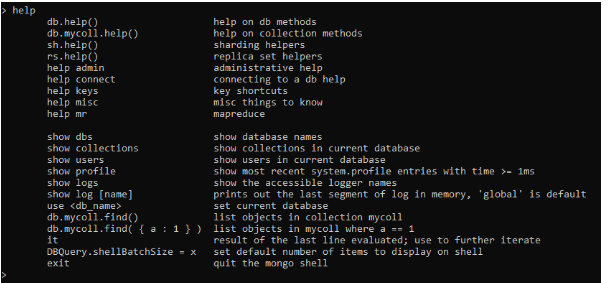
Use the show dbs command to Show all databases

show dbs



One important command will help you work with the Mongo shell easily: the **help** command. Run the help command to get a list of help options available in the mongo shell.

Help



## Disadvantages of the mongo shell

Although the Mongo shell is an excellent tool for learning and testing the MongoDB server, it is difficult to be used in a production environment. Being a shell inherently carries certain disadvantages. Let’s see what they are:

* The Mongo shell is strictly a console centric method of data manipulation. While some find it easy and quick, others might not find those characteristics appealing.
* If you are working on multiple sessions, you need multiple terminals.
* If the results are too long, they scroll away.
* Repetitive commands or debugging a function need the programmer to traverse the long command line history manually.

Alternative to mongodb mongo shell

So now you know the mongo shell has some disadvantages. At this point, you may want to know what other options are available. MongoDB developers have introduced drivers specific to each programming language to connect with the MongoDB databases when using MongoDB in your applications. You can find them [here](https://docs.mongodb.com/drivers/).

Additionally, many people prefer to use GUIs to work with databases nowadays. One of the best GUI tools for MongoDB is the [MongoDB Compass](https://www.mongodb.com/products/compass). Some other useful GUI tools are:

* [NoSQLBooster](https://nosqlbooster.com/)
* [Mongo Management Studio](http://mms.litixsoft.de/)
* [Robo 3T](https://robomongo.org/)

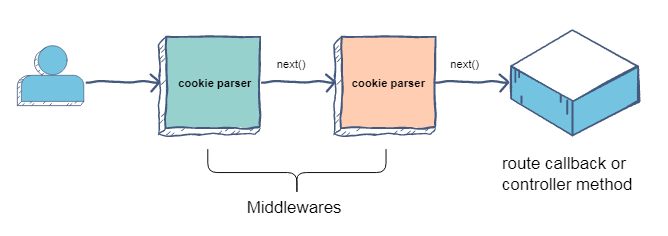
Remember that the best MongoDB GUI depends on the task that needs to be accomplished. MongoDB Compass is the go-to option if you need to avoid the command line completely. Robo 3T is simple and well supported by the community, while NoSQLBooster is shell centric smart GUI tool.

MongoDB Shell is the quickest way to connect, configure, query, and work with your MongoDB database. It acts as a command-line client of the MongoDB server.

You can start MongoDB Shell by executing mongo or mongosh command on the command prompt/terminal. mongosh is the new MongoDB shell with some more features than the old mongo shell.

Request Body Parsing In Express

Express body-parser is an npm library used to process data sent through an HTTP request body. It exposes four express middlewares for parsing text, JSON, url-encoded and raw data set through an HTTP request body. These middlewares are functions that process incoming requests before they reach the target controller.



## body-parser doesn’t have to be installed as a separate package because it is a dependency of express version 4.16.0+. body-parser isn’t a dependency between version 4.0.0 and 4.16.0, so it will be installed separately in projects locked to those versions. body-parser middlewares will be required by express in versions of express with body-parser dependency. Versions of Express wit

### bodyParser.json([options])

Returns middleware that only parses json and only looks at requests where the Content-Type header matches the type option. This parser accepts any Unicode encoding of the body and supports automatic inflation of gzip and deflate encodings.

A new body object containing the parsed data is populated on the request object after the middleware (i.e. req.body).

#### Options

The json function takes an optional options object that may contain any of the following keys:

##### inflate

When set to true, then deflated (compressed) bodies will be inflated; when false, deflated bodies are rejected. Defaults to true.

##### limit

Controls the maximum request body size. If this is a number, then the value specifies the number of bytes; if it is a string, the value is passed to the [bytes](https://www.npmjs.com/package/bytes) library for parsing. Defaults to '100kb'.

##### reviver

The reviver option is passed directly to JSON.parse as the second argument. You can find more information on this argument [in the MDN documentation about JSON.parse](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse#Example.3A_Using_the_reviver_parameter).

##### strict

When set to true, will only accept arrays and objects; when false will accept anything JSON.parse accepts. Defaults to true.

##### type

The type option is used to determine what media type the middleware will parse. This option can be a string, array of strings, or a function. If not a function, type option is passed directly to the [type-is](https://www.npmjs.org/package/type-is#readme) library and this can be an extension name (like json), a mime type (like application/json), or a mime type with a wildcard (like \*/\* or \*/json). If a function, the type option is called as fn(req) and the request is parsed if it returns a truthy value. Defaults to application/json.

##### verify

The verify option, if supplied, is called as verify(req, res, buf, encoding), where buf is a Buffer of the raw request body and encoding is the encoding of the request. The parsing can be aborted by throwing an error.

## How to use body-parser

The middlewares that body-parser exposes can be a part of a single route or for the whole app. You will need to expose and register the required middleware.

In the code block below, the JSON parser middleware is registered for the whole application. This means all requests will pass through this middleware.

const express = require("express");

const app = express();

require("dotenv").config();

app.get("/", (req, res) => {

res.send("<h1>Hi there, welcome to your app</h1>");

});

const jsonParser = express.json()

app.use(jsonParser);

app.post("/comment", (req, res) => {

res.send(req.body);

});

app.listen(process.env.PORT, () =>

console.log(`server running on port ${process.env.PORT}`)

## );

## NodeJs MongoDB Connection

* Node.js can be used in database applications.
* One of the most popular NoSQL database is MongoDB.

Install MongoDB Driver

Let us try to access a MongoDB database with Node.js.

To download and install the official MongoDB driver, open the Command Terminal and execute the following:

Download and install mongodb package:

C:\Users\*Your Name*>npm install mongodb

Now you have downloaded and installed a mongodb database driver.

Node.js can use this module to manipulate MongoDB databases:

var mongo = require('mongodb');

Creating a Database

To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database you want to create.

MongoDB will create the database if it does not exist, and make a connection to it.

## var MongoClient = require('mongodb').MongoClient; var url = "mongodb://localhost:27017/mydb"; MongoClient.connect(url, function(err, db) {   if (err) throw err;   console.log("Database created!");   db.close(); });

Save the code above in a file called "demo\_create\_mongo\_db.js" and run the file:

Run "demo\_create\_mongo\_db.js"

C:\Users\*Your Name*>node demo\_create\_mongo\_db.js

Which will give you this result:

Database created!

Creating a Collection

To create a collection in MongoDB, use the createCollection() method:

Create a collection called "customers":

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  dbo.createCollection("customers", function(err, res) {  
    if (err) throw err;  
    console.log("Collection created!");  
    db.close();  
  });  
});

Save the code above in a file called "demo\_mongodb\_createcollection.js" and run the file:

Run "demo\_mongodb\_createcollection.js"

C:\Users\*Your Name*>node demo\_mongodb\_createcollection.js

Which will give you this result:

Collection created!

Insert Into Collection

To insert a record, or *document* as it is called in MongoDB, into a collection, we use the insertOne() method.

The first parameter of the insertOne() method is an object containing the name(s) and value(s) of each field in the document you want to insert.

It also takes a callback function where you can work with any errors, or the result of the insertion:

## var MongoClient = require('mongodb').MongoClient; var url = "mongodb://localhost:27017/"; MongoClient.connect(url, function(err, db) {   if (err) throw err;   var dbo = db.db("mydb");   var myobj = { name: "Company Inc", address: "Highway 37" };   dbo.collection("customers").insertOne(myobj, function(err, res) {     if (err) throw err;     console.log("1 document inserted");     db.close();   }); });

Save the code above in a file called "demo\_mongodb\_insert.js" and run the file:

Run "demo\_mongodb\_insert.js"

C:\Users\*Your Name*>node demo\_mongodb\_insert.js

Which will give you this result:

1 document inserted

## Insert Multiple Documents

To insert multiple documents into a collection in MongoDB, we use the insertMany() method.

The first parameter of the insertMany() method is an array of objects, containing the data you want to insert.

It also takes a callback function where you can work with any errors, or the result of the insertion:

### Example

Insert multiple documents in the "customers" collection:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  var myobj = [  
    { name: 'John', address: 'Highway 71'},  
    { name: 'Peter', address: 'Lowstreet 4'},  
    { name: 'Amy', address: 'Apple st 652'},  
    { name: 'Hannah', address: 'Mountain 21'},  
    { name: 'Michael', address: 'Valley 345'},  
    { name: 'Sandy', address: 'Ocean blvd 2'},  
    { name: 'Betty', address: 'Green Grass 1'},  
    { name: 'Richard', address: 'Sky st 331'},  
    { name: 'Susan', address: 'One way 98'},  
    { name: 'Vicky', address: 'Yellow Garden 2'},  
    { name: 'Ben', address: 'Park Lane 38'},  
    { name: 'William', address: 'Central st 954'},  
    { name: 'Chuck', address: 'Main Road 989'},  
    { name: 'Viola', address: 'Sideway 1633'}  
  ];  
  dbo.collection("customers").insertMany(myobj, function(err, res) {  
    if (err) throw err;  
    console.log("Number of documents inserted: " + res.insertedCount);  
    db.close();  
  });  
});

Save the code above in a file called "demo\_mongodb\_insert\_multiple.js" and run the file:

Run "demo\_mongodb\_insert\_multiple.js"

C:\Users\*Your Name*>node demo\_mongodb\_insert\_multiple.js

Which will give you this result:

Number of documents inserted: 14

## Retrieve data

To select data from a collection in MongoDB, we can use the findOne() method.

The findOne() method returns the first occurrence in the selection.

The first parameter of the findOne() method is a query object. In this example we use an empty query object, which selects all documents in a collection (but returns only the first document).

### Example

Find the first document in the customers collection:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  dbo.collection("customers").findOne({}, function(err, result) {  
    if (err) throw err;  
    console.log(result.name);  
    db.close();  
  });  
});

To select data from a table in MongoDB, we can also use the find() method.

The find() method returns all occurrences in the selection.

The first parameter of the find() method is a query object. In this example we use an empty query object, which selects all documents in the collection.

No parameters in the find() method gives you the same result as **SELECT \*** in MySQL.

### Example

Find all documents in the customers collection:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  dbo.collection("customers").find({}).toArray(function(err, result) {  
    if (err) throw err;  
    console.log(result);  
    db.close();  
  });  
});

## Handling SQL Data From NodeJS

## 

## To be able to experiment with the code examples, you should have MySQL installed on your computer.

Install MySQL Driver

Once you have MySQL up and running on your computer, you can access it by using Node.js.

To access a MySQL database with Node.js, you need a MySQL driver. This tutorial will use the "mysql" module, downloaded from NPM.

To download and install the "mysql" module, open the Command Terminal and execute the following:

C:\Users\*Your Name*>npm install mysql

Now you have downloaded and installed a mysql database driver.

Node.js can use this module to manipulate the MySQL database:

var mysql = require('mysql');

Create Connection

Start by creating a connection to the database.

Use the username and password from your MySQL database.

demo\_db\_connection.js

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "*yourusername*",  
  password: "*yourpassword*"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
});

## Handling Cookies In NodeJS

Cookies are simple, small files/data that are sent to client with a server request and stored on the client side. Every time the user loads the website back, this cookie is sent with the request. This helps us keep track of the user’s actions.

The following are the numerous uses of the HTTP Cookies −

* Session management
* Personalization(Recommendation systems)
* User tracking

To use cookies with Express, we need the cookie-parser middleware. To install it, use the following code −

npm install --save cookie-parser

Now to use cookies with Express, we will require the **cookie-parser**. cookie-parser is a middleware which *parses cookies attached to the client request object*. To use it, we will require it in our **index.js** file; this can be used the same way as we use other middleware. Here, we will use the following code.

var cookieParser = require('cookie-parser');

app.use(cookieParser());

cookie-parser parses Cookie header and populates **req.cookies** with an object keyed by the cookie names. To set a new cookie, let us define a new route in your Express app like −

var express = require('express');

var app = express();

app.get('/', function(req, res){

res.cookie('name', 'express').send('cookie set'); //Sets name = express

});

app.listen(3000);

To check if your cookie is set or not, just go to your browser, fire up the console, and enter −

console.log(document.cookie);

You will get the output like (you may have more cookies set maybe due to extensions in your browser) −

"name = express"

The browser also sends back cookies every time it queries the server. To view cookies from your server, on the server console in a route, add the following code to that route.

console.log('Cookies: ', req.cookies);

Next time you send a request to this route, you will receive the following output.

Cookies: { name: 'express' }

Adding Cookies with Expiration Time

You can add cookies that expire. To add a cookie that expires, just pass an object with property 'expire' set to the time when you want it to expire. For example,

//Expires after 360000 ms from the time it is set.

res.cookie(name, 'value', {expire: 360000 + Date.now()});

Another way to set expiration time is using **'maxAge'** property. Using this property, we can provide relative time instead of absolute time. Following is an example of this method.

//This cookie also expires after 360000 ms from the time it is set.

res.cookie(name, 'value', {maxAge: 360000});

Deleting Existing Cookies

To delete a cookie, use the clearCookie function. For example, if you need to clear a cookie named **foo**, use the following code.

var express = require('express');

var app = express();

app.get('/clear\_cookie\_foo', function(req, res){

res.clearCookie('foo');

res.send('cookie foo cleared');

});

app.listen(3000);

res.cookie(name\_of\_cookie, value\_of\_cookie);

This can be explained by the following example :

|  |
| --- |
| let express = require('express');  let cookieParser = require('cookie-parser');  //setup express app  let app = express()    app.use(cookieParser());      //basic route for homepage  app.get('/', (req, res)=>{  res.send('welcome to express app');  });    //JSON object to be added to cookie  let users = {  name : "Ritik",  Age : "18"  }    //Route for adding cookie  app.get('/setuser', (req, res)=>{  res.cookie("userData", users);  res.send('user data added to cookie');  });    //Iterate users data from cookie  app.get('/getuser', (req, res)=>{  //shows all the cookies  res.send(req.cookies);  });    //server listens to port 3000  app.listen(3000, (err)=>{  if(err)  throw err;  console.log('listening on port 3000');  }); |

## Handling User Authentication With NodeJS

Authentication and authorization may seem like the same thing. But there's a big difference between getting into a house (authentication) and what you can do once you're there (authorization).

Authentication is the process of confirming a user's identity by obtaining credentials and using those credentials to validate their identity. If the certificates are valid, the authorization procedure begins.

You are probably already familiar with the authentication process, because we all go through it daily – whether at work (logging onto your computer) or at home (passwords or logging into a website). In fact, most "things" connected to the Internet require you to provide credentials to prove your identity.

Authorization is the process of granting authenticated users access to resources by verifying whether they have system access permissions or not. It also allows you to restrict access privileges by granting or denying specific licenses to authenticated users.

After the system authenticates your identity, authorization occurs, providing you full access to resources such as information, files, databases, finances, locations, and anything else.

This approval impacts your ability to access the system and the extent to which you can do so.

### How to Create a Node.js Server and Connect your Database

Now, add the following snippets to your app.js, index.js, database.js, and .env files in that order to establish our Node.js server and connect our database.

In our database.js.:

config/database.js:

const mongoose = require("mongoose");

const { MONGO\_URI } = process.env;

exports.connect = () => {

// Connecting to the database

mongoose

.connect(MONGO\_URI, {

useNewUrlParser: true,

useUnifiedTopology: true,

useCreateIndex: true,

useFindAndModify: false,

})

.then(() => {

console.log("Successfully connected to database");

})

.catch((error) => {

console.log("database connection failed. exiting now...");

console.error(error);

process.exit(1);

});

};

## JSON Web Tokens are an open, industry standard [**RFC 7519**](https://tools.ietf.org/html/rfc7519) method for representing claims securely between two parties.

Essentially, JWT are strings of data that can be used to authenticate and exchange information between a server and a client.

The flow of information is as follows:

* Client sends credentials to the server
* Server verifies the credentials, generates a JWT and sends it back as a response
* Subsequent requests from the client have a JWT in the request headers
* Server validates the token and if valid, provide the requested response.