Learning Resources

- Books
  - D. Brad, B. Dayley and C. Dayley, "Node.js, MongoDB and Angular Web Development: The definitive guide to using the MEAN stack to build web applications", Addison-Wesley Professional, 2nd Edition, 2017
  - Web Links (Strictly Referred):
    - https://www.mongodb.com/docs/

Course Outcome

- CO1: Relate the basics of Javascript (JS) and ReactJS
- CO2: Apply the concepts of props and State Management in React JS
- CO3: Examine Redux and Router with React JS
- CO4: Appraise Node JS environment and modular development
- CO5: Develop full stack applications using MongoDB

Overview

UNIT-4

- MongoDB:
  - Introduction to NoSQL
  - Understanding MongoDB datatypes
  - Building MongoDB Environment (premise and cloud based)
  - Administering Databases and User accounts
  - Configuring Access Control, Managing Collections
  - connecting to MongoDB from Node.js
  - Accessing and Manipulating Databases and Collections
  - Manipulating MongoDB documents from Node.js
  - Understanding Query objects,
  - sorting and limiting result sets

No SQL

- No SQL (not only SQL)
  - refers to **nonrelational** types of databases, and these databases store data in a format that's different from relational tables
  - can be queried using **idiomatic language** APIs, **declarative structured query** languages, and **query-by example** languages, which is why they are also referred to as "not only SQL" databases
  - are widely used in **real-time web applications** and **big data**, because their main advantages are **high scalability and high availability**.
  - are also the **preferred choice of developers** to an **agile development** paradigm by **rapidly adapting to changing requirements.**
  - allow the data to be stored in ways
    - more intuitive and easier to understand, or
    - closer to the way the data is used by applications—with fewer transformations required when storing or retrieving using NoSQL-style APIs

NoSQL vs. SQL

- SQL databases are relational, while NoSQL databases are non-relational.
- The relational database management system (RDBMS) is the basis for structured query language (SQL), which lets users access and manipulate data in highly structured tables.
- Tables are foundational model for database systems such as MS SQL Server, IBM DB2, Oracle, and MySQL. But with NoSQL databases, the data access syntax can be different from database to database.

NoSQL vs. Relational database

- RDBMS
  - The data in an RDBMS is stored in database objects that are called tables.
  - A table is a collection of related data entries, and it consists of columns and rows.
  - These databases require defining the schema upfront, that is, all of the columns and their associated datatypes must be known beforehand so applications can write data to the database.
  - They also store information linking multiple tables through the use of keys, thus creating a relationship across multiple tables.
  - In the simplest case, a key is used to retrieve a specific row so that it can be examined or modified.

NoSQL vs. Relational database

- No SQL
  - data can be stored without defining the schema upfront
  - which means you have the ability to get moving and iterate quickly, defining the data model as you go.
  - This can be suitable for specific business requirements, whether it's graph-based, column-oriented, document-oriented, or as a key-value store.

NoSQL vs. Relational database

- Relational databases were the most widely used models.
- RDBMS are still extremely ubiquitous with many businesses; however, the variety, velocity, and volume of data that's being accessed today sometimes requires a very different database to complement the relational database.
- This has sparked the adoption in some areas of NoSQL databases—which are also referred to as "nonrelational databases."
- Because of their ability to scale out horizontally and quickly, nonrelational databases can handle high traffic, which also makes them highly adaptable.

When to choose a NoSQL database?

- With businesses and organizations needing to innovate rapidly, being able to stay **agile and continue operating** at any scale is the name of the game.
- NoSQL databases **offer flexible schemas** and also support a **variety of data models** that are ideal for **building applications** that require **large data volumes** and **low latency** or **response times**—for example, online gaming and ecommerce web applications.
- NoSQL databases typically rely on de-normalized data, supporting the types of applications that use fewer tables (or containers) and whose data relationships are not modeled using references.

When not to choose a NoSQL database?

- Many classic back-office business applications in **finance, accounting, and enterprise resource planning** rely on highly normalized data to prevent **data anomalies as well as data duplication**.
- NoSQL database typically do not offer complex joins, sub-queries, and nesting of queries in a WHERE clause.

Benefits of No SQL

- Flexibility
- Scalability
- High performance
- Availability
- Highly Functional

https://www.oracle.com/in/database/nosql/what-is-nosql/

Types of No SQL

- **Key value**
- **Document**: document-oriented databases, these databases are used for storing, retrieving, and managing semi-structured data
- **Graph**: This database organizes data as nodes and relationships, which show the connections between nodes. Graph databases are applied in social networks, reservation systems, and fraud detection.
- **Wide column**: store and manage data in the form of tables, rows, and columns. They are broadly deployed in applications that require a column format to capture schema-free data.

Mongo Data Types

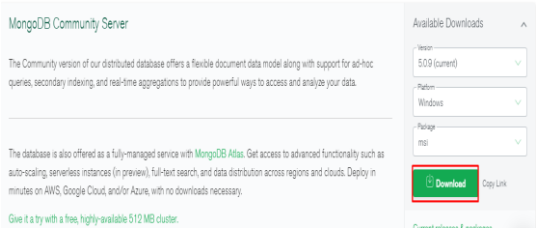
Type	Number	Alias	Notes
Double	1	"double"	
String	2	"string"	
Object	3	"object"	
Array	4	"array"	
Binary data	5	"binData"	
Undefined	6	"undefined"	Deprecated.
ObjectId	7	"objectId"	
Boolean	8	"bool"	
Date	9	"date"	
Null	10	"null"	

Mongo Data Types

Type	Number	Alias	Notes
Regular Expression	11	"regex"	
DBPointer	12	"dbPointer"	Deprecated.
JavaScript	13	"javascript"	
Symbol	14	"symbol"	Deprecated.
JavaScript code with scope	15	"javascriptWithScope"	Deprecated in MongoDB 4.4.
32-bit integer	16	"int"	
Timestamp	17	"timestamp"	
64-bit integer	18	"long"	
Decimal128	19	"decimal"	New in version 3.4.
Min key	-1	"minKey"	
Max key	127	"maxKey"	

Building MongoDB Environment (premise and cloud based)

- Navigate to https://www.mongodb.com/try/download/community and click Download to install latest MongoDB Community Server.

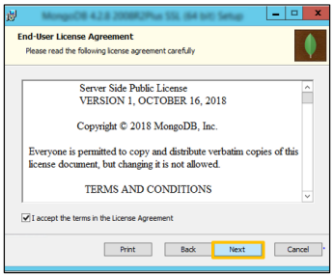


Building MongoDB Environment (premise and cloud based)

- Open MongoDB exe file.
  This will launch setup screen.

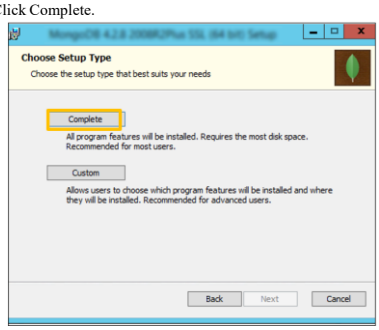
Building MongoDB Environment (premise and cloud based)

- Click Next.
- Accept the Licence Agreement terms and click Next.



Building MongoDB Environment (premise and cloud based)

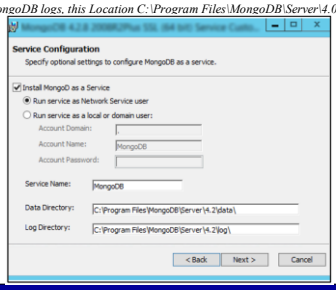
- Click Complete.



Building MongoDB Environment (premise and cloud based)

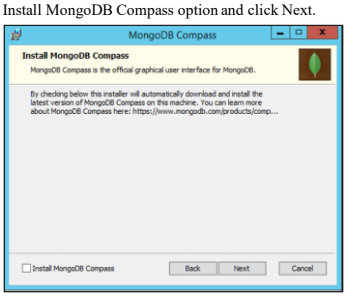
- Select the desired location for data and log of MongoDB and click Next.
- ***Note***: *1. For MongoDB data, this location C:\Program Files\MongoDB\Server\4.0\data will be selected by default.*

  2. For MongoDB logs, this Location C:\Program Files\MongoDB\Server\4.0\log will be selected by default.



Building MongoDB Environment (premise and cloud based)

- Clear Install MongoDB Compass option and click Next.

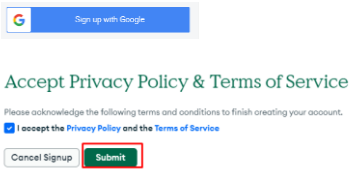


Building MongoDB Environment (premise and cloud based)

- Click Finish.
- Copy the MongoDB Path where MongoDB is installed (ex: C:\Program Files\MongoDB\Server\4.0\bin).
- Go to Control Panel > System and Security > System > Advanced System Setting and click Environment Variables > from System Variable section select Path > click Edit. Paste the copied path in path variable.
- Click Test Connection.

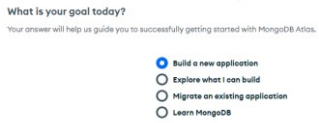
Building MongoDB Environment (cloud based)

- Click Start Free.
- Click Sign up with google and accept the terms of service and privacy policy and click Submit.



Building MongoDB Environment (cloud based)

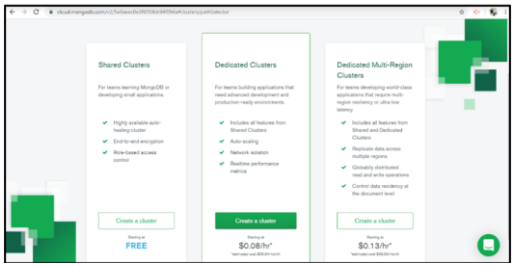
- Setup the things as per your requirements.
- E.g.



-

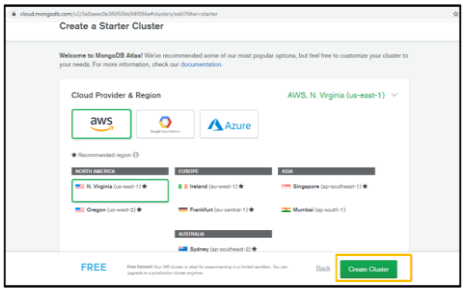
Building MongoDB Environment (cloud based)

- Select the required cluster and click Create a Cluster.



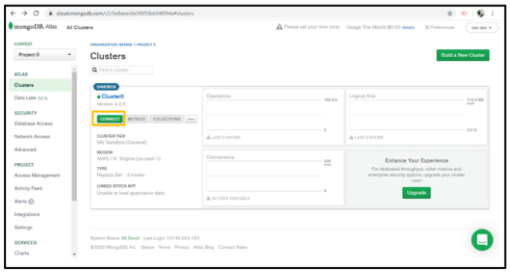
Building MongoDB Environment (cloud based)

- Select the Cloud Provider and Region and click Create Cluster.

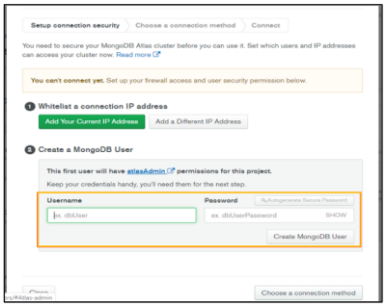


Building MongoDB Environment (cloud based)

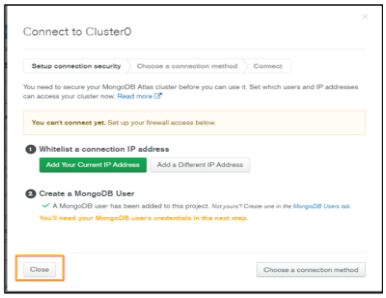
- Create MongoDB user
- Navigate to Clusters and select a cluster and click Connect.



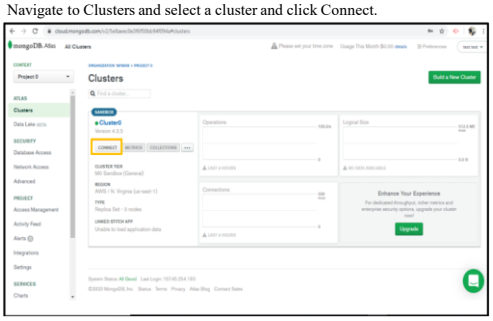
Building MongoDB Environment (cloud based)

- Enter Username and Password and click Create MongoDB User.
- 

Building MongoDB Environment (cloud based)

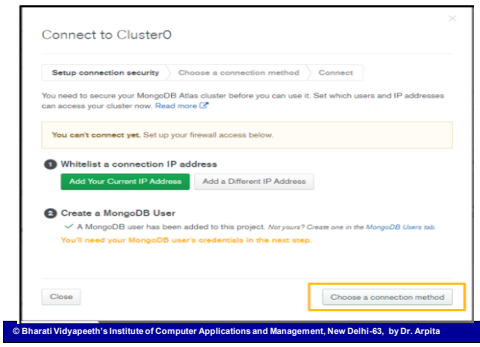
- Once the user is created successfully, click close.
- 

Building MongoDB Environment (cloud based)

- Navigate to Clusters and select a cluster and click Connect.
- 

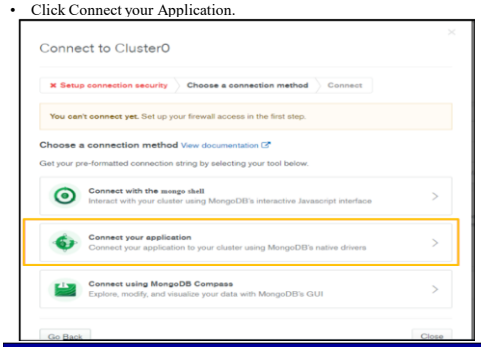
Building MongoDB Environment (cloud based)

- Click Choose a connection method.



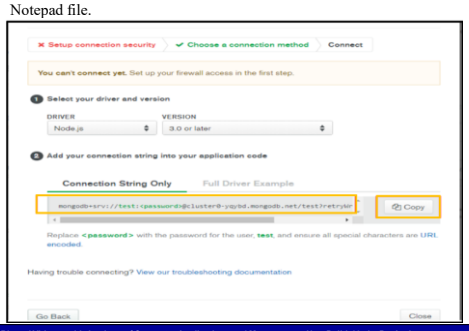
Building MongoDB Environment (cloud based)

- Click Connect your Application.



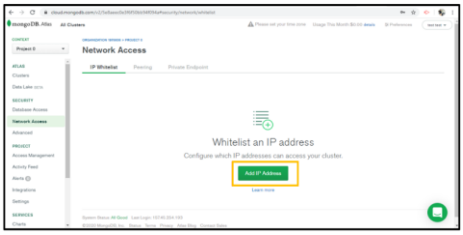
Building MongoDB Environment (cloud based)

- Click Copy. The connection string will be copied and you can save it in a Notepad file.



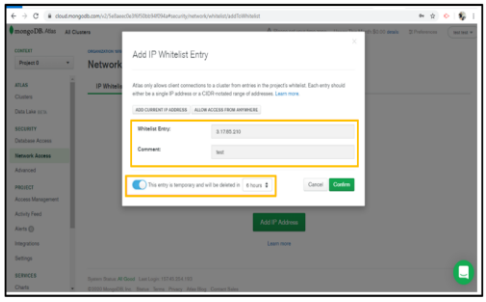
Building MongoDB Environment (cloud based)

- Whitelist the Sever IP.
- Navigate to Network Access and click Add IP Address.



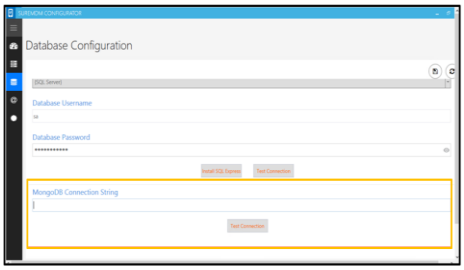
Building MongoDB Environment (cloud based)

- Enter IP address in Whitelist Entry field and enter Comment and click Confirm.



Building MongoDB Environment (cloud based)

- Navigate to SureMDM Configurator > Database Configuration and paste the MongoDB Connection String copied from step no.13 and click Test Connection..



Start Mongo DB

- To start database engine in windows : mongod
- starting MongoDB with a port and dbpath
- mongod –port 28008 –dbpath <mongo_data_location>/data/db

Parameter	Description
`--help, -h`	Returns basic help and usage text.
`--version`	Returns the version of MongoDB.
`--config <filename>,` `-f <filename>`	Specifies a configuration file that contains runtime-configurations.
`--quiet`	Reduces the amount of reporting sent to the console and log file.
`--port <port>`	Specifies a TCP port for `mongod` to listen for client connections. Default: `27017`.
`--bind_ip <ip address>`	Specifies the IP address on which `mongod` will bind to and listen for connections. Default: `All Interfaces`
`--repair`	Runs a repair routine on all databases.

Administering User Accounts

- Listing Users
  - use admin
  - show users
- Create new User
  - use test
  - db.createUser({ **user**: "testUser", **pwd**: "test", **roles**: ["readWrite", "dbAdmin"] })
- Remove User
  - use testDB
  - db.removeUser("testUser")

Configuring Access Control

- Creating User Admin
  - use admin
  - db.createUser({ user: "<username>",
  - pwd: "<password>",
  - roles: ["userAdminAnyDatabase"] })
- Turning on Authentication
  - mongod –dbpath <mongo_data_location>/data/db –auth
- add users with rights to the database:
  - use admin
  - db.auth("useradmin", "test")

Administering Databases

- Displaying a List of Databases
  - show dbs
- Changing the Current Database
  - db = db.getSiblingDB('testDB')
  - use testDB
- Creating Database
  - use newDB
  - db.createCollection("newCollection")
- Deleting Databases
  - use newDB
  - db.dropDatabase()
- Copying Databases
  - db.copyDatabase('customers', 'customers_archive')

Managing Collections

- Displaying a List of Collections in a Database
  - use test
  - show collections
- Creating Collections
  - db.createCollection("newCollection")
- Deleting Collections
  - use testDB
  - show collections
  - coll = db.getCollection("newCollection")
  - coll.drop()
  - show collections
- Finding Documents in a Collection
  - use testDB
  - coll = db.getCollection("newCollection")
  - coll.find()
  - coll.find({speed:"120mph"})

Managing Collections

- Adding Documents to a Collection
  - use testDB
  - coll = db.getCollection("newCollection")
  - coll.find()
  - coll.insert({ vehicle: "plane", speed: "480mph" })
  - coll.insert({ vehicle: "car", speed: "120mph" })
  - coll.insert({ vehicle: "train", speed: "120mph" })
  - coll.find()
- Deleting Documents in a Collection
  - use testDB
  - coll = db.getCollection("newCollection")
  - coll.find()
  - coll.remove({vehicle: "plane"})
  - coll.find()
  - coll.remove()
  - coll.find()

Managing Collections

- Updating Documents in a Collection
  - use testDB
  - coll = db.getCollection("newCollection")
  - coll.find()
  - coll.update({ speed: "120mph" },
    { $set: { speed: "150mph" , updated: true } },
    { upsert: false, multi: true })
  - coll.save({ "_id" : ObjectId("52a0caf33120fa0d0e424ddb"),
    "vehicle" : "plane", "speed" : "500mph" })
  - coll.find()

Adding the MongoDB Driver to Node.js

- Connecting to MongoDB from Node.js Using the MongoClient Object
  - var client = new MongoClient();
- Driver Syntax
  - mongodb://[username:password@]host[:port][/[database][?options]]
- Connect
  - client.connect('mongodb://MyDBServer:8088/MyDB');

Adding the MongoDB Driver to Node.js

Option	Description
`mongodb://`	Specifies that this string is using a MongoDB connection forma
`username`	(Optional) Specifies the user name to use when authenticating.
`password`	(Optional) Specifies the password to use when authenticating.
`host`	Specifies the host name or address of the MongoDB server. Yo multiple `host:port` combinations to connect to multiple Mo by separating them by a comma. For example: `mongodb://host1:270017,host2:27017,host3:2`
`port`	Specifies the port to use when connecting to the MongoDB ser `27017`.
`database`	Specifies the database name to connect to. Default is `admin`.
`options`	Specifies the `key/value` pairs of options to use when conne same options can be specified in the `dbOpt` and `serverOpt`

db_connect_url.js

```
var MongoClient = require('mongodb').MongoClient,
    Server = require('mongodb').Server;
var client = new MongoClient();
client.connect('mongodb://dbadmin:test@localhost:27017/testDB',
               { poolSize: 5, reconnectInterval: 500 },
function(err, db) {
  if (err){
    console.log("Connection Failed Via Client Object.");
  } else {
    console.log("Connected Via Client Object . . .");
    db.logout(function(err, result) {
      if(!err){
        console.log("Logged out Via Client Object . . .");
      }
      db.close();
          console.log("Connection closed . . .");
    });
  }
```


Accessing and Manipulating Databases

- Listing Databases

```
MongoClient.connect("mongodb://localhost/admin", function(err, db) {
  var adminDB = db.admin();
  adminDB.listDatabases(function(err, databases){
    console.log("Before Add Database List: ");
    console.log(databases);
  });
});
```


Accessing and Manipulating Databases

- Creating a Database

```
var MongoClient = require('mongodb').MongoClient;
MongoClient.connect("mongodb://localhost/", function(err, db) {
  var newDB = db.db("newDB");
  newDB.createCollection("newCollection", function(err, collection){
    if(!err){
      console.log("New Database and Collection Created");
    }
  });
});
```

Accessing and Manipulating Databases

- Deleting a Database

```
newDB.dropDatabase(function(err, results){
  <handle database delete here>
});
```


Accessing and Manipulating Collections

- Listing Collections

```
var newDB = db.db("newDB");
newDB.collections(function(err, collections){})
```

- Creating Collections

```
var newDB = db.db("newDB");
newDB.createCollection("newCollection", function(err, collection){ })
```

- Deleting Collections

```
var myDB = db.db("myDB ");
myDB.dropCollection("collectionA", function(err, results){ })
myDB.collection("collectionB", function(err, collB){
  collB.drop();
})
```


Manipulating MongoDB Documents from Node.js

- Adding Documents to a Collection

```
var MongoClient = require('mongodb').MongoClient;
function addObject(collection, object){
  collection.insert(object, function(err, result){
    if(!err){
      console.log("Inserted : ");
      console.log(result);
    }
  });
}
MongoClient.connect("mongodb://localhost/", function(err, db) {
  var myDB = db.db("astro");
  myDB.dropCollection("nebulae");
  myDB.createCollection("nebulae", function(err, nebulae){
   addObject(nebulae, {ngc:"NGC 7293", name:"Helix",
     type:"planetary",location:"Aquila"});
   addObject(nebulae, {ngc:"NGC 6543", name:"Cat's Eye",
     type:"planetary",location:"Draco"});
   addObject(nebulae, {ngc:"NGC 1952", name: "Crab",
     type:"supernova",location:"Taurus"});
  });
});
```

Manipulating MongoDB Documents from Node.js

- Getting Documents from a Collection

```
var MongoClient = require('mongodb').MongoClient;
MongoClient.connect("mongodb://localhost/", function(err, db) {
  var myDB = db.db("astro");
  myDB.collection("nebulae", function(err, nebulae){
    nebulae.find(function(err, items){
      items.toArray(function(err, itemArr){
        console.log("Document Array: ");
        console.log(itemArr);
      });
    });
    nebulae.find(function(err, items){
      items.each(function(err, item){
        if(item){
          console.log("Singular Document: ");
          console.log(item);
        }
      });
    });
    nebulae.findOne({type:'planetary'}, function(err, item){
      console.log("Found One: ");
      console.log(item);
    });
  });
});
```


Manipulating MongoDB Documents from Node.js

- Updating Documents in a Collection

```
var MongoClient = require('mongodb').MongoClient;
MongoClient.connect("mongodb://localhost/", function(err, db) {
  var myDB = db.db("astro");
  myDB.collection("nebulae", function(err, nebulae){
    nebulae.find({type:"planetary"}, function(err, items){
      items.toArray(function(err, itemArr){
        console.log("Before Update: ");
        console.log(itemArr);
        nebulae.update({type:"planetary", $isolated:1},
                       {$set:{type:"Planetary", updated:true}},
                       {upsert:false, multi:true, w:1},
                       function(err, results){
          nebulae.find({type:"Planetary"}, function(err, items){
            items.toArray(function(err, itemArr){
              console.log("After Update: ");
              console.log(itemArr);
              db.close();
            });
          });
```


Manipulating MongoDB Documents from Node.js

- Saving Documents in a Collection

```
var MongoClient = require('mongodb').MongoClient;
MongoClient.connect("mongodb://localhost/", function(err, db) {
  var myDB = db.db("astro");
  myDB.collection("nebulae", function(err, nebulae){
    nebulae.findOne({type:"supernova"}, function(err, item){
      console.log("Before Save: ");
      console.log(item);
      item.info = "Some New Info";
      nebulae.save(item, {w:1}, function(err, results){
        nebulae.findOne({_id:item._id}, function(err, savedItem)
          console.log("After Save: ");
          console.log(savedItem);
          db.close();
        });
      });
    });
  });
});
```

Manipulating MongoDB Documents from Node.js

- Deleting Documents from a Collection

```
var MongoClient = require('mongodb').MongoClient;
MongoClient.connect("mongodb://localhost/", function(err, db) {
  var myDB = db.db("astro");
  myDB.collection("nebulae", function(err, nebulae){
    nebulae.find(function(err, items){
      items.toArray(function(err, itemArr){
        console.log("Before Delete: ");
        console.log(itemArr);
        nebulae.remove({type:"planetary"}, function(err, results
          console.log("Deleted " + results + " documents.");
          nebulae.find(function(err, items){
            items.toArray(function(err, itemArr){
              console.log("After Delete: ");
              console.log(itemArr);
              db.close();
            });
          });
        });
      });
    });
  });
});
```


Manipulating MongoDB Documents from Node.js

- Removing a Single Document from a Collection

```
var MongoClient = require('mongodb').MongoClient;
MongoClient.connect("mongodb://localhost/", function(err, db) {
  var myDB = db.db("astro");
  myDB.collection("nebulae", function(err, nebulae){
    nebulae.find(function(err, items){
      items.toArray(function(err, itemArr){
        console.log("Before Delete: ");
        console.log(itemArr);
        nebulae.findAndRemove({type:"planetary"}, [['name', 1]],
                              {w:1}, function(err, results){
          console.log("Deleted " + results + " documents.");
          nebulae.find(function(err, items){
            items.toArray(function(err, itemArr){
              console.log("After Delete: ");
              console.log(itemArr);
              db.close();
            });
          });
        });
      });
    });
  });
});
```


Accessing MongoDB from Node.js

- Understanding Query Objects

  For example, to find all documents with a `count` value greater than `10` and `name` value equal to `test`, the `query` object would be

```
{count:{$gt:10}, name:'test'}
```

Operator	Description
`$eq`	Matches documents with fields that have a value equal to the value specified.
`$gt`	Matches values that are greater than the value specified in the query. For example: `{size:{$gt:5}}`
`$gte`	Matches values that are equal to or greater than the value specified in the query. For example: `{size:{$gte:5}}`
`$in`	Matches any of the values that exist in an array specified in the query. For example: `{name:{$in:['item1', 'item2']}}`
`$lt`	Matches values that are less than the value specified in the query. For example: `{size:{$lt:5}}`

Accessing MongoDB from Node.js

- Understanding Query Objects

Operator	Description
`$lte`	Matches values that are less than or equal to the value specified in the query. For example: `{size:{$lte:5}}`
`$ne`	Matches all values that are not equal to the value specified in the query. For example: `{name:{$ne:"badName"}}`
`$nin`	Matches values that do not exist in an array specified to the query. For example: `{name:{$nin:['item1', 'item2']}}`
`$or`	Joins query clauses with a logical OR; returns all documents that match the conditions of either clause. For example: `{$or:[{size:{$lt:5}}, {size:{$gt:10}}]}`
`$and`	Joins query clauses with a logical AND; returns all documents that match the conditions of both clauses. For example: `{$and:[{size:{$gt:5}}, {size:{$lt:10}}]}`

Understanding Query Options Objects

Operator	Description
`limit`	Specifies the maximum number of documents to return.
`sort`	Specifies the sort order of documents as an array of `[field, <sort_order>]` elements where `sort order` is `1` for ascending and `-1` for descending. For example: `sort:[['name':1],['value':-1]]`
`fields`	Specifies an object whose fields match fields that should be included or excluded from the returned documents. A value of `1` means include, a value of `0` means exclude. You can only include or exclude, not both. For example: `fields:{name:1,value:1}`
`skip`	Specifies the number of documents from the query results to skip before returning a document. Typically used when paginating result sets.
`hint`	Forces the query to use specific indexes when building the result set. For example: `hint:{'_id':1}`

Limiting Result Sets

- Limit the result sets that match a specific query in three ways:
  - simply only accept a limited number of documents
  - limit the fields returned,
  - page the results and get them in chunks.
- Limiting Results by Size

```
limit:<maximum_documents_to_return>

words.find({first:'p'}, {limit:5}, function(err, cursor){
  displayWords("Limiting words starting with p : ", cursor);
});
```

- Limiting Fields Returned in Objects
  - to exclude the fields **stats**, **value**, and **comments** when returning a document, you would use the following fields option:

```
{fields:{stats:0, value:0, comments:0}}

{fields:{name:1, value:1}}  ?
```

Sorting Result Sets

- The sort option is specified using an array of [field,<sort_order>] pairs, where sort_order is 1 for ascending and -1 for descending.

```
sort:[['value':-1]['name':1]]
```


Thank you
and
all the best!!