ADVANCED NODE JS AND DATABASE

NoSQL originally referring to non SQL or non relational is a database that provides a mechanism for storage and retrieval of data. This data is modeled in means other than the tabular relations used in relational databases. Such databases came into existence in the late 1960s, but did not obtain the NoSQL moniker until a surge of popularity in the early twenty-first century. NoSQL databases are used in real-time web applications and big data and their use are increasing over time. NoSQL systems are also sometimes called Not only SQL to emphasize the fact that they may support SQL-like query languages. A NoSQL database includes simplicity of design, simpler horizontal scaling to clusters of machines and finer control over availability. The data structures used by NoSQL databases are different from those used by default in relational databases which makes some operations faster in NoSQL. The suitability of a given NoSQL database depends on the problem it should solve. Data structures used by NoSQL databases are sometimes also viewed as more flexible than relational database tables. Many NoSQL stores compromise consistency in favor of availability, speed and partition tolerance. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages, lack of standardized interfaces, and huge previous investments in existing relational databases. Most NoSQL stores lack true ACID(Atomicity, Consistency, Isolation, Durability) transactions but a few databases, such as MarkLogic, Aerospike, FairCom c-treeACE, Google Spanner (though technically a NewSQL database), Symas LMDB, and OrientDB have made them central to their designs. Most NoSQL databases offer a concept of eventual consistency in which database changes are propagated to all nodes so queries for data might not return updated data immediately or might result in reading data that is not accurate which is a problem known as stale reads. Also some NoSQL systems may exhibit lost writes and other forms of data loss. Some NoSQL systems provide concepts such as write-ahead logging to avoid data loss. For distributed transaction processing across multiple databases, data consistency is an even bigger challenge. This is difficult for both NoSQL and relational databases. Even current relational databases do not allow referential integrity constraints to span databases. There are few systems that maintain both X/Open XA standards and ACID transactions for distributed transaction processing.
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 databases 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 a huge amount of data because of scalability, as the data grows NoSQL scale itself to handle that data in an 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 a 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 the 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 in 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 are 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. This 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. **Graph Databases**: Amazon Neptune, Neo4j
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 a huge amount of data needs to be stored and retrieved.
2. The relationship between the data you store is not that important
3. The data changes over time and is not structured.
4. Support of Constraints and Joins is not required at the database level
5. The data is growing continuously and you need to scale the database regularly to handle the data.

MongoDB system overview

What is MongoDB?

MongoDB is a document-oriented NoSQL database used for high volume data storage. Instead of using tables and rows as in the traditional relational databases, MongoDB makes use of collections and documents. Documents consist of key-value pairs which are the basic unit of data in MongoDB. Collections contain sets of documents and function which is the equivalent of relational database tables. MongoDB is a database which came into light around the mid-2000s.

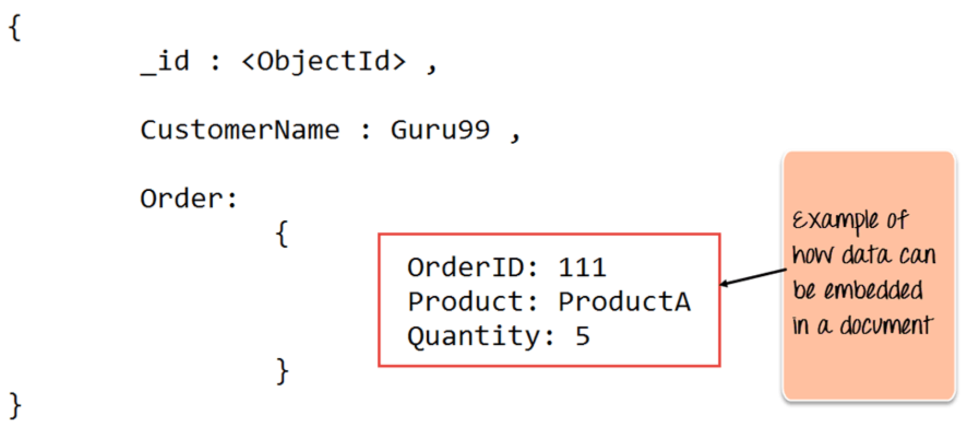
MongoDB Features
Each database contains collections which in turn contains documents. Each document can be different with a varying number of fields. The size and content of each document can be different from each other.
The document structure is more in line with how developers construct their classes and objects in their respective programming languages. Developers will often say that their classes are not rows and columns but have a clear structure with key-value pairs.
The rows (or documents as called in MongoDB) doesn't need to have a schema defined beforehand. Instead, the fields can be created on the fly.
The data model available within MongoDB allows you to represent hierarchical relationships, to store arrays, and other more complex structures more easily.
Scalability – The MongoDB environments are very scalable. Companies across the world have defined clusters with some of them running 100+ nodes with around millions of documents within the database

MongoDB Example

The below example shows how a document can be modeled in MongoDB.

1. The _id field is added by MongoDB to uniquely identify the document in the collection.
2. What you can note is that the Order Data (OrderID, Product, and Quantity) which in RDBMS will normally be stored in a separate table, while in MongoDB it is actually stored as an embedded document in

the collection itself. This is one of the key differences in how data is modeled in MongoDB.



```
{
        _id : <ObjectId> ,

        CustomerName : Guru99 ,

        Order:
                {
                        OrderID: 111
                        Product: ProductA
                        Quantity: 5
                }
}
```

Key Components of MongoDB Architecture

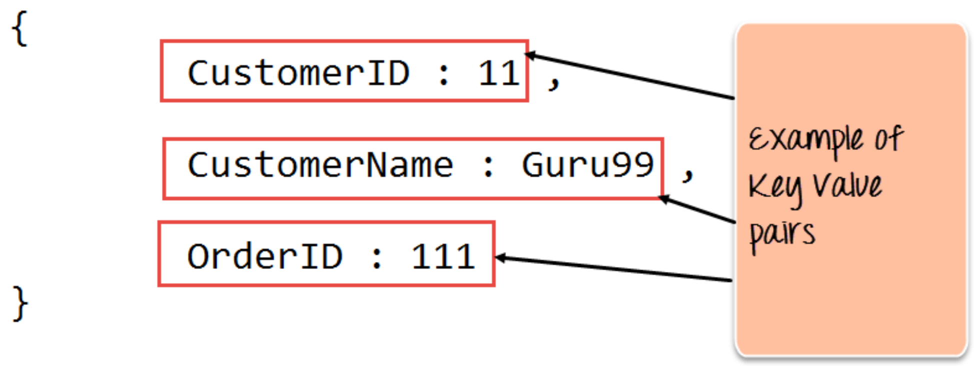
Below are a few of the common terms used in MongoDB

1. **_id** – This is a field required in every MongoDB document. The _id field represents a unique value in the MongoDB document. The _id field is like the document's primary key. If you create a new document without an _id field, MongoDB will automatically create the field. So for example, if we see the example of the above customer table, Mongo DB will add a 24 digit unique identifier to each document in the collection.

_Id	CustomerID	CustomerName	OrderID
563479cc8a8a4246bd27d784	11	Guru99	111
563479cc7a8a4246bd47d784	22	Trevor Smith	222
563479cc9a8a4246bd57d784	33	Nicole	333

2. **Collection** – This is a grouping of MongoDB documents. A collection is the equivalent of a table which is created in any other RDMS such as Oracle or MS SQL. A collection exists within a single database. As seen from the introduction collections don't enforce any sort of structure.
3. **Cursor** – This is a pointer to the result set of a query. Clients can iterate through a cursor to retrieve results.
4. **Database** – This is a container for collections like in RDMS wherein it is a container for tables. Each database gets its own set of files on the file system. A MongoDB server can store multiple databases.

5. **Document** – A record in a MongoDB collection is basically called a document. The document, in turn, will consist of field name and values.
6. **Field** – A name-value pair in a document. A document has zero or more fields. Fields are analogous to columns in relational databases.The following diagram shows an example of Fields with Key value pairs. So in the example below CustomerID and 11 is one of the key value pair's defined in the document.



7. **JSON** – This is known as JavaScript Object Notation. This is a human-readable, plain text format for expressing structured data. JSON is currently supported in many programming languages.

Just a quick note on the key difference between the _id field and a normal collection field. The _id field is used to uniquely identify the documents in a collection and is automatically added by MongoDB when the collection is created.

Why Use MongoDB?

Below are the few of the reasons as to why one should start using MongoDB

1. Document-oriented – Since MongoDB is a NoSQL type database, instead of having data in a relational type format, it stores the data in documents. This makes MongoDB very flexible and adaptable to real business world situation and requirements.
2. Ad hoc queries – MongoDB supports searching by field, range queries, and regular expression searches. Queries can be made to return specific fields within documents.
3. Indexing – Indexes can be created to improve the performance of searches within MongoDB. Any field in a MongoDB document can be indexed.

4. Replication – MongoDB can provide high availability with replica sets. A replica set consists of two or more mongo DB instances. Each replica set member may act in the role of the primary or secondary replica at any time. The primary replica is the main server which interacts with the client and performs all the read/write operations. The Secondary replicas maintain a copy of the data of the primary using built-in replication. When a primary replica fails, the replica set automatically switches over to the secondary and then it becomes the primary server.
5. Load balancing – MongoDB uses the concept of sharding to scale horizontally by splitting data across multiple MongoDB instances. MongoDB can run over multiple servers, balancing the load and/or duplicating data to keep the system up and running in case of hardware failure.

Data Modelling in MongoDB

As we have seen from the Introduction section, the data in MongoDB has a flexible schema. Unlike in SQL databases, where you must have a table's schema declared before inserting data, MongoDB's collections do not enforce document structure. This sort of flexibility is what makes MongoDB so powerful.

When modeling data in Mongo, keep the following things in mind

1. What are the needs of the application – Look at the business needs of the application and see what data and the type of data needed for the application. Based on this, ensure that the structure of the document is decided accordingly.
2. What are data retrieval patterns – If you foresee a heavy query usage then consider the use of indexes in your data model to improve the efficiency of queries.
3. Are frequent inserts, updates and removals happening in the database? Reconsider the use of indexes or incorporate sharding if required in your data modeling design to improve the efficiency of your overall MongoDB environment.

Difference between MongoDB & RDBMS

Below are some of the key term differences between MongoDB and RDBMS

RDBMS	MongoDB	Difference
Table	Collection	In RDBMS, the table contains the columns and rows which are used to store the data whereas, in MongoDB, this same structure is known as a collection. The collection contains documents which in turn contains Fields, which in turn are key-value pairs.
Row	Document	In RDBMS, the row represents a single, implicitly structured data item in a table. In MongoDB, the data is stored in documents.
Column	Field	In RDBMS, the column denotes a set of data values. These in MongoDB are known as Fields.
Joins	Embedded documents	In RDBMS, data is sometimes spread across various tables and in order to show a complete view of all data, a join is sometimes formed across tables to get the data. In MongoDB, the data is normally stored in a single collection, but separated by using Embedded documents. So there is no concept of joins in MongoDB.

Using the MongoDB Shell

The MongoDB shell is a great tool for navigating, inspecting, and even manipulating document data. If you're running MongoDB on your local machine, firing up the shell is as simple as typing mongo and hitting enter, which will connect to MongoDB at localhost on the standard port (27017). If you're connecting to a MongoDB Atlas cluster or other remote instance, then add the connection string after the command mongo .

Here are a few quick shell examples:

List Databases

```
> show dbs;
admin     0.000GB
config    0.000GB
local     0.000GB
my_database  0.004GB
>
```

List Collections

```
> use my_database;
> show collections;
users
posts
>
```

Count Documents in a Collection

```
> use my_database;
> db.users.count()
20234
>
```

Find the First Document in a Collection

```
> db.users.findOne()
{
    "_id": ObjectId("5ce45d7606444f199acfba1e"),
    "name": {given: "Alex", family: "Smith"},
    "email": "email@example.com"
    "age": 27
}
>
```

Find a Document by ID

```
> db.users.findOne({_id: ObjectId("5ce45d7606444f199acfba1e")})
```

```
{
    "_id": ObjectId("5ce45d7606444f199acfba1e"),
    "name": {given: "Alex", family: "Smith"},
    "email": "email@example.com",
    "age": 27
}
>
```

Querying MongoDB Collections

The MongoDB Query Language (MQL) uses the same syntax as documents, making it intuitive and easy to use for even advanced querying. Let’s look at a few MongoDB query examples.

Find a Limited Number of Results

```
> db.users.find().limit(10)
…
>
```

Find Users by Family name

```
> db.users.find({"name.family": "Smith"}).count()
1
>
```

Note that we enclose “name.family” in quotes, because it has a dot in the middle.

Query Documents by Numeric Ranges

```
// All posts having “likes” field with numeric value greater than one:
> db.post.find({likes: {$gt: 1}})
// All posts having 0 likes
> db.post.find({likes: 0})
// All posts that do NOT have exactly 1 like
```

```
> db.post.find({likes: {$ne: 1}})
```

Sort Results by a Field

```
// order by age, in ascending order (smallest values first)
> db.user.find().sort({age: 1})
{
    "_id": ObjectId("5ce45d7606444f199acfba1e"),
    "name": {given: "Alex", family: "Smith"},
    "email": "email@example.com",
    "age": 27
}
{
    _id: ObjectId("5effaa5662679b5af2c58829"),
    email: “email@example.com”,
    name: {given: “Jesse”, family: “Xiao”},
    age: 31
}
>

// order by age, in descending order (largest values first)
> db.user.find().sort({age: -1})
{
    _id: ObjectId("5effaa5662679b5af2c58829"),
    email: “email@example.com”,
    name: {given: “Jesse”, family: “Xiao”},
    age: 31
}
{
    "_id": ObjectId("5ce45d7606444f199acfba1e"),
    "name": {given: "Alex", family: "Smith"},
    "email": "email@example.com",
    "age": 27
}
>
```

Managing Indexes

MongoDB allows you to create indexes, even on nested fields in subdocuments, to keep queries performing well even as collections grow very large.

Create an Index

```
> db.user.createIndex({"name.family": 1})
Create a Unique Index
> db.user.createIndex({email: 1}, {unique: true})
```

Unique indexes allow you to ensure that there is at most one record in the collection with a given value for that field – very useful with things like email addresses!

See Indexes on a Collection

```
> db.user.getIndexes()
[
    {
        "v" : 2,
        "key" : {
            "_id" : 1
        },
        "name" : "_id_",
        "ns" : "my_database.user"
    },
    {
        "v" : 2,
        "key" : {
            "name.given" : 1
        },
        "name" : "name.given_1",
        "ns" : "my_database.user"
    }
]
```

Note that by default, collections always have an index on the _id field, for easy document retrieval by primary key, so any additional indexes will be listed after that.

Drop an Index

```
> db.user.dropIndex("name.given_1")
```

Request body parsing in Express

Middleware functions are functions that have access to the **request object (req)**, the **response object (res)**, and the next middleware function in the application's request-response cycle. These functions are used to modify **req** and **res** objects for tasks like parsing request bodies, adding response headers, etc.

Here is a simple example of a middleware function in action −

```
var express = require('express');
var app = express();

//Simple request time logger
app.use(function(req, res, next){
   console.log("A new request received at " + Date.now());

   //This function call is very important. It tells that more processing is
   //required for the current request and is in the next middleware
   function route handler.
   next();
});

app.listen(3000);
```

The above middleware is called for every request on the server. So after every request, we will get the following message in the console −

```
A new request received at 1467267512545
```

To restrict it to a specific route (and all its subroutes), provide that route as the first argument of ***app.use()***. For Example,

```
var express = require('express');
var app = express();

//Middleware function to log request protocol
```

```
app.use('/things', function(req, res, next){
   console.log("A request for things received at " + Date.now());
   next();
});

// Route handler that sends the response
app.get('/things', function(req, res){
   res.send('Things');
});

app.listen(3000);
```

Now whenever you request any subroute of '/things', only then it will log the time.

Order of Middleware Calls

One of the most important things about middleware in Express is the order in which they are written/included in your file; the order in which they are executed, given that the route matches also needs to be considered.

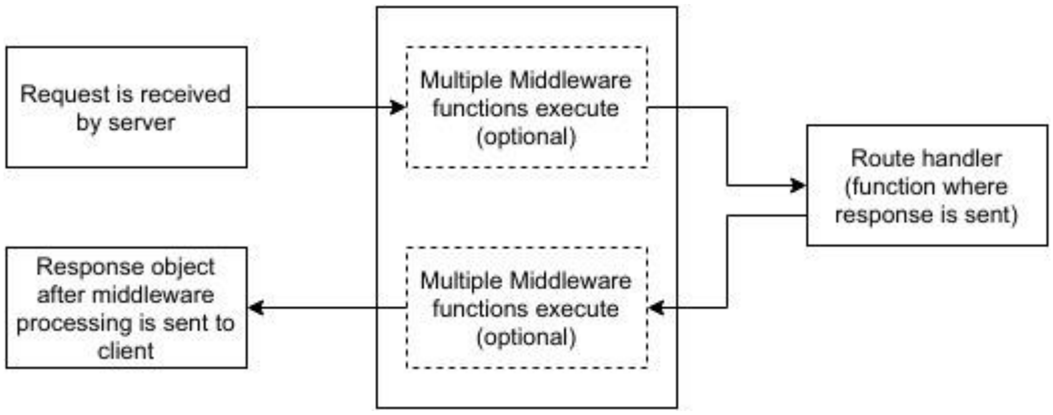
For example, in the following code snippet, the first function executes first, then the route handler and then the end function. This example summarizes how to use middleware before and after route handler; also how a route handler can be used as a middleware itself.

```
var express = require('express');
var app = express();

//First middleware before response is sent
app.use(function(req, res, next){
   console.log("Start");
   next();
});

//Route handler
app.get('/', function(req, res, next){
   res.send("Middle");
   next();
});

app.use('/', function(req, res){
   console.log('End');
});

app.listen(3000);
```

When we visit '/' after running this code, we receive the response as **Middle** and on our console −

```
Start
End
```

The following diagram summarizes what we have learnt about middleware −


Request is received by server
Multiple Middleware functions execute (optional)
Route handler (function where response is sent)
Multiple Middleware functions execute (optional)
Response object after middleware processing is sent to client

Now that we have covered how to create our own middleware, let us discuss some of the most commonly used community created middleware.

Third Party Middleware

A list of Third party middleware for Express is available here. Following are some of the most commonly used middleware; we will also learn how to use/mount these −

body-parser

This is used to parse the body of requests which have payloads attached to them. To mount body parser, we need to install it using **npm install** --save body-parser and to mount it, include the following lines in your index.js −

```
var bodyParser = require('body-parser');

//To parse URL encoded data
app.use(bodyParser.urlencoded({ extended: false }))

//To parse json data
app.use(bodyParser.json())
```

To view all available options for body-parser, visit its github page.

cookie-parser

It parses *Cookie* header and populate req.cookies with an object keyed by cookie names. To mount cookie parser, we need to install it using npm install --save cookie-parser and to mount it, include the following lines in your index.js −

```
var cookieParser = require('cookie-parser');
app.use(cookieParser())
```

express-session

It creates a session middleware with the given options. We will discuss its usage in the Sessions section.

We have many other third party middleware in ExpressJS. However, we have discussed only a few important ones here.

Node.js body parsing middleware.

Parse incoming request bodies in a middleware before your handlers, available under the `req.body` property.

Note As `req.body`'s shape is based on user-controlled input, all properties and values in this object are untrusted and should be validated before trusting. For example, `req.body.foo.toString()` may fail in multiple ways, for example the `foo` property may not be there or may not be a string, and `toString` may not be a function and instead a string or other user input.

Learn about the anatomy of an HTTP transaction in Node.js.

This does not handle multipart bodies, due to their complex and typically large nature. For multipart bodies, you may be interested in the following modules:

- busboy and connect-busboy
- multiparty and connect-multiparty
- formidable
- multer

This module provides the following parsers:

- JSON body parser
- Raw body parser
- Text body parser
- URL-encoded form body parser

Other body parsers you might be interested in:

- body
- co-body

Installation

```
$ npm install body-parser
```

API

```
var bodyParser = require('body-parser')
```

- The `bodyParser` object exposes various factories to create middlewares. All middlewares will populate the `req.body` property with the parsed body when the `Content-Type` request header matches the `type` option, or an empty object (`{}`) if there was no body to parse, the `Content-Type` was not matched, or an error occurred.

- The various errors returned by this module are described in the errors section.
- **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 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.
- **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 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.

- ## bodyParser.raw([options])
- Returns middleware that parses all bodies as a `Buffer` and only looks at requests where the `Content-Type` header matches the `type` option. This parser 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`). This will be a `Buffer` object of the body.
- **Options**
- The `raw` 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 library for parsing. Defaults to `'100kb'`.
- **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 library and this can be an extension name (like `bin`), a mime type (like `application/octet-stream`), or a mime type with a wildcard (like `*/*` or `application/*`). 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/octet-stream`.
- **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.
- ## bodyParser.text([options])
- Returns middleware that parses all bodies as a string and only looks at requests where the `Content-Type` header matches the `type` option. This parser supports automatic inflation of `gzip` and `deflate` encodings.
- A new `body` string containing the parsed data is populated on the `request` object after the middleware (i.e. `req.body`). This will be a string of the body.
- **Options**

- The `text` function takes an optional `options` object that may contain any of the following keys:
- **defaultCharset**
- Specify the default character set for the text content if the charset is not specified in the `Content-Type` header of the request. Defaults to `utf-8`.
- **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 library for parsing. Defaults to `'100kb'`.
- **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 library and this can be an extension name (like `txt`), a mime type (like `text/plain`), or a mime type with a wildcard (like `*/*` or `text/*`). If a function, the `type` option is called as `fn(req)` and the request is parsed if it returns a truthy value. Defaults to `text/plain`.
- **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.
- **bodyParser.urlencoded([options])**
- Returns middleware that only parses `urlencoded` bodies and only looks at requests where the `Content-Type` header matches the `type` option. This parser accepts only UTF-8 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`). This object will contain key-value pairs, where the value can be a string or array (when `extended` is `false`), or any type (when `extended` is `true`).
- **Options**
- The `urlencoded` function takes an optional `options` object that may contain any of the following keys:
- **extended**

- The extended option allows to choose between parsing the URL-encoded data with the querystring library (when false) or the qs library (when true). The “extended” syntax allows for rich objects and arrays to be encoded into the URL-encoded format, allowing for a JSON-like experience with URL-encoded. For more information, please see the qs library.
- Defaults to true, but using the default has been deprecated. Please research into the difference between qs and querystring and choose the appropriate setting.
- **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 library for parsing. Defaults to '100kb'.
- **parameterLimit**
- The parameterLimit option controls the maximum number of parameters that are allowed in the URL-encoded data. If a request contains more parameters than this value, a 413 will be returned to the client. Defaults to 1000.
- **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 library and this can be an extension name (like urlencoded), a mime type (like application/x-www-form-urlencoded), or a mime type with a wildcard (like */x-www-form-urlencoded). 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/x-www-form-urlencoded.
- **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

NodeJS MongoDB connection

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();
});
```

```
C:\Users\          >node demo_create_mongo_db.js
```

Adding and retrieving data to MongoDB from NodeJS

Insert Into Collection

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

A **document** in MongoDB is the same as a **record** in MySQL

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:

Example

Insert a 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");
  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();
  });
});
```

Run example »

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
```

Find One

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();
  });
});
```

Run example »

Save the code above in a file called "demo_mongodb_findone.js" and run the file:

Run "demo_mongodb_findone.js"

```
C:\Users\Your Name>node demo_mongodb_findone.js
```

Which will give you this result:

```
Company Inc.
```

Find All

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();
  });
});
```

Run example »

Save the code above in a file called "demo_mongodb_find.js" and run the file:

```
Run "demo_mongodb_find.js"

C:\Users\Your Name>node demo_mongodb_find.js
```

Which will give you this result:

```
[
  { _id: 58fdbf5c0ef8a50b4cdd9a84 , name: 'John', address: 'Highway
71'},
  { _id: 58fdbf5c0ef8a50b4cdd9a85 , name: 'Peter', address: 'Lowstreet
4'},
  { _id: 58fdbf5c0ef8a50b4cdd9a86 , name: 'Amy', address: 'Apple st
652'},
  { _id: 58fdbf5c0ef8a50b4cdd9a87 , name: 'Hannah', address: 'Mountain
21'},
  { _id: 58fdbf5c0ef8a50b4cdd9a88 , name: 'Michael', address: 'Valley
345'},
  { _id: 58fdbf5c0ef8a50b4cdd9a89 , name: 'Sandy', address: 'Ocean blvd
2'},
  { _id: 58fdbf5c0ef8a50b4cdd9a8a , name: 'Betty', address: 'Green Grass
1'},
  { _id: 58fdbf5c0ef8a50b4cdd9a8b , name: 'Richard', address: 'Sky st
331'},
  { _id: 58fdbf5c0ef8a50b4cdd9a8c , name: 'Susan', address: 'One way
98'},
  { _id: 58fdbf5c0ef8a50b4cdd9a8d , name: 'Vicky', address: 'Yellow
Garden 2'},
  { _id: 58fdbf5c0ef8a50b4cdd9a8e , name: 'Ben', address: 'Park Lane
38'},
  { _id: 58fdbf5c0ef8a50b4cdd9a8f , name: 'William', address: 'Central
st 954'},
  { _id: 58fdbf5c0ef8a50b4cdd9a90 , name: 'Chuck', address: 'Main Road
989'},
  { _id: 58fdbf5c0ef8a50b4cdd9a91 , name: 'Viola', address: 'Sideway
1633'}
]
```

Find Some

The second parameter of the `find()` method is the `projection` object that describes which fields to include in the result.

This parameter is optional, and if omitted, all fields will be included in the result.

Example

Return the fields "name" and "address" of 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({}, { projection: { _id: 0, name: 1,
address: 1 } }).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

Run example »

Save the code above in a file called "demo_mongodb_find_fields.js" and run the file:

Run "demo_mongodb_find_fields.js"

```
C:\Users\Your Name>node demo_mongodb_find_fields.js
```

Which will give you this result:

```
[
  { 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'}
]
```

You are not allowed to specify both 0 and 1 values in the same object (except if one of the fields is the _id field). If you specify a field with the value 0, all other fields get the value 1, and vice versa:

Example

This example will exclude "address" from the result:

```
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({}, { projection: { address: 0 }
}).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

Run example »

To exclude the _id field, you must set its value to 0:

Example

This example will return only the "name" field:

```
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({}, { projection: { _id: 0, name: 1 }
}).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

Run example »

Example

This example will give you the same result as the first example; return all fields except the _id field:

```
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({}, { projection: { _id: 0 }
}).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

Run example »

Example

You get an error if you specify both 0 and 1 values in the same object (except if one of the fields is the _id field):

```
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({}, { projection: { name: 1,
address: 0 } }).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

Run example »

The Result Object

As you can see from the result of the example above, the result can be converted into an array containing each document as an object.

To return e.g. the address of the third document, just refer to the third array object's address property:

Example

```
Return the address of the third document:

console.log(result[2].address);
```

Which will produce this result:

```
Apple st 652
```

Handling SQL databases from NodeJS

Node.js MySQL

Node.js can be used in database applications.

One of the most popular databases is MySQL.

MySQL Database

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

You can download a free MySQL database at https://www.mysql.com/downloads/.

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!");
});
```

Run example »

Save the code above in a file called "demo_db_connection.js" and run the file:

Run "demo_db_connection.js"

```
C:\Users\Your Name>node demo_db_connection.js
```

Which will give you this result:

```
Connected!
```

Now you can start querying the database using SQL statements.

Query a Database

Use SQL statements to read from (or write to) a MySQL database. This is also called "to query" the database.

The connection object created in the example above, has a method for querying the database:

```
con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  con.query(sql, function (err, result) {
    if (err) throw err;
    console.log("Result: " + result);
  });
});
```

The query method takes an sql statements as a parameter and returns the result

HTTP Cookies in Node.js

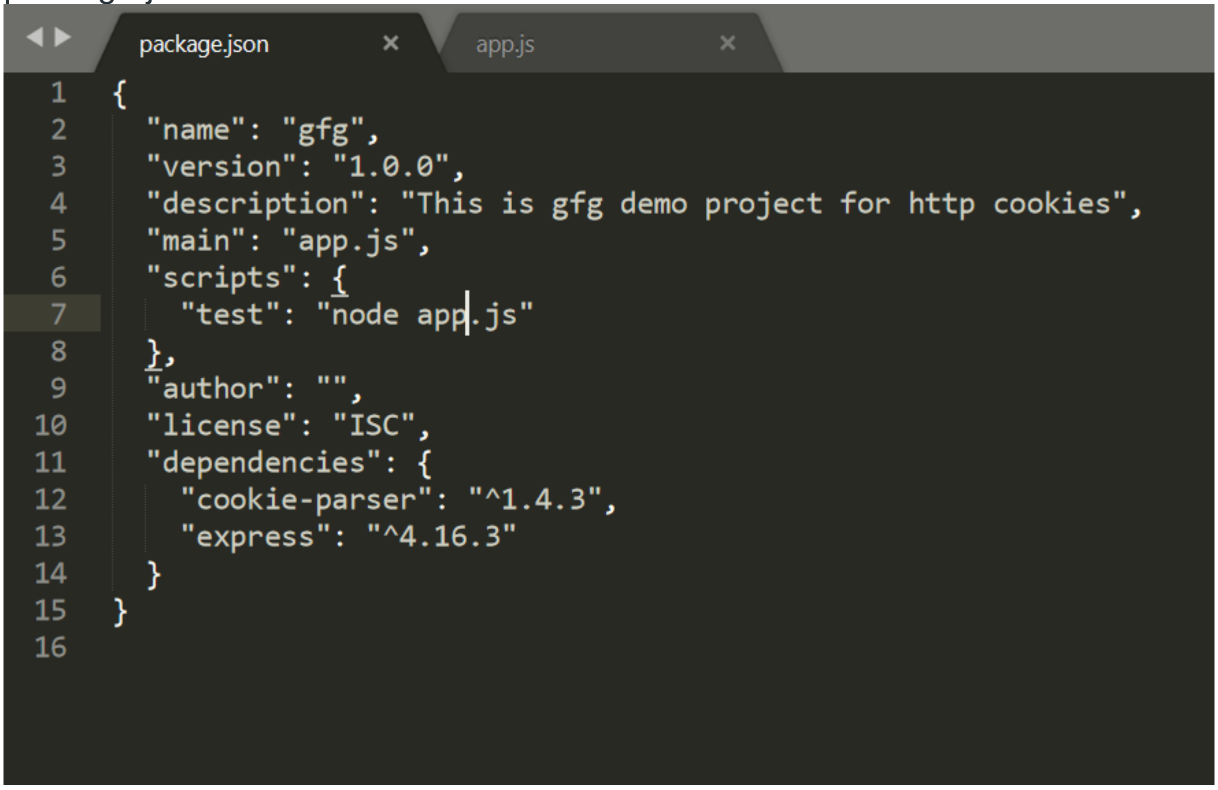
Cookies are small data that are stored on a client side and sent to the client along with server requests. Cookies have various functionality, they can be used for maintaining sessions and adding user-specific features in your web app. For this, we will use **cookie-parser** module of npm which provides middleware for parsing of cookies.
First set your directory of the command prompt to root folder of the project and run the following command:

```
npm init
```

This will ask you details about your app and finally will create a **package.json** file.
After that run the following command and it will install the required module and add them in your package.json file

```
npm install express cookie-parser --save
```

package.json file looks like this :



```
{
  "name": "gfg",
  "version": "1.0.0",
  "description": "This is gfg demo project for http cookies",
  "main": "app.js",
  "scripts": {
    "test": "node app.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "cookie-parser": "^1.4.3",
    "express": "^4.16.3"
  }
}
```

After that we will setup basic express app by writing following code in our app.js file in root directory .

```
let express = require('express');

//setup express app

let app = express()





//basic route for homepage

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

res.send('welcome to express app');

});
```

```
//server listens to port 3000

app.listen(3000, (err)=>{

if(err)

throw err;

console.log('listening on port 3000');

});
```

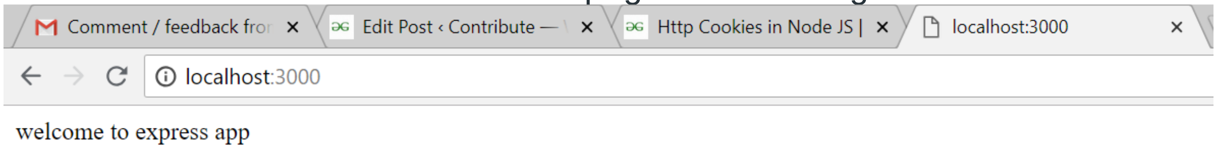
After that if we run the command

```
node app.js
```

It will start our server on port 3000 and if go to the url: localhost:3000, we will get a page showing the message :

`welcome to express app`

Here is screenshot of localhost:3000 page after starting the server :



welcome to express app

So until now we have successfully set up our express app now let’s start with cookies.

For cookies first, we need to import the module in our app.js file and use it like other middlewares.

```
var cookieParser = require('cookie-parser');
app.use(cookieParser());
```

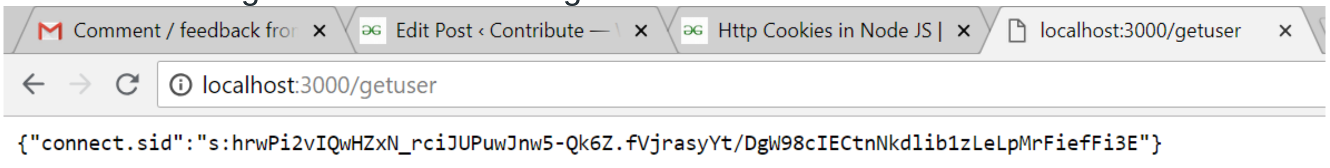
Let’s say we have a user and we want to add that user data in the cookie then we have to add that cookie to the response using the following code :

```
res.cookie(name_of_cookie, value_of_cookie);
```

This can be explained by the following example :

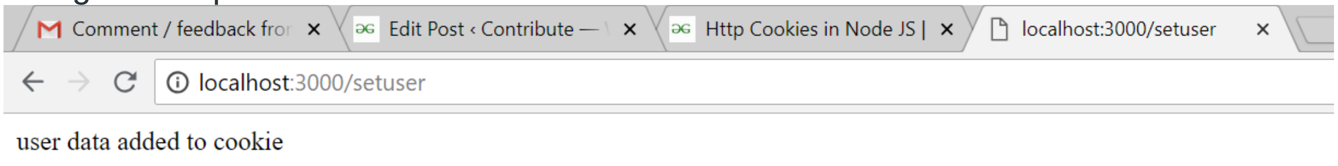
```
let express = require('express');

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');
});
```

So if we restart our server and make a get request to the route: localhost:3000/getuser before setting the cookies it is as follows :



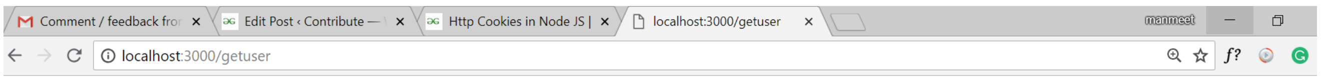
```
{"connect.sid":"s:hrwPi2vIQwHZxN_rciJUPuwJnw5-Qk6Z.fVjrasyYt/DgW98cIECtnNkdlib1zLeLpMrFiefFi3E"}
```

After making a request to localhost:3000/setuser it will add user data to cookie and gives output as follows :



```
user data added to cookie
```

Now if we again make a request to localhost:3000/getuser as this route is iterating user data from cookies using **req.cookies** so output will be as follows :



```
{"connect.sid":"s:hrwPi2vIQwHZxN_rciJUPuwJnw5-Qk6Z.fVjrasyYt/DgW98cIECtnNkdlib1zLeLpMrFiefFi3E","userData":{"name":"Ritik","Age":"18"}}
```

If we have multiple objects pushed in cookies then we can access specific cookie using **req.cookie.cookie_name** .

Adding Cookie with expiration Time

We can add a cookie with some expiration time i.e. after that time cookies will be destroyed automatically. For this, we need to pass an extra property to the res.cookie object while setting the cookies.

It can be done by using any of the two ways :

```
//Expires after 400000 ms from the time it is set.
res.cookie(cookie_name, 'value', {expire: 400000 + Date.now()});
```

```
//It also expires after 400000 ms from the time it is set.
res.cookie(cookie_name, 'value', {maxAge: 360000});
```

Destroy the cookies :
We can destroy cookies using following code :
`res.clearCookie(cookieName);`

Now let us make a logout route which will destroy user data from the cookie. Now our app.js looks like :

```
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);

});



//Route for destroying cookie

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

//it will clear the userData cookie

res.clearCookie('userData');
res.send('user logout successfully');

});




//server listens to port 3000

app.listen(3000, (err)=>{

if(err)

throw err;

console.log('listening on port 3000');

});
```

For destroying the cookie make get request to following link

Handling User Authentication with NodeJS

Introduction

Creating a user registration form employs the management of the registered user. This is where user role authentication comes into play. Role authentication ensures that non-admin users cannot make changes or access exclusive information. It grants administrative privileges to admin users and basic privileges to basic users.

You can build your own authentication functionality with web tokens like JSON Web Token (JWT) or use a trusted third-party customer identity and access management (CIAM) software like LoginRadius.

Goal

This tutorial helps you:

- understand the differences between the Admin role and the Basic user role;
- use JWT to authenticate users; and
- learn role-based authentication using JWT in a simple Node.js app.

Prerequisites

You have installed the following:

- Node
- MongoDB
- a Text Editor

You already understand JavaScript E56 Syntax.
Now that everything is in place, let's set up your database.

Set Up a Mongo Database

You'll store all your user data — which includes username, password, and role — in MongoDB.

Install a node package called Mongoose that will connect to MongoDB. Then create a user `schema` for your application.

```
npm init
npm install mongoose
```

`npm init` sets up your new project and creates a `package.json` file with the credentials.
After installing mongoose, create a new file `db.js` in the project's directory and require `mongoose`.

```
const Mongoose = require("mongoose")
```

With the help of mongoose, you can connect your application to MongoDB:

```
// db.js
const Mongoose = require("mongoose")
const localDB = `mongodb://localhost:27017/role_auth`
const connectDB = async () => {
  await Mongoose.connect(localDB, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  })
  console.log("MongoDB Connected")
}
module.exports = connectDB
```

The code snippet here connects to `mongodb://localhost:27017` and then specifies the name of the database `/role_auth`.

The function `connectDB` awaits for the connection, which contains the `URI` and `options` as a second parameter. If it connects without errors, it will log out `MongoDB Connected`. Error issues will be fixed while connecting to the database. After this, it exported the function for use in the server.

Set Up the Server

You need to install some dependencies that you'll use in this tutorial.

```
  npm i express nodemon
```

Express.js is a Node.js framework for building web applications quickly and easily.
Nodemon is a tool that watches the file system and automatically restarts the server when there is a change.
You require `express` in your application to listen for a connection on port `5000`. Create a new file `server.js` in the root directory and create the listening event:

```
const express = require("express")
const app = express()
const PORT = 5000
app.listen(PORT, () => console.log(`Server Connected to port ${PORT}`))
```

The next step is to test your application. Open up your `package.json` file and add the following to `scripts`:

```
"scripts": {
  "start": "node server.js",
  "dev": "nodemon server.js"
}
```

Open your terminal and run `npm run dev` to start the server.

Connect to the Database

Earlier, you've created a function that connects to MongoDB and exported that function. Now import that function into your `server.js`:

```
const connectDB = require("./db");
...
//Connecting the Database
connectDB();
```

You also need to create an error handler that catches every `unhandledRejection` error.

```
const server = app.listen(PORT, () =>
  console.log(`Server Connected to port ${PORT}`)
)
// Handling Error
process.on("unhandledRejection", err => {
  console.log(`An error occurred: ${err.message}`)
  server.close(() => process.exit(1))
})
```

The listening event is assigned to a constant `server`. If an `unhandledRejection` error occurs, it logs out the error and closes the `server` with an exit code of 1.

Create User Schema

Schema is like a blueprint that shows how the database will be constructed.
You'll structure a user schema that contains username, password, and role.

Create a new folder `model` in the project's directory, and create a file called `User.js`.
Now open `User.js` and create the user schema:

```
// user.js
const Mongoose = require("mongoose")
const UserSchema = new Mongoose.Schema({
  username: {
    type: String,
    unique: true,
    required: true,
  },
  password: {
    type: String,
    minlength: 6,
    required: true,
  },
  role: {
    type: String,
    default: "Basic",
    required: true,
  },
})
```

In the schema, the `username` will be unique, required, and will accept `strings`.
You've specified the minimum characters(6) the `password` field will accept.
The `role` field grants a default value (basic) that you can change if needed.
Now, you need to create a user model and export it:

```
const User = Mongoose.model("user", UserSchema)
module.exports = User
```

You've created the user model by passing the `UserSchema` as the second argument while the first argument is the name of the model `user`.

Perform CRUD Operations

You'll create functions that handle:

- adding users;
- getting all users;
- updating the role of users; and,
- deleting users.

Register Function

As the name implies, this function will handle the registrations of users.

Let's create a new folder named `Auth`. It will contain the Authentication file and the Route set-up file.
After creating the `Auth` folder, add two files — `Auth.js` and `Route.js`.
Now open up our `Auth.js` file and import that `User` model:

```
const User = require("../model/User")
```

The next step is to create an `async express` function that will take the user's data and register it in the database.
You need to use an Express middleware function that will grant access to the user's data from the body. You'll use this function in the `server.js` file:

```
const app = express()
app.use(express.json())
```

Let's go back to your `Auth.js` file and create the register function:

```
// auth.js
exports.register = async (req, res, next) => {
  const { username, password } = req.body
  if (password.length < 6) {
    return res.status(400).json({ message: "Password less than 6 characters"
})
  }
  try {
    await User.create({
      username,
      password,
    }).then(user =>
      res.status(200).json({
        message: "User successfully created",
        user,
      })
    )
  } catch (err) {
    res.status(401).json({
      message: "User not successful created",
      error: error.mesage,
      })
    }
}
```

The exported `register` function will be used to set up the routes. You got the username and password from the `req.body` and created a `tryCatch` block that will create the user if successful; else, it returns status code `401` with the error message.

Set Up Register Route

You'll create a route to `/register` using `express.Router`. Import the `register` function into your `route.js` file, and use it as the route's function:

```
const express = require("express")
const router = express.Router()
const { register } = require("./auth")
router.route("/register").post(register)
module.exports = router
```

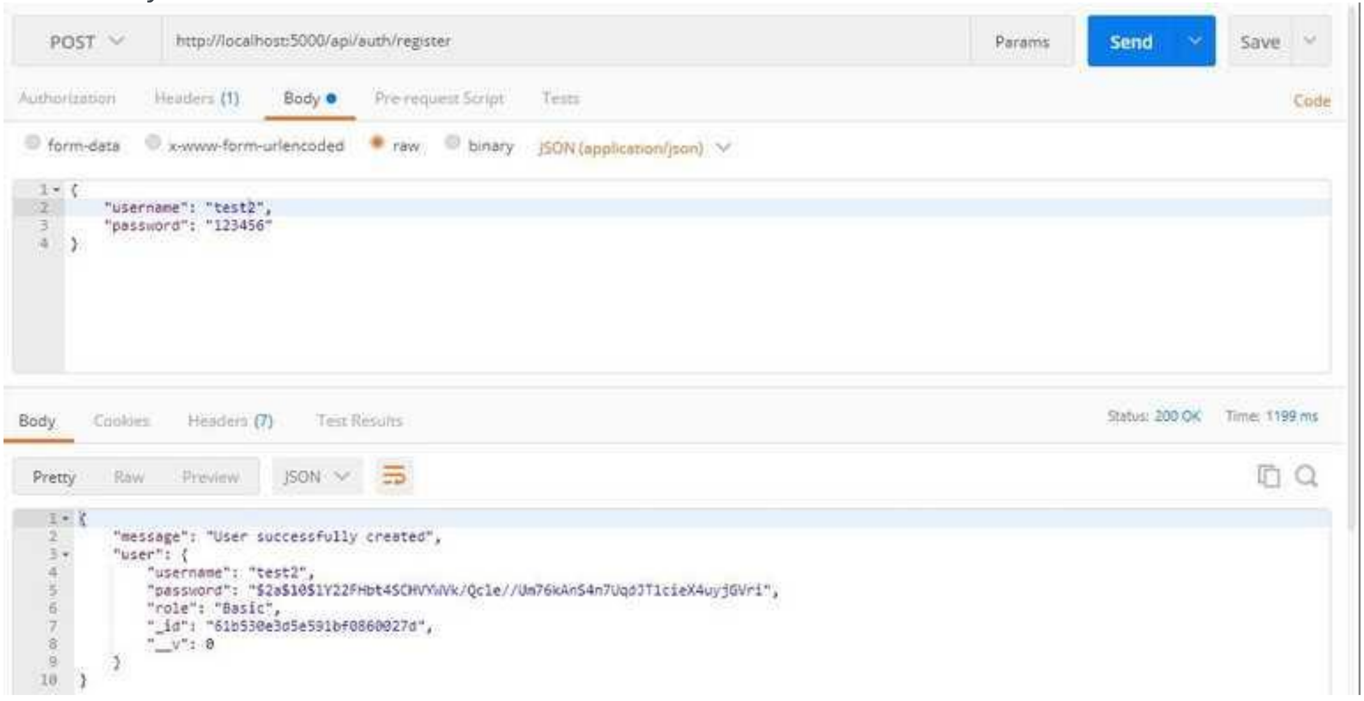
The last step is to import your `route.js` file as middleware in `server.js`:

```
app.use("/api/auth", require("./Auth/route"))
```

The server will use the `router` middleware function if there is a request to `/api/auth`.

Test the Register Route

You'll use Postman to test all the routes.
Open up Postman to send a `POST` request to `http://localhost:5000/api/auth/register` and pass the username and password to the body:



POST http://localhost:5000/api/auth/register

Body: raw, JSON (application/json)

```
{
    "username": "test2",
    "password": "123456"
}
```

Status: 200 OK Time: 1199 ms

```
{
    "message": "User successfully created",
    "user": {
        "username": "test2",
        "password": "$2a$10$1Y22FHbt4SCHVYWVk/Qcle//Um76kAnS4n7UqdJT1cieX4uyjGVri",
        "role": "Basic",
        "_id": "61b530e3d5e591bf0860027d",
        "__v": 0
    }
}
```

Login Function

You've created a function that adds registered users to the database. You have to create another function that will authenticate user credentials and check if the user is registered.

Open the Auth.js file and create the Login function, as follows:

```
// auth.js
exports.login = async (req, res, next) => {
  const { username, password } = req.body
  // Check if username and password is provided
  if (!username || !password) {
    return res.status(400).json({
      message: "Username or Password not present",
    })
  }
}
```

The login function returns status code 400 if the username and password were not provided. You need to find a user with the provided username and password:

```
exports.login = async (req, res, next) => {
  try {
    const user = await User.findOne({ username, password })
    if (!user) {
      res.status(401).json({
        message: "Login not successful",
        error: "User not found",
      })
    } else {
      res.status(200).json({
        message: "Login successful",
        user,
      })
    }
  } catch (error) {
    res.status(400).json({
      message: "An error occurred",
      error: error.message,
    })
  }
}
```

Here, it returns status code 401 when a user isn't found and 200 when a user is found. The code snippet wrapped all this in a tryCatch block to detect and output errors, if any.

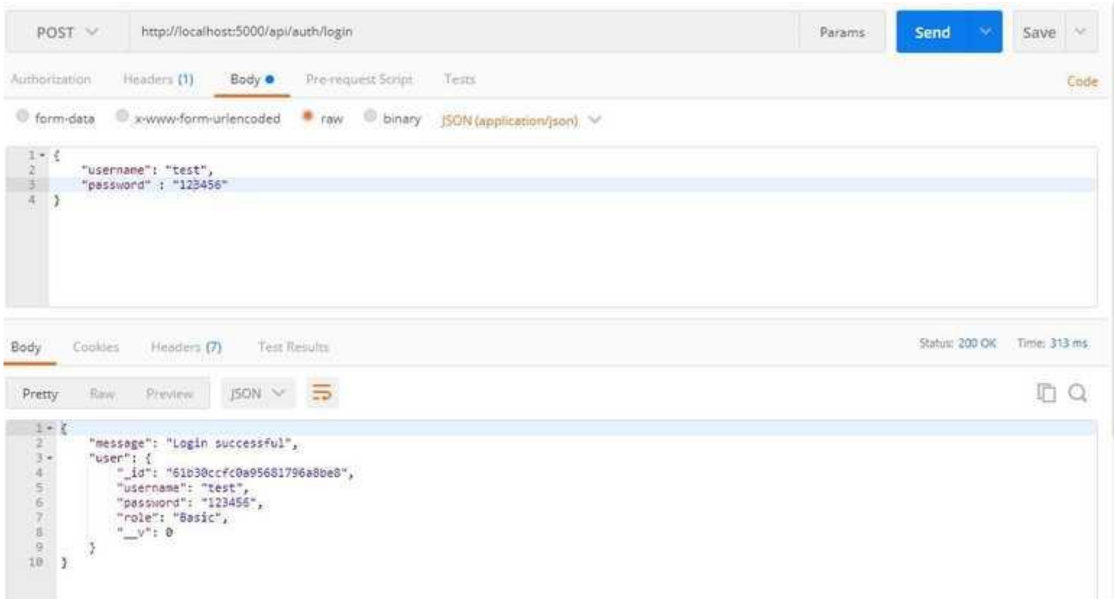
Set Up Login Route

To set up the login route, import the login function into your route.js:

```
const express = require("express");
const router = express.Router();
const { register, login } = require("./auth");
...
router.route("/login").post(login);
module.exports = router;
```

Test the Login Route

Make a POST request at http://localhost:5000/api/auth/login and pass a valid username and password to the body:



http://localhost:5000/api/auth/login

```
{
    "username": "test",
    "password" : "123456"
}
```

Status: 200 OK Time: 313 ms

```
{
    "message": "Login successful",
    "user": {
        "_id": "61b30ccfc0a95681796a8be8",
        "username": "test",
        "password": "123456",
        "role": "Basic",
        "__v": 0
    }
}
```

Update Function

This function will be responsibile for updating the role of a basic user to an admin user. Open the `auth.js` file and create the `update` function, as follows:

```
//auth.js
exports.update = async (req, res, next) => {
  const { role, id } = req.body
  // Verifying if role and id is presnt
  if (role && id) {
    // Verifying if the value of role is admin
    if (role === "admin") {
      await User.findById(id)
    } else {
      res.status(400).json({
        message: "Role is not admin",
      })
    }
  } else {
    res.status(400).json({ message: "Role or Id not present" })
  }
}
```

The first `if` statement verifies if `role` and `id` are present in the request body.
The second `if` statement checks if the value of `role` is admin. You should do this to avoid having over two roles.
After finding a user with that ID, you'll create a third `if` block that will check for the role of the user:

```
exports.update = async (req, res, next) => {
  const { role, id } = req.body;
  // First - Verifying if role and id is presnt
  if (role && id) {
    // Second - Verifying if the value of role is admin
    if (role === "admin") {
      // Finds the user with the id
      await User.findById(id)
        .then((user) => {
          // Third - Verifies the user is not an admin
          if (user.role !== "admin") {
            user.role = role;
            user.save((err) => {
              //Monogodb error checker
              if (err) {
                res
                  .status("400")
                  .json({ message: "An error occurred", error: err.message
});
                process.exit(1);
              }
              res.status("201").json({ message: "Update successful", user
});
            });
          } else {
            res.status(400).json({ message: "User is already an Admin" });
          }
        })
        .catch((error) => {
          res
            .status(400)
            .json({ message: "An error occurred", error: error.message });
        });

       ...
```

The third `if` block prevents assigning an admin role to an admin user, while the last `if` block checks if an error occurred when saving the role in the database.
The numerous `if` statements might be a little bit tricky but understandable. Please read the comments in the above code block for better understanding.

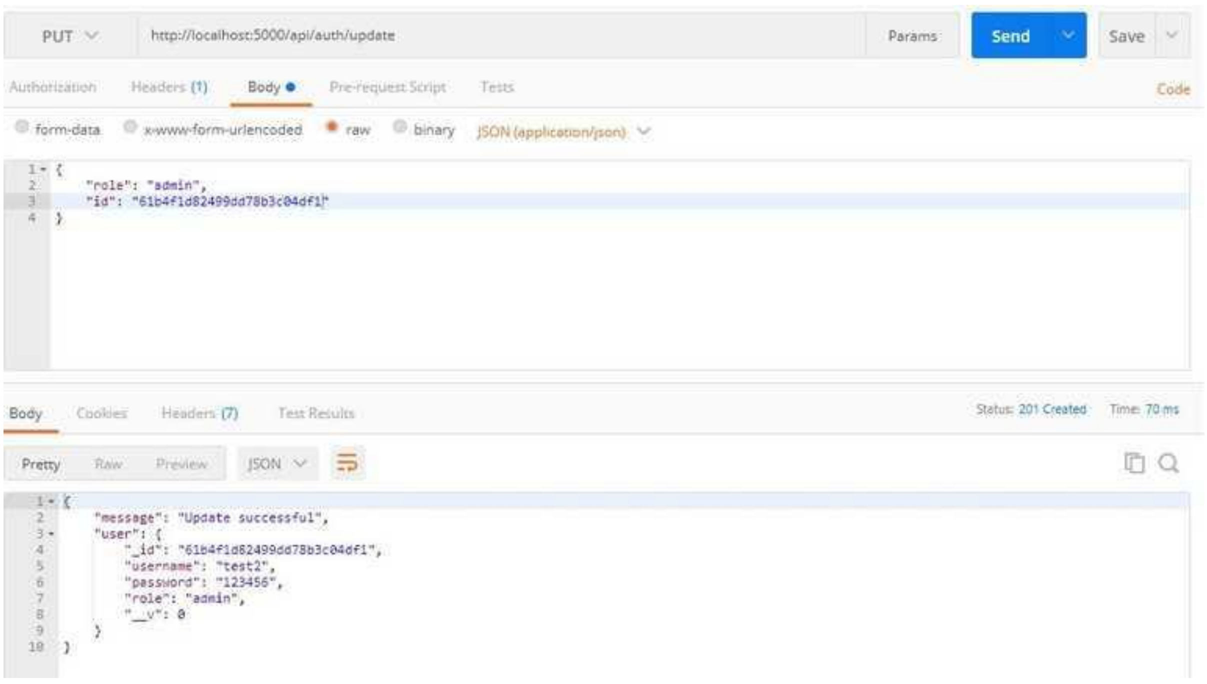
Set Up Update Route

Import the `update` function in your `route.js`, as follows:

```
const { register, login, update } = require("./auth");
...
router.route("/update").put(update);
```

Testing the Update Route

Send a `put` request to `http://localhost:5000/api/auth/update`:



http://localhost:5000/api/auth/update

```
{
    "role": "admin",
    "id": "61b4f1d82499dd78b3c04df1"
}
```

Status: 201 Created Time: 70 ms

```
{
    "message": "Update successful",
    "user": {
        "_id": "61b4f1d82499dd78b3c04df1",
        "username": "test2",
        "password": "123456",
        "role": "admin",
        "__v": 0
    }
}
```

Delete Function

The `deleteUser` function will remove a specific user from the database. Let's create this function in our `auth.js` file:

```
exports.deleteUser = async (req, res, next) => {
  const { id } = req.body
  await User.findById(id)
    .then(user => user.remove())
    .then(user =>
      res.status(201).json({ message: "User successfully deleted", user })
    )
    .catch(error =>
      res
        .status(400)
        .json({ message: "An error occurred", error: error.message })
    )
}
```

You remove the user based on the `id` you get from `req.body`.

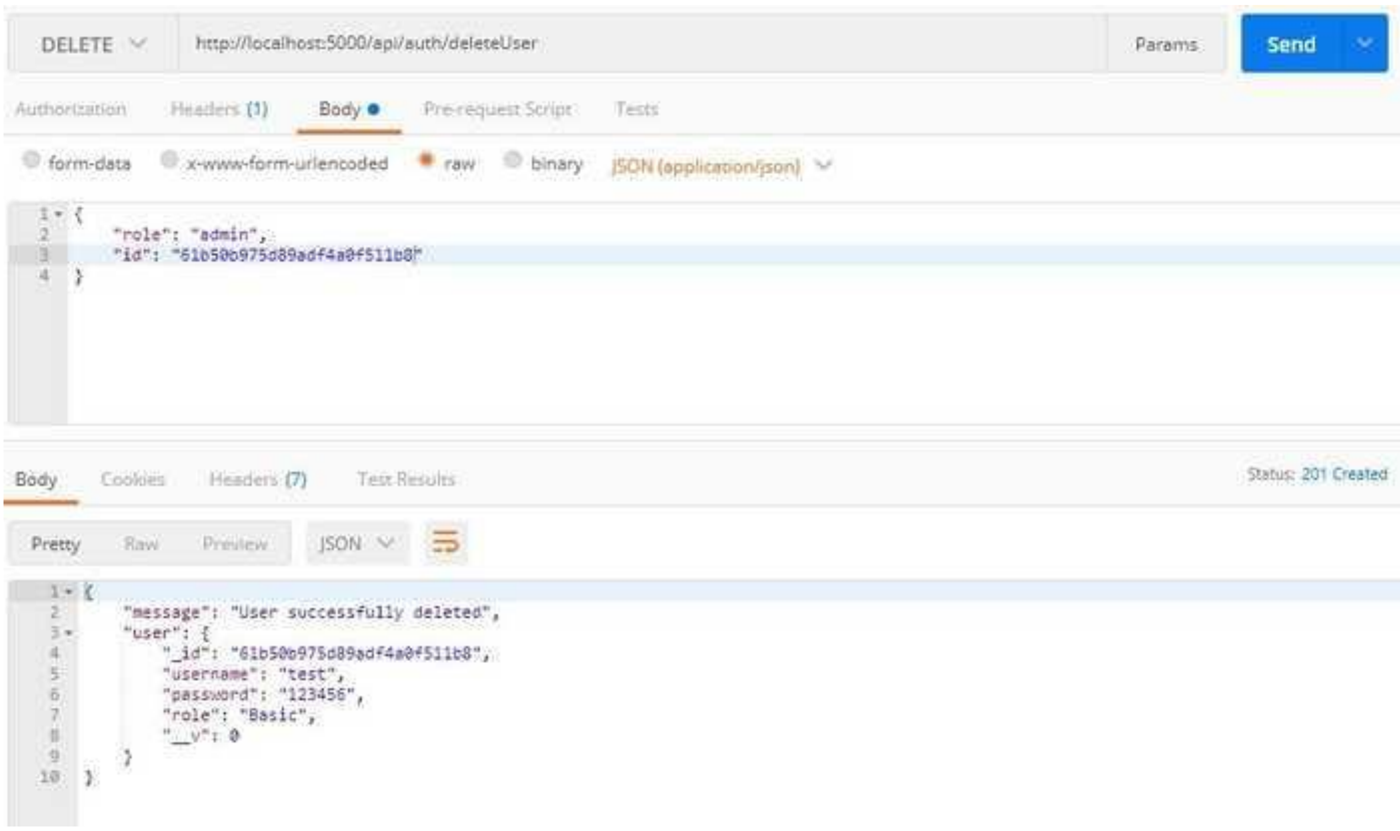
Set up the `deleteUser` Route

Open your `route.js` file to create a `delete` request to `/deleteUser`, using the `deleteUser` as its function:

```
const { register, login, update, deleteUser } = require("./auth");
...
router.route("/deleteUser").delete(deleteUser);
```

Test the `deleteUser` Route

Send a `delete` request to `http://localhost:5000/api/auth/deleteUser` by passing a valid `id` to the body:



```
{
    "role": "admin",
    "id": "61b50b975d89adf4a0f511b8"
}
```

```
{
    "message": "User successfully deleted",
    "user": {
        "_id": "61b50b975d89adf4a0f511b8",
        "username": "test",
        "password": "123456",
        "role": "Basic",
        "__v": 0
    }
}
```

Hash User Passwords

Saving user passwords in the database in plain text format is reckless. It is preferable to hash your password before storing it.

For instance, it will be tough to decipher the passwords in your database if they are leaked. Hashing passwords is a cautious and reliable practice.

Let's use `bcryptjs` to hash your user passwords.
Lets install `bcryptjs`:

```
npm i bcryptjs
```

After installing `bcryptjs`, import it into your `auth.js`

```
const bcrypt = require("bcryptjs")
```

Refactor Register Function

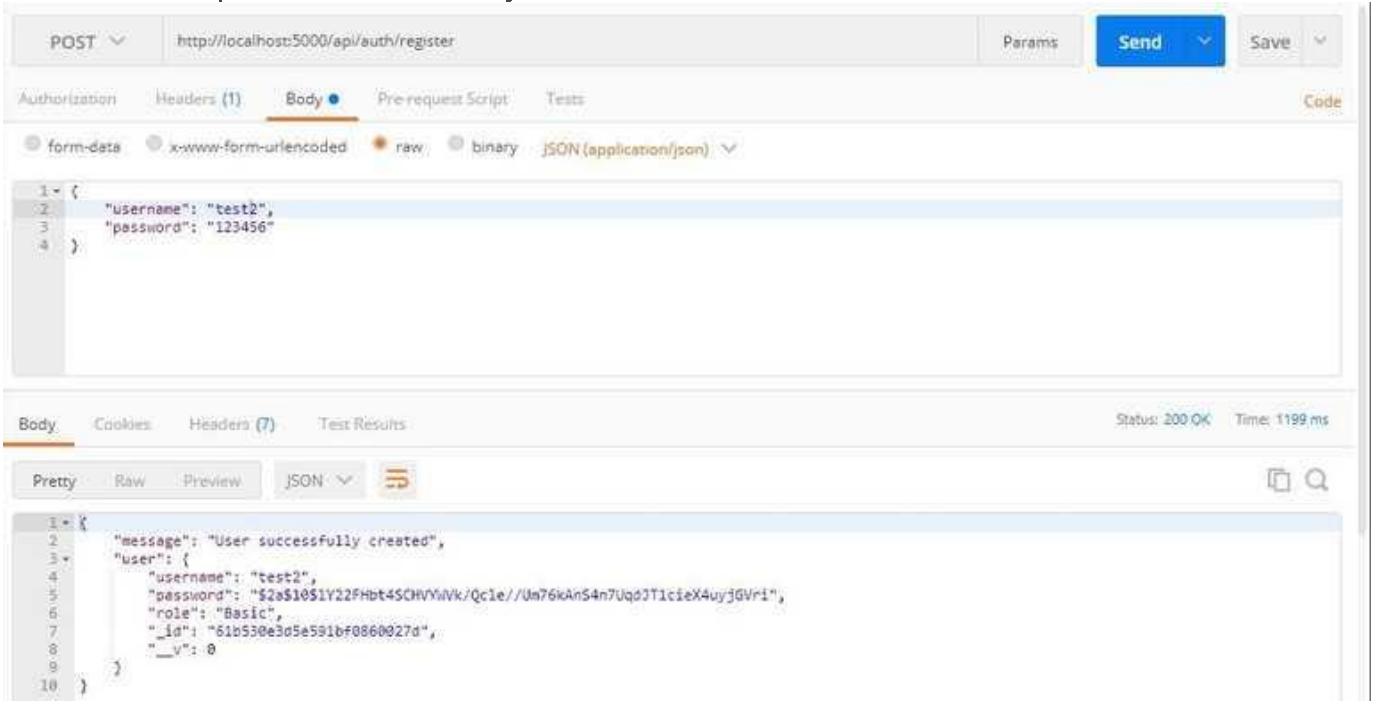
Instead of sending a plain text format to your database, lets hash the password using `bcrypt`:

```
exports.register = async (req, res, next) => {
  const { username, password } = req.body;

  ...
  bcrypt.hash(password, 10).then(async (hash) => {
    await User.create({
      username,
      password: hash,
    })
      .then((user) =>
        res.status(200).json({
          message: "User successfully created",
           user,
        })
      )
      .catch((error) =>
        res.status(400).json({
          message: "User not successful created",
          error: error.message,
        })
      );
  });
};
```

`bcrypt` takes in your password as the first argument and the number of times it will hash the password as the second argument. Passing a large number will take `bcrypt` a long time to hash the password, so give a moderate number like 10. `bcrypt` will return a promise with the hashed password; then, send that hashed password to the database.

Test the Register Function

Send a `POST` request to `http://localhost:5000/api/auth/register` and pass the username and password to the body:



```
{
    "username": "test2",
    "password": "123456"
}
```

```
{
    "message": "User successfully created",
    "user": {
        "username": "test2",
        "password": "$2a$10$1Y22FHbt4SCHVYWVk/Qcle//Um76kAnS4n7UqdJT1cieX4uyjGVri",
        "role": "Basic",
        "_id": "61b530e3d5e591bf0860027d",
        "__v": 0
    }
}
```

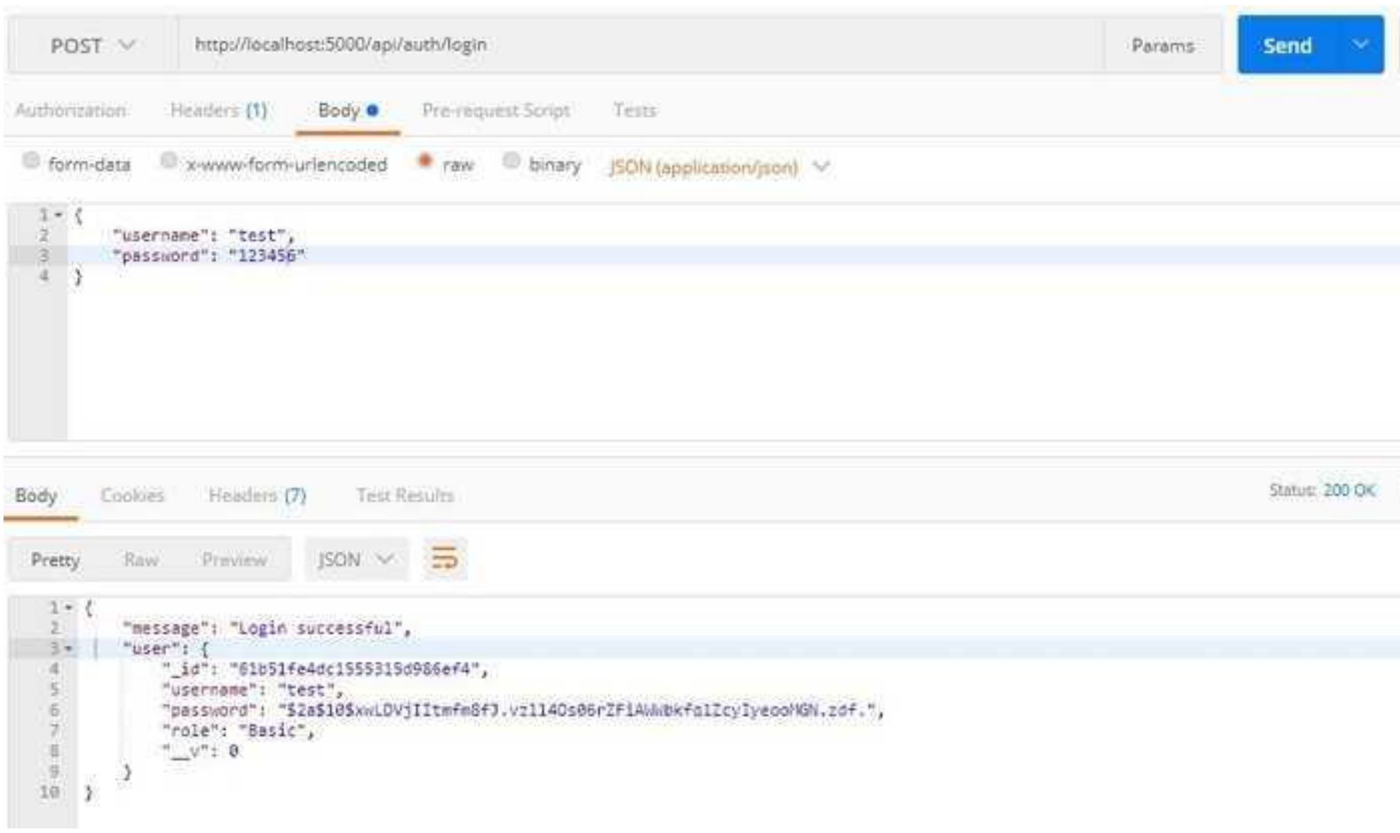
Refactor the Login Function

```
exports.login = async (req, res, next) => {
  const { username, password } = req.body
  // Check if username and password is provided
  if (!username || !password) {
    return res.status(400).json({
      message: "Username or Password not present",
    })
  }
  try {
    const user = await User.findOne({ username })
    if (!user) {
      res.status(400).json({
        message: "Login not successful",
        error: "User not found",
      })
    } else {
      // comparing given password with hashed password
      bcrypt.compare(password, user.password).then(function (result) {
        result
          ? res.status(200).json({
              message: "Login successful",
              user,
            })
          : res.status(400).json({ message: "Login not succesful" })
      })
    }
  } catch (error) {
    res.status(400).json({
      message: "An error occurred",
      error: error.message,
    })
  }
}
```

`bcrypt.compare` checks if the given password and the hashed password in the database are the same.

Test the Login Function

Send a `POST` request to `http://localhost:5000/api/auth/login` and pass a valid username and password to the body:



http://localhost:5000/api/auth/login

```
{
    "username": "test",
    "password": "123456"
}
```

```
{
    "message": "Login successful",
    "user": {
        "_id": "61b51fe4dc1555315d986ef4",
        "username": "test",
        "password": "$2a$10$xwLDVjIItmfm8fJ.vz114Os06rZF1AWWbkfalZcyIyeooMGN.zdf.",
        "role": "Basic",
        "__v": 0
    }
}
```

Authenticate Users with JSON Web Token (JWT)

JSON Web Token helps shield a route from an unauthenticated user. Using JWT in your application will prevent unauthenticated users from accessing your users' home page and prevent unauthorized users from accessing your admin page.

JWT creates a token, sends it to the client, and then the client uses the token for making requests. It also helps verify that you're a valid user making those requests.

You've to install JWT before using it in your application:

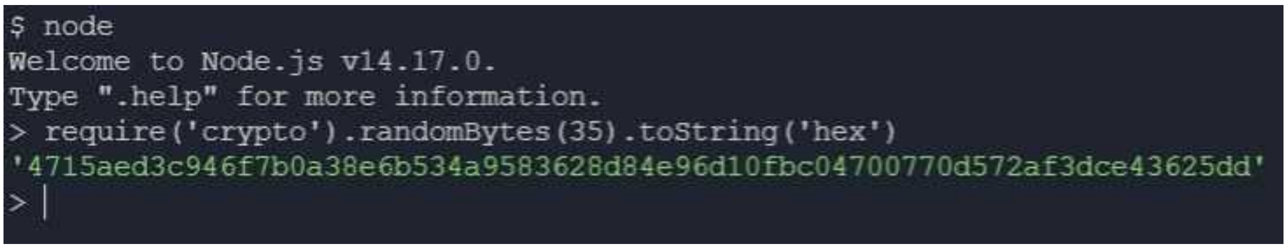
```
npm i jsonwebtoken
```

Refactor the Register Function

When a user registers, you'll send a token to the client using JWT as a cookie. To create this token, you need to set a secret string. You'll use the node's in-built package called `crypto` to create random strings:

```
node
require("crypto").randomBytes(35).toString("hex")
```

Output:



```
$ node
Welcome to Node.js v14.17.0.
Type ".help" for more information.
> require('crypto').randomBytes(35).toString('hex')
'4715aed3c946f7b0a38e6b534a9583628d84e96d10fbc04700770d572af3dce43625dd'
> |
```

Storing this secret string in an environment variable is a safe practice. If this secret string is leaked, unauthenticated users can create fake tokens to access the route.

Store your secret string in a variable:

```
const jwtSecret =
  "4715aed3c946f7b0a38e6b534a9583628d84e96d10fbc04700770d572af3dce43625dd"
```

Once you've created your `jwtSecret`, import `jsonwebtoken` as the token in the `register` function:

```
...
const jwt = require('jsonwebtoken')
const jwtSecret =
'4715aed3c946f7b0a38e6b534a9583628d84e96d10fbc04700770d572af3dce43625dd'
exports.register = async (req, res, next) => {
  const { username, password } = req.body;

  ...

  bcrypt.hash(password, 10).then(async (hash) => {
    await User.create({
      username,
      password: hash,
    })
      .then((user) => {
        const maxAge = 3 * 60 * 60;
        const token = jwt.sign(
          { id: user._id, username, role: user.role },
          jwtSecret,
          {
            expiresIn: maxAge, // 3hrs in sec
          }
        );
        res.cookie("jwt", token, {
          httpOnly: true,
          maxAge: maxAge * 1000, // 3hrs in ms
        });
        res.status(201).json({
          message: "User successfully created",
          user: user._id,
        });
      })
      .catch((error) =>
          res.status(400).json({
            message: "User not successful created",
            error: error.message,
          })
        );
    });
};
```

The code snippet created the token using JWT's `sign` function. This function takes in three parameters:

- the payload is the first parament that you'll pass to the function. This payload holds data concerning the user, and this data should not contain sensitive information like passwords;
- you passed your `jwtSecret` as the second parameter; and,
- how long the token will last as the third parameter.

After passing all these arguments, JWT will generate a token. After the token is generated, send it as a cookie to the client.

Refactor the Login Function

Also, generate a token for logged in users:

```
exports.login = async (req, res, next) => {

    ...

      bcrypt.compare(password, user.password).then(function (result) {
        if (result) {
          const maxAge = 3 * 60 * 60;
          const token = jwt.sign(
            { id: user._id, username, role: user.role },
            jwtSecret,
            {
              expiresIn: maxAge, // 3hrs in sec
            }
          );
          res.cookie("jwt", token, {
            httpOnly: true,
            maxAge: maxAge * 1000, // 3hrs in ms
          });
          res.status(201).json({
            message: "User successfully Logged in",
            user: user._id,
          });
        } else {
          res.status(400).json({ message: "Login not succesful" });
        }
      });
    }
  } catch (error) {
    res.status(400).json({
      message: "An error occurred",
      error: error.message,
    });
  }
};
```

Protect the Routes

To prevent unauthenticated users from accessing the private route, take the token from the cookie, verify the token, and redirect users based on role.

You'll get the token from the client using a node package called `cookie-parser`. Let's install the package before using it:

```
npm i cookie-parser
```

After installing it, import it into your `server.js` file and use it as a *middleware*:

```
const cookieParser = require("cookie-parser");
...
app.use(cookieParser());
```

You'll create your middleware that verifies the token and grants access to your private route.

Let's create a new folder in the project's folder named `middleware` and create a file called `auth.js`.

Admin Authentication

Open the `auth.js` file and create the middleware:

```
const jwt = require("jsonwebtoken")
const jwtSecret =
  "4715aed3c946f7b0a38e6b534a9583628d84e96d10fbc04700770d572af3dce43625dd"
exports.adminAuth = (req, res, next) => {
  const token = req.cookies.jwt
  if (token) {
    jwt.verify(token, jwtSecret, (err, decodedToken) => {
      if (err) {
        return res.status(401).json({ message: "Not authorized" })
      } else {
        if (decodedToken.role !== "admin") {
          return res.status(401).json({ message: "Not authorized" })
        } else {
          next()
        }
      }
    })
  } else {
    return res
      .status(401)
      .json({ message: "Not authorized, token not available" })
  }
}
```

The code snippet requests a token from the client, checks if a token is available, and verifies that token.

JWT verifies your token with your `jwtSecret` and returns a callback function. This function returns status code `401` if the token fails the authentication test.
When you've created the token, you passed a payload that contained the user's credentials. You'll get the role from the credentials and check if the user's role is admin. If the user is not an admin, you return status code `401`, but you'll call the `next` function if the user is an admin.

Basic User Authentication

You'll also authenticate basic users before granting them access to the users route. Let's create another middleware in your `auth.js` file that will authenticate basic users:

```
exports.userAuth = (req, res, next) => {
  const token = req.cookies.jwt
  if (token) {
    jwt.verify(token, jwtSecret, (err, decodedToken) => {
      if (err) {
        return res.status(401).json({ message: "Not authorized" })
      } else {
        if (decodedToken.role !== "Basic") {
          return res.status(401).json({ message: "Not authorized" })
        } else {
          next()
        }
      }
    })
  } else {
    return res
      .status(401)
      .json({ message: "Not authorized, token not available" })
  }
}
```

Protect the Routes

You'll have two routes: one for the user and the other for the admin. Let's import this middleware into your `server.js` file and protect your routes:

```
const { adminAuth, userAuth } = require("./middleware/auth.js");
...
app.get("/admin", adminAuth, (req, res) => res.send("Admin Route"));
app.get("/basic", userAuth, (req, res) => res.send("User Route"));
```

Updating user roles and deleting users should be done by an Admin, so you need to import this `auth.js` middleware into your `route.js` file to protect the `update` and `delete` routes.

`route.js`:

```
const { adminAuth } = require("../middleware/auth")
router.route("/update").put(adminAuth, update)
router.route("/deleteUser").delete(adminAuth, deleteUser)
```

Populate the Database with Admin User

You need to create an admin user in your database. Open up your terminal, and let's run some MongoDB methods:

```
mongo
```

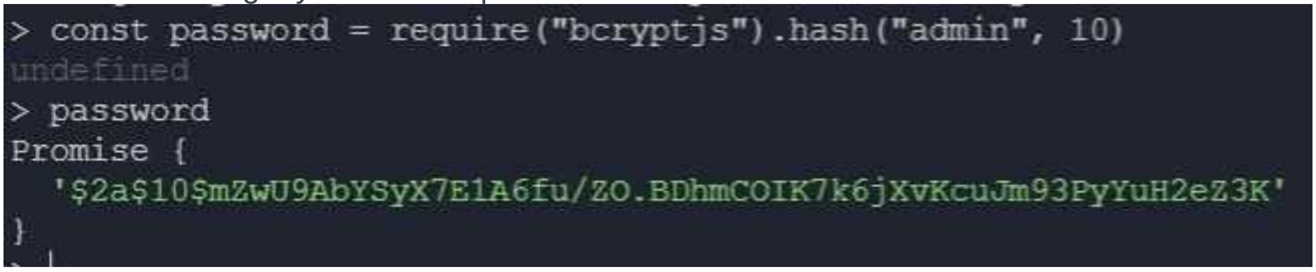
After mongo is started, you need to use the `role_auth` database:

```
use role_auth
```

Before adding your admin user to the database, you need to hash the password using `bcrypt` in the `node` terminal. Open node terminal in your project's directory:

```
const password = require("bcryptjs").hash("admin", 10)
password
```

After you've created the constant `password`, you need to enter `the password` in the node terminal to get your hashed password.



```
> const password = require("bcryptjs").hash("admin", 10)
undefined
> password
Promise {
  '$2a$10$mZwU9AbYSyX7E1A6fu/ZO.BDhmCOIK7k6jXvKcuJm93PyYuH2eZ3K'
}
```

You'll use the hashed password to create your admin:

```
db.users.insert({
  username: "admin",
  password: "$2a$10$mZwU9AbYSyX7E1A6fu/ZO.BDhmCOIK7k6jXvKcuJm93PyYuH2eZ3K",
  role: "admin",
})
```

To check if it was successfully created, run `db.users.find().pretty()` — this will output all users in the database.

Create the Login Form Using EJS

You'll use Embedded JavaScript (EJS) to create a front-end for your application.

Install the `ejs` package:

```
    npm i ejs
```

After you've installed `ejs`, you need to set `ejs` as your default *view engine* in your `server.js` file:

```
app.set("view engine", "ejs")
```

Render Embedded JavaScript

When making a `GET` request to specific routes, you'll render an `ejs` file:

```
app.get("/", (req, res) => res.render("home"))
app.get("/register", (req, res) => res.render("register"))
app.get("/login", (req, res) => res.render("login"))
app.get("/admin", adminAuth, (req, res) => res.render("admin"))
app.get("/basic", userAuth, (req, res) => res.render("user"))
```

Create EJS Files

By default, your application will look into the `views` folder when rendering an `ejs` file. You need to create the `views` folder in your project's folder and add your `ejs` files to it

Create a Home Page

Your home page will contain the links to `/login` and `/register` ejs file. Open up `home.ejs` and add these links:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Home page</title>
  </head>
  <body>
    <h1>Home Page</h1>
    <a href="/register"> Register</a> <br />
    <a href="/login">Login</a>
  </body>
</html>
```

Create a Registration Form

Embedded JavaScript (EJS) supports HTML syntax. You'll create the registration form in `register.ejs` using HTML syntax:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Register Page</title>
  </head>
  <body>
    <h1>Register Page</h1>
    <form>
      <div class="error" style="background-color: red;"></div>
      <br />
      <label for="username">Username</label><br />
      <input type="text" id="username" required /><br />
      <label for="password">Password</label><br />
      <input type="password" id="password" required /><br />
      <input type="submit" value="register" /><br />
    </form>
    <a href="/login">Already registered? Login</a>
  </body>
</html>
```

Add POST Request Functionality

You need to get the username and password that the user entered and pass it to the body when making the `POST` request:

```
...
<script>
  const form = document.querySelector('form')
  const username = document.querySelector('#username')
  const password = document.querySelector('#password')
  const display = document.querySelector('.error')
  form.addEventListener('submit', async (e) => {
     e.preventDefault()
     display.textContent = ''
     try {
       const res = await fetch('/api/auth/register', {
       method: 'POST',
        body: JSON.stringify({ username: username.value, password:
password.value }),
       headers: { 'Content-Type': 'application/json' }
       })
       const data = await res.json()
       if(res.status === 400 || res.status === 401){
        return display.textContent = `${data.message}. ${data.error ?
data.error : ''}`
       }
       data.role === "admin" ? location.assign('/admin') :
location.assign('/basic')
        } catch (err) {
          console.log(err.message)
        }
      })
</script>
</body>
</html>
```

The code snippet uses JavaScript's in-built library called `fetch` to send a POST request to `/api/auth/register`.
After the request has been sent, it stores the response to a constant `res`.
`res.json` will return the JSON you've passed as a response in the API.
When `res.json` returns the data, you store that data in a constant `data`.
If you get an error while making the request, display the error to the user. If an error isn't found, redirect the user based on their role on different routes.

Create a Login Form

Creating your login form and adding functionality to it will be similar to that of your registration. Open `login.ejs` and create this form:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Login Page</title>
  </head>
  <body>
    <h1>Login Page</h1>
    <form>
```

```
      <div class="error" style="background-color: red;"></div>
      <br />
      <label for="username">Username</label><br />
      <input type="text" id="username" required /><br />
      <label for="password">Password</label><br />
      <input type="password" id="password" required /><br />
      <input type="submit" value="login" /><br />
    </form>
    <a href="/register">Don't have an accout? Register</a>
  </body>
</html>
```

Add POST Request Functionality

```
    <script>
      const form = document.querySelector('form')
      const username = document.querySelector('#username')
      const password = document.querySelector('#password')
      const display = document.querySelector('.error')
      form.addEventListener('submit', async (e) => {
        e.preventDefault()
        display.textContent = ''
        try {
          const res = await fetch('/api/auth/login', {
            method: 'POST',
            body: JSON.stringify({ username: username.value, password:
password.value }),
            headers: { 'Content-Type': 'application/json' }
            })
          const data = await res.json()
          if (res.status === 400 || res.status === 401) {
            return display.textContent = `${data.message}. ${data.error ?
data.error : ''}`
          }
          data.role === "admin" ? location.assign('/admin') :
location.assign('/basic')
        } catch (err) {
            console.log(err.message)
          }

        })
      </script>
    </body>
    </html>
```

Add Registered Users to the Route

Once you've redirected users based on role to different routes, display all registered users on that route. You need to send a `GET` request to `/getUsers`.
Open `auth.js` file in `Auth` folder:

```
exports.getUsers = async (req, res, next) => {
  await User.find({})
    .then(users => {
      const userFunction = users.map(user => {
        const container = {}
        container.username = user.username
        container.role = user.role
        return container
      })
      res.status(200).json({ user: userFunction })
    })
    .catch(err =>
      res.status(401).json({ message: "Not successful", error: err.message
})
    )
}
```

The `User.find` method returns an array of users. After mapping through this array, it stores the username and role in the constant `container` and returns the `container`.

Display Registered User in `user` Route

You've rendered `user.ejs` when accessing the `/user` route. Now, you'll display all registered users to that route.

`user.ejs`:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>User page</title>
  </head>
  <body>
    <h1>Users</h1>
    <ul></ul>
    <script>
      const ul = document.querySelector("ul")
      const getUsers = async () => {
        const res = await fetch("/api/auth/getUsers")
        const data = await res.json()
        data.user.map(mappedUser => {
          if (mappedUser.username !== "admin") {
            let li = `<li> <b>Username</b> => ${mappedUser.username} <br>
<b>Role</b> => ${mappedUser.role} </li>`
            ul.innerHTML += li
          } else {
            return null
          }
        })
      }
      getUsers()
    </script>
  </body>
</html>
```

Add Update and Delete Function to the Admin Route

You'll also display registered users to the admin route but add `update` and `delete` functionality to the route:

admin.ejs:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Admin page</title>
  </head>
  <body>
    <div class="display" style="background-color: red;"></div>
    <h1>Users</h1>
    <ul></ul>
    <script>
      const ul = document.querySelector("ul")
      const display = document.querySelector(".display")
      const getUsers = async () => {
        const res = await fetch("/api/auth/getUsers")
        const data = await res.json()
        data.user.map(mappedUser => {
          if (mappedUser.username !== "admin") {
            let li = `<li> <b>Username</b> => ${mappedUser.username} <br>
<b>Role</b> => ${mappedUser.role} </li> <button class="edit">Edit
Role</button> <button class="delete">Delete User</button>`
            ul.innerHTML += li
          } else {
            return null
          }
          const editRole = document.querySelectorAll(".edit")
          const deleteUser = document.querySelector(".delete")
        })
      }
      getUsers()
    </script>
  </body>
</html>
```

Edit a User's Role

You'll create an event listener that will listen for a click on the `Edit Role` button. When the button is clicked, you'll send a `PUT` request to `/api/auth/update`:

```
<script>

    ...

    const editRole = document.querySelectorAll('.edit')
    const deleteUser = document.querySelector('.delete')
  editRole.forEach((button, i) => {
    button.addEventListener('click', async() => {
      display.textContent= ''
      const id = data.user[i+1].id
      const res = await fetch('/api/auth/update', {
      method: 'PUT',
      body: JSON.stringify({ role: 'admin', id}),
      headers: { 'Content-Type': 'application/json' }
      })
      const dataUpdate = await res.json()
      if (res.status === 400 || res.status === 401) {
        document.body.scrollTop = 0
        document.documentElement.scrollTop = 0
        return display.textContent = `${dataUpdate.message}.
${dataUpdate.error ? dataUpdate.error : ''}`
      }
      location.assign('/admin')
      })
    });

      ...
</script>
```

Delete Users

Deleting Users from the database should be the duty of an admin.

`admin.ejs`:

```
<script>

  ...

  const editRole = document.querySelectorAll('.edit')
  const deleteUser = document.querySelector('.delete')
  deleteUser.forEach((button, i)=> {
   button.addEventListener('click', async ()=> {
   display.textContent =''
   const id = data.user[i+1].id
   const res = await fetch('/api/auth/deleteUser', {
     method: 'DELETE',
     body: JSON.stringify({id}),
     headers: {'Content-Type': 'application/json'}
     })
   const dataDelete = await res.json()
   if (res.status === 401){
     document.body.scrollTop = 0
     document.documentElement.scrollTop = 0
     return display.textContent = `${dataUpdate.message}. ${dataUpdate.error
? dataUpdate.error : ''}`
   }
   location.assign('/admin')
    })
  })

    ...
</script>
```

You've created an event listener that listens for a click on the `Delete User` button. When the button is clicked, you'll send a `DELETE` request to `/api/auth/deleteUser`. *Please ensure the admin user is first on the list to avoid populating the database with an admin user again.*

Logout Functionality

To log out users, you need to remove the token from the client and redirect the client to the home page.

You'll create a `GET` request to `/logout` in the `server.js` file:

```
app.get("/logout", (req, res) => {
  res.cookie("jwt", "", { maxAge: "1" })
  res.redirect("/")
})
```

The code snippet replaced the JWT token with an empty string and gave it a lifespan of 1 second.

After creating the `GET` request, add a logout button to the admin's route and user's route:

```
...
<ul></ul>
<button class="logout"><a href="/logout">Log Out</a></button>
...
```

Node.js Authentication with LoginRadius

You can simply replace the many steps discussed above using LoginRadius. In turn, it helps you focus more on developing core application features while letting you quickly implement user signup and login and manager users.
In other words, LoginRadius is a SaaS-based customer identity and access management (CIAM) system with features to manage customer identity, privacy, and access. It is a simple, implementable solution for adding user authentication and authorization to your website.

Basically, LoginRadius handles user registration, login, and authentication. Other features of LoginRadius include:

- **Forms**: LoginRadius can automatically pre-create registration and login forms for you.
- **Authentication and Authorization**: It generates and sends a token to the user when login or signup is successful. Instead of using `JWT`, you can use this token to authenticate users

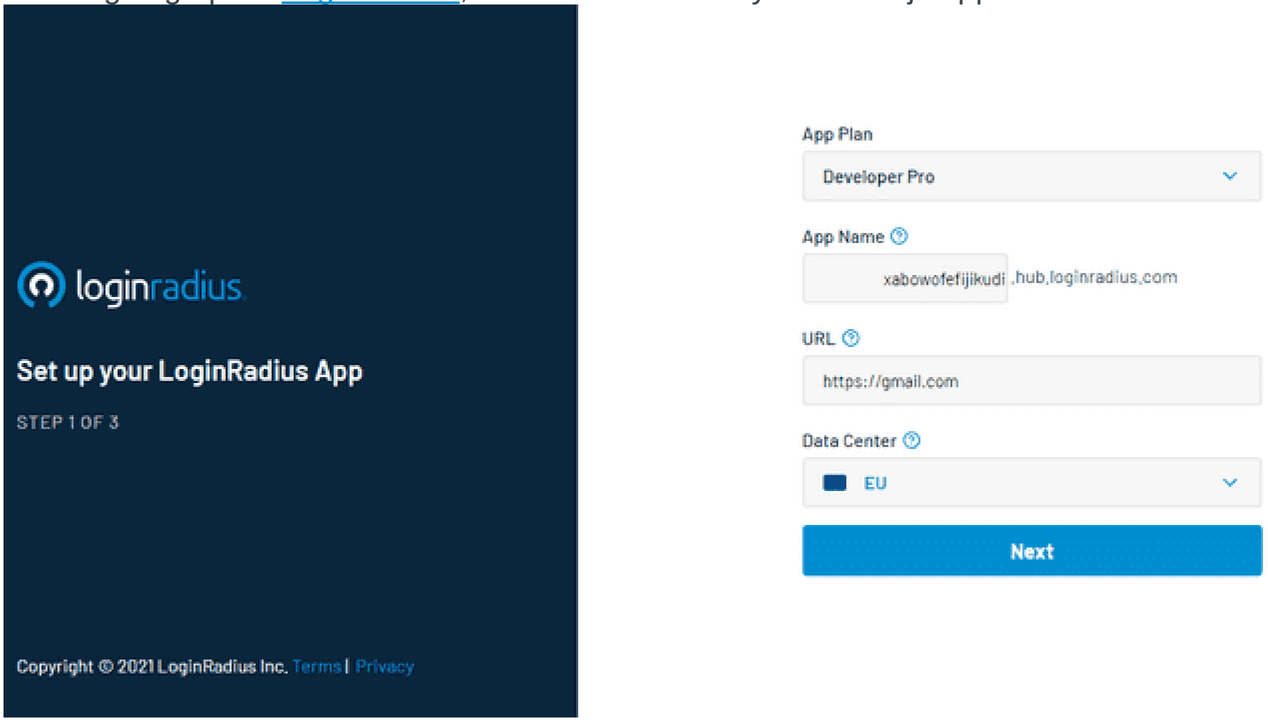
- **Security**: When using LoginRadius, you automatically have access to an admin console where you can control authentication factors, such as email, phone, and multi-factor auth for your Node.js app.

To get started with LoginRadius, you need to create an account with either the free plan or the Developer plan, customize your registration and login forms, and start managing your users.

How to Authenticate Your Node.js App with LoginRadius

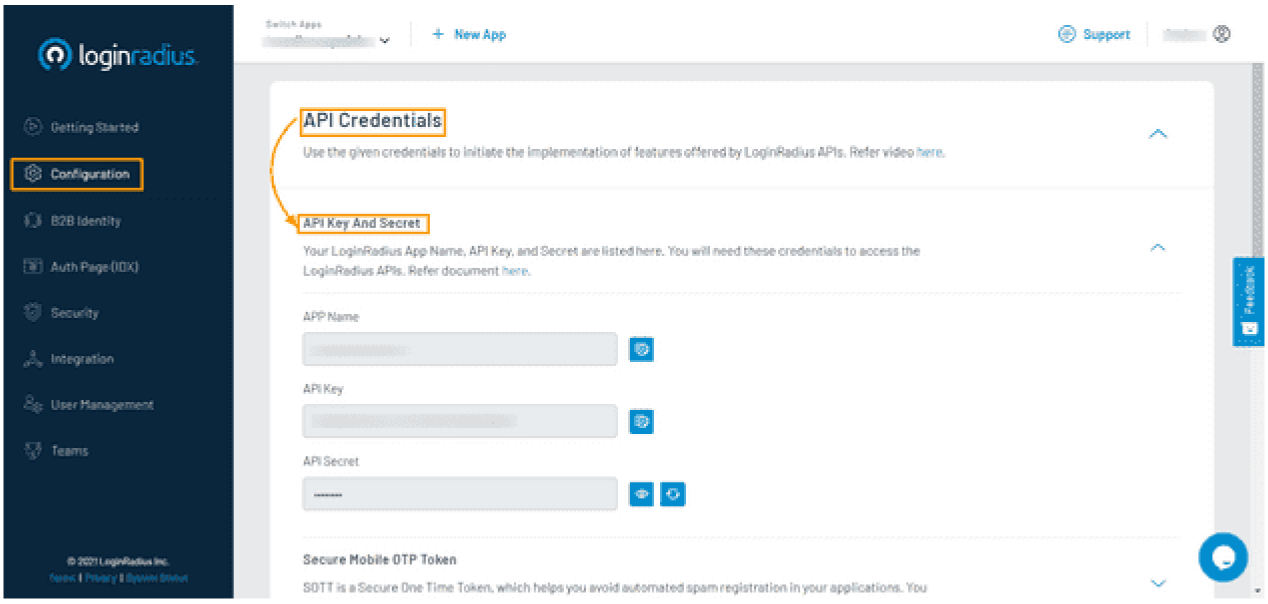
This section briefly covers how authentication works with LoginRadius.

After signing up for LoginRadius, choose a name for your Node.js app.


loginradius
Set up your LoginRadius App
STEP 1 OF 3
Copyright © 2021 LoginRadius Inc. Terms | Privacy
Next

App Plan: Developer Pro
App Name: xabowofefijikudi .hub.loginradius.com
URL: https://gmail.com
Data Center: EU

After completing your LoginRadius signup process, you can get your **App Name**, **API Key**, and **API Secret** from the `configuration` link on the sidebar. With these configurations, you can easily link the server-side of our application to LoginRadius.



API Credentials

Use the given credentials to initiate the implementation of features offered by LoginRadius APIs. Refer video here.

API Key And Secret

Your LoginRadius App Name, API Key, and Secret are listed here. You will need these credentials to access the LoginRadius APIs. Refer document here.

APP Name

API Key

API Secret

Secure Mobile OTP Token

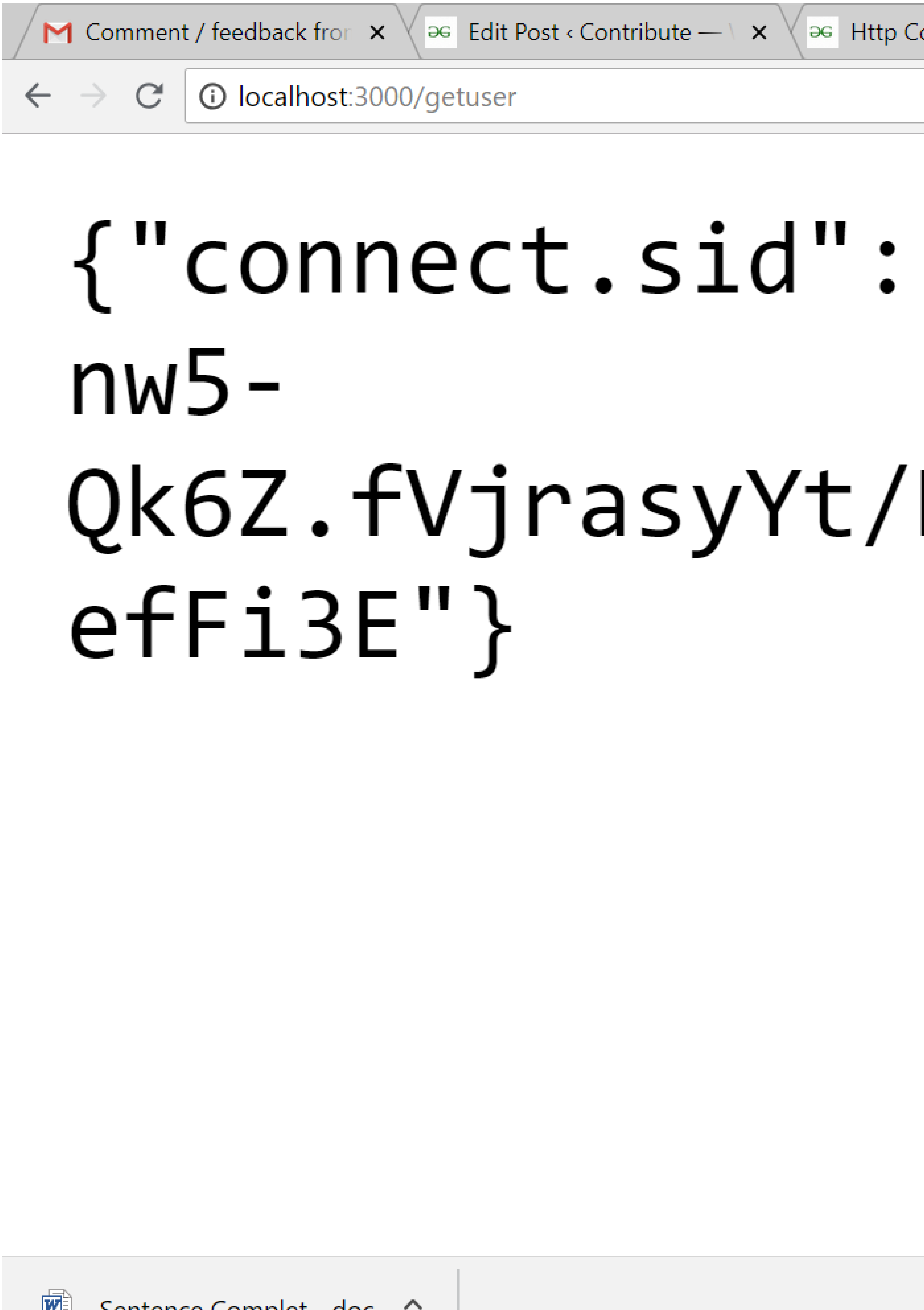
SOTT is a Secure One Time Token, which helps you avoid automated spam registration in your applications. You

LoginRadius automatically generates a link that will be used to authenticate users. This link contains the name of your LoginRadius application and a `URL` that authenticated users will be redirected to:

```
https://<LoginRadius-APP-
Name>.hub.loginradius.com/auth.aspx?action=login&return_url=<Return-URL>
```

An instance of the link is given below:

```
https://noderoleauth.hub.loginradius.com/auth.aspx?action=login&return_url=h
ttp://localhost:5000
```



```
{"connect.sid":
nw5-
Qk6Z.fVjrasyYt/
efFi3E"}
```

This is about basic use of HTTP cookies using cookie-parser middleware. Cookies can be used in many ways like maintaining sessions and providing each user a different view of the website based on their previous transactions on the website.