MongoDB shell commands

give all detailed exaplanination mongodb all complete shell commands

Nice — you want a complete, detailed MongoDB **shell commands** reference. Below I've packed a comprehensive, practical cheat-sheet: server start/stop, shell basics, CRUD, queries, aggregation, indexes, updates (incl. arrays), transactions, replica sets, sharding, users/roles, backup/restore, monitoring, and useful admin utilities — with examples you can copy-paste into `mongosh` (or your terminal). I'll call out deprecated/changed items where relevant. #HappyHacking 🐢

Quick notes

- Prefer `mongosh` (the modern interactive shell). Legacy `mongo` still exists on old installs.
- Many admin utilities run from the OS shell (not `mongosh`) — e.g. `mongod`, `mongos`, `mongodump`, `mongorestore`, `mongoexport`, `mongoimport`, `mongostat`, `mongotop`.
- Use `db` in shell to refer to current DB. `use mydb` switches DB.
- Commands shown as `db.collection.method()` are to be run inside `mongosh`. OS-level commands run from your terminal.

Start / stop MongoDB processes

Start `mongod` (single node)

bash

```bash
# basic
mongod --dbpath /data/db

# with port, bind IP, log, config file
mongod --dbpath /data/db --port 27017 --bind_ip 127.0.0.1 --logpath /var/log/mongod.log --fork
```

```
# using config file
mongod --config /etc/mongod.conf
```

Start `mongos` (sharded cluster query router)

```bash
mongos --configdb configReplSet/host1:27019,host2:27019 --port 27017
```

Start shell (mongosh)

```bash
mongosh              # connects to default localhost:27017
mongosh "mongodb://host:27017/?authSource=admin" -u user -p
```

Stop a server (from mongosh, needs clusterAdmin/admin privileges)

```js
// run in mongosh as admin:
db.adminCommand({ shutdown: 1 })
```

Shell basics & helpers

```js
show dbs                // list databases
use mydb                // switch to mydb (creates on first write)
db                   // shows current database
show collections          // list collections in current DB
db.getCollectionNames()     // same but returns array
db.myColl.countDocuments({}) // count documents (preferred)
db.myColl.estimatedDocumentCount() // fast estimate
db.help()                // DB helpers
db.getMongo().getDBs()      // returns DBs as object
it                   // shows more cursor results
```

Inspect a document

```js
db.myColl.findOne()        // show one document
db.myColl.find().pretty()  // readable output
```

Creating / dropping DBs and collections

```js

use newdb                  // switch/create
db.createCollection("logs", {capped:true, size: 1024*1024, max:10000})
db.getCollection("logs").drop()
db.dropDatabase()
```

CRUD — Create (Insert)

```js

// insert one
db.users.insertOne({name:"Alice", age:30})

// insert many
db.users.insertMany([{name:"Bob"}, {name:"Carol"}])

// upsert with update
db.users.updateOne({name:"Eve"}, {$set:{age:29}}, {upsert:true})
```

Read — find / findOne

```js

db.users.find()                        // cursor (all docs)
db.users.find().limit(5).sort({age:1})
db.users.find({age: {$gt: 25}}, {name:1, age:1, _id:0}) // projection
db.users.findOne({name:"Alice"})
db.users.find({}).toArray()            // convert cursor to array
db.users.explain("executionStats").find({age: {$gt:20}}) // explain
```

Useful cursor methods

```js
```

```
db.users.find().skip(10).limit(5).sort({createdAt:-1})
db.users.find().forEach(doc => printjson(doc))
db.users.find().batchSize(100)
```

Query operators (examples)

```js

// Comparison
db.items.find({ price: {$gt: 50, $lte: 100} })

// Equality / not equal
db.items.find({ status: {$ne: "archived"} })

// in / not in
db.items.find({ category: {$in: ["A","B"]} })

// exists / type / regex
db.items.find({ discount: {$exists: true } })
db.items.find({ qty: {$type: "int"} })
db.users.find({ name: {$regex: "^A", $options: "i"} })

// logical
db.users.find({ $and: [{age: {$gte:18}}, {age: {$lte:30}}] })
db.users.find({ $or: [{role:"admin"}, {score: {$gt:1000}}] })
db.users.find({ $nor: [{status:"banned"}] })
```

Projection (select fields)

```js

db.users.find({}, {name:1, email:1, _id:0})  // include name,email; hide _id
// can't mix inclusion and exclusion except for _id
```

Count vs countDocuments vs estimatedDocumentCount

```js
db.collection.countDocuments({status:"active"}) // accurate (scans indexes)
db.collection.estimatedDocumentCount()         // fast estimate (collection metadata)
```

Update operations

```js
// updateOne, updateMany
db.users.updateOne({name:"Alice"}, {$set: {age:31}})
db.users.updateMany({active:false}, {$inc: {score: -10}})

// replace
db.users.replaceOne({name:"Alice"}, {name:"Alice", age:31, roles:[]})

// find-and-modify (return old/new)
db.users.findOneAndUpdate({name:"Bob"}, {$inc:{visits:1}}, {returnDocument:"after"}) // returnDocument:"before" or "after"
```

Common update operators

- `$set` , `$unset` , `$inc` , `$mul` , `$rename` , `$min` , `$max`
- `$currentDate` — set to Date or timestamp
- Array operators: `$push` , `$addToSet` , `$pop` , `$pull` , `$pullAll`
- Example:

```js
// add to array with options
db.posts.updateOne({_id:1}, {$push: {tags: {$each: ["mongodb","db"], $slice:-5, $position:0}}})

// add unique
db.posts.updateOne({_id:1}, {$addToSet: {tags: "mongodb"}})
```

```
// remove element matching condition
db.posts.updateOne({_id:1}, {$pull: {tags: "oldtag"}})
```

Positional operator & arrayFilters

```js
// update first matching array element
db.orders.updateOne(
  { _id: 1, "items.sku": "ABC" },
  { $set: { "items.$.qty": 10 } }
)

// multiple positional filters
db.inventory.updateOne(
  {_id:1},
  {$set: {"items.$[elem].price": 9.99}},
  {arrayFilters: [{"elem.sku":"ABC"}]}
)
```

Bulk operations

```js
const bulk = db.collection.initializeUnorderedBulkOp();
bulk.insert({a:1});
bulk.find({b:1}).updateOne({$set:{c:1}});
bulk.find({d:1}).delete();
bulk.execute();
```

Or use `db.collection.bulkWrite([...])`:

```js
db.users.bulkWrite([
  { insertOne: { document: { name: "X" } } },
  { updateOne: { filter: {name:"Y"}, update: {$set: {age:40}}, upsert:true } }
])
```

Aggregation framework (powerful data processing)

```js
db.orders.aggregate([
 { $match: { status: "A" } },
 { $unwind: "$items" },
 { $group: { _id: "$items.sku", qtySold: { $sum: "$items.qty" }, total: { $sum: { $multiply: ["$items.qty", "$items.price"] } } } },
 { $sort: { qtySold: -1 } },
 { $limit: 10 }
])
```

Useful aggregation stages

- `$match`, `$project`, `$group`, `$sort`, `$limit`, `$skip`
- `$unwind` — deconstruct arrays
- `$lookup` — left outer join with another collection
- `$addFields` / `$set`
- `$replaceRoot` / `$replaceWith`
- `$facet` — multiple pipelines in one
- `$bucket`, `$bucketAuto` — bucketing
- `$sample` — random sample
- `$count` — count docs
- `$merge`, `$out` — write results to collection

Example `$lookup` (join)

```js
db.orders.aggregate([
 { $lookup:
  {
   from: "customers",
   localField: "cust_id",
   foreignField: "_id",
   as: "customer"
  }
 },
 { $unwind: "$customer" },
  { $project: { order_no:1, "customer.name":1 } }
])
```

Indexes

```js
// create single-field index (ascending)
db.users.createIndex({email: 1})

// compound index
db.users.createIndex({ lastName: 1, firstName: 1 })

// unique index
db.users.createIndex({username:1}, {unique:true})

// text index
db.articles.createIndex({title:"text", body:"text"})

// TTL index (expire documents)
db.sessions.createIndex({createdAt:1}, {expireAfterSeconds: 3600})

// hashed index (for sharding on hashed key)
db.users.createIndex({shardKey: "hashed"})

// wildcard index (for dynamic fields)
db.mycoll.createIndex({"$**": 1})

// list indexes
db.users.getIndexes()

// drop
db.users.dropIndex("username_1")
db.users.dropIndexes()  // drops all
```

Index hints & explain

```js
 db.users.find({email:"a@x"}).hint({email:1}).explain("executionStats")
```

Text search

```js
 db.articles.createIndex({content:"text"})
db.articles.find({ $text: { $search: "mongodb aggregation" } }, { score: { $meta: "textScore" }
}).sort({ score: { $meta: "textScore" } })
```

Aggregation explain & optimization tips

```js
 db.collection.aggregate([...]).explain("queryPlanner")
db.collection.aggregate([...]).explain("executionStats")
```

- Push `$match` early in pipeline.
- Use covered queries (projection only on indexed fields).
- Favor indexes for `$group` / `$sort` where possible.

Transactions (multi-document, replica-set or sharded)

```js
// inside mongosh
const session = db.getMongo().startSession();
const sessionDB = session.getDatabase("mydb");
session.startTransaction();

try {
  sessionDB.collection1.insertOne({a:1}, {session});
  sessionDB.collection2.updateOne({_id:2}, {$inc:{qty:1}}, {session});
  session.commitTransaction();
} catch (e) {
  session.abortTransaction();
} finally {
  session.endSession();
}
```

- Transactions require replica set or sharded cluster and proper writeConcern / readConcern settings.

Replica set management (basic)

```js
// on primary mongosh:
rs.initiate()                 // start replica set with default config
rs.initiate({_id:"rs0", members:[{_id:0, host:"localhost:27017"}]})
rs.status()                   // status
rs.conf()                     // current config
rs.add("host2:27017")         // add member
rs.remove("host2:27017")      // remove member
rs.stepDown()                 // step down primary
```

Sharding (high level)

```js
 sh.addShard("rsShard1/host1:27017,host2:27017")
sh.enableSharding("mydb")
sh.shardCollection("mydb.users", {shardKey:1})     // choose appropriate shard key
sh.status()
```

- Sharded clusters require config servers and `mongos` router.

Users, authentication & roles

```js
 use admin
db.createUser({
  user: "dbAdmin",
  pwd: "s3cret",
  roles: [ { role: "userAdminAnyDatabase", db: "admin" }, { role: "readWriteAnyDatabase", db: "admin" } ]
})
```

```
// create DB user with readWrite on specific DB
use mydb
db.createUser({ user: "app", pwd: "pwd", roles: [ {role:"readWrite", db:"mydb"} ] })

// auth
db.auth("user","pwd")
db.getUsers()
db.getRoles({showBuiltinRoles:true})
```

Backup & restore (OS shell utilities)

```bash
# dump entire DB or cluster
mongodump --uri="mongodb://user:pwd@host:27017/mydb" --out /backups/dump-$(date +%F)

# restore (from folder)
mongorestore --uri="mongodb://user:pwd@host:27017" /backups/dump-2025-08-01

# dump to archive compressed
mongodump --uri="mongodb://..." --archive=/backups/backup.archive --gzip

# restore from archive
mongorestore --uri="mongodb://..." --archive=/backups/backup.archive --gzip
```

Export / Import (JSON/CSV)

```bash
# export collection to JSON
mongoexport --uri="mongodb://..." --collection=users --out=users.json --jsonArray

# import JSON
mongoimport --uri="mongodb://..." --collection=users --file=users.json --jsonArray --drop

# import CSV
mongoimport --uri="mongodb://..." --collection=contacts --type=csv --file=contacts.csv --headerline
```

Monitoring & diagnostics (OS shell utilities)

```bash
mongostat --uri="mongodb://..."     # realtime metrics
mongotop --uri="mongodb://..."      # per-collection activity
# from mongosh
db.serverStatus()
db.currentOp()                // show current ops (need privileges)
db.killOp(opid)               // kill operation
```

Profiler & logs

```js
// enable profiling (0:off,1:slow ops,2:all)
db.setProfilingLevel(1, {slowms:100})

// view profiler entries
db.system.profile.find().sort({ts:-1}).limit(20)

// validate a collection
db.myColl.validate({full:true})

// view logs (if accessible)
db.adminCommand({getLog: "global"})
```

Maintenance tasks

```js
// compact collection (requires downtime in MMAPv1; wiredTiger reclaim)
db.runCommand({ compact: "myColl" })

// repair DB (locks DB)
mongod --repair --dbpath /data/db
```

```
// reIndex
db.myColl.reIndex()
```

Useful commands & tips summary

- Connection:

  ```bash
  mongosh "mongodb://user:pwd@host:27017/mydb?authSource=admin"
  ```

- Common admin commands (inside mongosh):

  ```js
  db.serverStatus()
  db.adminCommand({ping:1})
  db.adminCommand({getCmdLineOpts:1})
  db.adminCommand({buildInfo:1})
  ```

- Cursor helpers: `.hasNext()`, `.next()`, `.close()`
- Use `explain()` to profile queries:
  `db.coll.find({...}).explain("executionStats")`
- Use indexes to avoid collection scans. `db.coll.createIndex({field:1})`
- When updating arrays, prefer `$addToSet` to avoid duplicates.
- For massive writes, use `bulkWrite()` or `mongos` configuration for sharding.
- For production: enable authentication (`--auth` or in config), bind to specific IPs, configure TLS, create least-privilege users, enable access control and auditing if needed.

Example complete workflows

1) Create DB, user, and a collection with index

```js
use shop
db.createUser({user:"app", pwd:"apppwd", roles:[{role:"readWrite", db:"shop"}]})
```

```
db.products.insertMany([{sku:"A1",price:10},{sku:"A2",price:20}])
db.products.createIndex({sku:1}, {unique:true})
```

2) Aggregation: daily sales per product

```js
db.orders.aggregate([
 { $match: { createdAt: { $gte: ISODate("2025-08-01"), $lt: ISODate("2025-08-02") } } },
 { $unwind: "$items" },
 { $group: { _id: "$items.sku", unitsSold: { $sum: "$items.qty" }, revenue: { $sum: { $multiply: ["$items.qty","$items.price"] } } } },
 { $sort: { revenue: -1 } }
])
```

3) Simple transaction (replica set)

```js
const s = db.getMongo().startSession();
const sessionDB = s.getDatabase("shop");
s.startTransaction();
try {
 sessionDB.inventory.updateOne({sku:"A1"}, {$inc:{qty:-1}}, {session: s});
 sessionDB.orders.insertOne({customer:"C1", items:[{sku:"A1", qty:1}]}, {session: s});
 s.commitTransaction();
} catch(e) {
 s.abortTransaction();
} finally {
 s.endSession();
}
```

Differences between `mongosh` and older `mongo`

- `mongosh` has modern JS features, improved output, better helpers.
- Some helper commands differ; prefer `mongosh` for new projects.

Final tips & pitfalls

- **Pick your shard key carefully** — it determines distribution and performance.
- **Avoid unbounded** `$regex` queries without anchors or indexes.
- **Use** `countDocuments` if you need accurate counts; `estimatedDocumentCount` for speed.
- **Indexes take space and slow writes**; create them judiciously.
- **Use transactions only when necessary** — they add overhead.
- **Test replica set/sharding configs in staging** before production changes.

If you want, I can:

- Produce a printable **one-page cheat-sheet PDF** with the most-used commands.
- Generate runnable **examples** (scripts) for replica set init, sharding startup, or a sample aggregation tutorial.
- Provide **role-based commands** for admin/readOnly users or help securing a MongoDB instance.

Which of those would be most helpful next?

where is delete commands ?

Ah — you caught me 😅
I gave you every MongoDB shell command under the sun… except **delete**.
Let's fix that — here's the full **delete** section with details, examples, and gotchas.

Delete Commands in MongoDB Shell

1. Delete a Single Document

```js
db.collection.deleteOne(<filter>)
```

- Removes **only the first document** matching the filter.
- Example:

```js
db.users.deleteOne({ name: "Alice" })
```

Deletes one document where `name` is `"Alice"`.

---

2. Delete Multiple Documents

```js
db.collection.deleteMany(<filter>)
```

- Removes **all documents** matching the filter.
- Example:

```js
db.users.deleteMany({ status: "inactive" })
```

Deletes every user with `status` = `"inactive"`.

---

3. Delete All Documents in a Collection

```js
db.collection.deleteMany({})
```

- Equivalent to `TRUNCATE` in SQL — removes **all docs** but keeps the collection.
- Example:

```js
db.logs.deleteMany({})
```

---

4. Drop Entire Collection

```js
db.collection.drop()
```

- Deletes **all data and the collection itself** (metadata gone).
- Example:

js

```
db.orders.drop()
```

---

5. Drop an Entire Database

```js
db.dropDatabase()
```

- Removes **all collections** and the database.
- Example:

```js
use testdb
db.dropDatabase()
```

---

6. findAndModify with Remove

```js
db.collection.findOneAndDelete(<filter>, <options>)
```

- Deletes **one document** and returns the deleted doc.
- Example:

```js
db.users.findOneAndDelete({ age: { $lt: 18 } })
```

---

7. Deprecated

- `db.collection.remove()` — older method, replaced by `deleteOne` / `deleteMany`.
- Example (still works in legacy code):

```js
```

```
db.users.remove({ status: "inactive" })  // all matching
db.users.remove({ name: "Bob" }, true)   // second arg `true` = just one
```

Quick Reference Table

Command	Deletes	Keeps Collection?	Returns Deleted Doc?
`deleteOne(filter)`	First match only	✅	❌
`deleteMany(filter)`	All matches	✅	❌
`deleteMany({})`	All documents	✅	❌
`drop()`	Entire collection	❌	❌
`dropDatabase()`	Entire DB	❌	❌
`findOneAndDelete(filter, options)`	First match	✅	✅
`remove()` *(deprecated)*	One / many (legacy)	✅	❌

If you want, I can **add this delete section back into the big MongoDB cheat-sheet** so it becomes a single, complete reference without missing pieces.

Do you want me to merge it back?

what is difference between update and replace

Alright — here's the **clear difference between** `update` **and** `replace` in MongoDB (including modern equivalents `updateOne` / `updateMany` vs `replaceOne`), with examples so you can see the behavior.

1️⃣ `update` / `updateOne` / `updateMany`

- **Purpose**: Modify **part** of a document.
- **Uses update operators** like `$set`, `$unset`, `$inc`, `$push`, etc.
- Keeps **other fields** in the document unchanged (unless explicitly modified).

Example:

```js
db.users.insertOne({ _id: 1, name: "Alice", age: 25, city: "Delhi" })

db.users.updateOne(
  { _id: 1 },
  { $set: { age: 26 }, $unset: { city: "" } }
)
```

📌 Result:

```json
{ "_id": 1, "name": "Alice", "age": 26 }
```

- Only `age` changed, `city` removed.
- `name` stayed the same.

2️⃣ `replaceOne` (or old `update` without `$set`)

- **Purpose**: Replace the **entire document** with a new one.
- You **must** provide the **full new document** (except `_id`).
- All fields **not mentioned** in the replacement document will be **deleted**.

Example:

```js
db.users.insertOne({ _id: 2, name: "Bob", age: 30, city: "Mumbai" })

db.users.replaceOne(
  { _id: 2 },
  { name: "Bobby", country: "India" }
)
```

📌 **Result:**

```json
{ "_id": 2, "name": "Bobby", "country": "India" }
```

- `age` and `city` are **gone** because the replacement document didn't include them.

3️⃣ Old `update` behavior (pre-3.2)

Before MongoDB 3.2:

```js
db.collection.update(query, replacementOrModifiers, options)
```

- If second argument has **update operators** (`$set`), it acts like `updateOne` / `updateMany` .
- If **no update operators**, it acts like `replaceOne` .

Example:

```js
// Partial update
db.users.update({ _id: 1 }, { $set: { age: 27 } })

// Full replacement
db.users.update({ _id: 1 }, { name: "Alice Updated" })
```

✅ Key Difference Summary

Feature	updateOne / updateMany (with $ops)	replaceOne
Changes part of document	✅	❌ (replaces all)
Keeps other fields	✅	❌

Feature	updateOne / updateMany (with $ops)	replaceOne
Requires update operators	✅ (like `$set`)	❌
Removes unspecified fields	❌	✅
`_id` change allowed?	❌	❌

💡 Rule of thumb

- Use `updateOne` / `updateMany` if you want to **modify only specific fields**.
- Use `replaceOne` if you want to **completely overwrite** the document.

If you want, I can **add `update` vs `replace` with side-by-side examples** into the full MongoDB shell commands guide we made earlier, so your cheat-sheet has it all in one place. Would you like me to do that?