MyInstagram MongoDB Shell Scripts

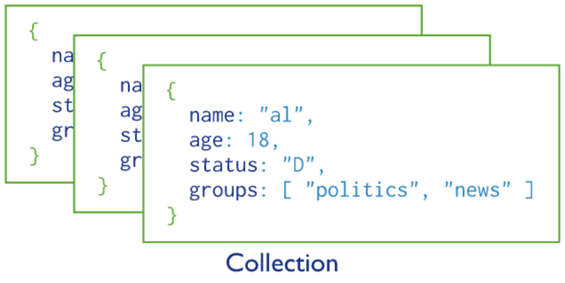
MongoDB is a Document-based NoSQL database. Unlike the Relational Databases, NoSQL Databases don't have a rigid schema. Also, MongoDB stores data in the form of documents in collections, instead of storing data in rows of tables. Also, MongoDB is a distributed(clustered) database, which makes it much more available and efficient in certain aspects than a Relational Database. Since the main aim of this study is not to discuss MongoDB in detail, and instead, is to discuss the design and implementation of Instagram's database in MongoDB, we won't dive deeper into the whats and hows of MongoD. This document's main aim is to make you familiar with some commands and terms in MongoDB. Let's start.

Basic Terms:

- Document : A document in MongoDB is nothing but an entry just like a row in Relational Databases. A document in MongoDB is a BSON document. These documents store the data in the form of Key-Value pairs.

```
{
        _id : 578,
        name : "John Doe",
        city : "New Delhi",
        interests : [ "chess", "cricket" ]
}
```

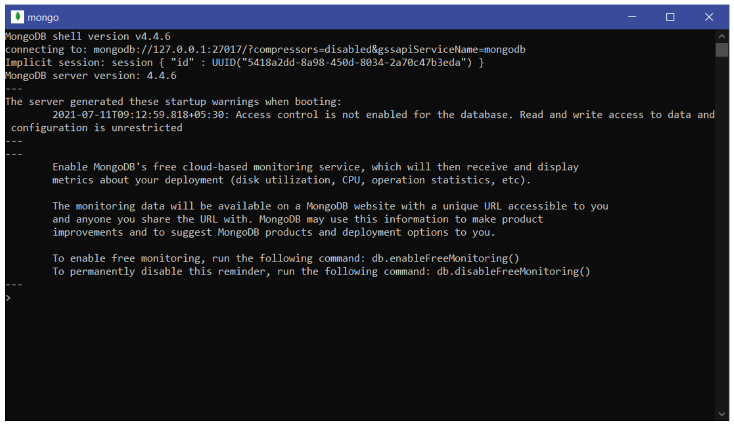
- Collection : A collection in MongoDB is nothing but a bunch of similar documents grouped and stored together just like a folder in a file system or a table in Relational Databases.



```
{
   name: "al",
   age: 18,
   status: "D",
   groups: [ "politics", "news" ]
}
```

Image from https://docs.mongodb.com/manual/core/databases-and-collections/

- Mongo Shell : Mongo shell is an interface just like “Command Prompt” in windows or “Terminal” in MacOS/Linux. All the queries for the database are to be executed here.

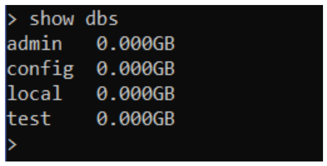


```
MongoDB shell version v4.4.6
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("5418a2dd-8a98-450d-8034-2a70c47b3eda") }
MongoDB server version: 4.4.6
---
The server generated these startup warnings when booting:
        2021-07-11T09:12:59.818+05:30: Access control is not enabled for the database. Read and write access to data and
 configuration is unrestricted
---
---
        Enable MongoDB's free cloud-based monitoring service, which will then receive and display
        metrics about your deployment (disk utilization, CPU, operation statistics, etc).

        The monitoring data will be available on a MongoDB website with a unique URL accessible to you
        and anyone you share the URL with. MongoDB may use this information to make product
        improvements and to suggest MongoDB products and deployment options to you.

        To enable free monitoring, run the following command: db.enableFreeMonitoring()
        To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
>
```

- Index : An index in MongoDB can be considered as a map which in turn helps us retrieve the desired data much faster. There are several types of indexing possible in MongoDB. For understanding indexing in MongoDB in a better way, visit here.

Basic Commands:

- *“show dbs”* : This commands shows us all the databases present on the server

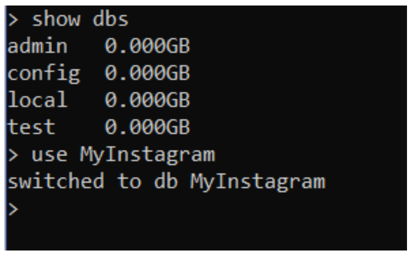


```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
test    0.000GB
>
```

- *“use <<Database Name>>”* : This command helps us switch inside the database we want to use.

  Note: Even if there doesn’t exist a database whose name is used for switching, MongoDB let’s you switch inside it. However, the database isn’t in reality created until a collection is created inside it. That means, there is no specific command to create a database in MongoDB, just use this command and switch into the one desired. If it doesn’t already exist, MongoDB will create one for you when a collection is created inside it. More about it can be read from here.

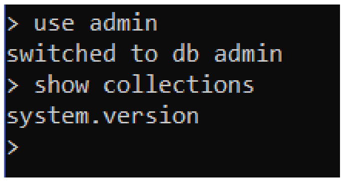
By default, the shell is in a database named “test”.



```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
test    0.000GB
> use MyInstagram
switched to db MyInstagram
>
```

Note: There does not exist a database named MyInstagram

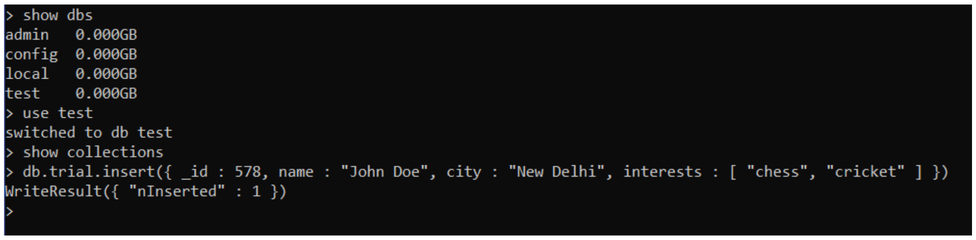
- *“show collections”* : This command lists down all the collections present in the database currently being used.



```
> use admin
switched to db admin
> show collections
system.version
>
```

Here, the admin database consists of just one collection named “system.version”.

- db.<<collection name>>.insert({ …. }) : This command inserts the document inside the { …. } in the collection mentioned in <<collection name>>.



```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
test    0.000GB
> use test
switched to db test
> show collections
> db.trial.insert({ _id : 578, name : "John Doe", city : "New Delhi", interests : [ "chess", "cricket" ] })
WriteResult({ "nInserted" : 1 })
>
```

Note: There was no collection named “trial” in the “test” database. However, just like a database, MongoDB itself creates a collection if there doesn’t exist one previously with that name.

There are many more utility commands, however, it is out of the scope of this report to mention them all here. You can find them here.

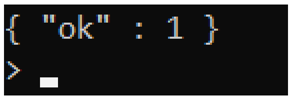
The next section states all the necessary shell scripts required for the implementation of MyInstagram in MongoDB.

Implementation of MyInstagram

This section of the report consists of all the shell scripts required to create the collections, assign these collections their schemas, for creating indexes wherever required and also to implement queries listed in “MyInstagram Query List.pdf”.

Collection Scripts:

To create these collections in your database, copy, paste and execute the following scripts one by one in your mongo shell. Upon successful execution and hence creation of a collection, you will receive an object similar to this in the shell:



```
{ "ok" : 1 }
>
```

1. “users” Collection:

   Each document in this collection consists of the details that Instagram maintains about a user. The settings and preferences of the user such as blocked profiles, posts a user is tagged in, private/public account, close friends etc. The script for creating this collection, along with its JSON Schema validator is given below.

```
db.createCollection("users", {
        validator : {
                $jsonSchema: {
                    bsonType : "object",
                    required : [ "createdAt", "tokenVersion", "fullname",
"username", "language", "password", "DOB", "gender", "blockedUsers",
"savedPosts", "followers", "following", "hashtags", "accountType",
"accountType2", "requests", "taggedIn", "liked", "account", "verified",
"blockUserFromComment", "manuallyApproveTags", "toConfirmTag",
"allowTagsFrom", "likeViewCount", "allowCommentFrom", "checkComment",
"offensiveWords", "hideStory", "allowAtMention", "closeFriends",
"allowToShareOnStory", "muteStories", "mutePosts", "manageNotification",
"highlight", "isMentionedIn", "storyArchive", "liveArchive",
"restrictAccounts", "countdown", "postCount", "blockedFrom", "guides",
"hideFromProfileGrid", "posts", "stories", "recentStory", "searchHistory"
],
                    properties : {

                        createdAt : {
                            bsonType : "date",
                        },

                        tokenVersion : {
```

```
                bsonType : "bool",
            },

            fullname : {
                bsonType : "string",
            },

            username : {
                bsonType : "string",
                maxLength : 30,
            },

            email : {
                bsonType : "string",
            },

            phoneNo : {
                bsonType : "string",
            },

            language : {
                bsonType : "string",
                maxLength : 50,
            },

            password : {
                bsonType : "string",
            },

            DOB : {
                bsonType : "date",
            },

            gender : {
                bsonType : "string",
                maxLength : 60,
            },

            blockedUsers : {
                bsonType : "array",
                items : {
                    bsonType : "objectId",
                },
            },

            savedPosts : {
                bsonType : "array",
                items : {
                    bsonType : "objectId",
                },
            },
                        followers : {
                            bsonType : "array",
                            items : {
                                bsonType : "objectId",
                            },
                        },

                        following : {
                            bsonType : "array",
                            items : {
                                bsonType : "objectId",
                            },
                            maxLength : 7500,
                        },

                        hashtags : {
                            bsonType : "array",
                            items : {
                                bsonType : "string",
                            },
                        },

                        accountType : {
                            bsonType : "bool",
                        },

                        accountType2 : {
                            enum : [ "normal", "business", "professional"
],
                        },

                        requests : {
                            bsonType : "array",
                            items : {
                                bsonType : "objectId",
                            },
                        },

                        taggedIn : {
                            bsonType : "array",
                            items : {
                                bsonType : "objectId",
                            },
                        },

                        liked : {
                            bsonType : "array",
                            items : {
                                bsonType : "objectId",
                            },
                        },
                        account : {
                            bsonType : "bool",
                        },

                        verified : {
                            bsonType : "bool",
                        },

                        blockUserFromComment : {
                            bsonType : "array",
                            items : {
                                bsonType : "objectId",
                            },
                        },

                        manuallyApproveTags : {
                            bsonType : "bool",
                        },

                        toConfirmTag : {
                            bsonType : "array",
                            items : {
                                bsonType : "objectId",
                            },
                        },

                        allowTagsFrom : {
                            enum : [ "all", "none", "following" ],
                        },

                        likeViewCount : {
                            bsonType : "bool",
                        },

                        allowCommentFrom : {
                            enum : [ "everyone", "following", "followers",
"followingFollowers" ],
                        },

                        checkComment : {
                            bsonType : "bool",
                        },

                        offensiveWords : {
                            bsonType : "array",
                            items : {
                                bsonType : "string",
                            },
                        },

                        hideStory : {
                            bsonType : "array",
                        items : {
                            bsonType : "objectId",
                        },
                    },

                    allowAtMention : {
                        enum : [ "all", "none", "following" ],
                    },

                    closeFriends : {
                        bsonType : "array",
                        items : {
                            bsonType : "objectId",
                        },
                    },

                    allowToShareOnStory : {
                        bsonType : "bool",
                    },

                    muteStories : {
                        bsonType : "array",
                        items : {
                            bsonType : "objectId",
                        },
                    },

                    mutePosts : {
                        bsonType : "array",
                        items : {
                            bsonType : "objectId",
                        },
                    },

                    manageNotification : {
                        bsonType : "object",
                        properties : {

                            posts : {
                                bsonType : "array",
                                items : {
                                    bsonType : "objectId"
                                },
                            },

                            stories : {
                                bsonType : "array",
                                items : {
                                    bsonType : "objectId"
                                },
                            },
        igtv : {
            bsonType : "array",
            items : {
                bsonType : "objectId"
            },
        },

        reels : {
            bsonType : "array",
            items : {
                bsonType : "objectId"
            },
        },

        liveVideos : {
            bsonType : "object",
            properties : {

                all : {
                    bsonType : "array",
                    items : {
                        bsonType : "objectId"
                    },
                },

                some : {
                    bsonType : "array",
                    items : {
                        bsonType : "objectId"
                    },
                },

                none : {
                    bsonType : "array",
                    items : {
                        bsonType : "objectId"
                    },
                },

            },
        },

    },
},

highlight : {

    bsonType : "array",
    items : {

        bsonType : "object",
        required : [ "title", "stories", "cover"
],
                                properties : {

                                    title : {
                                        bsonType : "string",
                                    },

                                    stories : {
                                        bsonType : "array",
                                        items : {
                                            bsonType : "objectId",
                                        },
                                    },

                                    cover : {
                                        bsonType : "string",
                                    },

                                },
                            },

                        },

                        isMentionedIn : {
                            bsonType : "array",
                            items : {
                                bsonType : "objectId",
                            },
                        },

                        storyArchive : {
                            bsonType : "bool",
                        },

                        liveArchive : {
                            bsonType : "bool",
                        },

                        restrictAccounts : {
                            bsonType : "array",
                            items : {
                                bsonType : "objectId",
                            },
                        },

                        countdown : {
                            bsonType : "array",
                            items : {
                                bsonType : "date"
                            },
                        },
                        profilePicture : {
                            bsonType : "string",
                        },

                        bio : {
                            bsonType : "string",
                            maxLength : 200,
                        },

                        website : {
                            bsonType : "string",
                            maxLength : 60,
                        },

                        postCount : {
                            bsonType : "int",
                        },

                        blockedFrom : {
                            bsonType : "array",
                            items : {
                                bsonType : "objectId",
                            },
                        },

                        guides : {
                            bsonType : "array",
                            items : {

                                bsonType : "object",
                                required : [ "title", "description",
"cover", "posts" ],

                                properties : {

                                    title : {
                                        bsonType : "string",
                                    },

                                    description : {
                                        bsonType : "string",
                                    },

                                    cover : {
                                        bsonType : "string",
                                    },

                                    posts : {
                                        bsonType : "array",
                                        items : {
                                            bsonType : "object",
                                            properties : {
                                                title : {
```

```
                            bsonType : "string",
                        },
                        description : {
                            bsonType : "string",
                        },
                        post_id : {
                            bsonType : "objectId"
                        },
                    }
                }
            },

        },

    },
},

hideFromProfileGrid : {
    bsonType : "array",
    items : {
        bsonType : "objectId",
    },
},

posts : {
    bsonType : "array",
    items : {
        bsonType : "objectId",
    },
},

stories : {
    bsonType : "array",
    items : {
        bsonType : "objectId",
    },
},

recentStory : {
    bsonType : "array",
    items : {
        bsonType : "objectId",
    },
},

searchHistory : {
    bsonType : "array",
    items : {
        bsonType : "string",
    },
},
                    },
                },
            },

        validationLevel: "strict",
        validationAction: "error",

    })
```

2. “posts” Collection

   Each document in this collection consists of the details that Instagram maintains about the posts made by the users. The settings of the post such as enable/disable commenting, archived/unarchived etc. These documents also contain the main data about the post such as the caption, location, images/videos etc.

   Note: Images and videos in this implementation are assumed to be stored on some other cloud storage service providers such as AWS etc. Hence, the data type maintained for images/videos everywhere in the database is “string”, since we’ll be storing links of the original storage location of those images/videos.

```
db.createCollection("posts", {
        validator : {
            $jsonSchema : {
                bsonType : "object",
                required : [ "u_id", "createdAt", "modifiedAt", "caption",
"type", "likes", "likedBy", "savedCount", "deleted", "commenting", "archived",
"count", "content", "sharedCount", "disabled", "insights" ],
                properties : {

                    u_id : {
                        bsonType : "objectId",
                    },

                    createdAt : {
                        bsonType : "date",
                    },

                    modifiedAt : {
                        bsonType : "date",
                    },

                    caption : {
                        bsonType : "string",
                        maxLength : 2200,
                    },
location : {
    bsonType : "string",
},

tagged : {
    bsonType : "array",
    items : {
        bsonType : "object",
        required : [ "u_id", "status" ],
        properties : {

            u_id : {
                bsonType : "objectId",
            },

            status : {
                enum : [ "enabled", "disabled" ],
            },

        },
    },
},

type : {
    enum : [ "reel", "post", "igtv" ],
},

likes : {
    bsonType : "int",
},

likedBy : {
    bsonType : "array",
    items : {
        bsonType : "objectId",
    },
},

savedCount : {
    bsonType : "int",
},

hashTags : {
    bsonType : "array",
    items : {
        bsonType : "string",
    },
},

deleted : {
    bsonType : "bool",
},

commenting : {
    bsonType : "bool",
},

archived : {
    bsonType : "bool",
},

count : {
    bsonType : "bool",
},

title : {
    bsonType : "string",
},

content : {
    bsonType : "array",
    items : {
        bsonType : "string",
    },
},

music : {
    bsonType : "string",
},

sharedCount : {
    bsonType : "int",
},

disabled : {
    bsonType : "bool",
},

insights : {
    bsonType : "object",
    properties : {
        profileVisits : {
            bsonType : "int",
        },
        follows : {
            bsonType : "int",
        },
        impressions : {
            bsonType : "int",
        },
    },
},
                },
            },
        },

        validationLevel: "strict",
        validationAction: "error",

    })
```

3. “stories” Collection

   Each document in this collection consists of the details that Instagram maintains about the stories posted by the users. The settings of a story such as enable/disable sharing, closeFriends/forAll etc. These documents also contain the main data about the story such as the quiz responses, question responses, images/videos etc.

```
db.createCollection("stories", {
        validator : {
            $jsonSchema : {
                bsonType : "object",
                required : [ "viewedBy", "createdAt", "viewType", "u_id",
"type", "mention", "archived", "sharedCount", "content", "deleted",
"disabled", "insights" ],
                properties : {

                    viewedBy : {
                        bsonType : "array",
                        items : {
                            bsonType : "objectId",
                        },
                    },

                    createdAt : {
                        bsonType : "date",
                    },

                    viewType : {
                        enum : [ "closeFriends", "all", "followers" ],
                    },

                    u_id : {
                        bsonType : "objectId",
                    },

                    type : {
```

```
                    bsonType : "object",
                    properties : {

                        questions : {
                            bsonType : "bool",
                        },

                        emojiSlider : {
                            bsonType : "bool",
                        },

                        poll : {
                            bsonType : "bool",
                        },

                        quiz : {
                            bsonType : "bool",
                        },

                        countdown : {
                            bsonType : "bool",
                        },

                    },

                },

                details : {
                    bsonType : "array",
                    items : {
                        bsonType : "object",
                        required : [ "u_id", "response" ],
                        properties : {

                            u_id : {
                                bsonType : "objectId",
                            },

                            responseOf : {
                                bsonType : "string"
                            },

                            response : {
                                bsonType : "string",
                            },

                        },
                    },
                },

                mention : {
                    bsonType : "array",
                    items : {
                        bsonType : "objectId",
                    },
                },

                location : {
                    bsonType : "string",
                },

                archived : {
                    bsonType : "bool",
                },

                music : {
                    bsonType : "string",
                },

                link : {
                    bsonType : "string",
                },

                sharedCount : {
                    bsonType : "int",
                },

                content : {
                    bsonType : "object",
                    required : [ "media" ],
                    properties : {

                        media : {
                            bsonType : "string",
                        },

                    },
                },

                deleted : {
                    bsonType : "bool",
                },

                disabled : {
                    bsonType : "bool",
                },

                insights : {
                    bsonType : "object",
                    properties : {
                        profileVisits : {
                            bsonType : "int",
                        },
                        impressions : {
```

```
                            bsonType : "int",
                        },
                        follows : {
                            bsonType : "int",
                        },
                        forward : {
                            bsonType : "int",
                        },
                        exitted : {
                            bsonType : "int",
                        },
                        next : {
                            bsonType : "int",
                        },
                    },
                },

            }
        }
    },

    validationLevel: "strict",
    validationAction: "error",

})
```

4. “comments” Collection

   Each document in this collection consists of the details that Instagram maintains about the comments posted by the users. These documents contain the text of comment, likes count, record of the users who liked the comment etc.

```
db.createCollection("comments", {
        validator : {
            $jsonSchema : {
                bsonType : "object",
                required : [ "p_id", "u_id", "likes", "likedBy", "createdAt",
"rComment", "commentText", "owner_id", "disabled", "bComment" ],
                properties : {

                    p_id : {
                        bsonType : "objectId",
                    },

                    u_id : {
                        bsonType : "objectId",
                    },
                    likes : {
                        bsonType : "int",
                    },

                    likedBy : {
                        bsonType : "array",
                        items : {
                            bsonType : "objectId",
                        },
                    },

                    createdAt : {
                        bsonType : "date",
                    },

                    rComment : {
                        bsonType : "bool",
                    },

                    commentText : {
                        bsonType : "string",
                    },

                    owner_id : {
                        bsonType : "objectId",
                    },

                    disabled : {
                        bsonType : "bool",
                    },

                    bComment : {
                        bsonType : "bool",
                    },

                },
            },
        },

        validationLevel: "strict",
        validationAction: "error",

    })
```

5. “deleteAccounts” Collection

   Each document in this collection just consists of the referenceID’s of users who’ve requested for their account’s deletion.

```
db.createCollection("deleteAccounts", {
        validator : {
            $jsonSchema : {
                bsonType : "object",
                required : [ "u_id" ],
                properties : {

                    u_id : {
                        bsonType : "objectId",
                    },

                },
            },
        },

        validationLevel: "strict",
        validationAction: "error",

    })
```

6. “liveVideos” Collection

   Each document in this collection just consists of the live sessions archived by the user.

```
db.createCollection("liveVideos", {
        validator : {
            $jsonSchema : {
                bsonType : "object",
                required : [ "video", "createdAt", "u_id" ],
                properties : {

                    video : {
                        bsonType : "string",
                    },

                    createdAt : {
                        bsonType : "date",
                    },
                    u_id : {
                        bsonType : "objectId",
                    },

                },
            },
        },

        validationLevel: "strict",
        validationAction: "error",

    })
```

7. “requestsForVerification” Collection

   Each document in this collection just consists of the various details asked by Instagram for verifying a user’s account, such as, legal-identification proof, category, known as, etc.

```
db.createCollection("requestsForVerification", {
        validator : {
            $jsonSchema : {
                bsonType : "object",
                required : [ "u_id", "fullname", "knownAs", "category",
"proof", "status" ],
                properties : {

                    u_id : {
                        bsonType : "objectId",
                    },

                    fullname : {
                        bsonType : "string",
                    },

                    knownAs : {
                        bsonType : "string",
                    },

                    category : {
                        bsonType : "string",
                    },

                    proof : {
                        bsonType : "string",
                    },
                    status : {
                        enum : [ "posted", "viewed", "underVerification",
"approved", "declined" ],
                    },

                },
            },
        },

        validationLevel: "strict",
        validationAction: "error",

    })
```

8. “reportLogs” Collection

   Each document in this collection just consists of the various details asked by Instagram for verifying a user’s account, such as, legal-identification proof, category, known as, etc.

```
db.createCollection("reportLogs", {
        validator : {
            $jsonSchema : {
                bsonType : "object",
                required : [ "object_id", "entity", "category" ],
                properties : {

                    object_id : {
                        bsonType : "objectId",
                    },

                    category : {
                        bsonType : "string",
                    },

                    entity : {
                        enum : [ "profile", "post", "story", "comment" ],
                    },

                },
            },
        },

        validationLevel: "strict",
        validationAction: "error",
    })
```

9. “hashTags” Collection

   Each document in this collection just consists of the various posts’ referceId’s corresponding to the hashtags which are used in those posts.

```
db.createCollection("hashTags", {
        validator : {
            $jsonSchema : {
                bsonType : "object",
                required : [ "_id", "posts" ],
                properties : {

                    _id : {
                        bsonType : "string"
                    },

                    posts : {
                        bsonType : "array",
                        items : {
                            bsonType : "objectId",
                        },
                    },

                },
            },
        },

        validationLevel: "strict",
        validationAction: "error",

    })
```

10. “logs” Collection

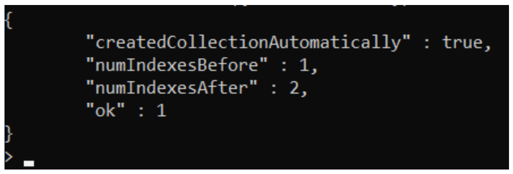
    Each document in this collection consists of a new log entry determining the action performed as well as the attributes interacted with. This is referred whenever a user asks for data such as a track of all previous usernames, emails etc.

```
db.createCollection("logs", {
        validator : {
            $jsonSchema : {
                bsonType : "object",
                required : [ "u_id", "createdAt", "attribute",
"activityRelated" ],
                properties : {

                    u_id : {
                        bsonType : "objectId",
                    },

                    createdAt : {
                        bsonType : "date",
                    },

                    attribute : {
                        bsonType : "string",
                    },

                    value : {
                        bsonType : "string",
                    },

                    activityRelated : {
                        bsonType : "bool",
                    },

                },
            },
        },

        validationLevel: "strict",
        validationAction: "error",

    })
```

After successful creation of all these collections in a database, one can move to storing data in these. Shell scripts for creating indexes and performing CRUD Operations (Create, Read, Update, Delete) are all included in the following subsections.

Index Creation Scripts:

To create the required indexes on the collections created using the scripts above, execute the following scripts. Upon successful execution of each of these scripts, you will receive an acknowledgement object in return on the shell similar to this:



```
{
        "createdCollectionAutomatically" : true,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

- "users" Collection

  a. Index on "username" attribute

  ```
  db.users.createIndex({ username : 1 })
  ```

  b. Index on "phoneNo" attribute

  ```
  db.users.createIndex({ phoneNo : 1 })
  ```

  c. Index on "email" attribute

  ```
  db.users.createIndex({ email : 1 })
  ```

- "comments" Collection

  a. Complex index on "_id", post's "p_id" and comment poster's "u_id" attributes

  ```
  db.comments.createIndex({ _id : 1, p_id : 1, u_id : 1})
  ```

- "logs" Collection

  a. Complex index on "u_id", "activityRelated" attributes

  ```
  db.logs.createIndex({ _id : 1, activityRelated : 1 })
  ```

b. Complex index on”u_id”, “attribute” attributes

```
db.logs.createIndex({ u_id : 1, attribute : 1 })
```

- “reportLogs” Collection

  a. Index on “category” attribute

  ```
  db.reportLogs.createIndex({ category : 1 })
  ```

  b. Index on “entity” attribute

  ```
  db.reportLogs.createIndex({ entity : 1 })
  ```

- “posts” Collection

  a. Index on “location” attribute

  ```
  db.posts.createIndex({ location : 1 })
  ```

After successful creation of these indexes, we can now move to the shell scripts for query executions.

Scripts for Query Execution:

These scripts are solely for performing CRUD operations on the database we’ve set up in the previous 2 sub-sections.

Note:

- <<value>> : This resembles the value to be entered while making entries. This is to be replaced by the values to be inserted.

  Example : “<<Your name>>” => “John Doe”
  <<u_id>> => ObjectId(“60f12xxxxxxxxxxxxx6c7c1”)

- Some queries have log queries in the end and are important since without those queries being executed, logs cannot be maintained.

Let’s Start.

- Signup

  ➔ First we’ll check whether the username we are trying to sign up with is already taken up or not. If it is already taken up, change the desired username, else continue to (b)

```
db.users.find({ username : "<<Desired Username>>" }, { _id : 0, username : 1
}).pretty()
```

  If this query does not return anything, it means that the username is not already taken up and hence can be used for signing up.

  If this query returns an object similar to { "username" : "<<Desired Username>>" }, then the username is already taken up and you must change your username before proceeding. Execute (a) once again for the changed username to see if that is taken up already or not too.

  ➔ Check for the phone no. or email used for signing up.

  - If using email

```
db.users.find( { email : "<<Your Email>>" }, { email : 1, _id : 0 }
).pretty()
```

    If this query does not return anything, it means that the email is not already taken up and hence can be used for signing up.

    If this query returns an object similar to { "email" : "<<Your Email>>" }, then the email is already taken up and you must change your email before proceeding. Execute this query once again for the changed email to see if that is taken up already or not too.

  - Is using phoneNo

```
db.users.find( { phoneNo : "<<Your PhoneNo.>>" }, { phoneNo : 1,
_id : 0 } ).pretty()
```

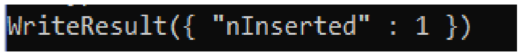
    If this query does not return anything, it means that the phone number is not already taken up and hence can be used for signing up.

    If this query returns an object similar to { "phoneNo" : "<<Your PhoneNo.>>" }, then the phone number is already taken up and you must change your phone number before proceeding. Execute this query once again for the changed phone number to see if that is taken up already or not too.

➔ Now that we've checked for both the username and the email/phone number, we can Sign Up using the following script.

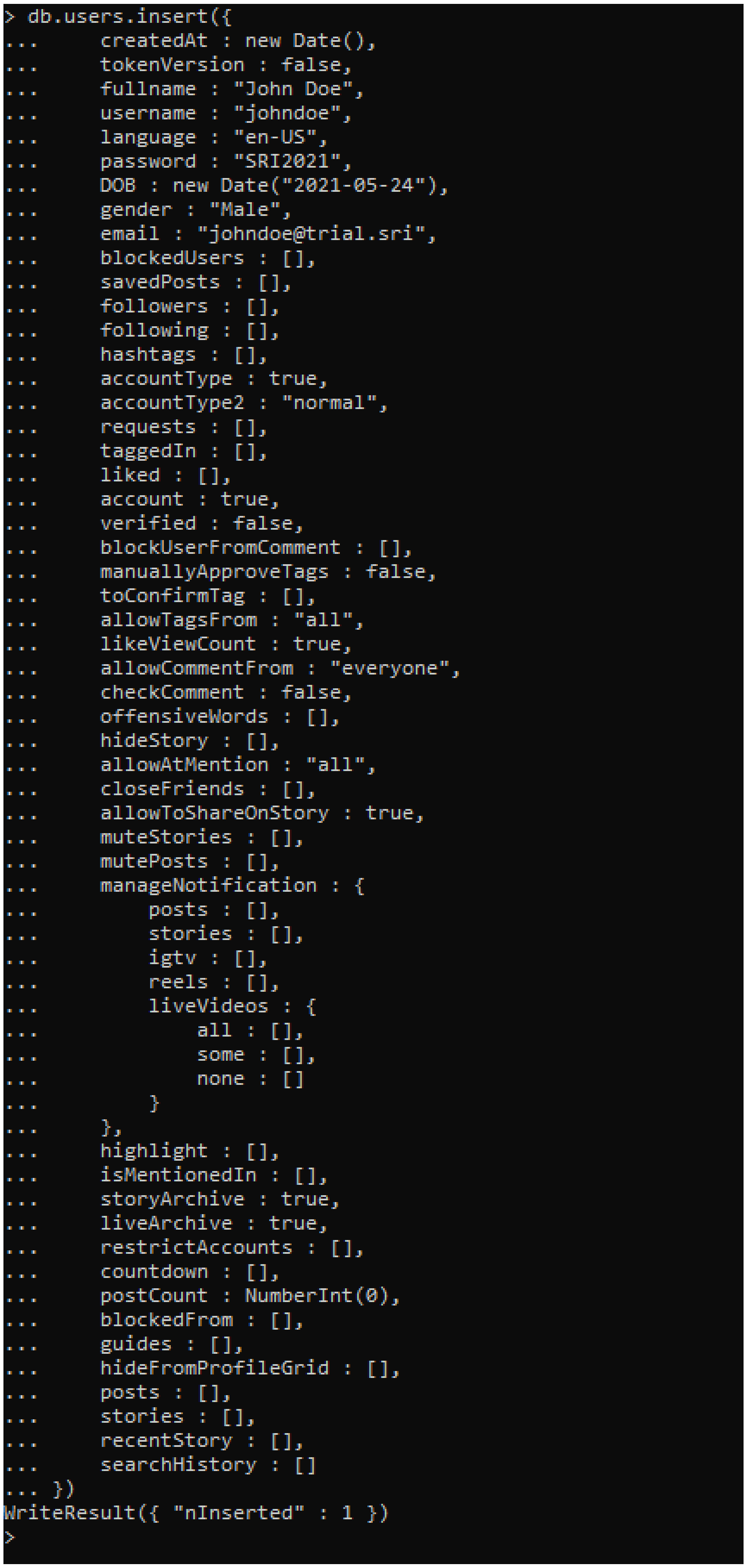
```
db.users.insert({
    createdAt : new Date(),
    tokenVersion : false,
    fullname : "<<Your Name>>",
    username : "<<Desired Username>>",
    language : "en-US",
    password : "<<Your Password>>",
    DOB : new Date("YYYY-mm-dd"),
    gender : "<<Your Gender>>",
    phoneNo/email : "<<Enter either your phoneNo/email and strike off
the other one from field name>>",
    blockedUsers : [],
    savedPosts : [],
    followers : [],
    following : [],
    hashtags : [],
    accountType : true,
    accountType2 : "normal",
    requests : [],
    taggedIn : [],
    liked : [],
    account : true,
    verified : false,
    blockUserFromComment : [],
    manuallyApproveTags : false,
    toConfirmTag : [],
    allowTagsFrom : "all",
    likeViewCount : true,
    allowCommentFrom : "everyone",
    checkComment : false,
    offensiveWords : [],
    hideStory : [],
    allowAtMention : "all",
    closeFriends : [],
    allowToShareOnStory : true,
    muteStories : [],
    mutePosts : [],
    manageNotification : {
        posts : [],
        stories : [],
        igtv : [],
        reels : [],
        liveVideos : {
            all : [],
            some : [],
            none : []
        }
    },
    highlight : [],
    isMentionedIn : [],
    storyArchive : true,
    liveArchive : true,
    restrictAccounts : [],
    countdown : [],
    postCount : NumberInt(0),
    blockedFrom : [],
    guides : [],
    hideFromProfileGrid : [],
    posts : [],
    stories : [],
    recentStory : [],
    searchHistory : []
})
```

Upon successful execution of this query, you'll be receiving an object similar to this on shell:



```
WriteResult({ "nInserted" : 1 })
```

Example:



```
> db.users.insert({
...     createdAt : new Date(),
...     tokenVersion : false,
...     fullname : "John Doe",
...     username : "johndoe",
...     language : "en-US",
...     password : "SRI2021",
...     DOB : new Date("2021-05-24"),
...     gender : "Male",
...     email : "johndoe@trial.sri",
...     blockedUsers : [],
...     savedPosts : [],
...     followers : [],
...     following : [],
...     hashtags : [],
...     accountType : true,
...     accountType2 : "normal",
...     requests : [],
...     taggedIn : [],
...     liked : [],
...     account : true,
...     verified : false,
...     blockUserFromComment : [],
...     manuallyApproveTags : false,
...     toConfirmTag : [],
...     allowTagsFrom : "all",
...     likeViewCount : true,
...     allowCommentFrom : "everyone",
...     checkComment : false,
...     offensiveWords : [],
...     hideStory : [],
...     allowAtMention : "all",
...     closeFriends : [],
...     allowToShareOnStory : true,
...     muteStories : [],
...     mutePosts : [],
...     manageNotification : {
...         posts : [],
...         stories : [],
...         igtv : [],
...         reels : [],
...         liveVideos : {
...             all : [],
...             some : [],
...             none : []
...         }
...     },
...     highlight : [],
...     isMentionedIn : [],
...     storyArchive : true,
...     liveArchive : true,
...     restrictAccounts : [],
...     countdown : [],
...     postCount : NumberInt(0),
...     blockedFrom : [],
...     guides : [],
...     hideFromProfileGrid : [],
...     posts : [],
...     stories : [],
...     recentStory : [],
...     searchHistory : []
... })
WriteResult({ "nInserted" : 1 })
>
```

If not receiving such an object, and receiving an error object, check the credentials and try again.

Log Queries:

To implement these queries, "_id" of the user is required. It can be retrieved using the following query.

```
db.users.find({ username : "<<Your Username>>" }, { _id : 1
}).pretty()
```

The "_id" received in the result object is the user's "_id". This can now be used to execute the queries below.

If signed up using email:

```
db.logs.insert({ u_id : <<user_id>>, createdAt : new Date(), attribute
: "SignUp", value : { username : "<<Your Username>>", fullname :
"<<Your name>>", email : "<<Your Email>>" }, activityRelated : false
})
```

If signed up using phone number:

```
db.logs.insert({ u_id : <<user_id>>, createdAt : new Date(), attribute
: "SignUp", value : { username : "<<Your Username>>", fullname :
"<<Your name>>", phoneNo : "<<Your PhoneNo.>>" }, activityRelated :
false })
```

- Login

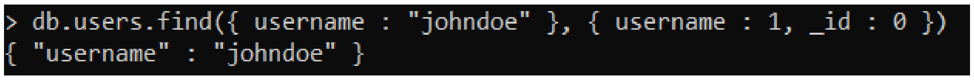
  There are 3 ways in which a user can login. A user can either login using his/her username or email or phoneNo. The three queries below are for each of these scenarios and hence a user should execute the valid queries.

  ➔ Logging In using username

  1. Check whether the username is present or not

```
db.users.find({ username : "<<Your Username>>" }, { username : 1, _id : 0
}).pretty()
```

If this does not return anything, that means the username queried is invalid. If an object => { "username" : "<<Your Username>>" } is returned, execute (2).



```
> db.users.find({ username : "johndoe" }, { username : 1, _id : 0 })
{ "username" : "johndoe" }
```

2. Verify password corresponding to this username

```
db.users.find( { username : "<<Your Username>>", password : "<<Your
Password>>" }, { username : 1, password : 1, _id : 0 } ).pretty()
```

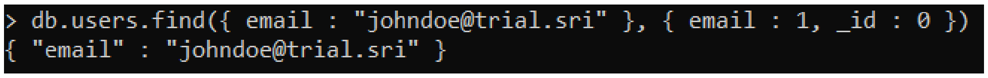
If this does not return anything, the password entered is incorrect. If an object => { "username" : "<<Your Username>>", "password" : "<<Your Password>>" } is returned, the credentials you entered are correct and you're now logged in.

➔ Logging In using email

1. Check whether the email is present or not

```
db.users.find({ email : "<<Your Email>>" }, { email : 1, _id : 0
}).pretty()
```

If this does not return anything, that means the email queried is invalid. If an object => { "email" : "<<Your Email>>" } is returned, execute (2).



```
> db.users.find({ email : "johndoe@trial.sri" }, { email : 1, _id : 0 })
{ "email" : "johndoe@trial.sri" }
```

2. Verify password corresponding to this email

```
db.users.find({ email : "<<Your Email>>", password : "<<Your Password>>"
}, { email : 1, password : 1, _id : 0 }).pretty()
```

If this does not return anything, the password entered is incorrect. If an object => { "email" : "<<Your Email>>", "password" : "<<Your Password>>" } is returned, the credentials you entered are correct and you're now logged in.

➔ Logging In using phone number

1. Check whether the phone number is present or not

```
db.users.find({ phoneNo : "<<Your PhoneNo.>>" }, { phoneNo : 1, _id : 0
}).pretty()
```

If this does not return anything, that means the phoneNo queried is invalid. If an object => { "phoneNo" : "<<Your PhoneNo.>>" } is returned, execute (2).

2. Verify password corresponding to this phone number

```
db.users.find( { phoneNo : "<<Your PhoneNo.>>", password : "<<Your
Password>>" }, { phoneNo : 1, password : 1, _id : 0 } ).pretty()
```

If this does not return anything, the password entered is incorrect. If an object => { "phoneNo" : "<<Your PhoneNo.>>", "password" : "<<Your Password>>" } is returned, the credentials you entered are correct and you're now logged in.

Log Queries:

```
db.logs.insert({ u_id : <<user_id>>, createdAt : new Date(), attribute :
"Login", activityRelated : false })
```

- Update Profile

Here, there are 8 details one can update about himself or herself. For each of these details, a separate query has to be executed.

➔ Update username

1. Check whether the new username you want to set is already in use or not

```
db.users.find({ username : "<<Your New Username>>" }, { username : 1, _id
: 0 }).pretty()
```

If this returns an object => { "username" : "<<Your New Username>>" }, then you can't set this Username as your new Username since it's already been taken by some other account. If this does not return anything, execute (2).

2. Set the desired username as your new username

```
db.users.update( { username : "<<Your Current Username>>" }, { $set : {
username : "<<Your New Username>>" } } )
```

If this returns an object => WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }), this means that your Username has successfully been updated. If this returns an object => WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 }), an error occurred, verify the current Username you entered is correct or not.

Log Queries:

```
db.logs.insert({ u_id : <<user_id>>, createdAt : new Date(),
attribute : "username", value : "<<Your New Username>>",
activityRelated : false })
```

➔ Update email

1. Check whether the new email you want to set is already in use or not

```
db.users.find( { email : "<<Your New Email>>" }, { email : 1, _id : 0 }
).pretty()
```

If this returns an object => { "email" : "<<Your New Email>>" }, then you can't set this Email as your new Email since it's already been taken by some other account. If this does not return anything, execute (2).

2. Set the desired email as your new email

```
db.users.update( { username : "<<Your Username>>" }, { $set : { email :
"<<Your New Email>>" } } )
```

If this returns an object => WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }), this means that your Email has successfully been updated. If this returns an object => WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 }), an error occurred, verify the Username you entered is correct or not.

Log Queries:

```
db.logs.insert({ u_id : <<user_id>>, createdAt : new Date(),
attribute : "email", value : "<<Your New Email>>",
activityRelated : false })
```

➔ Update Phone Number

1. Check whether the new phone number you want to set is already in use or not

```
db.users.find( { phoneNo : "<<Your New PhoneNo.>>" }, { phoneNo : 1, _id :
0 } ).pretty()
```

If this returns an object => { "phoneNo" : "<<Your New PhoneNo.>>" }, then you can't set this PhoneNo. as your new PhoneNo. since it's already been taken by some other account. If this does not return anything, execute (2).

2. Set the desired phone number as your new phone number

```
db.users.update( { username : "<<Your Username>>" }, { $set : { phoneNo :
"<<Your New PhoneNo.>>" } } )
```

If this returns an object => WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }), this means that your PhoneNo. has successfully been updated. If this returns an object => WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 }), an error occurred, verify the Username you entered is correct or not.

Log Queries:

```
db.logs.insert({ u_id : <<user_id>>, createdAt : new Date(),
attribute : "phoneNo", value : "<<Your New PhoneNo.>>",
activityRelated : false })
```

➔ Update name

1. Set your new name

```
db.users.update( { username : "<<Your Username>>" }, { $set : { fullname :
"<<Your New Name>>" } } )
```

If this returns an object => WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }), this means that your Name has successfully been updated. If this returns an object => WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 }), an error occurred, verify the Username you entered is correct or not.

Log Queries:

```
db.logs.insert({ u_id : <<user_id>>, createdAt : new Date(),
attribute : "fullname", value : "<<Your New Name>>",
activityRelated : false })
```

➔ Update Bio

1. Set your new bio

```
db.users.update( { username : "<<Your Username>>" }, { $set : { bio :
"<<Your New Bio>>" } } )
```

If this returns an object => WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }), this means that your Bio has successfully been updated. If this returns an object => WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 }), an error occurred, verify the Username you entered is correct or not.

Log Queries:

```
db.logs.insert({ u_id : <<user_id>>, createdAt : new Date(),
attribute : "bio", value : "<<Your New Bio>>", activityRelated
: false })
```

➔ Update Website

1. Set your new website

```
db.users.update( { username : "<<Your Username>>" }, { $set : { website :
"<<Your New Website>>" } } )
```

If this returns an object => WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }), this means that your Website has successfully been updated. If this returns an object => WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 }), an error occurred, verify the Username you entered is correct or not.

➔ Update Gender

1. Set your new gender

```
db.users.update( { username : "<<Your Username>>" }, { $set : { gender :
"<<Your New Gender>>" } } )
```

If this returns an object => WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }), this means that your Gender has successfully been updated. If this returns an object => WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 }), an error occurred, verify the Username you entered is correct or not.

➔ Update DOB

1. Set your new DOB

```
db.users.update( { username : "<<Your Username>>" }, { $set : { DOB :
"<<Your New DOB>>" } } )
```

If this returns an object => WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }), this means that your DOB has successfully been updated. If this returns an object => WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 }), an error occurred, verify the Username you entered is correct or not.

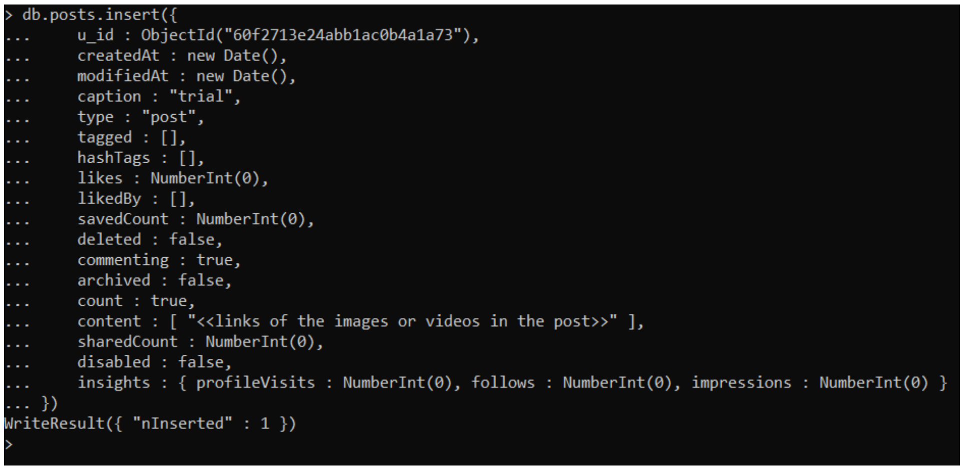
- Create a post

This is a 4 step query. However only one of them can be implemented through shell script. The following shell script written below is for making an entry of this new post in the “posts” Collection. However, the rest of the queries are not possible to be executed in a real scenario, and hence are explained in language along with their shell queries.

1. Create a new entry in “posts” Collection

```
db.posts.insert({
    u_id : <<Post creator/owner's _id>>,
    createdAt : new Date(),
    modifiedAt : new Date(),
    caption : "<<Caption of the post>>",
    type : "<<IGTV/Reel/Post>>",
    tagged : [],
    hashTags : [],
    likes : NumberInt(0),
    likedBy : [],
    savedCount : NumberInt(0),
    deleted : false,
    commenting : true,
    archived : false,
    count : true,
    content : [ "<<links of the images or videos in the post>>" ],
    sharedCount : NumberInt(0),
    disabled : false,
    insights : { profileVisits : NumberInt(0), follows : NumberInt(0),
impressions : NumberInt(0) }
})
```

Example:



```
> db.posts.insert({
...     u_id : ObjectId("60f2713e24abb1ac0b4a1a73"),
...     createdAt : new Date(),
...     modifiedAt : new Date(),
...     caption : "trial",
...     type : "post",
...     tagged : [],
...     hashTags : [],
...     likes : NumberInt(0),
...     likedBy : [],
...     savedCount : NumberInt(0),
...     deleted : false,
...     commenting : true,
...     archived : false,
...     count : true,
...     content : [ "<<links of the images or videos in the post>>" ],
...     sharedCount : NumberInt(0),
...     disabled : false,
...     insights : { profileVisits : NumberInt(0), follows : NumberInt(0), impressions : NumberInt(0) }
... })
WriteResult({ "nInserted" : 1 })
>
```

Now, the queries ahead will require the “_id” of the newly created post. However, since this is a shell query, we won’t be receiving the “_id” of the post even after successful insertion. Hence, in order to get the “_id” of the newly created post, execute the following query.

```
db.posts.find({ u_id : <<user_id>> }, { _id : 1 }).sort({createdAt :
-1}).limit(1).pretty()
```

Now that you’ve got the “_id” of the newly created post, execute the queries below.

2. Enter the newly created post’s “_id” in the corresponding user’s posts[]

```
db.users.update({_id : <<user_id>>}, { $push : { posts : <<post_id>> } })
```

3. For each user in the tagged[] of the newly created post, check whether the user has manuallyApproveTags attribute’s value true or false. If true, add the “_id” of the post to user’s toConfirmTag[] or else to the taggedIn[].

```
db.users.update({_id : <<user_id>>}, { $push : { toConfirmTag : <<post_id>> }
})
```

```
db.users.update({_id : <<user_id>>}, { $push : { taggedIn : <<post_id>> } })
```

Log Queries:

For each user in the tagged[] of the newly created post, execute this log query too.

```
db.logs.insert({ u_id : <<taggedUser_ID>>, createdAt : new Date(),
attribute : "tagged", value : <<post_id>>, activityRelated : true })
```

4. For each hashtag in the hashtags[] of the newly created post, insert posts "_id" in that hashtag's posts[] in "hashTags" collection.

```
db.hashTags.update({ _id : "#hello" }, { $push : { posts : <<post_id>> } },
{upsert : true})
```

- Like/Unlike a post

  ➔ If liked a post

  1. Increment post's like counter as well as insert user's "_id" in post's likedBy[]

```
db.posts.update( { _id : <<post_id>> }, { $push : { likedBy : <<user_id>> },
$inc : { likes : 1 } } )
```

If this returns an object => WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }), this means that you've liked the post successfully. If this returns an object => WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 }), an error occurred, verify the <<post_id>> you entered is correct or not. Execute (2) only if (1) is executed successfully.

  2. Insert the post's "_id" in user's liked[]

```
db.users.update({ _id : <<user_id>> }, { $push : { liked : <<post_id>> } })
```

  ➔ If unliked a post

  1. Decrement post's like counter as well as remove user's "_id" from post's likedBy[]

```
db.posts.update( { _id : <<post_id>> }, { $pull : { likedBy : <<user_id>> },
$inc { likes : -1 } } )
```

If this returns an object => WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }), this means that you've unliked the post successfully. If this returns an object => WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 }), an error occurred, verify the <<post_id>> you entered is correct or not.

2. Remove post's "_id" from user's liked[]

```
db.users.update({ _id : <<user_id>> }, { $pull : { liked : <<post_id>> } })
```

- Like/Unlike a comment

  ➔ If liked a comment

  1. Increment comment's like counter as well as insert user's "_id" in comment's likedBy[]

```
db.comments.update( { _id : <<comment_id>> }, { $push : { likedBy :
<<user_id>> }, $inc : { likes : 1 } } )
```

If this returns an object => WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }), this means that you've liked the comment successfully. If this returns an object => WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 }), an error occurred, verify the <<comment_id>> you entered is correct or not.

  ➔ If unliked a comment

  1. Decrement comment's like counter as well as remove user's "_id" from comment's likedBy[]

```
db.comments.update( { _id : <<comment_id>> }, { $pull : { likedBy :
<<user_id>> }, $inc : { likes : -1} } )
```

If this returns an object => WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }), this means that you've unliked the comment successfully. If this returns an object => WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 }), an error occurred, verify the <<comment_id>> you entered is correct or not.

- Comment on a post

  This is a 3-step query.

  1. Check whether the user is blocked from commenting by the post's owner or not.

```
db.users.find({ _id : <<postOwner_ID>> }, { blockUserFromComment : 1, _id : 0
}).pretty()
```

     If the array returned contains user's "_id", bComment for the comment will be true, else false.

  2. Check whether the user is restricted by the post's owner or not.

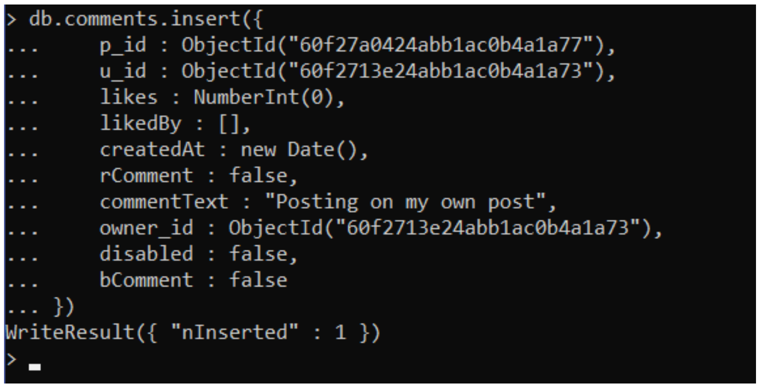
```
db.users.find({ _id : <<postOwner_ID>> }, { restrictAccounts : 1, _id : 0
}).pretty()
```

     If the array returned contains user's "_id", rComment for the comment will be true, else false.

  3. Post a comment

```
db.comments.insert({
    p_id : <<post_id>>,
    u_id : <<user_id>>,
    likes : NumberInt(0),
    likedBy : [],
    createdAt : new Date(),
    rComment : <<depends on 2>>,
    commentText : "<<The commented text>>",
    owner_id : <<PostOwner_ID>>,
    disabled : false,
    bComment : <<depends on 1>>
})
```

     Example:



```
> db.comments.insert({
...     p_id : ObjectId("60f27a0424abb1ac0b4a1a77"),
...     u_id : ObjectId("60f2713e24abb1ac0b4a1a73"),
...     likes : NumberInt(0),
...     likedBy : [],
...     createdAt : new Date(),
...     rComment : false,
...     commentText : "Posting on my own post",
...     owner_id : ObjectId("60f2713e24abb1ac0b4a1a73"),
...     disabled : false,
...     bComment : false
... })
WriteResult({ "nInserted" : 1 })
>
```

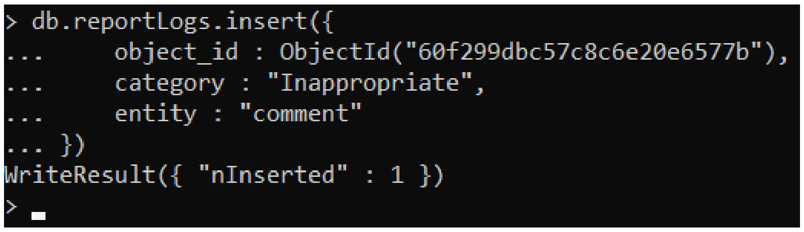
- Report

  This is a single step query.

  ➔ Create new entry in “reportLogs” Collection

```
db.reportLogs.insert({
    object_id : <<_idOfTheEntityReported>>,
    category : "<<Spam/Inappropriate/etc.>>",
    entity : "<<post/profile/story/comment>>"
})
```

Example:



```
> db.reportLogs.insert({
...     object_id : ObjectId("60f299dbc57c8c6e20e6577b"),
...     category : "Inappropriate",
...     entity : "comment"
... })
WriteResult({ "nInserted" : 1 })
>
```

- Block/Unblock a user

  This is a 2 step query, both for blocking as well as unblocking

  ➔ User1 blocks User2

  1. Insert <<user2_id>> in User1’s blockedUsers[] as well as remove <<user2_id>> from User1’s following[] and followers[]

```
db.users.update( { _id : <<user1_id>> }, { $push : { blockedUsers :
<<user2_id>> }, $pull : { following : <<user2_id>>, followers : <<user2_id>> }
} )
```

2. Insert <<user1_id>> in User2's blockedFrom[] as well as remove <<user1_id>> from User2' following[] and followers []

```
db.users.update( { _id : <<user2_id>> }, { $push : { blockedFrom :
<<user1_id>> }, $pull : { following : <<user1_id>>, followers : <<user1_id>> }
} )
```

➔ User1 unblocks User2

1. Remove <<user2_id>> from User1's blockedUsers[]

```
db.users.update( { _id : <<user1_id>> }, { $pull : { blockedUsers :
<<user2_id>> } } )
```

2. Remove <<user1_id>> from User2's blockedFrom[]

```
db.users.update( { _id : <<user2_id>> }, { $pull : { blockedFrom :
<<user1_id>> } } )
```

- Save/Unsave a post

This is a 2 step query, both for saving as well as for unsaving.

➔ If saving

1. Insert <<post_id>> in User's savedPosts[]

```
db.users.update( { _id : <<user_id>> }, { $push : { savedPosts :
<<post_id>> } } )
```

2. Increment "savedCount" of the post

```
db.posts.update({ _id : <<post_id>> }, { $inc : { savedCount : 1 } })
```

➔ If unsaving

1. Remove <<post_id>> from User's savedPosts[]

```
db.users.update( { _id : <<user_id>> }, { $pull : { savedPosts :
```

```
<<post_id>> } } )
```

2. Decrement “savedCount” of the post

```
db.posts.update({ _id : <<post_id>> }, { $inc : { savedCount : -1 } })
```

- Follow/Unfollow a user

  User1 follows/unfollows User2

  ➔ To follow

  1. Check whether User2’s account is public or private.

```
db.users.find( { _id : <<user2_id>> }, { accountType : 1, _id : 0 }
).pretty()
```

If this returns an object => { "accountType" : true }, execute query (2) and (3), else if this returns an object => { "accountType" : false }, execute query (4).

2. Insert << user1_id>> in User2’s followers[]

```
db.users.update( { _id : <<user2_id>> }, { $push : { followers :
<<user1_id>> } } )
```

If this returns an object => WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }), this means that you've added <<user1_id>> in User2's followers[] successfully. If this returns an object => WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 }), an error occurred, verify the <<user2_id>> you entered is correct or not.

3. Insert <<user2_id>> in User1’s following[]

```
db.users.update({ _id : <<user1_id>> }, { $push : { following :
<<user2_id>> } })
```

4. Insert <<user1_id>> in User2’s requests[]

```
db.users.update( { _id : <<user2_id>> }, { $push : { requests :
<<user1_id>> } } )
```

➔ To unfollow

1. Remove <<user1_id>> from User2’s followers[]

```
(i) db.users.update( { _id : <<user2_id>> }, { $pull : { followers :
<<user1_id>> } } )
```

2. Remove <<user2_id>> from User1’s following[]

```
db.users.update({ _id : <<user1_id>> }, { $pull : { following :
<<user2_id>> } })
```

- Accept/Decline a follow request

  User1 accepts/declines User2’s follow request

  ➔ If accepting

  1. Insert <<user1_id>> in User2’s following[]

```
db.users.update( { _id : <<user2_id>> }, { $push : { following :
<<user1_id>> } } )
```

If this returns an object => WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }), this means that you've added <<user1_id>> in User2's following[] successfully. If this returns an object => WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 }), an error occurred, verify the <<user2_id>> you entered is correct or not.

2. Insert <<user2_id>>in User1’s followers[]

```
db.users.update({ _id : <<user1_id>> }, { $push : { followers :
<<user2_id>> } })
```

  ➔ If declining

  1. Remove <<user2_id>> from User1’s requests[]

```
db.users.update({ _id : <<user1_id>> }, { $pull : { requests :
<<user2_id>> } })
```

- Follow/Unfollow a hashtag

  ➔ Follow

```
db.users.update({ _id : <<user_id>> }, { $push : { hashtags :
"<<hashtag>>" } })
```

➔ Unfollow

```
db.users.update({ _id : <<user_id>> }, { $pull : { hashtags : "<<hashtag>>"
} })
```

- Search fullname, username, place and hashtag

  A user can search for a user, place and hashtag.

  ➔ Search for a user via username or fullname

```
db.users.find({ $or : [ { username : "<<searched string>>" }, { fullname :
"<<searched string>>" } ] }, { _id : 1, username : 1, fullname : 1
}).pretty()
```

  ➔ Search for a place

```
db.posts.find({ location : "<<searched string>>" }, { _id : 1, content : 1
}).pretty()
```

  ➔ Search a hashtag

```
db.hashTags.find({ _id : "<<searched string>>" }).pretty()
```

  Also, one needs to store the searched string in the searchHistory[] if a user profile was searched, i.e., a username or fullname.

```
db.users.update({ _id : <<user_id>> }, { $push : { searchHistory :
"<<searched string>>" } })
```

- Archive/Unarchive a post

  ➔ If archiving

  1. Set “archived” of post to true

```
db.posts.update( { _id : <<post_id>> }, { $set : { archived : true } })
```

2. Decrement “postCount” of user

```
db.users.update( { _id : <<owner_id>> }, { $inc { postCount : -1 } })
```

➔ If unarchiving

1. Set “archived” of post to false

```
db.users.update( { _id : <<owner_id>> }, { $inc : { postCount : 1 } })
```

2. Increment “postCount” of user

```
db.posts.update( { _id : <<post_id>> }, { $set : { archived : false } })
```

- Change/Forgot Password

  This query requires logging out the account from all the devices in which it is logged in but it is not something that can be achieved through shell queries. Hence, here is only a query to change your password.

```
db.users.update({ _id : <<user_id>> }, { $set : { password : <<Your New
Password>> } })
```

Log Queries:

```
db.logs.insert({ u_id : <<user_id>>, createdAt : new Date(), attribute :
"password", activityRelated : false })
```

- Turn commenting on a post ON/OFF

  This is for enabling or disabling comment feature on a particular post/

➔ Turn On

1. Set “commenting” of post to true

```
db.posts.update({ _id : <<post_id>> }, { $set : { commenting : true } })
```

➔ Turn Off

1. Set “commenting” of post to false

```
db.posts.update({ _id : <<post_id>> }, { $set : { commenting : false } })
```

- Change Profile Details Visibility

  A user can toggle between public/private account types.

➔ Set to private account

1. Set “accountType” of user to false

```
db.users.update({ _id : <<user_id>> }, { $set : { accountType : false } })
```

Log Queries:

```
db.logs.insert({ u_id : <<user_id>>, createdAt : new Date(),
attribute : "privacy", value : "private", activityRelated :
false })
```

➔ Set to public account

1. Set “accountType” of user to true

```
db.users.update({ _id : <<user_id>> }, { $set : { accountType : true } })
```

Log Queries:

```
db.logs.insert({ u_id : <<user_id>>, createdAt : new Date(),
attribute : "privacy", value : "public", activityRelated :
false })
```

- Delete/Restore a Post/Story

  A user can delete his/her post/story and can also restore it within 30 days.

➔ Post

★ Delete a post

1. Set “deleted” of post to true

```
db.posts.update({ _id : <<post_id>> }, { $set : { deleted :
true } })
```

2. Decrement “postCount” of user

```
db.users.update({ _id : <<user_id>> }, { $inc : { postCount :
-1 } })
```

★ Restore a post

1. Set deleted of post to false

```
db.posts.update({ _id : <<post_id>> }, { $set : { deleted :
false } })
```

2. Increment “postCount” of user

```
db.users.update({ _id : <<user_id>> }, { $inc : { postCount : 1
} })
```

➔ Story

★ Delete a story

1. Set “deleted” of story to true

```
db.stories.update({ _id : <<story_id>> }, { $set : { deleted :
true } })
```

★ Restore a story

```
db.stories.update({ _id : <<story_id>> }, { $set : { deleted :
false } })
```

- View saved posts

  Since these are shell queries, the user won’t in real be able to have look at the posts, instead, will be able to have the “_id”s of the posts he/she have saved and then, can retrieve entire details of those posts.

1. Collect “_id”s of the saved posts

```
db.users.find({ _id : <<user_id>> }, { _id : 0, savedPosts : 1 }).pretty()
```

2. Retrieve all the posts in the savedPosts[] received from above from “posts” Collection

```
db.posts.find({ _id : { $in : [<<Paste body of the array received in previous
query>>] }}).pretty()
```

- View post(s) a particular user or the user himself/herself is/are tagged in

  Since these are shell queries, the user won’t in real be able to have look at the posts, instead, will be able to have the “_id”s of the posts a user is tagged In and then, can retrieve entire details of those posts.

1. Collect “_id”s of the posts the user is tagged in.

```
db.users.find({ _id : <<user_id>> }, { _id : 0, taggedIn : 1 }).pretty()
```

2. Retrieve all the posts in the taggedIn[] received from above from “posts” Collection

```
db.posts.find({ _id : { $in : [<<Paste body of the array received in previous
query>>] }}).pretty()
```

- View posts i’ve liked

  Since these are shell queries, the user won’t in real be able to have look at the posts, instead, will be able to have the “_id”s of the posts he/she has liked and then, can retrieve entire details of those posts.

1. Collect “_id”s of the posts the user has liked

```
db.users.find({ _id : <<user_id>> }, { _id : 0, liked : 1 }).pretty()
```

2. Retrieve all the posts in the liked[] received from above from “posts” Collection

```
db.posts.find({ _id : { $in : [<<Paste body of the array received in previous
query>>] }}).pretty()
```

- View Archived Posts

  Since these are shell queries, the user won’t in real be able to have look at the posts, instead, will be able to have the “_id”s of the posts he/she has archived along with the entire details of those posts.

  1. Collect “_id”s of the posts the user has archived

```
db.posts.find({ u_id : <<user_id>>, archived : true, deleted : false
}).pretty()
```

- View Archived Stories

  Since these are shell queries, the user won’t in real be able to have look at the stories, instead, will be able to have the “_id”s of the stories he/she has archived along with the entire details of those stories.

```
db.stories.find({ u_id : <<user_id>>, archived : true, deleted : false
}).pretty()
```

- Enable/Temporarily Disable Account

  1. Check password before proceeding

```
db.users.find({ username : "<<Your Username>>", password : "<<Your Password>>"
}, { username : 1, password : 1, _id : 0 }).pretty()
```

If this does not return anything, the password entered is incorrect. If an object => { "username" : "<<Your Username>>", "password" : "<<Your Password>>" } is returned, the credentials you entered are correct and you can now proceed to (2).

  2. Set user’s “account” attribute to true/false

➔ If enabling

```
db.users.update({ _id : <<user_id>> }, { $set  : { account : true } })
```

➔ If disabling

```
db.users.update({ _id : <<user_id>> }, { $set  : { account : false }
})
```

3. Mark each post of the user as available/unavailable

   For this, we first need “_id”s of all the posts from posts[] of the user

```
db.users.find({ _id : <<user_id>> }, { _id : 0, posts : 1 }).pretty()
```

   Now, for each “_id” in the posts[], update disabled attribute as false if disabling, else true.

➔ If enabling

```
db.posts.update({ _id : <<post_id from posts[]>> }, { $set : {
disabled : false } })
```

➔ If disabling

```
db.posts.update({ _id : <<post_id from posts[]>> }, { $set : {
disabled : true } })
```

4. Mark each comment of the user as available/unavailable

➔ If enabling

```
db.posts.updateMany({ u_id : <<user_id>> }, { $set : { disabled :
false } })
```

➔ If disabling

```
db.posts.updateMany({ u_id : <<user_id>> }, { $set : { disabled :
```

```
false } })
```

5. Add/Remove <<user_id>> to/from all its followers' following[]

   ➔ If enabling

     This is a 2 step process.

     a. First we'll need the "_id"s of the users who used to follow <<user_id>>. This information can be retrieved from the user_id's followers[].

     ```
     db.users.find({ _id : <<user_id>> }, { _id : 0, followers : 1
     }).pretty()
     ```

     Now, for each "_id" in the followers[] received, execute (b)

     b. Add <<user_id>> to all the users' following[] which are in user_id's followers[]

     ```
     db.users.update({ _id : <<IDfromFollowers[]>> }, { $push : {
     following : <<user_id>> } })
     ```

   ➔ If disabling

     ```
     db.users.updateMany({ following : <<user_id>> }, { $pull : {
     following : <<user_id>> } })
     ```

6. Add/Remove <<user_id>> to/from all its followings' followers[]

   ➔ If enabling

     This is a 2 step process.

     c. First we'll need the "_id"s of the users who the user used to follow. This information can be retrieved from the user_id's following[].

     ```
     db.users.find({ _id : <<user_id>> }, { _id : 0, following : 1
     }).pretty()
     ```

Now, for each "_id" in the following[] received, execute (b)

d. Add <<user_id>> to all the users' followers[] which are in user_id's following[]

```
db.users.update({ _id : <<IDfromFollowing[]>> }, { $push : {
followers : <<user_id>> } })
```

➔ If disabling

```
db.users.updateMany({ followers : <<user_id>> }, { $pull : {
followers : <<user_id>> } })
```

7. Mark stories of the user as available/unavailable

   For this, we first need "_id"s of all the stories from stories[] of the user

```
db.users.find({ _id : <<user_id>> }, { _id : 0, stories : 1
}).pretty()
```

   Now, for each "_id" in the stories[], update disabled attribute as false if disabling, else true.

   ➔ If enabling

```
db.stories.update({ _id : <<story_id from stories[]>> }, { $set : {
disabled : false } })
```

   ➔ If disabling

```
db.stories.update({ _id : <<story_id from stories[]>> }, { $set : {
disabled : true } })
```

8. Mark user disabled in all of the posts he/she is tagged in

   For this, we first need "_id"s of all the posts the user is tagged in from taggedIn[] of the user

```
db.users.find({ _id : <<user_id>> }, { _id : 0, taggedIn : 1
}).pretty()
```

Now, for each "_id" in the taggedIn[], update the status attribute as 'disabled' if disabling, else 'enabled'.

➔ If enabling

```
db.posts.update({ _id : <<post_id from taggedIn[]>> }, { $set : {
"tagged.i.status" : "enabled" } })
```

➔ If disabling

```
db.posts.update({ _id : <<post_id from taggedIn[]>> }, { $set : {
"tagged.i.status" : "disabled" } })
```

- View activity related to my account

```
db.logs.find({ u_id : <<user_id>>, activityRelated : true }, { _id : 0,
createdAt : 1, value : 1, attribute : 1 }).sort({ createdAt : -1 }).pretty()
```

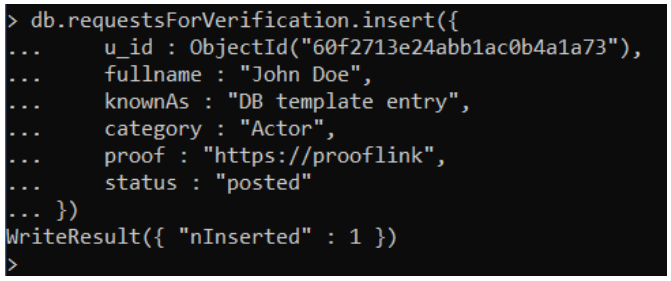
- Login Activity Log

```
db.logs.find({ u_id : <<user_id>>, attribute : "Login" }, { _id : 0, createdAt
: 1, value : 1, attribute : 1 }).sort({ createdAt : -1 }).pretty()
```

- Request Account Verification

```
db.requestsForVerification.insert({
    u_id : <<user_id>>,
    fullname : "<<Your Name>>",
    knownAs : "<<Your Celebrity Name>>",
    category : "<<Musician/Actor/etc.>>",
    proof : "<<link of image>>",
    status : "posted"
})
```

Example:



```
> db.requestsForVerification.insert({
...     u_id : ObjectId("60f2713e24abb1ac0b4a1a73"),
...     fullname : "John Doe",
...     knownAs : "DB template entry",
...     category : "Actor",
...     proof : "https://prooflink",
...     status : "posted"
... })
WriteResult({ "nInserted" : 1 })
>
```

- Switch to Professional Account

  This is a 2 step query, however, executing this completely through shell is not possible.

```
db.users.update({ _id : <<user_id>> }, { $set : { accountType2 : <<new_type>>,
accountType : true } })
```

  After successful execution of this query, perform “Accept Follow Request” query for all the user_id’s in requests[] of the user.

- Story Archive(Location)

```
db.stories.find({ u_id : <<user_id>>, deleted : false, location : { $ne : "" }
}).pretty()
```

- View Entire History of Account

```
db.logs.find({ u_id : <<user_id>>, activityRelated : false }, { _id : 0,
createdAt : 1, value : 1, attribute : 1 }).sort({ createdAt : -1 }).pretty()
```

- Delete Comment

```
db.comments.deleteOne({ _id : <<comment_id>> })
```

- Comments’ features

  There are several features for comments. Queries for each of them are below.

- ➔ User1 blocks/unblocks User2 from commenting
  - ★ Blocks

```
db.users.update({ _id : <<user1_id>> }, { $push : { blockUserFromComment :
<<user2_id>> } })
```

  - ★ Unblocks

```
db.users.update({ _id : <<user1_id>> }, { $pull : { blockUserFromComment :
<<user2_id>> } })
```

- ➔ User enables/disables comment checking
  - ★ Enables

```
db.users.update({ _id : <<user_id>> }, { $set : { checkComment : true } })
```

  - ★ Disables

```
db.users.update({ _id : <<user_id>> }, { $set : { checkComment : false } })
```

- ➔ User chooses who can comment on his/her post

```
db.users.update({ _id : <<user_id>> }, { $set : {  allowCommentsFrom :
"<<Value>>" } })
```

- ➔ Insert/Remove manual filter phrases/words in offensiveWords [ ] of user

```
db.users.update({ _id : <<user_id>> }, { $set : { offensiveWords :
<<[arrayOfWords]>> } })
```

- View and manage Search History

To view search history:

```
db.users.find({ _id : <<user_id>> }, { _id : 0, searchHistory : 1 }).pretty()
```

To remove an entry from search history:

```
db.users.update({ _id : <<user_id>> }, { $pull : { searchHistory : "<<Entry to
be removed>>" } })
```

To clear entire search history:

```
db.users.update({ _id : <<user_id>> }, { $set : { searchHistory : [] } })
```

- View and manage Recently Deleted Posts

  Since these are shell queries, the user won't in real be able to have look at the posts, instead, will be able to have the "_id"s of the posts he/she has deleted recently (<30 days).

```
db.posts.find({ u_id : <<user_id>>, deleted : true }, { _id : 1 }).pretty()
```

- Edit Posts

  ➔ If caption is updated

```
db.posts.update({ _id : <post_id>> }, { $set : { caption : "<<new>>",
modifiedAt : new Date() } })
```

  ➔ If location is updated

```
db.posts.update({ _id : <post_id>> }, { $set : { location : "<<new>>",
modifiedAt : new Date() } })
```

  ➔ If title is updated

```
db.posts.update({ _id : <post_id>> }, { $set : { title : "<<new>>", modifiedAt
: new Date() } })
```

  ➔ If new people are tagged/old tags are removed

```
db.posts.update({ _id : <post_id>> }, { $set : { tagged : "<<new>>" } })
```

- Remove Follower

  User1 removes User2 as his follower

1. Remove <<user2_id>> from user1's followers[]

```
db.users.update({ _id : <<user1_id>> }, { $pull : { followers : <<user2_id>> }
})
```

2. Remove <<user1_id>> from user2's following[]

```
db.users.update({ _id : <<user2_id>> }, { $pull : { following : <<user1_id>> }
})
```

- Hide/Unhide Like and View Counts

  ➔ To hide

```
db.users.update({ _id : <<user_id>> }, { $set : { likeViewCount : false } })
```

  ➔ To unhide

```
db.users.update({ _id : <<user_id>> }, { $set : { likeViewCount : true } })
```

- Hide/Unhide Likes and Views count from a post

  ➔ To hide

```
db.posts.update({ _id : <<post_id>> }, { $set : { count : false } })
```

  ➔ To unhide

```
db.posts.update({ _id : <<post_id>> }, { $set : { count : true } })
```

- Allow tags from

```
db.users.update({ _id : <<user_id>> }, { $set : {  allowTagsFrom : "<<Value>>"
} })
```

- Manually Approve Tags

➔ To approve tags manually

```
db.users.update({ _id : <<user_id>> }, { $set : { manuallyApproveTags : true }
})
```

➔ Let everyone satisfying the criteria of “allowTagsFrom” tag you

```
db.users.update({ _id : <<user_id>> }, { $set : { manuallyApproveTags : false
} })
```

- Hide/Unhide story from particular user(s):

  User1 hides/unhides from User2.

  ➔ Hides

```
db.users.update({ _id : <<user1_id>> }, { $push : { hideStory : <<user2_id>>
} })
```

  ➔ Unhides

```
db.users.update({ _id : <<user1_id>> }, { $pull : { hideStory : <<user2_id>>
} })
```

- Remove Tag

  User removes tag from a post.

  1. Remove <<post_id>> from user’s taggedIn[]

```
db.users.update({ _id : <<user_id>> }, { $pull : { taggedIn : <<post_id>> } })
```

  2. Remove <<user_id>> from post’s tagged[]

```
db.posts.update({ _id : <<post_id>> }, { $pull : { tagged : <<user_id>> } })
```

- Allow @Mention

A check of this attribute should always be performed before mentioning a user anywhere.

1. Update “allowAtMention” of user

```
db.users.update({ _id : <<user_id>> }, { $set : { allowAtMention : "<<Value>>"
} })
```

2. Query to check this attribute before mentioning a user

```
db.users.find({ _id : <<USER_id>> }, { _id : 0, allowAtMention : 1 }).pretty()
```

- Close Friends

  User1 adds/removes User2 to/from his/her closeFriends[]

  ➔ If adds

```
db.users.update({ _id : <<user1_id>> }, { $push : { closeFriends :
<<user2_id>> } })
```

  ➔ If removes

```
db.users.update({ _id : <<user1_id>> }, { $pull : { closeFriends :
<<user2_id>> } })
```

- Allow resharing of posts to stories

  User can himself/herself decide whether he wants other users on the platform to share his/her post on their story or not.

  ➔ To allow

```
db.users.update({ _id : <<user_id>> }, { $set : { allowToShareOnStory : true }
})
```

  ➔ To disallow

```
db.users.update({ _id : <<user_id>> }, { $set : { allowToShareOnStory : false }
})
```

- Manage Notifications

  A user can manage his/her notifications for each user, such as, receive a notification whenever a particular user posts a video/photo/reel etc.

  ➔ Post

    User1 enables/disables User2’s post notifications

    ★ Enable

```
db.users.update({ _id : <<user2_id>> }, { $push : { "manageNotification.posts"
: <<user1_id>> } })
```

    ★ Disable

```
db.users.update({ _id : <<user2_id>> }, { $pull : { "manageNotification.posts"
: <<user1_id>> } })
```

  ➔ Story

    User1 enables/disables User2’s story notifications

    ★ Enable

```
db.users.update({ _id : <<user2_id>> }, { $push : {
"manageNotification.stories" : <<user1_id>> } })
```

    ★ Disable

```
db.users.update({ _id : <<user2_id>> }, { $pull : {
"manageNotification.stories" : <<user1_id>> } })
```

  ➔ IGTV

    User1 enables/disables User2’s IGTV notifications

    ★ Enable

```
db.users.update({ _id : <<user2_id>> }, { $push : { "manageNotification.igtv" :
<<user1_id>> } })
```

    ★ Disable

```
db.users.update({ _id : <<user2_id>> }, { $pull : { "manageNotification.igtv" :
<<user1_id>> } })
```

➔ Reels

User1 enables/disables User2's Reels' notifications

★ Enable

```
db.users.update({ _id : <<user2_id>> }, { $push : { "manageNotification.reels"
: <<user1_id>> } })
```

★ Disable

```
db.users.update({ _id : <<user2_id>> }, { $pull : { "manageNotification.reels"
: <<user1_id>> } })
```

➔ Live

User1 enables/disables User2's Live's notifications. For Live, a user can set 3 different types of settings. A user can either choose to receive none, some or notifications for all the live sessions of a particular user.

★ Enable

```
db.users.update({ _id : <<user2_id>> }, { $push : {
"manageNotification.liveVideos.<<all/some/none>>" : <<user1_id>> } })
```

Choose one from the values in <<>> in the query. Eiter all, some or none.

★ Disable

```
db.users.update({ _id : <<user2_id>> }, { $pull : {
"manageNotification.liveVideos.<<all/some/none>>" : <<user1_id>> } })
```

Choose one from the values in <<>> in the query. Eiter all, some or none according to where the <<user1_id>> is already present.

- Maintain statistics for insights

A user can see various types of statistical information for his/her posts and stories if his/her account type is Professional or Business.

➔ For story

<<value>> => profileVisits/impressions/follows/forward/exitted/next

```
db.stories.update({ _id : <<story_id>> }, { $inc : { "insights.<<value>>" : 1
} })
```

➔ For post

<<value>> => profileVisits/impressions/follows

```
db.posts.update({ _id : <<post_id>> }, { $inc : { "insights.<<value>>" : 1 }
})
```

- Mute/Unmute stories

User1 mutes/unmutes User2's stories

➔ Mute

```
db.users.update({ _id : <<user1_id>> }, { $push : { muteStories : <<user2_id>>
} })
```

➔ Unmute

```
db.users.update({ _id : <<user1_id>> }, { $pull : { muteStories : <<user2_id>>
} })
```

- Mute/Unmute posts and stories

User1 mutes/unmutes User2's posts and stories

➔ Mute

```
db.users.update({ _id : <<user1_id>> }, { $push : { muteStories :
<<user2_id>>, mutePosts : <<user2_id>> } })
```

➔ Unmute

```
db.users.update({ _id : <<user1_id>> }, { $pull : { muteStories :
<<user2_id>>, mutePosts : <<user2_id>> } })
```

- Add/Remove a story to/from a highlight

  ➔ Add

```
db.users.update({ _id : <<user_id>> }, { $push : { "highlight.i.stories" :
<<story_id>> } })
```

  ➔ Remove

```
db.users.update({ _id : <<user_id>> }, { $pull : { "highlight.i.stories" :
<<story_id>> } })
```

- Restrict/Unrestrict an account

  User1 restricts/unrestricts user2's account

  ➔ Restrict

```
db.users.update({ _id : <<user1_id>> }, { $push : { restrictAccounts :
<<user2_id>> } })
```

  ➔ Unrestrict

```
db.users.update({ _id : <<user1_id>> }, { $pull : { restrictAccounts :
<<user2_id>> } })
```

- Polls

  Store response of user on a story's polls feature.

```
db.stories.update({ _id : <<story_id>> }, { $push : { details : { u_id :
<<user_id>>, response : "<<value>>", responseOf : "polls" } } })
```

Log Queries:

```
db.logs.insert({ u_id : <<user_id>>, createdAt : new Date(), attribute :
"polls", value : <<owner_id>>, activityRelated : false })
```

- Questions

  Store response of user on a story's questions feature.

```
db.stories.update({ _id : <<story_id>> }, { $push : { details : { u_id :
<<user_id>>, response : "<<value>>", responseOf : "questions" } } })
```

Log Queries:

```
db.logs.insert({ u_id : <<user_id>>, createdAt : new Date(), attribute :
"questions", value : <<owner_id>>, activityRelated : false })
```

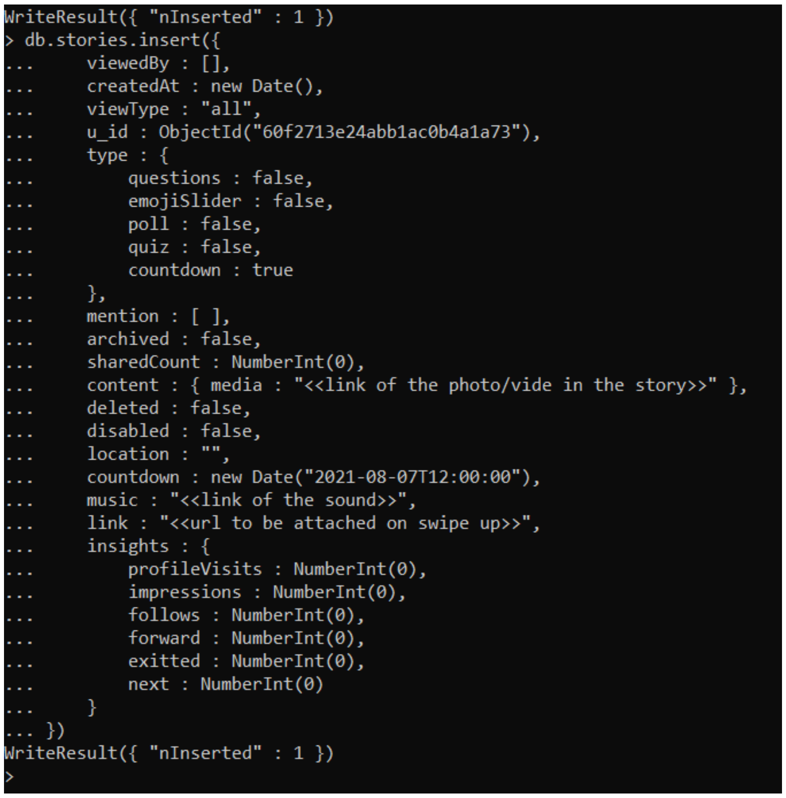
- Post a story

  This is a minimum 3 step query.

  ➔ Create new entry in "stories" Collection

```
db.stories.insert({
    viewedBy : [],
    createdAt : new Date(),
    viewType : "<<closeFriends/followers/all>>",
    u_id : <<user_id>>,
    type : {
        questions : <<value>>,
        emojiSlider : <<value>>,
        poll : <<value>>,
        quiz : <<value>>,
        countdown : <<value>>
    },
    mention : [ <<user_id of mentioned users>> ],
    archived : false,
    sharedCount : NumberInt(0),
    content : { media : "<<link of the photo/vide in the story>>" },
    deleted : false,
    disabled : false,
    location : "",
    countdown : new Date("YYYY-mm-ddTHH:MM:ss"),
    music : "<<link of the sound>>",
    link : "<<url to be attached on swipe up>>",
    insights : {
        profileVisits : NumberInt(0),
        impressions : NumberInt(0),
        follows : NumberInt(0),
        forward : NumberInt(0),
        exitted : NumberInt(0),
        next : NumberInt(0)
    }
})
```

Example:



```
WriteResult({ "nInserted" : 1 })
> db.stories.insert({
...     viewedBy : [],
...     createdAt : new Date(),
...     viewType : "all",
...     u_id : ObjectId("60f2713e24abb1ac0b4a1a73"),
...     type : {
...         questions : false,
...         emojiSlider : false,
...         poll : false,
...         quiz : false,
...         countdown : true
...     },
...     mention : [ ],
...     archived : false,
...     sharedCount : NumberInt(0),
...     content : { media : "<<link of the photo/vide in the story>>" },
...     deleted : false,
...     disabled : false,
...     location : "",
...     countdown : new Date("2021-08-07T12:00:00"),
...     music : "<<link of the sound>>",
...     link : "<<url to be attached on swipe up>>",
...     insights : {
...         profileVisits : NumberInt(0),
...         impressions : NumberInt(0),
...         follows : NumberInt(0),
...         forward : NumberInt(0),
...         exitted : NumberInt(0),
...         next : NumberInt(0)
...     }
... })
WriteResult({ "nInserted" : 1 })
>
```

➔ Insert story's "_id" in user's stories[] and recentStory[]. However, we will not receive the story's "_id" even after successful execution of the above query. Hence, to determine the "_id" of the created story, execute the query below.

```
db.stories.find({ u_id : <<user_id>> }, { _id : 1 }).sort( { createdAt : -1 }
).limit(1).pretty()
```

Now that you have the "_id" of the newly created story, execute the queries below.

```
db.users.update({ _id : <<user_id>> }, { $push : { stories : <<story_id>> } })
db.users.update({ _id : <<user_id>> }, { $push : { recentStory : <<story_id>>
} })
```

➔ For each user in mention[] of the story, create a new entry in the “logs” Collection.

```
db.logs.insert({
    u_id : <<user_id from mention[]>>,
    createdAt : new Date(),
    attribute : "mentioned",
    value : <<story_id>>,
    activityRelated : true
})
```

➔ Also, if a countdown is set in the story, that countdown should also be inserted in the user's countdown[].

```
db.users.update({ _id : <<user_id>> }, { $push : { countdown : <<countdown
date>> } })
```

- Emoji Slider

  Store response of user on a story’s emoji slider feature.

```
db.stories.update({ _id : <<story_id>> }, { $push : { details : { u_id :
<<user_id>>, response : "<<value>>", responseOf : "emojiSlider" } } })
```

Log Queries:

```
db.logs.insert({ u_id : <<user_id>>, createdAt : new Date(), attribute :
"emojiSlider", value : <<owner_id>>, activityRelated : false })
```

- Quiz

  Store response of user on a story’s quiz feature.

```
db.stories.update({ _id : <<story_id>> }, { $push : { details : { u_id :
```

```
<<user_id>>, response : "<<value>>", responseOf : "quiz" } } })
```

Log Queries:

```
db.logs.insert({ u_id : <<user_id>>, createdAt : new Date(), attribute :
"quiz", value : <<owner_id>>, activityRelated : false })
```

- See who and how many saw my story

```
db.stories.find({ _id : <<story_id>> }, { _id : 0, viewedBy : 1 }).pretty()
```

This query will return an array of "_id"s of the users who've seen your story

- Hide/Unhide a post you're tagged in from the tagged posts section on "My Profile page"

  ➔ Hide

```
db.users.update({ _id : <<user_id>> }, { $pull : { taggedIn : <<post_id>> } })
```

  ➔ Unhide

```
db.users.update({ _id : <<user_id>> }, { $push : { taggedIn : <<post_id>> } })
```

- View comment/post liked by

  ➔ Post

```
db.posts.find({ _id : <<user_id>> }, { _id : 0, likedBy : 1 }).pretty()
```

  ➔ Comment

```
db.comments.find({ _id : <<user_id>> }, { _id : 0, likedBy : 1 }).pretty()
```

- View all posts with a hashtag

  ➔ Get "_id"s of all the posts with the hashtag you are looking for.

```
db.hashTags.find({ _id : "<<hashtag>>" }).pretty()
```

- View followers/following of a user

  User1 is viewing User2's followers/following

  ➔ Check whether User2's account is public/private.

```
db.users.find({ _id : <<user2_id>> }, { _id : 0, accountType : 1 }).pretty()
```

If the "accountType" attribute's value is true in the return object, execute (a) else execute (b).

a. Since the account is private, first we need to check whether User1 follows User2 or not. To do so, execute the following query.

```
db.users.find({ _id : <<user2_id>>, followers : <<user1_id>> }, { _id : 1
}).pretty()
```

If this returns an object { _id : <<user2_id>> }, User1 follows User2 and hence you can execute (b). If (a) does not return anything, then it is not appropriate to execute (b).

b. View followers/following

➔ Followers

```
db.users.find({ _id : <<user2_id>> }, { _id : 0, followers : 1 }).pretty()
```

➔ Following

```
db.users.find({ _id : <<user2_id>> }, { _id : 0, following : 1 }).pretty()
```

- Add/remove a post to/from a guide

  ➔ Add

```
db.users.update({ _id : <<user_id>> }, { $push : { "guides.i.posts" : { title :
```

```
"<<?>>", description : "<<?>>", post_id : <<post_id>> } } })
```

➔ Remove

```
db.users.update({ _id : <<user_id>> }, { $pull : { "guides.i.posts" :
<<entry_id>> } })
```

<<entry_id>> here refers to the “_id” of the embedded document of the post that one wants to remove from i’th guide.

- Remove IGTV/Reel from profile grid

```
db.users.update({ _id : <<user_id>> }, { $push : { hideFromProfileGrid :
<<post_id>> } })
```

- Delete Account

1. Set “account” attribute of user to false

```
db.users.update({ _id : <<<user_id>> }, { $set : { account : false } })
```

2. Create a new entry in “deleteAccounts” Collection

```
db.deleteAccounts.insert({ u_id : <<user_id>> })
```