UNIT III Advanced NODE JS AND Database

UNIT III ADVANCED NODE JS AND DATABASE

Introduction to NoSQL databases – MongoDB system overview - Basic querying with MongoDB shell – Request body parsing in Express – NodeJS MongoDB connection – Adding and retrieving data to MongoDB from NodeJS – Handling SQL databases from NodeJS – Handling Cookies in NodeJS – Handling User Authentication with node js

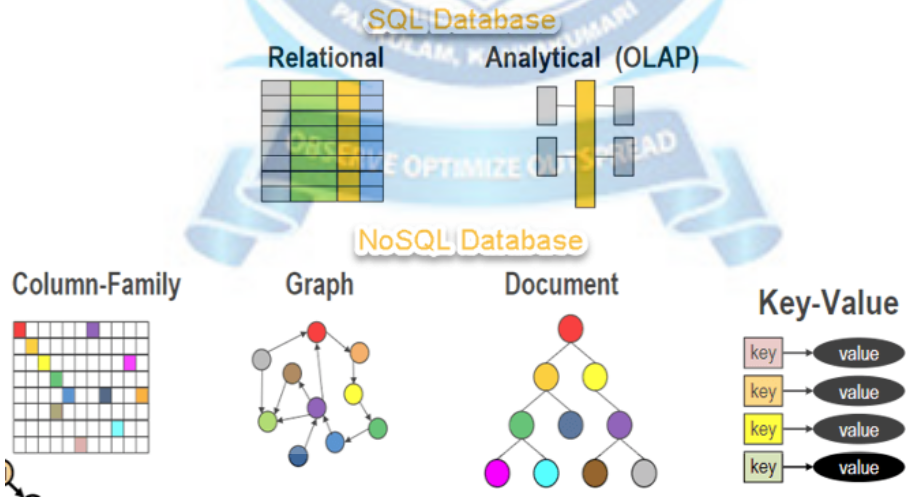
3.1 Introduction to NoSQL databases

What is NoSQL?

NoSQL Database is a non-relational Data Management System, that does not require a fixed schema. It avoids joins, and is easy to scale. The major purpose of using a NoSQL database is for distributed data stores with humongous data storage needs. NoSQL is used for Big data and real-time web apps. For example, companies like Twitter, Facebook and Google collect terabytes of user data every single day.

NoSQL database stands for "Not Only SQL" or "Not SQL." Though a better term would be "NoREL", NoSQL caught on. Carl Strozz introduced the NoSQL concept in 1998.

Traditional RDBMS uses SQL syntax to store and retrieve data for further insights. Instead, a NoSQL database system encompasses a wide range of database technologies that can store structured, semi-structured, unstructured and polymorphic data. Let's understand about NoSQL with a diagram in this NoSQL database tutorial:



Why NoSQL?

The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data. The system response time becomes slow when you use RDBMS for massive volumes of data.

To resolve this problem, we could “scale up” our systems by upgrading our existing hardware. This process is expensive.

Brief History of NoSQL Databases

- 1998- Carlo Strozzi use the term NoSQL for his lightweight, open-source relational database
- 2000- Graph database Neo4j is launched
- 2004- Google BigTable is launched
- 2005- CouchDB is launched
- 2007- The research paper on Amazon Dynamo is released
- 2008- Facebooks open sources the Cassandra project
- 2009- The term NoSQL was reintroduced

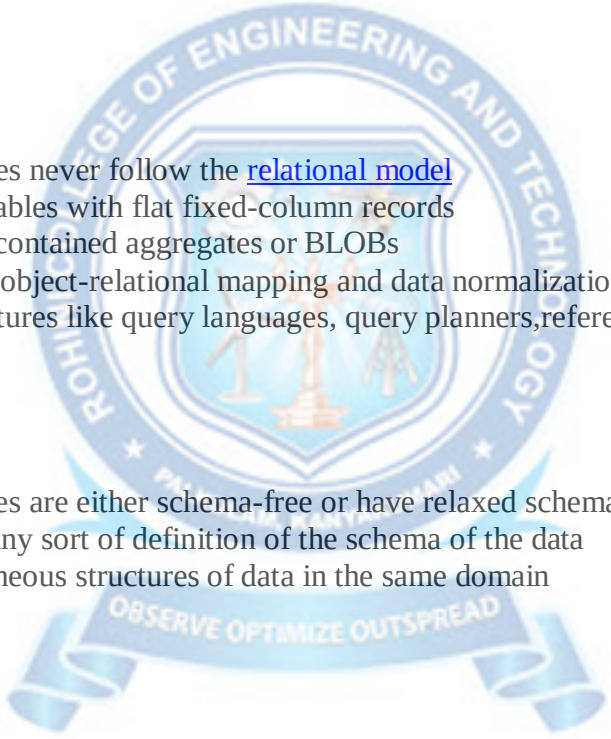
Features of NoSQL

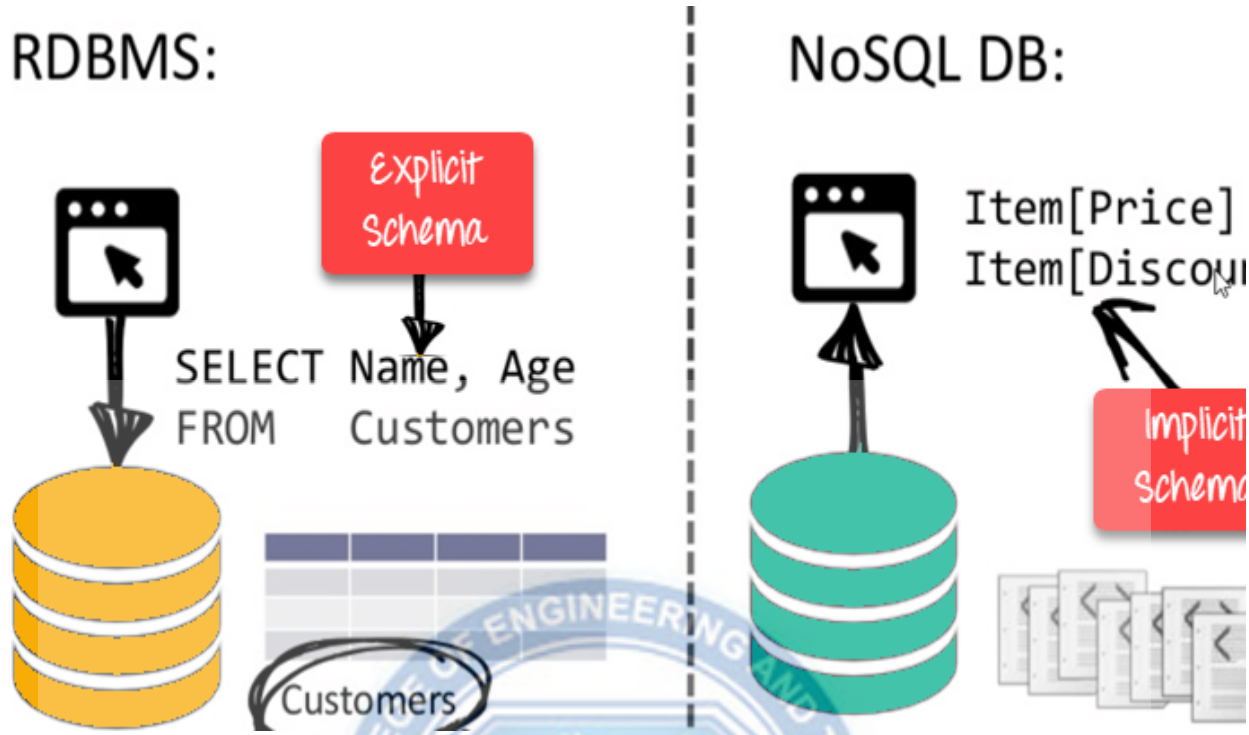
Non-relational

- NoSQL databases never follow the relational model
- Never provide tables with flat fixed-column records
- Work with self-contained aggregates or BLOBs
- Doesn’t require object-relational mapping and data normalization
- No complex features like query languages, query planners,referential integrity joins, ACID

Schema-free

- NoSQL databases are either schema-free or have relaxed schemas
- Do not require any sort of definition of the schema of the data
- Offers heterogeneous structures of data in the same domain





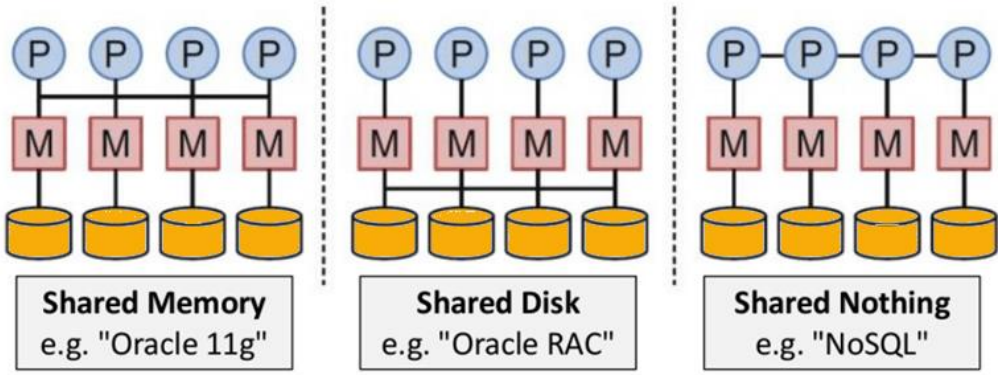
NoSQL is Schema-Free

Simple API

- Offers easy to use interfaces for storage and querying data provided
- APIs allow low-level data manipulation & selection methods
- Text-based protocols mostly used with HTTP REST with JSON
- Mostly used no standard based NoSQL query language
- Web-enabled databases running as internet-facing services

Distributed

- Multiple NoSQL databases can be executed in a distributed fashion
- Offers auto-scaling and fail-over capabilities
- Often ACID concept can be sacrificed for scalability and throughput
- Mostly no synchronous replication between distributed nodes Asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication
- Only providing eventual consistency
- Shared Nothing Architecture. This enables less coordination and higher distribution.

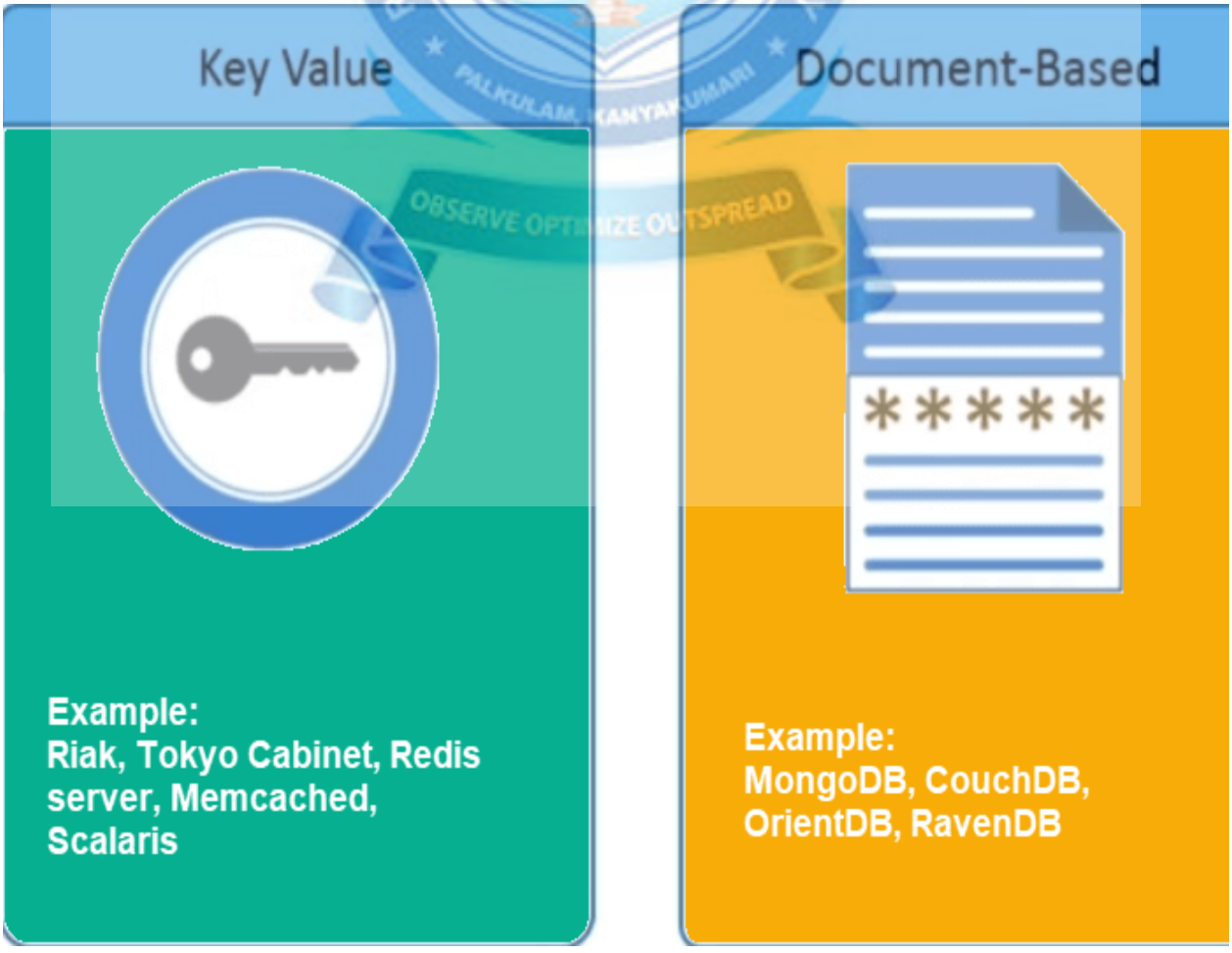


NoSQL is Shared Nothing.

Types of NoSQL Databases

NoSQL Databases are mainly categorized into four types: Key-value pair, Column-oriented, Graph-based and Document-oriented. Every category has its unique attributes and limitations. None of the above-specified database is better to solve all the problems. Users should select the database based on their product needs.
Types of NoSQL Databases:

- Key-value Pair Based
- Column-oriented Graph
- Graphs based
- Document-oriented

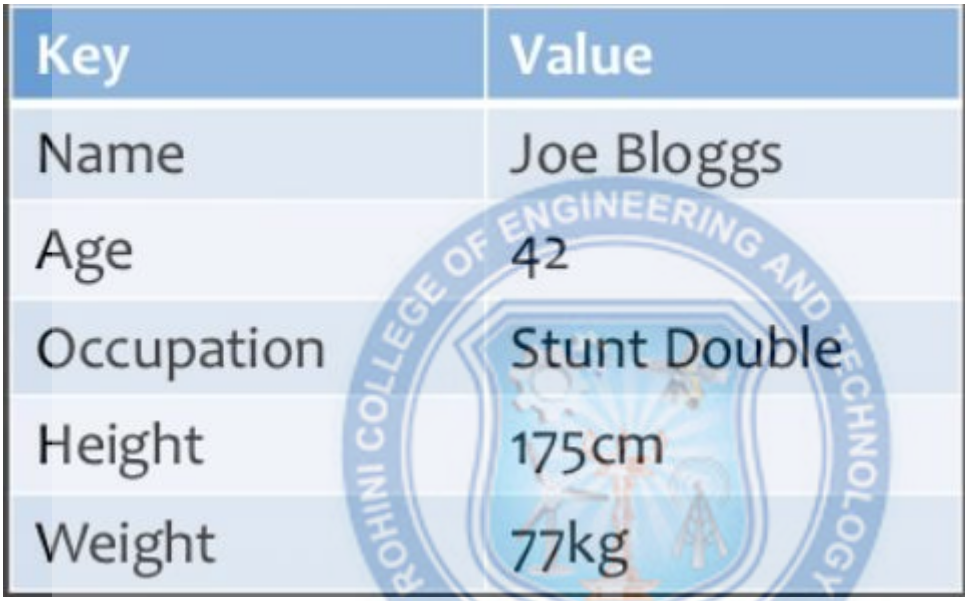


Key Value Pair Based

Data is stored in key/value pairs. It is designed in such a way to handle lots of data and heavy load.

Key-value pair storage databases store data as a hash table where each key is unique, and the value can be a JSON, BLOB(Binary Large Objects), string, etc.

For example, a key-value pair may contain a key like “Website” associated with a value like “Guru99”.



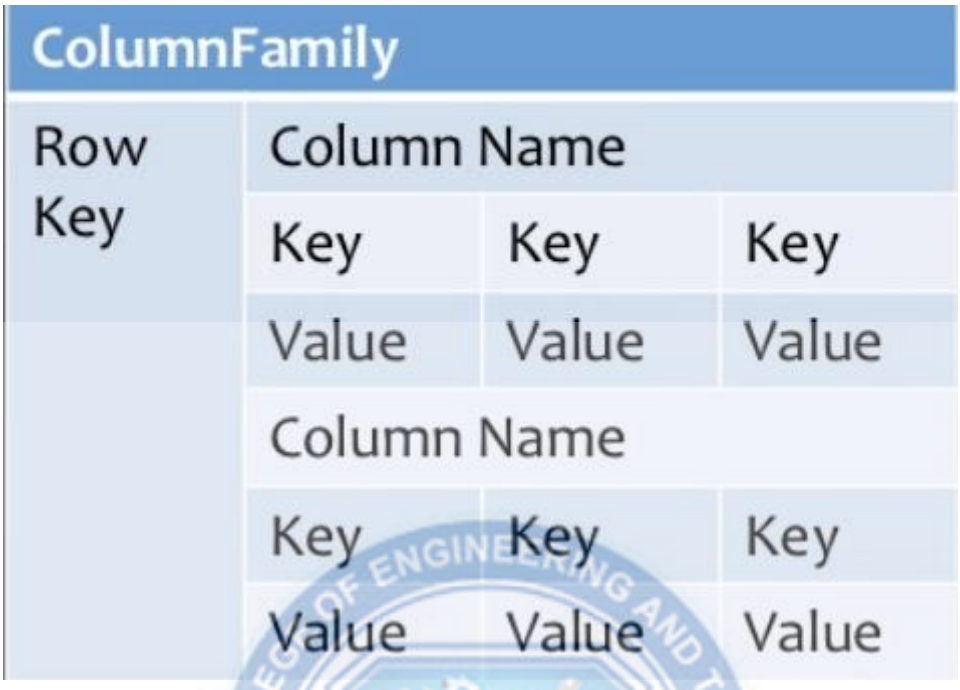
Key	Value
Name	Joe Bloggs
Age	42
Occupation	Stunt Double
Height	175cm
Weight	77kg

It is one of the most basic NoSQL database example. This kind of NoSQL database is used as a collection, dictionaries, associative arrays, etc. Key value stores help the developer to store schema-less data. They work best for shopping cart contents.

Redis, Dynamo, Riak are some NoSQL examples of key-value store DataBases. They are all based on Amazon’s Dynamo paper.

Column-based

Column-oriented databases work on columns and are based on BigTable paper by Google. Every column is treated separately. Values of single column databases are stored contiguously.



Column based NoSQL database

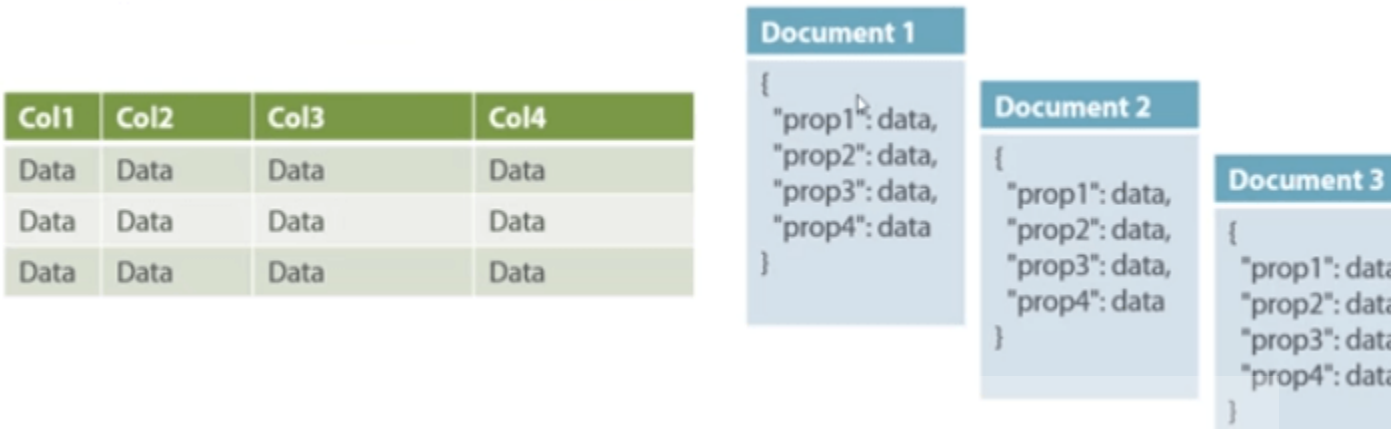
They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.

Column-based NoSQL databases are widely used to manage data warehouses, business intelligence, CRM, Library card catalogs,

HBase, Cassandra, HBase, Hypertable are NoSQL query examples of column based database.

Document-Oriented:

Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document. The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.



Relational Vs. Document

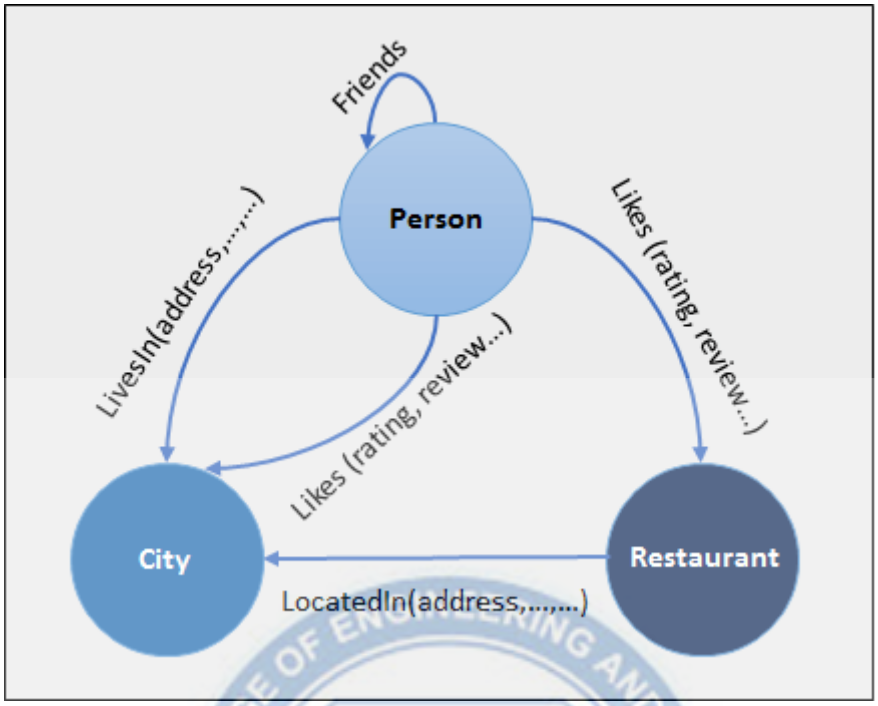
In this diagram on your left you can see we have rows and columns, and in the right, we have a document database which has a similar structure to JSON. Now for the relational database, you have to know what columns you have and so on. However, for a document database, you have data store like JSON object. You do not require to define which make it flexible.

The document type is mostly used for CMS systems, blogging platforms, real-time analytics & e-commerce applications. It should not use for complex transactions which require multiple operations or queries against varying aggregate structures.

Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes, MongoDB, are popular Document originated DBMS systems.

Graph-Based

A graph type database stores entities as well the relations amongst those entities. The entity is stored as a node with the relationship as edges. An edge gives a relationship between nodes. Every node and edge has a unique identifier.



Compared to a relational database where tables are loosely connected, a Graph database is a multi-relational in nature. Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.

Graph base database mostly used for social networks, logistics, spatial data.

Neo4J, Infinite Graph, OrientDB, FlockDB are some popular graph-based databases.

Advantages of NoSQL

- Can be used as Primary or Analytic Data Source
- Big Data Capability
- No Single Point of Failure
- Easy Replication
- No Need for Separate Caching Layer
- It provides fast performance and horizontal scalability.
- Can handle structured, semi-structured, and unstructured data with equal effect
- Object-oriented programming which is easy to use and flexible
- NoSQL databases don’t need a dedicated high-performance server
- Support Key Developer Languages and Platforms
- Simple to implement than using RDBMS
- It can serve as the primary data source for online applications.
- Handles big data which manages data velocity, variety, volume, and complexity
- Excels at distributed database and multi-data center operations
- Eliminates the need for a specific caching layer to store data
- Offers a flexible schema design which can easily be altered without downtime or service disruption

Disadvantages of NoSQL

- No standardization rules
- Limited query capabilities
- RDBMS databases and tools are comparatively mature
- It does not offer any traditional database capabilities, like consistency when multiple transactions are performed simultaneously.
- When the volume of data increases it is difficult to maintain unique values as keys become difficult
- Doesn't work as well with relational data
- The learning curve is stiff for new developers
- Open source options so not so popular for enterprises.

3.2 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

1. 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.
2. 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.
3. 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.
4. The data model available within MongoDB allows you to represent hierarchical relationships, to store arrays, and other more complex structures more easily.
5. 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

Key Components of MongoDB Architecture

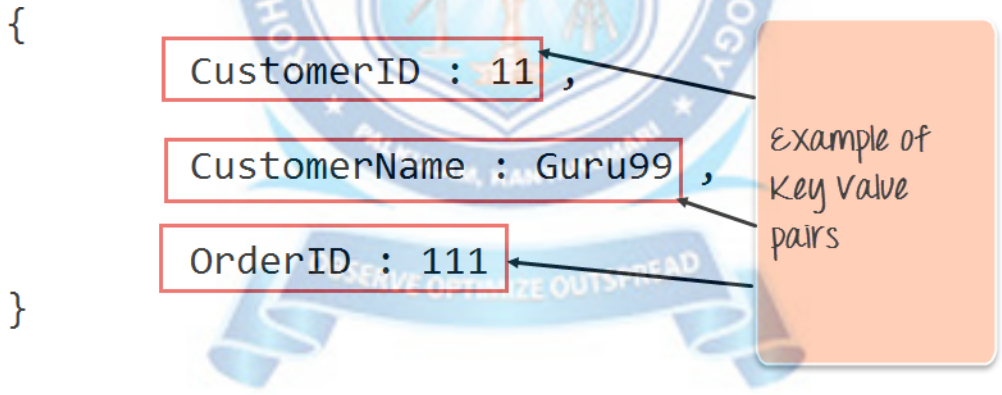
Below are a few of the common terms used in MongoDB

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.

3.3 Basic querying with MongoDB shell

3.4 Request body parsing in Express

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

body-parser doesn't have to be installed as a separate package because it is a dependency of express version 4.16.0+. body-parser isn't a dependency between version 4.0.0 and 4.16.0, so it will be installed separately in projects locked to those versions. body-parser middlewares will be required by express in versions of express with body-parser dependency. Versions of Express without body-parser will have to install it separately.

3.5 NodeJS MongoDB connection

Access MongoDB in Node.js

Learn how to access document-based database MongoDB using Node.js in this section.

In order to access MongoDB database, we need to install MongoDB drivers. To install native mongodb drivers using NPM, open command prompt and write the following command to install MongoDB driver in your application.

```
npm install mongodb --save
```

This will include mongodb folder inside node_modules folder. Now, start the MongoDB server using the following command. (Assuming that your MongoDB database is at C:\MyNodeJSConsoleApp\MyMongoDB folder.)

```
mongod -dbpath C:\MyNodeJSConsoleApp\MyMongoDB
```

Connecting MongoDB

The following example demonstrates connecting to the local MongoDB database.

app.js

```
var MongoClient = require('mongodb').MongoClient;

// Connect to the db
MongoClient.connect("mongodb://localhost:27017/MyDb", function (err, db) {

    if(err) throw err;

    //Write databse Insert/Update/Query code here..

});
```

In the above example, we have imported mongodb module (native drivers) and got the reference of MongoClient object. Then we used MongoClient.connect() method to get the

reference of specified MongoDB database. The specified URL "mongodb://localhost:27017/MyDb" points to your local MongoDB database created in MyMongoDB folder. The connect() method returns the database reference if the specified database is already exists, otherwise it creates a new database.

Now you can write insert/update or query the MongoDB database in the callback function of the connect() method using db parameter.

Insert Documents

The following example demonstrates inserting documents into MongoDB database.

app.js

```
var MongoClient = require('mongodb').MongoClient;

// Connect to the db
MongoClient.connect("mongodb://localhost:27017/MyDb", function (err, db) {

   db.collection('Persons', function (err, collection) {

      collection.insert({ id: 1, firstName: 'Steve', lastName: 'Jobs' });
      collection.insert({ id: 2, firstName: 'Bill', lastName: 'Gates' });
      collection.insert({ id: 3, firstName: 'James', lastName: 'Bond' });
      db.collection('Persons').count(function (err, count) {
         if (err) throw err;

         console.log('Total Rows: ' + count);
      });
   });

});
```

In the above example, db.collection() method creates or gets the reference of the specified collection. Collection is similar to table in relational database. We created a collection called Persons in the above example and insert three documents (rows) in it. After that, we display the count of total documents stored in the collection.

Running the above example displays the following result.

```
> node app.js
```

```
Total Rows: 3
```

3.6 Adding and retrieving data to MongoDB from NodeJS

MongoDB, the most popular NoSQL database, is an open-source document-oriented database. The term 'NoSQL' means 'non-relational'. It means that MongoDB isn't based on the table-like relational database structure but provides an altogether different mechanism for the storage and retrieval of data. This format of storage is called BSON (similar to JSON format).

Mongoose module

This module of Node.js is used for connecting the MongoDB database as well as for manipulating the collections and databases in MongoDB. The Mongoose. connect() method is used for connecting

the MongoDB database which is running on a particular server on your machine. We can also use promises, in this method in resolving the object containing all the methods and properties required for collection manipulation and in rejecting the error that occurs during connection.

Prerequisites

- **Node JS**
- **MongoDB**
- **Mongoose**

Approach for Retrieve Data

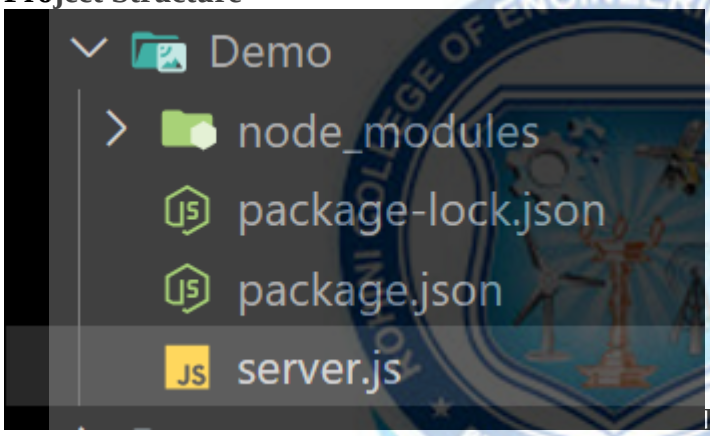
1. Install requires dependencies like express for creating server and mongoose for connecting ,fetching and modification in database.

2. Connect to your mongodb database using mongoose.connect() method and provide your database URL.

3. Created dummy data and create mongoose shema like GFGSchema and inset data in database when mongoose.connect() method called.

4. Create a rote called "/" which helps to retrieve data form specified database using find() method which returns array of object.

Project Structure



Project Structure

Steps to Setup Project

Step 1: Create a folder called Demo.

mkdir Demo

Step 2: Inside the root directory(Demo) initialize the app using following command.

cd Demo

npm init -y

Step 3: Install dependencies like mogoose express by following command.

npm i mongoose express

Step 4: Create server.js file inside root directory(Demo) and add the given code below.

mkdir server.js

Updated package.json look like

```
"dependencies": {
        "express": "^4.19.2",
        "mongoose": "^8.4.0"
      }
```

//server.js

```
//require('dotenv').config();
const express = require('express');
const mongoose = require('mongoose');
const app = express();

const port = process.env.PORT || 3000;

mongoose
  .connect("mongodb://localhost:27017/GFGDatabase")
```

```
    .then((err) => {

        console.log("Connected to the database");
        addDataToMongodb();
    });
const data = [
    {
        name: "John",
        class: "GFG"
    },
    {
        name: "Doe",
        class: "GFG"
    },
    {
        name: "Smith",
        class: "GFG"
    },
    {
        name: "Peter",
        class: "GFG"
    }
]
const gfgSchema = new mongoose
    .Schema({
        name: { type: String, required: true },
        class: { type: String, required: true },
    });

const GFGCollection = mongoose
    .model("GFGCollection", gfgSchema);

async function addDataToMongodb() {
    await GFGCollection
        .deleteMany();
    await GFGCollection
        .insertMany(data);
    console.log("Data added to MongoDB");
}

app.get('/', async (req, res) => {
    try {
        const data = await GFGCollection.find();
        res.json(data);
        console.log(data);
    } catch (err) {
        console.log(err);
        res.status(500).send("Internal Server Error");
    }
});

app
    .listen(port, () => {
        console.log(`Server is running on port ${port}`);
```

});

Steps to run Project:

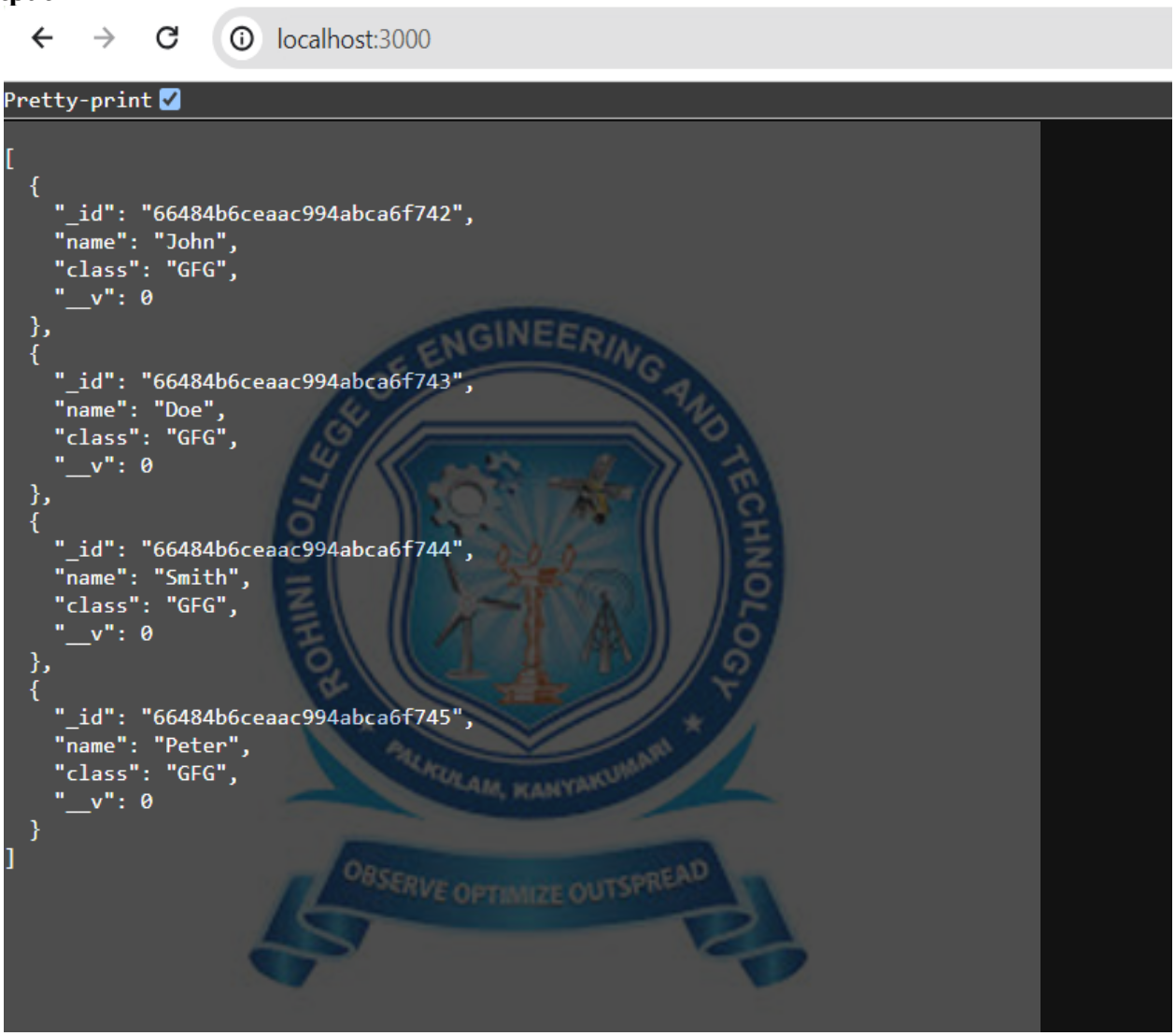
1. Navigate to root directory (Demo) and run following command.

node server.js

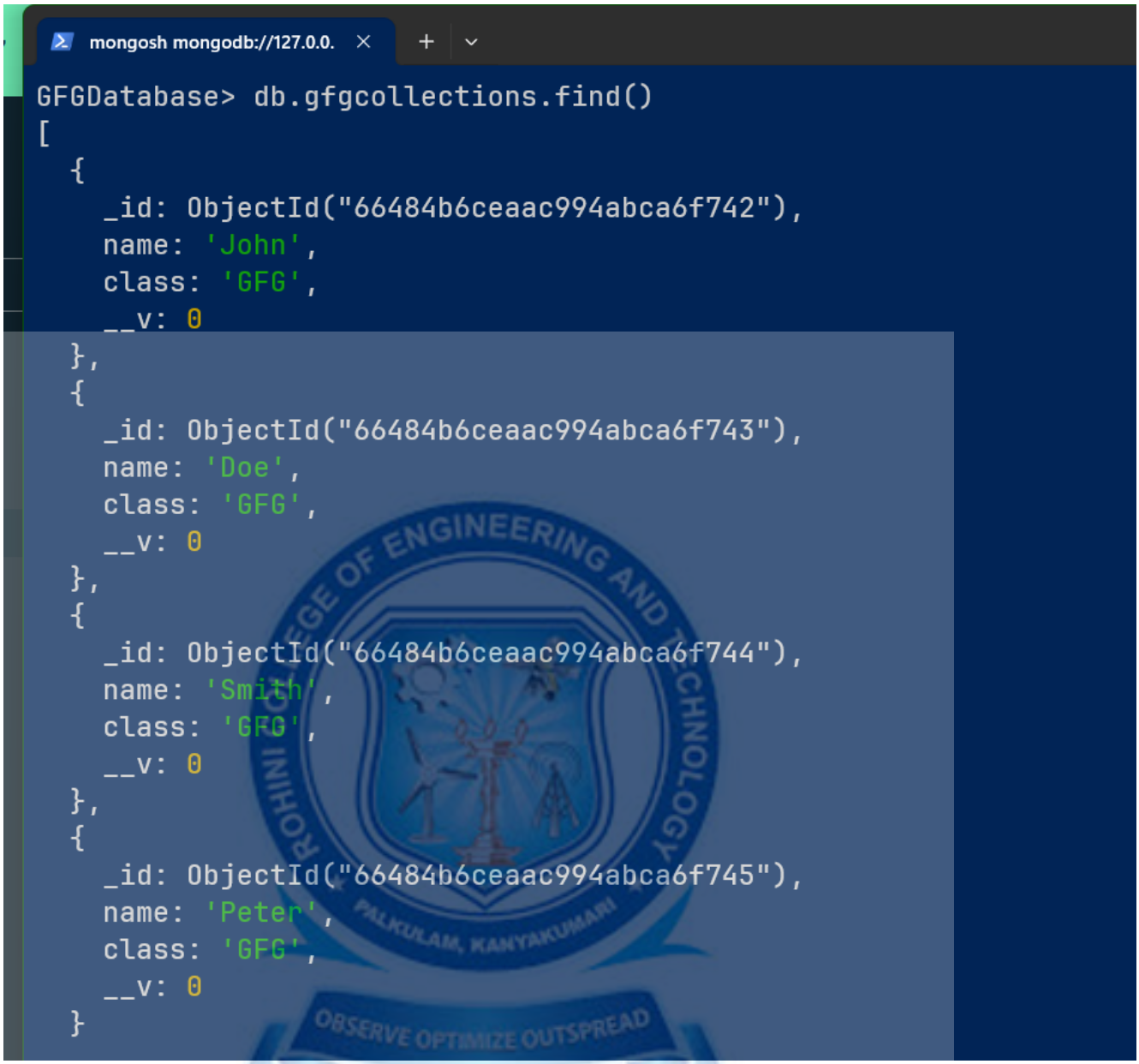
2. Open browser and type URL:

localhost:3000/

Output



Output in Browser



Output in Terminal

3.7 Handling SQL databases from NodeJS

Node.js can interact with SQL databases like MySQL, PostgreSQL, and SQLite using various database drivers and ORMs (Object-Relational Mappers).

1. Choosing a SQL Database for Node.js

Common SQL databases used with Node.js:

- **MySQL – Fast, widely used, good for web applications.**
- **PostgreSQL – More advanced, supports complex queries and data types.**
- **SQLite – Lightweight, best for local or small-scale applications.**
- **Microsoft SQL Server – Enterprise-level, often used in Windows environments.**

2. Installing Database Drivers

Each SQL database has its own driver for Node.js.

For MySQL

```
npm install mysql2
```

For PostgreSQL

```
npm install pg
```

For SQLite

```
npm install sqlite3
```

3. Connecting to a Database

MySQL Example

```
const mysql = require('mysql2');

// Create a connection
const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'password',
    database: 'testdb'
});

// Connect to MySQL
connection.connect(err => {
    if (err) {
        console.error('Error connecting to MySQL:', err);
        return;
    }
    console.log('Connected to MySQL database');
});
```

PostgreSQL Example

```
const { Client } = require('pg');


const client = new Client({
    host: 'localhost',
    user: 'postgres',
    password: 'password',
    database: 'testdb',
    port: 5432
});


client.connect()
    .then(() => console.log("Connected to PostgreSQL"))
    .catch(err => console.error("Connection error", err));
```

SQLite Example

```
const sqlite3 = require('sqlite3').verbose();
const db = new sqlite3.Database('./testdb.sqlite', sqlite3.OPEN_READWRITE, err => {
    if (err) {
        console.error(err.message);
    } else {
        console.log("Connected to SQLite database");
    }
});
```

4. Executing SQL Queries

Once connected, you can run **queries** to interact with the database.

MySQL Query Example

```
connection.query('SELECT * FROM users', (err, results) => {
    if (err) {
        console.error('Error executing query:', err);
```

```
    } else {
        console.log('User data:', results);
    }
});
```

PostgreSQL Query Example

```
client.query('SELECT * FROM users')
    .then(res => console.log(res.rows))
    .catch(err => console.error("Query error", err));
```

SQLite Query Example

```
    db.all('SELECT * FROM users', [], (err, rows) => {
    if (err) {
        console.error(err.message);
    } else {
        console.log(rows);
    }
});
```

5. Using Parameterized Queries (to Prevent SQL Injection)

Instead of inserting values directly into the query string (which is vulnerable to SQL injection), use **parameterized queries**.

```
const sql = 'SELECT * FROM users WHERE id = ?';
connection.query(sql, [1], (err, results) => {
    if (err) throw err;
    console.log(results);
});
```

PostgreSQL

```
client.query('SELECT * FROM users WHERE id = $1', [1])
  .then(res => console.log(res.rows))
  .catch(err => console.error(err));
```

SQLite

```
db.get('SELECT * FROM users WHERE id = ?', [1], (err, row) => {
  if (err) throw err;
  console.log(row);
});
```

6. Inserting Data

MySQL

```
const insertQuery = 'INSERT INTO users (name, age) VALUES (?, ?)';
connection.query(insertQuery, ['Alice', 30], (err, result) => {
  if (err) throw err;
  console.log('Inserted ID:', result.insertId);
});
```

PostgreSQL

```
client.query('INSERT INTO users (name, age) VALUES ($1, $2) RETURNING id', ['Alice', 30])
  .then(res => console.log('Inserted ID:', res.rows[0].id))
  .catch(err => console.error(err));
```

SQLite

```
db.run('INSERT INTO users (name, age) VALUES (?, ?)', ['Alice', 30], function(err) {
  if (err) throw err;
  console.log('Inserted ID:', this.lastID);
});
```

3.8 Handling Cookies in NodeJS

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:

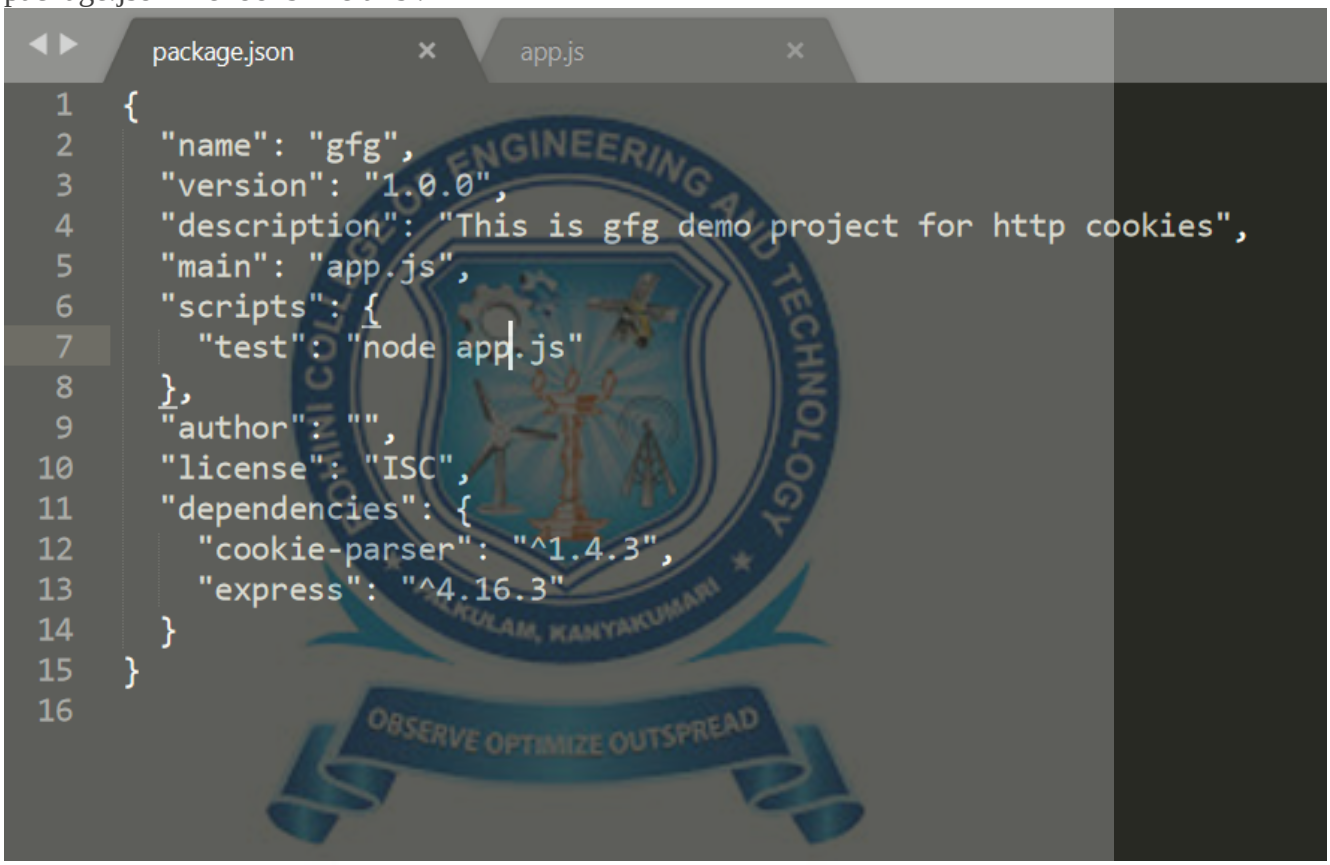
```
npm init
```

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 :



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');
});
```

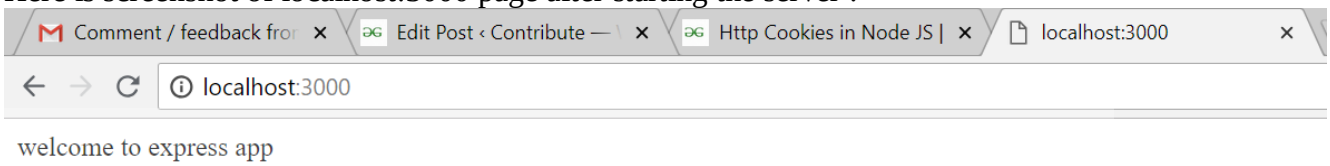
After that if we run the command

```
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 :



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 :

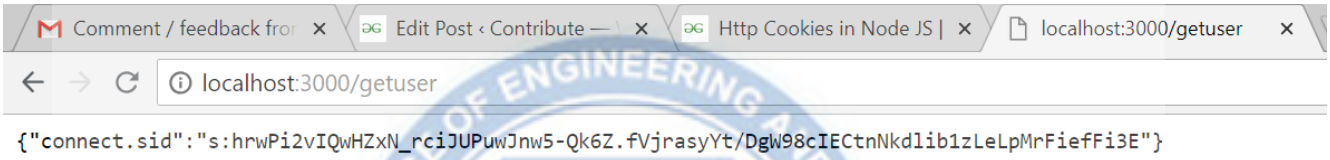
```
res.cookie(name_of_cookie, value_of_cookie);
```

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');
```

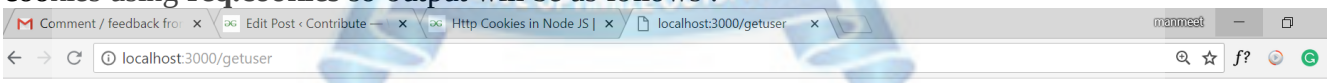
```
});

//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 :



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

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 :

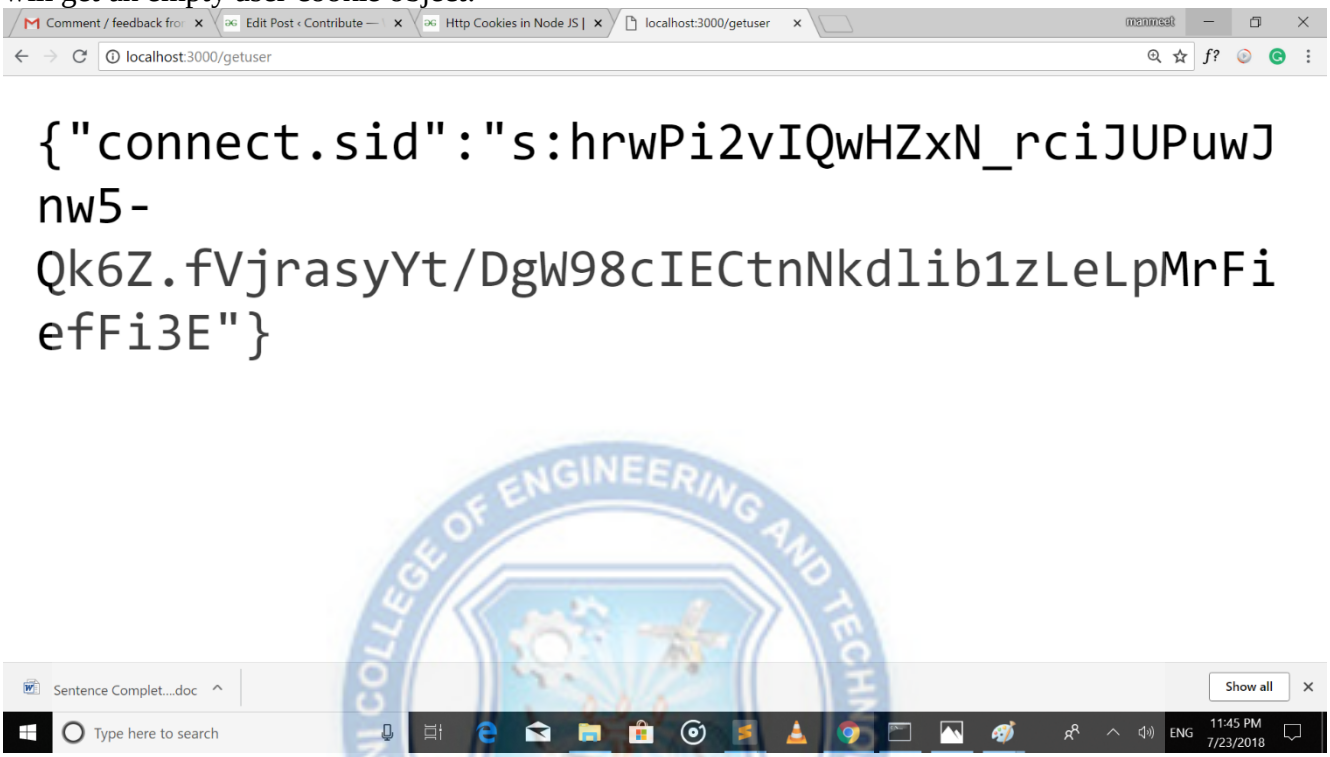
```
res.clearCookie(cookieName);
```

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: user logged out[/caption]

To check whether cookies are destroyed or not make a get request to localhost:3000/getuserand you will get an empty user cookie object.



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.

10.1 Handling User Authentication with node js

Authentication in NodeJS involves verifying the identity of users accessing a web application or API endpoint. It typically involves processes such as user login, session management, and token-based authentication to ensure secure access to resources.

What is Authentication?

Authentication is the process of verifying the identity of a user or system. In the context of web development, authentication is commonly used to grant access to users based on their credentials, such as username and password.

Why Use Authentication?

Authentication is crucial for protecting sensitive information and restricting access to authorized users. By implementing authentication mechanisms, you can ensure that only authenticated users can access certain features or resources within your application.

Handle Authentication in NodeJS:

Authentication in NodeJS can be implemented using various techniques, including:

- **Session-Based Authentication:** In session-based authentication, the server creates a session for each authenticated user and stores session data on the server. This session data is used to validate subsequent requests from the user.

- **Token-Based Authentication:** Token-based authentication involves issuing a unique token to each authenticated user upon login. This token is then sent with subsequent requests as an authorization header or a cookie to authenticate the user.

- **Middleware:** Middleware functions can be used to enforce authentication and authorization rules for specific routes or endpoints in your application. These middleware functions can check for valid authentication tokens or session data before allowing access to protected resources.

```
const passport = require('passport');
    const LocalStrategy = require('passport-local').Strategy;

    passport.use(new LocalStrategy(
       (username, password, done) => {
          // Validate username and password
          // Example: Check against database
       }
    ));

    app.post('/login', passport.authenticate('local'), (req, res) => {
       // Authentication successful
       res.send('Authentication successful');
    });

    function isAuthenticated(req, res, next) {
       if (req.isAuthenticated()) {
          return next();
       }
       res.status(401).send('Unauthorized');
    }

    app.get('/profile', isAuthenticated, (req, res) => {
       // Return user profile data
       res.send(req.user);
    });
```