Upgrades & Maintenance

- Version compatibility
- Rolling upgrades
- FCV (Feature Compatibility Version)
- Compacting & cleaning up data
- Lab: Perform mock upgrade steps

---

0. CSC Story / Use Case (Use this narrative in class)

CSC runs a **Compliance & Filings platform** on MongoDB.

- Production cluster: 3-node replica set on **Linux** (on-prem or cloud).
- A legacy **Windows** MongoDB instance still supports an old reporting component.
- Dev/QA engineers in **Mumbai** use **MacBooks** and local MongoDB for testing.
- New projects are moving to **MongoDB Atlas**.

MongoDB 4.4 is nearing end of life.
CSC has decided to standardize on **MongoDB 6.x**.

Your job as a DBA / SRE:

1. Check **version compatibility** (drivers, tools, FCV).
2. Perform a **rolling upgrade** of the production replica set (no downtime).
3. Understand and manage **FCV (Feature Compatibility Version)**.
4. Run **compaction / cleanup** after the upgrade to reclaim space.
5. Practice a **mock upgrade** on Windows, Mac, Linux, and Atlas.

We'll treat this as an **upgrade from 4.4 → 5.0 → 6.0** (conceptually).

You can substitute your actual versions.

---

1. Version Compatibility – What You MUST Check First

1.1 Three Layers of Compatibility

Before touching anything, CSC admins must check:

1. **MongoDB Server version**
2. **Drivers / clients** (Java, .NET, Node.js, Python, etc.)
3. **Tools** (mongodump, mongorestore, mongosh, BI Connector, etc.)

Rule of thumb:

- Drivers are generally compatible with **±1 major server version**, but **always check** the driver compatibility matrix for your actual stack.
- Database tools of version X are typically **forward/backward compatible** with server versions X−1, X, X+1.

💼 CSC use case explanation

CSC's Compliance Portal uses:

- Java Spring microservices
- .NET Core services
- A Node.js UI

Bangalore tech lead checks that:

- Java driver version used is supported on MongoDB 6.0
- mongosh and backup tools are also upgraded on admin servers

This prevents runtime surprises after upgrading the server.

1.2 Check Current Server Version

From **mongosh**:

```
db.version()

db.serverStatus().version
```

Record this for:

- Linux primaries/secondaries
- Windows legacy instance
- Mac local dev instance (if needed)

---

2. FCV – Feature Compatibility Version

FCV controls which **feature set** the cluster uses.
You *must* understand FCV for major upgrades.

2.1 Check FCV

In mongosh:

```
db.adminCommand({ getParameter: 1, featureCompatibilityVersion: 1 })
```

Example result:

```
{
  featureCompatibilityVersion: { version: "4.4" }
}
```

2.2 Key Rules

- Before upgrade **binaries**, FCV is at **old version** (e.g., "4.4").
- After upgrade **binaries** on all nodes and validation, you set FCV to **new version** (e.g., "6.0").
- FCV **must be supported** by all nodes in the cluster.

2.3 Set FCV After Successful Upgrade

```
db.adminCommand({ setFeatureCompatibilityVersion: "6.0" })
```

💼 CSC use case explanation

CSC wants the upgrade to be reversible until they’re confident.

So for some days:

- Binaries run 6.0
- FCV stays at "5.0" or "4.4", allowing easier rollback.

  Once stable, they bump FCV to "6.0".

---

3. Rolling Upgrades – Zero Downtime Approach (Linux / Windows)

3.1 General Flow for a Replica Set

Use this for **Linux production** and **Windows replica members** if any.

For each node:

1. **Confirm it is SECONDARY**, then step it down if needed.
2. **Stop mongod.**
3. **Install new binary** (upgrade MongoDB).
4. **Start mongod.**
5. **Wait for it to rejoin** as SECONDARY and be fully replicated.
6. Repeat for each SECONDARY.
7. Step down PRIMARY and upgrade it last.

3.1.1 Pre-checks (Do this once)

In mongosh on primary:

```
rs.status()
db.adminCommand({ getParameter: 1, featureCompatibilityVersion: 1 })
```

Take a **backup** (logical or snapshot) before starting the rolling upgrade.

3.2 Linux – Node Upgrade Steps (Replica Set Member)

Step 1 – Identify the Node

From primary:

```
rs.status()
```

Note name and state of each member.

Step 2 – Step Down Node (if it's primary)

On primary:

```
rs.stepDown(60)
```

Wait until another node becomes primary.

Step 3 – Stop mongod on Secondary

```
sudo systemctl stop mongod
```

Step 4 – Upgrade MongoDB Binaries

- Add new repo / update package:

```
sudo apt-get update

sudo apt-get install -y mongodb-org=6.0.x
```

(or use yum / dnf depending on distro.)

Step 5 – Start mongod

```
sudo systemctl start mongod
```

Step 6 – Check Status

From mongosh:

```
rs.status()
```

Ensure upgraded node is SECONDARY and replicating fine.

Repeat for all secondaries, then primary.

💼 CSC use case explanation

CSC's main production replica set in Bangalore is 3 nodes (Linux).
Rolling upgrade ensures:

- No downtime for US and Europe clients.
- One node at a time is upgraded.
- If something breaks, they can roll back a single node.

3.3 Windows – Standalone or Replica Node

Step 1 – Stop the Service

```
net stop MongoDB
```

Step 2 – Install New Version

1. Download .msi for new MongoDB version.

2. Run installer → choose "Upgrade" / install to same location.

Step 3 – Start Service

```
net start MongoDB
```

Step 4 – Verify

```
mongosh
db.version()
```

If this Windows instance is a **replica set member**, you apply the same "SECONDARY first, PRIMARY last" pattern.

💼 CSC use case explanation

A legacy compliance report server in Mumbai runs on Windows with its own MongoDB. They stop downtime for 15–30 minutes at night, upgrade the Windows mongod, and validate.

3.4 Mac – Dev / QA Instances

Typically **standalone.**

Step 1 – Stop Service

Homebrew:

```
brew services stop mongodb-community
```

Step 2 – Upgrade

```
brew update
brew upgrade mongodb-community
```

Step 3 – Restart

```
brew services start mongodb-community
```

Step 4 – Verify

```
Mongosh

db.version()
```

💼 CSC use case explanation

Dev/QA in Mumbai upgrade local MongoDB to match production version, so they can reproduce bugs accurately.

3.5 Atlas – Upgrading Cluster Version

On Atlas, you don’t touch binaries manually.

Step 1 – Open Cluster → “Upgrade Version”

1. Login to Atlas.
2. Go to **Clusters** → select cluster (e.g., csc-compliance-prod).
3. Click **“…”** → **Upgrade Version** (or **“Modify”** depending on UI).
4. Choose the target MongoDB version (e.g., 6.0).
5. Confirm.

Atlas will perform a **rolling upgrade** automatically:

- Upgrades secondaries first
- Steps down and upgrades primary last
- Minimal downtime

Step 2 – Check FCV afterwards (with mongosh)

```
db.adminCommand({ getParameter: 1, featureCompatibilityVersion: 1 })
```

If all stable:

```
db.adminCommand({ setFeatureCompatibilityVersion: "6.0" })
```

💼 CSC use case explanation

CSC has begun moving high-priority applications to Atlas.
Upgrades are done via the UI with automated, safe rolling upgrades and built-in rollback support.

---

4. Compacting & Cleaning Up Data

After upgrades, or after large deletions / archival, CSC might want to **reclaim disk space** and ensure indexes are healthy.

4.1 Self-Managed (Linux / Windows / Mac)

4.1.1 Compact a Collection (WiredTiger)

In mongosh:

```
use csc_compliance

db.runCommand({ compact: "filings" })
```

⚠️ This can be heavy; do it in a maintenance window, especially on big collections.

4.1.2 repairDatabase

for Full Cleanup (Careful!)

```
use csc_compliance

db.repairDatabase()
```

- Requires extra disk space.
- Locks the DB; use only when necessary.

4.1.3 TTL Indexes & Archival

For auto-cleaning old reminders:

```
db.reminders.createIndex(
  { expiresAt: 1 },
  { expireAfterSeconds: 0 }
)
```

- Old documents get removed automatically.
- Space is reclaimed over time by WiredTiger.

💼 **CSC use case explanation**

After tax season, CSC marks old filings as archived and moves them to another collection.
Then they **compact** hot collections and rely on TTL for old data like reminders.

4.2 Atlas – Compaction & Cleanup

In Atlas:

- WiredTiger handles compaction automatically at lower level.
- You can still:
    - Use compact on specific collections (with care).
    - Use TTL indexes for auto-expiry.
    - Use Atlas online archive or separate cluster for cold data.

5. LAB – Mock Upgrade & Maintenance (Windows, Mac, Linux, Atlas)

Use this as a **90–120 minute hands-on** module.

LAB Part A – Pre-Upgrade Checks (All Platforms)

1. **Check version**

```
db.version()
```

1.
2. **Check FCV**

```
db.adminCommand({ getParameter: 1, featureCompatibilityVersion: 1 })
```

2.
3. **Check replica status (for Linux/Windows replica set)**

```
rs.status()
```

3.
4. **Take backup**
    - mongodump (logical)
    - Snapshot (if possible)

💼 Use case explanation

Bangalore and Mumbai teams simulate the upgrade by collecting all these pre-checks into a change record for audit.

LAB Part B – Linux Rolling Upgrade (3-node Test Replica Set)

Goal: Upgrade from 4.4 to 6.0 on Linux RS.

For each secondary node:

1. Confirm role:

```
rs.status()
```

1.
2. Stop mongod:

```
sudo systemctl stop mongod
```

2.
3. Upgrade package (simulate with a version bump, even if same version):

```
sudo apt-get update
sudo apt-get install -y mongodb-org=6.0.x
```

3.
4. Start mongod:

```
sudo systemctl start mongod
```

4.
5. Verify:

```
db.version()
rs.status()
```

After all secondaries, **step down and upgrade primary**.

💼 Use case explanation

CSC SREs in Bangalore simulate the exact sequence they'll follow on the real production cluster.

LAB Part C – Windows Standalone Upgrade

1. Stop service:

```
net stop MongoDB
```

1.
2. Run new .msi installer → Upgrade.
3. Start service:

```
net start MongoDB
```

3.
4. Verify:

```
mongosh
db.version()
```

LAB Part D – Mac Dev Upgrade

1. Stop:

```
brew services stop mongodb-community
```

1.
2. Upgrade:

```
brew upgrade mongodb-community
```

2.
3. Start:

```
brew services start mongodb-community
```

3.
4. Verify version.

LAB Part E – Atlas Cluster Upgrade

1. In Atlas → open cluster → **Upgrade Version** to target release.
2. Observe rolling upgrade progress (nodes being upgraded).
3. After success, connect via mongosh and:

```
db.version()
db.adminCommand({ getParameter: 1, featureCompatibilityVersion: 1 })
```

3.
4. Set FCV to final target version:

```
db.adminCommand({ setFeatureCompatibilityVersion: "6.0" })
```

💼 Use case explanation

Mumbai team practices Atlas upgrades so they can later handle production change windows independently.

LAB Part F – Compact & Cleanup

On any test cluster:

1. Delete some large data from filings and reminders.
2. Run:

```
db.runCommand({ compact: "filings" })
```

2.
3. Check disk usage before/after (OS tools or db.stats()).
4. Create TTL index for test:

```
db.reminders.createIndex(
  { createdAt: 1 },
  { expireAfterSeconds: 60 }
)
```

4.
5. Insert test docs with createdAt: new Date() and watch them expire.

💼 Use case explanation

CSC uses TTL for temporary reminders and compaction for heavily updated collections after major filing seasons.

---

6. Quick Upgrade Checklist for CSC Admins

Before Upgrade

- Confirm application & driver compatibility
- Check current server versions & FCV
- Validate replica health (rs.status())
- Take backup (logical + snapshot if available)
- Plan rolling upgrade order

During Upgrade

- Upgrade secondaries first
- Upgrade primary last
- Monitor replication lag and logs
- Avoid FCV change until all nodes upgraded & stable

After Upgrade

- Validate app smoke tests
- Set featureCompatibilityVersion to new version
- Plan compaction & cleanup if needed
- Update documentation + runbook
