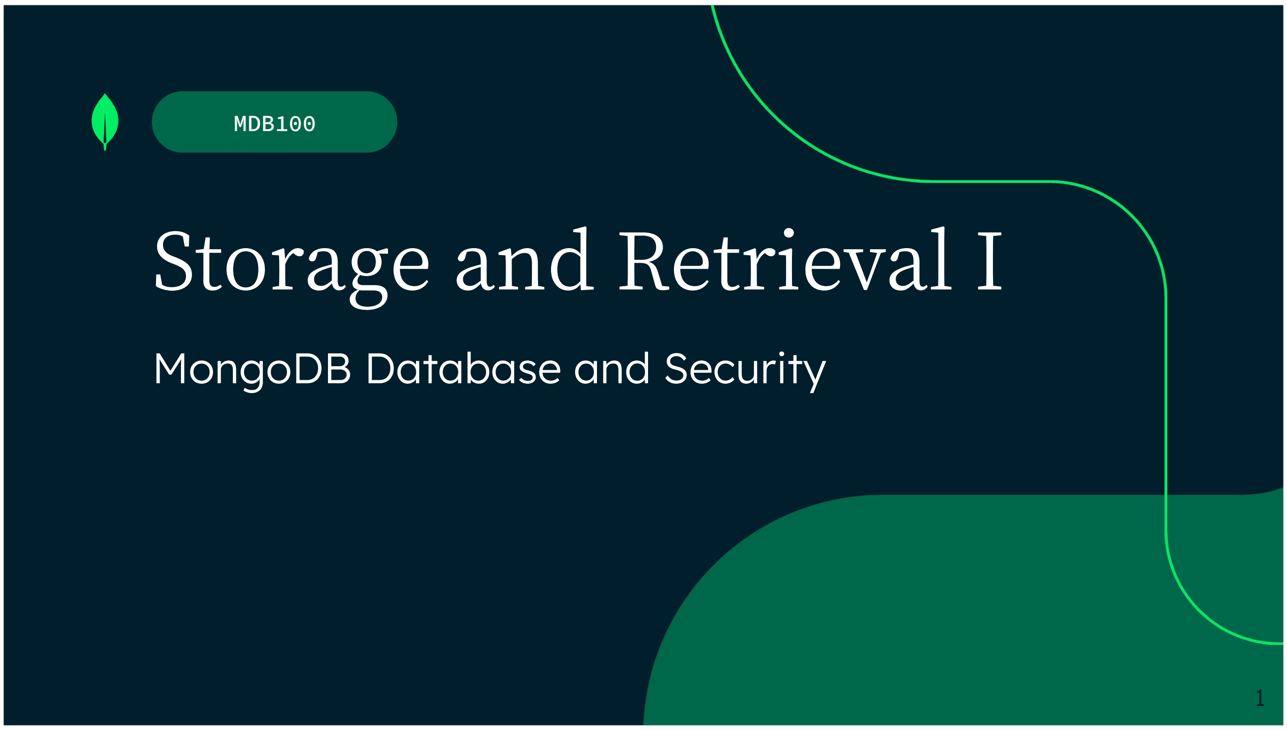

MDB100

Storage and Retrieval I

MongoDB Database and Security

Release: 20250221

Topics we cover

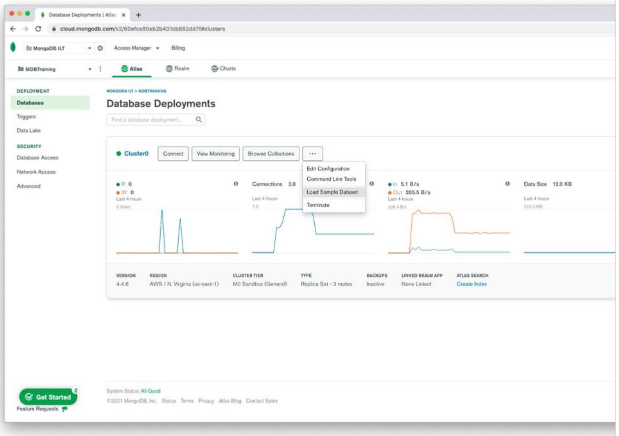
- Creating Documents
- Reading Documents
    - Cursors
- Updating Documents
    - Absolute Changes
    - Relative Changes
    - Conditional Changes
- Deleting Documents

Load Sample Data

In your Atlas cluster,
Click the three dots [...]
Select Load Sample Dataset

Click **Browse Collections** to view the databases and collections we loaded.

We will be using the **sample_training** database.



Follow the instructions to load sample data set in Atlas.
Click on the Collections Button to see a list of databases, out of which we would be using the **sample_training** database.

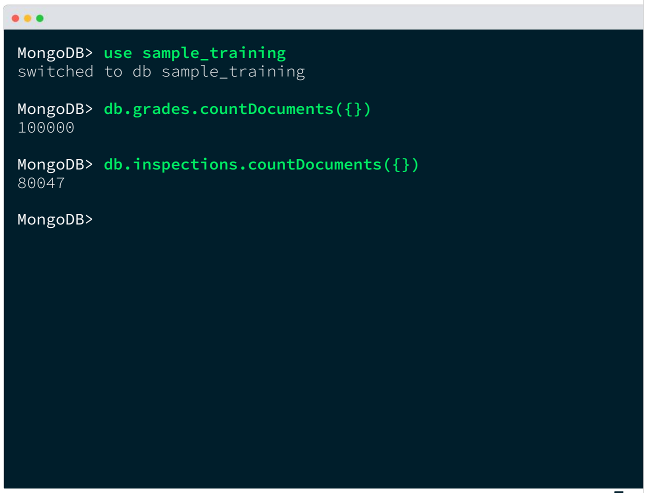
Validate Loaded Data

Connect to the MongoDB Atlas Cluster using mongosh

Verify that data is loaded

Database to use: **sample_training**

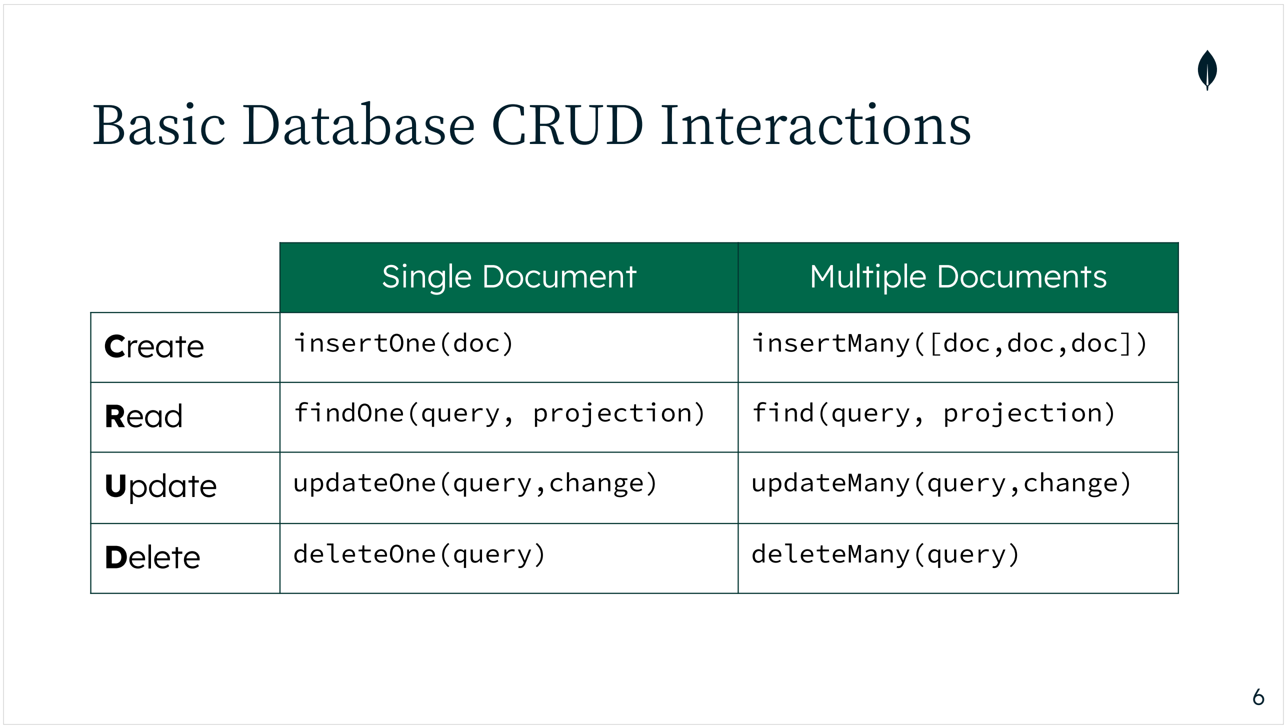
Collections to verify: **grades** and **inspections**



```
MongoDB> use sample_training
switched to db sample_training

MongoDB> db.grades.countDocuments({})
100000

MongoDB> db.inspections.countDocuments({})
80047

MongoDB>
```

Validate the loaded data by checking collection counts for **sample_training.grades** and **sample_training.inspections.**

countDocuments causes the query to return just the number of results found.



Basic Database CRUD Interactions

	Single Document	Multiple Documents
Create	`insertOne(doc)`	`insertMany([doc,doc,doc])`
Read	`findOne(query, projection)`	`find(query, projection)`
Update	`updateOne(query,change)`	`updateMany(query,change)`
Delete	`deleteOne(query)`	`deleteMany(query)`

MongoDB APIs allow us to perform Create Read Update and Delete operations options to perform single or multiple operations.

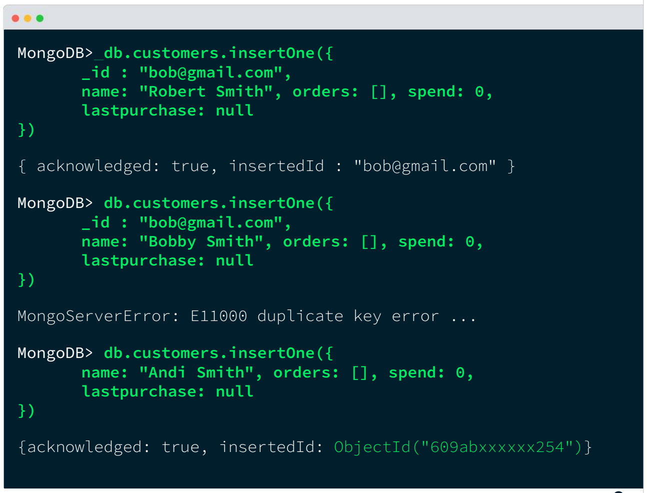
Creating Documents

Creating New Documents - insertOne()

insertOne() adds a document to a collection.

Documents are essentially Objects.

_id field must be unique, it will be added if not supplied.



```
MongoDB> db.customers.insertOne({
        _id : "bob@gmail.com",
        name: "Robert Smith", orders: [], spend: 0,
        lastpurchase: null
})
{ acknowledged: true, insertedId : "bob@gmail.com" }

MongoDB> db.customers.insertOne({
        _id : "bob@gmail.com",
        name: "Bobby Smith", orders: [], spend: 0,
        lastpurchase: null
})
MongoServerError: E11000 duplicate key error ...

MongoDB> db.customers.insertOne({
        name: "Andi Smith", orders: [], spend: 0,
        lastpurchase: null
})
{acknowledged: true, insertedId: ObjectId("609abxxxxxx254")}
```

`insertOne()` adds a document to the collection on which it is called. It is the most basic way to add a new document to a collection.

There are a very few default constraints, the document - which is represented by a language object - **Document, Dictionary, Object must be <16MB**

It must have a **unique value for _id**. If we don't provide one, MongoDB will assign it a GUID of type ObjectId - **a MongoDB GUID type 12 bytes long.**

`{ "acknowledged": true, ... }` means it has succeeded in writing the data to one member of the replica set however we have not specified whether we need it to be on more than one, or even flushed to disk by default.

We can request stronger write guarantees as we will explain later.

Add Multiple Documents - insertMany()

Accepts an array of documents.

Single network call normally.

Reduces network time.

Returns an object with information about each insert.

```
// 1000 Network Calls
MongoDB> let st = ISODate()
for(let d=0;d<1000;d++) {
db.orders.insertOne({ product: "socks", quantity: d})
}
print(`${ISODate()-st} milliseconds`)

9106ms

// 1 Network call, same data
MongoDB> let st = ISODate()
let docs = []
for(let d=0;d<1000;d++) {
       docs.push({ product: "socks", quantity: d})
}
db.orders.insertMany(docs)
print(`${ISODate()-st} milliseconds`)

51ms
```

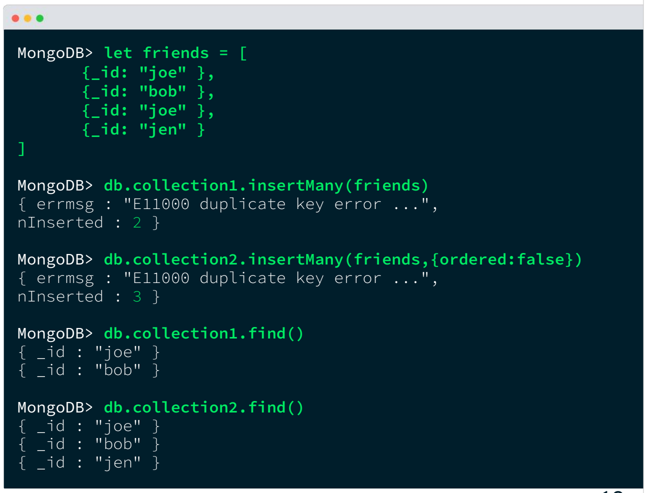
`insertMany()` can add multiple new documents. Often 1000 at a time.
This avoids the need for a network round trip per document, which is really slow
Returns a document showing the success/failure of each and any primary keys assigned
Limit of 48MB or 100,000 documents data in a single call to the server, but a larger batch is broken up behind the scenes by the driver
There is a way to bundle Insert, Update and Delete operations into a single network call too called BulkWrite.

Order of operations in insertMany()

insertMany() can be ordered or unordered.

Ordered (default) stops on first error.

Unordered reports errors but continues; can be reordered by the server to make the operation faster.

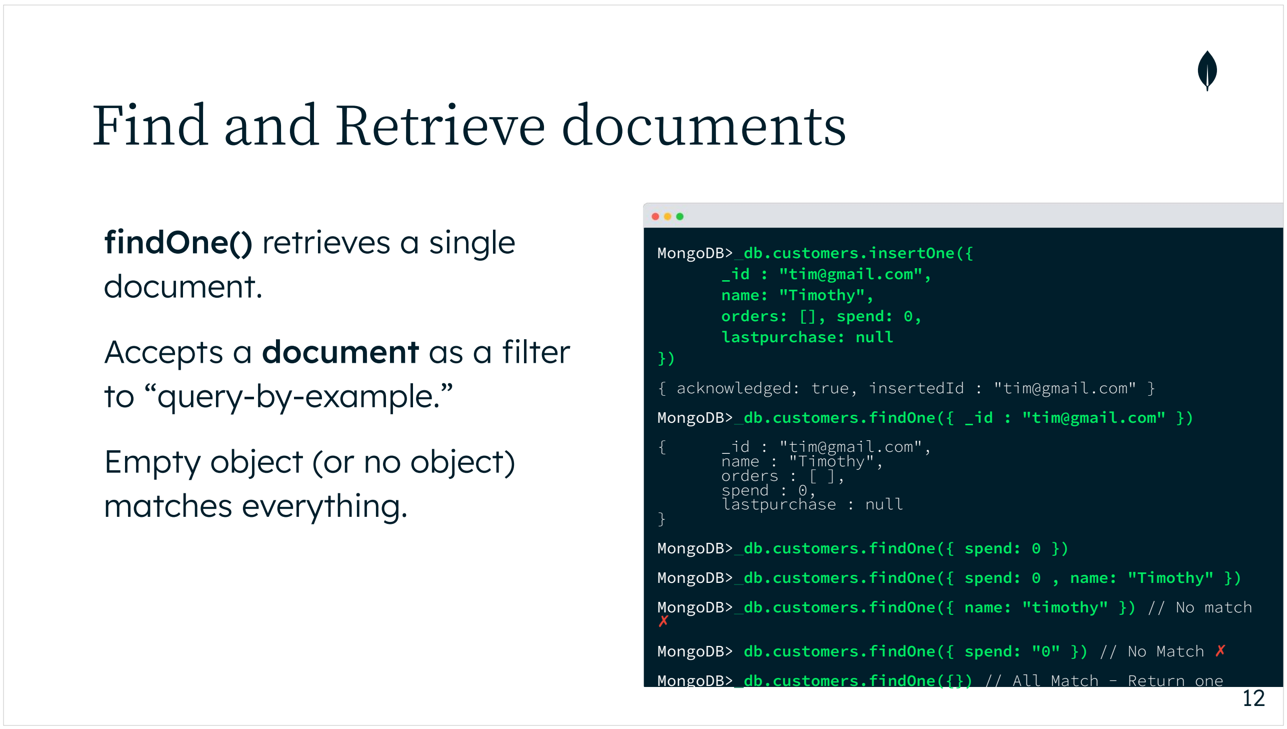


```
MongoDB> let friends = [
       {_id: "joe" },
       {_id: "bob" },
       {_id: "joe" },
       {_id: "jen" }
]

MongoDB> db.collection1.insertMany(friends)
{ errmsg : "E11000 duplicate key error ...",
nInserted : 2 }

MongoDB> db.collection2.insertMany(friends,{ordered:false})
{ errmsg : "E11000 duplicate key error ...",
nInserted : 3 }

MongoDB> db.collection1.find()
{ _id : "joe" }
{ _id : "bob" }

MongoDB> db.collection2.find()
{ _id : "joe" }
{ _id : "bob" }
{ _id : "jen" }
```

If we opt for strict ordering then:

- It must stop on first error
- No reordering or parallelism can be done so slower in sharded cluster.

Reading Documents



Find and Retrieve documents

findOne() retrieves a single document.

Accepts a **document** as a filter to “query-by-example.”

Empty object (or no object) matches everything.

```
MongoDB> db.customers.insertOne({
        _id : "tim@gmail.com",
        name: "Timothy",
        orders: [], spend: 0,
        lastpurchase: null
})
{ acknowledged: true, insertedId : "tim@gmail.com" }
MongoDB> db.customers.findOne({ _id : "tim@gmail.com" })
{       _id : "tim@gmail.com",
        name : "Timothy",
        orders : [ ],
        spend : 0,
        lastpurchase : null
}
MongoDB> db.customers.findOne({ spend: 0 })
MongoDB> db.customers.findOne({ spend: 0 , name: "Timothy" })
MongoDB> db.customers.findOne({ name: "timothy" }) // No match
✗
MongoDB> db.customers.findOne({ spend: "0" }) // No Match ✗
MongoDB> db.customers.findOne({}) // All Match - Return one
```

We can retrieve a document using `findOne().` `findOne()` takes an Object as an argument

We return the first document we find where all the members match. If there are multiple matches there is no way to predict which is 'first' in this case.

Here we add a record for customer Timothy using `insertOne()`

Then we query by the `_id` field - which has the user’s email and we find the record - this returns an object - and mongosh prints what is returned.

We can also query by any other field - although only `_id` has an index by default so the others here are less efficient for now.

We can supply multiple fields, and if they all match we find the record - Someone called Timothy who has spent 0 dollars.

Note that the order of the fields in the query does not matter here - we can think of the comma as just meaning **AND**

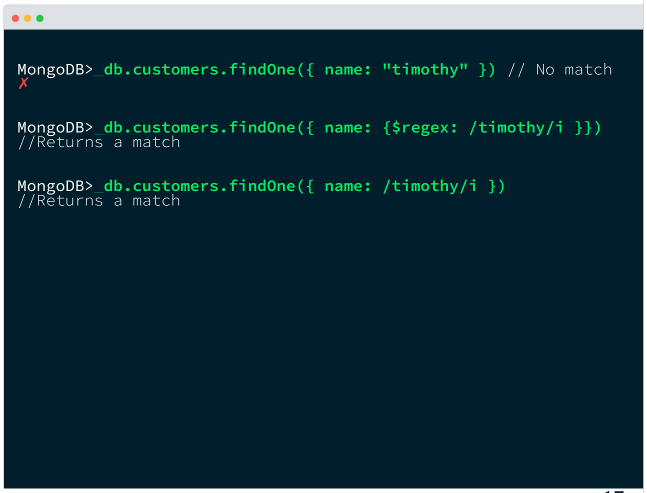
`db.customers.findOne({ spend: "0" })` fails - because it's looking for the String "0" not the number 0 so doesn't match.

An Empty object matches everything. However, due to the inherent nature of findOne() it would return us only one document.

Find and Retrieve documents

Regex can be used to find string values without needing exact matching related to case sensitivity

$regex operator is optional in syntax and can be omitted to get the same result



```
MongoDB> db.customers.findOne({ name: "timothy" }) // No match
✗

MongoDB> db.customers.findOne({ name: {$regex: /timothy/i }})
//Returns a match

MongoDB> db.customers.findOne({ name: /timothy/i })
//Returns a match
```

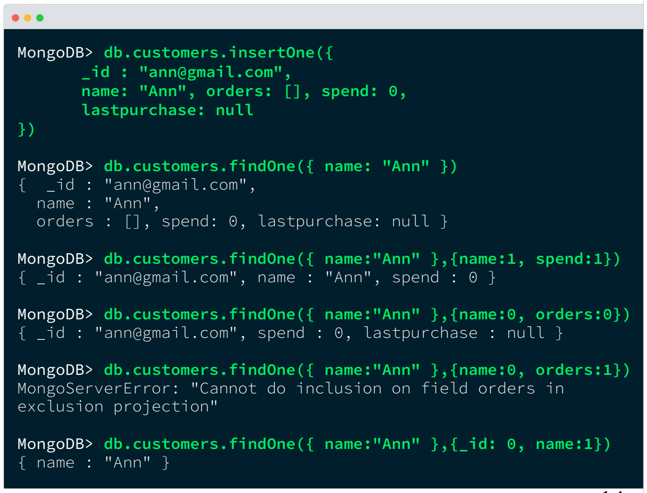
The example is done in javascript regex since mongosh is a js REPL. The regex structure is entirely language dependent based on the driver you are working with.

Projection: choosing the fields to return

Find operations can include a **projection** parameter.

Projections only return a subset of each document.

Projections include/exclude a set of fields.



```
MongoDB> db.customers.insertOne({
       _id : "ann@gmail.com",
       name: "Ann", orders: [], spend: 0,
       lastpurchase: null
})

MongoDB> db.customers.findOne({ name: "Ann" })
{  _id : "ann@gmail.com",
  name : "Ann",
  orders : [], spend: 0, lastpurchase: null }

MongoDB> db.customers.findOne({ name:"Ann" },{name:1, spend:1})
{ _id : "ann@gmail.com", name : "Ann", spend : 0 }

MongoDB> db.customers.findOne({ name:"Ann" },{name:0, orders:0})
{ _id : "ann@gmail.com", spend : 0, lastpurchase : null }

MongoDB> db.customers.findOne({ name:"Ann" },{name:0, orders:1})
MongoServerError: "Cannot do inclusion on field orders in
exclusion projection"

MongoDB> db.customers.findOne({ name:"Ann" },{_id: 0, name:1})
{ name : "Ann" }
```

We can select the fields to return by providing an object with those fields and a value of 1 for each.

Documents can be large; with the help of projection we can have MongoDB return a subset of the fields.

_id is always returned by default.

We can instead choose what field NOT to return by providing an object with fields set to 0.

We cannot mix and match 0 and 1 - as what should it do with any other fields?

There is an exception where we can use **_id: 0** it to remove **_id** from the projection and project only the fields that are required `{ _id:0, name : 1 }`

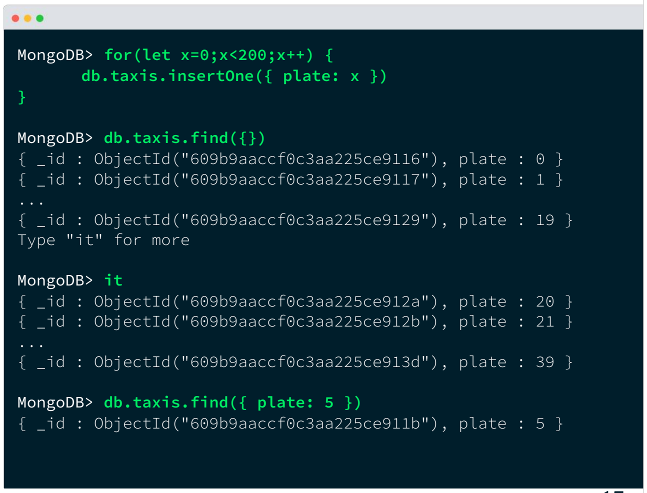
There are some more advanced projection options, including projecting parts of an array and projecting computed fields using aggregation but those are not covered here.

Fetch multiple documents using find()

find() returns a cursor object rather than a single document

We fetch documents from the cursor to get all matches

mongosh fetches and displays 20 documents from the cursor object.



```
MongoDB> for(let x=0;x<200;x++) {
        db.taxis.insertOne({ plate: x })
}

MongoDB> db.taxis.find({})
{ _id : ObjectId("609b9aaccf0c3aa225ce9116"), plate : 0 }
{ _id : ObjectId("609b9aaccf0c3aa225ce9117"), plate : 1 }
...
{ _id : ObjectId("609b9aaccf0c3aa225ce9129"), plate : 19 }
Type "it" for more

MongoDB> it
{ _id : ObjectId("609b9aaccf0c3aa225ce912a"), plate : 20 }
{ _id : ObjectId("609b9aaccf0c3aa225ce912b"), plate : 21 }
...
{ _id : ObjectId("609b9aaccf0c3aa225ce913d"), plate : 39 }

MongoDB> db.taxis.find({ plate: 5 })
{ _id : ObjectId("609b9aaccf0c3aa225ce911b"), plate : 5 }
```

Find returns a cursor object, by default the shell then tries to print that out.

The cursor object prints out by displaying its next 20 documents and setting the value of a variable called it to itself.

If we type `it` - then it tries to print the cursor again - and display the next 20 objects.

As a programmer - cursors won't do anything until we look at them.

We can add `.pretty()` to a cursor object to make the shell display larger documents with newlines and indentation.

Cursors

Using Cursors

Here, we store the result of find to a variable.

We then manually iterate over the cursor.

The query is not actually run until we fetch results from the cursor.

```
MongoDB> let mycursor = db.taxis.find({})

MongoDB> while (mycursor.hasNext()) {
       let doc = mycursor.next();
       printjson(doc)
}
{ _id : ObjectId("609b9aaccf0c3aa225ce9117"), plate : 0 }
{ _id : ObjectId("609b9aaccf0c3aa225ce9118"), plate : 1 }
   ...
{ _id : ObjectId("609b9aaccf0c3aa225ce91dd"), plate : 199 }

MongoDB> let mycursor = db.taxis.find({}) // No Output

MongoDB> mycursor.forEach( doc => { printjson(doc) })

//This does nothing - does not even contact the server!
MongoDB> for(let x=0;x<100;x++) {
       let c = db.taxis.find({})
}
```

`mycursor` is a cursor object, it knows the database, collection and query we want to run.

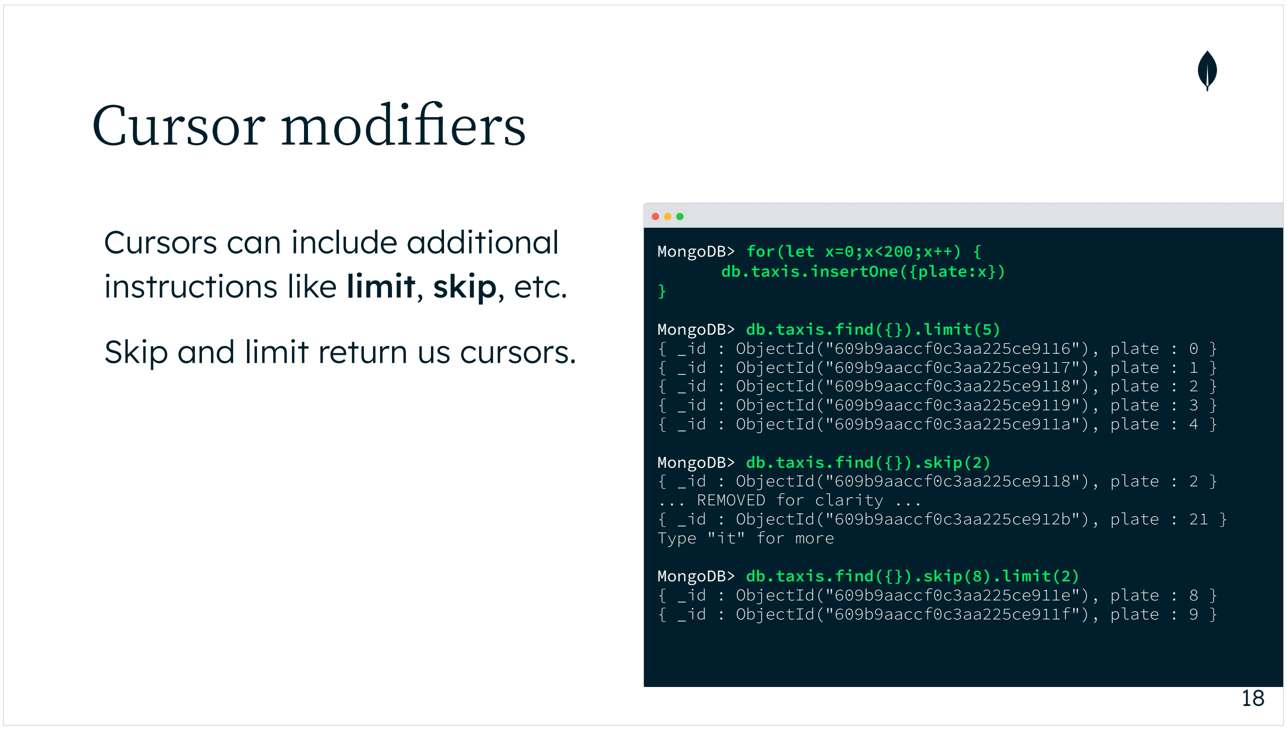
Until we do something with it it has not run the query - it has not even contacted the server.

It has methods - importantly , in mongosh `hasNext()` and `next()` to check for more values and fetch them.

We can iterate over a cursor in various ways depending on our programming language.

If we don't fetch information from a cursor - it never executes the find - this might not be expected when doing simple performance tests like the one below.

To pull the results from a cursor in a shell for testing speed we can use `db.collection.find(query).itcount()`



Cursor modifiers

Cursors can include additional instructions like **limit**, **skip**, etc.

Skip and limit return us cursors.

```
MongoDB> for(let x=0;x<200;x++) {
        db.taxis.insertOne({plate:x})
}

MongoDB> db.taxis.find({}).limit(5)
{ _id : ObjectId("609b9aaccf0c3aa225ce9116"), plate : 0 }
{ _id : ObjectId("609b9aaccf0c3aa225ce9117"), plate : 1 }
{ _id : ObjectId("609b9aaccf0c3aa225ce9118"), plate : 2 }
{ _id : ObjectId("609b9aaccf0c3aa225ce9119"), plate : 3 }
{ _id : ObjectId("609b9aaccf0c3aa225ce911a"), plate : 4 }

MongoDB> db.taxis.find({}).skip(2)
{ _id : ObjectId("609b9aaccf0c3aa225ce9118"), plate : 2 }
... REMOVED for clarity ...
{ _id : ObjectId("609b9aaccf0c3aa225ce912b"), plate : 21 }
Type "it" for more

MongoDB> db.taxis.find({}).skip(8).limit(2)
{ _id : ObjectId("609b9aaccf0c3aa225ce911e"), plate : 8 }
{ _id : ObjectId("609b9aaccf0c3aa225ce911f"), plate : 9 }
```

We can add a limit instruction to the cursor to stop the query when it finds enough results.

We can add a skip instruction to the cursor to tell it to ignore the first N results.

The Skip is always performed before the limit when computing the answer.

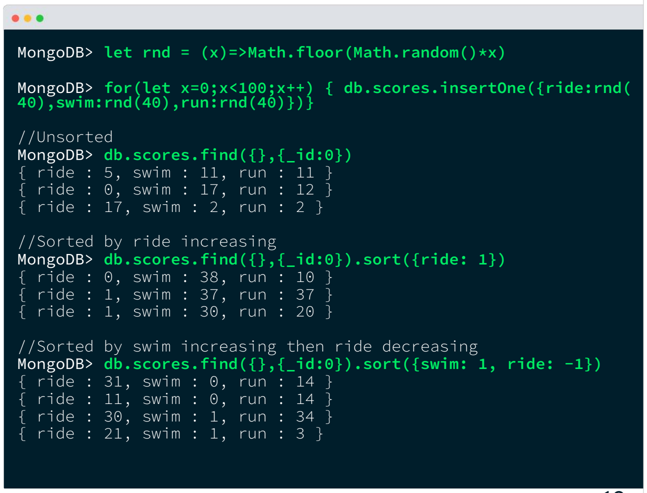
This can be used for simple paging of results - although it's not the optimal way of doing so.

Skip has a cost on the server - skipping a large number of documents is not advisable.

Sorting Results

Use **sort()** cursor modifier to retrieve results in a specific order

Specify an object listing fields in the order to sort and sort direction



```
MongoDB> let rnd = (x)=>Math.floor(Math.random()*x)

MongoDB> for(let x=0;x<100;x++) { db.scores.insertOne({ride:rnd(
40),swim:rnd(40),run:rnd(40)})}

//Unsorted
MongoDB> db.scores.find({},{_id:0})
{ ride : 5, swim : 11, run : 11 }
{ ride : 0, swim : 17, run : 12 }
{ ride : 17, swim : 2, run : 2 }

//Sorted by ride increasing
MongoDB> db.scores.find({},{_id:0}).sort({ride: 1})
{ ride : 0, swim : 38, run : 10 }
{ ride : 1, swim : 37, run : 37 }
{ ride : 1, swim : 30, run : 20 }

//Sorted by swim increasing then ride decreasing
MongoDB> db.scores.find({},{_id:0}).sort({swim: 1, ride: -1})
{ ride : 31, swim : 0, run : 14 }
{ ride : 11, swim : 0, run : 14 }
{ ride : 30, swim : 1, run : 34 }
{ ride : 21, swim : 1, run : 3 }
```

With Skip and Limit sorting can be very important so we skip to limit to what we expect.

We cannot assume anything about the order of unsorted results.

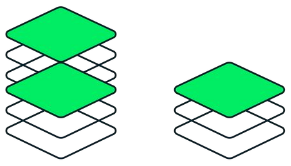
Sorting results without an index is very inefficient - we cover this when talking about indexes later.

Cursors work in batches

Cursors fetch results from the server in batches.

The default batch size in the shell is 101 documents during the initial call to find() with a limit of 16MB.

If we fetch more than the first 100 document from a cursor it fetches in 16MB batches in the shell or up to 48MB in some drivers.



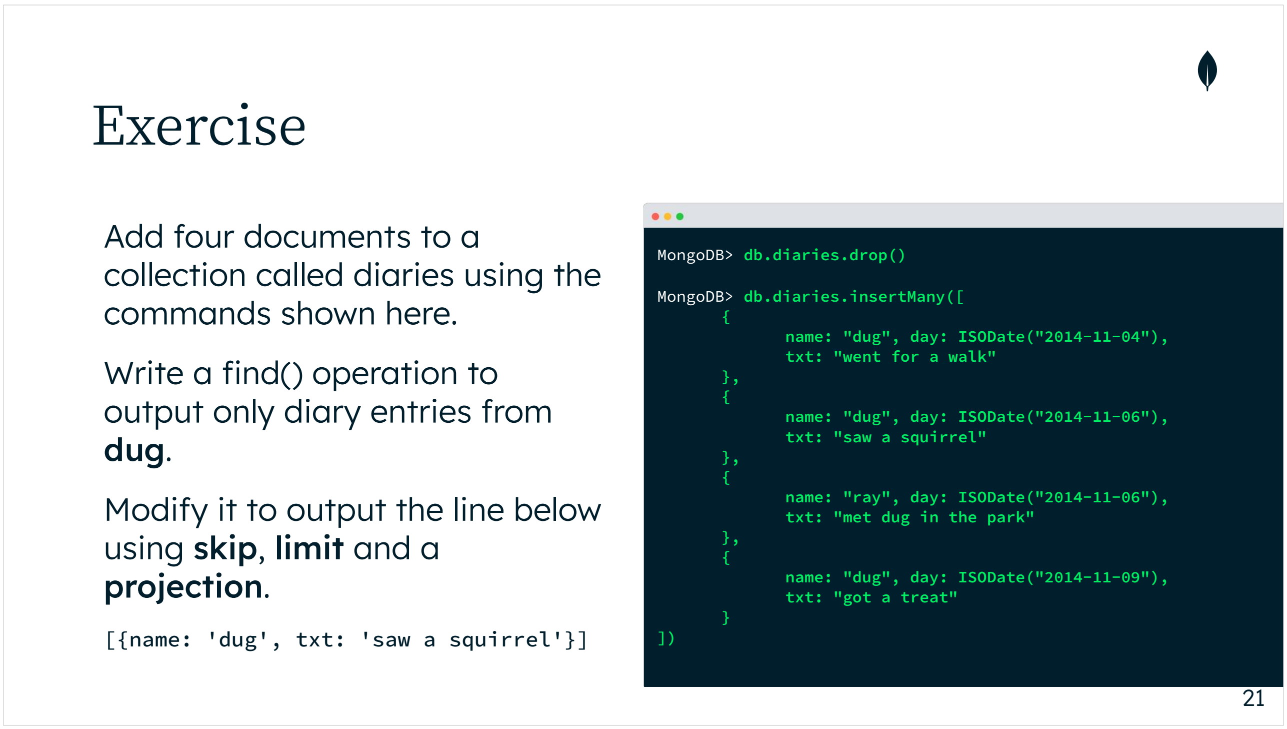
Rather than make a call to the server every time we get the next document from a cursor, the server fetches the result in batches and stores them at the client or shell end until we want them.

Fetching documents one by one would be slow.

Fetching all documents at once would use too much client RAM.

We can change the batch size on the cursor if we need to but it's still limited to 16M.

Fetching additional data from a cursor uses a function called getmore() behind the scenes, it fetches 16MB at a time.



Exercise

Add four documents to a collection called diaries using the commands shown here.

Write a find() operation to output only diary entries from **dug**.

Modify it to output the line below using **skip**, **limit** and a **projection**.

```
[{name: 'dug', txt: 'saw a squirrel'}]
```

```
MongoDB> db.diaries.drop()

MongoDB> db.diaries.insertMany([
        {
                name: "dug", day: ISODate("2014-11-04"),
                txt: "went for a walk"
        },
        {
                name: "dug", day: ISODate("2014-11-06"),
                txt: "saw a squirrel"
        },
        {
                name: "ray", day: ISODate("2014-11-06"),
                txt: "met dug in the park"
        },
        {
                name: "dug", day: ISODate("2014-11-09"),
                txt: "got a treat"
        }
])
```

Answers at the end

Bonus Exercise: Combined Query

Use the **sample_training.companies** collection to find the largest company (by number of employees) that has fewer than 200 employees.

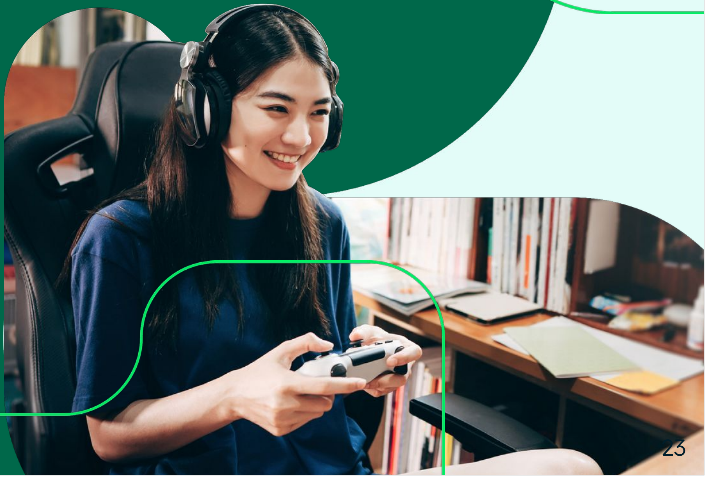
Write a query that prints out only the company name and number of employees.

1. Use find() with a query and a projection
2. Apply some of the cursor operators: sort(), skip() and limit()

Is there an operator you can use to find less than?

Answers at the end

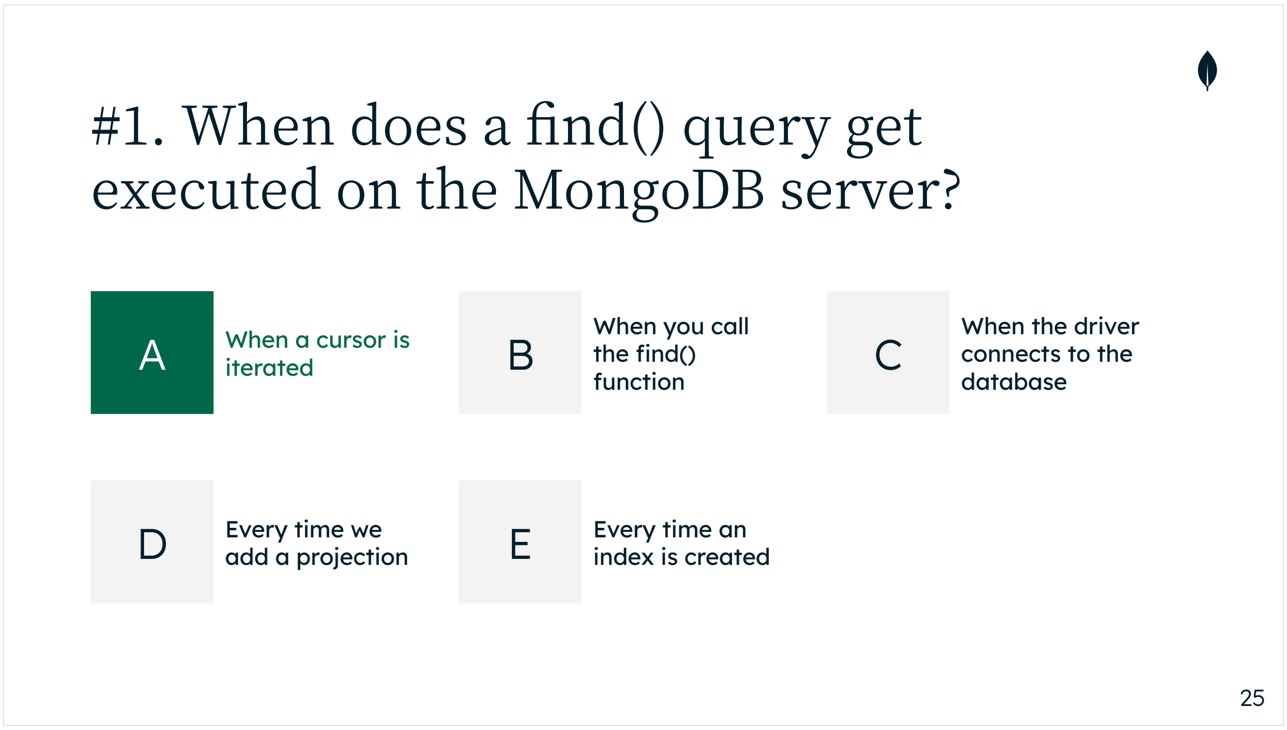
Quiz Time!



#1. When does a find() query get executed on the MongoDB server?

A	When a cursor is iterated	B	When you call the find() function	C	When the driver connects to the database
D	Every time we add a projection	E	Every time an index is created		

Answer in the next slide.



#1. When does a find() query get executed on the MongoDB server?

[✓ A] When a cursor is iterated

B When you call the find() function

C When the driver connects to the database

D Every time we add a projection

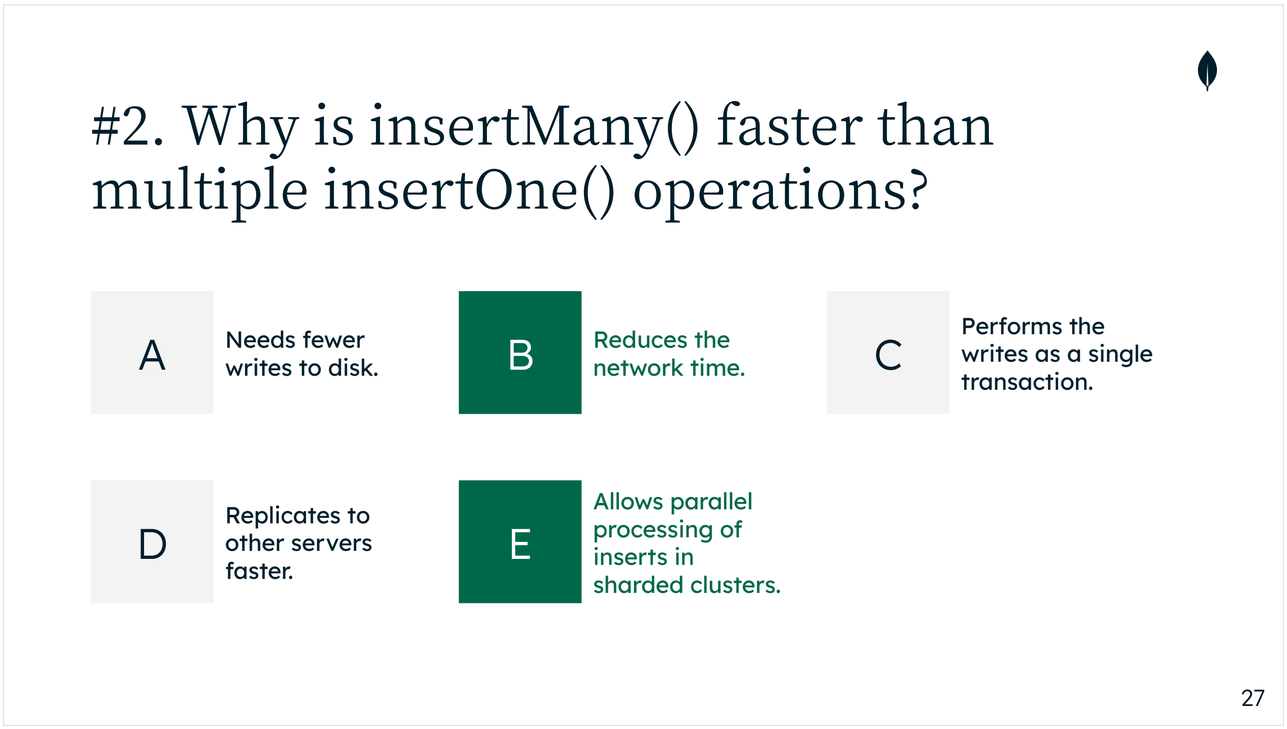
E Every time an index is created

find() returns a cursor object rather than a document/s. The shell starts retrieving the first 20 results but by default find() on its own does not retrieve documents.
Calling find() does not return any values until you start retrieve data with the cursor.
The find() query does not have relationship with a connection pool or the driver connection.
The creation of a cursor, adding a projection, or creating an index do not execute the find query.

#2. Why is insertMany() faster than multiple insertOne() operations?

A	Needs fewer writes to disk.
B	Reduces the network time.
C	Performs the writes as a single transaction.
D	Replicates to other servers faster.
E	Allows parallel processing of inserts in sharded clusters.

Answer in the next slide.



#2. Why is insertMany() faster than multiple insertOne() operations?

A Needs fewer writes to disk.

[✓ B] Reduces the network time.

C Performs the writes as a single transaction.

D Replicates to other servers faster.

[✓ E] Allows parallel processing of inserts in sharded clusters.

InsertMany can load documents in parallel if we had a sharded cluster with the option of (ordered:false). Since it is a bulk operation, it can batch process upto 48MB or 100,000 documents into a single round trip call to the db server, thus reducing the network time needed. It is not a single atomic operation however, as each individual document is inserted atomically.

Recap

Using Bulk writes vs. Single Writes has better network performance

find() returns us a cursor object which the shell then pulls from

Exercise Answers

Answer -Exercise: find, skip and limit

Write a find() to output only diary entries from "dug":

```
MongoDB> db.diaries.find({name:"dug"})

{"_id" : ObjectId("609ba812cf0c3aa225ce91de"), "name" : "dug", "day" : ISODate("2014-
11-04T00:00:00Z"), "txt" : "went for a walk" }

{"_id" : ObjectId("609ba812cf0c3aa225ce91df"), "name" : "dug", "day" : ISODate("2014-
11-06T00:00:00Z"), "txt" : "saw a squirrel" }

{"_id" : ObjectId("609ba812cf0c3aa225ce91e1"), "name" : "dug", "day" : ISODate("2014-
11-09T00:00:00Z"), "txt" : "got a treat" }
```

Modify it to output the line below using skip, limit and a projection:

```
MongoDB> db.diaries.find({name:"dug"},{_id:0,day:0}).skip(1).limit(1)

{ name: "dug", txt: "saw a squirrel" }
```

Answer -Exercise: Combined Query

What company in the companies collection that has fewer than 200 employees has the most employees?

```
MongoDB> db.companies.find(
    {number_of_employees:{$lt:200}},
    {number_of_employees:1, name:1, _id: 0}
).sort({number_of_employees:-1}).limit(1)


{name: 'uShip',number_of_employees: 190}
```