Advanced Database Project: Document Stores and MongoDB

Sivaporn Homvanish (0472422)

Tzu-Man Wu (0475596)

Table of contents

Background **3**
- Introduction of Database Management System 3
- SQL vs NoSQL 3
- Document Database 4

Introduction to MongoDB **4**
- What is MongoDB 4
- Key differences between SQL and MongoDB terminologies 4
- Installation 5
- Connect to MongoDB 6
- MongoDB Data Storage Structure 9
- Basics and CRUD Operations 10
  - Create operation 12
  - Read operation 13
  - Update operation 16
  - Delete operation 20

Implementation **21**
- Application Overview 21
- Environment setup 23
  - NodeJS with MongoDB Atlas (Cloud services) 23
  - Installation MongoDB NodeJS driver 23
  - Connect Application with MongoDB Atlas 24
- Function Design 28
  - Add product 28
  - Display products 29
  - Edit product 31
  - Delete product 33

Conclusion **34**

References **34**

1. Background

1.1. Introduction of Database Management System

Database Management System (DBMS) is a software package for managing database. It provides several kinds of operations such as create, retrieve, update data including managing data manipulation. The DBMS essential serves as an interface that bridge between end users or applications with a database to ensure data integrity and consistency.

1.2. SQL vs NoSQL

SQL stands for Structured Query Language. It is a standard language for relational database management and data handling. It allows manipulating structured data whose entities/variables are associated with certain relations.

NoSQL stands for Non-Structured Query Language. It is designed to deal with huge and intensive data that have various demands for modern applications such as different data structures, customization, complex real-time data, etc. It combines a wide variety of different database technologies to support nowadays technologies.

Features	SQL	NoSQL
Type	Table-based database	Various types such as ● Document-based database ● Key-value pairs ● Graph database ● Wide-column stores
Scaling	Vertical scaling, it scales by a power of its hardware	Horizontal scaling, it scales by increasing servers in the pool resources to reduce the load
Flexibility	Fix schemas which identified since predefine phase	High flexibility due to dynamic schemas
ACID Compliance	Comply with ACID	Sacrifice ACID compliance for flexibility and performance
Examples	● MySql ● Oracle ● Sqlite ● Postgres ● MS-SQL	● MongoDB ● Redis ● Hbase ● Neo4j ● CouchDB

1.3. Document Database

According to the need for unstructured data, the rapid growth of cloud computing and high demands of a requirement. Document database is introduced to loosen the restrictions on database schemas by using the document data model.

Key Advantages: [1]

- Independent document units help increase performance and distribute data across servers.
- Easy to apply application logic without translation between application and SQL queries.
- Support unstructured data that provide flexibility for data migration and usage.

2. Introduction to MongoDB

2.1. What is MongoDB

MongoDB is an open-source document database. It provides capability and flexibility of querying and indexing data. MongoDB is one of NoSQL database which is a schemaless data model that gives user suppleness to work on various data structures.

Below is the structure of MongoDB together with the example of the data format. This structure is totally different from a normal SQL query. However, it provides flexibility for various data formations.

2.2. Key differences between SQL and MongoDB terminologies

Dues to many types of databases, terminologies of each database are different. The following table presents the concept and several SQL terminologies that consistent with MongoDB concept and terminologies.

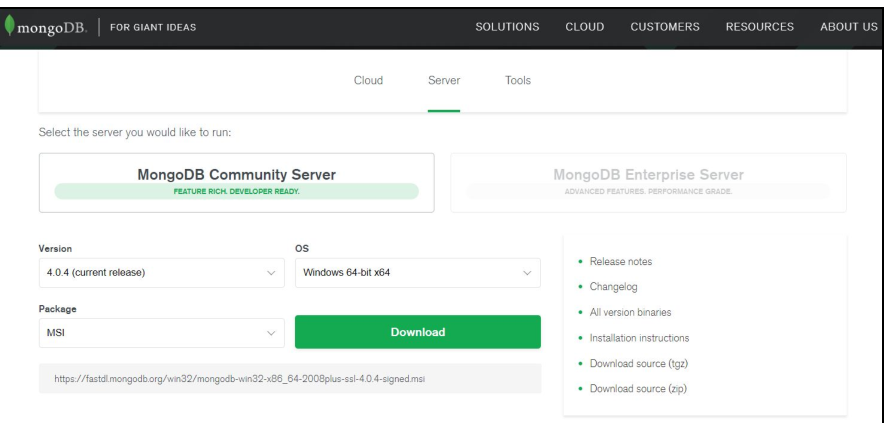
Comparison between SQL and MongoDB terminologies

SQL Terms	MongoDB Terms
Database	Database
Table	Collection
Row	Document
Column	Field
Joins	Embedded documents, linking

2.3. Installation

Install MongoDB

1. Download MongoDB installer (.msi) that compatible with your window version from https://www.mongodb.com/download-center. This report is used window x64 bits community edition.



2. Run MongoDB installer (.msi file) by navigating to the directory that stores the program and follows the wizard instruction.

Start MongoDB as a windows services

1. Open Command Prompt as an Administrator.
2. Create MongoDB database directory for storing data.
3. Change file path to the directory that you need to store database and create a data directory

```
C:\Users\ASUS>D:

D:\>cd D:\PALM-BDMA\BDMA-Homework\Advance DB\MongoDB\mongoDB_AdvDB

D:\PALM-BDMA\BDMA-Homework\Advance DB\MongoDB\mongoDB_AdvDB>md "\data\db"
```

4. Start MongoDB by running mongod.exe. You can point to your database directory by running command --dbpath following with directory path

```
D:\PALM-BDMA\BDMA-Homework\Advance DB\MongoDB\mongoDB_AdvDB>"D:\Program Files\MongoDB\Server\4.0\bin\mongod.exe"
--dbpath="D:\PALM-BDMA\BDMA-Homework\Advance DB\MongoDB\mongoDB_AdvDB\data\db"
```

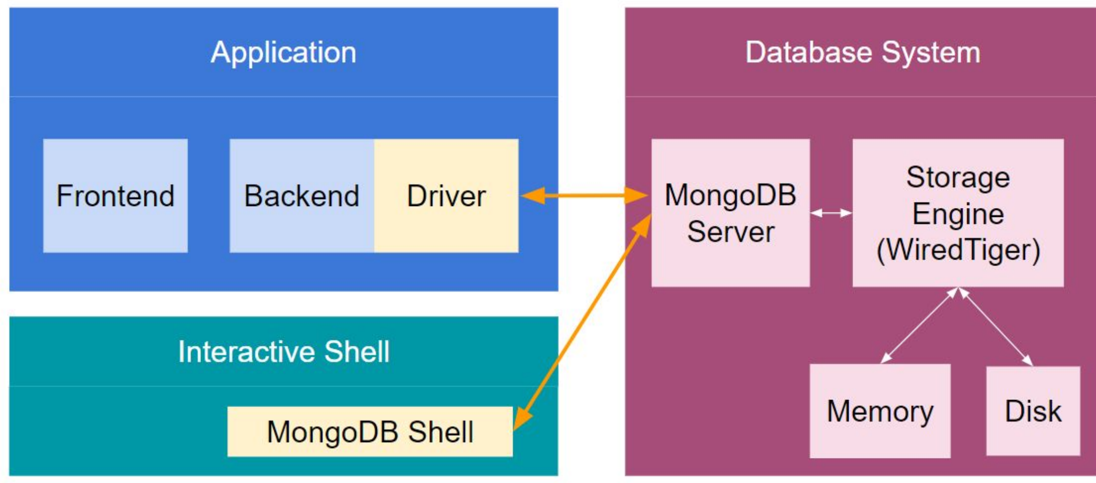
5. Open another 'Command Prompt' as an Administrator to connect to MongoDB by running **mongo.exe**

```
D:\PALM-BDMA\BDMA-Homework\Advance DB\MongoDB\mongoDB_AdvDB>"D:\Program Files\MongoDB\Server\4.0\bin\mongo.exe"
```

2.4. Connect to MongoDB

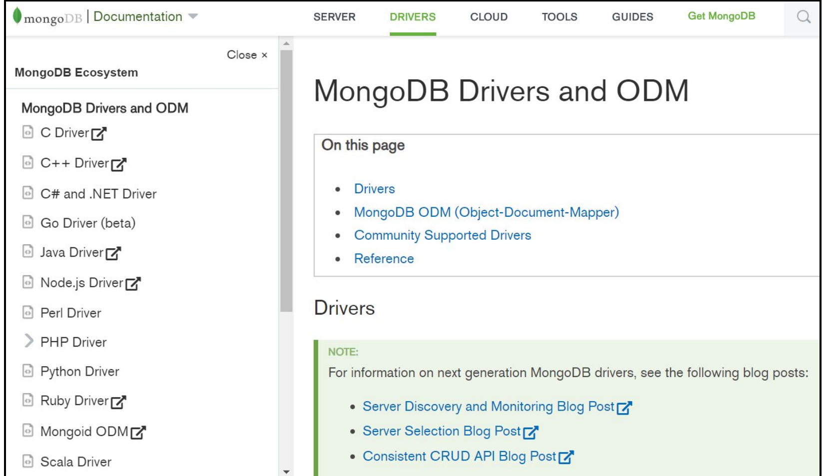
Overview

To connect to MongoDB Server, there are multiple choices to install the MongoDB client. The figure below shows two main method, drivers and shell. If users create their own application and desire to use MongoDB to manage data, they have to install a MongoDB driver which is matching their programming language. The driver will send the queries from backend server code to MongoDB server. On the other hand, users can also only use MongoDB shell to interact with MongoDB server. After MongoDB server receives the queries, it will communicate with Storage Engine. Here the MongoDB default Engine is WiredTiger. The storage engine can manage and work with data efficiently, so it handles all data access such as data-read and data-write with memory and disk.



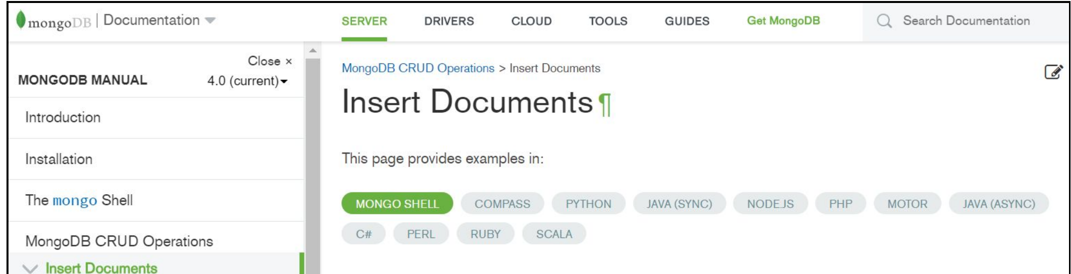
Drivers

For different programming languages, MongoDB provides different kinds of drivers to let application interact with MongoDB Server. After installing the driver, programmers can embed it inside their application code to link MongoDB server. Multiple MongoDB drivers can be found on the official MongoDB Docs page in the following link:
https://docs.mongodb.com/ecosystem/drivers/



Shell

From the MongoDB official website as below, there are multiple methods of different programming languages for users to be selected.[2]



Insert Command from Mongo Shell [2]

```
db.inventory.insertOne(
   { item: "canvas", qty: 100, tags: ["cotton"], size: { h: 28, w: 35.5, uom: "cm" } }
)
```

Insert Command from NodeJS [2]

```
await db.collection('inventory').insertOne({
  item: 'canvas',
  qty: 100,
  tags: ['cotton'],
  size: { h: 28, w: 35.5, uom: 'cm' }
});
```

Insert Command from PHP [2]

```
copy
$insertOneResult = $db->inventory->insertOne([
    'item' => 'canvas',
    'qty' => 100,
    'tags' => ['cotton'],
    'size' => ['h' => 28, 'w' => 35.5, 'uom' => 'cm'],
]);
```

Insert Command from JAVA [2]

```
copy
Document canvas = new Document("item", "canvas")
        .append("qty", 100)
        .append("tags", singletonList("cotton"));

Document size = new Document("h", 28)
        .append("w", 35.5)
        .append("uom", "cm");
canvas.put("size", size);

collection.insertOne(canvas);
```

The shell allows us to write queries which are very similar to the queries in different drivers. Using MongoDB shell is an easier way to learn and connect with MongoDB no matter which programming language is used. Although the syntax of programming languages are different, the core method to deal with data is similar. Therefore, we will use MongoDB shell which is connected to our local MongoDB server to introduce how to write commands in the upcoming part.

Start to run MongoDB server in the background of Windows

If you are a Windows user, after following 2.3 Installation, MongoDB Server will run automatically once you turn on your computer. There is another option that allows users to stop running MongoDB server and start running it again by Command Prompt.

1. Open Command Prompt and run as Administrator.
2. Type **net stop MongoDB** to stop MongoDB server
3. Open Command Prompt again
4. Type **mongod --dbpath** following with directory path to start MongoDB server
5. Keep the server running (Do not close)

```
C:\WINDOWS\system32>mongod --dbpath "D:\Program Files\MongoDB\Server\4.0\data\db"
2018-12-12T22:01:30.614+0100 I CONTROL  [main] Automatically disabling TLS 1.0, to force-enable TLS 1.0 specify --sslDis
abledProtocols 'none'
2018-12-12T22:01:31.222+0100 I CONTROL  [initandlisten] MongoDB starting : pid=19564 port=27017 dbpath=D:\Program Files\
MongoDB\Server\4.0\data\db 64-bit host=asus
2018-12-12T22:01:31.223+0100 I CONTROL  [initandlisten] targetMinOS: Windows 7/Windows Server 2008 R2
2018-12-12T22:01:31.223+0100 I CONTROL  [initandlisten] db version v4.0.4
2018-12-12T22:01:31.223+0100 I CONTROL  [initandlisten] git version: f288a3bdf201007f3693c58e140056adf8b04839
2018-12-12T22:01:31.224+0100 I CONTROL  [initandlisten] allocator: tcmalloc
2018-12-12T22:01:31.224+0100 I CONTROL  [initandlisten] modules: none
2018-12-12T22:01:31.224+0100 I CONTROL  [initandlisten] build environment:
2018-12-12T22:01:31.224+0100 I CONTROL  [initandlisten]     distmod: 2008plus-ssl
2018-12-12T22:01:31.224+0100 I CONTROL  [initandlisten]     distarch: x86_64
2018-12-12T22:01:31.224+0100 I CONTROL  [initandlisten]     target_arch: x86_64
2018-12-12T22:01:31.224+0100 I CONTROL  [initandlisten] options: { storage: { dbPath: "D:\Program Files\MongoDB\Server\4
.0\data\db" } }
2018-12-12T22:01:31.238+0100 I STORAGE  [initandlisten] Detected data files in D:\Program Files\MongoDB\Server\4.0\data\
db created by the 'wiredTiger' storage engine, so setting the active storage engine to 'wiredTiger'.
2018-12-12T22:01:31.238+0100 I STORAGE  [initandlisten] wiredtiger_open config: create,cache_size=3535M,session_max=2000
0,eviction=(threads_min=4,threads_max=4),config_base=false,statistics=(fast),log=(enabled=true,archive=true,path=journal
```

Use Mongo Shell to connect to MongoDB Server

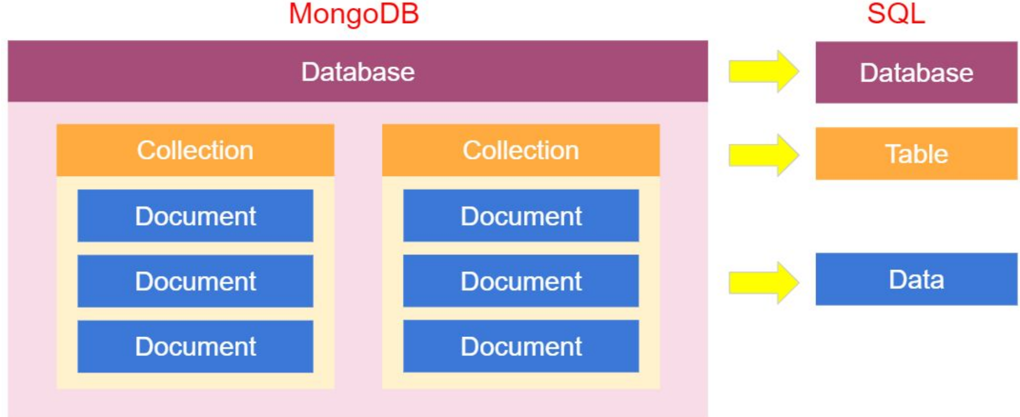
1. Open another Command Prompt
2. Type mongo

```
C:\WINDOWS\system32>mongo
MongoDB shell version v4.0.4
connecting to: mongodb://127.0.0.1:27017
Implicit session: session { "id" : UUID("1f82f89c-0411-46d8-8913-918e06e81419") }
MongoDB server version: 4.0.4
Server has startup warnings:
2018-12-15T22:44:48.110+0100 I CONTROL  [initandlisten]
```

2.5. MongoDB Data Storage Structure

Database & Collection & Documents

MongoDB as a NoSQL document-based database has a different data storage structure from SQL. From the figure below, in MongoDB, users can have multiple databases. Each database can have multiple collections and each collection can have multiple documents. Compared to SQL, collection in MongoDB equivalent to a table in SQL, and documents equal several rows in a SQL table.



JSON Document

MongoDB uses JSON documents to store its data. There is an example of a JSON document below. The document is always surrounding by curly brackets. Inside the curly brackets, it stores fields which each field consists of a key and value. The key is the name of an attribute and it is normally put inside quotation marks(optional). Value can be different kinds of data types such as string, number, boolean, array, object, and even another document. In other words, we can have nested documents inside a document. For the value, quotation marks are used depending on your data type.

```
{
    "_id" : ObjectId("5c1508201e0007f2dfc39d65"),      → Field
    "name" : "The Lord of the Rings",
    "price" : 10,
    "in_stock" : true,
    "store" : [
        {   "name" : "paris_1",
            "address" : {
                "zipcode" : "105",
                "street" : "Main street" }
        },
        {      "name" : "paris_2",
            "address" : {
                "zipcode" : "107",
                "street" : "2nd street" }                → Embedded Document
        }
    ]
}
```

Key

Value

2.6. Basics and CRUD Operations

CRUD stands for **C**reate, **R**ead, **U**pdate and **D**elete operations. These are fundamental operations that use to manage documents of the collection in MongoDB.

After finish installation and start MongoDB server (section 2.4), you can use CRUD operations for query or editing data in the database. To see other useful commands apart from CRUD, you can type **help** to see more information.

```
> help
        db.help()                    help on db methods
        db.mycoll.help()             help on collection methods
        sh.help()                    sharding helpers
        rs.help()                    replica set helpers
        help admin                   administrative help
        help connect                 connecting to a db help
        help keys                    key shortcuts
        help misc                    misc things to know
        help mr                      mapreduce

        show dbs                     show database names
        show collections             show collections in current database
        show users                   show users in current database
        show profile                 show most recent system.profile entries with time >= 1ms
        show logs                    show the accessible logger names
        show log [name]              prints out the last segment of log in memory, 'global' is default
        use <db_name>                set current database
        db.foo.find()                list objects in collection foo
        db.foo.find( { a : 1 } )     list objects in foo where a == 1
        it                           result of the last line evaluated; use to further iterate
        DBQuery.shellBatchSize = x   set default number of items to display on shell
        exit                         quit the mongo shell
```

The following command below is the frequency used commands:

- show dbs
    - List all of the databases in the MongoDB server
- show collections
    - List all of the collections in the MongoDB server
- use <db_name>
    - Select a database to be used

Below are the CRUD syntax and structure. [3]

In the following part, we will use a simple example to show how to use CRUD operations. We create a database named **sample** and a collection named **bookstoreproduct** to store the information of the books we sell.

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> show dbs
admin   0.000GB
local   2.758GB
shop    0.000GB
MongoDB Enterprise Cluster0-shard-0:PRIMARY> use sample
switched to db sample
```

2.6.1. Create operation

To create data into the database, MongoDB provides **insertOne** and **insertMany** command as below depending on how many data you intend to create. For these two functions, you can put two kinds of arguments. The first one is your data, and the second one is writeConcern for optional use. The writeConcern argument allows you to set many extra conditions such as the timeout option to specify a time limit to prevent write operations from blocking indefinitely and another option to request acknowledgment that the write operation has been written to the on-disk journal. However, if we use only data argument and omit the writeConcern, MongoDB will assign writeConcern default value to that command automatically. [2]

Create collection methods	Description
insertOne(<data>,<options>)	Inserts a single document into a collection.
insertMany(<data>,<options>)	Inserts multiple documents into a collection.

Now, we try to use **insertOne** command to insert one data into our bookstoreproduct collection.

Command:

```
db.bookstoreproduct.insertOne(
    {   "name": "The Lord of the Rings",
        "price": 10,
        "in_stock": true,
        "store":
                [
                    {
                        "name": "paris_1",
                        "address":
                                    {   "zipcode": "105",
                                        "street": "Main street"
                                    }
                    },
                    {
                        "name": "paris_2",
                        "address":
                                    {   "zipcode": "107" ,
                                        "street": "2nd street"
                                    }
                    }
                ]
    }
)
```

After executing the command, we get the result as below. If inserting is completed, we will receive a unique automatically generating id for the inserted data. The id is related to the order of the data you insert, and it can be changed later by the users as well.

Result:

```
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5c14d49b1a346021e5c4ed63")
}
```

Below is the data we just inserted:

```
{
        "_id" : ObjectId("5c14d49b1a346021e5c4ed63"),
        "name" : "The Lord of the Rings",
        "price" : 10,
        "in_stock" : true,
        "store" : [
                {
                        "name" : "paris_1",
                        "address" : {
                                "zipcode" : "105",
                                "street" : "Main street"
                        }
                },
                {
                        "name" : "paris_2",
                        "address" : {
                                "zipcode" : "107",
                                "street" : "2nd street"
                        }
                }
        ]
}
```

Next, we try to use **insertMany** command to insert more than one data at one time into our bookstoreproduct collection. For this operation, all required documents for inserting should create as a 1 object. So, the syntax must be covered with **square blankets**. For example, db.bookstoreproduct.insertMany(**[**<document>,<document>,... **]**).

```
db.bookstoreproduct.insertMany( [
    { "name": "TEST 1", "price": 59, "in_stock": false, "store": [{"name": "ULB", "address": {"zipcode":"1050", "street":"ULB street"}}]},
    { "name": "TEST 2", "price": 59, "in_stock": false, "store": [{"name": "ULB", "address": {"zipcode":"1050", "street":"ULB street"}}]},
    { "name": "TEST 3", "price": 59, "in_stock": false, "store": [{"name": "ULB", "address": {"zipcode":"1050", "street":"ULB street"}}]}
 ] )
```

2.6.2. Read operation

To read data from the database, MongoDB provides **find** and **findOne** command as below depending on how many data you intend to show. For these two functions, you can put two kinds of arguments. The first one is your filter condition using query operators to return the matching documents in a collection, and the second one is the fields you intend to return in the matching documents.

Read collection methods	Description
find(<filter>,<options>)	Return a cursor object(allow us to cycle through the results) which only show the first 20 documents by default at one time.
findOne(<filter>,<options>)	Only return the first matching document in the collection based on the filter.

Example: find

Command:

```
db.bookstoreproduct.find()
```

Result:

It will only show the first 20 documents by default because if it always returns all the data, it will take too long if we have a million documents. By typing **it**, MongoDB will use the find's cursor to fetch and display the next bunch of data on the screen.[3]

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.bookstoreproduct.find()
{ "_id" : ObjectId("5c14d49b1a346021e5c4ed63"), "name" : "The Lord of the Rings", "price" : 10,
"in_stock" : true, "store" : [ { "name" : "paris_1", "address" : { "zipcode" : "105", "street" :
 "Main street" } }, { "name" : "paris_2", "address" : { "zipcode" : "107", "street" : "2nd stree
t" } } ] }
{ "_id" : ObjectId("5c1554054ca2136cc73bb8a7"), "name" : "TEST 1", "price" : 59, "in_stock" : fa
lse, "store" : [ { "name" : "ULB", "address" : { "zipcode" : "1050", "street" : "ULB street" } }
 ] }
{ "_id" : ObjectId("5c1554054ca2136cc73bb8a8"), "name" : "TEST 2", "price" : 20, "in_stock" : tr
ue, "store" : [ { "name" : "UPC", "address" : { "zipcode" : "167", "street" : "UPC street" } } ]
 }
{ "_id" : ObjectId("5c1554054ca2136cc73bb8a9"), "name" : "TEST 3", "price" : 35, "in_stock" : tr
ue, "store" : [ { "name" : "TUE", "address" : { "zipcode" : "2080", "street" : "TUE street" } }
] }
```

To see the result in a well-format, we can use .pretty() method. It will show the data in the database that easier to read.

Command:

```
db.bookstoreproduct.find().pretty()
```

Result:

```
{
        "_id" : ObjectId("5c14d49b1a346021e5c4ed63"),
        "name" : "The Lord of the Rings",
        "price" : 10,
        "in_stock" : true,
        "store" : [
                {
                        "name" : "paris_1",
                        "address" : {
                                "zipcode" : "105",
                                "street" : "Main street"
                        }
                },
                {
                        "name" : "paris_2",
                        "address" : {
                                "zipcode" : "107",
                                "street" : "2nd street"
                        }
                }
        ]
}
```

Sometimes we just want to find a subset of our data, so we can use a filter to fetch specific documents. There is an example below to show how to use find() to get the information of books which price is greater than 25. Here we use a reserved operator of MongoDB **$gt** which means "greater than".

Command:

```
db.bookstoreproduct.find({price: {$gt: 25}})
```

Result:

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.bookstoreproduct.find({price: {$gt: 25}}).pretty()
{
        "_id" : ObjectId("5c1554054ca2136cc73bb8a7"),
        "name" : "TEST 1",
        "price" : 59,
        "in_stock" : false,
        "store" : [
                {
                        "name" : "ULB",
                        "address" : {
                                "zipcode" : "1050",
                                "street" : "ULB street"
                        }
                }
        ]
}
{
        "_id" : ObjectId("5c1554054ca2136cc73bb8a9"),
        "name" : "TEST 3",
        "price" : 35,
        "in_stock" : true,
        "store" : [
                {
                        "name" : "TUE",
                        "address" : {
                                "zipcode" : "2080",
                                "street" : "TUE street"
                        }
                }
        ]
}
```

To get the name of books which price is greater than 25, we add the second argument {name: 1} which means only get the key "name" information of the document.

Command:

```
db.bookstoreproduct.find({price: {$gt: 25}}, {name: 1})
```

Result:

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.bookstoreproduct.find({price: {$gt: 25}}, {name: 1}).pretty()
{ "_id" : ObjectId("5c1554054ca2136cc73bb8a7"), "name" : "TEST 1" }
{ "_id" : ObjectId("5c1554054ca2136cc73bb8a9"), "name" : "TEST 3" }
```

To only return the first matching document in the collection based on the filter.

Example: findOne

Command:

```
db.bookstoreproduct.findOne({price: {$gt: 25}}, {name: 1})
```

Result:

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.bookstoreproduct.findOne({price: {$gt: 25}}, {name: 1})
{ "_id" : ObjectId("5c1554054ca2136cc73bb8a7"), "name" : "TEST 1" }
```

2.6.3. Update operation

Update operator is used for modifying and adding extra data to the database. MongoDB provides **updateOne, updateMany,** and **replaceOne** commands to select documents that needed to be updated. There are 3 arguments, the first one is your filter condition using query operators to return the matching documents in a collection, the second one is information that you need to update over the existing value. The last one is the option for your command. Below are the syntax and structures of update operation.

Update collection methods	Description
updateOne(<filter>,<data>,<options>)	Update a first single document in the collection based on the filter.
updateMany(<filter>,<data>,<options>)	Update all document in the collection based on the filter.
replaceOne(<filter>,<data>,<options>)	Replace a first single document in the collection based on the filter.

Operators for filter update operation

Operation names	Description
$set	Replace the value of the field with a specific value of the operation. If the field does not exist, $set will add a new field to the document. This operation has the following form: { $set: { <field1>: <value1>, ... } }

$min	Update value while the specified value of the operation is less than the current value of the field. If specified field does not exist, $min will set the field to the specified value in the operation. This operation has the following form: { $min: { <field1>: <value1>, ... } }
$max	Update value while the specified value of the operation is greater than the current value of the field. If specified field does not exist, $max will set the field to the specified value in the operation. This operation has the following form: { $max: { <field1>: <value1>, ... } }
$inc	Increment the value of the field with the specified value of the operation. This operation has the following form: { $inc: { <field1>: <amount1>, <field2>: <amount2>, ... } }
$rename	Update the name of the field with specified value of the operation. This operation has the following form: {$rename: { <field1>: <newName1>, <field2>: <newName2>, ... } }

Example: updateOne

This is current data in the database.

```
{
        "_id" : ObjectId("5c1509931e0007f2dfc39d66"),
        "name" : "Smuffs and Friends comics ",
        "price" : 19,
        "in_stock" : true,
        "store" : [
                {
                        "name" : "brussels_1",
                        "address" : {
                                "zipcode" : "906",
                                "street" : "Main street"
                        }
                },
                {
                        "name" : "brussels_2",
                        "address" : {
                                "zipcode" : "912",
                                "street" : "2nd street"
                        }
                }
        ]
}
```

Command:

```
db.bookstoreproduct.updateOne({_id : ObjectId("5c1509931e0007f2dfc39d66")},
{$set: {name : "Smurfs and Friends Comics"}})
```

Result:

Book's name has been updated from Smuffs to Smurf.

```
{
        "_id" : ObjectId("5c1509931e0007f2dfc39d66"),
        "name" : "Smurfs and Friends Comics",
        "price" : 19,
        "in_stock" : true,
        "store" : [
                {
                        "name" : "brussels_1",
                        "address" : {
                                "zipcode" : "906",
                                "street" : "Main street"
                        }
                },
                {
                        "name" : "brussels_2",
                        "address" : {
                                "zipcode" : "912",
                                "street" : "2nd street"
                        }
                }
        ]
}
```

Example: updateMany

We have inserted 3 books to the collection. All of the books price are set to 59. In this example, we will use updateMany() to update the prices of the books to 69.

Below is the inserted books information.

```
db.bookstoreproduct.insertMany( [
   { "name": "TEST 1", "price": 59, "in_stock": false, "store": [{"name": "ULB", "address": {"zipcode":"1050", "street":"ULB street"}}]},
   { "name": "TEST 2", "price": 59, "in_stock": false, "store": [{"name": "ULB", "address": {"zipcode":"1050", "street":"ULB street"}}]},
   { "name": "TEST 3", "price": 59, "in_stock": false, "store": [{"name": "ULB", "address": {"zipcode":"1050", "street":"ULB street"}}]}
 ] )
```

Command:

We used reserved operator $regex to find all book names that contain "TEST" inside. Then, we updated the price to 69.

```
db.bookstoreproduct.updateMany( {"name": {$regex: /TEST/}},{$set: {"price": 69}})
```

Result:

```
{
        "_id" : ObjectId("5c153d251e0007f2dfc39d68"),
        "name" : "TEST 1",
        "price" : 69,
        "in_stock" : false,
        "store" : [
                {
                        "name" : "ULB",
                        "address" : {
                                "zipcode" : "1050",
                                "street" : "ULB street"
                        }
                }
        ]
}
{
        "_id" : ObjectId("5c153d251e0007f2dfc39d69"),
        "name" : "TEST 2",
        "price" : 69,
        "in_stock" : false,
        "store" : [
                {
                        "name" : "ULB",
                        "address" : {
                                "zipcode" : "1050",
                                "street" : "ULB street"
                        }
                }
        ]
}
{
        "_id" : ObjectId("5c153d251e0007f2dfc39d6a"),
        "name" : "TEST 3",
        "price" : 69,
        "in_stock" : false,
        "store" : [
                {
                        "name" : "ULB",
                        "address" : {
                                "zipcode" : "1050",
                                "street" : "ULB street"
                        }
                }
        ]
}
```

Example: replaceOne

Replace the document details of the first book which contains “TEST” in its name.

Command:

```
db.bookstoreproduct.replaceOne( {"name": {$regex: /TEST/}}, {"price": 49} )
```

Result:

From the previous example result of updateMany(). Book name “TEST 1” has ObjectID(5c153d251e0007f2dfc39d68). However, when we use replaceOne() operation. It will replace all the document. So, the current document of ObjectID(5c153d251e0007f2dfc39d68) structure will be changed to {“price: 49”} based on replacement command that was run.

```
{ "_id" : ObjectId("5c153d251e0007f2dfc39d68"), "price" : 49 }
{
        "_id" : ObjectId("5c153d251e0007f2dfc39d69"),
        "name" : "TEST 2",
        "price" : 69,
        "in_stock" : false,
        "store" : [
                {
                        "name" : "ULB",
                        "address" : {
                                "zipcode" : "1050",
                                "street" : "ULB street"
                        }
                }
        ]
}
{
        "_id" : ObjectId("5c153d251e0007f2dfc39d6a"),
        "name" : "TEST 3",
        "price" : 69,
        "in_stock" : false,
        "store" : [
                {
                        "name" : "ULB",
                        "address" : {
                                "zipcode" : "1050",
                                "street" : "ULB street"
                        }
                }
        ]
}
```

2.6.4. Delete operation

Delete operator is used to remove documents from a collection. There are 2 arguments, the first one is your filter condition using query operators to return the matching documents in a collection, the second one is the option for your command. Below are the syntax and structures of delete operation.

Update collection methods	Description
deleteOne(<filter>,<options>)	Delete a first single document from a collection in the database based on the filter
deleteMany(<filter>,<options>)	Delete all document from a collection in the database based on the filter

Example: deleteOne()

Currently, we have 3 documents in the database. We can show the number of collections by using .count() function.

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.bookstoreproduct.find().count()
3
```

Command:

```
db.bookstoreproduct.deleteOne({name:"TEST"})
```

Result:

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.bookstoreproduct.deleteOne({name:"TEST"})
{ "acknowledged" : true, "deletedCount" : 1 }
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.bookstoreproduct.find().count()
2
```

Example: deleteMany()

There are 5 documents in the database. We will delete all books that its name contains "TEST". From the previous example, there are only 2 books left which are "TEST 2" and "TEST 3" to be deleted

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.bookstoreproduct.find().count()
5
```

Command:

```
db.bookstoreproduct.deleteMany({name:{$regex: /TEST/}})
```

Result:

After running deleteMany(), Book name "TEST2" and "TEST 3" were deleted from the database.

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.bookstoreproduct.deleteMany({name:{$regex: /TEST/}})
{ "acknowledged" : true, "deletedCount" : 2 }
```

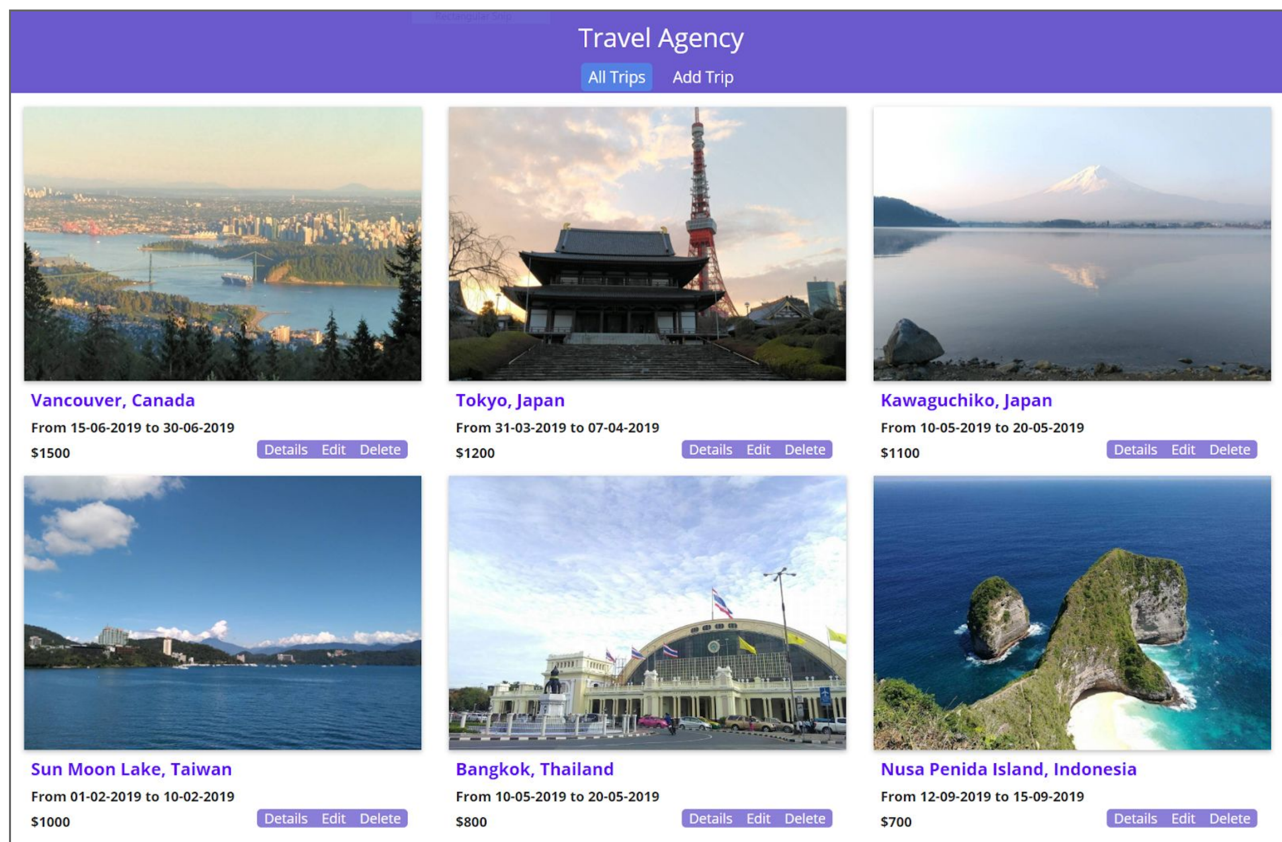
For more information about CRUD operations, you can refer to the official MongoDB site in the following link: https://docs.mongodb.com/manual/crud/

3. Implementation

3.1. Application Overview

In this project, we implemented 'Travel Agency' webpage to store data about the trips that the company provided to customers. Users can see trip lists, add more trips, deleted and modify trip details such as location, date, price, image, and detail. This website was implemented by Node.js together with MongoDB Altlas cloud database. More explanation will be explained in the following section.

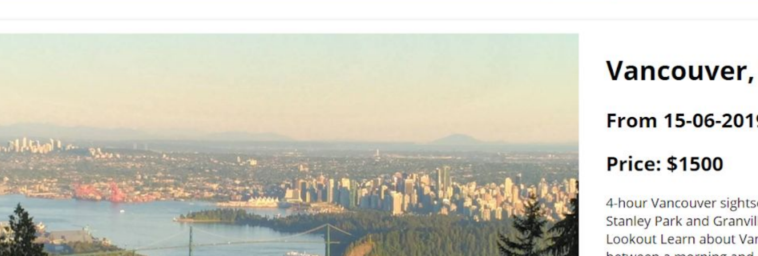
The main page of the website:



Details page:

Travel Agency

All Trips Add Trip



Vancouver, Canada

From 15-06-2019 to 30-06-2019

Price: $1500

4-hour Vancouver sightseeing tour Visit Vancouver's most famous sites including Gastown, Chinatown, Stanley Park and Granville Island Soak up 360-degree views of Vancouver from atop the Vancouver Lookout Learn about Vancouver's history, architecture and culture from an informative guide Choose between a morning and afternoon tour to suit your schedule Hotel pickup and drop-off included Opt to add tickets to the Vancouver Art Gallery or FlyOver Canada.

Add Trip page:

Travel Agency

All Trips Add Trip

Location

Start Date

DD-MM-YYYY

End Date

DD-MM-YYYY

Price

Image

Detail

Create Trip

3.2. Environment setup

3.2.1. NodeJS with MongoDB Atlas (Cloud services)

What is NodeJS?

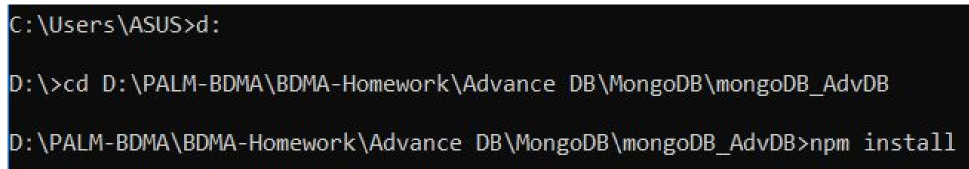
NodeJS is an open source run-time server environment that executes JavaScript code outside of a browser and it is compatible with various operating systems e.g. OS X, Microsoft, and Linux. There is an asynchronous feature for all APIs of NodeJS that help server get a response faster.[4][5]

What is MongoDB Atlas?

MongoDB Atlas is a could services that fully-managed database by handling complex configuration and helping users to seamlessly integrate their business with the newest database facilities. Atlas also provides a friendly user interface and API that helps users reduce database management time.[6]

3.2.2. Installation MongoDB NodeJS driver

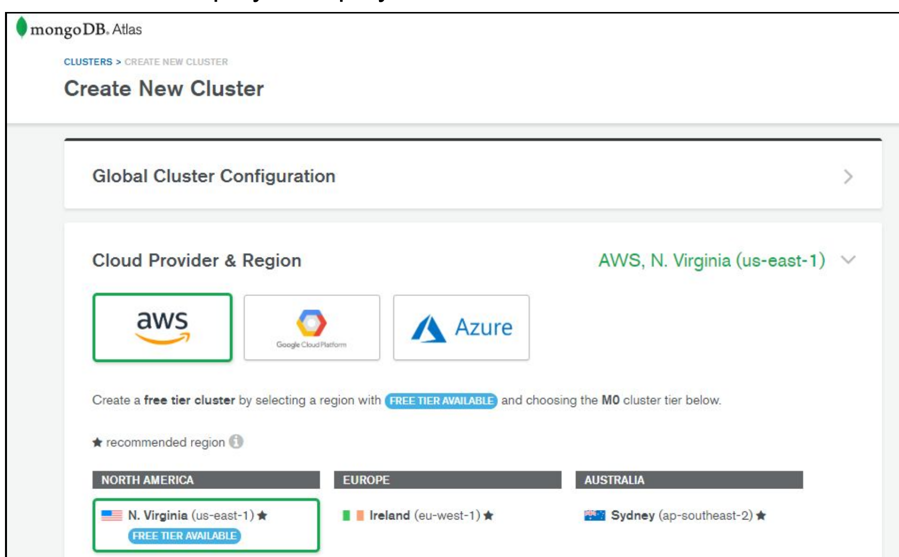
Download MongoDB NodeJS driver from https://mongodb.github.io/node-mongodb-native/ and use the command below to install. [7]



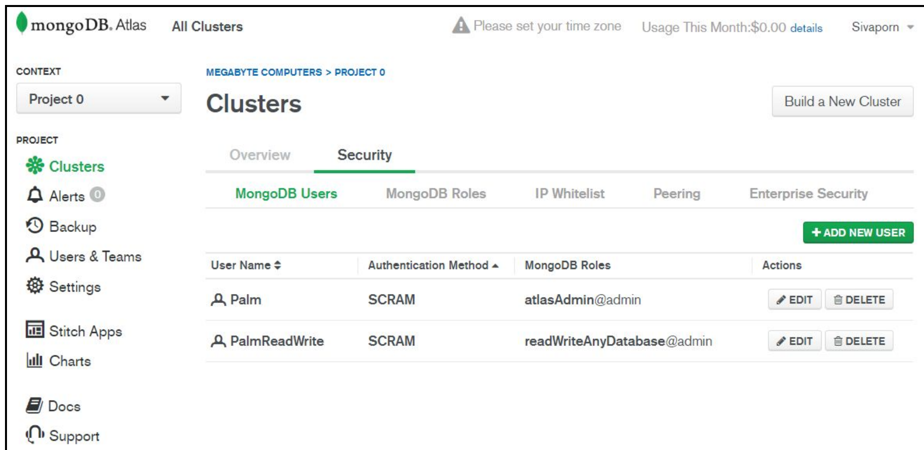
3.2.3. Connect Application with MongoDB Atlas

How to create MongoDB Atlas cluster? [3]

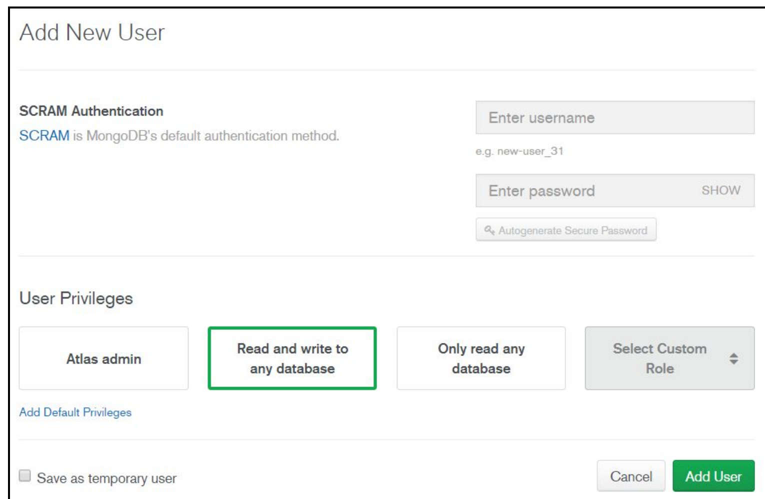
1. Create an Atlas user account by the following link: https://www.mongodb.com/cloud/atlas
2. Create Atlas Cluster
    a. Select 'Cloud Provider & Region'
    b. Select Cluster Tier
    c. Name your cluster
    d. Click 'Deploy'to deploy the cluster



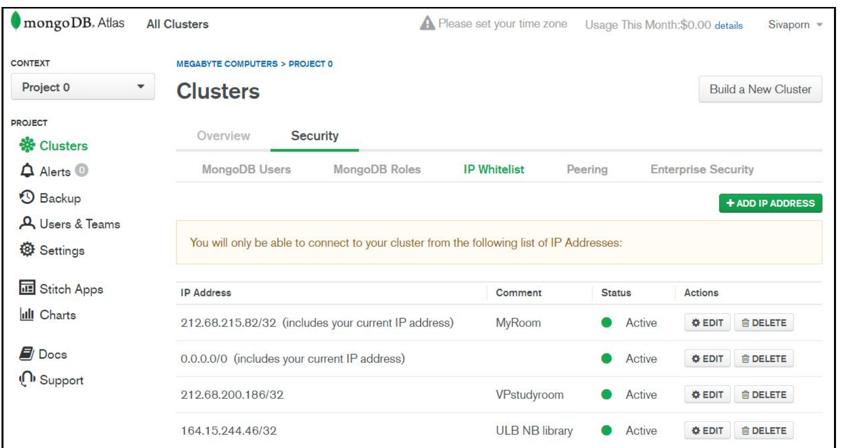
3. Configure security for the cluster
    a. Select 'Security'
    b. Go to 'MongoDB users' tab, click '+ Add new user'



    c. Enter username and password
    d. Select user privileges
    e. Click 'Add user'

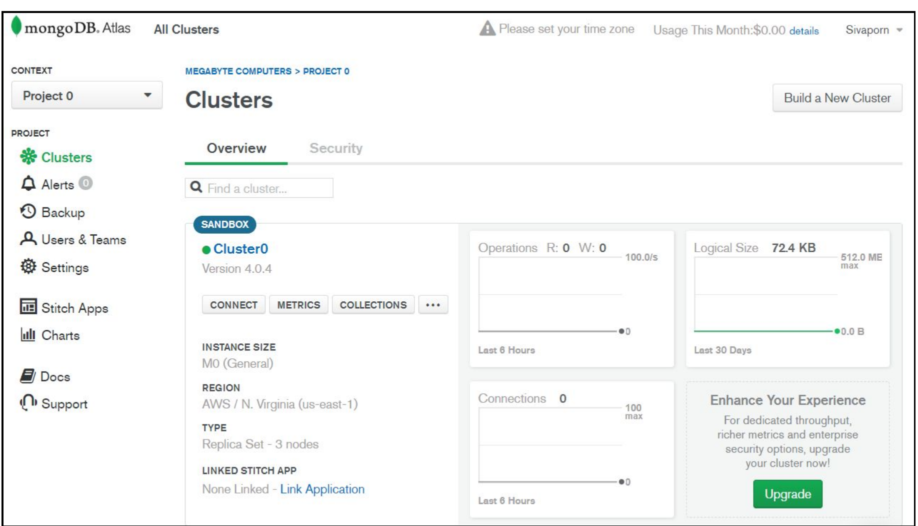


4. Go to 'IP Whitelist' tab, click '+ Add IP Address'

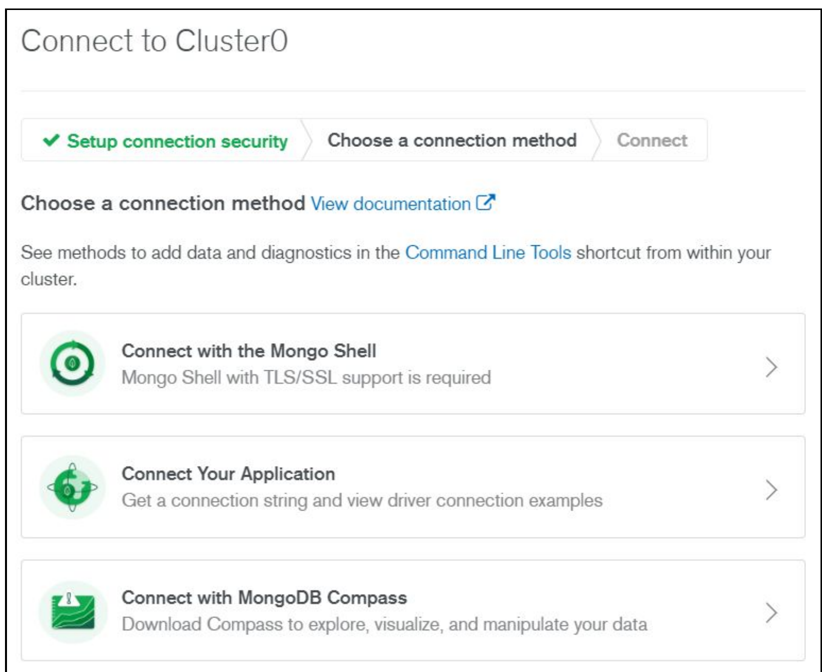


How to connect to MongoDB Atlas?

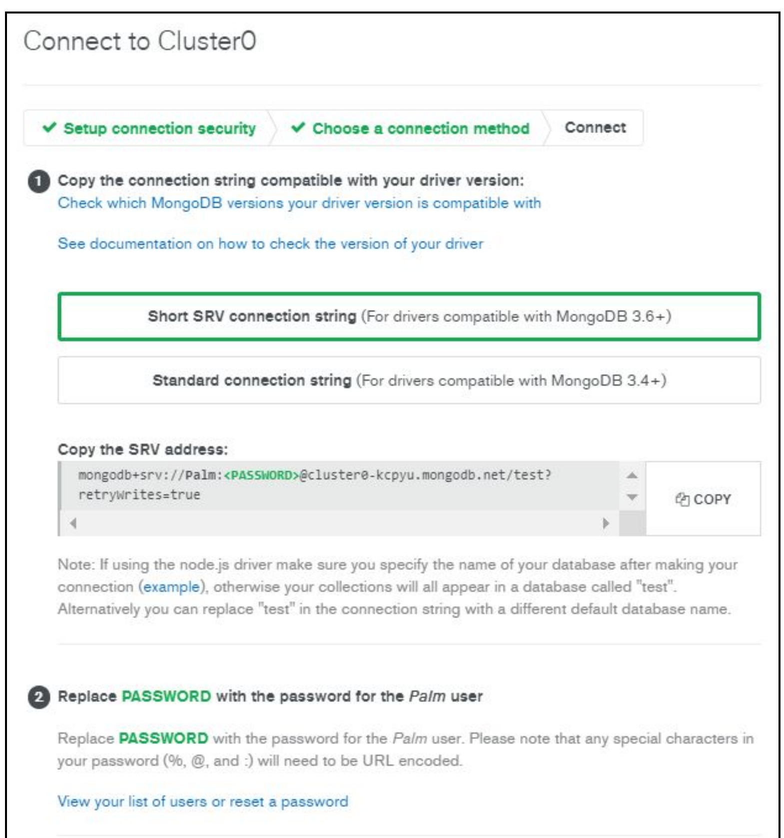
1. Select 'Overview' and click 'Connect'



2. Choose 'Connect Your Application'



3. Select ‘Short SRV connection string (shell 3.6+)’ and copy the SRV address



4. Replace <Password> for your user and specify the database name. Otherwise, your collection will be in default database named ‘test’

5. Connect Atlas cluster to NodeJS application

```
let _db;
const mongodb = require('mongodb');
const MongoClient = mongodb.MongoClient;
const mongoDbUrl =
'mongodb+srv://Palm:xxxxxxxxxxx@cluster0-v2sng.mongodb.net/shop?retryWrites=true';
  MongoClient.connect(mongoDbUrl)
    .then(client => {
      _db = client;
      callback(null, _db);
    })
    .catch(err => {
      callback(err);
    });
};
```

How to start the connection?

1. Open Command Prompt, go to your application directory and run the command 'npm start'. This command will call your application. However, you will get an error because the connection between application and Atlas cluster is not connected yet.

```
C:\Users\ASUS>d:

D:\>cd D:\PALM-BDMA\BDMA-Homework\Advance DB\MongoDB\mongoDB_AdvDB

D:\PALM-BDMA\BDMA-Homework\Advance DB\MongoDB\mongoDB_AdvDB>npm start

> mongodb-demo@0.1.0 start D:\PALM-BDMA\BDMA-Homework\Advance DB\MongoDB\mongoDB_AdvDB
> react-scripts start
Starting the development server...
Compiled successfully!

You can now view mongodb-demo in the browser.

  Local:            http://localhost:3000/
  On Your Network:  http://192.168.0.102:3000/

Note that the development build is not optimized.
To create a production build, use yarn build.
```

2. Repeat step 1) by opening a new window and run 'npm run start:server' to make a connection between your application and Atlas cluster

```
C:\Users\ASUS>d:

D:\>cd D:\PALM-BDMA\BDMA-Homework\Advance DB\MongoDB\mongoDB_AdvDB

D:\PALM-BDMA\BDMA-Homework\Advance DB\MongoDB\mongoDB_AdvDB>npm run start:server

> mongodb-demo@0.1.0 start:server D:\PALM-BDMA\BDMA-Homework\Advance DB\MongoDB\mongoDB_AdvDB
> node ./backend/app.js

(node:34288) DeprecationWarning: current URL string parser is deprecated, and will be removed in a future version.
To use the new parser, pass option { useNewUrlParser: true } to MongoClient.connect.
```

3. Refresh your application connection

3.3. Function Design

According to the advantage and flexibility of MongoDB with other programming languages, we selected NodeJS to develop our 'Travel Agency' website. In this topic, we will show how we integrate NodeJS application with MongoDB Altas database. [3]

3.3.1. Add product

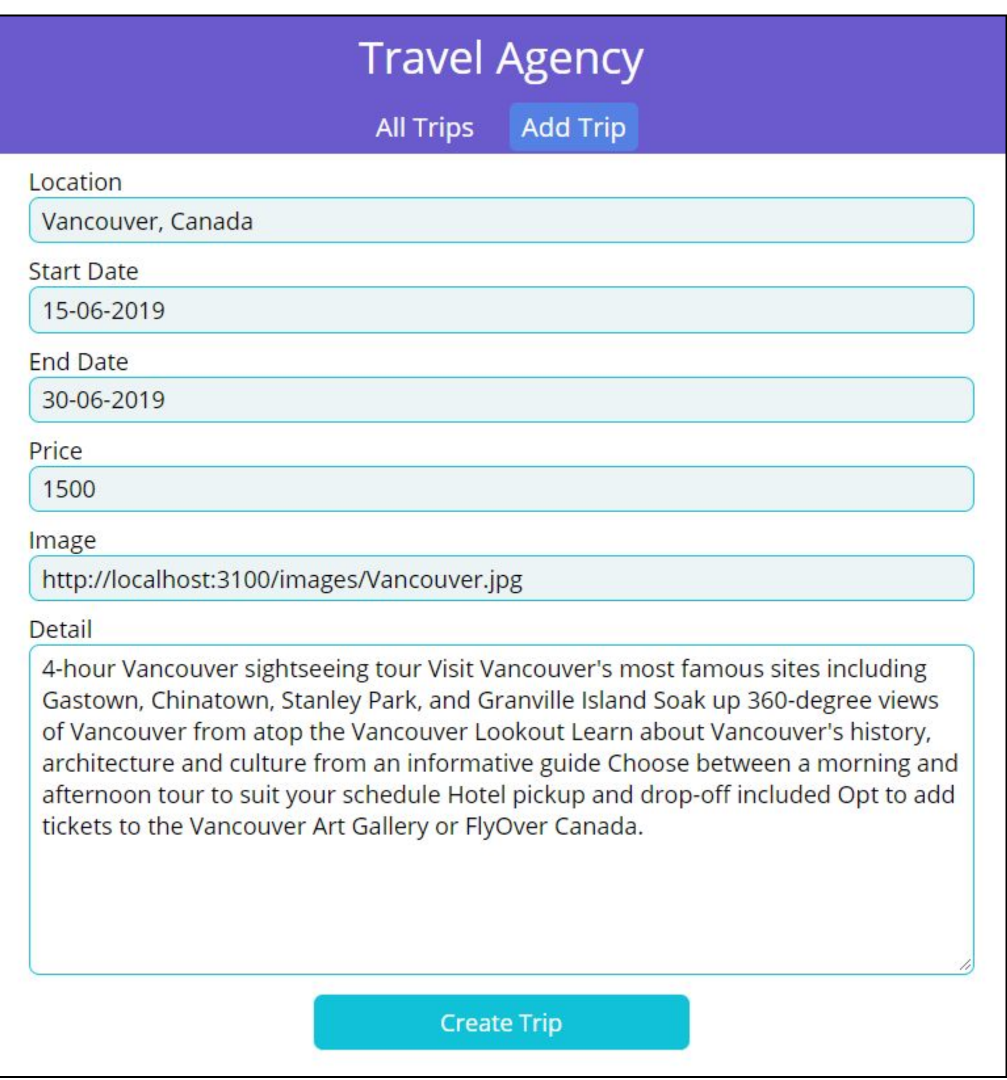
To insert data, we need to use MongoDB NodeJS syntax below:

```
db.collection(<collection name>).insertOne( <your doucument information>)
.then(result =>{
            //adding actions. You can add log or response message, etc.
  })
.catch(err => {
            //adding log or error message
  });
```

Below is a part of our implementation for adding the trip to the database:

```
router.post('', (req, res, next) => {
  const newTrip = {
    location: req.body.location,
    detail: req.body.detail,
    startdate: req.body.startdate,
    enddate: req.body.enddate,
    price: Decimal128.fromString(req.body.price.toString()),
    image: req.body.image
  };
  db.collection('products')
    .insertOne(newTrip)
    .then(result => {
      console.log(result);
      res.status(201).json({ message: 'Insert Trip Success', productId: result.insertedId });
    })
    .catch(err => {
      console.log(err);
      res.status(500).json({ message: 'Insert Trip Fail'} );
    });
});
```

The result of adding a trip on the website:



3.3.2. Display products

To display all of the data, we need to use MongoDB NodeJS syntax below:

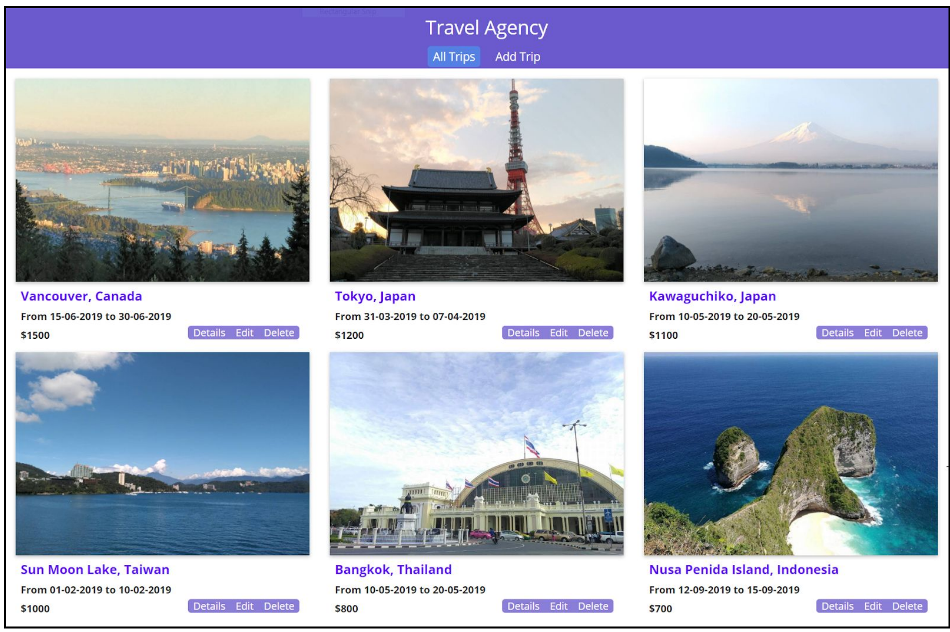
```
db.collection(<collection name>).find()
.forEach(
        //adding actions.
 )
.then(
        //adding actions. You can add log or response message, etc.
 ).catch(err => {
        //adding log or error message
 });
```

Below is a part of our implementation for display all the trips on the website:

```
  const trips = [];
  db.collection('products')
    .find()
    .sort({price: -1})
    .forEach(tripDoc => {
      tripDoc.price = tripDoc.price.toString();
      trips.push(tripDoc);
    })
    .then(result => {
      res.status(200).json(trips);
    })
    .catch(err => {
      console.log(err);
      res.status(500).json({ message: 'error'} );
    });
});
```

Our implementation is using find().forEach() function in order to read the data one by one and display them on the main page.

The result of displaying the trip on the website:



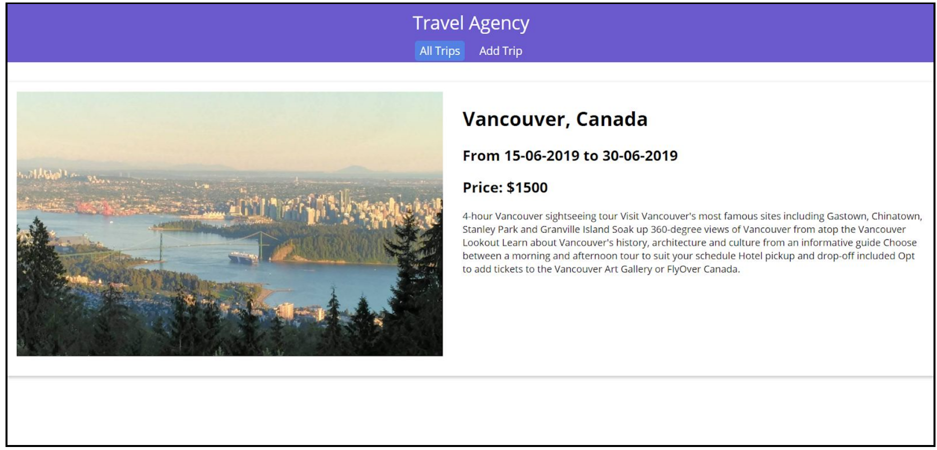
To display one single data, we need to use MongoDB NodeJS syntax below:

```
db.collection(<collection name>).findOne(<fitered by document information>)
.then(
        //adding actions. You can add log or response message, etc.
 ).catch(err => {
        //adding log or error message
 });
```

Below is a part of our implementation for display the trips on the website:

```
router.get('/:id', (req, res, next) => {
  db.collection('products')
    .findOne({_id: new ObjectId(req.params.id)})
    .then(tripDoc =>{
      tripDoc.price = tripDoc.price.toString();
      res.status(200).json(tripDoc);
    }).catch(err => {
      console.log(err);
      res.status(500).json({ message: 'error' });
    });
});
```

The result of displaying the detail of a trip on the website:



3.3.3. Edit product

To update data, we need to use MongoDB NodeJS syntax below:

```
db.collection(<collection name>).updateOne(<fitered by document information>, {$set: <update information>})
.then(result => {
    //adding actions. You can add log or response message, etc.
})
.catch(err => {
    //adding log or error message
});
```

Below is a part of our implementation to update the trips on the website:

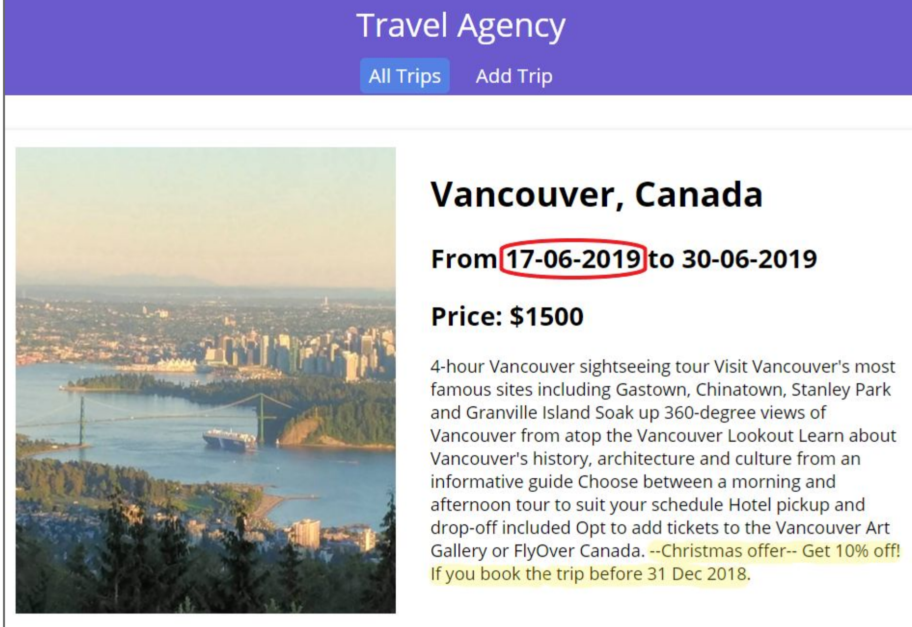
```
router.patch('/:id', (req, res, next) => {
  const updatedTrip = {
    location: req.body.location,
    detail: req.body.detail,
    startdate: req.body.startdate,
    enddate: req.body.enddate,
    price: Decimal128.fromString(req.body.price.toString()),
    image: req.body.image
  };
  db.collection('products')
    .updateOne({ _id: new ObjectId(req.params.id)},
      {
        $set: updatedTrip
      }
    )
    .then(result => {
      res.status(200).json({ message: 'Updated Product success', productId: req.params.id });
    })
    .catch(err => {
      console.log(err);
      res.status(500).json({ message: 'Updated Product Fail'} );
    });
});
```

The result of editing the trip on the website:

Once we click 'Edit' button on the main page, it will direct you to another page to update data.

In this example, we updated 'Start Date' and 'Detail' as shown below

The ‘Start Date’ and ‘Detail’ has been updated.



3.3.4. Delete product

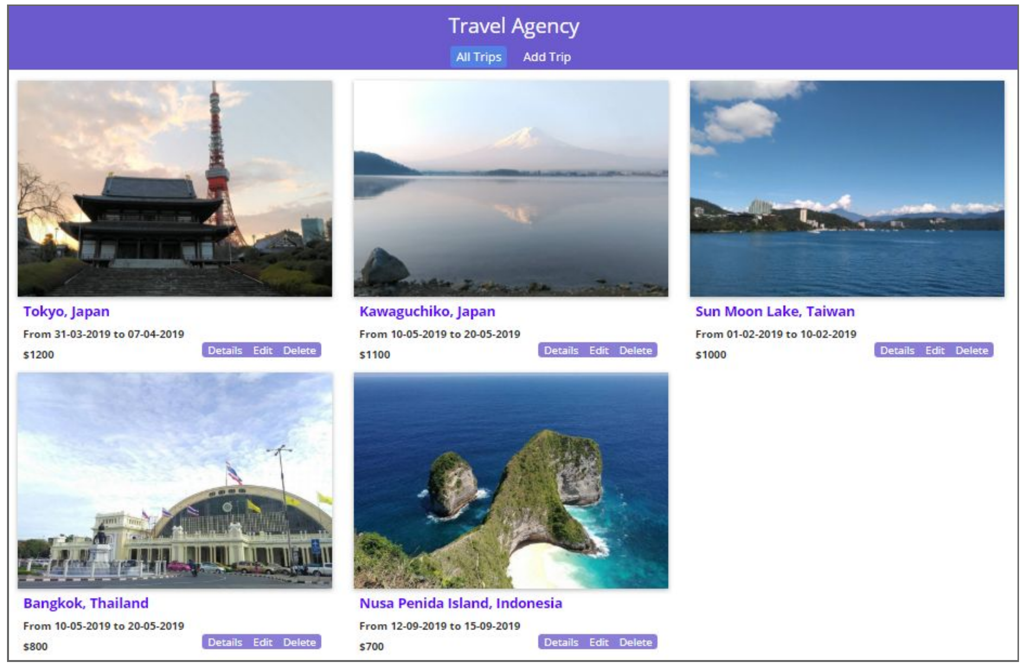
To delete data, we need to use MongoDB NodeJS syntax below:

```
db.collection(<collection name>).deleteOne(<filtered by document information>)
.then( result => {
        //adding actions. You can add log or response message, etc.
})
.catch(err => {
        //adding log or error message
});
```

Below is a part of our implementation to delete the trips on the website:

```
router.delete('/:id', (req, res, next) => {
  db.collection('products')
    .deleteOne({ _id: new ObjectId(req.params.id) })
    .then(result => {
      res.status(200).json({ message: 'Delete Product Success' });
    })
    .catch(err => {
      console.log(err);
      res.status(500).json({ message: 'Delete Product Fail'} );
    });
});
```

The result of deleting the Vancouver trip on the website:



4. Conclusion

MongoDB is a powerful document database that provides a lot of flexibilities for users. It can be used with various programming languages with similar query syntax as normal MongoDB query. The schemaless property provides suppleness of rapid growth of complexity of data. Developers can migrate data from the existing database to MongoDB and start running MongoDB database promptly. Moreover, it also has MongoDB Atlas cloud services to support business nowadays.

5. References

[1] MongoDB. (2018). Document Databases. [online] Available at: https://www.mongodb.com/document-databases.

[2] Docs.mongodb.com. (2018). MongoDB Documentation. [online] Available at: https://docs.mongodb.com/.

[3] udemy. (2018). [online] Available at: https://www.udemy.com/mongodb-the-complete-developers-guide/.

[4] En.wikipedia.org. (2018). Node.js. [online] Available at: https://en.wikipedia.org/wiki/Node.js.

[5] www.tutorialspoint.com. (2018). Node.js Introduction. [online] Available at: https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm .

[6] MongoDB. (2018). Fully Managed MongoDB, hosted on AWS, Azure, and GCP. [online] Available at: https://www.mongodb.com/cloud/atlas.

[7] Mongodb.github.io. (2018). Installation Guide. [online] Available at: http://mongodb.github.io/node-mongodb-native/3.1/installation-guide/installation-guide/.