MongoDB

Day1

Nada Mohamed Ahmed Hassan Eleshmawy

Mansoura Open Source

Date : 3/4/2025

2)open mongo shell and view the help.

- Open MongoDB Shell : **Mongosh**

```
C:\Users\nadam>mongosh
Current Mongosh Log ID: 67ee7a20067a136c1ec73bf7
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.3.1
Using MongoDB:          8.0.0
Using Mongosh:          2.3.1
mongosh 2.4.2 is available for download: https://www.mongodb.com/try/download/shell

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

------
   The server generated these startup warnings when booting
   2025-03-25T20:40:27.023+02:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
------

test>
```

- view the help : **db.help**

```
ecommerce> db.help

  Database Class:

    getMongo                                Returns the current database connection
    getName                                 Returns the name of the DB
    getCollectionNames                      Returns an array containing the names of all collections in the current database.
    getCollectionInfos                      Returns an array of documents with collection information, i.e. collection name and options, for the current
database.
    runCommand                              Runs an arbitrary command on the database.
    adminCommand                            Runs an arbitrary command against the admin database.
    aggregate                               Runs a specified admin/diagnostic pipeline which does not require an underlying collection.
    getSiblingDB                            Returns another database without modifying the db variable in the shell environment.
    getCollection                           Returns a collection or a view object that is functionally equivalent to using the db.<collectionName>.
    dropDatabase                            Removes the current database, deleting the associated data files.
    createUser                              Creates a new user for the database on which the method is run. db.createUser() returns a duplicate user erro
r if the user already exists on the database.
    updateUser                              Updates the user's profile on the database on which you run the method. An update to a field completely repla
ces the previous field's values. This includes updates to the user's roles array.
    changeUserPassword                      Updates a user's password. Run the method in the database where the user is defined, i.e. the database you cr
eated the user.
    logout                                  Ends the current authentication session. This function has no effect if the current session is not authentica
ted.
    dropUser                                Removes the user from the current database.
    dropAllUsers                            Removes all users from the current database.
    auth                                    Allows a user to authenticate to the database from within the shell.
    grantRolesToUser                        Grants additional roles to a user.
    revokeRolesFromUser                     Removes a one or more roles from a user on the current database.
    getUser                                 Returns user information for a specified user. Run this method on the user's database. The user must exist on
 the database on which the method runs.
    getUsers                                Returns information for all the users in the database.
    createCollection                        Create new collection
    createEncryptedCollection               Creates a new collection with a list of encrypted fields each with unique and auto-created data encryption ke
ys (DEKs). This is a utility function that internally utilises ClientEnryption.createEncryptedCollection.
    createView                              Create new view
    createRole                              Creates a new role.
    updateRole                              Updates the role's profile on the database on which you run the method. An update to a field completely repla
```

3)identify your current working database and show a list of available databases.

- identify your current working database : **db**

```
test> db
test
```

- show a list of available databases : **show dbs**

```
test> show dbs
admin       40.00 KiB
config      72.00 KiB
depiDB      72.00 KiB
ecommerce   56.00 KiB
local       88.00 KiB
test>
```

4) create a new database called “Facebook” and use it.

use Facebook

Note : MongoDB creates a database only when data is inserted.

```
test> use Facebook
switched to db Facebook
Facebook> show dbs
admin       40.00 KiB
config      72.00 KiB
depiDB      72.00 KiB
ecommerce   56.00 KiB
local       88.00 KiB
Facebook>
```

5)Create Collection with name “posts” which has facebook post properties [“post_text”,”images”,”likes”,”comments”, ”Datetime”,”owner”,”live”]

To create collection we have 2 ways : explicit or implicit

```
db.createCollection("posts", {options});
```

Then insert

```
db.posts.insertOne({
  post_text: "My First Post!",
  images: ["image1.jpg", "image2.png"],
  likes: 110,
  comments: [
    { user: "nada", text: "Hello first" },
    { user: "mohamed", text: "second comment" }
  ],
  Datetime: new Date(),
  owner: "user123",
  live: true
})
```

Or directly

```
db.posts.insertOne({
  post_text: "My First Post!",
  images: ["image1.jpg", "image2.png"],
  likes: 110,
  comments: [
    { user: "Nada", text: "Hello first" },
    { user: "Mohamed", text: "second comment" }
  ],
  Datetime: new Date(),
```

```
  owner: "Nada Mohamed Ahmed",
  live: true
})
```

```
Facebook> db.posts.insertOne({
...   post_text: "My First Post!",
...   images: ["image1.jpg", "image2.png"],
...   likes: 110,
...   comments: [
...     { user: "Nada", text: "Hello first" },
...     { user: "Mohamed", text: "second comment" }
...   ],
...   Datetime: new Date(),
...   owner: "Nada Mohamed Ahmed",
...   live: true
... })
{
  acknowledged: true,
  insertedId: ObjectId('67ee81a0067a136c1ec73bf8')
}
Facebook> show collections
posts
Facebook> db.posts.find();
[
  {
    _id: ObjectId('67ee81a0067a136c1ec73bf8'),
    post_text: 'My First Post!',
    images: [ 'image1.jpg', 'image2.png' ],
    likes: 110,
    comments: [
      { user: 'Nada', text: 'Hello first' },
      { user: 'Mohamed', text: 'second comment' }
    ],
    Datetime: ISODate('2025-04-03T12:40:00.651Z'),
    owner: 'Nada Mohamed Ahmed',
    live: true
  }
]
Facebook>
```

6)Create Capped Collection with name users with Size 5 MB , 10 users Maximum and must has username field “String” and email end with “@gmail.com”.

We will use the options:

- capped : to start making restrictions on size and number. The size and max values are as mentioned in the question.

- validator : will be used to enforce restrictions and define the schema structure(to ensure that the username field is a string, and the email field ends with @gmail.com.)

```
db.createCollection("users", {
  capped: true,
  size: 5 * 1024 * 1024,
  max: 10,
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["username", "email"],
      properties: {
        username: {
          bsonType: "string",
          description: "Username must be a string"
        },
        email: {
          bsonType: "string",
          pattern: "^.+@gmail\\.com$",
          description: "Email must end with @gmail.com"
        }}}});
```

```
Facebook> db.createCollection("users",{capped:true,size:5*1024*1024,max:10,validator:{$jsonSchema:{bsonType:"object",required:["username","email"],propertie
s:{username:{bsonType:"string",description:"username must be a string"},email:{bsonType:"string",pattern:"^.*@gmail\\.com$",description:"email must end with
 @gmail.com"}}}})
{ ok: 1 }
Facebook>
```

7)Insert 20 post “ordered Insert”.

We will use

```
db.posts.insert({data});
```

Or

```
db.posts.insertOne({data});
db.posts.insertMany([{data},{data}]);
```

Here we will use

```
for (let i = 1; i <= 20; i++) {
  db.posts.insertOne({
    post_text: `Post ${i}`,
    images: [`image${i}.jpg`],
    likes: Math.floor(Math.random() * 1000),
    comments: [{ user: `User${i}`, text: `Great post ${i}!` }],
    Datetime: new Date(),
    owner: `user${i}`,
    live: true
  });
}
```

```
Facebook> for (let i = 1; i <= 20; i++) {
...   db.posts.insertOne({
...     post_text: `Post ${i}`,
...     images: [`image${i}.jpg`],
...     likes: Math.floor(Math.random() * 1000),
...     comments: [{ user: `User${i}`, text: `Great post ${i}!` }],
...     Datetime: new Date(),
...     owner: `user${i}`,
...     live: true
...   });
... }
{
  acknowledged: true,
  insertedId: ObjectId('67ee88f3067a136c1ec73c0c')
}
Facebook> db.posts.find();
[
  {
    _id: ObjectId('67ee81a0067a136c1ec73bf8'),
    post_text: 'My First Post!',
    images: [ 'image1.jpg', 'image2.png' ],
    likes: 110,
    comments: [
      { user: 'Nada', text: 'Hello first' },
      { user: 'Mohamed', text: 'second comment' }
    ],
    Datetime: ISODate('2025-04-03T12:40:00.651Z'),
    owner: 'Nada Mohamed Ahmed',
    live: true
  },
  {
    _id: ObjectId('67ee88f3067a136c1ec73bf9'),
    post_text: 'Post 1',
    images: [ 'image1.jpg' ],
    likes: 783,
    comments: [ { user: 'User1', text: 'Great post 1!' } ],
    Datetime: ISODate('2025-04-03T13:11:15.419Z'),
    owner: 'user1',
```

8)Insert 10 users.

```
for (let i = 1; i <= 10; i++) {
  db.users.insertOne({
    username: `user${i}`,
    email: `user${i}@gmail.com`
  });
}
```

```
Facebook> for (let i = 1; i <= 10; i++) {db.users.insertOne({username:`user${i}`,email:`user${i}@gmail.com` });}
{
  acknowledged: true,
  insertedId: ObjectId('67ee897a067a136c1ec73c16')
}
```

9)Display all users.

Using : **db.users.find();**

```
Facebook> db.users.find();
[
  {
    _id: ObjectId('67ee897a067a136c1ec73c0d'),
    username: 'user1',
    email: 'user1@gmail.com'
  },
  {
    _id: ObjectId('67ee897a067a136c1ec73c0e'),
    username: 'user2',
    email: 'user2@gmail.com'
  },
  {
    _id: ObjectId('67ee897a067a136c1ec73c0f'),
    username: 'user3',
    email: 'user3@gmail.com'
  },
  {
    _id: ObjectId('67ee897a067a136c1ec73c10'),
    username: 'user4',
    email: 'user4@gmail.com'
  },
  {
    _id: ObjectId('67ee897a067a136c1ec73c11'),
    username: 'user5',
    email: 'user5@gmail.com'
  },
  {
    _id: ObjectId('67ee897a067a136c1ec73c12'),
    username: 'user6',
    email: 'user6@gmail.com'
  },
  {
    _id: ObjectId('67ee897a067a136c1ec73c13'),
    username: 'user7',
    email: 'user7@gmail.com'
  },
```

10)Display user “Mohamed” posts

db.posts.find({ owner: "Mohamed" })

```
Facebook> db.posts.find({ owner: "Mohamed" })

Facebook>
```

Because I don’t have

For test

Add a post with owner: "Mohamed"

```
db.posts.insertOne({
  post_text: "Mohamed post",
  images: ["image1.jpg", "image2.png"],
  likes: 100,
  comments: [
    { user: "Nada", text: "first" },
    { user: "Mohamed", text: "second" }
  ],
  Datetime: new Date(),
  owner: "Mohamed",
  live: true
})
```

```
Facebook> db.posts.insertOne({
...   post_text: "Mohamed post",
...   images: ["image1.jpg", "image2.png"],
...   likes: 100,
...   comments: [
...     { user: "Nada", text: "first" },
...     { user: "Mohamed", text: "second" }
...   ],
...   Datetime: new Date(),
...   owner: "Mohamed",
...   live: true
... })
{
  acknowledged: true,
  insertedId: ObjectId('67ee8b05067a136c1ec73c17')
}
```

Then run
db.posts.find({ owner: "Mohamed" }) again

```
Facebook> db.posts.find({ owner: "Mohamed" })
[
  {
    _id: ObjectId('67ee8b05067a136c1ec73c17'),
    post_text: 'Mohamed post',
    images: [ 'image1.jpg', 'image2.png' ],
    likes: 100,
    comments: [
      { user: 'Nada', text: 'first' },
      { user: 'Mohamed', text: 'second' }
    ],
    Datetime: ISODate('2025-04-03T13:20:05.766Z'),
    owner: 'Mohamed',
    live: true
  }
]
Facebook>
```

11)Update Mohamed ‘s posts set likes 10000

Posts → updateMany

db.posts.updateMany({ owner: "Mohamed" },{$set:{likes:1000}})

```
Facebook> db.posts.updateMany({ owner: "Mohamed" },{$set:{likes:1000}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
Facebook> db.posts.find({ owner: "Mohamed" })
[
  {
    _id: ObjectId('67ee8b05067a136c1ec73c17'),
    post_text: 'Mohamed post',
    images: [ 'image1.jpg', 'image2.png' ],
    likes: 1000,
    comments: [
      { user: 'Nada', text: 'first' },
      { user: 'Mohamed', text: 'second' }
    ],
    Datetime: ISODate('2025-04-03T13:20:05.766Z'),
    owner: 'Mohamed',
    live: true
  }
]
Facebook>
```

12)delete Mohamed ‘s posts

Run a find() command to check the posts before deletion.
Use deleteMany() to delete those posts.

db.posts.deleteMany({ owner: "Mohamed" });

```
Facebook> db.posts.find({ owner: "Mohamed" })
[
  {
    _id: ObjectId('67ee8b05067a136c1ec73c17'),
    post_text: 'Mohamed post',
    images: [ 'image1.jpg', 'image2.png' ],
    likes: 1000,
    comments: [
      { user: 'Nada', text: 'first' },
      { user: 'Mohamed', text: 'second' }
    ],
    Datetime: ISODate('2025-04-03T13:20:05.766Z'),
    owner: 'Mohamed',
    live: true
  }
]
Facebook> db.posts.deleteMany({ owner: "Mohamed" });
{ acknowledged: true, deletedCount: 1 }
Facebook> db.posts.find({ owner: "Mohamed" })

Facebook>
```

13)What is sharding ?

Sharding is a method used in distributed databases to horizontally partition data across multiple servers or nodes. It helps manage large datasets by breaking the data into smaller chunks (called "shards") and distributing them across multiple servers. Each shard is responsible for a portion of the data, which can improve performance, scalability, and fault tolerance.

In MongoDB, for example, sharding involves:

- **Shard Key**: A field used to determine how data will be distributed across shards.

Sharding is useful when your database grows too large to fit on a single machine or when the number of read/write operations exceeds what a single server can handle.

14)What is replication?

Replication is the process of duplicating data across multiple machines (or nodes) to ensure high availability and data redundancy. In a replicated database system, multiple copies (replicas) of the same data exist on different servers. This allows for fault tolerance, load balancing, and improved read performance.
Replication ensures that if one node fails, another can take over as the primary to continue serving requests.

15)What is a failover mechanism ?

Failover is a process in a system or application that automatically switches to a backup system or component when the primary system fails. This ensures that services remain available and operational even in the event of a failure.

16)What are embedded documents ?

Embedded documents refer to a type of data structure in MongoDB where a document contains another document or an array of documents as a value of a field.

This allows for the storage of related data in a nested, hierarchical format.

For example, in the post task :

If the comments field contains an array of embedded documents. This structure allows for faster access to related data, as it’s stored in a single document.

17)What ACID?

ACID stands for Atomicity, Consistency, Isolation, and Durability, which are four key properties that ensure reliable processing of database transactions:

- Atomicity: A transaction is treated as a single unit of work. Either all of the operations in the transaction are completed, or none of them are.

- Consistency: A transaction will bring the database from one valid state to another, ensuring that all data integrity

constraints are met.

- Isolation: Transactions are isolated from one another. Changes made by one transaction are not visible to others until the transaction is completed.

- Durability: Once a transaction is committed, its changes are permanent and survive system failures.

These properties guarantee that database transactions are processed reliably and ensure data integrity.

18) User Management Methods?

User management refers to the process of creating, modifying, and managing user accounts and permissions in a database or application. It is essential for controlling access to resources and ensuring security. Common user management methods include:

1. Authentication: Verifying the identity of a user based on credentials (e.g., username and password, tokens, biometrics).

2. Authorization: Granting or restricting access to resources based on the user's role or permissions.

3. Roles and Permissions: Assigning specific roles (e.g., admin, read-only, editor) to users, and then granting them permissions to perform actions (e.g., read, write, update)

on specific resources.

4. Password Management: Enforcing password policies (e.g., length, complexity, expiration) and handling password resets.

5. User Groups: Grouping users based on their roles or functions and applying policies to entire groups.

6. Audit Logs: Tracking user actions and interactions with the system for security and compliance purposes.

7. Multi-Factor Authentication (MFA): Enhancing security by requiring multiple forms of authentication, such as a password and a verification code sent to a user's mobile device.

8. Access Control Lists (ACLs): Defining who can access which resources and what actions they can perform.

19)Import Inventory Database using this command in terminal

mongorestore --db Inventory path_to_Inventory_folder.

mongorestore --db Inventory "C:\Users\nadam\Downloads\open source\MongoDB\Day1\lec\inv\inv"

```
C:\Program Files\MongoDB\Tools\100\bin>mongorestore --db Inventory "C:\Users\nadam\Downloads\open source\MongoDB\Day1\le
c\inv\inv"
2025-04-03T16:17:23.390+0200    The --db and --collection flags are deprecated for this use-case; please use --nsInclude
 instead, i.e. with --nsInclude=${DATABASE}.${COLLECTION}
2025-04-03T16:17:23.392+0200    building a list of collections to restore from C:\Users\nadam\Downloads\open source\Mong
oDB\Day1\lec\inv\inv dir
2025-04-03T16:17:23.395+0200    reading metadata for Inventory.orders from C:\Users\nadam\Downloads\open source\MongoDB\
Day1\lec\inv\inv\orders.metadata.json
2025-04-03T16:17:23.395+0200    reading metadata for Inventory.products from C:\Users\nadam\Downloads\open source\MongoD
B\Day1\lec\inv\inv\products.metadata.json
2025-04-03T16:17:23.395+0200    reading metadata for Inventory.users from C:\Users\nadam\Downloads\open source\MongoDB\D
ay1\lec\inv\inv\users.metadata.json
2025-04-03T16:17:23.416+0200    restoring Inventory.products from C:\Users\nadam\Downloads\open source\MongoDB\Day1\lec\
inv\inv\products.bson
2025-04-03T16:17:23.421+0200    restoring Inventory.users from C:\Users\nadam\Downloads\open source\MongoDB\Day1\lec\inv
\inv\users.bson
2025-04-03T16:17:23.427+0200    restoring Inventory.orders from C:\Users\nadam\Downloads\open source\MongoDB\Day1\lec\in
v\inv\orders.bson
2025-04-03T16:17:23.427+0200    finished restoring Inventory.products (11 documents, 0 failures)
2025-04-03T16:17:23.431+0200    finished restoring Inventory.users (2 documents, 0 failures)
2025-04-03T16:17:23.444+0200    finished restoring Inventory.orders (3 documents, 0 failures)
2025-04-03T16:17:23.444+0200    restoring indexes for collection Inventory.products from metadata
2025-04-03T16:17:23.444+0200    index: &idx.IndexDocument{Options:primitive.M{"default_language":"english", "language_ov
erride":"language", "name":"name_text", "textIndexVersion":3, "v":2, "weights":primitive.M{"name":1}}, Key:primitive.D{p
rimitive.E{Key:"_fts", Value:"text"}, primitive.E{Key:"_ftsx", Value:1}}, PartialFilterExpression:primitive.D(nil)}
2025-04-03T16:17:23.444+0200    no indexes to restore for collection Inventory.orders
2025-04-03T16:17:23.445+0200    no indexes to restore for collection Inventory.users
2025-04-03T16:17:23.473+0200    16 document(s) restored successfully. 0 document(s) failed to restore.
```

Make sure we have Inventory Database

Show dbs

Use database

use Inventory

And see collections

Show collections

```
test> use Inventory
switched to db Inventory
Inventory> show dbs
Facebook   112.00 KiB
Inventory  140.00 KiB
admin       40.00 KiB
config     108.00 KiB
depiDB      72.00 KiB
ecommerce   56.00 KiB
local       88.00 KiB
Inventory> show collections
orders
products
users
Inventory>
```

20)Select products with price less than 1000 or greater than 5000.

First see all the collection to know more about it

db.products.find()

```
Inventory> db.products.find()
[
  {
    _id: ObjectId('589ba2fb2742a35b47dad21b'),
    price: 244.20000000000002
  },
  {
    _id: ObjectId('589ba2fb2742a35b47dad21c'),
    name: 'Iphone7',
    price: 16072.47645795,
    category: 'Phone',
    vendor: 'Apple',
    stock: [ 20, 70 ],
    quantity: 10
  },
  {
    _id: ObjectId('589ba2fb2742a35b47dad21d'),
    name: 'Samaung TV',
    price: 11122.1,
    category: 'TV',
    vendor: { name: 'Samaung', phone: '123' },
    stock: [ 5, 70, 80, 34 ],
    quantity: 5
  },
  {
    _id: ObjectId('589ba2fb2742a35b47dad21e'),
    name: 'Toshiba Laptop',
    price: 11122.1,
    category: 'Laptop',
    vendor: { name: 'Toshiba', phone: '011111321' },
```

To Select products with price less than 1000 or greater than 5000

```
db.products.find({
      $or :[
            {price :{$lt:1000} },
            {price :{$gt:5000}}
      ]})
```

```
Inventory> db.products.find({$or :[{price :{$lt:1000} },{price :{$gt:5000}} ]})
[
  {
    _id: ObjectId('589ba2fb2742a35b47dad21b'),
    price: 244.20000000000002
  },
  {
    _id: ObjectId('589ba2fb2742a35b47dad21c'),
    name: 'Iphone7',
    price: 16072.47645795,
    category: 'Phone',
    vendor: 'Apple',
    stock: [ 20, 70 ],
    quantity: 10
  },
  {
    _id: ObjectId('589ba2fb2742a35b47dad21d'),
    name: 'Samaung TV',
    price: 11122.1,
    category: 'TV',
    vendor: { name: 'Samaung', phone: '123' },
    stock: [ 5, 70, 80, 34 ],
    quantity: 5
  },
  {
    _id: ObjectId('589ba2fb2742a35b47dad21e'),
    name: 'Toshiba Laptop',
    price: 11122.1,
    category: 'Laptop',
```

21)Select products where the name field contains at least one element that starts with LG , To ,Sa.

$options: "i" → Case-insensitive search

```
db.products.find({
  name: {
    $regex: "^(LG|To|Sa)",
    $options: "i"
  }
});
```

```
Inventory> db.products.find({name: { $regex: "^(LG|To|Sa)", $options:"i" }});
[
  {
    _id: ObjectId('589ba2fb2742a35b47dad21d'),
    name: 'Samaung TV',
    price: 11122.1,
    category: 'TV',
    vendor: { name: 'Samaung', phone: '123' },
    stock: [ 5, 70, 80, 34 ],
    quantity: 5
  },
  {
    _id: ObjectId('589ba2fb2742a35b47dad21e'),
    name: 'Toshiba Laptop',
    price: 11122.1,
    category: 'Laptop',
    vendor: { name: 'Toshiba', phone: '011111321' },
    stock: [ 55, 67, 23, 1 ],
    quantity: 80
  },
  {
    _id: ObjectId('589ba2fb2742a35b47dad221'),
    name: 'LG TV',
    price: 11122.1,
    category: 'TV',
    vendor: 'LG',
    stock: [ 70 ],
    quantity: 23
```

22)Select products where the stock field value is an array and should contain numbers 99 , 999.

We will use array operator → $all

```
db.products.find({
  stock: { $all: [99, 999] }
});
```

```
Inventory> db.products.find({ stock: { $all: [99, 999] } });
[
  {
    _id: ObjectId('589ba2fb2742a35b47dad224'),
    name: 'Samaung Phone',
    price: 11122.1,
    category: 'Phone',
    vendor: 'Samsumg',
    stock: [ 99, 999 ],
    quantity: 230
  }
]
Inventory>
```

23)Select products with stock field contains value is greater than 99.

We will use array operator → $elemMatch

```
db.products.find({
  stock: { $elemMatch: { $gt: 99 } }
});
```

```
Inventory> db.products.find({ stock: { $elemMatch: { $gt: 99 } } });
[
  {
    _id: ObjectId('589ba2fb2742a35b47dad220'),
    name: 'Laptop Apple',
    price: 44122.476457950004,
    category: 'Laptop',
    vendor: 'Apple',
    stock: [ 300, 350, 600 ],
    quantity: 2
  },
  {
    _id: ObjectId('589ba2fb2742a35b47dad222'),
    name: 'Iphone6',
    price: 11122.1,
    category: 'Phone',
    vendor: 'Apple',
    stock: [ 100, 400 ],
    quantity: 199
  },
  {
    _id: ObjectId('589ba2fb2742a35b47dad224'),
    name: 'Samaung Phone',
    price: 11122.1,
    category: 'Phone',
    vendor: 'Samsumg',
    stock: [ 99, 999 ],
    quantity: 230
  }
```

24)Select products where stock field contains 3 elements.

We will use array operator → $size

```
db.products.find({
  stock: { $size: 3 }
});
```

```
Inventory> db.products.find({ stock: { $size: 3 } });
[
  {
    _id: ObjectId('589ba2fb2742a35b47dad220'),
    name: 'Laptop Apple',
    price: 44122.476457950004,
    category: 'Laptop',
    vendor: 'Apple',
    stock: [ 300, 350, 600 ],
    quantity: 2
  },
  {
    _id: ObjectId('589ba2fb2742a35b47dad223'),
    price: 11122.1,
    category: 'Laptop',
    vendor: 'HP',
    stock: [ 70, 30, 50 ],
    quantity: 31,
    name: 'HP Laptop'
  }
]
Inventory>
```

26)Select products where the vendor is not Apple , Sony, LG or HP.

```
db.products.find({
  vendor: { $nin: ["Apple", "Sony", "LG", "HP"] }
});
```

```
Inventory> db.products.find({ vendor: { $nin: ["Apple", "Sony", "LG", "HP"] } });
[
  {
    _id: ObjectId('589ba2fb2742a35b47dad21b'),
    price: 244.20000000000002
  },
  {
    _id: ObjectId('589ba2fb2742a35b47dad21d'),
    name: 'Samaung TV',
    price: 11122.1,
    category: 'TV',
    vendor: { name: 'Samaung', phone: '123' },
    stock: [ 5, 70, 80, 34 ],
    quantity: 5
  },
  {
    _id: ObjectId('589ba2fb2742a35b47dad21e'),
    name: 'Toshiba Laptop',
    price: 11122.1,
    category: 'Laptop',
    vendor: { name: 'Toshiba', phone: '011111321' },
    stock: [ 55, 67, 23, 1 ],
    quantity: 80
  },
  {
    _id: ObjectId('589ba2fb2742a35b47dad224'),
    name: 'Samaung Phone',
    price: 11122.1,
    category: 'Phone',
    vendor: 'Samsumg',
```

27)Select products where the price field is not exists.

```
db.products.find({
  price: { $exists: 0 }
});
```

```
Inventory> db.products.find({ price: { $exists: 0 } });
[
  {
    _id: ObjectId('58cf8dde542691a9246db715'),
    name: 'Catel',
    vendor: 'OSVendor'
  }
]
Inventory>
```