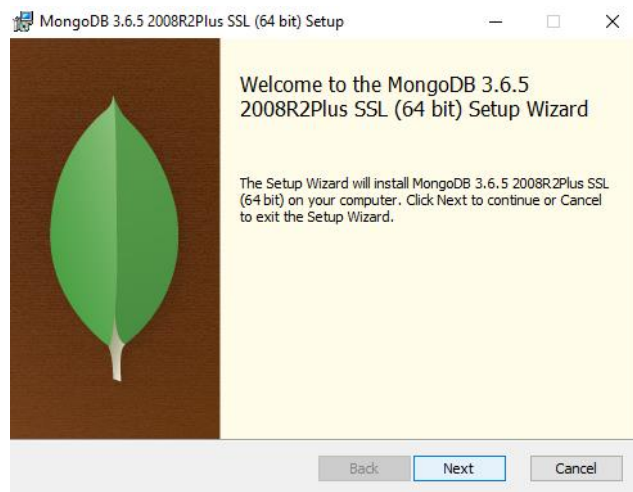


1) Install and setup MongoDB database server

1.1 Download and install MongoDB community server for windows 64 bit version.

<https://www.mongodb.com/download-center#community>

1.2 In Windows Explorer, locate the downloaded MongoDB .msi file, which typically is located in the default Downloads folder. Double-click the .msi file. The Windows Installer guides you through the installation process.



1.3 Setting up the data directory path and log using configuration file

MongoDB requires a data directory to store all data. MongoDB's default data directory path is the absolute path `\data\db` on the drive from which you start MongoDB. It is better you set the necessary database parameters such as path for data file and log.

Open `mongod.cfg` under `C:\Program Files\MongoDB\Server\3.6\bin` and use the following parameter to set and save the file.

```
systemLog:
  destination: file
  path: c:\data\log\mongod.log
storage:
  dbPath: c:\data\db
```

Create two folder where you want to store the data and its log files.

Eg. `c:\data\log` and `c:\data\db`

1.4 Starting MongoDB server in your local machine

Run mongod command from command prompt to start MongoDB

C:\Program Files\MongoDB\Server\3.6\bin\ mongod

Or

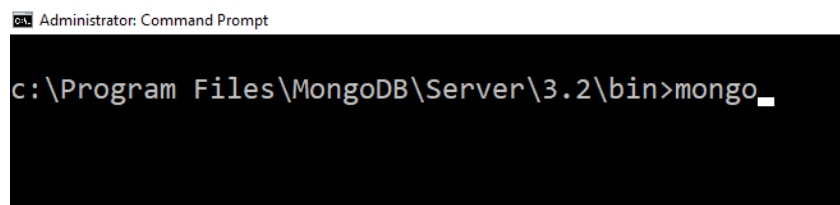
C:\Program Files\MongoDB\Server\3.6\bin\ mongod --conf mongod.cfg

It shows that mongod is listening at port 27017

2018-06-06T11:25:53.688+0800 I NETWORK [initandlisten] waiting for connections on port 27017

1.5 Create Databases and Collection using MongoDB Client

Run mongo command in command line to connect MongoDB server running at port 27017.

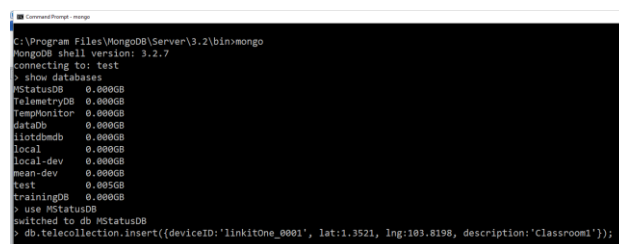


```
Administrator: Command Prompt
c:\Program Files\MongoDB\Server\3.2\bin>mongo_
```

Show current databases available in MongoDB with show databases command. The command use DATABASE_NAME will logically create a new database if it doesn't exist, otherwise it will return the existing database.

The logically created database can be seen once you insert collection and the document into it. Once again, new collection is created if it doesn't exist while issuing insert command.

- > show databases;
- > use MStatusDB
- > db.telecollection.insert({deviceID:'linkitOne_0001', lat:1.3521, lng:103.8198, description:'Classroom1'});



```
Administrator: Command Prompt
c:\Program Files\MongoDB\Server\3.2\bin>mongo
MongoDB shell version: 3.2.7
connecting to: test
> show databases
  MStatusDB   0.000GB
  TelemetryDB 0.000GB
  TempMonitor 0.000GB
  datab      0.000GB
  liotdbmb   0.000GB
  local      0.000GB
  local-dev  0.000GB
  mean-dev   0.000GB
  test       0.000GB
  trainingDB 0.000GB
> use MStatusDB
switched to db MStatusDB
> db.telecollection.insert({deviceID:'linkitOne_0001', lat:1.3521, lng:103.8198, description:'Classroom1'})
```

MongoDB Shell Lab Practice – Basic Commands

1) Create a database named “mydb”

```
> use mydb
```

2) Show all databases

```
> show dbs
```

3) Show currently selected database.

```
> db
```

4) Create a collection named “emp”.

```
> db.createCollection("emp")
```

5) Show all collections of the selected database.

```
> show collections
```

6) Insert following documents in emp collection:

```
> db.emp.insert([
{ empno : 1,ename : "Sachin",sal : 60000,desige : "Manager",dept : "Purchase" },
{ empno : 2,ename : "Kohali",sal : 50000,desige : "Manager",dept : "Sales" }
])
```

7) Using the Find() function. List all documents.

```
> db.emp.find()
```

8) List all documents with formatted output.

```
> db.emp.find.pretty()
```

9) List the document of an employee whose name is “Sachin”

```
> db.emp.find({ename:"Sachin"})
```

10) List the documents of employee whose salary is less than 30000

```
> db.emp.find({sal : {$lt : 30000}})
```

11) List employees whose designation is manager and department is sales.

```
> db.emp.find( { desige : "Manager",dept : "Sales"})
```

12) List employees whose salary is less than 50000 and designation is manager or department is admin.

```
> db.emp.find(
{sal : {$lt : 50000},
$or : [
{ design : "Manager"},{dept : "Admin"}
]})
```

13) Arrange the records by name in descending order.

```
> db.emp.find().sort({ename : -1})
```

14) List first 3 documents of emp collection.

```
> db.emp.find().limit(3)
```

15) Skip first 3 documents of emp collection.

```
> db.emp.find().skip(3)
```

16) List 3rd and 4th documents of emp collection.

```
> db.emp.find().limit(2).skip(2)
```

17) Count no. of employees.

```
> db.emp.find().count()
```

18) List distinct designation.

```
> db.emp.distinct("design")
```

2) NodeJS Application and MongoDB

NodeJS is a javascript based server side network computing services. It runs all process in single-threaded with non-blocking or asynchronous mode. As an asynchronous event driven JavaScript runtime, Node is designed to build scalable network applications. It can handle many connections concurrently. Upon each connection the callback is fired and ready accept next call, but if there is no work to be done, Node will sleep.

Development of NodeJS service application starts from creation of Node project via Node Package Manager. NodeJS can add and manage external libraries and modules using javascript package manager, Node Package Manager (npm). NPM is installed together with NodeJS <https://nodejs.org/en/download/>.

To use application with Express framework, it is advisable to create your own application. For a clean start, you create empty folder and install required library framework using npm tool.

2.1 Download and install NodeJS <https://nodejs.org/en/download/>

2.2 To test your NodeJS is properly installed, run following commands in NodeJS prompt.

```
node -v
```

```
v6.2.2
```

To develop NodeJS application, it is advisable to use a specific package or folder for the application. NodeJS application package include application package information stored in `package.json` and node modules or APIs eg. MongoDB, MQTT that required for the application.

NodeJS application can be created as module or service to share with other users.

2.3 Run following commands via nodejs command prompt, it will create NodeJS application called "nodejs_app"

```
d:\IIOT_Lab>mkdir nodejs_app  
d:\IIOT_Lab>cd nodejs_app  
d:\IIOT_Lab\nodejs_app>
```

2.4 Use the npm init command to create a package.json file for your application

```
d:\IIOT_Lab\nodejs_app>npm init
```

By following steps, package.json file is created in project folder.

```
npm
C:\>mkdir nodejs_test

C:\>cd nodejs_test

C:\nodejs_test>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (nodejs_test) nodejs_app1
version: (1.0.0)
description: First nodejs application
entry point: (index.js)
test command:
git repository:
keywords:
author: student
license: (ISC) _
```

Example package.json file:

```
{
  "name": "nodejs_app1",
  "version": "1.0.0",
  "description": "First nodejs application", "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "student",
  "license": "ISC"
}
```

2.5 Make the MongoDB driver and its dependencies by executing the following npm command.

```
d:\IIOT_Lab\nodejs_app>npm install mongodb --save
```

2.6 To write the application, create javascript page called “data_ingestion.js” in any text editor eg. Notepad or Notepad++ and save file extension with js. Add following lines to create mongodb and MongoClient objects.

```
var mongodb = require('mongodb'); // includes mongoDB
var MongoClient = mongodb.MongoClient; //initialises the mongoDB client
```

2.7 Since NodeJS is non-blocking code, insert the persistence data while connecting to the MongoDB database server using with callback function named “setUpCollection” as follow:

Note:

- MongoDB is running at 127.0.0.1 on port 27017

```
var mongodbURI = 'mongodb://127.0.0.1:27017/MStatusDB';
mongodbClient.connect(mongodbURI, {useNewUrlParser: true}, setupCollection);
```

2.8 Upon successfully connection, it calls callback function "setupCollection" insert/update the data.

```
function setupCollection(err, mclient){
    if(err) throw err;

    db = mclient.db("MStatusDB");
    collection=db.collection("all_mdata");
    collection.insertOne(
    {
        "deviceId" : "1",
        "temperature" : 24.0,
        "humidity" : 79.0,
        "pressure" : 110.0,
        "vibration" : 25.5,
        "factoryid" : 1.0,
        "level" : 1.0,
        "datetime" : new Date()
    },
    function(err,doc) {
        if(err) {
            console.log("Insert fail");
        }
        else
            console.log("Insert successfully");
    }
    );
}
```

Additional Exercise 1

npm install mongodb --save

Following shows the database Connection using pure MongoDB driver:

```
var MongoClient = require('mongodb').MongoClient
MongoClient.connect('mongodb://localhost:27017/testDB', function (err, db) {
  if (err) throw err
  db.collection('allrecords').find().toArray(function (err, result) {
    if (err) throw err
    console.log(result)
  })
})
```

Mongo DB and Collections

Collections are the equivalent of tables in traditional databases and contain all your documents. A database can have many collections. So how do we go about defining and using collections. Well there are a couple of methods that we can use. Let's jump straight into code and then look at the code.

```
var MongoClient = require('mongodb').MongoClient
MongoClient.connect('mongodb://localhost:27017/trainingDB', function (err, db) {
  if(err) { return console.dir(err); }
```

```
var collection = db.collection('admission');
db.collection('admission', {strict:true}, function(err, collection) {});
});
```

```
var MongoClient = require('mongodb').MongoClient
MongoClient.connect('mongodb://localhost:27017/trainingDB', function (err, db) {
  if(err) { return console.dir(err); }
```

```
var collection = db.collection('admission');
var rec1 = {'adminnumber':'170000','barcode':'T10000','subject':'Digital
Communication'};
var rec2 = {'adminnumber':'170001','barcode':'T10001','subject':'Digital
Communication'};
```

```
collection.insert(rec1);
collection.insert(rec2, {w:1}, function(err, result) {});

});
```


Updating the data in mongodb collection

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

// Connect to the db
MongoClient.connect("mongodb://localhost:27017/trainingDB", function(err, db) {
  if(err) { return console.dir(err); }

  var collection = db.collection('admission');
  var rec3 = {'adminnumber':'170002','barcode':'T10000','subject':'Digital
Fundamentals 2'};

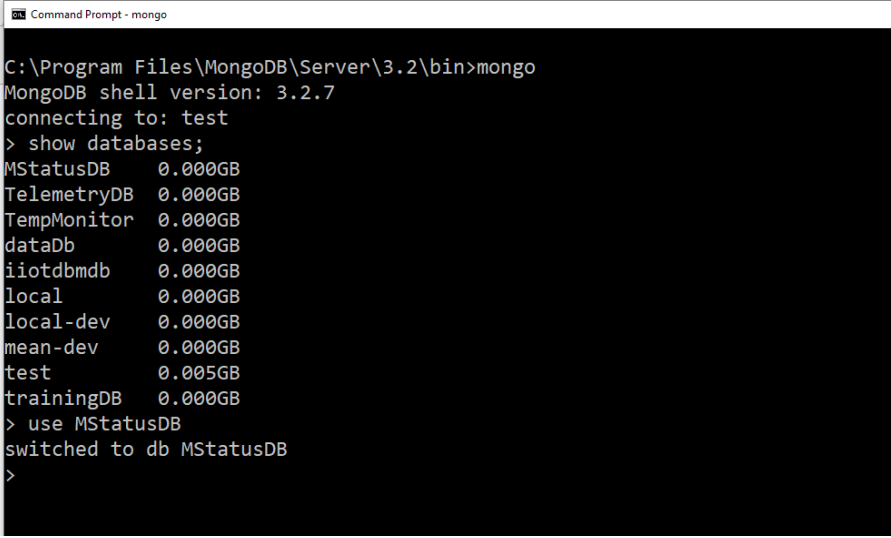
  collection.insert(rec3, {w:1}, function(err, result) {});
  collection.update(
    {'adminnumber':'170002'},
    {$set:{ subject:'Object Oriented Programming'}},
    {w:1},
    function(err, result) {}
  );
};
```

3) Display telemetry data using aggregation framework in MongoDB

Query using MongoDB command prompt to view data

CRUD operations in MongoDB perform on the collections of the documents. For example, read operations retrieves documents from a collection; i.e. queries a collection for documents. Update operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

The database we use is MStatusDB and the name of the collection is all_mdata.



```
Command Prompt - mongo
C:\Program Files\MongoDB\Server\3.2\bin>mongo
MongoDB shell version: 3.2.7
connecting to: test
> show databases;
MStatusDB      0.000GB
TelemetryDB    0.000GB
TempMonitor    0.000GB
dataDb         0.000GB
iiotdbmdb      0.000GB
local          0.000GB
local-dev      0.000GB
mean-dev       0.000GB
test           0.005GB
trainingDB     0.000GB
> use MStatusDB
switched to db MStatusDB
>
```

After connecting db, switch to MStatusDB and run following query to get the all data from the device 1.

```
db.all_mdata.find(
  {deviceId:'1' }
)
```

```
db.all_mdata.find(
{
  $and: [
    {deviceId:'1'}, {"temperature" : 33}
  ]
}
)
```

```
db.all_mdata.find(
{
  $and: [
    {deviceId:'1'}, {"temperature" : {$gt: 29}}
  ]
}
)
```

).pretty()

Additional Notes 1: NodeJS Application as Service to Retrieve Data from MongoDB

1. Run following commands via nodejs command prompt, it will create NodeJS application called "nodejs_app"

```
d:\IIOT_Lab>mkdir nodejs_app_service  
d:\IIOT_Lab>cd nodejs_app_service  
d:\IIOT_Lab\nodejs_app_service>
```

2. Use the npm init command to create a package.json file for your application

```
d:\IIOT_Lab\nodejs_app_service>npm init
```

By following steps, package.json file is created in project folder.

3. Make the MongoDB driver and its dependencies by executing the following npm command.

```
d:\IIOT_Lab\nodejs_app>npm install mongodb --save
```

4. When you run NodeJS application as service, application needs to define proper data type and uri. Use express light weight framework to handler uri mapping and data type handling during the response. By running following command to add or install express application framework in your NodeJS application

```
d:\IIOT_Lab\nodejs_app_service>npm install --save express
```

5. To write the application, create javascript page called "data_service.js" in any text editor eg. Notepad or Notepad++ and save file extension with js. Add following lines to create express, mongodb and MongoClient objects.

```
var express = require("express");  
var mongodb = require('mongodb'); // includes mongoDB  
var MongoClient = mongodb.MongoClient; //initialises the mongoDB client
```

6. Express service needs to serve or listen at an assigned port eg. 3300 with a predefined uri with http get method eg. <http://127.0.0.1:3300/alldata>. Following lines will prepare application with express framework uri alldata and server is listening at port 3300.

```
var app = express();  
app.get("/alldata", function(req, res){  
  getData(res);  
});  
app.listen("3300", function(){  
  console.log('Server up: http://localhost:3300');
```

```
});
```

7. Since NodeJS is non-blocking code, insert the persistence data while connecting to the Mongodb database server using with callback function named "setupCollection" as follow:

Note:

- Mongodb is running at 127.0.0.1 on port 27017

```
var mongodbURI = 'mongodb://127.0.0.1:27017/MStatusDB';
mongodbClient.connect(mongodbURI, {useNewUrlParser: true}, setupCollection);
```

8. Once application is successfully connected to MongoDB, it calls setupCollection callback function. The setupCollection is just to prepare db object to get ready.

```
function setupCollection(err, mclient){
  if(err) throw err;
  db = mclient.db("MStatusDB");
}
```

9. When user hit the address <http://127.0.0.1:3300/alldata> uri, application redirect to function getData(). It retrieves the data from MongoDB and return as json object. Add following getData() function.

```
function getData(responseObj){
  db.collection("all_mdata").find({}).limit(3).toArray(function(err, docs){
    if ( err ) throw err;
    responseObj.json(docs);
  });
}
```

Exercise 2

Create NodeJS express server which can connect to backend database MongoDB and create telemetryDB and collection devices. Server can handle post method with json data passed by the device_form.html. When user submit the device information NodeJS express service insert data into the collection devices. Use the previous sample to display the inserted devices.

Example device data in json format:

```
{
  "DeviceID": "f4:b8:5e:cc:63:b4",
  "Latitude": "1.274255",
  "Longitude": "103.799954",
  "Description": "GT_L102"
}
```

Declare MongoClient using mongodb library and adddeviceinfo function in middleware_utils.js

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

exports.adddeviceinfo = function(req, res, next)
{
  var res_data = JSON.parse(JSON.stringify(req.body));
  MongoClient.connect('mongodb://localhost:27017/telemetryDB', function (err,
db) {
    if(err) { return console.dir(err); }

    var collection = db.collection('devices');
    collection.insert(res_data)
    res.status(200).end(JSON.stringify(res_data ));
  });
}
```

Creating new device entry point with device_form.html and calling adddeviceinfo expressserver2.js

```
app.get('/newdevice', function (req, res) {
  res.sendFile(path.join(__dirname + '/device_form.html'));
})
app.post('/adddevice', mw.adddeviceinfo)
```

Data Aggregation in MongoDB

Data aggregation in MongoDB can be performed in single purpose, map-reduce and aggregation pipeline (aggregation framework) approaches.

Single purpose aggregation produce simple aggregation result on single collection.

In map-reduce operations have two phases: a *map* stage that processes each document and *emits* one or more objects for each input document, and *reduce* phase that combines the output of the map operation

Aggregation pipeline (aggregation framework) use data processing pipeline concept which is similar to the shell pipeline command. At each stage of pipeline (command), input documents produce output document(s) as the input for the preceding stage (command). There is a set of possible stages and each of those is taken as a set of documents as an input and produces a resulting set of documents. Documents enter multi-stages pipeline and process the documents into an aggregated result.

The most basic pipeline stages provide filtering, grouping and sorting documents by specific field or fields as well as tools for aggregating the contents of arrays, including arrays of documents. In addition, pipeline stages can use operators for tasks such as calculating the average or concatenating a string.

Aggregation for collection of database in MongoDB, following pipeline stages appear in an array.

\$match – Filter the amount of documents based on value and the result is given as input to the next stage.

\$group – Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.

\$sort – Reorders the document stream by a specified sort key. (1 ascending, -1 descending)

\$project – Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.

`$skip` – With this, it is possible to skip forward in the list of documents for a given amount of documents.

`$limit` – This limits the amount of documents to look at, by the given number starting from the current positions.

`$unwind` – Deconstructs an array field from the input documents to output a document for each element.

Accumulators (`$group`)

Accumulator expressions are available for use in the `$group` stage. Accumulators are operators that maintain their state (e.g. totals, maximums, minimums, and related data) as documents progress through the pipeline.

When used as accumulators in the `$group` stage, these operators take as input a single expression, evaluating the expression once for each input document, and maintain their state for the group of documents that share the same group key.

`$avg`: Returns an average of numerical values. Ignores non-numeric values.

`$sum`: Returns a sum of numerical values. Ignores non-numeric values.

`$max`: Returns the highest expression value for each group.

`$min`: Returns the lowest expression value for each group.

`$first`: Returns a value from the first document for each group. Order is only defined if the documents are in a defined order.

`$last`: Returns a value from the last document for each group. Order is only defined if the documents are in a defined order.

`$push`: Returns an array of expression values for each group.

`$addToSet`: Returns an array of unique expression values for each group. Order of the array elements is undefined.

`$stdDevPop`: Returns the population standard deviation of the input values.

`$stdDevSamp`: Returns the sample standard deviation of the input values.

Problem Statement:

One manufacturing site has collected sensors data from different machine in factories. Each machine has various sensors eg. pressure, temperature, vibration and so on. Data logger collect the data from sensors every 5 mins.

3.2. Download raw_data_script.txt from LMS. Insert the data into the collection named "all_mdata" in database named "MStatusDB". A total of 40 documents are inserted into collection.

```
db.all_mdata.insertMany([...])
```

3.3. There are data from two different factories. First to aggregate the average data for all sensors based on factory, you can write single stage with \$group and accumulator &avg. If you run following query in MongoDB shell, two summary documents will be return as follow:

Query:

```
db.all_mdata.aggregate([
  {"$group": {"_id": {"factoryid": "$factoryid"},
    "average temperature": {"$avg": "$temperature"},
    "average humidity": {"$avg": "$humidity"},
    "average pressure": {"$avg": "$pressure"},
    "average vibration": {"$avg": "$vibration"},
  }}
]).pretty()
```

The output:

```
{
  "_id": {
    "factoryid": 2
  },
  "average temperature": 24.509999999999998,
  "average humidity": 68.5,
  "average pressure": 90,
  "average vibration": 55.95
}
{
  "_id": {
    "factoryid": 1
  },
  "average temperature": 27.309999999999995,
  "average humidity": 76.55,
  "average pressure": 106.35,
  "average vibration": 55.95
}
```



```
}
```

3.4. If we want to filter by factory or by device, aggregation pipeline needs to be in multiple stages. For example, to know the maximum value of the sensors for the device number 2 in factory 1.

First combine \$match stage to filter documents of the factory 1 and \$group stage. Please take note that deviceid variable is needed to filter the document of the specific device. Hence in the \$group stage, device id need to be retrieved using \$first operator as shown.

```
db.all_mdata.aggregate([
  {"$match": {"factoryid": 1}},
  {"$group": {"_id": { "deviceId" : "$deviceId"},
    "max temperature": {"$max": "$temperature"},
    "max humidity ": {"$max": "$humidity"},
    "max pressure": {"$max": "$pressure"},
    "max vibration": {"$max": "$vibration"},
    "deviceid": { "$first": "$deviceId"}
  }},
]).pretty()
```

To sort the data in terms of device id, it can add \$sort stage with deviceid field.

```
 {"$sort": {"deviceid": 1}},
```

3.5. After retrieving the aggregated result of all sensors of the 5 devices in factory 1, filter the summarised document of the specific device eg. 2.

```
db.all_mdata.aggregate([
  {"$match": {"factoryid": 1}},
  {"$group": {"_id": { "deviceId" : "$deviceId"},
    "max temperature": {"$max": "$temperature"},
    "max humidity ": {"$max": "$humidity"},
    "max pressure": {"$max": "$pressure"},
    "max vibration": {"$max": "$vibration"},
    "deviceid": { "$first": "$deviceId"}
  }},
  {"$match": {"deviceid": "2"}}
]).pretty()
```

The Output:

```
{
  "_id" : {
```

```
        "deviceid" : "2"
    },
    "max temperature" : 28.5,
    "max humidity " : 87,
    "max pressure" : 90,
    "max vibration" : 68.5,
    "deviceid" : "2"
}
```

3.6. To display specific data of the result document, add the \$project stage in aggregation pipeline. For example, user want to view only the temperature and humidity, and deviceid data following \$project is appended to the existing aggregation pipeline.

```
{ "$project" : { _id: 0, "deviceid" : 1 , "max temperature" : 1, "max humidity" : 1 } }
```

The Output:

```
{ "max temperature" : 28.5, "max humidity" : 87, "deviceid" : "2" }
```

References:

Mead, A (2018). *Advanced Node.js Development* Packt Publishing Ltd

<https://nodejs.org/en/docs/>

<https://docs.mongodb.com/manual>