1. INDEX IN MONGODB

1.1. Introduction

Generally, indexes in MongoDB is similar to indexes in others database systems. It help provide high performance read operation for frequently used queries.

- MongoDB provides different types of indexes for different purposes and different types of content. There are 6 types of index in MongoDB:
    + Single field indexes
    + Compound indexes
    + Multikey indexes
    + Text indexes
    + Hashed indexes
    + Geospatial indexes and Queries
- MongoDB provide various properties. Some index properties which you can select when building an index:
    + TTL indexes
    + Unique indexes
    + Sparse indexes
- MongoDB can use the **intersection** of multiple indexes to fulfill queries. In general, each index intersection involves two indexes; however, MongoDB can employ multiple/nested index intersections to resolve a query.

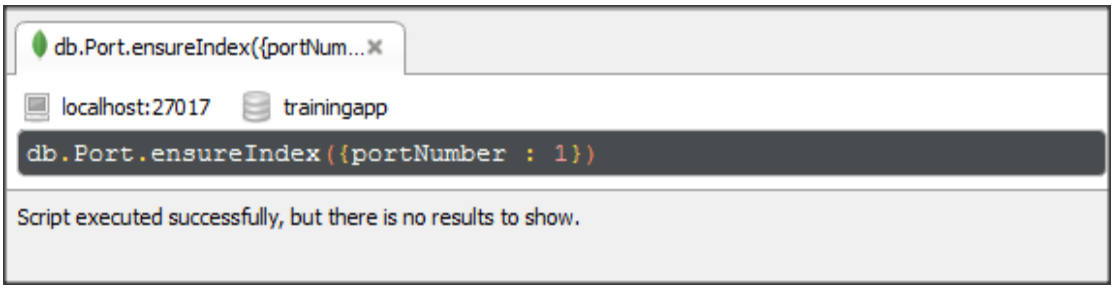
1.1.1. Types of indexes

a. Single field indexes

Single field indexes include data from a single field of documents in a collection. Each collection has 1 **default single index** which is created from **_id** field. See follow sample document for the Port collection:

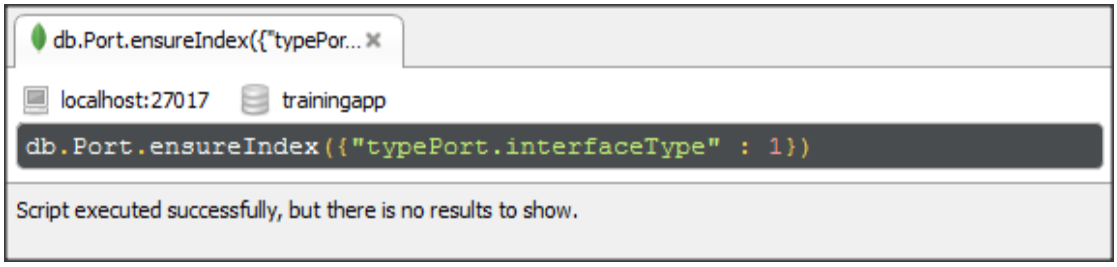
```
{
  "_id": 49,
  "_class": "training.model.Port",
  "namePort": "Port 1",
  "typePort": {
    "idType": 2,
    "nameType": "ENI_G",
    "intefaceType": "GIGAETHERNET"
  },
  "portNumber": 1,
  "createDate": new Date(1403585517000) /*6/24/2014, 11:51:57 AM*/,
  "modifiedDate": new Date(1403585517000) /*6/24/2014, 11:51:57 AM*/,
  "device": {
    "$ref": "Device",
    "$id": 1
  }
}
```

If we have to query on value of "portNumber" many times, we can create an index like:



Picture 1.1. Single field index – example 1

Other word, if we want to query on "interfaceType" of "typePort" field, we can create an index like:

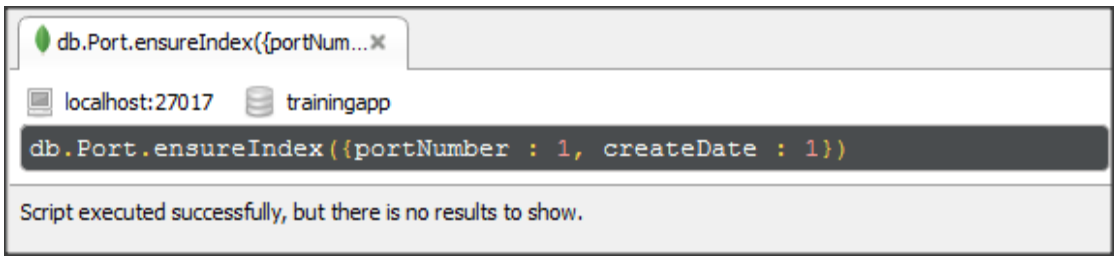


Picture 1.2. Single field index – example 2

Two above indexes are type of single field index.

b. Compound fields indexes

Compound field indexes include data from more than one field of documents in a collection. For example, use above sample document for the Port collection. If we want to find documents base on both “portNumber” field and “createDate” field, we can create a compound index like:



Picture 1.3. Compound fields index

* portNumber is called **Index prefix** of above compound index.

* Restriction of Compound Index:

- Index can apply for query: by portNumber field, or both portNumber field and createDate

- Index can apply for query: by only createDate

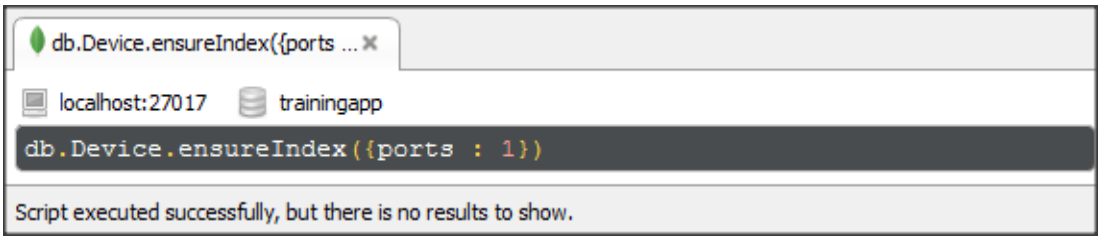
It mean compound indexes only can fulfill queries which include Index prefix.

c. Multikey indexes

Multikey indexes is a type of index use for index a field which hold an array value. It references an array and records a match if query includes any value in the array. When we use multikey indexes, MongoDB will add index items for each item in the array. See “ports” filed of follow document :

```
{
  "_id": 1,
  "_class": "training.model.Device",
  "name": "Switch1",
  "ipAddress": "10.1.1.1",
  "status": "Active",
  "ports": [
    1,
    2,
    3,
    4
  ],
  "createDate": new Date(1402977732000) /*6/17/2014, 11:02:12 AM*/,
  "modifiedDate": new Date(1402977732000) /*6/17/2014, 11:02:12 AM*/,
  "description": "This is a switch. For test."
}
```

We can create a multikey index like:

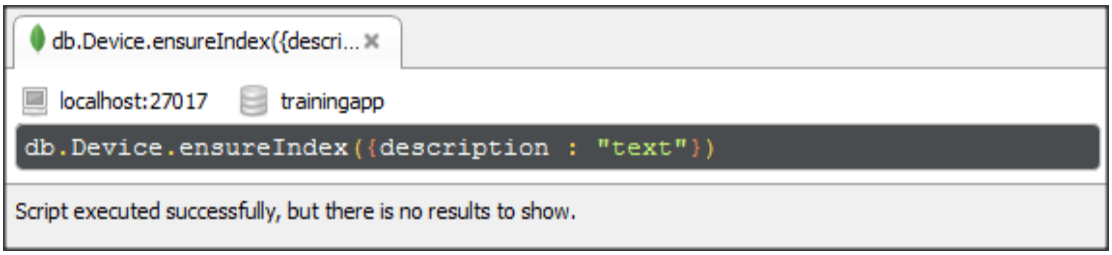


Picture 1.4. Multikey index

d.Text indexes

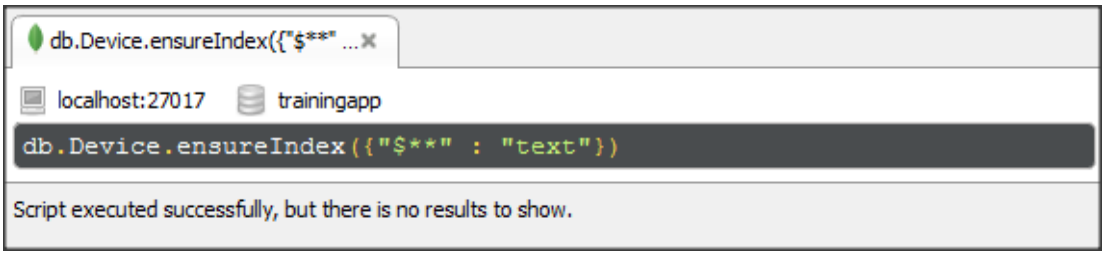
Text indexes support search with string content in documents. We can create text index for a specific field or all field that contain string content.

- Create text index for "description" field of above document:



Picture 1.5. Text index with one field

- Create text index for all fields: use wildcard specifier ($**)



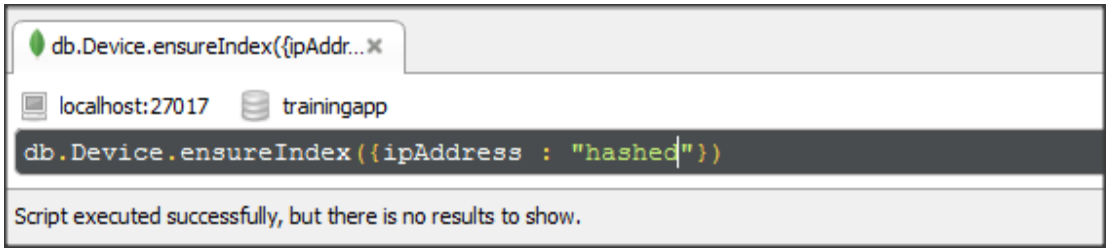
Picture 1.6. Text index with all fields

* To create text indexes, we need enable text search when start MongoDB by:

--setParameter textSearchEnabled=true

e. Hash indexes

Hash index maintain entries with hash of values of the hashed indexed field. The hash function collapse sub-documents and computers the hash for entire value but does not support multikey indexes. MongoDB can use hash index to support equality queries but not range queries. Create a hash index like following:



Picture 1.7. Hash index

g. Geospatial Indexes and Queries

Geospatial Indexes and Queries support location-base searches on data that is stored as either GeoJSON objects or legacy coordinate pairs. There are 2 main types of Geospatial indexes which you can build. These are **2dsphere** and **2d.**

The kind of index which are chose depend on storing your location data.

- To calculate geometry over an Earth-like sphere, store your location data on a spherical surface and use 2dsphere index. For example, see sample document below:

```
{
  "_id": {
    "$oid": "53ccd72af2abd7d71dce1ec7"
  },
  "name": "Lab4",
  "loc": {
    "type": "Point",
    "coordinates": [
      100,
      10
    ]
  }
}
```

We create 2dsphere index like:

```
db.Location.ensureIndex({"loc" : "2dsphere"})
```

- To calculate distances on a Euclidean plane, store your location data as legacy coordinate pairs and use a 2d index. For example, see sample document below:

```
{
  "_id": {
    "$oid": "53ccd72af2abd7d71dce1ec7"
  },
  "name": "Lab4",
  "loc": [
    40.731,
    -73.999
  ]
}
```

We create 2d index like:

```
db.Location.ensureIndex({"loc" : "2d"})
```

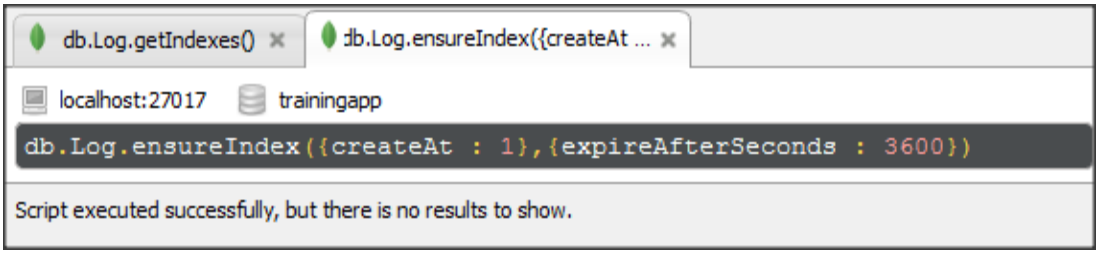
1.1.2. Index properties

a. TTL indexes

TTL (Time to live) indexes is used for TTL collections, which expire data after a period of time. It's a special indexes that MongoDB can use to automatically remove documents after a certain amount of time. This is ideal for some types of information like machine generated event data, logs, and session information that only need to persist in a database for a limited amount of time. See sample document for a Log collection follow:

```
{
  "_id": {
    "$oid": "53bf87a8ca0f8ec25b982bbc"
  },
  "createAt": new Date(1405059897000) /*7/11/2014, 1:24:57 PM*/,
  "logType": "Fail",
  "logMessage": "Switch up fail"
}
```

We can create TTL index like below:

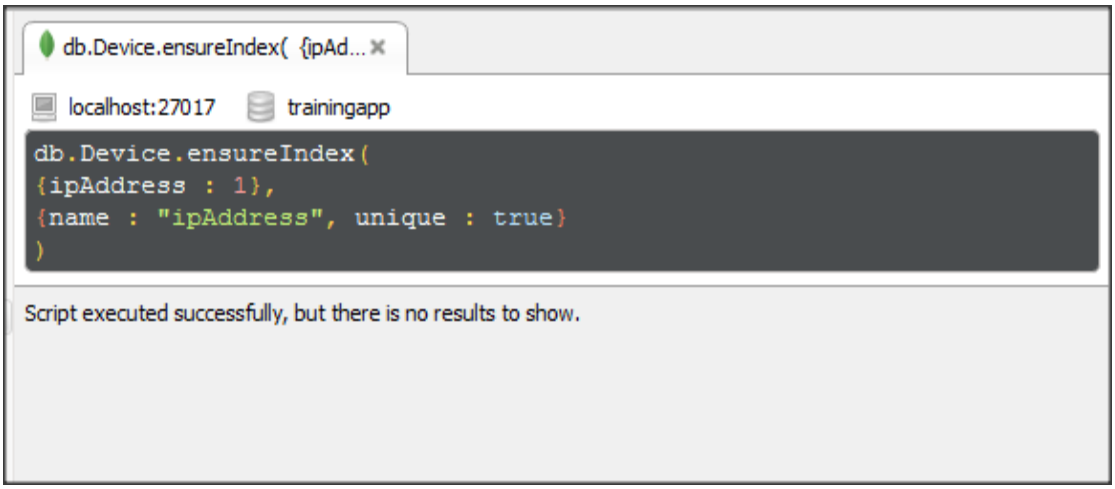


Picture 1.8. TTL index

New document is created in Log collection. And this documents in Log collection will be deleted by automatic after 1h.

b. Unique Indexes

Unique indexes cause MongoDB to reject all documents that contain a duplicate value for the indexes field. Default index which is create from _id field is a unique indexes. To create an unique index we need to set unique property is true.



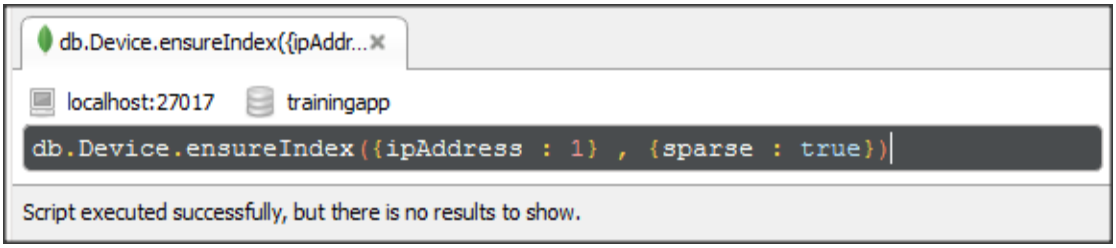
Picture 1.9. Unique index

**** If there are duplicated values of indexed field in database. We can't create the unique index for that field.***

c. Sparse Indexes

Sparse indexes skip over documents which don’t include indexed field. It's useful when we have large of number documents and small percent of them contain indexed field. It help query operation will be faster much.

To create a sparse index we use command like:



Picture 1.10. Sparse index

2.1.1. 1.1.3 Index Intersection

See sample document for collection Device again:

```
{
  "_id": 1,
  "_class": "training.model.Device",
  "name": "Switch1",
  "ipAddress": "10.1.1.1",
  "status": "Active",
  "ports": [
    1,
    2,
    3,
    4
  ],
  "createDate": new Date(1402977732000) /*6/17/2014, 11:02:12 AM*/,
  "modifiedDate": new Date(1402977732000) /*6/17/2014, 11:02:12 AM*/,
  "description": "This is a switch. For test."
}
```

We create two indexes for this collection like:

```
db.Device.ensureIndex({ipAddress : 1},{name : "ipAddress"})
```

```
db.Device.ensureIndex({status : 1},{name : "status"})
```

Then we query like:

```
db.Device.find({ipAddress : "10.1.1.1", status : "Active"})
```

Both “status” index and “ipAddress” index are used to fulfill the query.

*** Difference between Index Intersection and Compound Index.**

Index Intersection	Compound Index
- Intersection of 2 indexes which can be single index, compound index, or others indexes. - Can’t not support queries when don’t have proper existing indexes.	- Include 2 or many fields in document. - Can’t support queries which don’t include Index prefix.

See below example:

There are a compound index:

```
db.Device.ensureIndex({ipAddress : 1, createDate : -1})
```

It can support query like:

```
db.Device.find({
    ipAddress : "10.1.1.1",
    createDate : {"$lt" : new Date()}

})
```

It can't support query like:

```
db.Device.find({
    createDate : {"$lt" : new Date()}
})
```

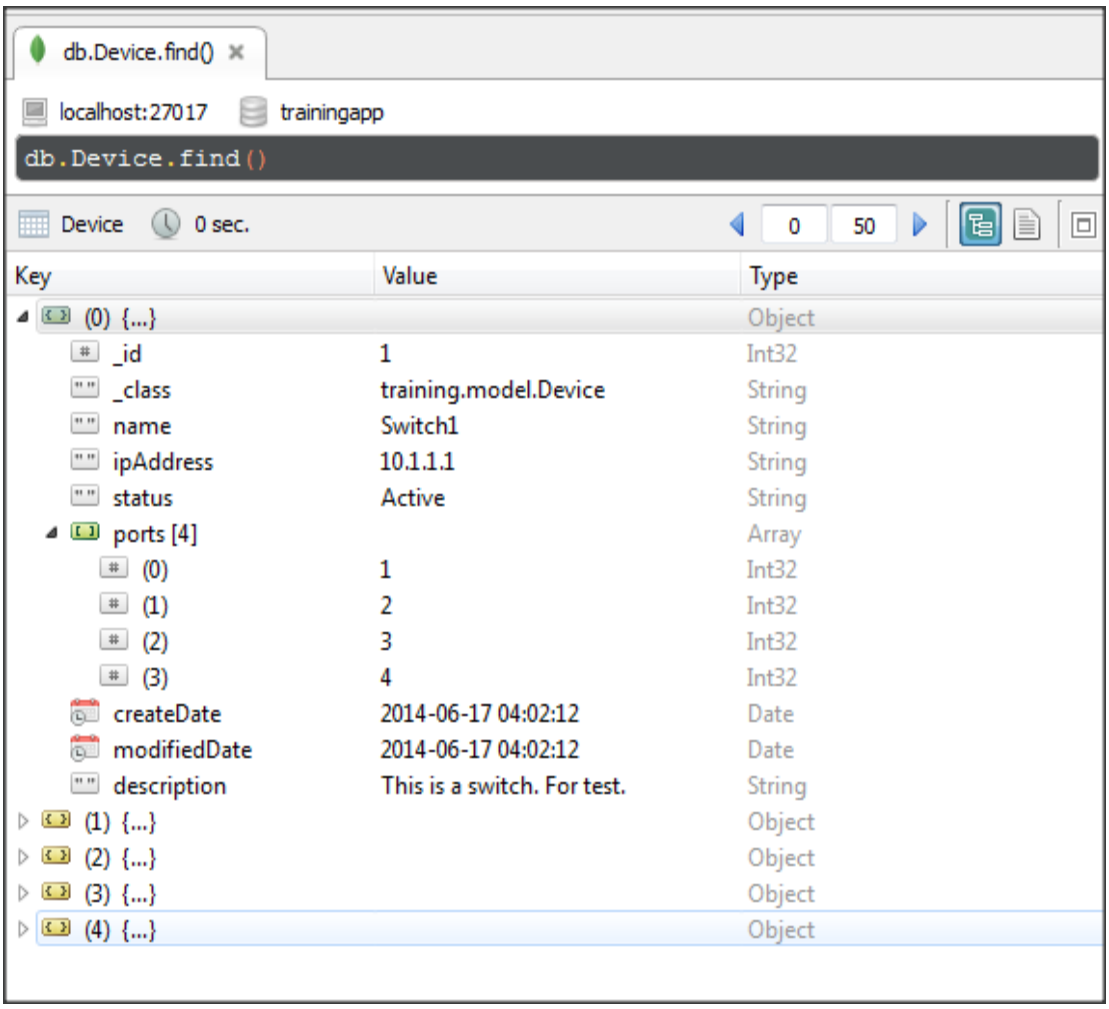
because the query doesn't include **index prefix.**

But if we have 2 separated indexes like following, the index intersection will support for both queries above.

```
db.Device.ensureIndex({ipAddress : 1})
db.Device.ensureIndex({createDate : -1})
```

2.2. 1.2. Management

To example for indexes management, use follow collection like below.



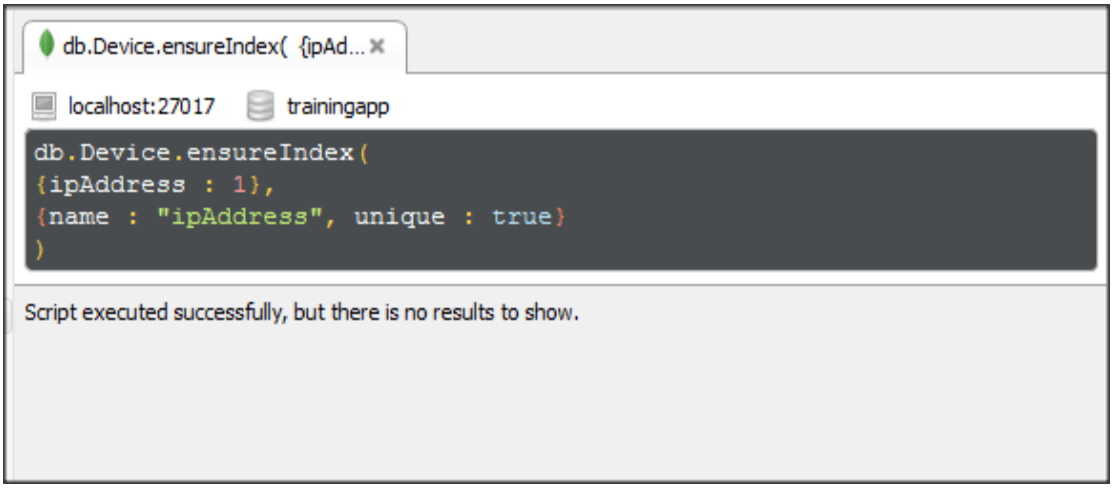
Picture 1.11. Device collection

1.2.1. Create indexes

Syntax:

*db.**collectionName**.ensureIndex({**field or set of fields**},[{**index properties**}])*

If we have to search devices by ipAddress many times, we can create an index from ipAddress field. And ipAddress is unique field so we can set unique property for this index. In Robomongo, we use follow command:



Picture 1.12. Create ipAddress index

In MongoTemplate, we do:

```
public void createIndex() {
    // 1. Get IndexOperations from Device document
    IndexOperations indexOps = mongoTemplate.indexOps(Device.class);
    // 2. Create an index object
    Index index = new Index();
    index.on("ipAddress", Direction.ASC);
    index.named("ipAddress");
    index.unique();
    // 3. Create an index in Device document
    indexOps.ensureIndex(index);
}
```

Picture 1.13. Create ipAddress index in java

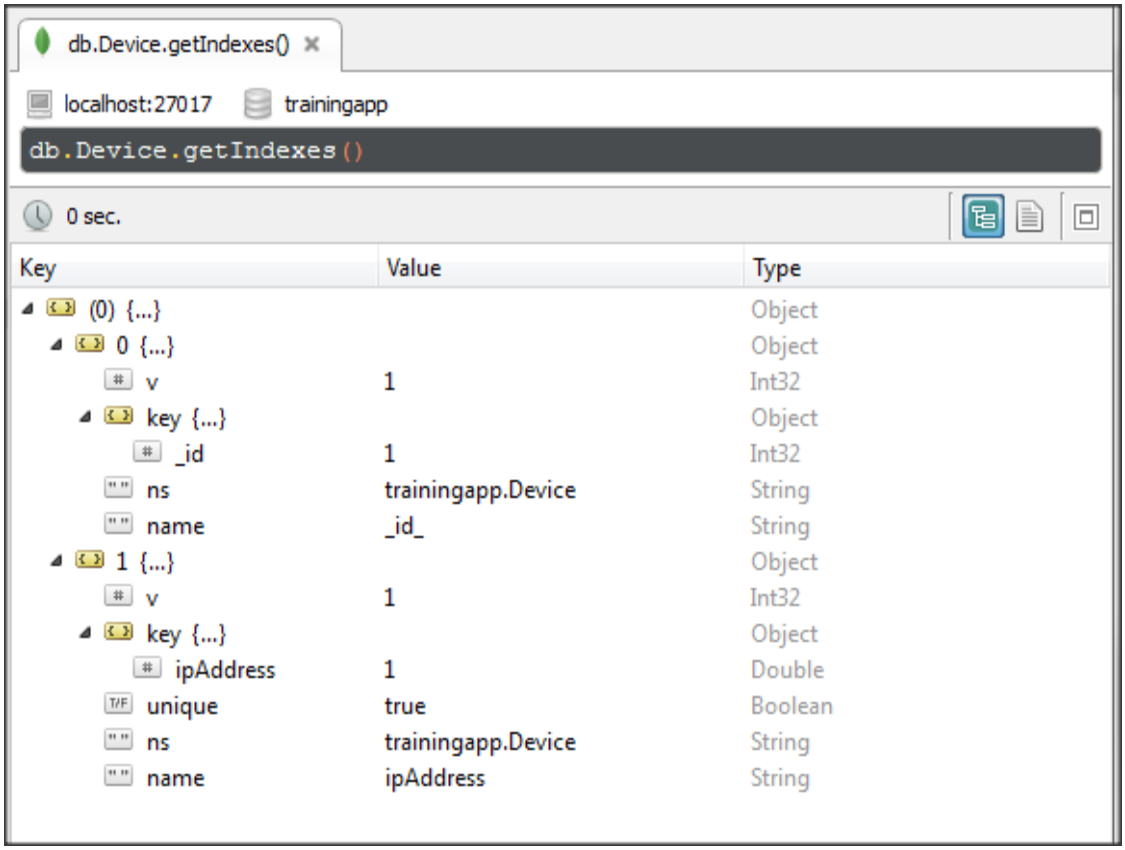
1.2.2. Get indexes of a collection

To see result of above example, we can use getIndexes() operation of MongoDB.

Syntax:

*db.**collectionName**.getIndexes()*

In Robomongo:



Picture 1.14. Get indexes in database

Above result, Device collection has 2 indexes.

+ The first is default index for each collection which is created by MongoDB from _id field.

+ The second is ipAddress index which is created by our example.

In MongoTemplate, we do:

```
public void getIndexes() {
    // 1. Get IndexOperations from Device document
    IndexOperations indexOps = mongoTemplate.indexOps(Device.class);
    // 2. Get list IndexInfo
    List<IndexInfo> indexes = indexOps.getIndexInfo();
    // 3. Display information of each index in list
    for(IndexInfo indexInfo : indexes) {
        System.out.println(indexInfo.toString());
    }
}
```

Picture 1.15. Get indexes using java

1.2.3. Remove indexes:

Remove an index of a document:

Syntax:

db.collectionName.dropIndex({indexName})

Or

db.collectionName.dropIndex({field : value})

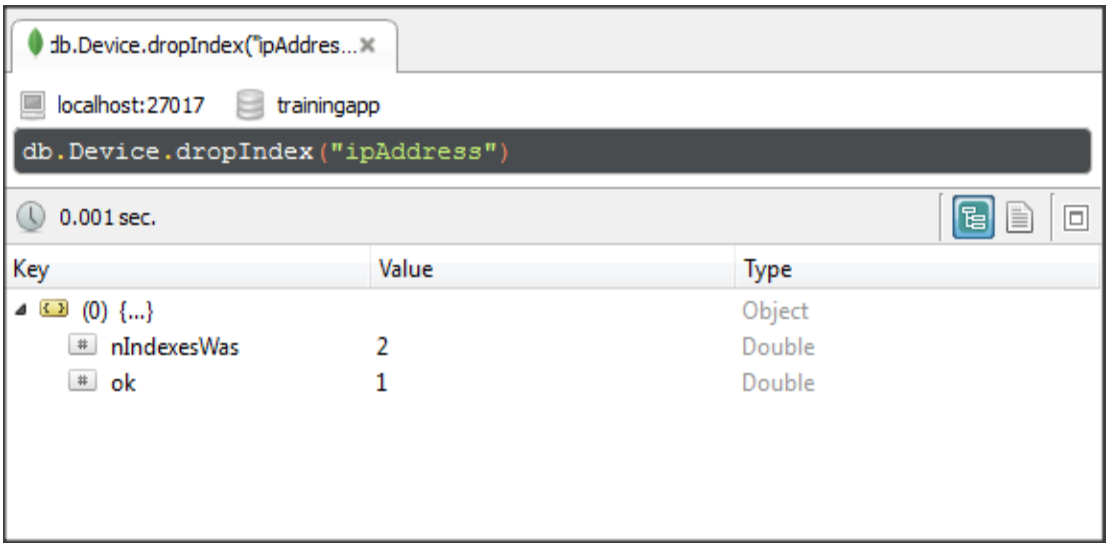
+ value: -1 or 1 depend on order of index when we created before.

Remove all indexes of a document:

Syntax:

db.collectionName.dropIndexes()

In Robomongo, we use follow command:



Picture 1.16. Drop indexes

In MongoTemplate, we do:

```
public void dropIndex() {
    // 1. Get IndexOperations from Device document
    IndexOperations indexOps = mongoTemplate.indexOps(Device.class);
    // 2. Drop ipAddress index
    indexOps.dropIndex("ipAddress");
}
```

Picture 1.17. Drop indexes using java

1.2.4. Rebuild an index

As you perform inserts, updates and deletes, your indexes will become fragmented both internally and externally. Internal fragmentation is you have a high percentage of free space on your index pages, meaning that database needs to read more pages when scanning the index. External fragmentation is when the pages of the index are not in order any more, so database has to do more work, especially in IO terms to read the index.
So we need rebuild index. MongoDB provide we an operation is:

db.collectionName.reIndex()

1.2.5. Speed comparison of query with indexes and not

We will use Log collection with 3000 documents for testing speed of query when indexes is set up and not. See Log collection .

Using java codes to insert documents:

```
public void insertLogForTest() {
    Log log = new Log();
    for(int i = 1; i < 3000; i++) {
        log.setCreateAt(new Date());
        log.setLogMessage("Message " + i);
        log.setLogType("Success");
        logRepository.save(log);
    }
}
```

Picture 1.18. Insert documents into Log collection

Situation: Find switches which are created after "2014-07-21" . We will choose "createAt" field to ensure index. Compare time to complete find query.

Using following codes block to check in java:

```java
public class App {

    private static ClassPathXmlApplicationContext classPathXmlApplicationContext;

    public static void main(String[] args) throws ParseException {
        classPathXmlApplicationContext = new ClassPathXmlApplicationContext(
                "applicationContext.xml");
        IReadOperations readOperations = classPathXmlApplicationContext
                .getBean("readOperations", IReadOperations.class);
        Long start, end;
        SimpleDateFormat sdf = new SimpleDateFormat("yy-MM-dd");
        Date date = sdf.parse("2014-07-21");

        // Time before processing query in milliseconds
        start = System.currentTimeMillis();
        // Query method
        readOperations.findLogs(date);
        // Time in completing query method
        end = System.currentTimeMillis();

        System.out.println("Take " + (end - start) + " miliseconds.");
    }
}
```

Picture 1.19. Java codes block for test speed of index

findLogs method is:

```java
@Override
public List<Log> findLogs(Date createAt) {
    Query query = new Query();
    query.addCriteria(Criteria.where("createAt").gt(createAt));
    query.with(new Sort(Direction.ASC, "createAt"));
    return mongoTemplate.find(query, Log.class);
}
```

Picture 1.20. Get Log documents in java

See explain() of query before set up index.

```
db.Log.find(
    {"createAt" : {"$gt" : ISODate("2014-07-21")}}
).explain()
```

0.004 sec.

Key	Value	Type	
(0) {...}		Object	
cursor	BasicCursor	String	Use no index
isMultiKey	false	Boolean	
n	2999	Int32	
nscannedObjects	3001	Int32	
nscanned	3001	Int32	
nscannedObjectsAllPl...	3001	Int32	
nscannedAllPlans	3001	Int32	
scanAndOrder	false	Boolean	
indexOnly	false	Boolean	
nYields	23	Int32	
nChunkSkips	0	Int32	
millis	3	Int32	
server	vnguyenkhoa:27017	String	
filterSet	false	Boolean	
stats {...}		Object	

Picture 1.21. Speed comperison without index– explain query

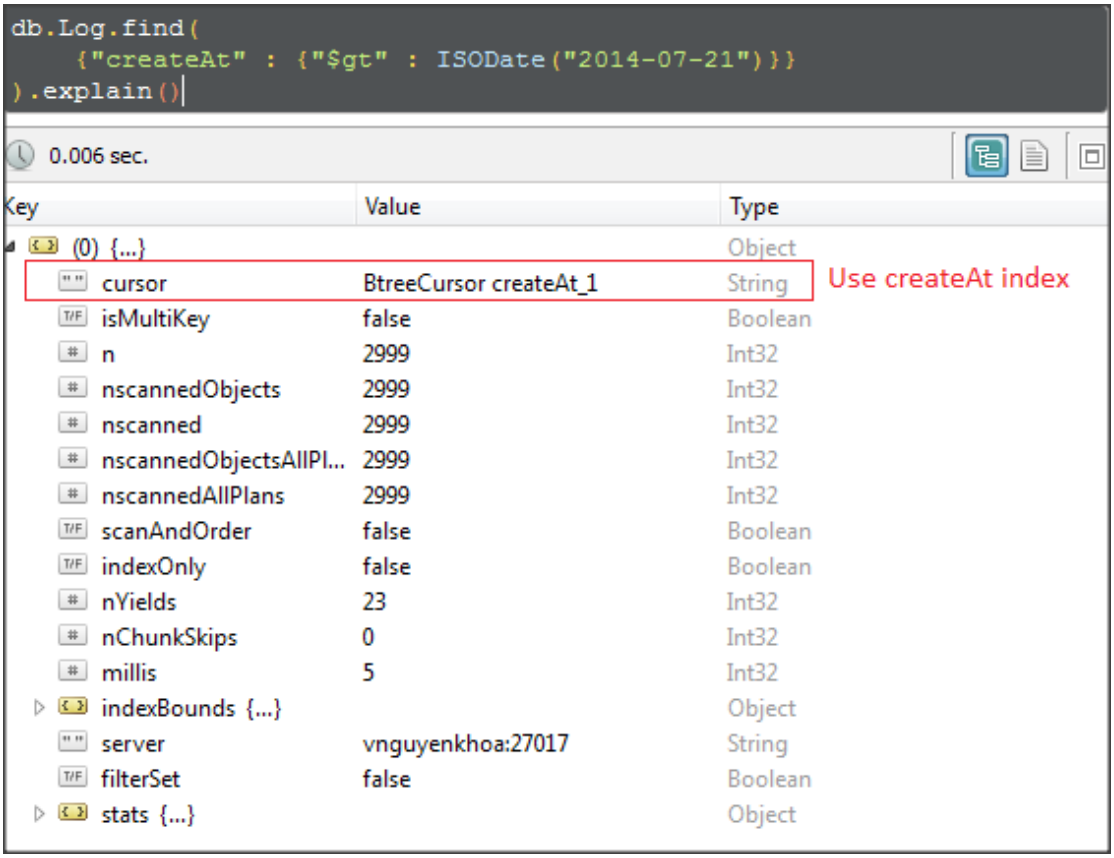
Time to process query:

```
Take 542 miliseconds.
```

Then we set up index:

```
db.Log.ensureIndex({"createAt" : 1})
```

See explain() of query after set up index.



Picture 1.22. Speed comperison with index– explain query

Time to process query:

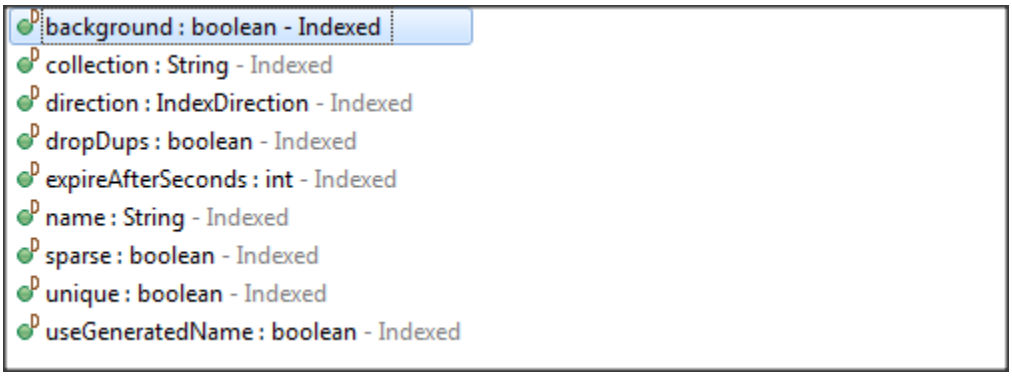
```
Take 508 miliseconds.
```

1.2.6. Indexed annotation (@Indexed)

a. Indexed annotation

Indexed annotation is a type of mapping annotation in Spring mongo data. It's applied at the **field level** to describe how to index the field.

It also provide us options to build other types of indexes as the following:



How to Indexed annotation work:

- Check existence of indexes which correspond to indexed field.

- If those indexes are not exists, they will be created.

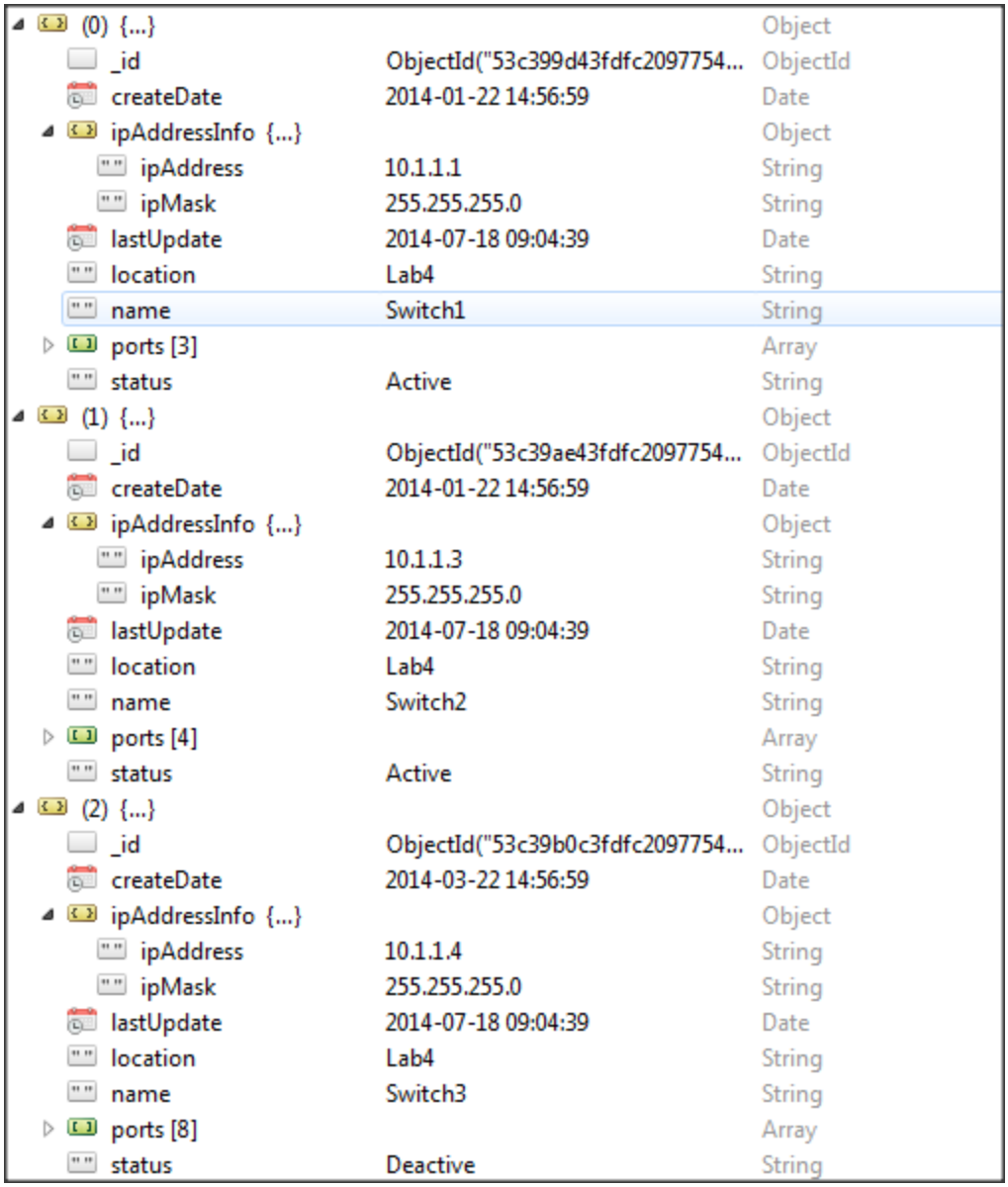
- The application will process other operations.

Example: Index two field “name” and “ipAddressInfo” with unique type. Then use repository to insert documents that is exists in database. See how to indexed annotation work. See Switch model class is used for mapping with Switch collection like the following:

```
@Document(collection=MongoCollection.SWITCH_COLLECTION)
public class Switch {

    @Id
    private ObjectId id;

    @Indexed(unique=true)
    private IPAddressInfo ipAddressInfo;

    @Indexed(unique=true)
    private String name;

    private String location;
    private String status;
    private List<SwitchPort> ports;
    private Date createDate;
    private Date modifiedDate;

    // getter & setter are omitted
```

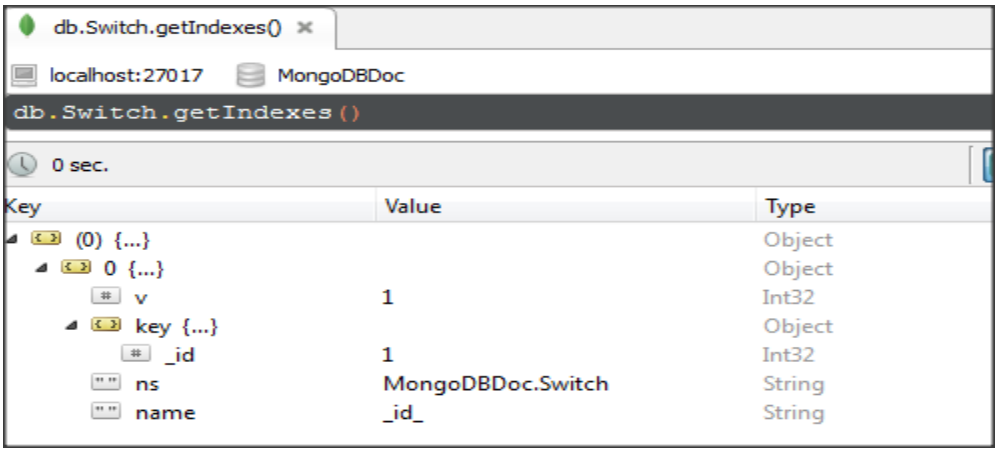
Picture 1.23. Switch model class

Switch collection before insert new documents:



Picture 1.24. Sample Switch collection

All indexes of this collection:



Picture 1.25. All index of Switch collection

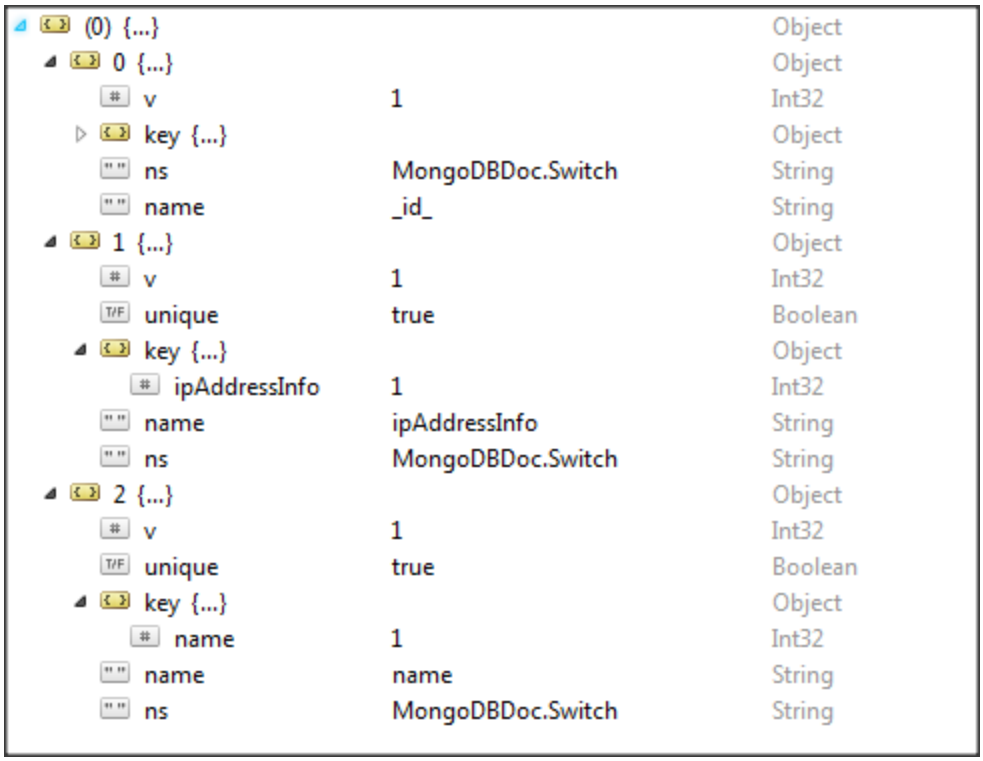
*** Insert a document which has field “name” which is exists**

Java codes:

```
public void insertDuplicate() {
    try {
        // 1. Repair a switch which is exists in database
        Switch sw = new Switch();
        sw.setCreateDate(new Date());
        sw.setModifiedDate(new Date());
        sw.setName("Switch1");
        sw.setLocation("Lab4");
        // 2. Using switch repository to save above switch
        switchRepository.save(sw);
    } catch(MongoException e) {
        System.out.println(e.getMessage());
    } catch (DuplicateKeyException e) {
        System.out.println(e.getMessage());
    }
}
```

Picture 1.25. All index of Switch collection

See all indexes of collection again:



Picture 1.26. All index of Switch collection

But there is no new document in database and the application occur error like:

```
{ "serverUsed" : "localhost:27017" , "ok" : 1 , "n" : 0 , "err" :
"insertDocument :: caused by :: 11000 E11000 duplicate key error index:
MongoDBDoc.Switch.$name  dup key: { : \"Switch1\" }" , "code" : 11000};
nested exception is com.mongodb.MongoException$DuplicateKey: { "serverUsed"
: "localhost:27017" , "ok" : 1 , "n" : 0 , "err" : "insertDocument ::
caused by :: 11000 E11000 duplicate key error index: MongoDBDoc.Switch.
$name  dup key: { : \"Switch1\" }" , "code" : 11000}
```

The application occur error because the unique indexes of “name” field is added. So this unique indexes doesn't allow new document with name “Switch1” is inserted.

*** Insert a document which has duplicated field “ipAddress”**

Java codes:

```
public void insertDuplicate() {
    try {
        // 1. Repair a switch which is exists in database
        IPAddressInfo ipAddressInfo = new IPAddressInfo();
        ipAddressInfo.setIpAddress("10.1.1.1");
        ipAddressInfo.setIpMask("255.255.255.0");
        Switch sw = new Switch();
        sw.setCreateDate(new Date());
        sw.setModifiedDate(new Date());
        sw.setName("Switch6");
        sw.setIpAddressInfo(ipAddressInfo);
        sw.setLocation("Lab4");
        // 2. Using switch repository to save above switch
        switchRepository.save(sw);
    } catch(MongoException e) {
        System.out.println(e.getMessage());
    } catch (DuplicateKeyException e) {
        System.out.println(e.getMessage());
    }
}
```

Picture 1.27. Example use model class in java 2

The application occur error like:

```
{ "serverUsed" : "localhost:27017" , "ok" : 1 , "n" : 0 , "err" :
"insertDocument :: caused by :: 11000 E11000 duplicate key error index:
MongoDBDoc.Switch.$ipAddressInfo  dup key: { : { ipAddress: \"10.1.1.1\",
ipMask: \"255.255.255.0\" } }" , "code" : 11000}; nested exception is
com.mongodb.MongoException$DuplicateKey: { "serverUsed" : "localhost:27017"
, "ok" : 1 , "n" : 0 , "err" : "insertDocument :: caused by :: 11000 E11000
duplicate key error index: MongoDBDoc.Switch.$ipAddressInfo  dup key: { : {
ipAddress: \"10.1.1.1\", ipMask: \"255.255.255.0\" } }" , "code" : 11000}
```

** **Notice** that there is an excepted case with **save()** method of repository. If we insert new document with **_id field** which is exists in database. The exists document will be **updated** instead of inserting new document.*

b. Restriction

There are some cases which we can't create index by using indexed annotation.

Case 1: There is an exists index of indexed field in database.

See Switch model class again:

```
@Document(collection=MongoCollection.SWITCH_COLLECTION)
public class Switch {

    @Id
    private ObjectId id;

    @Indexed(unique=true)
    private IPAddressInfo ipAddressInfo;

    @Indexed(unique=true)
    private String name;

    private String location;
    private String status;
    private List<SwitchPort> ports;
    private Date createDate;
    private Date modifiedDate;

    // getter & setter are omitted
```

Picture 1.28. Switch model class

If the database has 1 index of field “name” like:

```
{
  "v": 1,
  "key": {
    "name": 1
  },
  "name": "name",
  "ns": "MongoDBDoc.Switch"
}
```

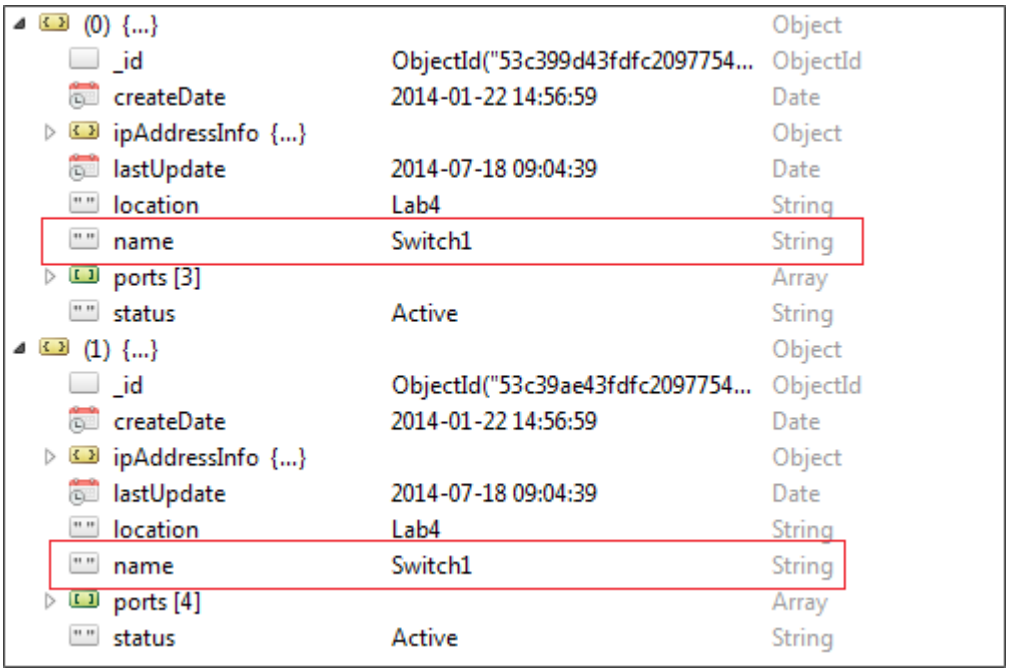
When the application use Swich model is executed, there is an error like:

```
"errmsg" : "Index with name: name already exists with different options"
```

The application occur error because there is an not unique index of field “name” with. So driver can't add unique index for field “name”. It mean the **@Indexed** can't help application override exists index in database.

Case 2: Collection include documents which have duplicated values at index field.

See sample documents in Switch collection:



Picture 1.29. Sample documents in Switch collection

Executing the application which use Switch model, there is error like:

```
"errmsg" : "E11000 duplicate key error index: MongoDBDoc.Switch.$name  dup key: { : \"Switch1\" }"
```

The application occur error because there are duplicated values of field “name” in database (see Unique indexes section for more detail).

1.2.7. CompoundIndex annotation (@CompoundIndex)

CompoundIndex annotation is a type of mapping annotation in Spring mongo data. It's applied at the **document level** to describe how to create compound index. See Switch model class which is used for mapping Switch collection like following:

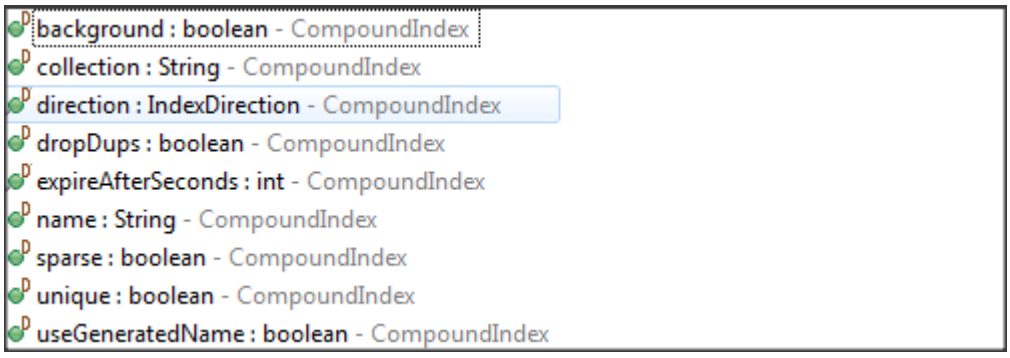
```
@Document(collection = MongoCollection.SWITCH_COLLECTION)
@CompoundIndexes({
        @CompoundIndex(def = "{'name' : 1, 'createDate' : -1}"),
        @CompoundIndex(def = "{'status' : 1, 'location' : 1}", name = "status_location")
})
public class Switch {

    @Id
    private ObjectId id;
    private IPAddressInfo ipAddressInfo;
    private String name;
    private String location;
    private String status;
    private List<SwitchPort> ports;
    private Date createDate;
    private Date modifiedDate;

    // getter & setter are omitted
```

Picture 1.30. Switch model class

- All compound indexes are defined in **@CompoundIndexes**.

- Every compound index include a required option is **def** to describe what fields is used for indexing.

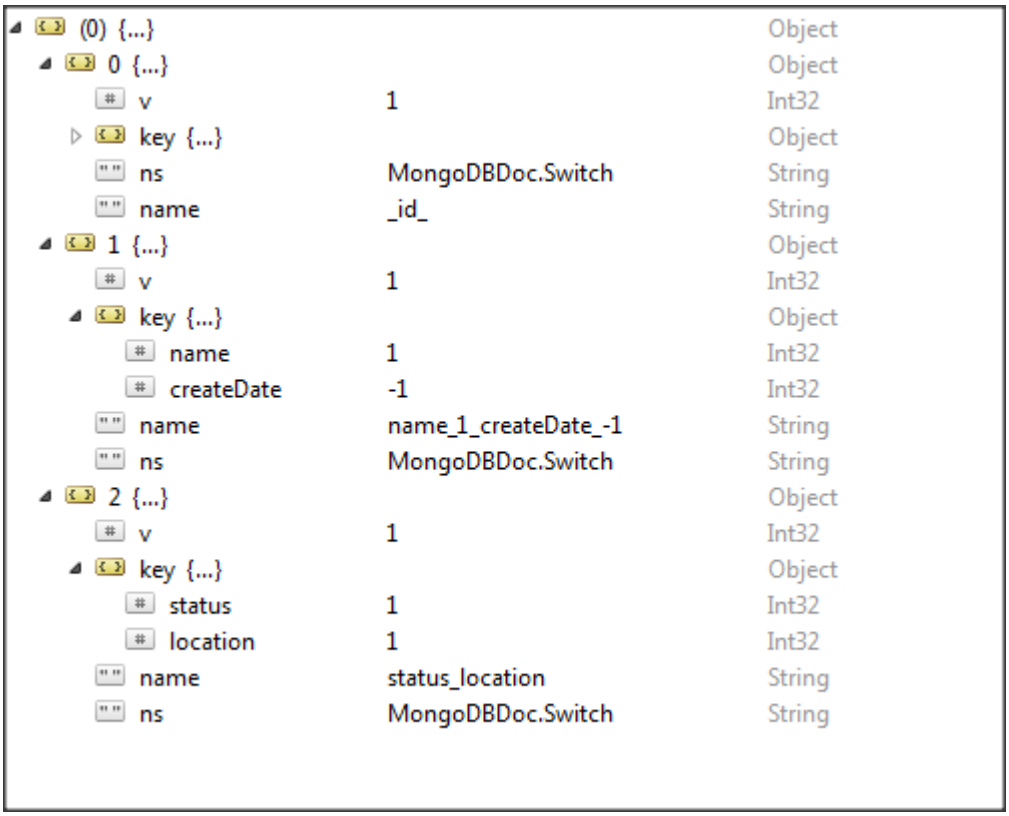
- Beside that, there are other options to build other types of compound index like:



Example: Creating 2 compound indexes by using annotation (see above model):

- Compound index of two field: name and createDate.
- Compound index of two field: status and location. And name is: status_location.

The result:


(0) {...} Object
0 {...} Object
v 1 Int32
key {...} Object
ns MongoDBDoc.Switch String
name _id_ String
1 {...} Object
v 1 Int32
key {...} Object
name 1 Int32
createDate -1 Int32
name name_1_createDate_-1 String
ns MongoDBDoc.Switch String
2 {...} Object
v 1 Int32
key {...} Object
status 1 Int32
location 1 Int32
name status_location String
ns MongoDBDoc.Switch String

2. QUERY OPERATION

2.1. Read operations

Syntax:

*db.**collectionName**.find([{**query criteria**}], [{**projection**}]).**cursor-modifier**()*

- **collectionName**: include needed documents.
- **query criteria**: include conditions to find documents.
- **projection**: specify a list of fields to return or a list of field to exclude.
- **cursor-modifier**: modify result such as: limit, skip, sort, order.

There is a special case of find() method which return a single document:

*db.**collectionName**.findOne([{**query criteria**}], [{**projection**}])*

There are some types of read operations like the following:

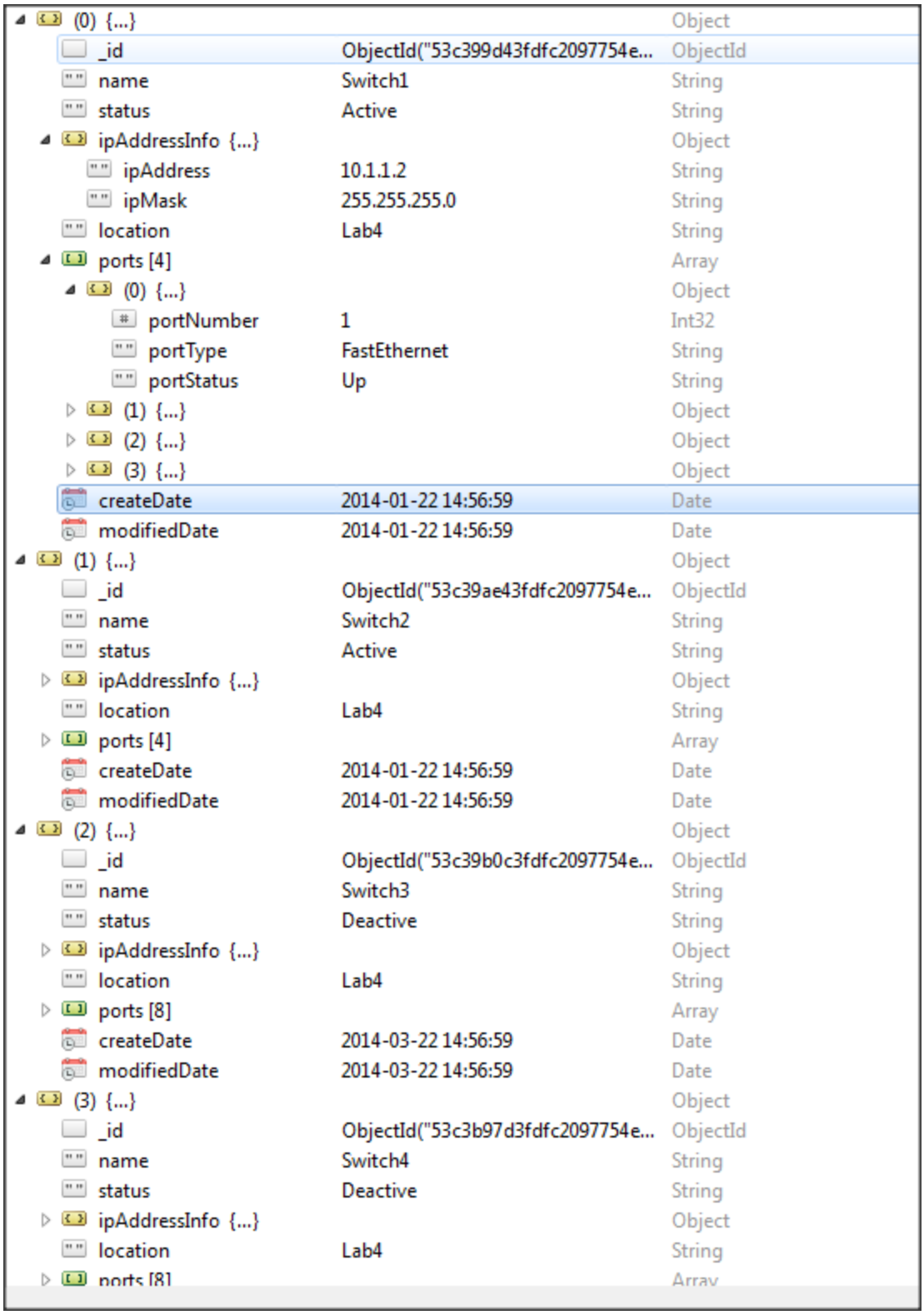
No	Type	Description
1	Get all documents	- Command: db.**collection**.find(). - Get all documents in the collection.
2	Get list of documents with **equality conditions**.	- Command: db.**collection**.find(<query criteria>) - Query criteria: {**<field>** : **<value>**} - Get list of documents contain **<field>** with specified **<value>**.
3	Get list of documents with conditions using **query operators**.	- Command: db.**collection**.find(<query criteria>). - Query criteria: detail in the following. - Get list of document contain **<field>** with: specifed value, range values, type, etc.
4	Get list of documents with conditions which include **embedded documents**.	- Command: db.**collection**.find(<query criteria>). - Query criteria: {**<field>** : **<embedded document>**}, or {**<field>** : **<field of embedded document>**}. - Get list of document contain **<field>** exact match embedded document or exact match within fields of embedded document.

5	Get list of documents with conditions which include **arrays**.	- Command: db.**collection**.find(<query criteria>). - Query criteria: {**<field>** : **<an array>**}, {**<field>** : **<an element>**} or {**<field>** : **<specified element>**}. - Get list of document contain **<field>** exact match an array or specified values in the array.
6	Get list of documents with **specified fields**.	- Command: db.**collection**.find(<query criteria>, <projection>). - Query criteria: like from No1 to No5. - Projection: include fields [{**<field>** : 1}] or exclude fields [{**<fields>** : 0}] - Get list of documents which only include specified fields.
7	Get list of documents with **cursor modifiers**.	- Command: db.**collection**.find(<query criteria>, <projection>).**cursor-modifier**(). - Query criteria: like from No1 to No5. - Projection: like No6. - Cursor modifier: sort(), limit(), skip(), order(). - Support for sorting, paging, increase performance of read operations.

Table 2.1. Types of read operations.

In the following, we will build query conditions to get necessary documents by combine query operators. But how to combine operators is various and depend on the application requirements. See full operators in Query Operators.

Using collection with documents that contain fields similar to the following (Switch collection):



Picture 2.1. Switch collection

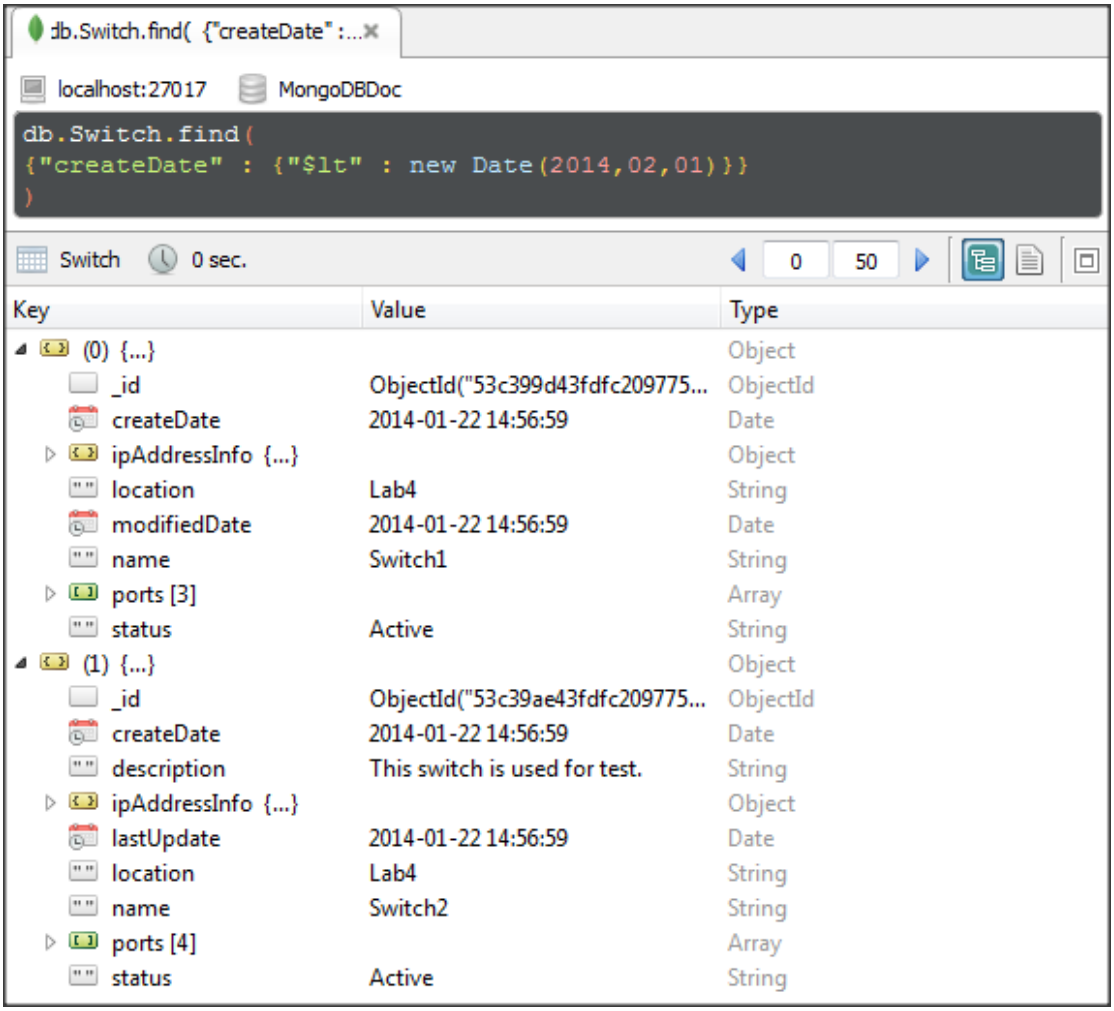
2.1.1. Using operator $lt

Format: {field : {$lt : query-value}}

Description: Matches values that are less than query-value.

Example: Find switches which are created before 01-02-2014.

In Robomongo:



Picture 2.2. Operator $lt in Robomongo

Using MongoTemplate:

```
try {
    // 1. Create date to query
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    Date date = sdf.parse("2014-02-01");
    // 2. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.and("createDate").lt(date);
    // 3. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 4. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
} catch (ParseException e) {
    // Return null list if occur exception
    System.out.println("Parse Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.3. Operator $lt using java

Using Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'createDate' : {'$lt' : ?0}}")
List<Switch> ltOperator(Date date);
```

Picture 2.4. Method using operator $lt in Switch repository

Then use in application:

```
try {
    // 1. Create date to query
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    Date date = sdf.parse("2014-02-01");
    // 2. Query result by using switchRepository
    return switchRepository.ltOperator(date);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
} catch (ParseException e) {
    // Return null list if occur exception
    System.out.println("Parse Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.5. Using repository in application

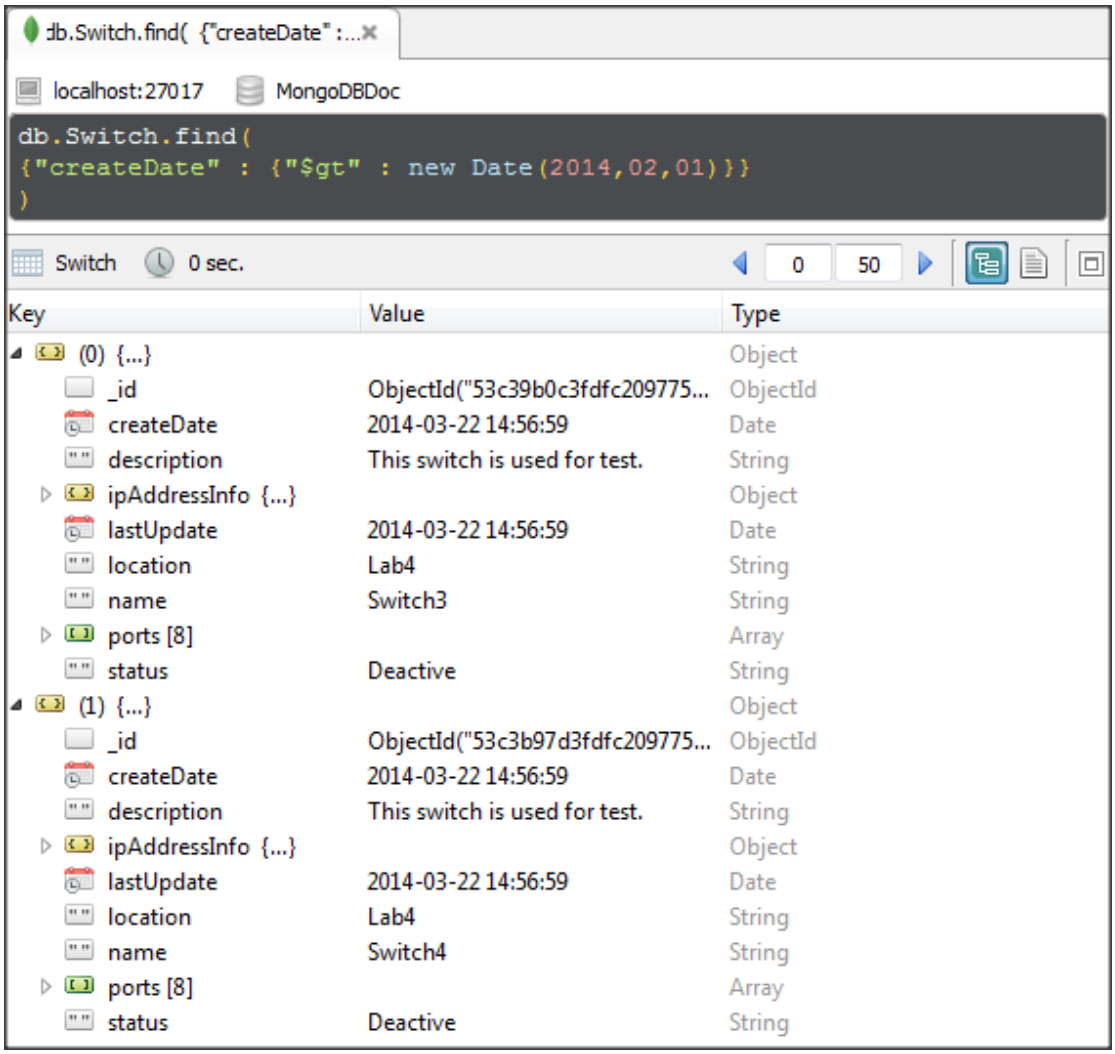
2.1.2. Using operator $gt

Format: {field : {$gt : query-value}}

Description: Matches values that are great than query-value.

Example: Find switches which is created after 01-02-2014.

In Robomongo:



Picture 2.6. Operator $gt in Robomongo

Using MongoTemplate:

```
try {
    // 1. Create date to query
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    Date date = sdf.parse("2014-02-01");
    // 2. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.and("createDate").gt(date);
    // 3. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 4. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
} catch (ParseException e) {
    // Return null list if occur exception
    System.out.println("Parse Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.7. Operator $lt using java

Using Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'createDate' : {'$gt' : ?0}}")
List<Switch> gtOperator(Date date);
```

Picture 2.8. Method using operator $gt in Switch repository

Then use in application:

```
try {
    // 1. Create date to query
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    Date date = sdf.parse("2014-02-01");
    // 2. Query result by using switchRepository
    return switchRepository.gtOperator(date);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
} catch (ParseException e) {
    // Return null list if occur exception
    System.out.println("Parse Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.9. Using repository in application

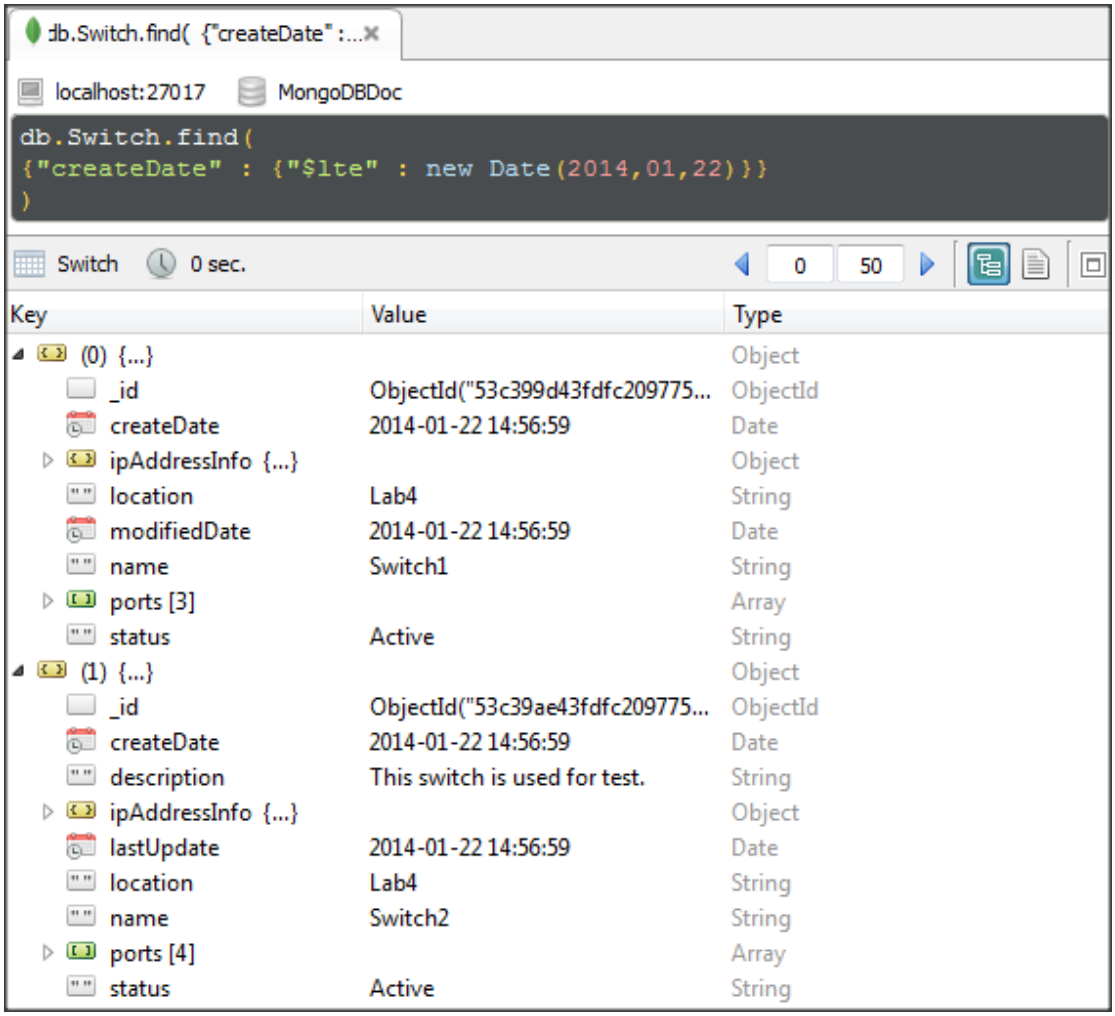
2.1.3. Using operator $lte

Format: {field : {$lte : query-value}}

Description: Matches values that are less than or equal query-value.

Example: Find switches which are created on 22-01-2014 or before.

In Robomongo:



Picture 2.10. Operator $lte in Robomongo

Using MongoTemplate:

```
try {
    // 1. Create date to query
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    Date date = sdf.parse("2014-01-22");
    // 2. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.and("createDate").lte(date);
    // 3. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 4. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
} catch (ParseException e) {
    // Return null list if occur exception
    System.out.println("Parse Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.11. Operator $lte using java

Using Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'createDate' : {'$lte' : ?0}}")
List<Switch> lteOperator(Date date);
```

Picture 2.12. Method using operator $lte in Switch repository

Then use in application:

```
try {
    // 1. Create date to query
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    Date date = sdf.parse("2014-1-22");
    // 2. Query result by using switchRepository
    return switchRepository.lteOperator(date);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
} catch (ParseException e) {
    // Return null list if occur exception
    System.out.println("Parse Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.13. Using repository in application

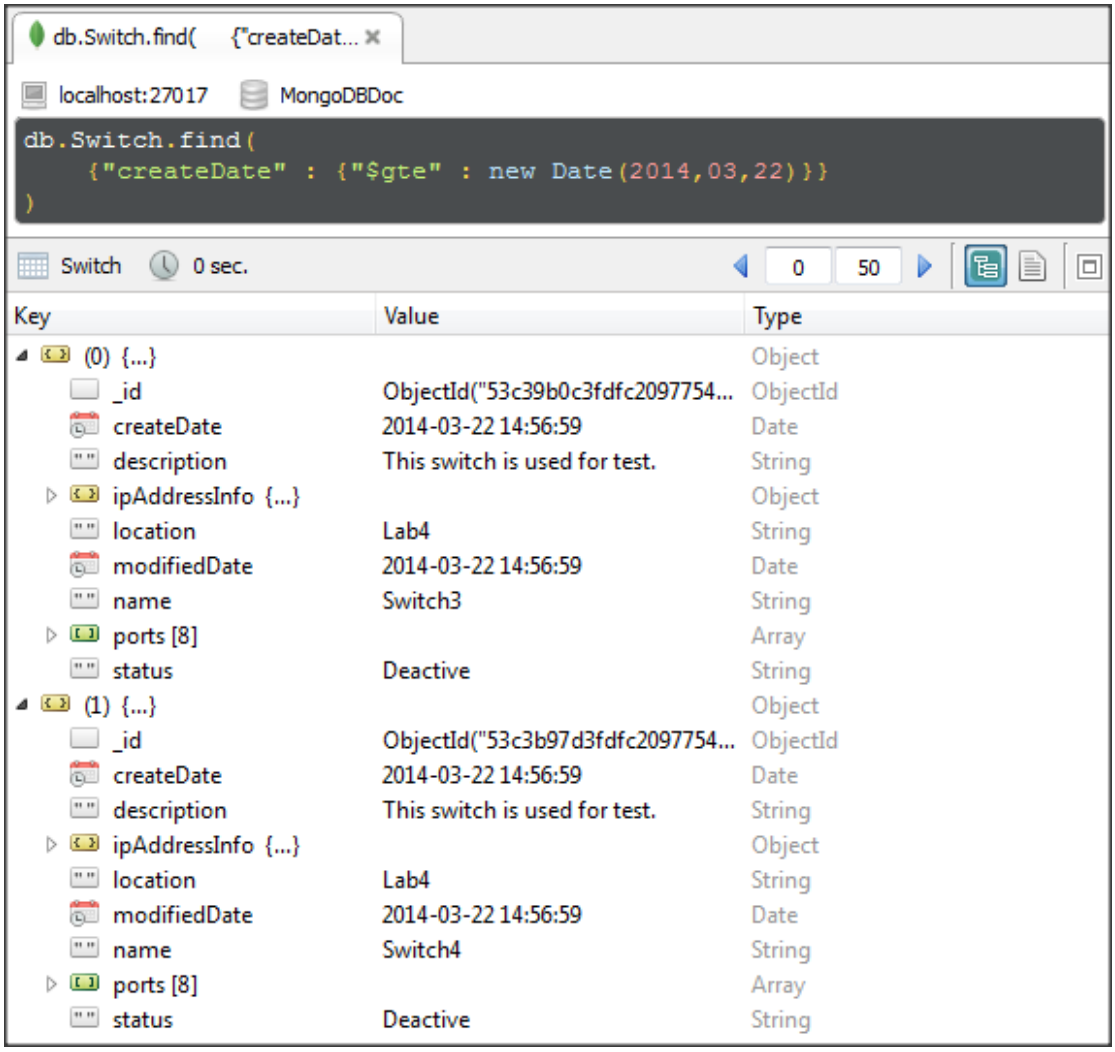
2.1.4. Using operator $gte

Format: {field : {$gte : query-value}}

Description: Matches values that are great than or equal query-value.

Example: Find switches which are created on 22-03-2014 or after.

In Robomongo:



Picture 2.14. Operator $gte in Robomongo

Using MongoTemplate:

```
try {
    // 1. Create date to query
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    Date date = sdf.parse("2014-03-22");
    // 2. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.and("createDate").gte(date);
    // 3. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 4. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
} catch (ParseException e) {
    // Return null list if occur exception
    System.out.println("Parse Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.15. Operator $gte using java

Using Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'createDate' : {'$gte' : ?0}}")
List<Switch> gteOperator(Date date);
```

Picture 2.16. Method using operator $gte in Switch repository

Then use in application:

```
try {
    // 1. Create date to query
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    Date date = sdf.parse("2014-03-22");
    // 2. Query result by using switchRepository
    return switchRepository.gteOperator(date);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
} catch (ParseException e) {
    // Return null list if occur exception
    System.out.println("Parse Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.17. Using repository in application

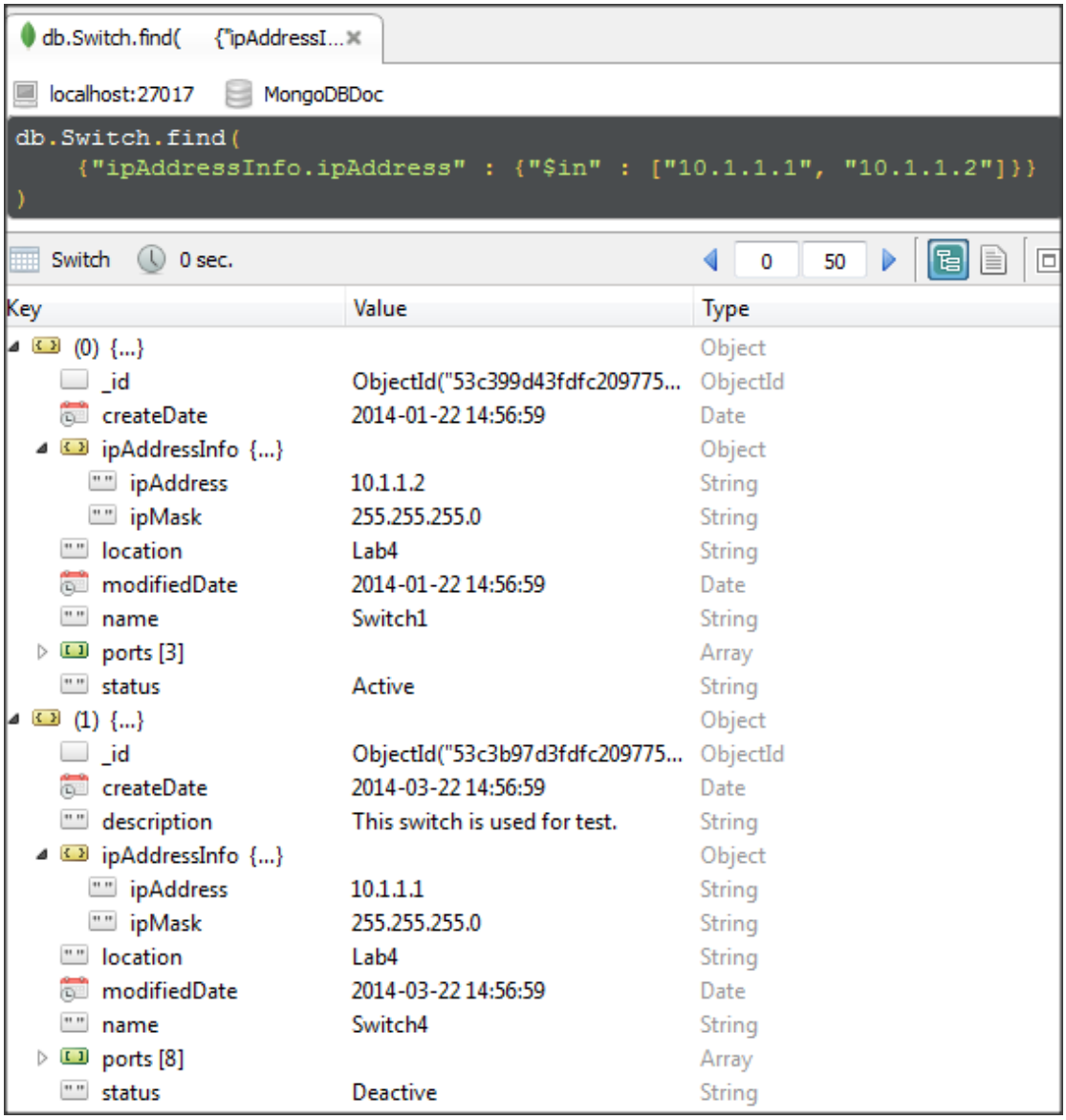
2.1.5. Using operator $in

Format: {field : {$in : array}

Description: Matches values that are one of elements in array.

Example: Find switches which have ip address is 10.1.1.1 or 10.1.1.2

In Robomongo:



Picture 2.18. Operator $in Robomongo

Using MongoTemplate:

```
try {
    // 1. Create ipAddress array
    ArrayList<String> ips = new ArrayList<String>();
    ips.add("10.1.1.1");
    ips.add("10.1.1.2");
    // 2. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.and("ipAddressInfo.ipAddress").in(ips);
    // 3. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 4. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.19. Operator $in using java

Using Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'ipAddressInfo.ipAddress'} : {'$in' : ?0}")
List<Switch> inOperator(ArrayList<String> ips);
```

Picture 2.20. Method using operator $in in Switch repository

Then use in application:

```
try {
    // 1. Create ipAddress array
    ArrayList<String> ips = new ArrayList<String>();
    ips.add("10.1.1.1");
    ips.add("10.1.1.2");
    // 2. Query result by using switchRepository
    return switchRepository.inOperator(ips);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.21. Using repository in application

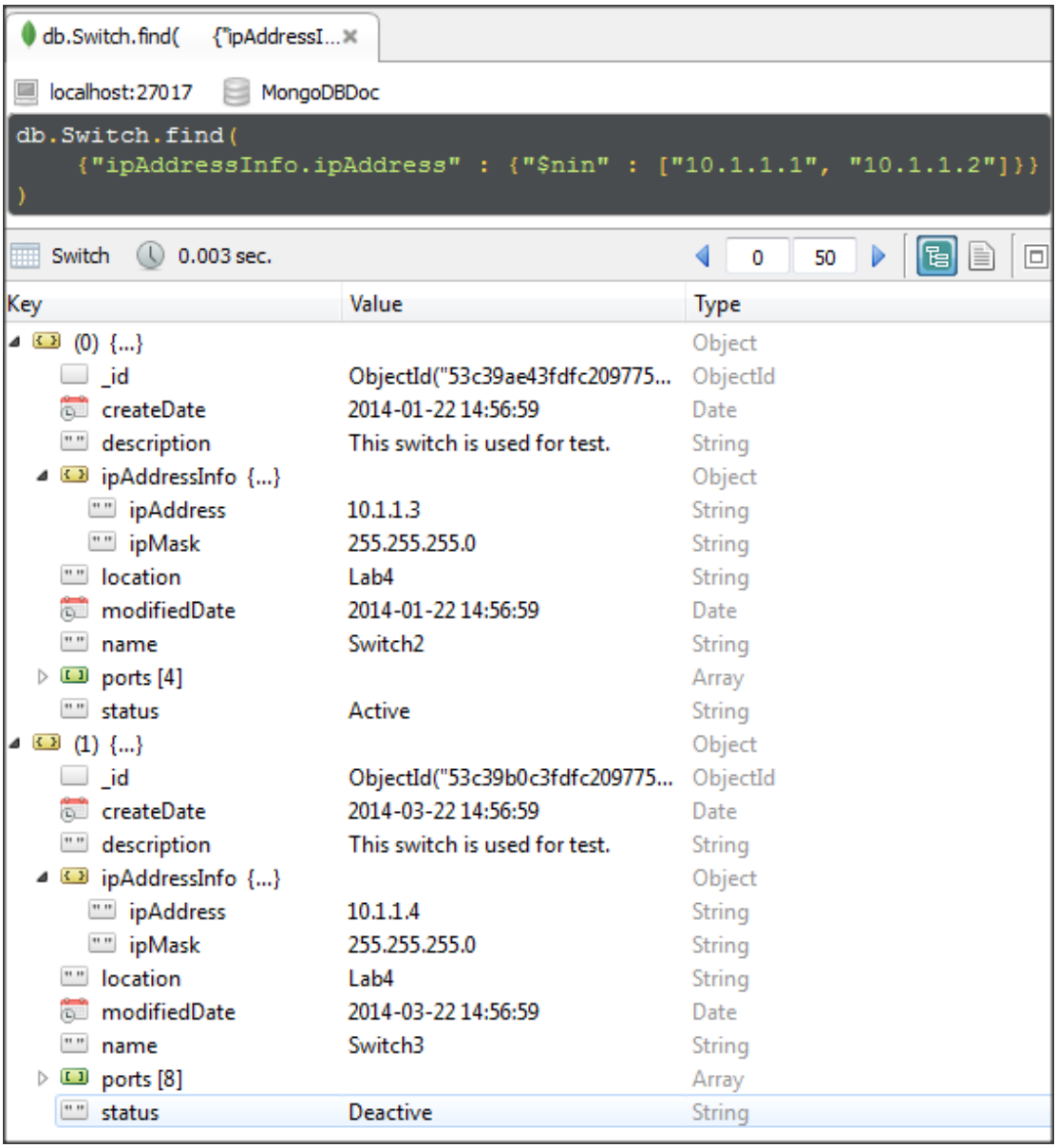
2.1.6. Using operator $nin

Format: {field : {$nin : array}

Description: Matches values that are not one of elements in array.

Example: Find switches which have ip address is 10.1.1.1 or 10.1.1.2

In Robomongo:



Picture 2.22. Operator $nin in Robomongo

Using MongoTemplate:

```
try {
    // 1. Create ipAddress array
    ArrayList<String> ips = new ArrayList<String>();
    ips.add("10.1.1.1");
    ips.add("10.1.1.2");
    // 2. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.and("ipAddressInfo.ipAddress").nin(ips);
    // 3. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 4. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.23. Operator $nin using java

Using Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'ipAddressInfo.ipAddress'} : {'$nin' : ?0}")
List<Switch> ninOperator(ArrayList<String> ips);
```

Picture 2.24. Method using operator $nin in Switch repository

Then use in application:

```
try {
    // 1. Create ipAddress array
    ArrayList<String> ips = new ArrayList<String>();
    ips.add("10.1.1.1");
    ips.add("10.1.1.2");
    // 2. Query result by using switchRepository
    return switchRepository.ninOperator(ips);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.25. Using repository in application

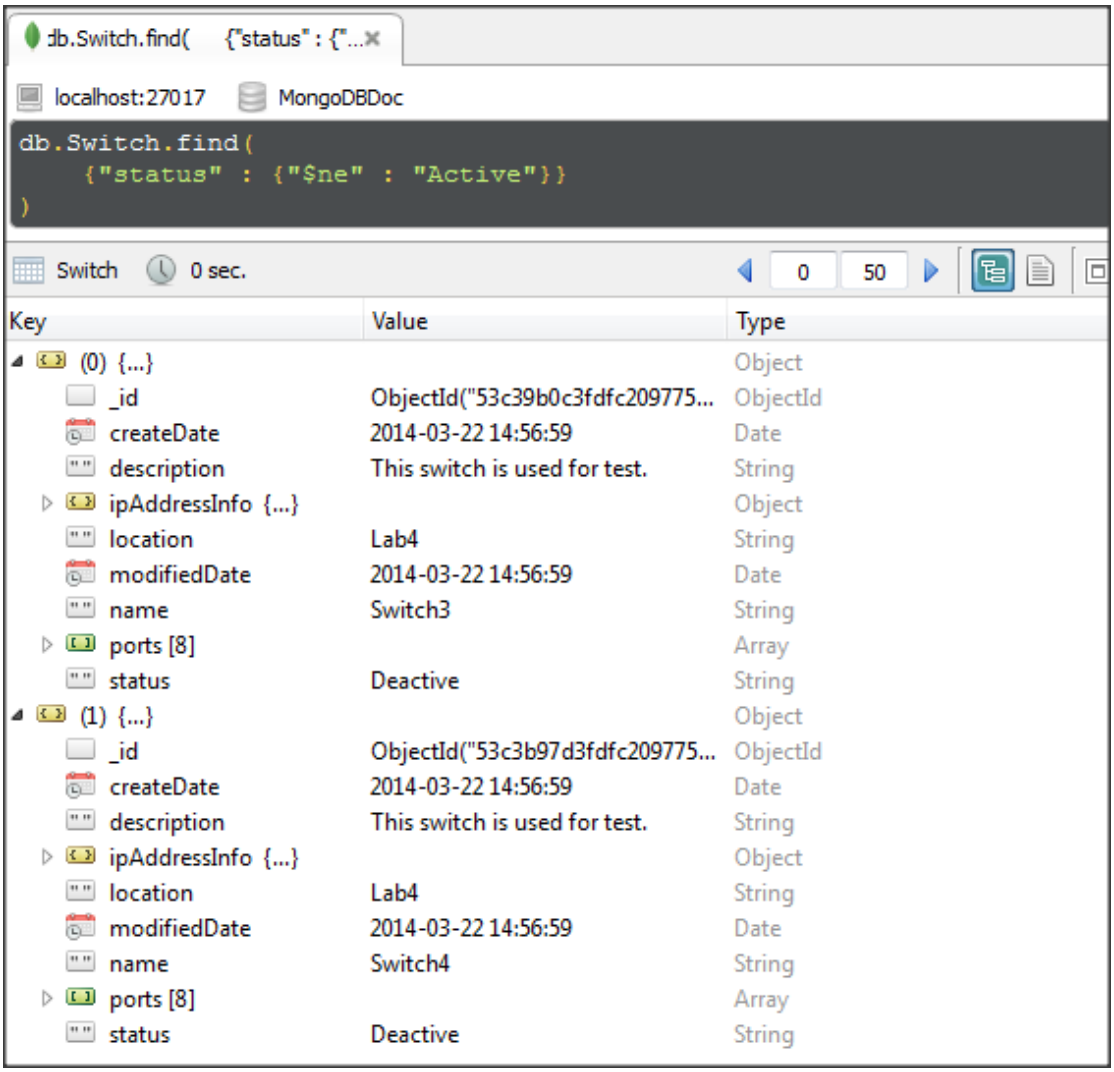
2.1.7. Using operator $ne

Format: {field : {$ne : query-value}

Description: Matches values that are not equal query-value.

Example: Find switches which are not active.

In Robomongo:



Picture 2.26. Operator $ne in Robomongo

Using MongoTemplate:

```
try {
    // 1. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.and("status").ne("Active");
    // 2. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 3. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.27. Operator $ne using java

Using Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'status' : {'$ne' : ?0}}")
List<Switch> neOperator(String status);
```

Picture 2.28. Method using operator $ne in Switch repository

Then use in application:

```
try {
    // 1. Query result by using switchRepository
    return switchRepository.neOperator("Active");
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

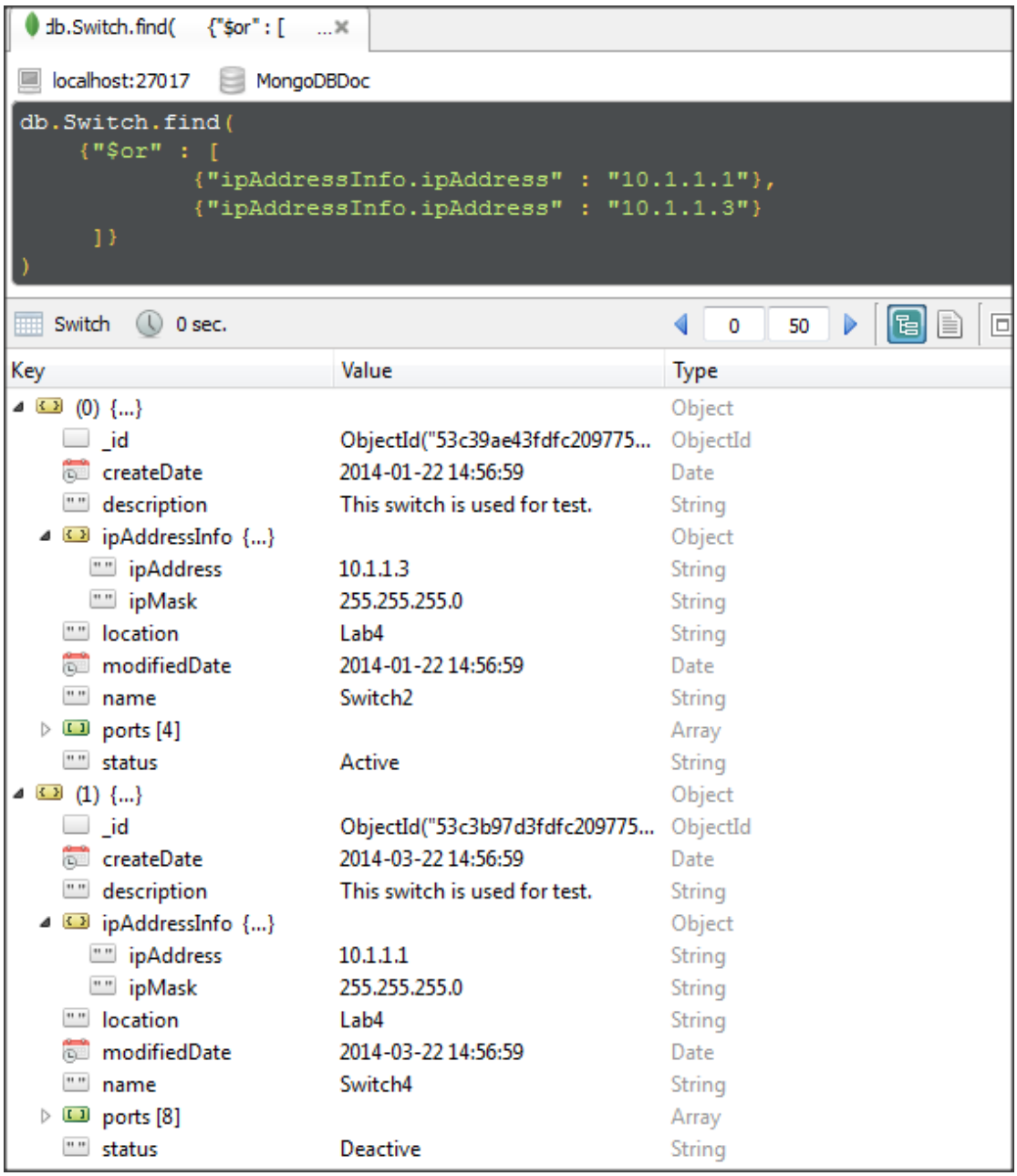
Picture 2.29. Using repository in application

2.1.8. Using operator $or

Format: {$or : [criteria 1, criteria 2, …, criteria *n*]}

Description: Matches values that match least 1 criteria.

Example: Find switches which have ip address is either 10.1.1.1 or 10.1.1.3.



Picture 2.30. Operator $or in Robomongo

Using MongoTemplate:

```
try {
    // 1. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.orOperator(
            Criteria.where("ipAddressInfo.ipAddress").is("10.1.1.1"),
            Criteria.where("ipAddressInfo.ipAddress").is("10.1.1.3"));
    // 2. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 3. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.31. Operator $or using java

Using Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'$or' : [{'ipAddressInfo.ipAddress' : ?0},"
        + "{'ipAddressInfo.ipAddress' : ?1}]}")
List<Switch> orOperator(String ip1, String ip2);
```

Picture 2.32. Method using operator $or in Switch repository

Then use in application:

```
try {
    // 1. Query result by using switchRepository
    return switchRepository.orOperator("10.1.1.1", "10.1.1.3");
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.33. Using repository in application

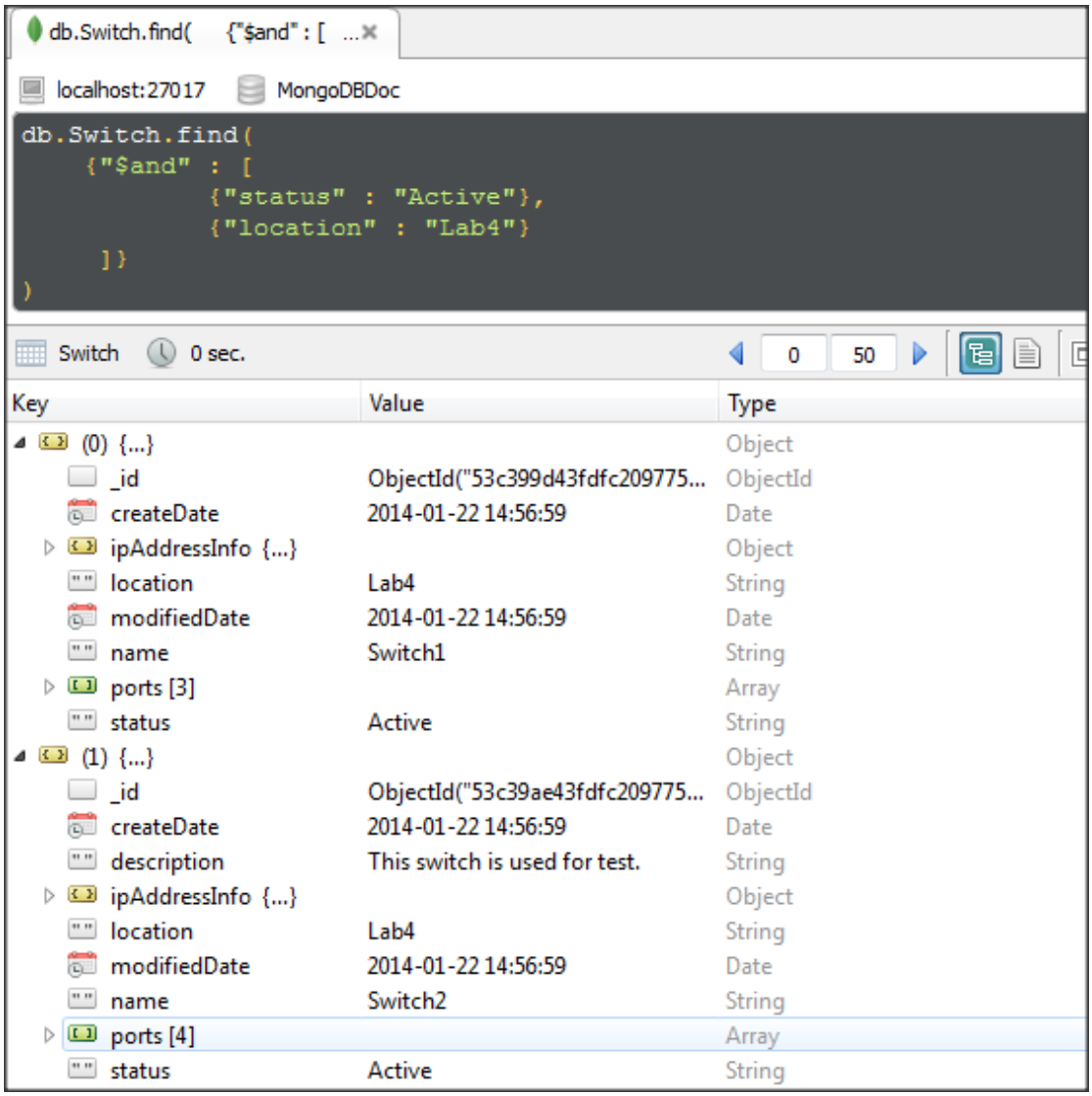
2.1.9. Using operator $and

Format: {$and : [criteria 1, criteria 2, …, criteria *n*]}

Description: Matches values that match all criteria in criteria array.

Example: Find switches which are active and are in Lab4.

In Robomongo:



Picture 2.34. Operator $and in Robomongo

Using MongoTemplate:

```
try {
    // 1. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.andOperator(
            Criteria.where("status").is("Active"),
            Criteria.where("location").is("Lab4"));
    /** You also use $and operator like:
     * criteria.and("status").is("Active");
     * criteria.and("location").is("Lab4");
    **/
    // 2. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 3. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.35. Operator $and using java

Using Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'$and' : [{'status' : 'Active'},"
        + "{'location' : 'Lab4'}]}")
List<Switch> andOperator(String status, String location);
```

Picture 2.36. Method using operator $and in Switch repository

Then use in application:

```
public List<Switch> andOperatorUseRepo() {
    try {
        // 1. Query result by using switchRepository
        return switchRepository.andOperator("Active", "Lab4");
    } catch (MongoException e) {
        // Return null list if occur exception
        System.out.println("Mongo Exc: " + e.getMessage());
        return Collections.<Switch> emptyList();
    }
}
```

Picture 2.37. Using repository in application

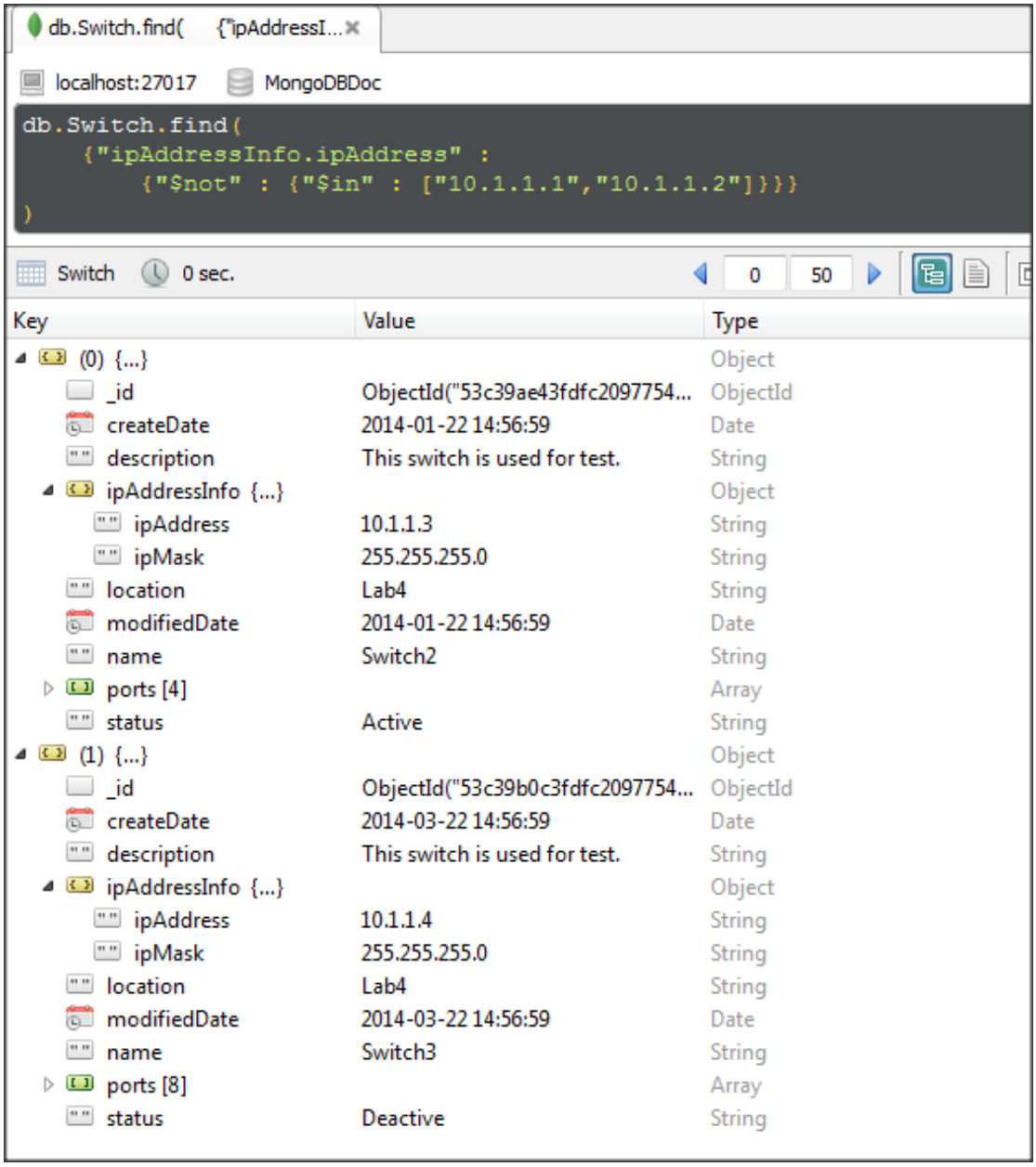
2.1.10. Using operator $not

Format: {field : {$not : operator-expression}}

Description: Matches values that do not match operator-expression.

Example: Find switches which have ip address is not 10.1.1.1 or 10.1.1.3.

In Robomongo:



Picture 2.38. Operator $not in Robomongo

Using Mongotemplate:

```
try {
    // 1. Create ipAddress array
    ArrayList<String> ips = new ArrayList<String>();
    ips.add("10.1.1.1");
    ips.add("10.1.1.2");
    // 2. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.and("ipAddressInfo.ipAddress").not().in(ips);
    // 3. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 4. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.39. Operator $not using java

Using Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'$and' : [{'status' : 'Active'},"
        + "{'location' : 'Lab4'}]}")
List<Switch> andOperator(String status, String location);
```

Picture 2.40. Method using operator $not in Switch repository

Then use in application:

```
try {
    // 1. Create ipAddress array
    ArrayList<String> ips = new ArrayList<String>();
    ips.add("10.1.1.1");
    ips.add("10.1.1.2");
    // 2. Query result by using switchRepository
    return switchRepository.notOperator(ips);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.41. Using repository in application

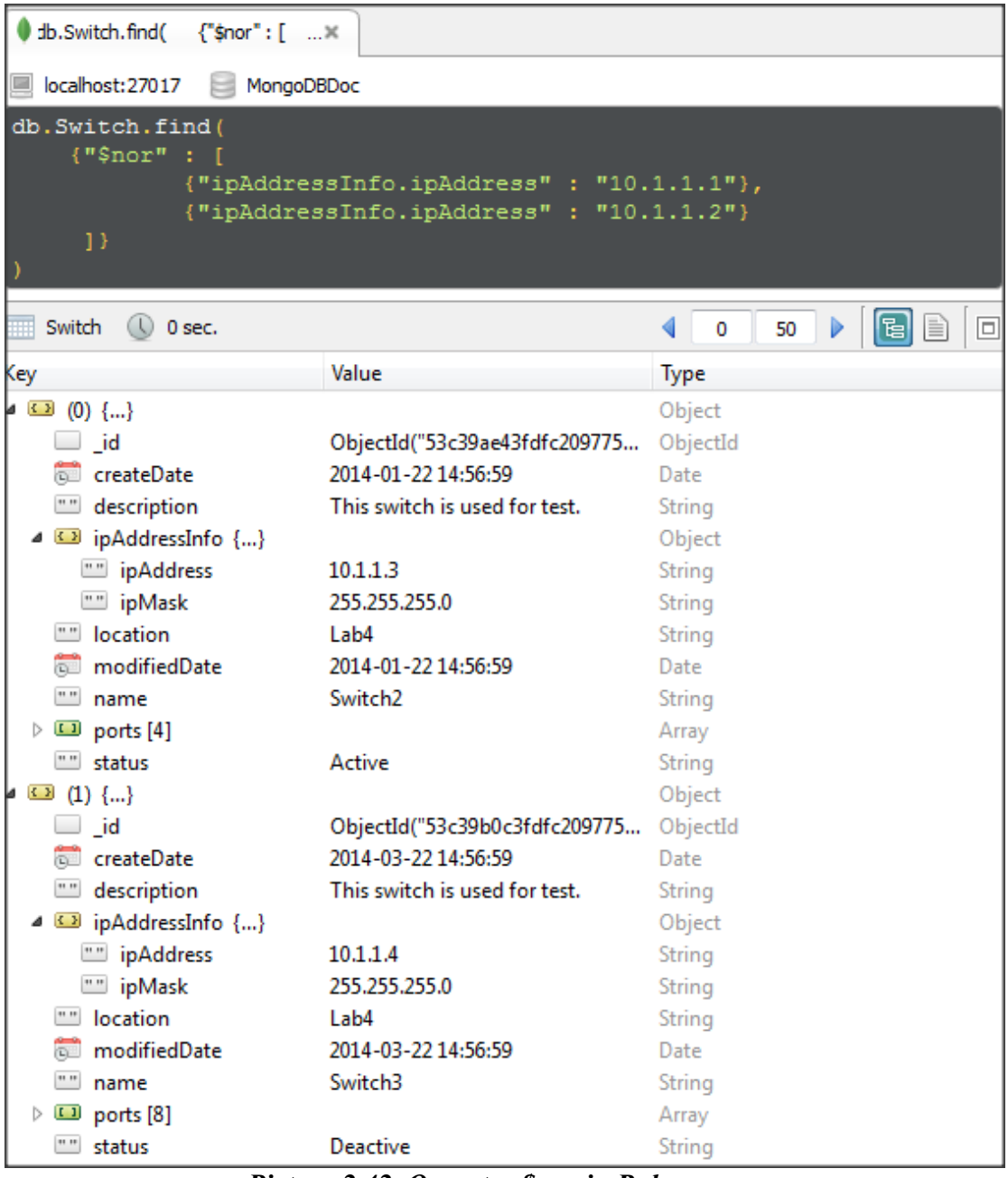
2.1.11. Using operator $nor

Format: {$nor : {criteria 1, criteria 2, . . . , criteria *n*}}

Description: Matches values that do not match all criteria in criteria array.

Example: Find switches which have ip address is not 10.1.1.1 or 10.1.1.3.

In Robomongo:



Picture 2.42. Operator $nor in Robomongo

Using MongoTemplate:

```
try {
    // 1. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.norOperator(
            Criteria.where("ipAddressInfo.ipAddress").is("10.1.1.1"),
            Criteria.where("ipAddressInfo.ipAddress").is("10.1.1.2"));
    // 2. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 3. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.43. Operator $nor using java

Using Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'$nor' : [{'ipAddressInfo.ipAddress' : ?0},"
        + "{'ipAddressInfo.ipAddress' : ?1}]}")
List<Switch> norOperator(String ip1, String ips2);
```

Picture 2.44. Method using operator $nor in Switch repository

Then use in application:

```
try {
    // 1. Query result by using switchRepository
    return switchRepository.norOperator("10.1.1.1", "10.1.1.2");
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.45. Using repository in application

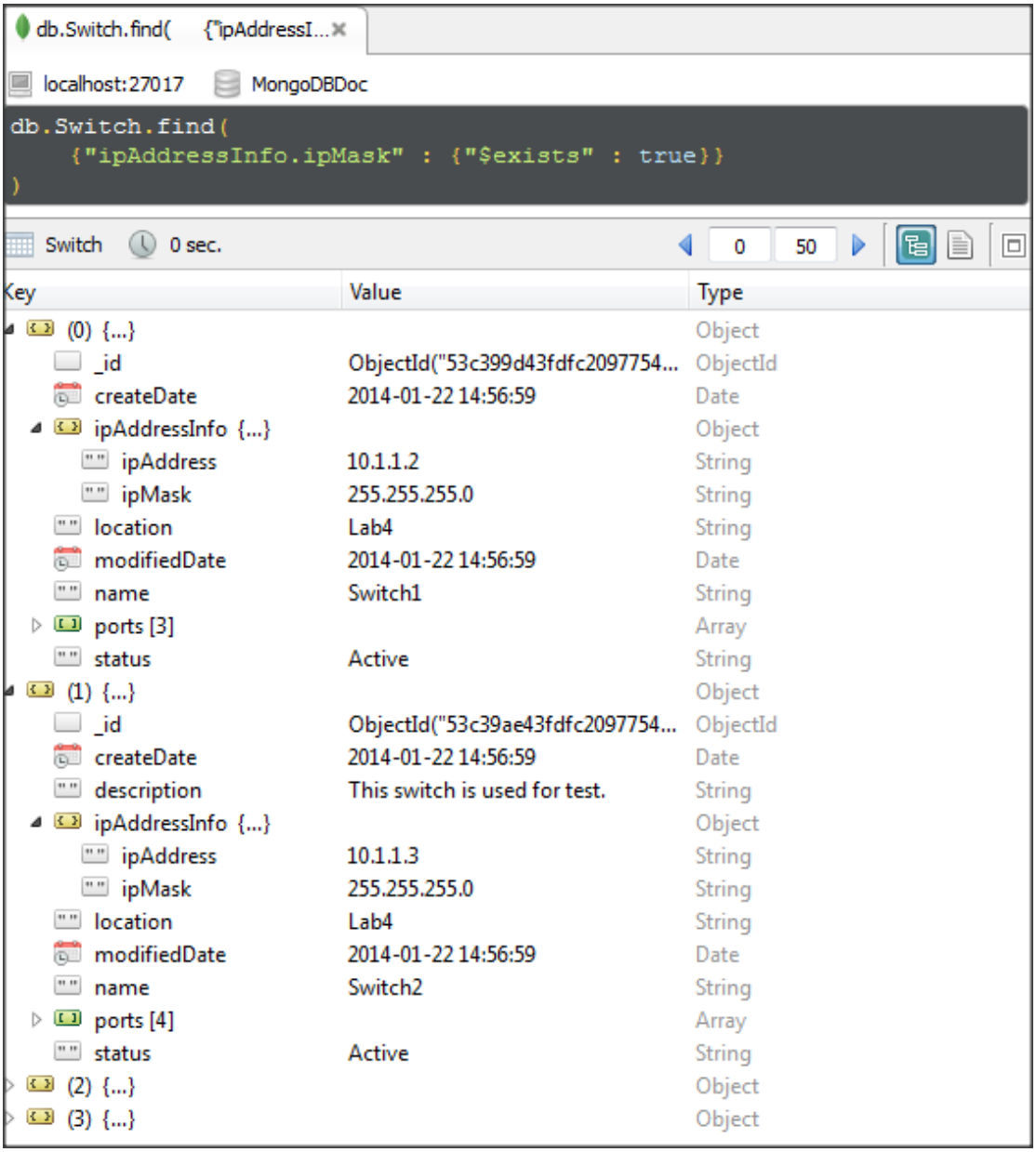
2.1.12. Using operator $exists

Format: {field : {$exists : boolean-value}}

Description: Matches values that have specified field.

Example: Find switches which have field “ipMask” in sub-document ipAddressInfo.

In Robomongo:



Picture 2.46. Operator $exists in Robomongo

Using MongoTemplate:

```
try {
    // 1. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.and("ipAddressInfo.ipMask").exists(true);
    // 2. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 3. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.47. Operator $exists using java

Using Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'ipAddressInfo.ipMask' : {'$exists' : ?0}}")
List<Switch> existsOperator(boolean isMaks);
```

Picture 2.48. Method using operator $exists in Switch repository

Then use in application:

```
try {
    // 1. Query result by using switchRepository
    return switchRepository.existsOperator(true);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.49. Using repository in application

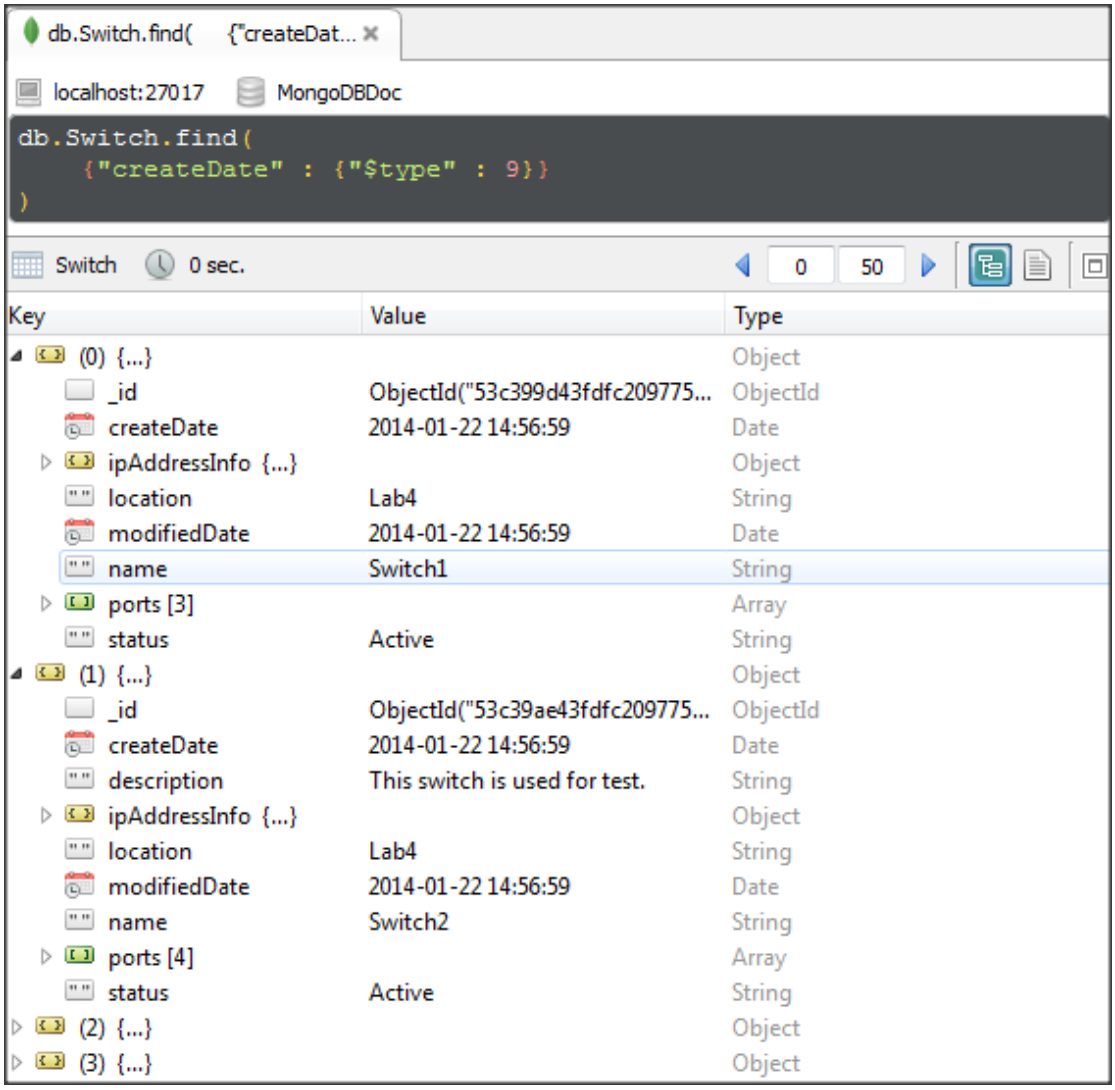
2.1.13. Using operator $type

Format: {field : {$type : typeCode}}

Description: Matches values that have type of field is equal typeCode. (See full typeCode in Types of data).

Example: Find switches which have field createDate is date type.

In Robomongo:



Picture 2.50. Operator $type in Robomongo

Using MongoTemplate:

```
try {
    // 1. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.and("createDate").type(9);
    // 2. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 3. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.51. Operator $type using java

Using Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'createDate' : {'$type' : ?0}}")
List<Switch> typeOperator(int typeCode);
```

Picture 2.52. Method using operator $type in Switch repository

Then use in application:

```
try {
    // 1. Query result by using switchRepository
    return switchRepository.typeOperator(9);
    // 9 is type of date in MongoDB
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.53. Using repository in application

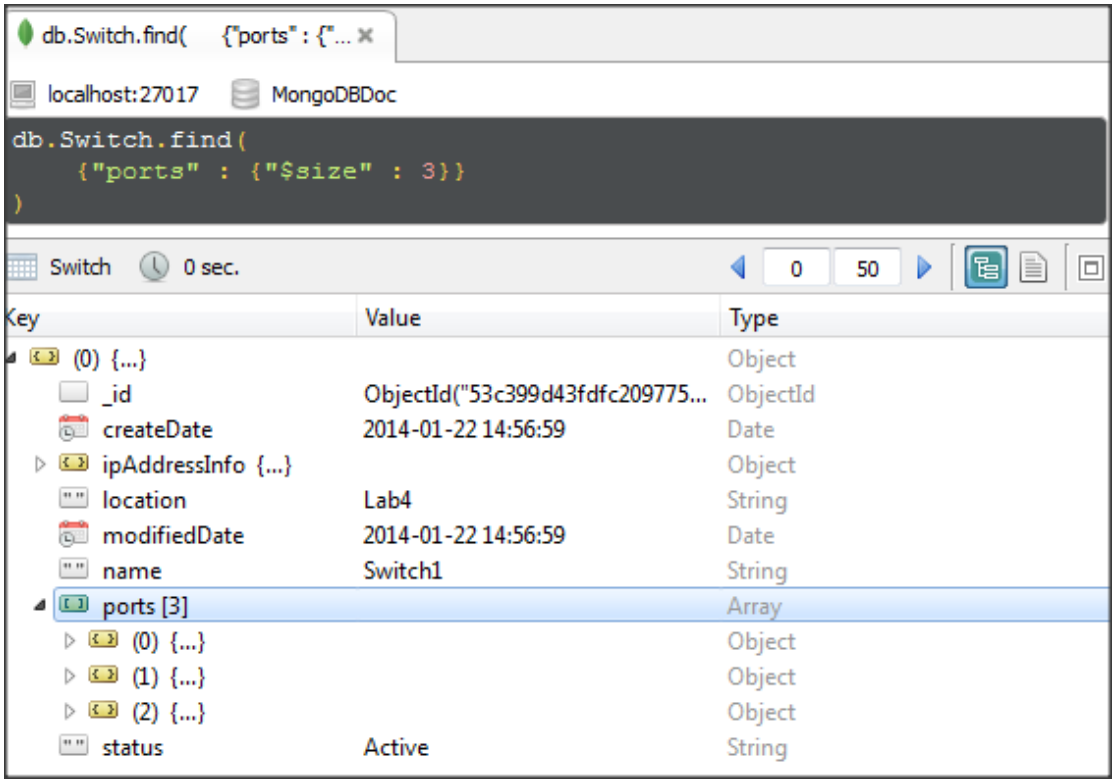
2.1.14. Using operator $size

Format: {field : {$size : numberOfElement}}

Description: Matches values have size of field is equal numberOfElement.

Example: Find switches which have 3 ports.

In Robomongo:



Picture 2.54. Operator $size in Robomongo

Using MongoTemplate:

```
try {
    // 1. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.and("ports").size(3);
    // 2. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 3. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.55. Operator $size using java

Using Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'ports' : {'$size' : ?0}}")
List<Switch> sizeOperator(int numberOfElements);
```

Picture 2.56. Method using operator $size in Switch repository

Then use in application:

```
try {
    // 1. Query result by using switchRepository
    return switchRepository.sizeOperator(3);
    // 9 is type of date in MongoDB
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.57. Using repository in application

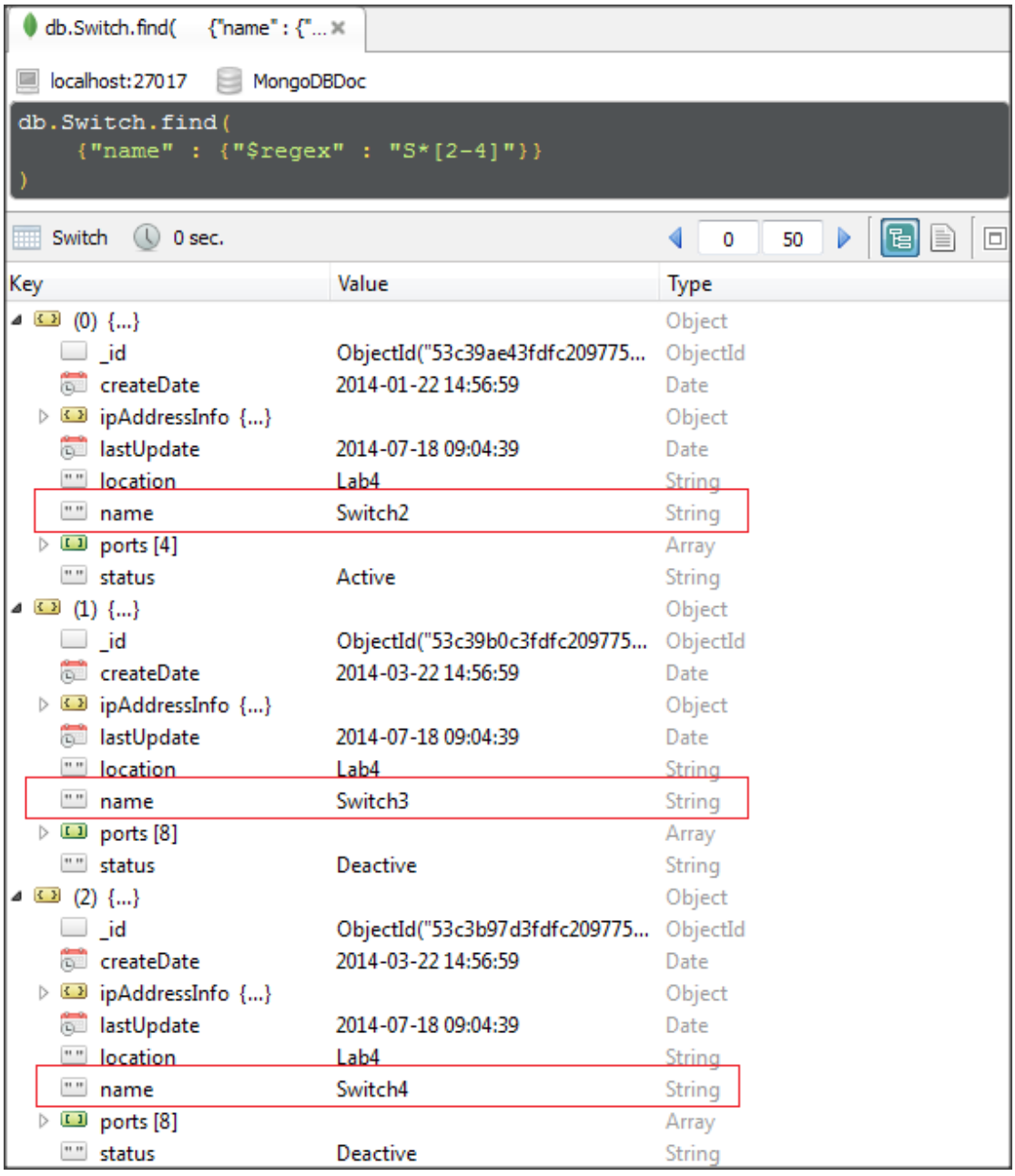
2.1.15. Using operator $regex

Format: {field : {$regex : pattern}}

Description: Matches values match with pattern (MongoDB uses Perl compatible regular expressions).

Example: Find switches which have name is S*[2-4].

In Robomongo:



Picture 2.58. Operator $regex in MongoTemplate

Using MongoTemplate:

```
try {
    // 1. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.regex("S*[2-4]");
    // 2. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 3. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.59. Operator $regex using java

Using Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'name' : {'$regex' : ?0}}")
List<Switch> regexOperator(String pattern);
```

Picture 2.60. Method using operator $regex in Switch repository

Then use in application:

```
try {
    // 1. Query result by using switchRepository
    return switchRepository.regexOperator("S*[2-4]");
    // 9 is type of date in MongoDB
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.61. Using repository in application

2.1.16. Combine read operators

To get necessary documents, we usually need build complex query criteria. This section will show how to build complex query criteria by using combine operators.

a. Using operators $and, $or, $in, $size, $lt

We need get switches have:

2. Status is active.
3. Have 4 ports.
4. IP address is: 10.1.1.2 or 10.1.1.3
5. Create before Feb 2014

So we can build a query criteria like:

```
2. {"status" : "Active"}
3. {"ports" : {"$size" : 4}}
4. {"ipAddressInfo.ipAddress" : {"$in" : ["10.1.1.2", "10.1.1.3"]}}
5. {"createDate" : {"$lt" : new Date(2014,02,01)}}
```

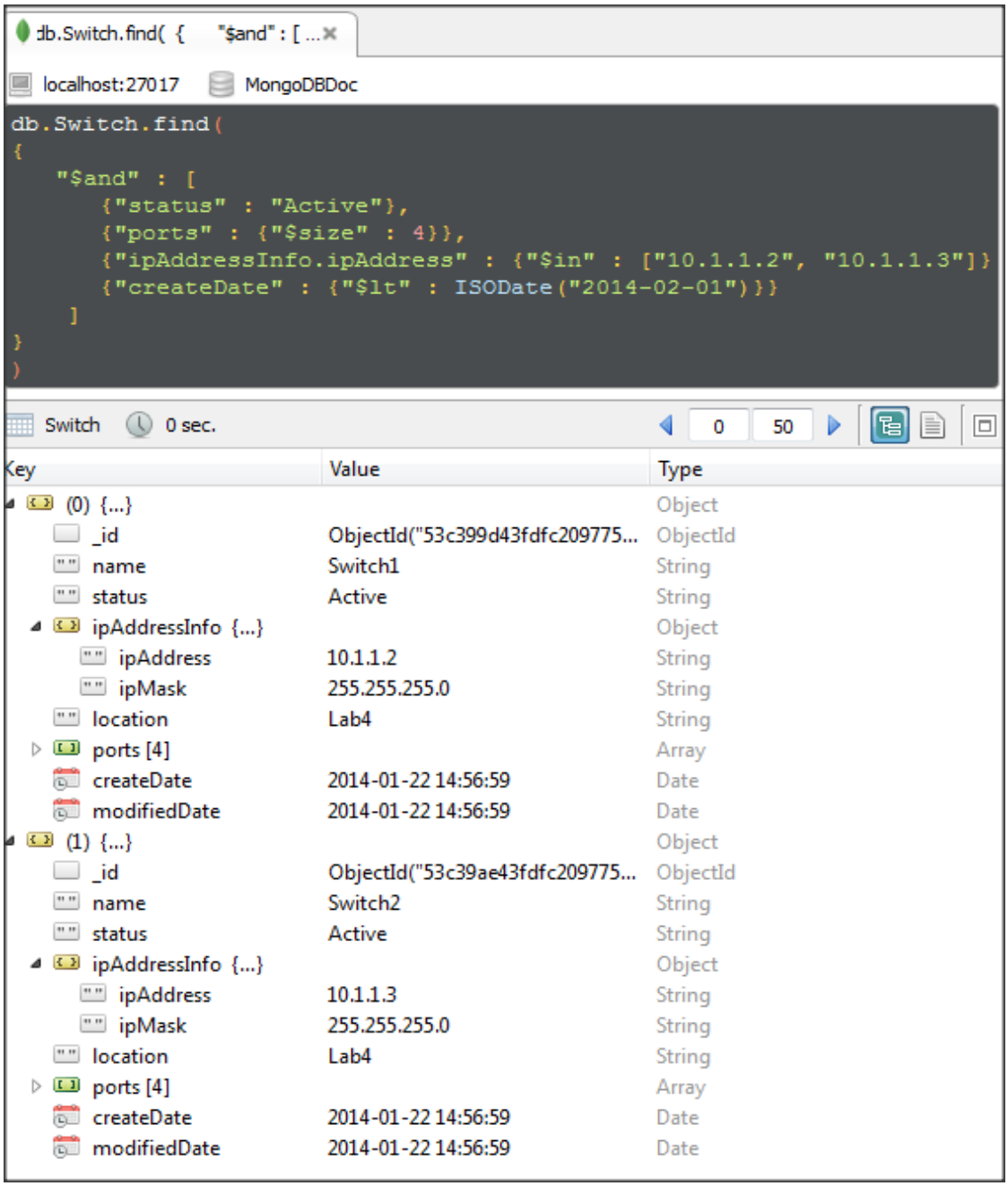
In above criteria, you can change

```
{"ipAddressInfo.ipAddress" : {"$in" : ["10.1.1.2", "10.1.1.3"]}}
```

to

```
{"$or" : [
    {"ipAddressInfo.ipAddress" : "10.1.1.1"},
    {"ipAddressInfo.ipAddress" : "10.1.1.2"}
]}
```

In Robomongo:



Picture 2.62. Combine operators – example 1.

Use MongoTemplate:

```
try {
    // 1. Create ipAddress array
    ArrayList<String> ips = new ArrayList<String>();
    ips.add("10.1.1.1");
    ips.add("10.1.1.2");
    // 2. Create date to query
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    Date date = sdf.parse("2014-02-01");
    // 3. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.and("status").is("Active");
    criteria.and("ports").size(4);
    criteria.and("ipAddressInfo.ipAddress").in(ips);
    criteria.and("createDate").lt(date);
    // 4. Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // 5. Query result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
} catch (ParseException e) {
    // Return null list if occur exception
    System.out.println("Parse Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.63. Combine operators using java – example 1

Use Repository:

- First, we need create method in our SwitchRepository:

```
@Query(value="{'$and' : ["
        + "{'status' : ?0},"
        + "{'ports' : {'$size' : ?1}},"
        + "{'ipAddressInfo.ipAddress' : {'$in' : ?2}},"
        + "{'createDate' : {'$lt' : ?3}}"
        + "]}")
List<Switch> getSwitches(String status, int sizeOfPort, ArrayList<String> ips, Date date);
```

Picture 2.63. Combine operators using repository - example 1

- Then use in application:

```
try {
    // 1. Create ipAddress array
    ArrayList<String> ips = new ArrayList<String>();
    ips.add("10.1.1.1");
    ips.add("10.1.1.2");
    // 2. Create date to query
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    Date date = sdf.parse("2014-02-01");
    // 3. Query result by using switchRepository
    return switchRepository.getSwitches("Active", 4, ips, date);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
} catch (ParseException e) {
    // Return null list if occur exception
    System.out.println("Parse Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.65. Using repository in application

b. Using operators $and, $lt, $ne, $size, $exists, $type

We need get switches with:

a. Have 8 ports.

b. IP address is not “10.1.1.1”.

c. Exist createDate and modifiedDate fields.

d. And type of them are date type.

We can build a query criteria like:

a. {"ports" : {"$size" : 8}}

b. {"ipAddressInfo.ipAddress" : {"$ne" : "10.1.1.1"}}

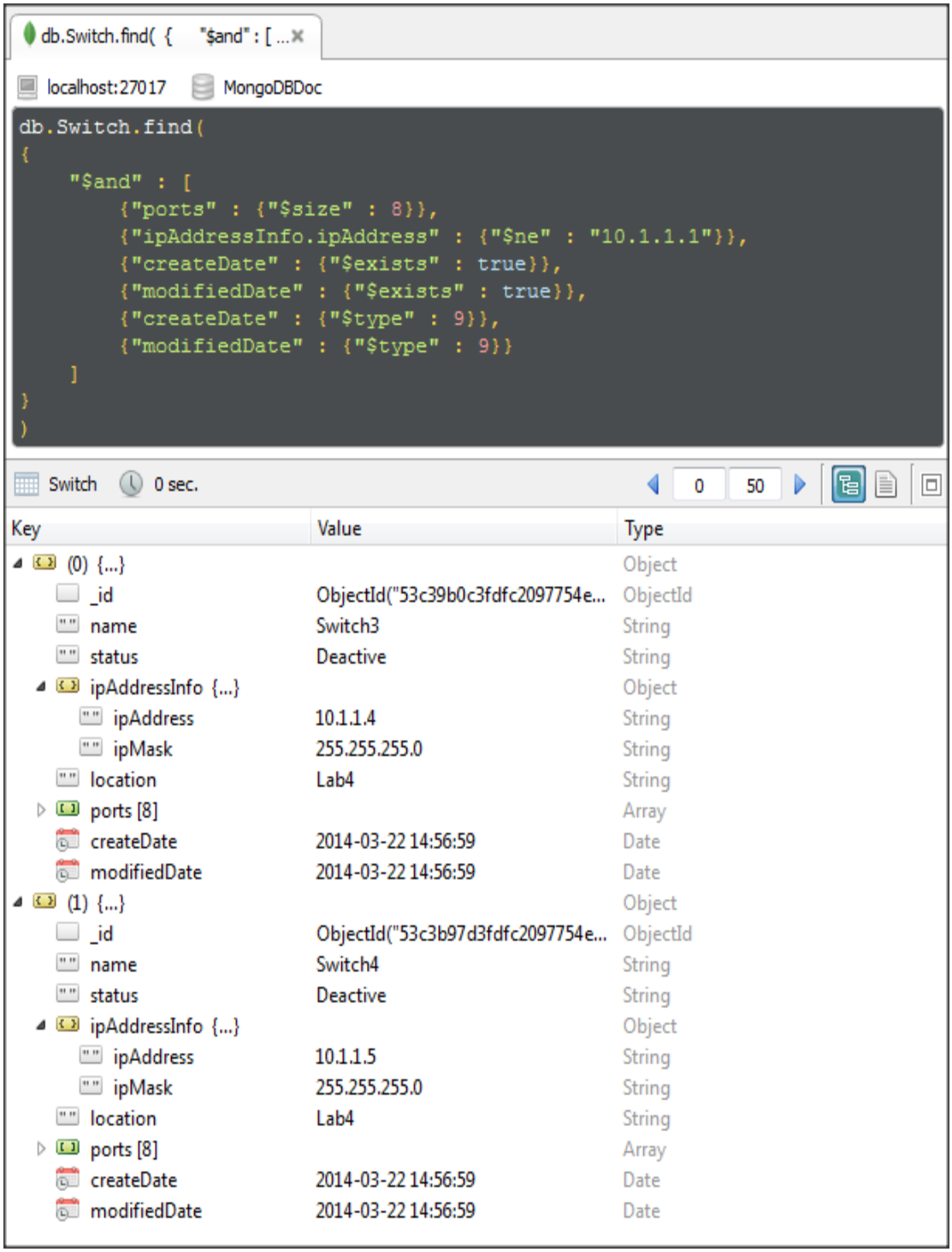
c. {"createDate" : {"$exists" : true}}

{"modifiedDate" : {"$exists" : true}}

d. {"createDate" : {"$type" : 9}}

{"modifiedDate" : {"$type" : 9}}

In Robomongo:



Picture 2.66. Combine operators – example 2

Use MongoTemplate:

```
try {
    // 1. Build criteria for query
    Criteria criteria = new Criteria();
    criteria.and("ports").size(8);
    criteria.and("ipAddressInfo.ipAddress").ne("10.1.1.1");
    criteria.and("createDate").exists(true);
    criteria.and("createDate").type(9);
    criteria.and("modifiedDate").exists(true);
    criteria.and("modifiedDate").type(9);
    // Create query object
    Query query = new Query();
    query.addCriteria(criteria);
    // Return result by using mongoTemplate
    return mongoTemplate.find(query, Switch.class);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.67. Combine operators using java – example 2

Use Repository:

First, we need create method in our SwitchRepository:

```
@Query(value="{'$and' : ["
        + "{'ports' : {'$size' : ?0}},"
        + "{'ipAddressInfo.ipAddress' : {'$ne' : ?1}},"
        + "{'createDate' : {'$exists' : true}},"
        + "{'createDate' : {'$type' : ?2}},"
        + "{'modifiedDate' : {'$exists' : true}},"
        + "{'modifiedDate' : {'$type' : ?2}}"
        + "]}")
List<Switch> getSwitches(int sizeOfPort, String notIpAddress, int type);
```

Picture 2.68. Combine operators using repository - example 2

Then use in application:

```
try {
    return switchRepository.getSwitches(8, "10.1.1.1", 9);
} catch (MongoException e) {
    // Return null list if occur exception
    System.out.println("Mongo Exc: " + e.getMessage());
    return Collections.<Switch> emptyList();
}
```

Picture 2.69. Using repository in application

2.2. Modifies operations

In MongoDB, both update() and save() modify existing documents in a collection. Update() provides additional control over the modification. For example, you can modify existing data or modify a group of documents that match a query with Update(). So we will practice with update() operation.

See full operators in Update Operators.

Syntax of update():

db.collectionName.update(<query>, <update>, <optional>)

Syntax of save():

db.collectionName.save(<document>)

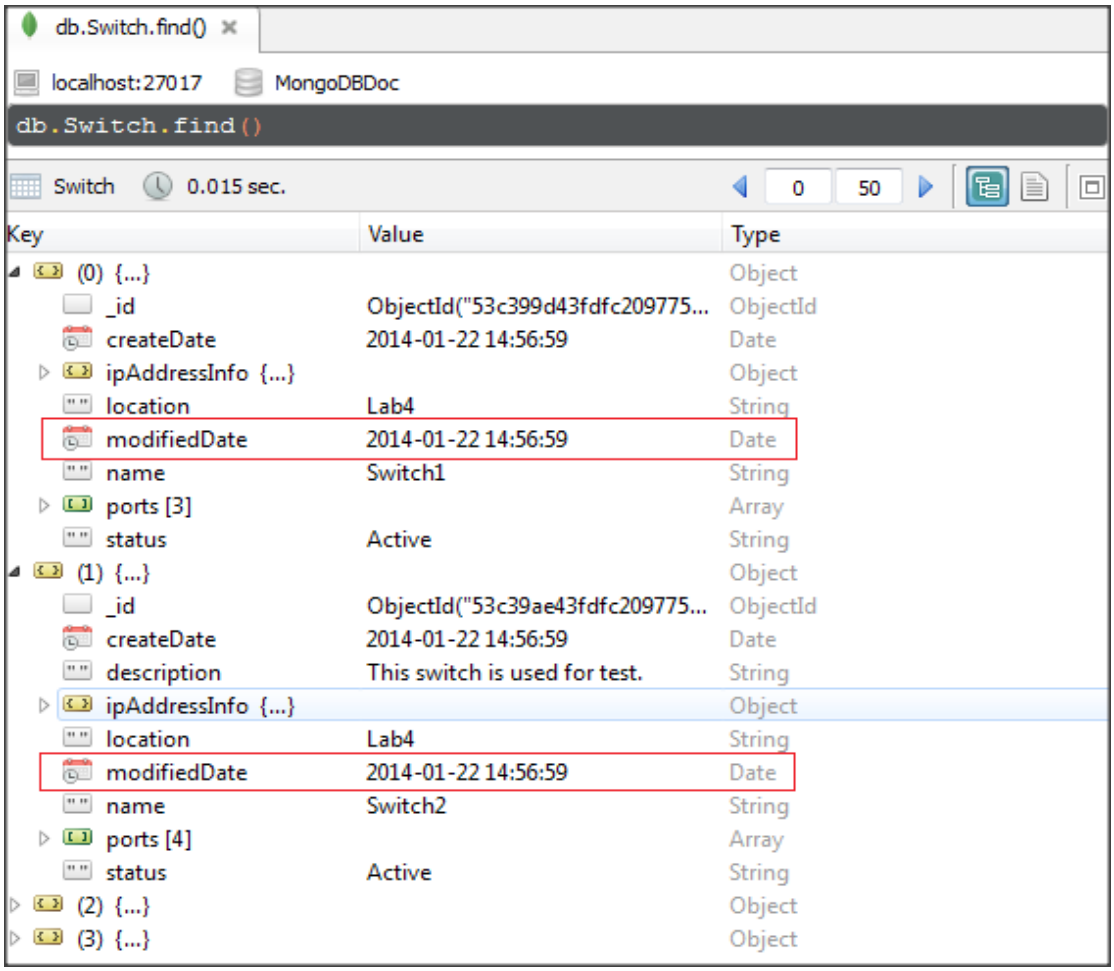
2.2.1. Using operators $rename

Format: {$rename : {oldName, newName}}

Description: Change name of field to new name.

Example: Rename modifiedDate field to lastUpdate for all Switches.

Before update:

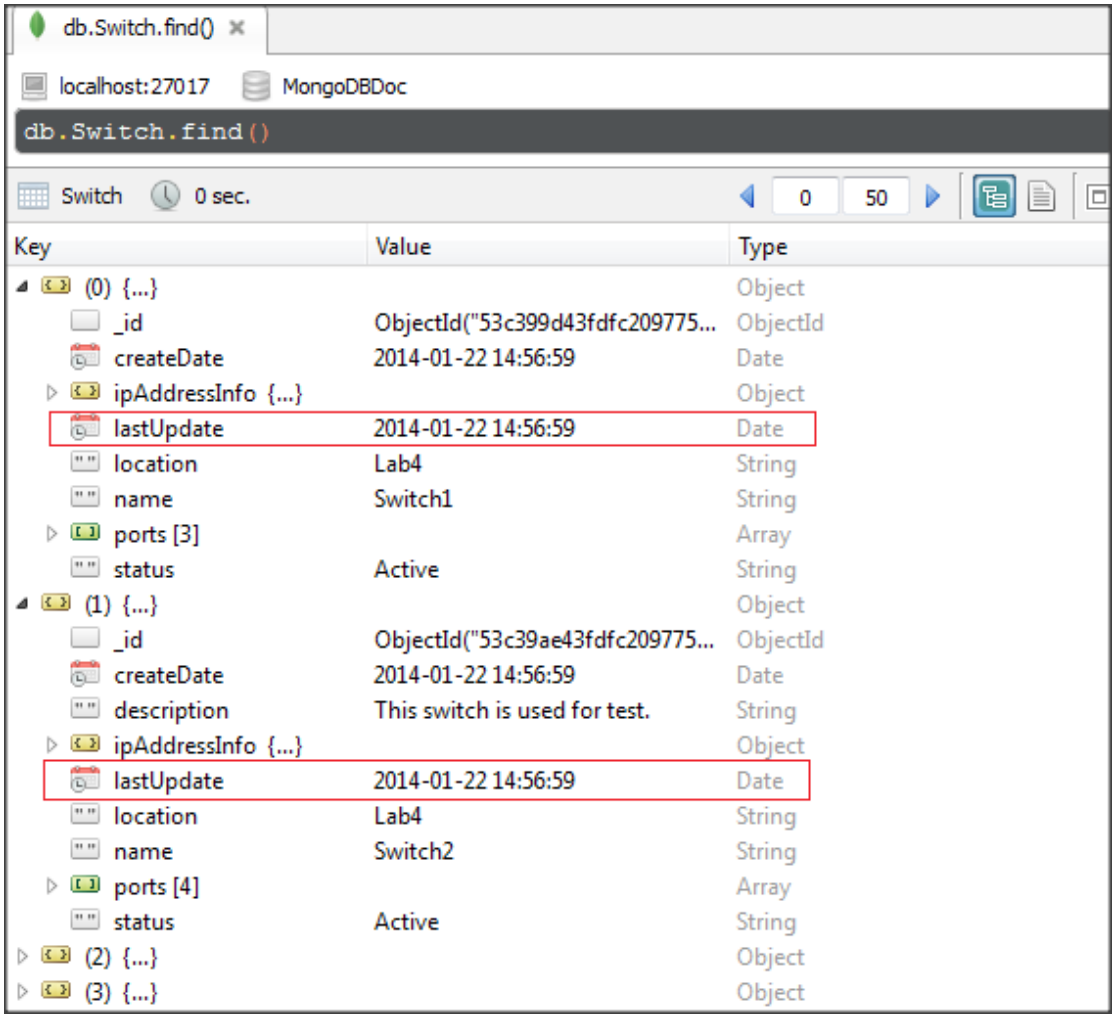


Picture 2.70. Operator $rename – before update

Command to update:

```
db.Switch.update(
    {"_id" : {"$exists" : true}},
    {"$rename" : {"modifiedDate" : "lastUpdate"}},
    {"multi" : true}
)
```

The result:



Picture 2.71. Operator $rename – result of update

Using MongoTemplate:

```
try {
    // 1. Select all documents in database to update
    Query query = new Query();
    query.addCriteria(Criteria.where("_id").exists(true));
    // 2. Build update object
    Update update = new Update();
    update.rename("modifiedDate", "lastUpdate");
    // 3. Update by using mongoTemplate
    mongoTemplate.updateMulti(query, update, Switch.class);
} catch (MongoException e) {
    System.out.println("Mongo Exc: " + e.getMessage());
}
```

Picture 2.72. Operator $rename using java

2.2.2. Using operators $set

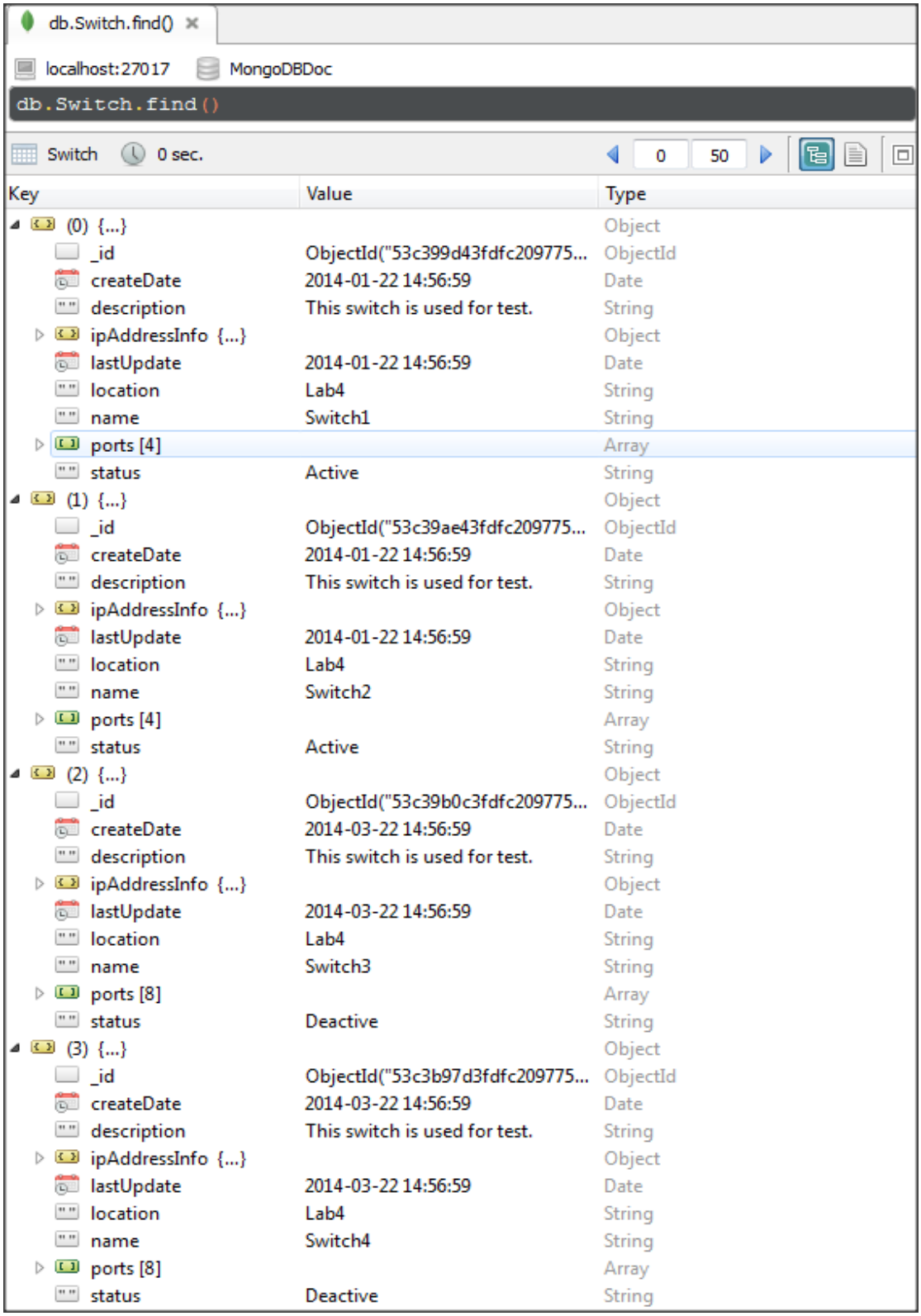
Format: {$set : {field : update-value}}

Description: Change the value of field or create new field with update-value.

Example: Add “switchTable” field for Switche1 like:

```
switchTable : [
        {port : 1, station : “Mac A”, vlan : 100},
        {port : 2, station : “Mac B”, vlan : 100},
        {port : 3, station : “Mac C”, vlan : 200},
        {port : 4, station : “Mac D”, vlan : 200}
]
```

Before update:

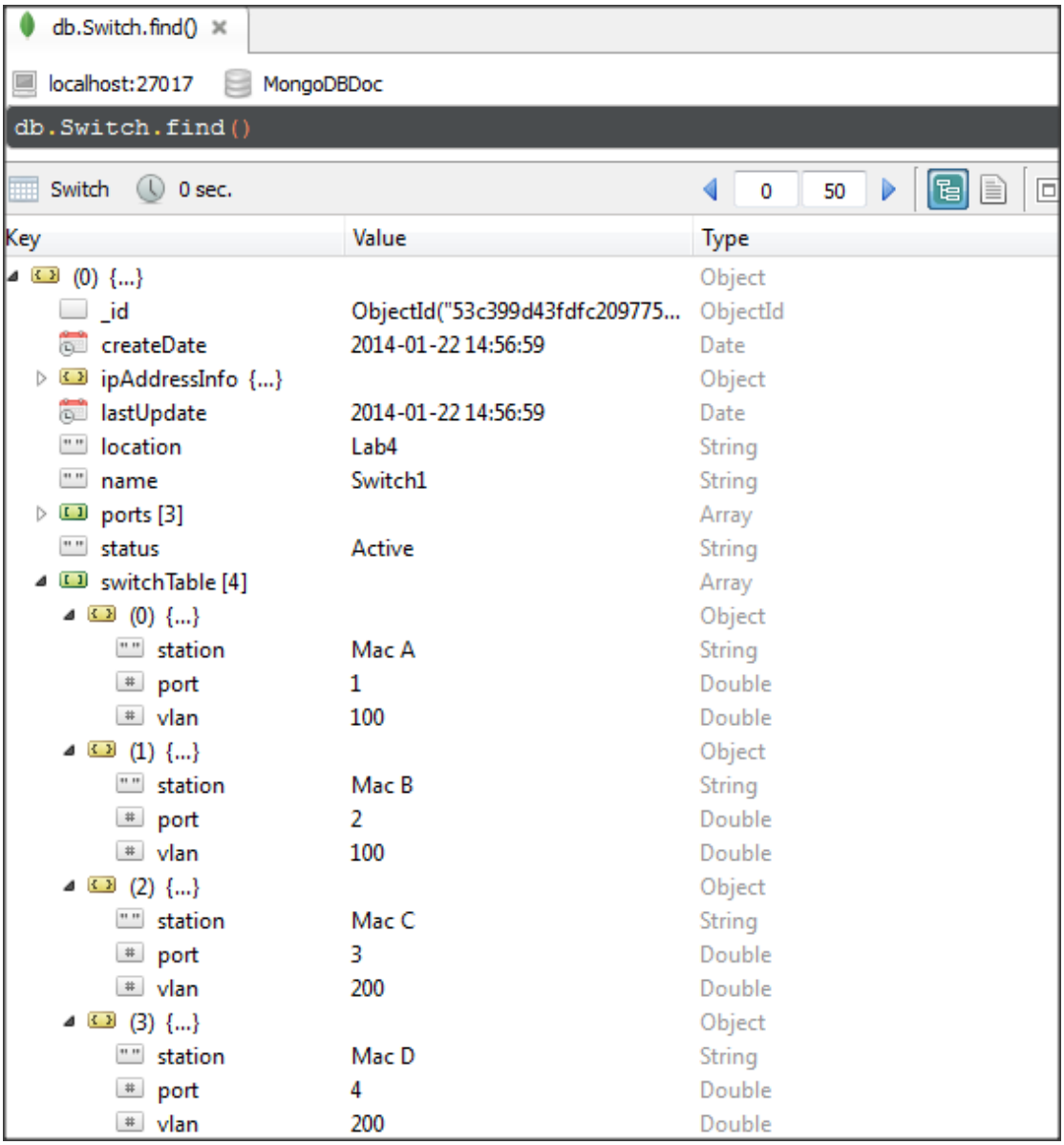


Picture 2.73. Operator $set – before update without switchTable field

Command to update:

```
db.Switch.update(
{"name" : "Switch1"},
{"$set" : {"switchTable" : [
            {"station" : "Mac A", "port" : 1, "vlan" : 100},
            {"station" : "Mac B", "port" : 2, "vlan" : 100},
            {"station" : "Mac C", "port" : 3, "vlan" : 200},
            {"station" : "Mac D", "port" : 4, "vlan" : 200}
    ]}}
)
```

The result:



Picture 2.74. Operator $set – result of update with switchTable field

Using MongoTemplate:

```
try {
    // 1. Select all documents in database to update
    Query query = new Query();
    query.addCriteria(Criteria.where("name").is("Switch1"));
    // 2. Build update object
    BasicDBObject element1 = new BasicDBObject();
    element1.put("station", "Mac A");
    element1.put("port", "1");
    element1.put("vlan", "100");
    BasicDBObject element2 = new BasicDBObject();
    element2.put("station", "Mac B");
    element2.put("port", "2");
    element2.put("vlan", "100");
    BasicDBObject element3 = new BasicDBObject();
    element3.put("station", "Mac C");
    element3.put("port", "3");
    element3.put("vlan", "200");
    BasicDBObject element4 = new BasicDBObject();
    element3.put("station", "Mac D");
    element3.put("port", "4");
    element3.put("vlan", "200");
    ArrayList<BasicDBObject> value = new ArrayList<BasicDBObject>();
    value.add(element1);
    value.add(element2);
    value.add(element3);
    value.add(element4);
    Update update = new Update();
    update.set("switchTable", value);
    // 3. Update by using mongoTemplate
    mongoTemplate.updateMulti(query, update, Switch.class);
} catch (MongoException e) {
    System.out.println("Mongo Exc: " + e.getMessage());
}
```

Picture 2.75. Operator $set using java

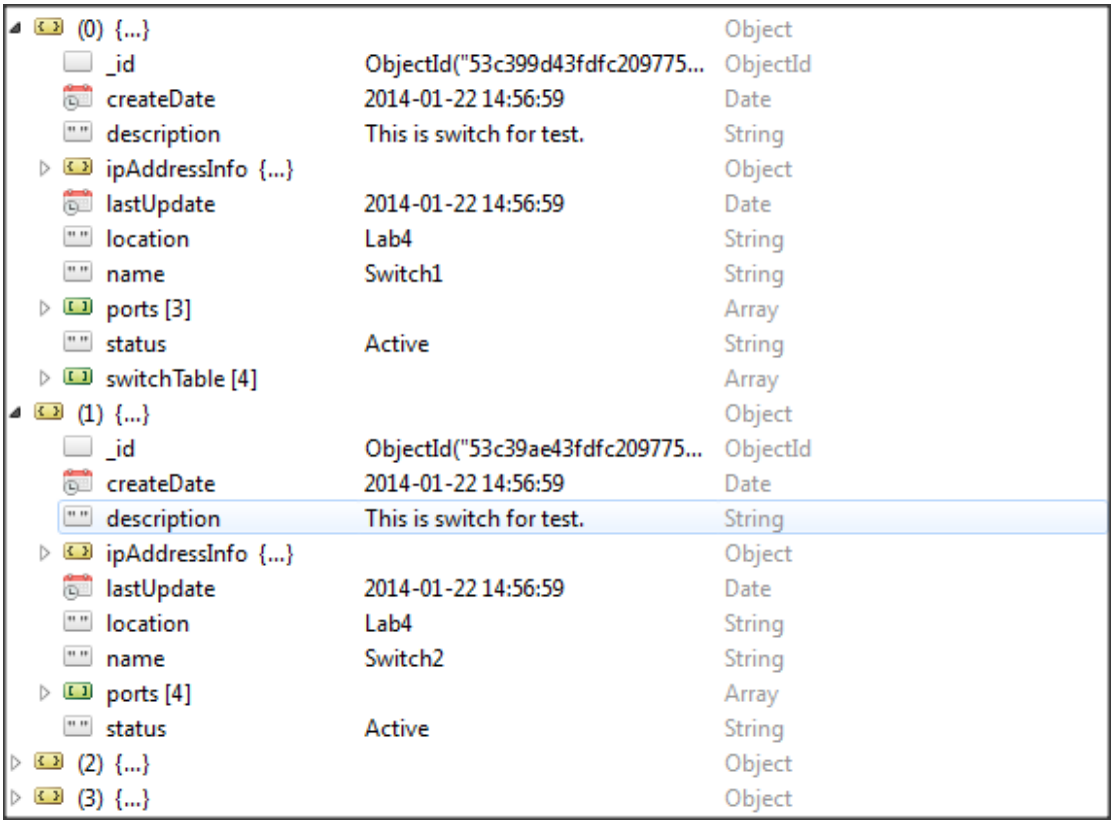
2.2.3. Using operators $unset

Format: {$unset : {field : 1}}

Description: Remove specified field in documents. Number 1 is usually used in $unset operator. But it can others value like: “”, ...

Example: Remove “description” field in all switches.

Before update:

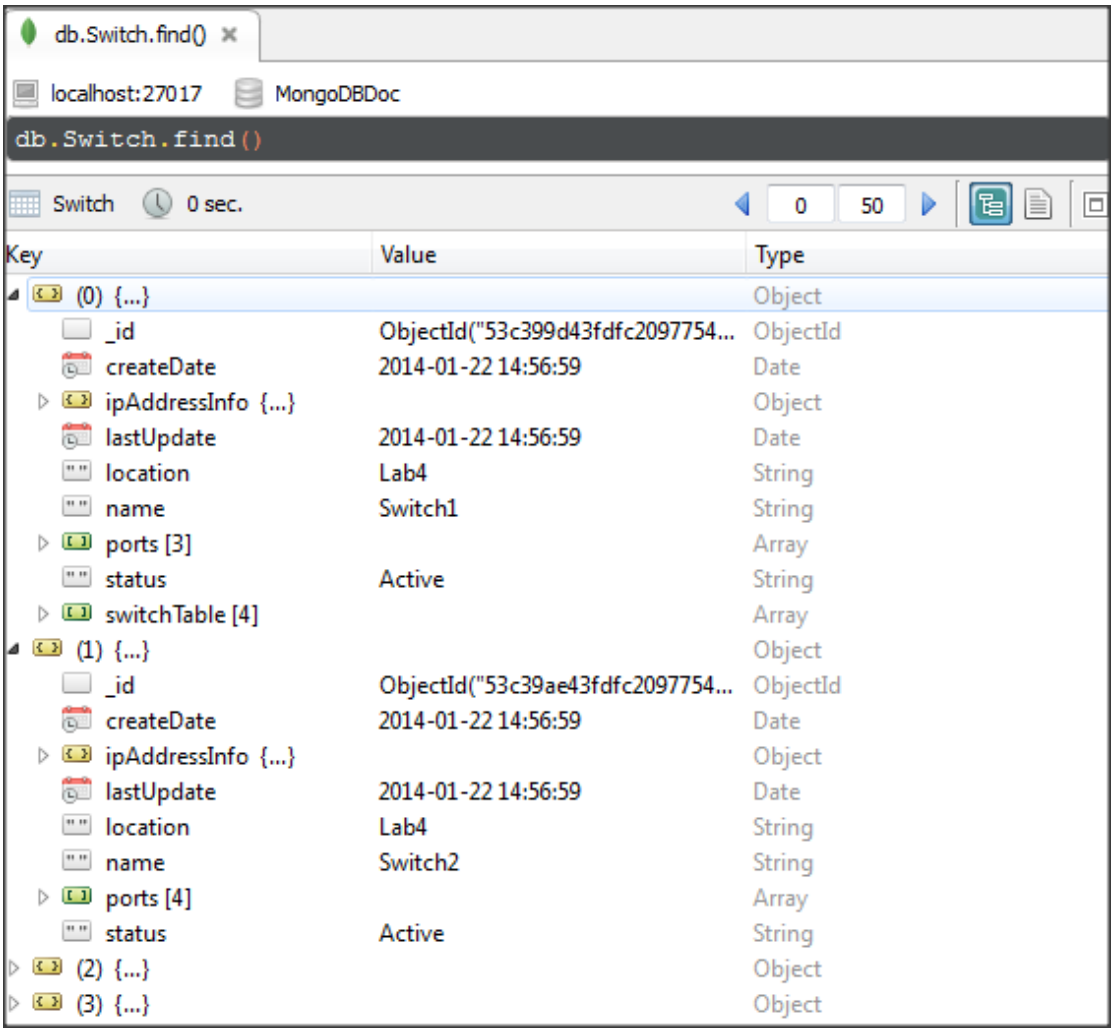


Picture 2.76. Operator $unset – before update with description field

Command to update:

```
db.Switch.update(
    {"_id" : {"$exists" : true}},
    {"$unset" : {"description" : ""}},
    {"multi" : true}
)
```

The result:



Picture 2.77. Operator $unset – result of update without description field

Using MongoTemplate:

```
try {
    // 1. Select all documents in database to update
    Query query = new Query();
    query.addCriteria(Criteria.where("name").is("Switch1"));
    // 2. Build update object
    Update update = new Update();
    update.unset("description");
    // 3. Update by using mongoTemplate
    mongoTemplate.updateMulti(query, update, Switch.class);
} catch (MongoException e) {
    System.out.println("Mongo Exc: " + e.getMessage());
}
```

Picture 2.78. Operator $unset using java

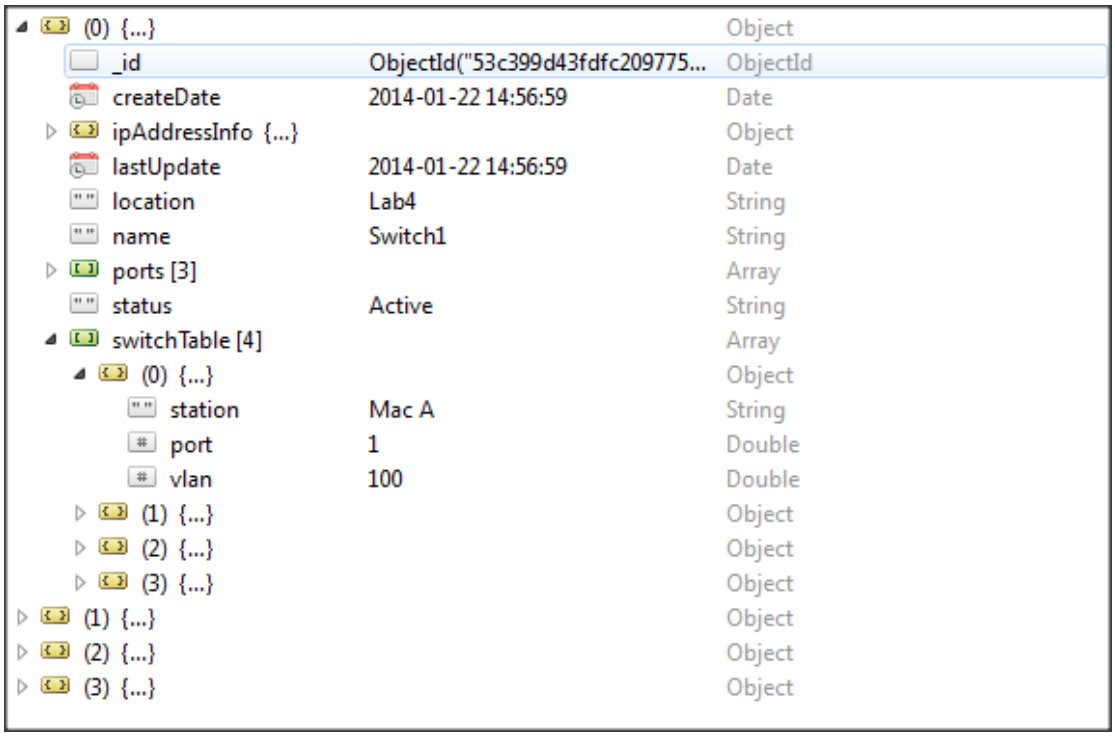
2.2.4. Using operators $inc

Format: {$inc : {field : amount}}

Description: Increment the value of field by specified amount.

Example: Increase first element 's vlan to 200 in switchTable array of Switch1.

Before update:

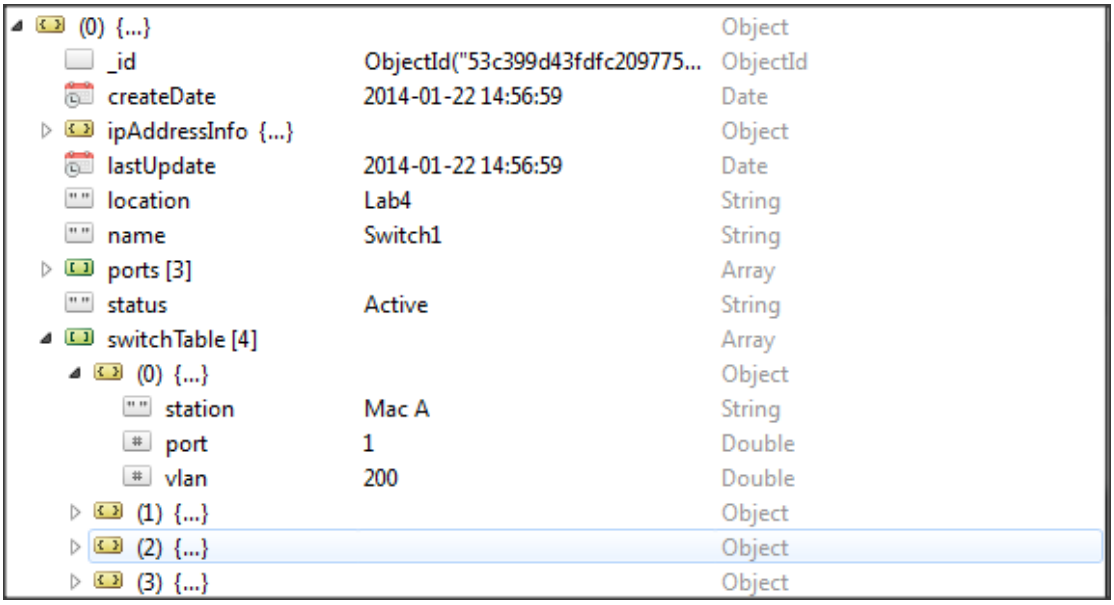


Picture 2.79. Operator $inc – before update

Command to update:

```
db.Switch.update(
    {"name" : "Switch1"},
    {"$inc" : {"switchTable.0.vlan" : 100}}
)
```

The result:



Picture 2.80. Operator $inc – result of update

Using MongoTemplate:

```
try {
    // 1. Select all documents in database to update
    Query query = new Query();
    query.addCriteria(Criteria.where("name").is("Switch1"));
    // 2. Build update object
    Update update = new Update();
    update.inc("switchTable.0.vlan", 100);
    // 3. Update by using mongoTemplate
    mongoTemplate.updateMulti(query, update, Switch.class);
} catch (MongoException e) {
    System.out.println("Mongo Exc: " + e.getMessage());
}
```

Picture 2.81. Operator $inc – result of update

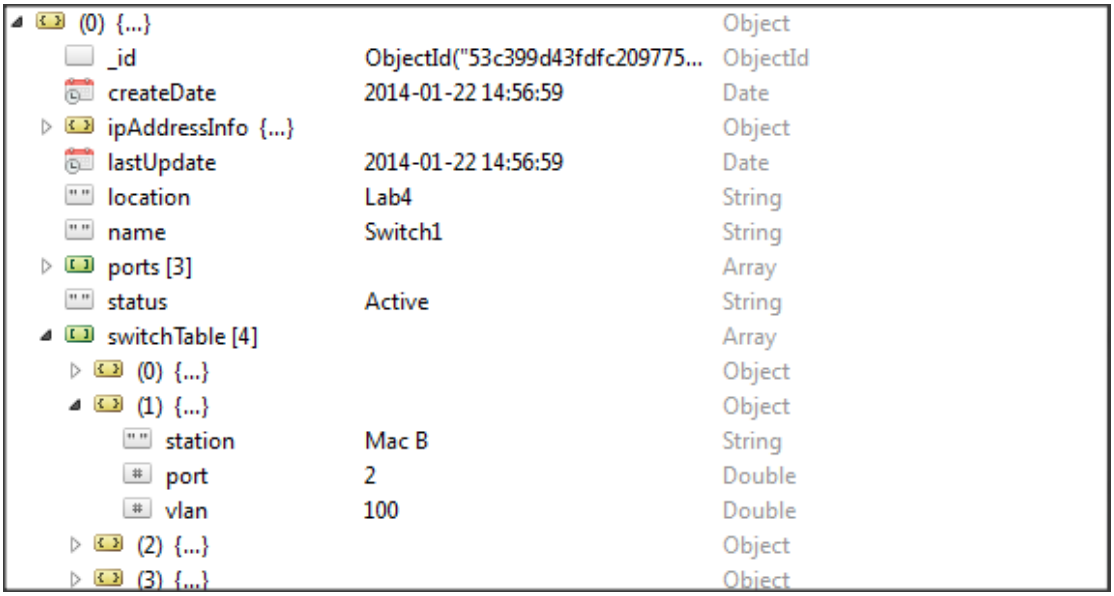
2.2.5. Using operators $mul

Format: {$mul : {field : amount}}

Description: Multiple the value of field by specified amount.

Example: Multiple second element 's vlan with 2 in switchTable array of Switch1.

Before update:

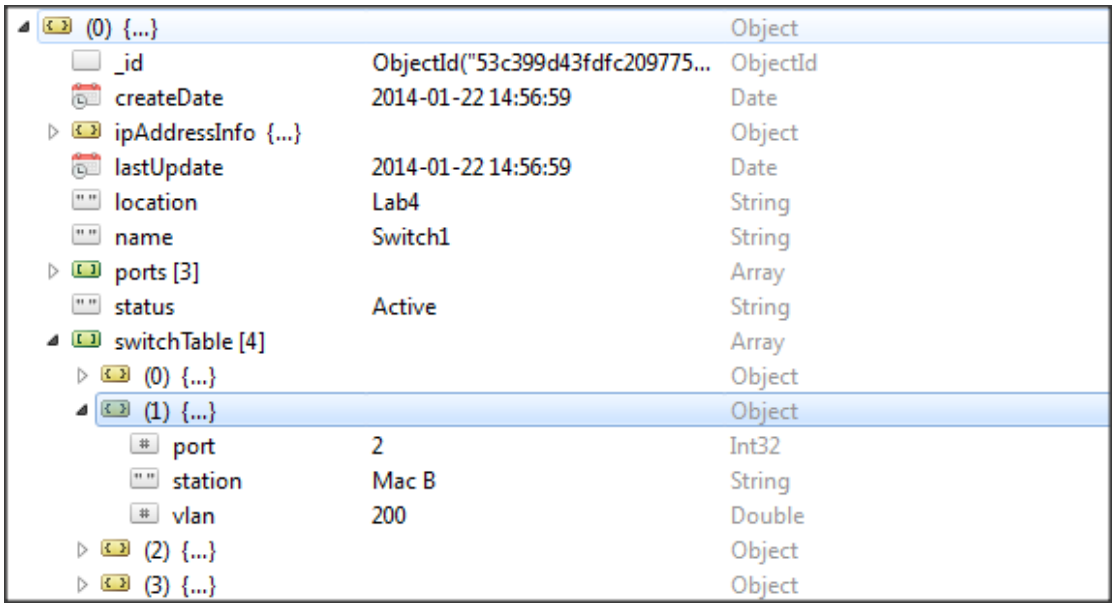


Picture 2.82. Operator $mul – before update

Command to update:

```
db.Switch.update(
    {"name" : "Switch1"},
    {"$mul" : {"switchTable.1.vlan" : 2}}
)
```

The result:



Picture 2.83. Operator $mul – result of update

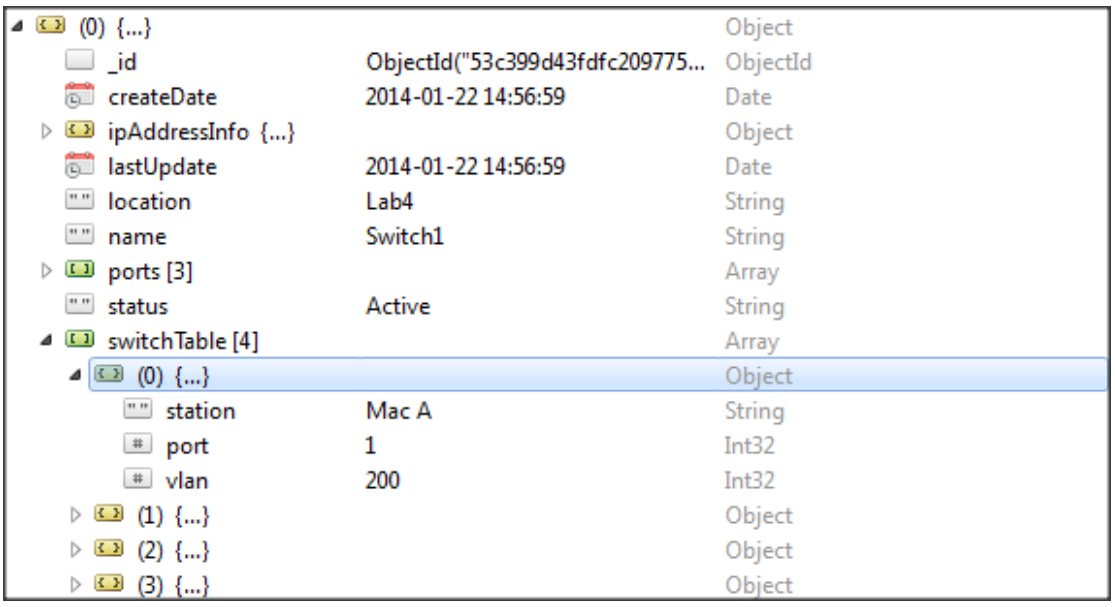
2.2.6. Using operators $min

Format: {$min : {field : new-value}}

Description: Only update field if new-value is less than old value of field.

Example: Change first element 's vlan from 200 to 100 in switchTable array of Switch1.

Before update:

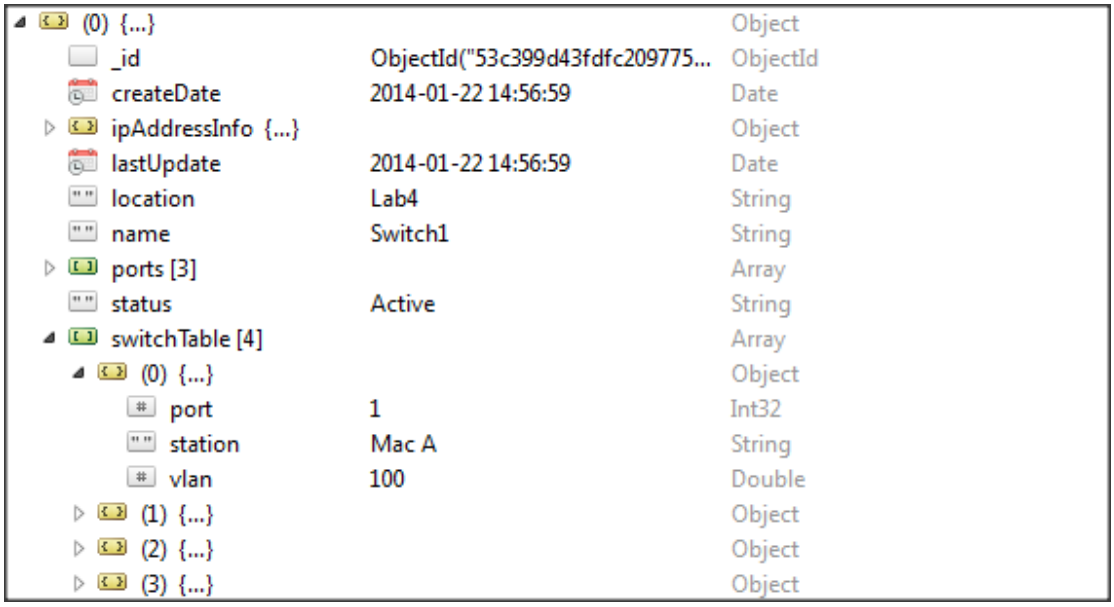


Picture 2.84. Operator $min – before update

Command to update:

```
db.Switch.update(
    {"name" : "Switch1"},
    {"$min" : {"switchTable.0.vlan" : 100}}
)
```

The result:



Picture 2.85. Operator $min – result of update

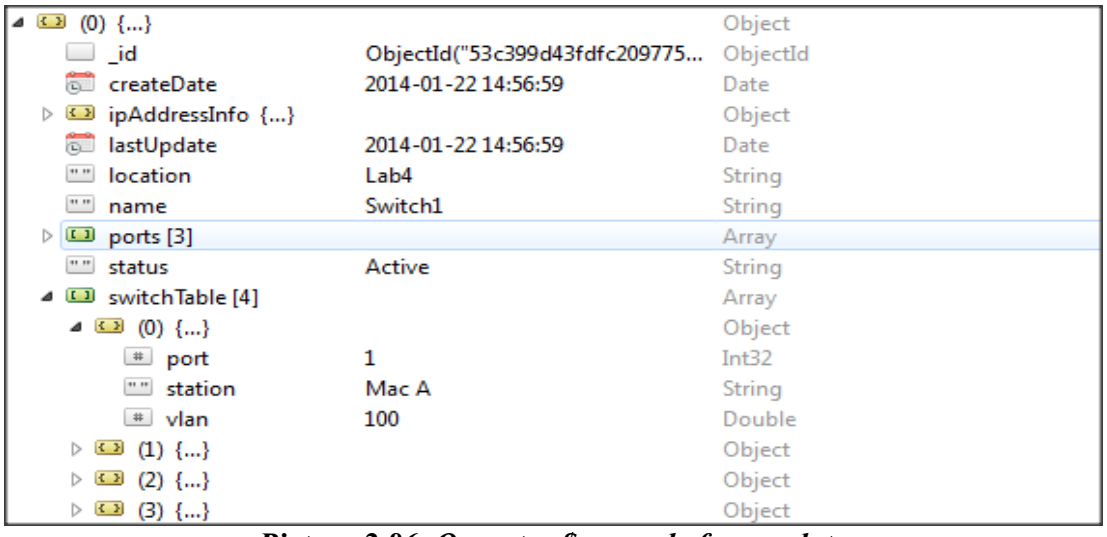
2.2.7. Using operators $max

Format: {$max : {field : new-value}}

Description: Only update field if new-value is great than old value of field.

Example: Change first element 's vlan from 100 to 200 in switchTable array of Switch1.

Before update:

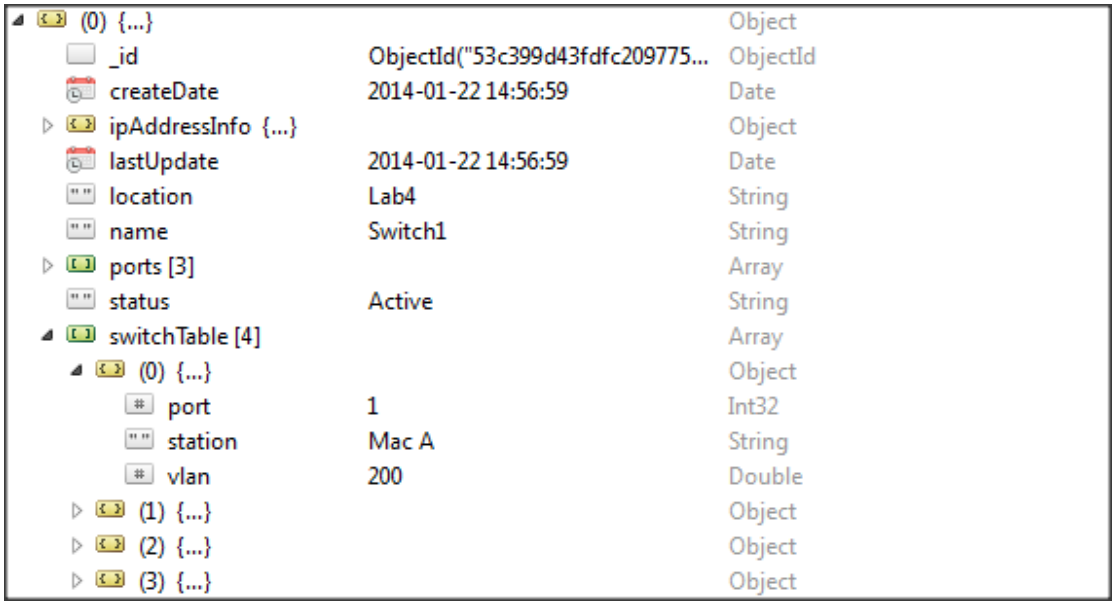


Picture 2.86. Operator $max – before update

Command to update:

```
db.Switch.update(
    {"name" : "Switch1"},
    {"$max" : {"switchTable.0.vlan" : 200}}
)
```

The result:



Picture 2.87. Operator $max – result of update

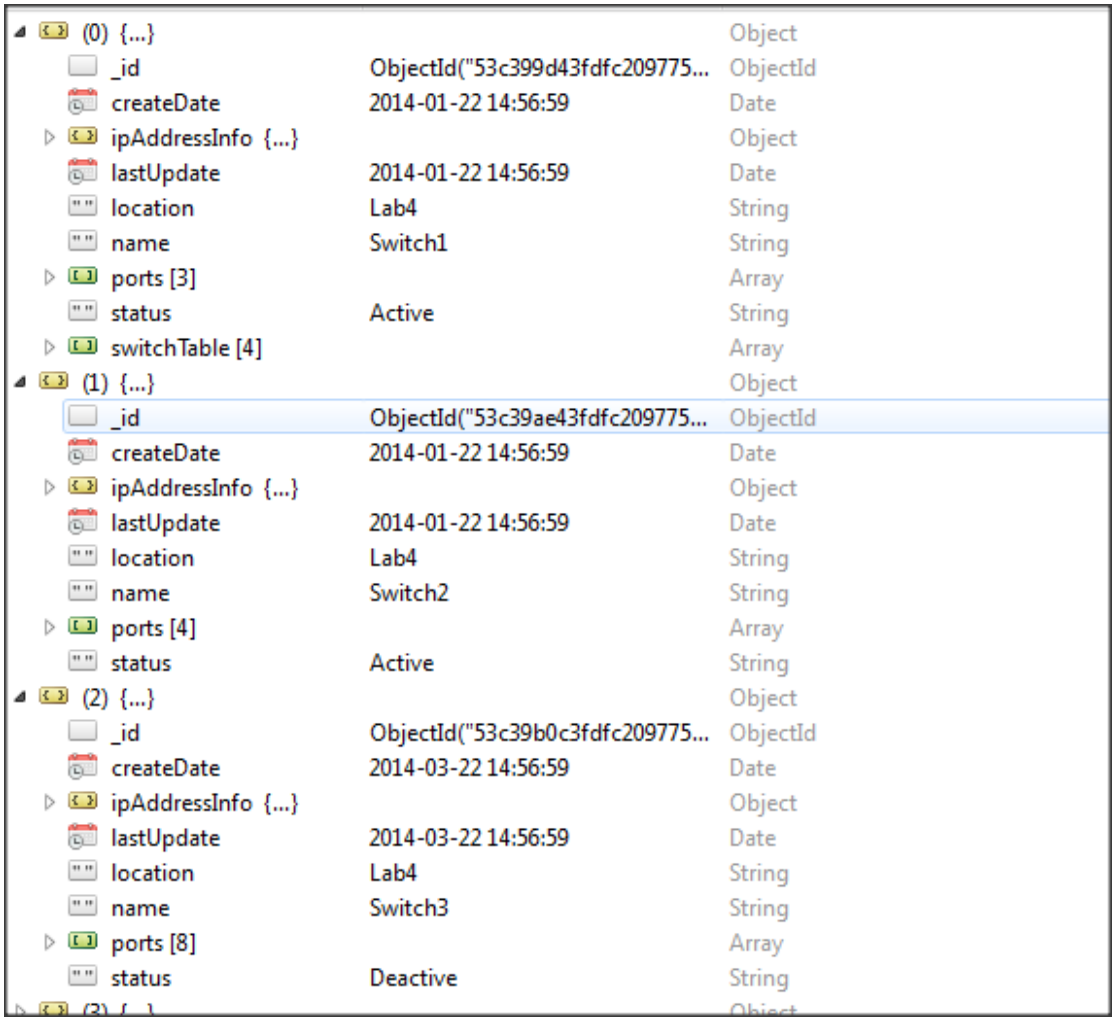
2.2.8. Using operators $currentDate

Format: {$currentDate : {field : true or $type}}

Description: Set value of field is current date. Default type is “date” but we can set $type by lowercase “timestamp” or “date”.

Example: Set “lastUpdate” field of all switches are currentDate.

Before update:

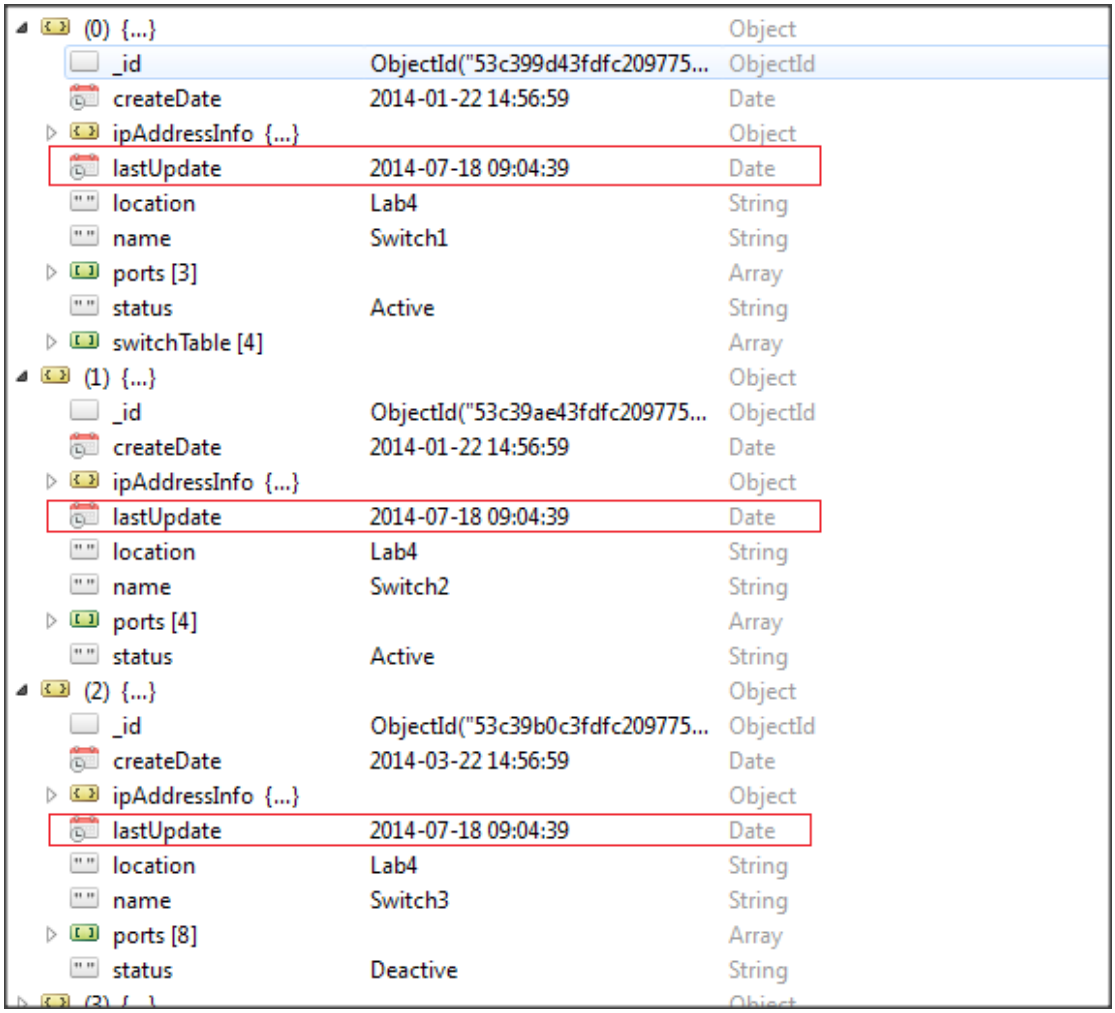


Picture 2.88. Operator $currentDate – before update

Command to update:

```
db.Switch.update(
    {"_id" : {"$exists" : true}},
    {"$currentDate" : {"lastUpdate" : true}},
    {"multi" : true}
)
```

The result:



Picture 2.89. Operator $currentDate – result of update

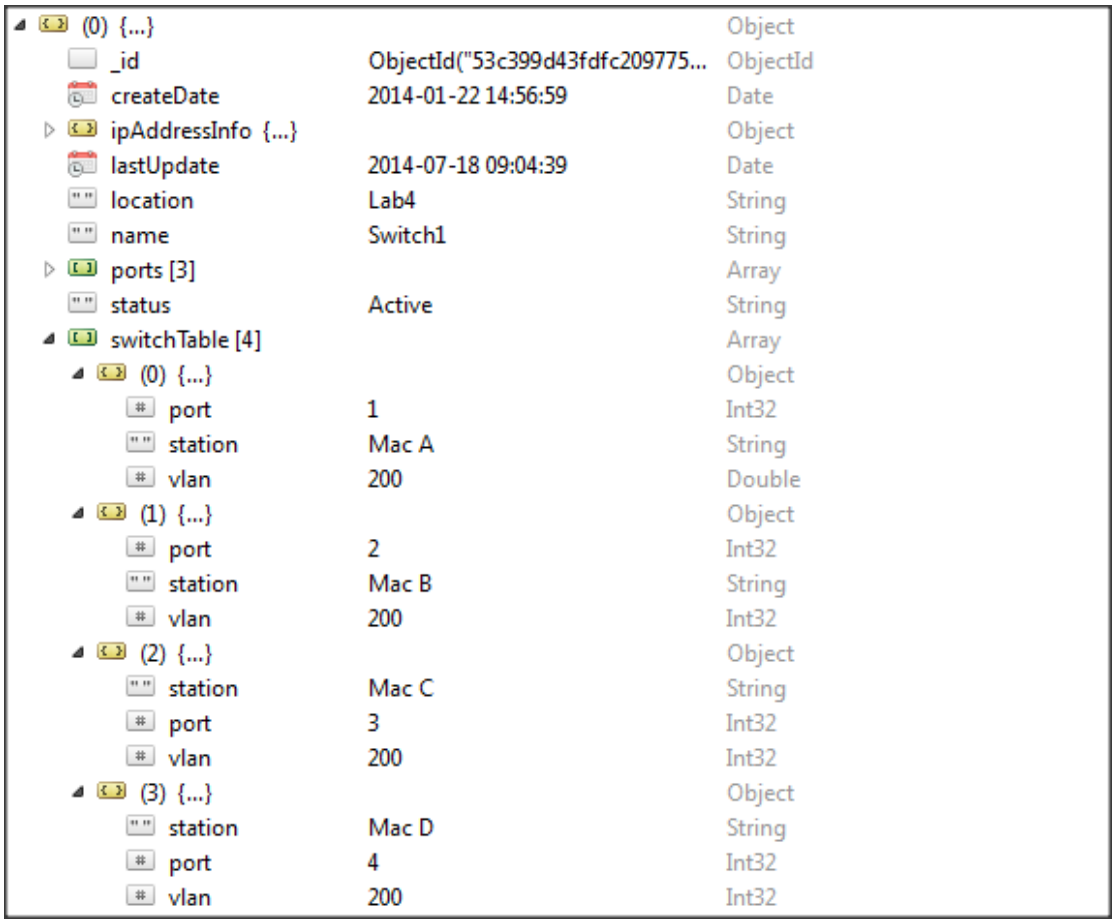
2.2.9. Using operators $pop

Format: {$pop : {field : boolean-value}}

Description: Remove first element of an array if boolean-value = -1 or remove last element of an array if boolean-value = 1.

Example: Remove first element in switchTable array of Switch1.

Before update:

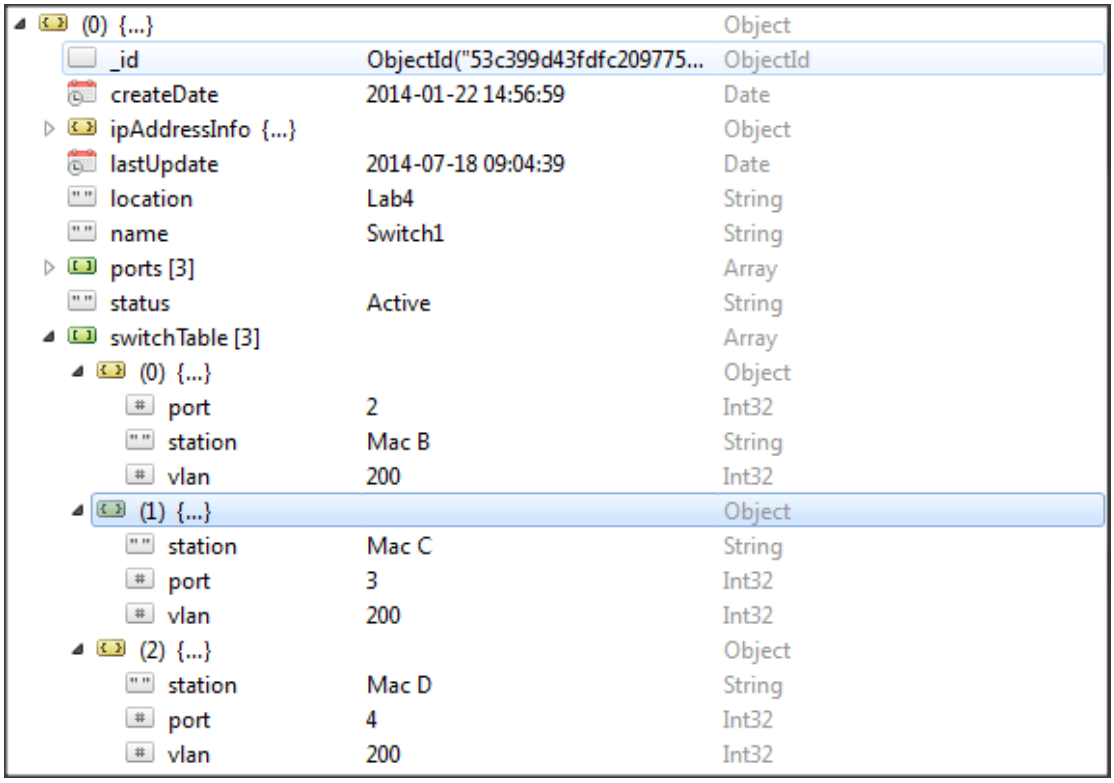


Picture 2.90. Operator $pop – before update

Command to update:

```
db.Switch.update(
    {"name" : "Switch1"},
    {"$pop" : {"switchTable" : -1}}
)
```

The result:



Picture 2.91. Operator $pop – result of update

Using MongoTemplate:

```
try {
    // 1. Select all documents in database to update
    Query query = new Query();
    query.addCriteria(Criteria.where("name").is("Switch1"));
    // 2. Build update object
    Update update = new Update();
    update.pop("switchTable", Position.FIRST);
    // 3. Update by using mongoTemplate
    mongoTemplate.updateFirst(query, update, Switch.class);
} catch (MongoException e) {
    System.out.println("Mongo Exc: " + e.getMessage());
}
```

Picture 2.92. Operator $pop using java

2.2.10.Using operators $pullAll

Format: {$pullAll : {field : value array}}

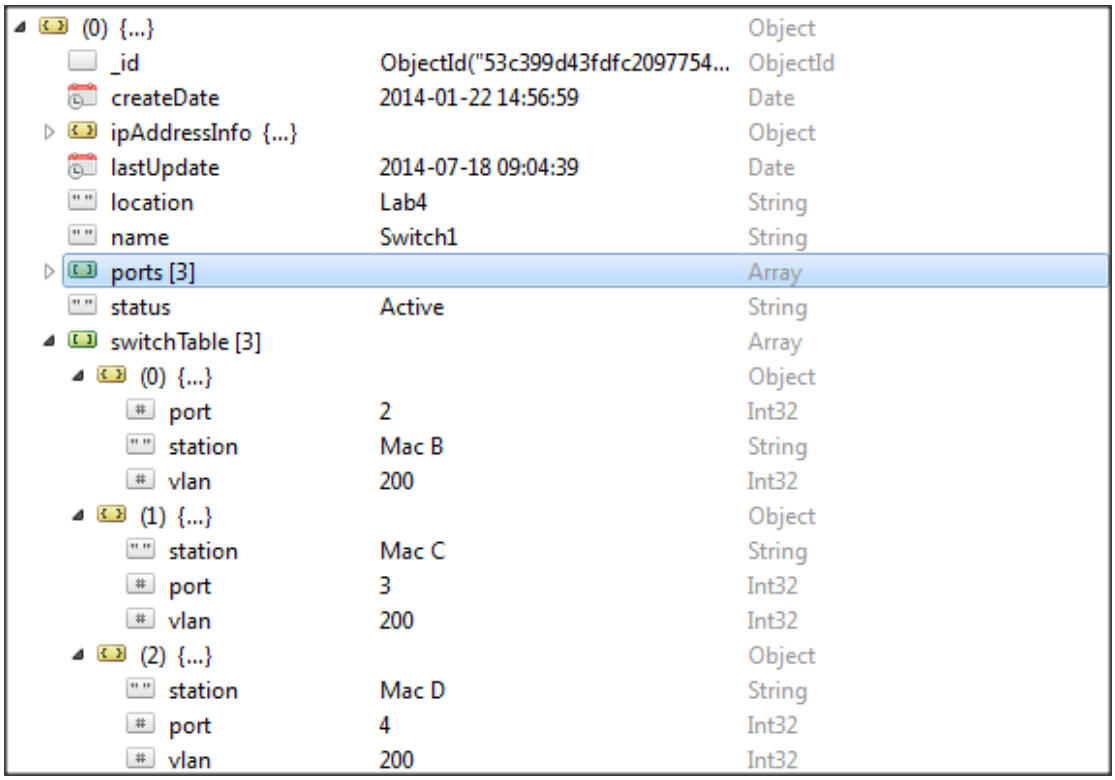
Description: Removes all matching values from an array.

Example: Remove 2 following elements in switchTable array of Switch1:

[{port : 2, station : “Mac B”, vlan : 200} ,

{port : 3, station : “Mac C”, vlan : 200}]

Before update:

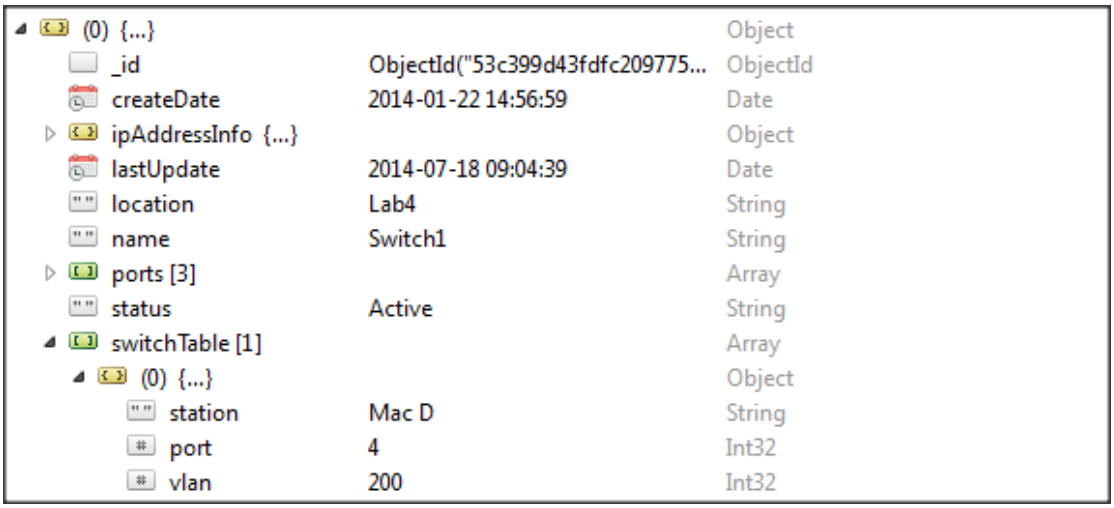


Picture 2.93. Operator $pullAll – before update

Command to update:

```
db.Switch.update(
    {"name" : "Switch1"},
    {"$pullAll" : {"switchTable" : [
            {
            "port" : 2,
            "station" : "Mac B",
            "vlan" : 200
            },
            {
            "station" : "Mac C",
            "port" : 3,
            "vlan" : 200
            }
        ]}}
)
```

Result:



Picture 2.94. Operator $pullAll – result of update

Using MongoTemplate:

```
try {
    // 1. Select all documents in database to update
    Query query = new Query();
    query.addCriteria(Criteria.where("name").is("Switch1"));
    // 2. Build update object
    BasicDBObject element2 = new BasicDBObject();
    element2.put("station", "Mac B");
    element2.put("port", "2");
    element2.put("vlan", "100");
    BasicDBObject element3 = new BasicDBObject();
    element3.put("station", "Mac C");
    element3.put("port", "3");
    element3.put("vlan", "200");
    ArrayList<BasicDBObject> value = new ArrayList<BasicDBObject>()
    value.add(element2);
    value.add(element3);
    Update update = new Update();
    update.pullAll("switchTable", value.toArray());
    // 3. Update by using mongoTemplate
    mongoTemplate.updateFirst(query, update, Switch.class);
} catch (MongoException e) {
    System.out.println("Mongo Exc: " + e.getMessage());
}
```

Picture 2.95. Operator $pullAll using java

2.2.11. Using operators $push

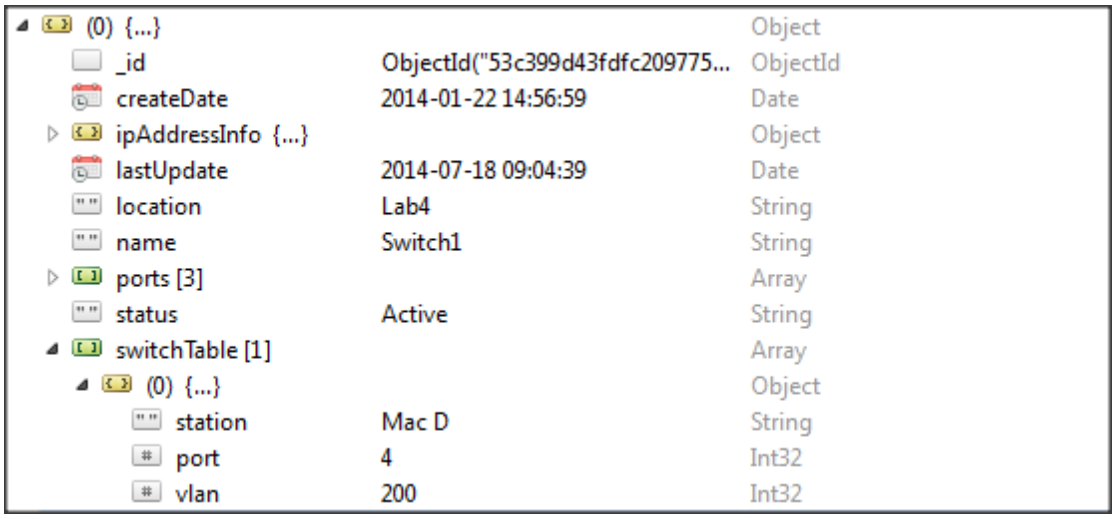
Format: {$push : {field : value element}}

Description: Add an element into an array.

Example: Add following element to switchTable field of Switch1:

{port : 2, station : “Mac B”, vlan : 200}

Before update:

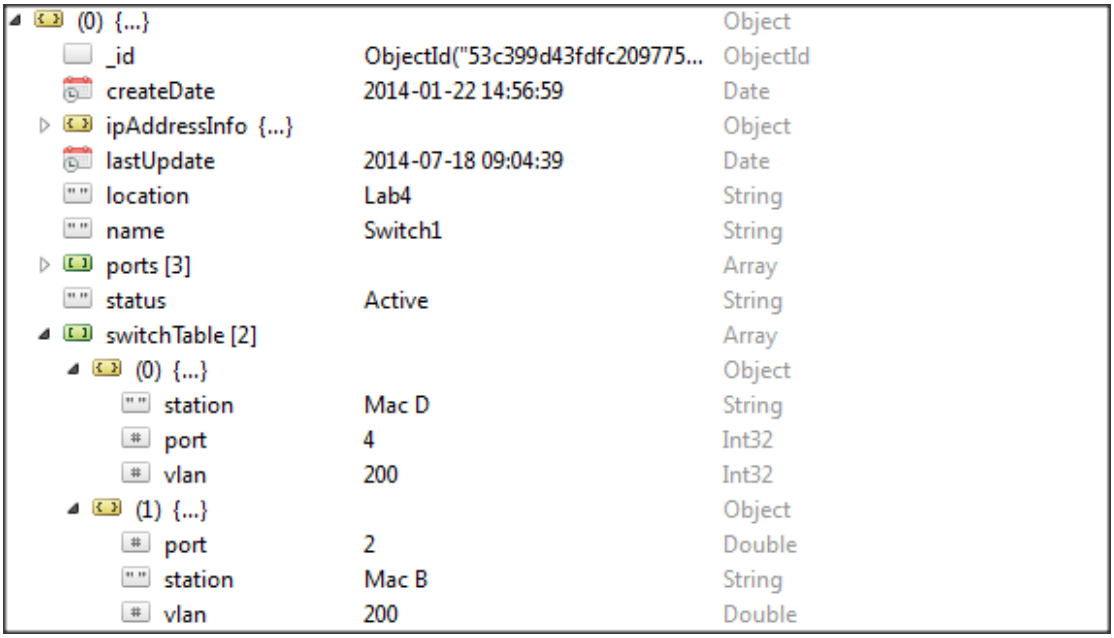


Picture 2.96. Operator $push – before update

Command to update:

```
db.Switch.update(
    {"name" : "Switch1"},
    {"$push" : {"switchTable" : {
                    "port" : 2,
                    "station" : "Mac B",
                    "vlan" : 200
        }}}
)
```

The result:



Picture 2.97. Operator $push – result of update

Using MongoTemplate:

```
try {
    // 1. Select all documents in database to update
    Query query = new Query();
    query.addCriteria(Criteria.where("name").is("Switch1"));
    // 2. Build update object
    BasicDBObject value = new BasicDBObject();
    value.put("station", "Mac B");
    value.put("port", "2");
    value.put("vlan", "100");
    Update update = new Update();
    update.push("switchTable", value);
    // 3. Update by using mongoTemplate
    mongoTemplate.updateFirst(query, update, Switch.class);
} catch (MongoException e) {
    System.out.println("Mongo Exc: " + e.getMessage());
}
```

Picture 2.98. Operator $push using java

a. Modifier operator $each

Format: {$push : {field : {$each : array}}}

Description: Modified the $push operator to allow add multiple elements into field which is an array.

b. Modifier operator $slice

Format: {$push : {field : {$each : array, $slice : number}}}

Description: It must appear with the $each modifier. To limit number of elements in an array. If <number> is nagative, array contain only last <number> elements. If <number> is positive, array contain only first <number> elements.

c. Modifier operator $sort

Format: {$push : {field : {$each : array, $sort : {field : Order}}}}

Description: It must appear with the $each modifier. Reorder documents in array.

d. Modifier operator $position

Format: {$push : {field : {$each : array, $position : number}}}

Description: It must appear with the $each modifier. Specify the position in array to add elements. Default $push operation will add elements in last of array.

e. Example for using modifiers 1

Add following elements to switchTable field of Switch1:

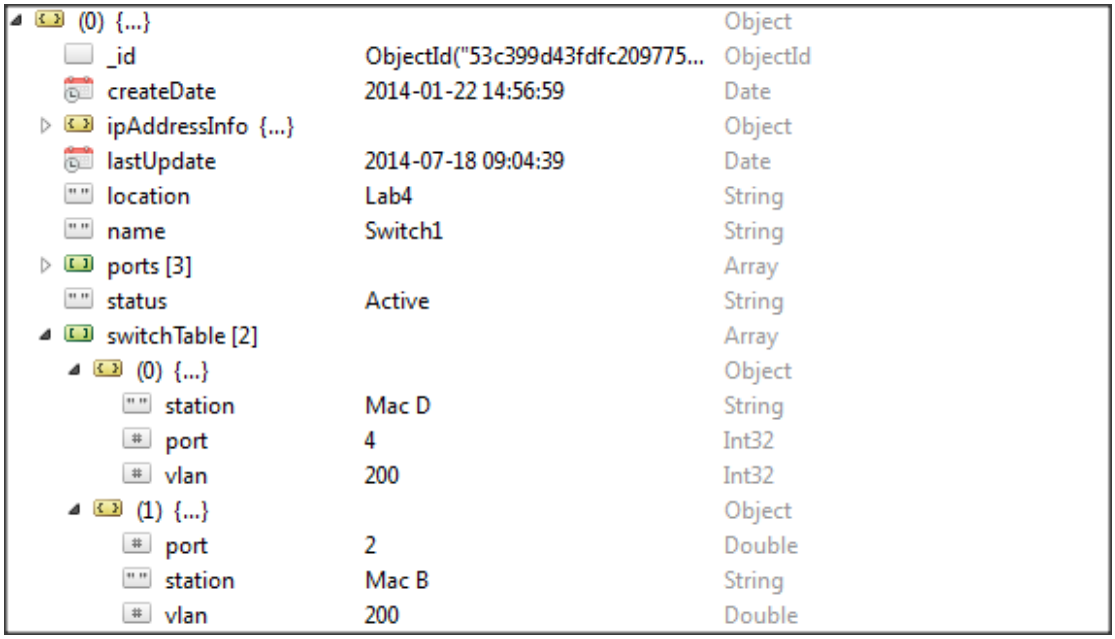
{port : 1, station : “Mac A”, vlan : 200},

{port : 2, station : “Mac B”, vlan : 200},

{port : 3, station : “Mac C”, vlan : 200}

And order documents by ascending order of port field.

Before update:

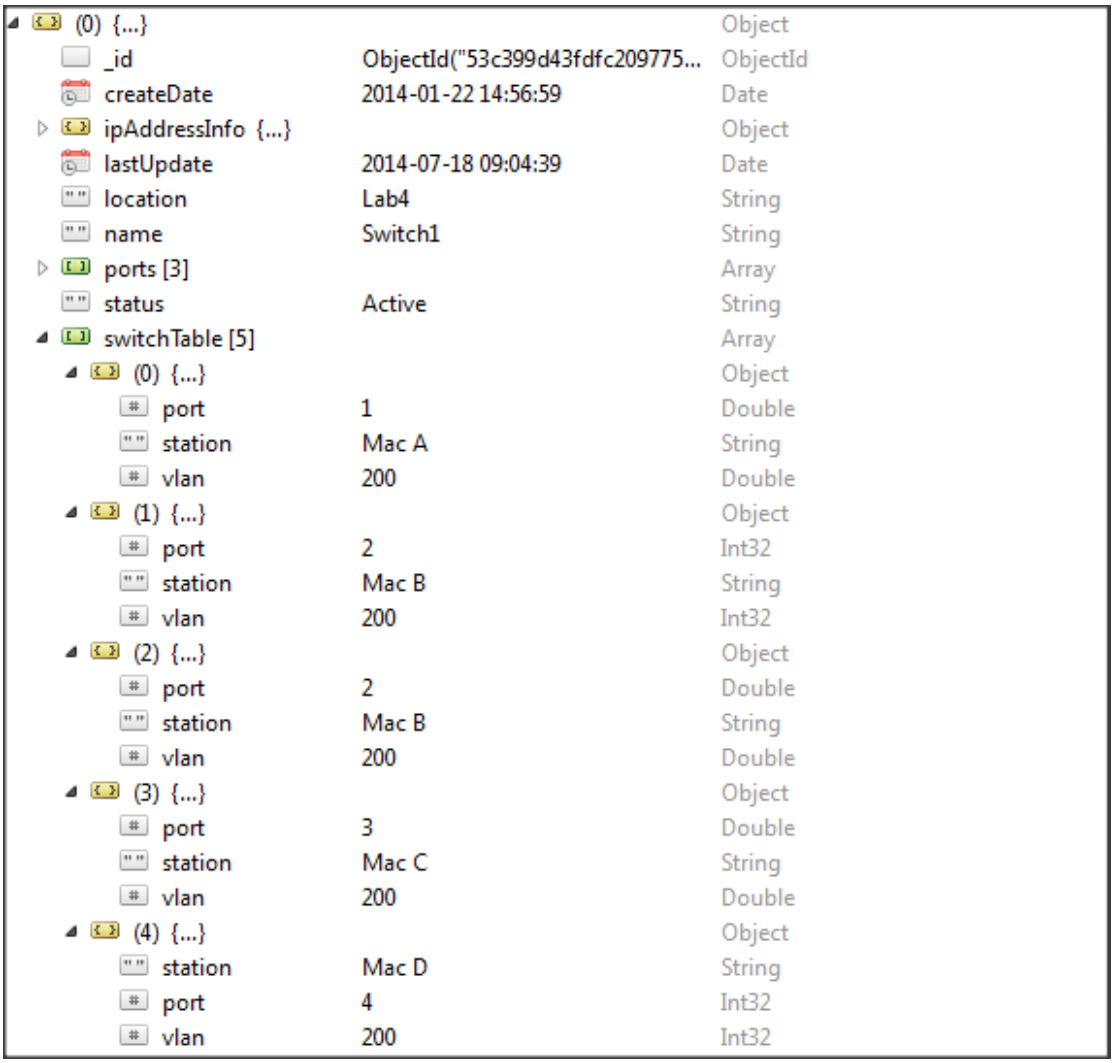


Picture 2.99. Operator $push with modifiers operator – example 1

Command to update:

```
db.Switch.update(
    {"name" : "Switch1"},
    {"$push" : {"switchTable" : {"$each" : [
                    {"port" : 1, "station" : "Mac A", "vlan" : 200},
                    {"port" : 2, "station" : "Mac B", "vlan" : 200},
                    {"port" : 3, "station" : "Mac C", "vlan" : 200}],
                              "$sort" : {"port" : 1}

        }}}
)
```

The result:



Picture 2.100. Operator $push with modifiers – result of example 1

f. Example for using modifiers 2

Add following elements to switchTable field of Switch1:

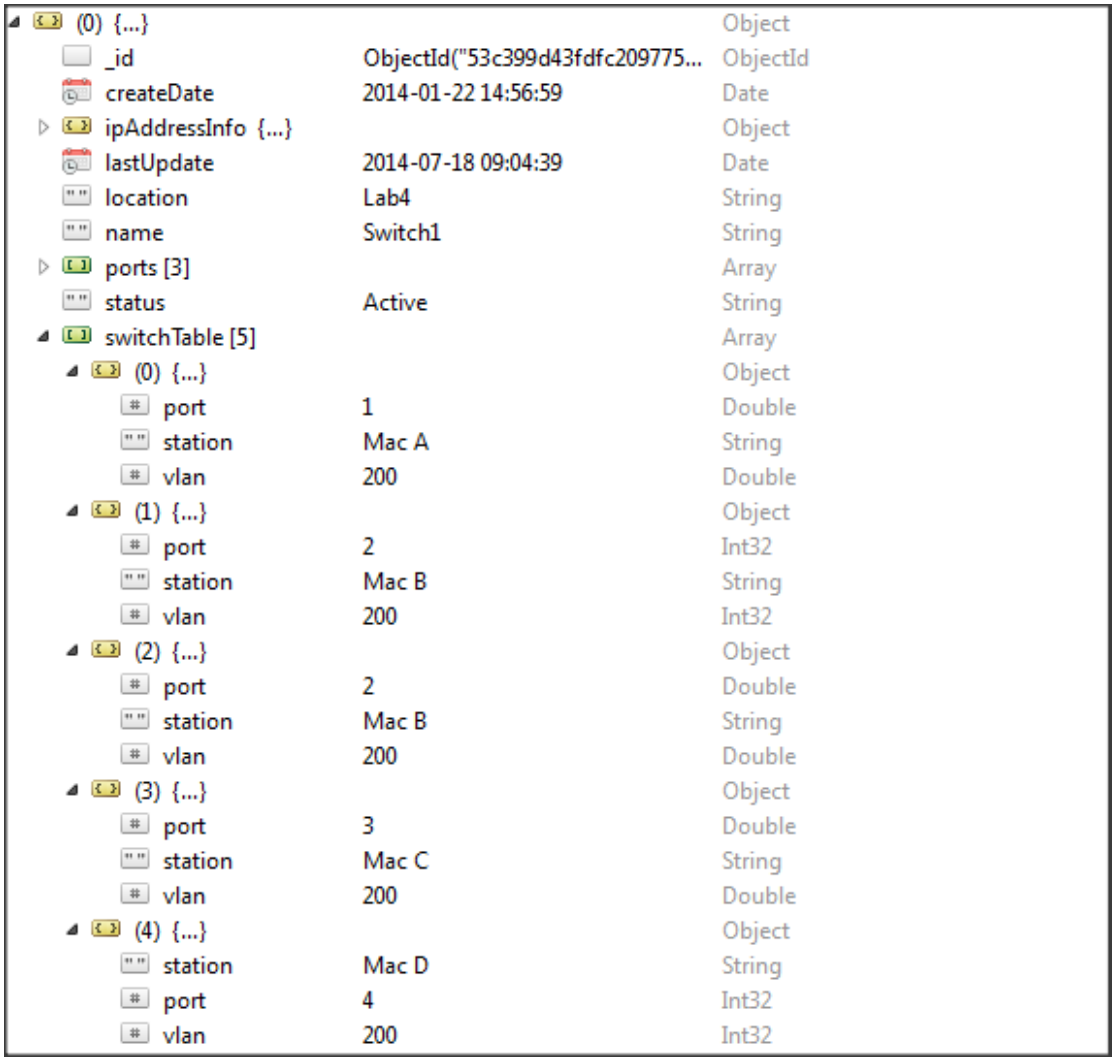
{port : 1, station : “Mac A”, vlan : 200},

{port : 2, station : “Mac B”, vlan : 200},

{port : 3, station : “Mac C”, vlan : 200}

And limit 5 newest elements in array.

Before update:

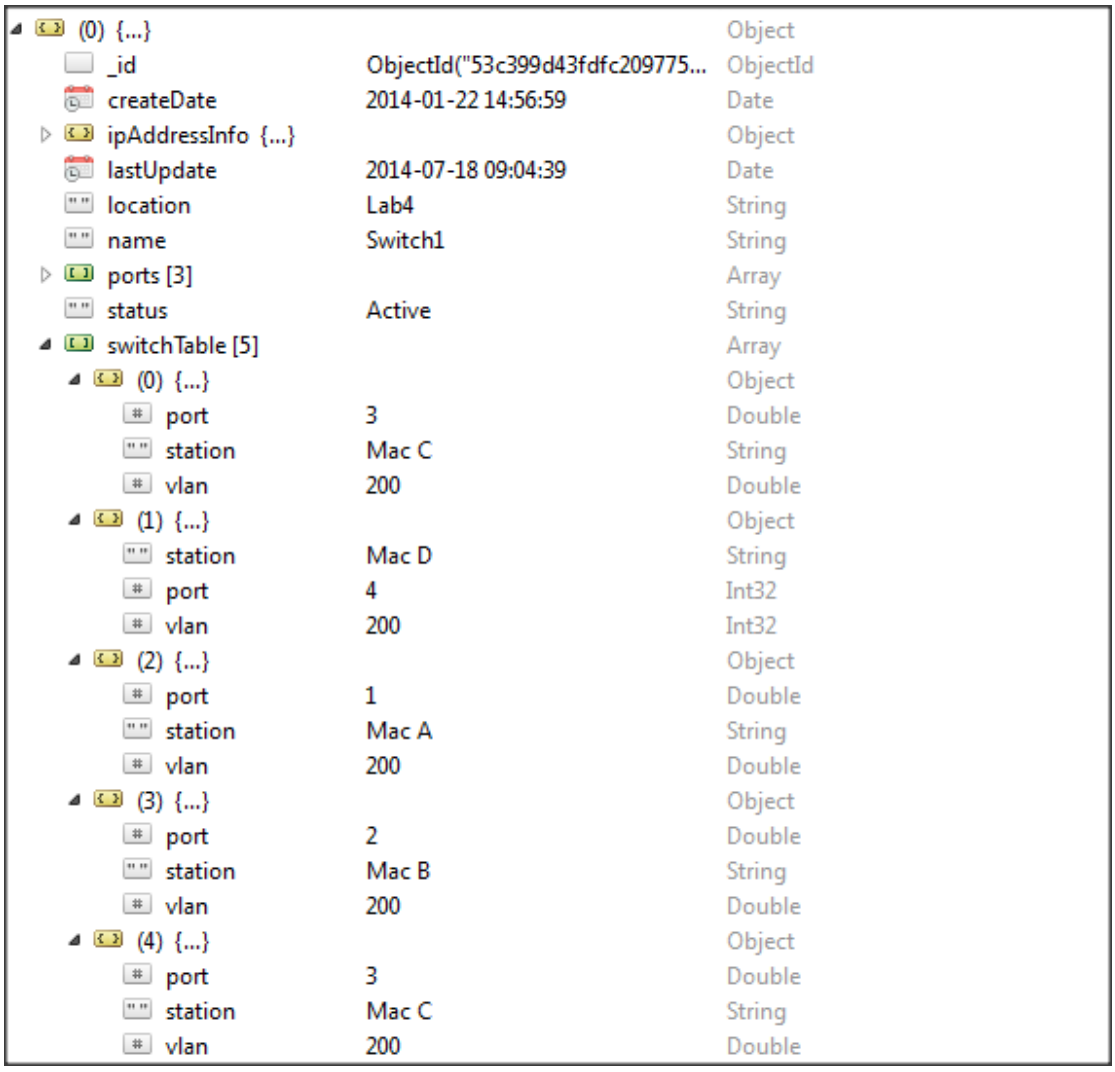


Picture 2.42. Operator $push with modifiers operator – example 2

Command to update:

```
db.Switch.update(
    {"name" : "Switch1"},
    {"$push" : {"switchTable" : {"$each" : [
                    {"port" : 1, "station" : "Mac A", "vlan" : 200},
                    {"port" : 2, "station" : "Mac B", "vlan" : 200},
                    {"port" : 3, "station" : "Mac C", "vlan" : 200}],
                                "$slice" : -5

        }}}
)
```

The result:



Picture 2.101. Operator $push with modifiers – result of example 2

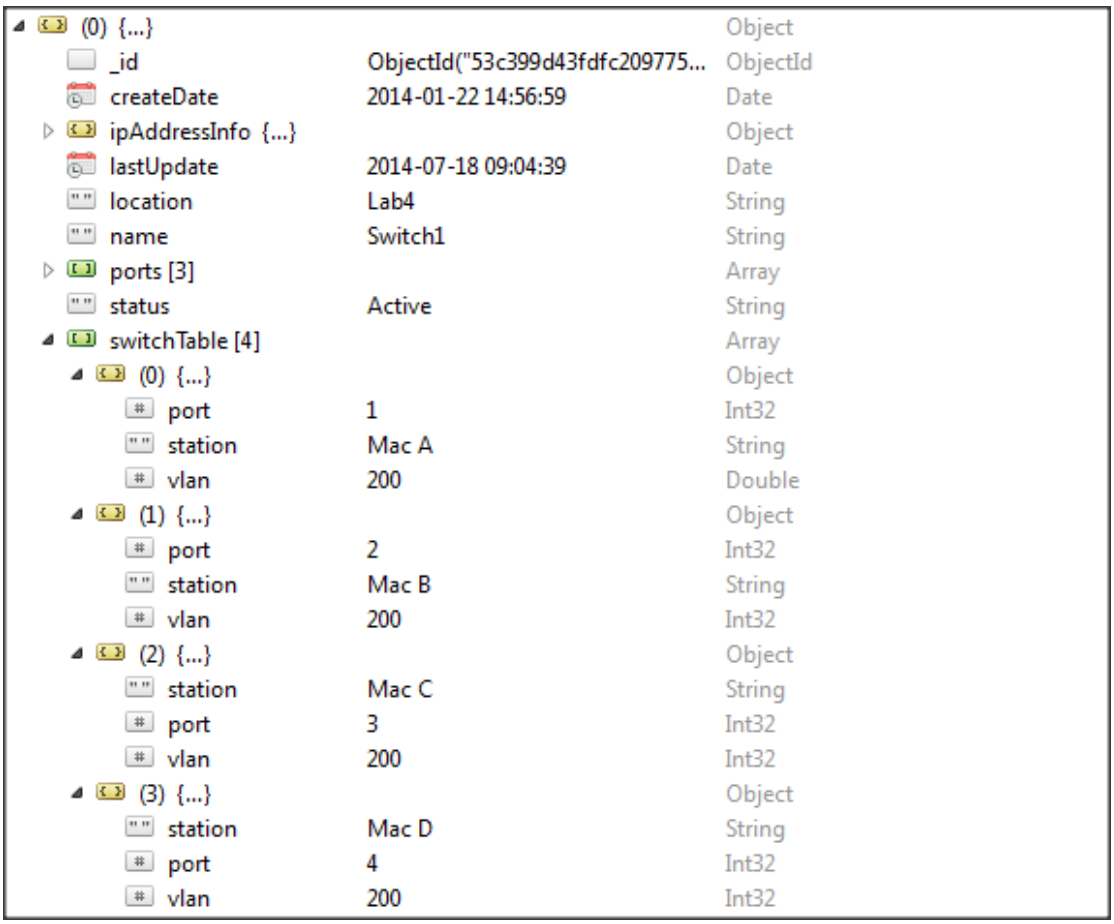
2.2.12.Using operators $pull

Format: {$pull : {field : operation-expression}}

Description: Remove elements in array that match operation-expression.

Example: Remove elements in switchTable array of Switch1 that have vlan is equal 200.

Before update:

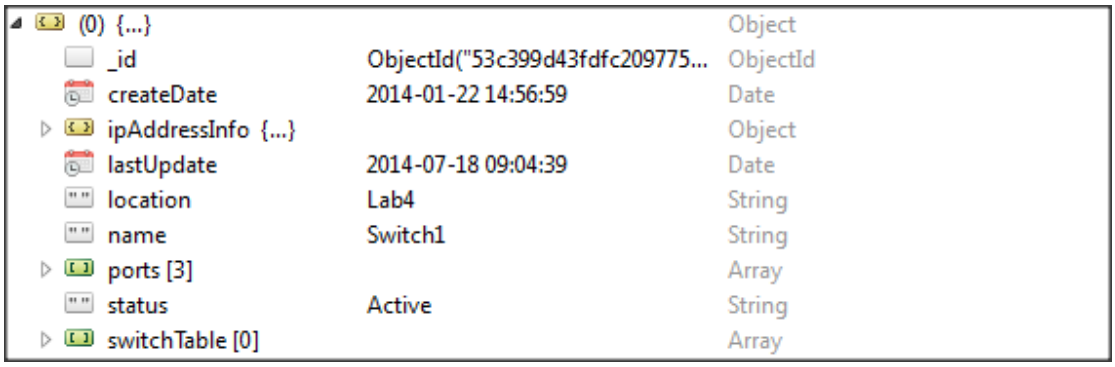


Picture 2.102. Operator $pull – before update

Command to update:

```
db.Switch.update(
    {"name" : "Switch1"},
    {"$pull" : {"switchTable" : {"vlan" : 200}}}
)
```

The result:



Picture 2.103. Operator $pull – result of update

Using MongoTemplate:

```
try {
    // 1. Select all documents in database to update
    Query query = new Query();
    query.addCriteria(Criteria.where("name").is("Switch1"));
    // 2. Build update object
    BasicDBObject value = new BasicDBObject();
    value.append("vlan", 200);
    Update update = new Update();
    update.pull("switchTable", value);
    // 3. Update by using mongoTemplate
    mongoTemplate.updateFirst(query, update, Switch.class);
} catch (MongoException e) {
    System.out.println("Mongo Exc: " + e.getMessage());
}
```

Picture 2.104. Operator $pull using java

2.2.13.Using operators $addToSet

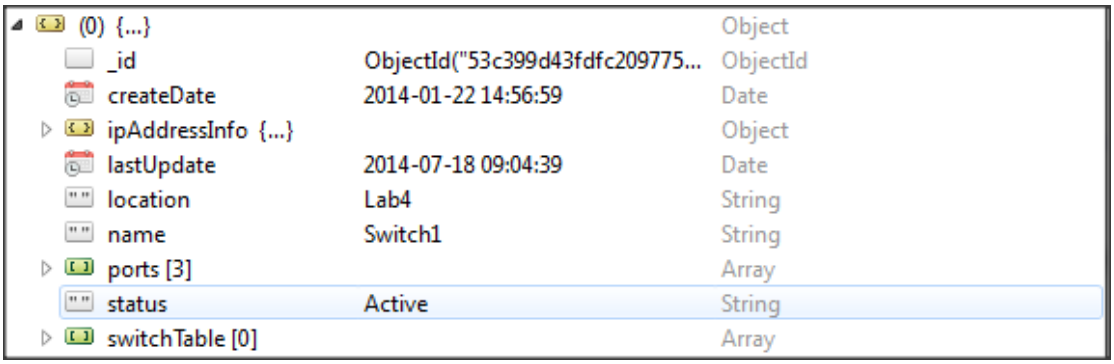
Format: {$addToSet : {field : value element}}

Description: Add an element into an array. It like $push operator but it only use for adding element which is not exist in array.

Example: Add following elements to switchTable field of Switch1:

{port : 2, station : “Mac B”, vlan : 200},

Before update:

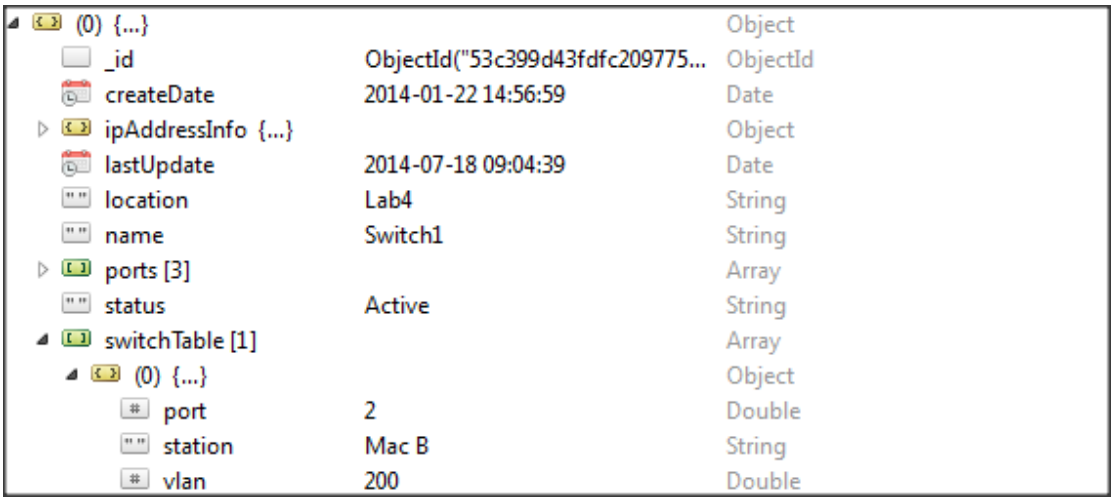


Picture 2.105. Operator $addToSet – before update

Command to update:

```
db.Switch.update(
    {"name" : "Switch1"},
    {"$addToSet" : {"switchTable" : {
                "port" : 2,
                "station" : "Mac B",
                "vlan" : 200
        }}}
)
```

The result:



Picture 2.106. Operator $addToSet – result of update

Using MongoTemplate:

```
try {
    // 1. Select all documents in database to update
    Query query = new Query();
    query.addCriteria(Criteria.where("name").is("Switch1"));
    // 2. Build update object
    BasicDBObject value = new BasicDBObject();
    value.put("station", "Mac B");
    value.put("port", "2");
    value.put("vlan", "100");
    Update update = new Update();
    update.addToSet("switchTable", value);
    // 3. Update by using mongoTemplate
    mongoTemplate.updateFirst(query, update, Switch.class);
} catch (MongoException e) {
    System.out.println("Mongo Exc: " + e.getMessage());
}
```

Picture 2.107. Operator $addToSet using java

* Note: We can add multiple elements by using modifier $each. See modifier $each.

2.3 Remove operations

MongoDB use remove() method to remove documents.

Syntax:

db.**collectionName**.remove([{selection criteria}], single)

+ Selection criteria : is criteria to select document which is needed remove.

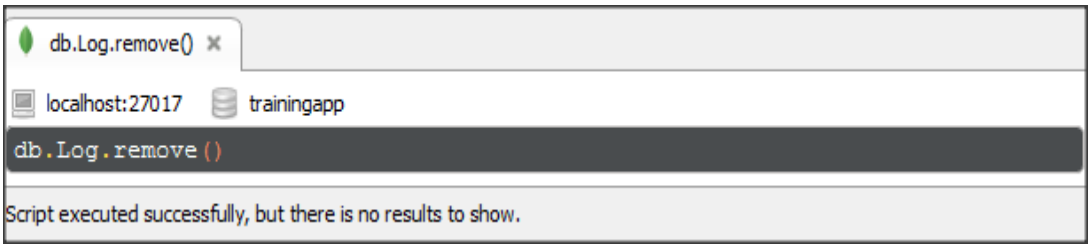
+ Single : only delete single document or delete all documents which pass criteria. Value is true or false. Default value is false.

2.3.1. Remove all documents

Criteria: no

Single: false

In Robomongo:



Picture 2.108. Remove all documents

Use MongoTemplate:

```
public void removeAllDocuments() {
    try {
        mongoTemplate.remove(new Query(), Log.class);
    } catch (MongoException e) {
        LoggerFactory.getLogger(getClass()).error("Remove error", e);
    }
}
```

Picture 2.109. Remove all documents using java

Use Repository:

```
public void removeAllDocuments() {
    try {
        logRepository.deleteAll();
    } catch (MongoException e) {
        LoggerFactory.getLogger(getClass()).error("Remove error", e);
    }
}
```

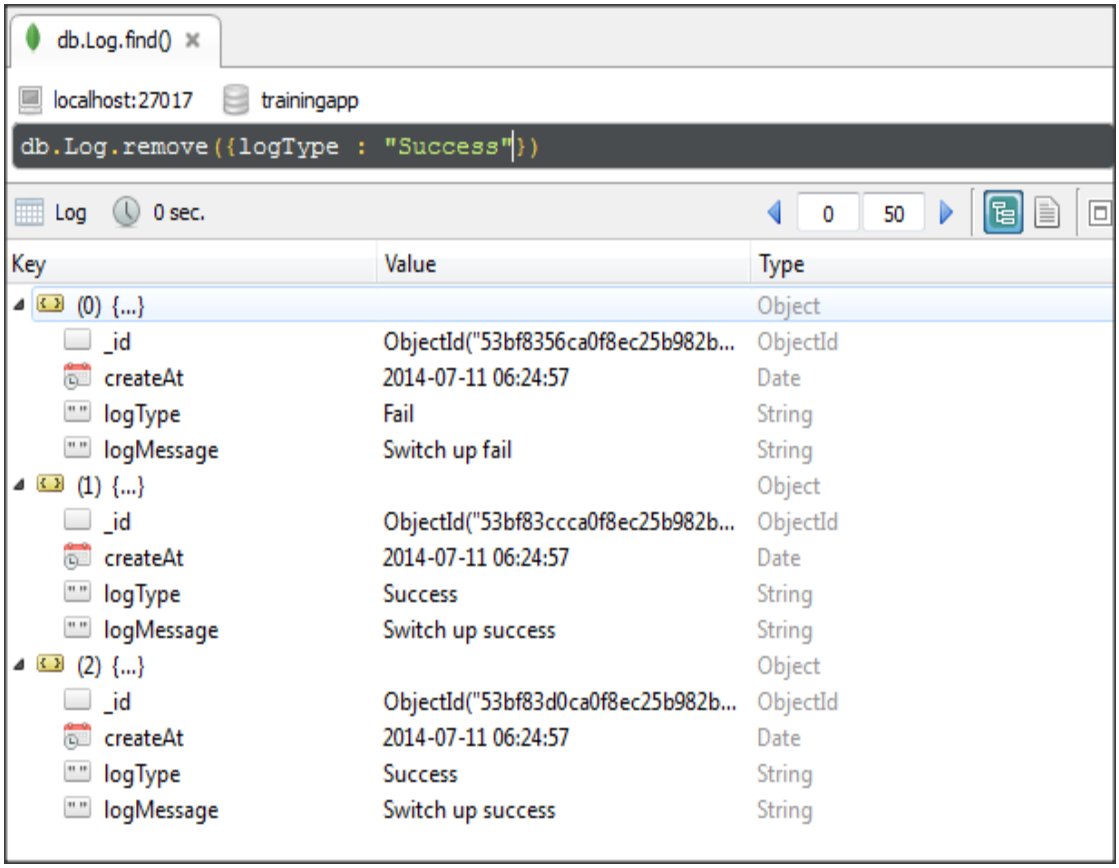
Picture 2.110. Remove all documents using repository

2.3.2. Remove documents by criteria

Criteria: logType equal “Success”.

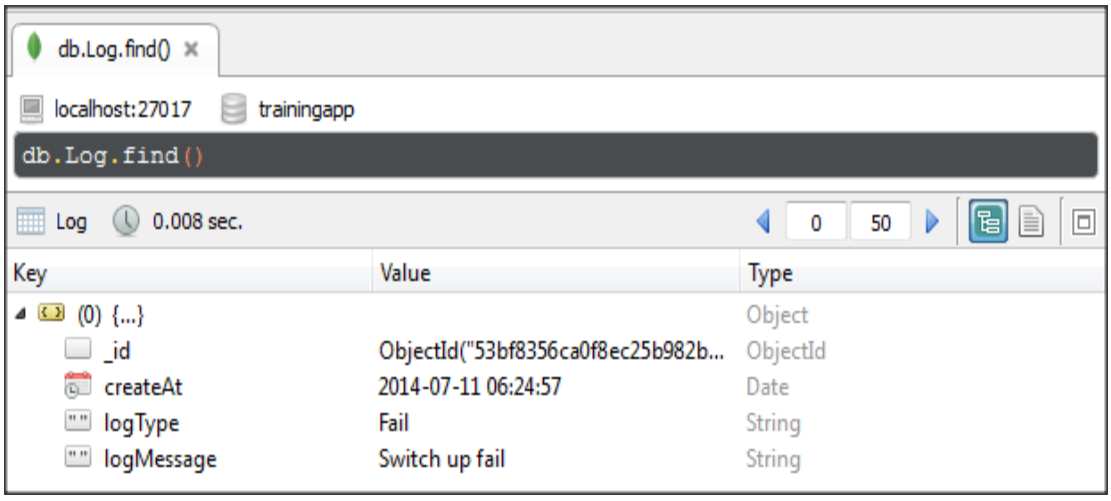
Single: no

In Robomongo:



Picture 2.111. Remove documents by criteria

The result:



Picture 2.112. The result after remove docuements

Use MongoTemplate:

```
public void removeDocuments() {
    Query query = new Query();
    query.addCriteria(Criteria.where("logType").is("Success"));
    try {
        mongoTemplate.remove(query, Log.class);
    } catch (MongoException e) {
        LoggerFactory.getLogger(getClass()).error("Remove error", e);
    }
}
```

Picture 2.113. Remove documents by criteria using java

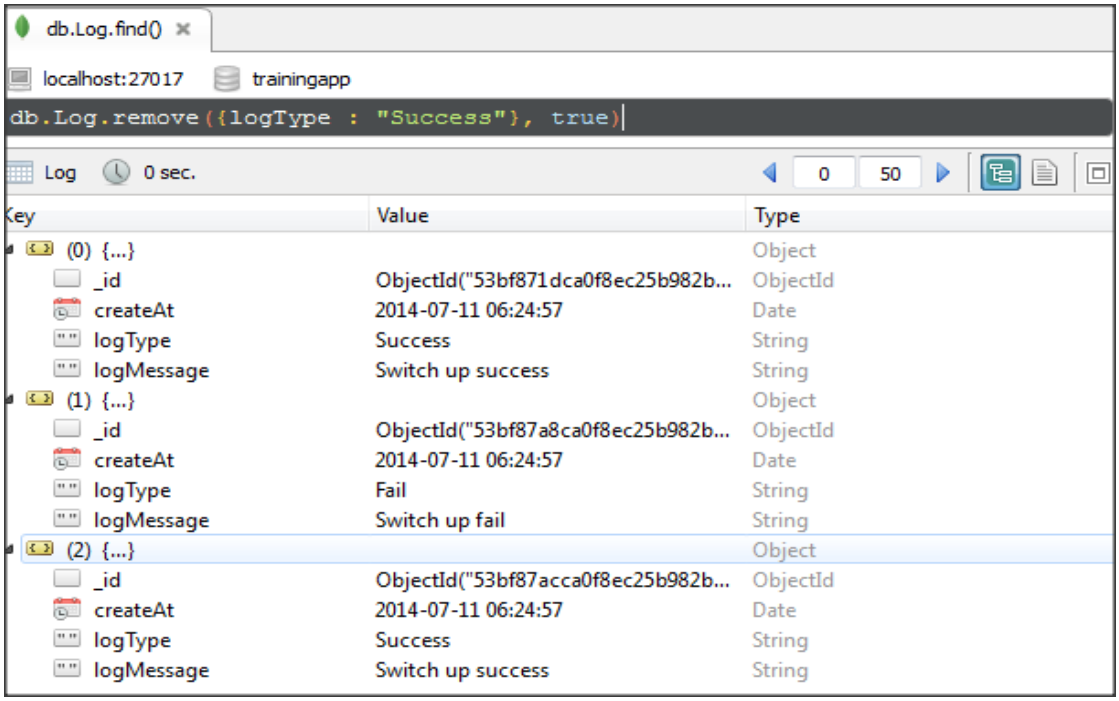
Use Repository:

```
public void removeDocuments() {
    try {
        // 1. Create a log object
        Log log = new Log();
        log.setLogType("Success");
        // 2. Use repository to delete
        logRepository.delete(log);
    } catch (MongoException e) {
        LoggerFactory.getLogger(getClass()).error("Remove error", e);
    }
}
```

Picture 2.114. Remove documents by criteria using repository

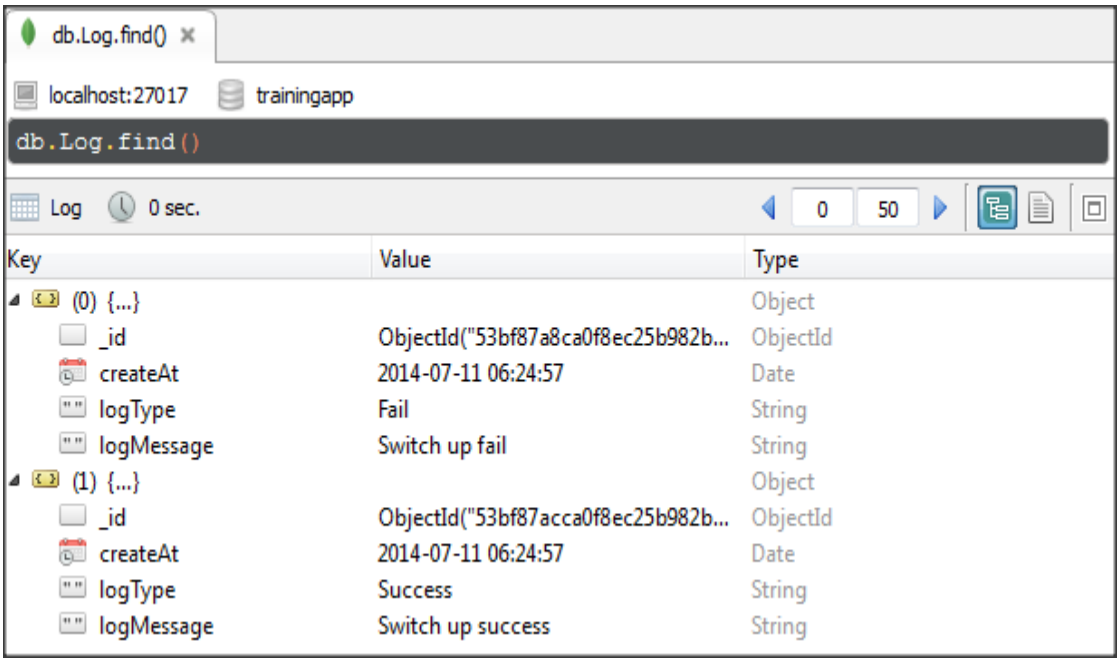
2.3.3. Remove single document with criteria

To remove single document, we can remove by _id field, or others which is an unique field. Beside, we also set single option in remove() method to remove single documents.



Picture 2.115. Remove single document by criteria

The result:



Picture 2.116. The result of result single document with criteria

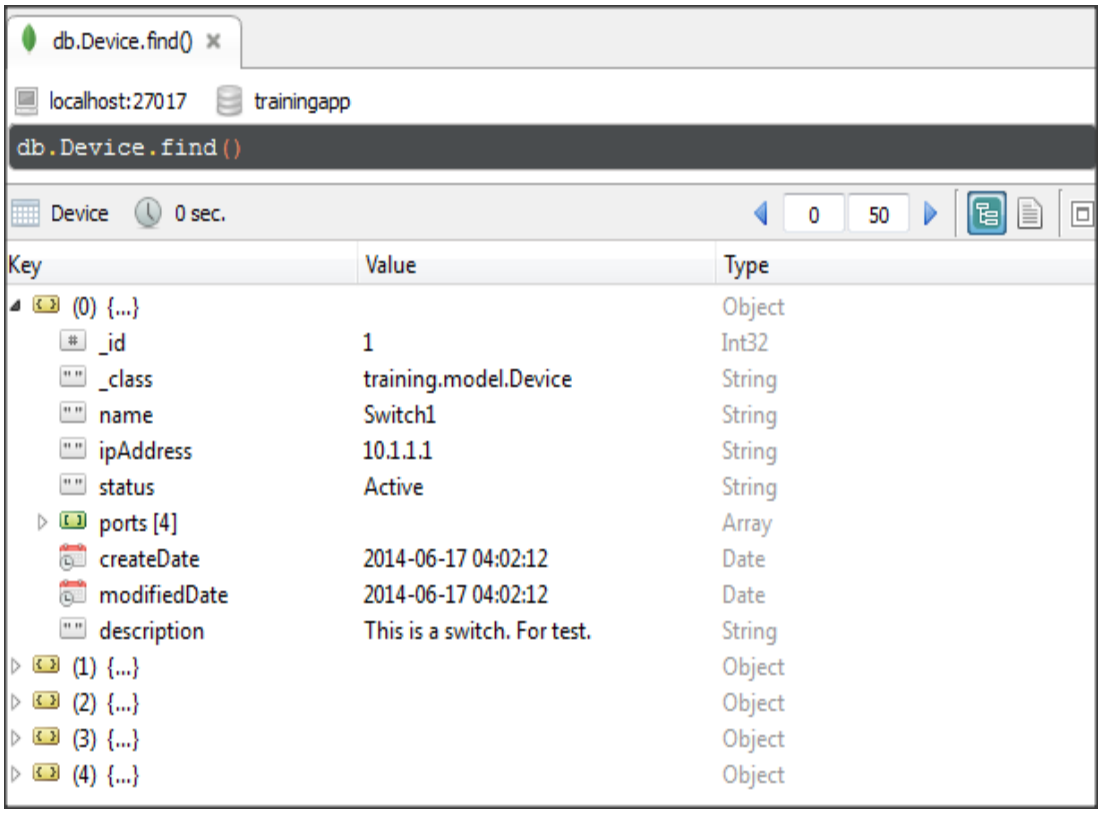
3. DBREFS

DBRefs are references from one document to another using the value of the first document's _id field, collection name, and, optionally, its database name. By including these names, DBRefs allow documents located in multiple collections to be more easily linked with documents from a single collection.

Example:

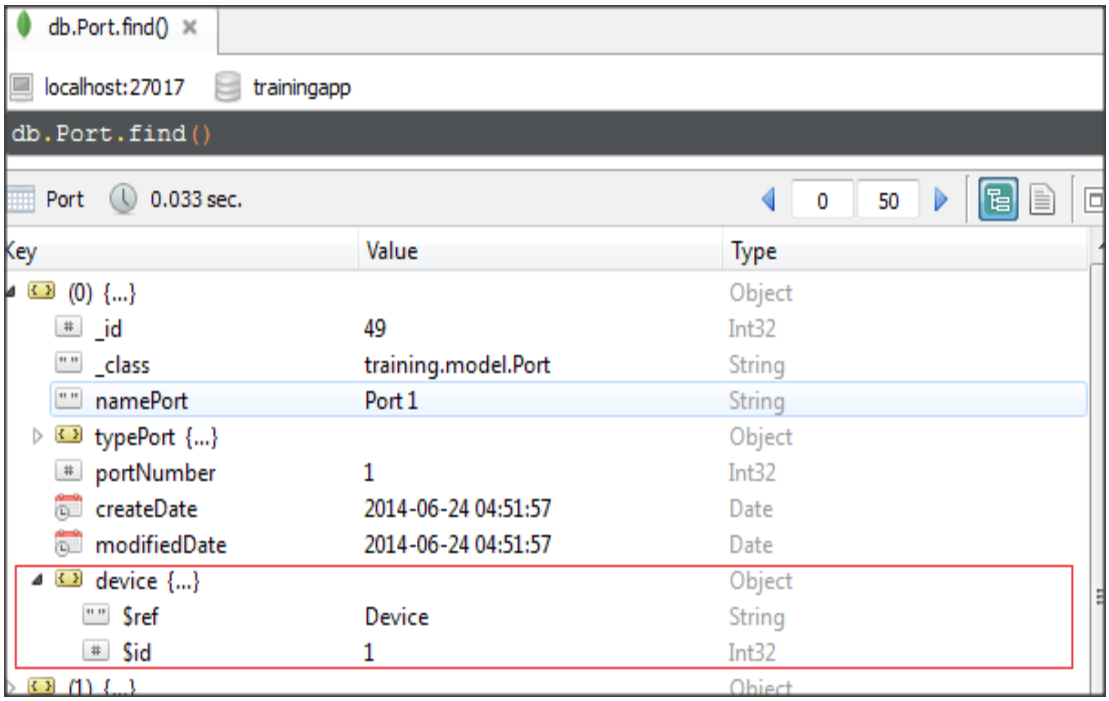
Our database store information of devices in a collection Device. And each device has many ports. I want to save information of ports in other collection. So I use DBrefs to reference port collection and device collection.

Device collection:



Picture 3.1. Device collection

Port collection:



Picture 3.2. Port collection

Each port document will has a field “device” additionally. This field will be used to reference to device which include this port.

Field “device” has 2 field:

+ $ref : is collection which is referenced.

+ $id : _id of document which is referenced.

* The order of fields in the **DBRef** matters, and you must use the above sequence when using a **DBRef.**

3.1 Query DBRefs

MongoDB don't provide tools to use DBRefs. You must use drivers which provide helper method query for DBRefs. Spring Mongo data provide an easiest way to use DBRefs is **@DBRef annotation**. You need set **@DBRef** for proper field in model class. For example:

```
import java.util.Date;

@Document(collection=Collection.PORT_COLLECTION)
public class Port {

    @Id
    private int idPort;
    private String namePort;
    private Type typePort;
    private int portNumber;

    @JsonIgnore
    @DBRef(db=Collection.DEVICE_COLLECTION)
    private Device device;

    private Date createDate;
    private Date modifiedDate;

    public int getIdPort() {
        return idPort;
    }
    public void setIdPort(int idPort) {
        this.idPort = idPort;
    }
```

Picture 3.3. Port model class

Then we use Port class for query. Information of referenced device will be gotten automatically. For example:

```
public void getPortExample() {
    Port port = portRepository.findByIdPort(50);
    System.out.println(port.toString());
}
```

Result:

```
Port [idPort=50, namePort=Port 2
 typePort=training.model.Type@63ee39cc, portNumber=2
 device=Device [id=1, name=Switch1/n ipAddress=10.1.1.1
 createDate=Tue Jun 17 11:02:12 ICT 2014, modifiedDate=Tue Jun 17 11:02:12 ICT 2014]
 createDate=Tue Jun 24 11:51:57 ICT 2014
 modifiedDate=Tue Jun 24 11:51:57 ICT 2014]
```

3.2 Restriction

DBRef is only a convention which tells driver to auto-load referenced documents. So our application will must perform addition queries.

Compare with embedded document:

- The storage overhead is smaller.
- The performance overhead is higher.

Because of DBRef is expensive for query. So it shouldn’t be used unless we work with big system that require scalability.

Appendix

3.1. MongoDB Operators

3.1.1. Query Operators

Comparison

Name	Description
`$gt`	Matches values that are greater than the value specified in the query.
`$gte`	Matches values that are greater than or equal to the value specified in the query.
`$in`	Matches any of the values that exist in an array specified in the query.
`$lt`	Matches values that are less than the value specified in the query.
`$lte`	Matches values that are less than or equal to the value specified in the query.
`$ne`	Matches all values that are not equal to the value specified in the query.
`$nin`	Matches values that **do not** exist in an array specified to the query.

Logical

Name	Description
`$or`	Joins query clauses with a logical **OR** returns all documents that match the conditions of either clause.
`$and`	Joins query clauses with a logical **AND** returns all documents that match the conditions of both clauses.
`$not`	Inverts the effect of a query expression and returns documents that do *not* match the query expression.
`$nor`	Joins query clauses with a logical **NOR** returns all documents that fail to match both clauses.

Element

Name	Description
`$exists`	Matches documents that have the specified field.
`$type`	Selects documents if a field is of the specified type.

Evaluation

Name	Description
$mod	Performs a modulo operation on the value of a field and selects documents with a specified result.
$regex	Selects documents where values match a specified regular expression.
$text	Performs text search.
$where	Matches documents that satisfy a JavaScript expression.

Geospatial

Name	Description
$geoWithin	Selects geometries within a bounding GeoJSON geometry.
$geoIntersects	Selects geometries that intersect with a GeoJSON geometry.
$near	Returns geospatial objects in proximity to a point.
$nearSphere	Returns geospatial objects in proximity to a point on a sphere.

3.1.2. Projection Operators

Array

Name	Description
$all	Matches arrays that contain all elements specified in the query.
$elemMatch	Selects documents if element in the array field matches all the specified $elemMatch condition.
$size	Selects documents if the array field is a specified size.

Name	Description
$	Projects the first element in an array that matches the query condition.
$elemMatch	Projects only the first element from an array that matches the specified $elemMatch condition.
$meta	Projects the document's score assigned during $text operation.
$slice	Limits the number of elements projected from an array. Supports skip and limit slices.

3.1.3. Update Opertors

Fields

Name	Description
`$inc`	Increments the value of the field by the specified amount.
`$mul`	Multiplies the value of the field by the specified amount.
`$rename`	Renames a field.
`$setOnInsert`	Sets the value of a field upon document creation during an upsert. Has no effect on update operations that modify existing documents.
`$set`	Sets the value of a field in a document.
`$unset`	Removes the specified field from a document.
`$min`	Only updates the field if the specified value is less than the existing field value.
`$max`	Only updates the field if the specified value is greater than the existing field value.
`$currentDate`	Sets the value of a field to current date, either as a Date or a Timestamp.

Array

Operators

Name	Description
`$`	Acts as a placeholder to update the first element that matches the query condition in an update.
`$addToSet`	Adds elements to an array only if they do not already exist in the set.
`$pop`	Removes the first or last item of an array.
`$pullAll`	Removes all matching values from an array.
`$pull`	Removes all array elements that match a specified query.
`$pushAll`	*Deprecated.* Adds several items to an array.
`$push`	Adds an item to an array.

Modifiers

Name	Description
`$each`	Modifies the `$push` and `$addToSet` operators to append multiple items for array updates.
`$slice`	Modifies the `$push` operator to limit the size of updated arrays.
`$sort`	Modifies the `$push` operator to reorder documents stored in an array.
`$position`	Modifies the `$push` operator to specify the position in the array to add elements.

Bitwise

Name	Description
`$bit`	Performs bitwise **AND**, **OR**, and **XOR** updates of integer values.

Isolation

Name	Description
`$isolated`	Modifies behavior of multi-updates to increase the isolation of the operation.

3.1.4. Types of data in MongoDB

Type	Number
Double	1
String	2
Object	3
Array	4
Binary data	5
Undefined	6
Object id	7
Boolean	8
Date	9
Null	10
Regular Expression	11
JavaScript	13
Symbol	14
JavaScript (with scope)	15
32-bit integer	16
Timestamp	17
64-bit integer	18
Min key	255
Max key	127

3.2. Collections in examples

a. Switch collection:

Switch collection

b. Log collection:

Log collection

c. Port collection:

Port collection

d. Device collection:

Device collection