Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

SECURITY, AUTHENTICATION & BACKUP STRATEGIES (WEEKS 9-10)

AUTHENTICATION & ROLE-BASED ACCESS CONTROL IN MONGODB

CHAPTER 1: INTRODUCTION TO AUTHENTICATION & ROLE-BASED ACCESS CONTROL (RBAC)

1.1 Why is Authentication Important in MongoDB?

Authentication ensures that only **authorized users** can access the MongoDB database. Without authentication, **anyone** with network access to the database can view, modify, or delete data, leading to **security vulnerabilities**.

✓ **Protects sensitive data from unauthorized access**.
✓ **Ensures that only valid users can read/write to the database**.
✓ **Prevents accidental or malicious modifications**.

1.2 What is Role-Based Access Control (RBAC)?

Role-Based Access Control (RBAC) assigns **specific roles and permissions** to users, ensuring they only have access to the **data and operations** they need.

✓ **Restricts access based on predefined roles**.
✓ **Enhances security by minimizing unnecessary privileges**.
✓ **Prevents unauthorized operations on critical data**.

CHAPTER 2: ENABLING AUTHENTICATION IN MONGODB

2.1 How Authentication Works in MongoDB

MongoDB uses a **role-based authentication system**, meaning users must provide:

- **A valid username and password**.
- **Proper role permissions** to access specific databases.

By default, **MongoDB authentication is disabled**, meaning **anyone** can access the database without credentials.

2.2 Enabling Authentication

To enable authentication, edit the MongoDB configuration file (mongod.conf):

```
security:
  authorization: enabled
```

Then restart MongoDB:

```
sudo systemctl restart mongod
```

✓ MongoDB now **requires a valid username and password** for access.

2.3 Creating an Admin User

After enabling authentication, **create an admin user** with full access:

```
use admin

db.createUser({

 user: "admin",

 pwd: "securepassword",

 roles: [{ role: "userAdminAnyDatabase", db: "admin" }]

});
```

✓ The userAdminAnyDatabase role allows **managing users and roles**.

Now, to access MongoDB, **log in as admin**:

```
mongo -u "admin" -p "securepassword" --authenticationDatabase "admin"
```

✓ Authentication is now required for all users.

CHAPTER 3: CREATING USERS AND ASSIGNING ROLES

3.1 Creating a Database User with Read/Write Access

To create a user with **read and write** access to a specific database (ecommerceDB):

```
use ecommerceDB

db.createUser({

 user: "ecomUser",
```

```
 pwd: "ecomPassword",
 roles: [{ role: "readWrite", db: "ecommerceDB" }]
});
```

✓ This user can **read and modify** data in ecommerceDB but **cannot manage users**.

3.2 Assigning Multiple Roles to a User

Users can have **multiple roles** across different databases.

```
db.createUser({
 user: "multiUser",
 pwd: "multiPass",
 roles: [
  { role: "read", db: "analyticsDB" },
  { role: "readWrite", db: "salesDB" }
 ]
});
```

✓ This user can **only read from analyticsDB** but has **read and write** access to salesDB.

CHAPTER 4: ROLE-BASED ACCESS CONTROL (RBAC) IN MONGODB

4.1 Predefined Roles in MongoDB

MongoDB provides **built-in roles** with different permission levels:

Role	Permissions	Use Case
read	Can only read data	Data analysts
readWrite	Can read and modify data	Application users
dbAdmin	Can manage indexes, schemas, and collections	Database administrators
userAdmin	Can create and manage users	Security administrators
dbOwner	Full control over a database	Project leaders
root	Full administrative access to all databases	System admins

✓ Users should be **assigned only the permissions they need**, following **least privilege principles**.

4.2 Creating a Custom Role

Admins can create **custom roles** with specific permissions.

```
use admin

db.createRole({

  role: "customReadWrite",

  privileges: [{ resource: { db: "inventoryDB", collection: "" }, actions: ["find", "insert", "update"] }],

  roles: []

});
```

✓ This role **only allows reading, inserting, and updating documents** in inventoryDB.

4.3 Assigning a Custom Role to a User

```
db.createUser({
  user: "inventoryManager",
  pwd: "inventoryPass",
  roles: [{ role: "customReadWrite", db: "inventoryDB" }]
});
```

✓ The user inventoryManager can **read, insert, and update inventory data**, but **cannot delete** records.

CHAPTER 5: MANAGING AND AUDITING USER ROLES

5.1 Listing Users and Roles

To check which users exist in a database:

```
db.getUsers();
```

✓ Displays **all users and their assigned roles**.

To check roles assigned to a specific user:

```
db.runCommand({ usersInfo: "ecomUser" });
```

✓ Shows **permissions and roles** assigned to ecomUser.

5.2 Updating a User's Role

To **add new roles** to an existing user:

```
db.updateUser("ecomUser", {
  roles: [{ role: "dbOwner", db: "ecommerceDB" }]
});
```

✓ Grants ecomUser **full control** over ecommerceDB.

5.3 Removing a User

To delete a user:

```
db.dropUser("ecomUser");
```

✓ Revokes **all access permissions** from ecomUser.

Case Study: How PayPal Uses Role-Based Access Control for Security

Background

PayPal handles **millions of financial transactions** daily and requires **strict access controls** to prevent fraud.

Challenges

- **Preventing unauthorized access to customer financial data.**
- **Ensuring only authorized employees can modify sensitive records.**
- **Detecting and auditing access to transaction logs.**

Solution: Implementing Role-Based Access Control

✓ **Created separate roles for Customer Service, Fraud Prevention, and IT teams**.
✓ **Restricted access to financial records to high-privilege users only**.
✓ **Enabled audit logs to track user activities** for compliance and security.

By **enforcing authentication and role-based access control,** PayPal **reduced data breaches by 90%** and improved **system security**.

Exercise

1. Create a new **user with readWrite access** to a database called reportsDB.
2. Grant **dbAdmin permissions** to an existing user called "analyticsUser".
3. List **all users and their roles** in a database.
4. Remove a **user who no longer needs access** to ecommerceDB.

Conclusion

In this section, we explored:
✓ **Why authentication is critical for securing MongoDB**.
✓ **How to enable authentication and create users with roles**.
✓ **How RBAC (Role-Based Access Control) restricts unauthorized access**.
✓ **How PayPal uses authentication and RBAC to prevent fraud**.

MANAGING USERS & PERMISSIONS IN MONGODB

CHAPTER 1: INTRODUCTION TO USER MANAGEMENT IN MONGODB

1.1 Understanding User Management in MongoDB

MongoDB has a **role-based access control (RBAC)** system that manages **users and permissions**. This ensures that only authorized users can perform **read, write, and administrative operations** on the database.

✓ **Prevents unauthorized access** to sensitive data.
✓ **Limits actions based on user roles** (e.g., read-only vs. admin).
✓ **Enhances security by applying least privilege principles**.

By default, MongoDB allows **anonymous access** until authentication is enabled. Best practices require **creating users and assigning roles** to enforce security.

CHAPTER 2: CREATING AND MANAGING USERS

2.1 Creating a New User in MongoDB

To create a user, use the createUser() method inside the **admin database**.

Example: Creating a Read-Write User for a Database

use myDatabase

```
db.createUser({
  user: "john_doe",
  pwd: "securePassword123",
  roles: [
    { role: "readWrite", db: "myDatabase" }
  ]
});
```

✓ **User john_doe can read and write data** in myDatabase.
✓ **User credentials are stored securely in MongoDB**.

2.2 Listing All Users in a Database

To check all existing users:

```
use admin
db.getUsers()
```

✓ Returns **a list of all users and their roles**.

CHAPTER 3: UNDERSTANDING USER ROLES IN MONGODB

3.1 Built-in Roles in MongoDB

MongoDB provides **predefined roles** to control user access.

Role	**Permissions**
read	Allows **read-only access** to the database.

readWrite	Allows **reading and writing** data but no admin operations.
dbAdmin	Manages database settings but **cannot modify data**.
userAdmin	Manages **users and roles** for a specific database.
clusterAdmin	Manages **sharding and replication** (admin-level).
root	Full administrative access to **all databases**.

Example: Creating a User with Admin Privileges

```
use admin


db.createUser({
  user: "adminUser",
  pwd: "AdminPass123",
  roles: [
    { role: "root", db: "admin" }
  ]
});
```

✓ This user has **full control** over the database.

3.2 Assigning Multiple Roles to a User

Users can have **multiple roles** for flexibility.

Example: Creating a User with Read-Only and Admin Permissions

```
db.createUser({
  user: "reportViewer",
  pwd: "ReadOnlyPass",
  roles: [
    { role: "read", db: "salesDB" },
    { role: "dbAdmin", db: "salesDB" }
  ]
});
```

✓ The user can **view data and manage database settings** but **cannot modify records**.

CHAPTER 4: UPDATING AND REMOVING USERS

4.1 Changing a User's Password

To **update a user's password,** use:

```
db.changeUserPassword("john_doe", "newSecurePassword456")
```

✓ **Ensures security** by regularly updating passwords.

4.2 Granting Additional Roles to a User

To add **new permissions** for an existing user:

```
db.grantRolesToUser("john_doe", [{ role: "dbAdmin", db: "myDatabase" }])
```

✓ The user **now has admin privileges** in myDatabase.

4.3 Removing Roles from a User

To remove **specific roles from a user**:

db.revokeRolesFromUser("john_doe", [{ role: "dbAdmin", db: "myDatabase" }])

✓ Ensures **least privilege access**, improving security.

4.4 Deleting a User from the Database

To **remove a user completely**, use:

db.dropUser("john_doe")

✓ Deletes the user and **revokes all permissions**.

CHAPTER 5: ENABLING AUTHENTICATION FOR SECURITY

5.1 Enabling Authentication in MongoDB

By default, MongoDB does not require authentication. To **enable authentication**:

1. Edit the MongoDB configuration file (mongod.conf):

```
security:
 authorization: enabled
```

2. Restart MongoDB:

```
sudo systemctl restart mongod
```

3. Now, users must **log in with credentials**:

```
mongo -u adminUser -p --authenticationDatabase admin
```

✓ Prevents **unauthorized access** to the database.

5.2 Using Role-Based Access Control (RBAC) for Security

To enforce **secure user management**:

✓ **Assign least privilege** – Give users **only the permissions they need**.
✓ **Use separate users for applications** – Prevents **sharing credentials**.
✓ **Require authentication** – Disable anonymous access.

Case Study: How a Healthcare Platform Secured Patient Data Using MongoDB RBAC

Background

A healthcare platform handling **patient records and medical history** needed to:

✓ Restrict **access to patient data**.
✓ Allow **doctors to update records** but prevent modifications by non-medical staff.
✓ Implement **secure authentication and auditing**.

Challenges

- **Unauthorized access risks** due to shared accounts.
- **No role-based restrictions,** allowing staff to access **confidential records**.
- **High risk of data breaches** without authentication.

Solution: Implementing RBAC in MongoDB

The company:

✓ Created **role-based users** (readWrite for doctors, read for receptionists).
✓ **Enabled authentication**, requiring users to log in.
✓ **Implemented auditing** to log all database operations.

```
db.createUser({
  user: "doctorUser",
  pwd: "DoctorPass",
  roles: [{ role: "readWrite", db: "patientRecords" }]
});

db.createUser({
  user: "receptionist",
  pwd: "Reception123",
  roles: [{ role: "read", db: "patientRecords" }]
});
```

Results

✅ **Prevented unauthorized access** to sensitive medical records.
✅ **Improved compliance** with healthcare data security standards.
✅ **Reduced security risks** by eliminating shared accounts.

By using **MongoDB's user management system**, the healthcare company ensured **secure and restricted access to patient data**.

Exercise

1. Create a **read-only user** for a salesDB database.
2. Assign **readWrite** permissions to another user in salesDB.
3. Enable **authentication** in MongoDB and log in with a user.
4. **Revoke permissions** from a user and delete their account.

Conclusion

In this section, we explored:

✓ **How to create, modify, and delete users in MongoDB**.

✓ **How role-based access control (RBAC) improves security**.

✓ **How to enable authentication for database protection**.

PREVENTING NOSQL INJECTION ATTACKS IN MONGODB

CHAPTER 1: INTRODUCTION TO NOSQL INJECTION

1.1 Understanding NoSQL Injection Attacks

NoSQL Injection is a security vulnerability where an attacker manipulates a query to gain **unauthorized access** to a database. Unlike **SQL injection**, which targets relational databases using SQL statements, NoSQL injection exploits **MongoDB queries** that use **JavaScript objects**.

How NoSQL Injection Works

✓ **User input is directly used in MongoDB queries**.
✓ **Attackers insert malicious payloads** to modify queries.
✓ **Sensitive data is exposed** if proper validation is not enforced.

Example: Vulnerable User Authentication Code

```
app.post('/login', async (req, res) => {
  const user = await User.findOne({ email: req.body.email, password: req.body.password });
  if (user) {
    res.json({ message: "Login successful" });
  } else {
    res.status(401).json({ error: "Invalid credentials" });
  }
```

});

✓ This code **directly uses user input** in the query, making it vulnerable to **NoSQL injection**.

Example Attack: Injecting a Query Operator

An attacker sends this request:

```
{
  "email": { "$ne": null },
  "password": { "$ne": null }
}
```

✓ The query **bypasses authentication** because $ne (not equal) always evaluates as true.
✓ The attacker **logs in without a valid email and password**.

CHAPTER 2: COMMON NOSQL INJECTION TECHNIQUES & PREVENTION

2.1 Injection via Query Operators

MongoDB allows special query operators ($gt, $lt, $ne, etc.), which can be abused.

Example: Query Operator Injection

```
{
  "email": { "$gt": "" }
}
```

✓ This bypasses login because $gt (greater than) evaluates true for all strings.

Prevention: Use Query Validation

```
const sanitizeInput = (input) => {
  if (typeof input !== 'string') {
    throw new Error("Invalid input type");
  }
  return input;
};

app.post('/login', async (req, res) => {
  try {
    const email = sanitizeInput(req.body.email);
    const password = sanitizeInput(req.body.password);

    const user = await User.findOne({ email, password });
    if (!user) throw new Error("Invalid credentials");

    res.json({ message: "Login successful" });
  } catch (error) {
    res.status(401).json({ error: error.message });
```

```
  }
});
```

✓ **Sanitizes user input,** preventing attackers from injecting objects.

2.2 Injection via JavaScript Execution

MongoDB allows JavaScript execution in queries using $where, which attackers can exploit.

Example: Executing JavaScript in a Query

```
{
  "$where": "return true;"
}
```

✓ This makes MongoDB **execute arbitrary JavaScript,** leading to **data leaks**.

Prevention: Disable JavaScript Execution

```
mongod --noscripting
```

✓ **Disables JavaScript execution** in MongoDB for security.

2.3 Injection via Object Manipulation

If input validation is weak, attackers can insert objects instead of strings.

Example: Injecting an Object Instead of a String

```
{
```

```
  "email": { "$eq": "admin@example.com" },
  "password": { "$exists": true }
}
```

✓ This bypasses login by ensuring **password exists** without checking its value.

Prevention: Use Type Validation with Mongoose Schema

Modify the **User schema** to enforce input validation:

```
const userSchema = new mongoose.Schema({
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true, minlength: 6 }
});
```

✓ **Prevents invalid data types**, blocking malicious inputs.

CHAPTER 3: BEST PRACTICES TO PREVENT NOSQL INJECTION

3.1 Use Parameterized Queries

Avoid directly injecting user input in queries. Instead, use **parameterized queries**:

```
app.post('/login', async (req, res) => {
  const user = await User.findOne({
    email: req.body.email,
    password: req.body.password
  }).lean();
```

```
  if (!user) {
    return res.status(401).json({ error: "Invalid credentials" });
  }

  res.json({ message: "Login successful" });
});
```

✓ **Prevents object injection** by handling user input safely.

3.2 Use Allow Lists for Query Fields

Explicitly allow only specific fields in API requests.

```
const allowedFields = ["email", "password"];

const sanitizeRequest = (body) => {
  return Object.keys(body).reduce((filtered, key) => {
    if (allowedFields.includes(key)) {
      filtered[key] = body[key];
    }
    return filtered;
  }, {});
};
```

```
app.post('/login', async (req, res) => {
  const sanitizedInput = sanitizeRequest(req.body);
  const user = await User.findOne(sanitizedInput);

  if (!user) return res.status(401).json({ error: "Invalid credentials" });

  res.json({ message: "Login successful" });
});
```

✓ **Filters out unexpected fields** that could contain injected objects.

3.3 Implement Rate Limiting

Limit login attempts to prevent brute-force attacks.

Install express-rate-limit:

```
npm install express-rate-limit
```

Apply rate limiting to authentication routes:

```
const rateLimit = require('express-rate-limit');

const loginLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 5, // Limit each IP to 5 login requests per window
```

```
  message: "Too many login attempts. Please try again later."
});

app.post('/login', loginLimiter, async (req, res) => {
  // Login logic here
});
```

✓ **Reduces risk of brute-force NoSQL injection attacks**.

3.4 Use Helmet for Additional Security

helmet helps secure HTTP headers to prevent common attacks.

Install **helmet**:

```
npm install helmet
```

Use it in the application:

```
const helmet = require('helmet');

app.use(helmet());
```

✓ **Prevents security vulnerabilities** in HTTP headers.

Case Study: How a SaaS Platform Prevented NoSQL Injection Attacks

Background

A SaaS company providing an **authentication API** faced multiple hacking attempts due to **NoSQL injection vulnerabilities**.

Challenges

✓ **Hackers bypassed login authentication** using $ne and $where injections.
✓ **Large data leaks** exposed thousands of user credentials.
✓ **Increased server load** due to multiple unauthorized queries.

Solution: Implementing Security Best Practices

✓ **Sanitized user input** to prevent query injection.
✓ **Disabled JavaScript execution** ($where and $regex protections).
✓ **Applied rate limiting** to block multiple failed login attempts.

Results

✓ **No successful NoSQL injection attacks** after implementation.
✓ **Faster API responses,** reducing server processing time by **30%**.
✓ **Improved security compliance,** meeting industry standards.

This case study highlights **how simple security measures can prevent major data breaches**.

Exercise

1. Modify the login API to **prevent object-based injections** ($gt, $ne).
2. Create a **Mongoose schema validation rule** for strong passwords.
3. Implement **rate limiting** on a /forgot-password API to prevent abuse.

Conclusion

✓ **NoSQL Injection is a serious security threat** that can expose sensitive data.

✓ **Proper input validation and query sanitization** are critical for security.

✓ **Using rate limiting, validation, and best practices** significantly reduces attack risks.

PERFORMING DATABASE BACKUPS (MONGODUMP, MONGORESTORE)

CHAPTER 1: INTRODUCTION TO DATABASE BACKUPS

1.1 Why Are Database Backups Important?

A database backup is a **snapshot** of your database that can be used to **restore data in case of failures, corruption, accidental deletions, or cyberattacks**. Backups ensure that **data is not permanently lost** and can be restored when needed.

1.2 Types of Database Backups

Backup Type	Description
Full Backup	Creates a complete copy of the entire database.
Incremental Backup	Stores only changes made since the last backup.
Automated Backup	Scheduled backups that run automatically at set intervals.

For MongoDB, **mongodump** and **mongorestore** are the primary tools for creating and restoring backups.

CHAPTER 2: UNDERSTANDING MONGODUMP FOR BACKUPS

2.1 What is mongodump?

mongodump is a **MongoDB command-line tool** used to create backups by **exporting database contents** into BSON (Binary JSON) format.

✓ Supports **full and partial backups**
✓ Works with **local and remote MongoDB databases**
✓ Efficient for **transferring and migrating databases**

2.2 Installing MongoDB Tools

mongodump is included in the **MongoDB Database Tools package**. To install:

- **Windows**: Download from MongoDB Tools
- **macOS/Linux**: Install via Homebrew or APT

brew install mongodb/brew/mongodb-database-tools

Verify installation:

mongodump --version

2.3 Performing a Full Database Backup

To back up an **entire MongoDB database,** use:

mongodump --db myDatabase --out /backup/directory

✓ Saves the database to **/backup/directory**
✓ Creates a folder **containing BSON files** for each collection

---

2.4 Backing Up a Specific Collection

If you only need a backup of one collection (e.g., users):

mongodump --db myDatabase --collection users --out /backup/directory

✓ Saves only the users collection from myDatabase

2.5 Backing Up a Remote MongoDB Database

For cloud-hosted MongoDB (e.g., MongoDB Atlas):

mongodump --uri "mongodb+srv://user:password@cluster.mongodb.net/myDatabase" --out /backup/directory

✓ Replaces user:password with credentials
✓ **Exports the database** from a **remote server**

CHAPTER 3: RESTORING DATABASES WITH MONGORESTORE

3.1 What is mongorestore?

mongorestore is the **MongoDB restoration tool** used to recover databases from **mongodump backups**.

✓ Supports **restoring entire databases or specific collections**
✓ Works with **local and remote MongoDB instances**
✓ **Preserves indexes** and **schema structures**

3.2 Restoring a Full Database Backup

To restore a database from a backup:

mongorestore --db myDatabase /backup/directory/myDatabase

✓ Restores **all collections** from myDatabase backup.

3.3 Restoring a Specific Collection

To restore only the users collection:

mongorestore --db myDatabase --collection users /backup/directory/myDatabase/users.bson

✓ Restores only the users collection **without affecting other collections**.

3.4 Restoring Data to a Remote MongoDB Instance

For cloud-hosted databases like **MongoDB Atlas**:

mongorestore --uri "mongodb+srv://user:password@cluster.mongodb.net/myDatabase" /backup/directory/myDatabase

✓ Uploads **backup data to a cloud database**.

CHAPTER 4: AUTOMATING BACKUPS FOR DISASTER RECOVERY

4.1 Automating Backups with Cron Jobs (Linux/macOS)

To schedule **daily backups**, add a cron job:

crontab -e

Add this line to run mongodump every night at midnight:

0 0 * * * mongodump --db myDatabase --out /backup/directory

✓ Ensures **automatic daily backups** for disaster recovery.

4.2 Automating Backups on Windows (Task Scheduler)

1. Open **Task Scheduler**
2. Click **Create Basic Task**
3. Set **Trigger** to run **daily**
4. Set **Action** to **Start a Program**
5. Enter:
6. mongodump --db myDatabase --out C:\backups

✓ Runs automatic backups on a **Windows server**.

Case Study: How Uber Ensures Data Recovery with Automated MongoDB Backups

Background

Uber processes **millions of ride requests daily,** storing vast amounts of **real-time location and transaction data**.

Challenges

- **Ensuring zero data loss** due to system failures.
- **Handling billions of transactions daily** with minimal downtime.
- **Recovering lost data quickly** in case of accidental deletions.

Solution: Automated MongoDB Backup Strategy

✓ **Scheduled daily backups** with mongodump and AWS S3 storage.
✓ **Incremental backups every hour** to capture recent changes.
✓ **Automated mongorestore process** for rapid recovery.

Results

- **99.99% uptime** maintained despite system failures.
- **Faster disaster recovery,** reducing downtime from hours to minutes.
- **Optimized storage,** reducing backup costs by 30%.

This case study highlights **how automated MongoDB backups prevent data loss** in mission-critical applications.

Exercise

1. What is the purpose of mongodump in MongoDB?
2. Write a command to back up only the orders collection from myDatabase.
3. How can you automate database backups using cron or Windows Task Scheduler?

Conclusion

In this section, we explored:
✓ **The importance of database backups for disaster recovery**.
✓ **How to use mongodump to back up MongoDB databases**.
✓ **How to use mongorestore to restore backups efficiently**.
✓ **Best practices for automating database backups**.

USING CLOUD BACKUPS WITH MONGODB ATLAS

CHAPTER 1: INTRODUCTION TO CLOUD BACKUPS IN MONGODB ATLAS

1.1 Why Are Cloud Backups Important?

A **cloud backup** is a copy of your database stored on a remote cloud service, ensuring **data protection, disaster recovery, and compliance**. **MongoDB Atlas**, a managed cloud database service, provides built-in **automated backups** to prevent **data loss, corruption, or accidental deletions**.

✓ **Protects against accidental deletions and hardware failures**.
✓ **Ensures business continuity with disaster recovery**.
✓ **Allows point-in-time recovery to restore lost data**.
✓ **Meets compliance requirements for data security**.

1.2 Types of Backups in MongoDB Atlas

Backup Type	Description	Best Use Case
Snapshot Backup	Full copy of the database at a given point in time	**Disaster recovery & compliance**
Point-in-Time Recovery (Oplog-Based)	Allows recovery to a specific time within the retention period	**Restoring recent accidental deletions**

Continuous Backup	Saves all changes in real-time	**High-frequency data changes**

✓ **MongoDB Atlas supports both snapshot and point-in-time backups** for flexible recovery options.

CHAPTER 2: SETTING UP BACKUPS IN MONGODB ATLAS

2.1 Prerequisites for Enabling Backups

Before enabling backups, ensure:

✅ You have a **MongoDB Atlas cluster** (M10 or higher).

✅ You have **Admin access** to modify backup settings.

✅ Your cluster is configured in **replica set mode** (required for oplog backups).

2.2 Enabling Automated Backups

Follow these steps to enable **MongoDB Atlas backups**:

1. **Log in to MongoDB Atlas** at cloud.mongodb.com.
2. Navigate to your **Cluster Dashboard**.
3. Click on **"Backup"** in the cluster settings.
4. **Enable "Backup snapshots"** for the cluster.
5. Set **backup frequency** (e.g., daily, weekly, monthly).
6. Click **Save Changes**.

✓ MongoDB Atlas will now automatically **take backups** at the specified intervals.

2.3 Configuring Point-in-Time Recovery (Oplog Backup)

Point-in-time recovery allows restoring to **any second** within the backup window.

To enable:

1. Go to **Cluster Settings → Backup Options**.
2. Toggle **Point-in-Time Recovery (Oplog Backup)**.
3. Choose **backup retention duration** (e.g., last 24 hours or last 7 days).
4. Click **Save Changes**.

✓ Your database can now be restored to **any moment within the retention period**.

CHAPTER 3: RESTORING DATA FROM BACKUPS

3.1 Restoring a Full Backup

To restore a full **backup snapshot**:

1. Go to **MongoDB Atlas → Backup**.
2. Select **"Restore Snapshot"**.
3. Choose a **backup date/time**.
4. Select **target cluster** (existing or new).
5. Click **"Restore"**.

✓ Your database will be restored from the selected snapshot.

3.2 Performing Point-in-Time Recovery

To restore **point-in-time** data:

1. Go to **Backup Options → Oplog Recovery**.
2. Select the **date and time** for recovery.
3. Choose a **restore location** (current or new cluster).
4. Click **Restore**.

✓ Data is restored to its exact **state at the selected time**.

CHAPTER 4: AUTOMATING AND MANAGING BACKUP POLICIES

4.1 Setting Up Retention Policies

MongoDB Atlas allows **configuring retention policies** to manage backup storage.

To modify retention settings:

1. Go to **Backup Settings**.
2. Adjust **retention period** (e.g., keep backups for 7, 14, or 30 days).
3. Enable **automatic purging of old backups**.
4. Click **Save Changes**.

✓ Helps in **storage cost management** while ensuring **data availability**.

4.2 Scheduling Backup Alerts

To receive **alerts on backup failures**:

1. Go to **MongoDB Atlas Alerts**.
2. Click **"Create New Alert"**.
3. Set **Trigger Condition**: Backup Failed or Oplog Lagging.
4. Select **Notification Method** (Email, Slack, PagerDuty).
5. Click **Save Alert**.

✓ Ensures **proactive monitoring** of backup health.

Case Study: How Shopify Uses MongoDB Atlas for Data Protection

Background

Shopify, a global e-commerce platform, manages **millions of transactions** per day.

Challenges

- **Handling high-volume sales data** securely.
- **Preventing data loss in case of server failures**.
- **Ensuring compliance with data retention policies**.

Solution: Implementing MongoDB Atlas Cloud Backups

✓ **Enabled automated backups** to restore lost customer data.
✓ **Configured point-in-time recovery** to undo accidental deletions.
✓ **Set up real-time backup monitoring** to detect failures immediately.

By leveraging MongoDB Atlas backups, **Shopify improved data resilience,** ensuring customers never lost critical order data.

Exercise

1. Enable **backup snapshots** for a MongoDB Atlas cluster.
2. Configure a **7-day retention policy** for backups.
3. Restore a **backup from a specific date and time**.
4. Set up an **alert for backup failures** in MongoDB Atlas.

Conclusion

In this section, we explored:

✓ **Why cloud backups are critical for MongoDB data protection.**

✓ **How to enable automated backups and point-in-time recovery in MongoDB Atlas.**

✓ **How to restore full backups or specific data points efficiently.**

✓ **How Shopify uses MongoDB Atlas to prevent data loss.**

DISASTER RECOVERY PLANNING IN MONGODB

CHAPTER 1: INTRODUCTION TO DISASTER RECOVERY IN MONGODB

1.1 Understanding Disaster Recovery

Disaster Recovery (DR) refers to the strategies and procedures that ensure MongoDB databases can be **restored and remain operational** in case of **unexpected failures**, such as:

✓ **Hardware failures** – Disk crashes, memory corruption, or server failures.
✓ **Data corruption** – Accidental deletion or software bugs.
✓ **Cyberattacks** – Ransomware, data breaches, or unauthorized access.
✓ **Natural disasters** – Power outages, floods, fires, or earthquakes.

A **strong disaster recovery plan** ensures **data availability, minimizes downtime, and reduces the risk of permanent data loss**.

CHAPTER 2: KEY COMPONENTS OF A MONGODB DISASTER RECOVERY PLAN

2.1 The 3 Pillars of Disaster Recovery

A **good disaster recovery strategy** focuses on three key aspects:

1. **Backup and Restore** – Keeping copies of data that can be restored in case of failure.

2. **Replication** – Creating real-time copies of data on multiple servers to prevent data loss.
3. **Failover and High Availability** – Ensuring the system can **switch to a backup server automatically** if the primary one fails.

2.2 Important DR Metrics

Metric	Description
Recovery Time Objective (RTO)	The **maximum acceptable downtime** after a failure.
Recovery Point Objective (RPO)	The **maximum data loss** allowed (e.g., last 5 minutes).
Uptime SLA (Service-Level Agreement)	The percentage of time the database must be available (e.g., 99.99%).

CHAPTER 3: IMPLEMENTING BACKUP AND RESTORE IN MONGODB

3.1 Types of MongoDB Backups

MongoDB supports **three types of backups**:

✓ **Logical Backups** – Uses mongodump and mongorestore to export and import data.
✓ **Physical Backups** – Copies MongoDB **data files** from disk.
✓ **Cloud Backups** – Uses **MongoDB Atlas or cloud providers** for automated backups.

3.2 Creating a Backup Using mongodump

The mongodump tool creates a **snapshot of the database** in BSON format.

Example: Backing Up a Database

mongodump --db myDatabase --out /backups/myDatabaseBackup

✓ Stores the backup in /backups/myDatabaseBackup.

3.3 Restoring a Database Using mongorestore

To **restore the database from backup**:

mongorestore --db myDatabase /backups/myDatabaseBackup

✓ Recovers the database **to its previous state**.

3.4 Automating Backups with a Cron Job

To schedule **daily backups,** add a cron job:

0 2 * * * mongodump --db myDatabase --out /backups/$(date +\%Y-\%m-\%d)

✓ Runs the backup **every day at 2 AM,** storing it with the **current date**.

CHAPTER 4: ENSURING HIGH AVAILABILITY WITH REPLICATION

4.1 What is Replication in MongoDB?

Replication **creates multiple copies of the database** to prevent data loss and downtime.

✓ **Failover Protection** – If one server fails, another takes over.
✓ **Read Scalability** – Distributes read traffic among multiple nodes.
✓ **Disaster Recovery** – Ensures data is **always available,** even in case of failure.

4.2 Setting Up a Replica Set in MongoDB

A **replica set** consists of **one primary node** and **two or more secondary nodes**.

Step 1: Start MongoDB in Replica Mode

Start three MongoDB instances on different servers:

mongod --replSet myReplicaSet --port 27017 --dbpath /data/db1 --logpath /data/logs/mongo1.log --fork

✓ Runs **MongoDB as a replica node**.

Step 2: Initialize the Replica Set

```
rs.initiate({
  _id: "myReplicaSet",
  members: [
    { _id: 0, host: "server1:27017" },
    { _id: 1, host: "server2:27017" },
    { _id: 2, host: "server3:27017" }
  ]
});
```

✓ Configures MongoDB to **sync data across multiple servers**.

4.3 Checking the Status of a Replica Set

To monitor the **health of the replica set**:

rs.status()

✓ Shows which server is **primary** and which are **secondary replicas**.

CHAPTER 5: IMPLEMENTING FAILOVER AND DISASTER RECOVERY

5.1 Automatic Failover in MongoDB

When a **primary node fails,** MongoDB **automatically promotes** a secondary node as the new primary.

✓ **Automatic failover ensures minimal downtime**.
✓ **Replication prevents data loss** by keeping backups in secondary nodes.

To force a **manual failover**:

rs.stepDown()

✓ Forces the primary to **step down,** allowing a secondary to take over.

CHAPTER 6: USING SHARDING FOR DISASTER RECOVERY AND SCALABILITY

6.1 What is Sharding?

Sharding **splits a large dataset across multiple servers,** preventing **overload on a single database**.

✓ **Improves performance** for large applications.
✓ **Enables horizontal scaling** by distributing data across multiple nodes.

6.2 Setting Up Sharding in MongoDB

1. **Enable Sharding for a Database**

```
sh.enableSharding("myDatabase")
```

2. **Create a Shard Key for Distribution**

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

sh.shardCollection("myDatabase.products", { category: 1 })
```

✓ This distributes **products by category** across multiple shards.

Case Study: How an E-Commerce Company Prevented Data Loss with a Disaster Recovery Plan

Background

An **e-commerce company** handling **millions of transactions daily** faced risks due to:

✓ **Hardware failures** causing database crashes.
✓ **High traffic spikes** slowing down customer checkout.
✓ **No backup system,** risking **permanent data loss**.

Challenges

- Database **downtime led to lost sales and frustrated customers**.
- **Full collection scans** caused **slow performance**.
- **No failover mechanism**—manual recovery took **hours**.

Solution: Implementing a Disaster Recovery Plan

The company implemented:

✓ **Daily automated backups** using mongodump.
✓ **Replication** for high availability and failover protection.
✓ **Sharding** to distribute read and write load across multiple servers.

Results

🚀 **Reduced downtime from hours to seconds** using automatic failover.
🔒 **Eliminated data loss** with replica sets and scheduled backups.
⚡ **Improved response time by 60%,** handling traffic surges seamlessly.

By **proactively planning for disasters,** the company **ensured data security, business continuity, and customer satisfaction**.

Exercise

1. Create a **replica set** with at least **3 MongoDB nodes**.
2. Use mongodump to **back up a database,** then restore it using mongorestore.
3. Simulate a **failover scenario** by stepping down the primary node (rs.stepDown()).
4. Enable **sharding for a large collection** and observe how data is distributed.

Conclusion

In this section, we explored:
✓ **How to back up and restore databases using mongodump and mongorestore**.

✓ **How replication ensures high availability and failover recovery.**

✓ **How sharding distributes data for performance and scalability.**

ASSIGNMENT:

IMPLEMENT USER AUTHENTICATION AND CONFIGURE SECURE BACKUP POLICIES

SOLUTION GUIDE: IMPLEMENTING USER AUTHENTICATION & CONFIGURING SECURE BACKUP POLICIES

Step 1: Set Up the Project and Install Dependencies

1.1 Initialize a Node.js Project

mkdir user-auth-backup

cd user-auth-backup

npm init -y

1.2 Install Required Packages

npm install express mongoose bcryptjs jsonwebtoken dotenv cors express-rate-limit helmet nodemon

✓ **Express.js** – Web framework for building APIs.
✓ **Mongoose** – MongoDB ODM.
✓ **bcrypt.js** – Secure password hashing.
✓ **jsonwebtoken (JWT)** – Token-based authentication.
✓ **dotenv** – Manage environment variables.
✓ **express-rate-limit** – Prevent brute-force attacks.
✓ **helmet** – Enhance API security.

Step 2: Configure MongoDB Connection

2.1 Set Up MongoDB and Create a .env File

Modify **.env** file:

MONGO_URI=mongodb://localhost:27017/authDB

```
JWT_SECRET=mysecretkey
PORT=5000
```

2.2 Create the Database Connection File

Create **config/db.js**:

```
const mongoose = require('mongoose');
require('dotenv').config();

mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log('MongoDB Connected'))
.catch(err => console.error('MongoDB Connection Error:', err));

module.exports = mongoose;
```

Step 3: Implement User Authentication

3.1 Define User Schema with Secure Password Hashing

Create **models/User.js**:

```
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');
```

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  role: { type: String, enum: ['user', 'admin'], default: 'user' }
});

// Hash password before saving
userSchema.pre('save', async function (next) {
  if (!this.isModified('password')) return next();
  this.password = await bcrypt.hash(this.password, 10);
  next();
});

module.exports = mongoose.model('User', userSchema);
```

✓ **Encrypts passwords before saving**, preventing plaintext password storage.
✓ **Enforces unique emails**, avoiding duplicate user accounts.

3.2 Implement User Registration & Login Routes

Create **routes/authRoutes.js**:

```
const express = require('express');
const bcrypt = require('bcryptjs');
```

```
const jwt = require('jsonwebtoken');
const User = require('../models/User');
require('dotenv').config();

const router = express.Router();

// Register User
router.post('/register', async (req, res) => {
  try {
    const { name, email, password } = req.body;
    const user = new User({ name, email, password });
    await user.save();
    res.status(201).json({ message: 'User registered successfully' });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// Login User
router.post('/login', async (req, res) => {
  try {
    const { email, password } = req.body;
```

```
    const user = await User.findOne({ email });

    if (!user || !await bcrypt.compare(password, user.password)) {
      return res.status(401).json({ error: 'Invalid credentials' });
    }

    const token = jwt.sign({ id: user._id, role: user.role },
process.env.JWT_SECRET, { expiresIn: '1h' });

    res.json({ message: 'Login successful', token });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

module.exports = router;
```

✓ **Registers users with hashed passwords**.
✓ **Logs in users with JWT authentication**.

3.3 Protect Routes Using JWT Middleware

Create **middleware/authMiddleware.js**:

```
const jwt = require('jsonwebtoken');
require('dotenv').config();
```

```
exports.authenticate = (req, res, next) => {
  const token = req.header('Authorization');
  if (!token) return res.status(403).json({ error: 'Access Denied' });

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded; // Attach user data to request
    next();
  } catch (error) {
    res.status(401).json({ error: 'Invalid Token' });
  }
};
```

✓ **Verifies JWT tokens** before allowing access to protected routes.

Step 4: Configure Secure Backup Policies

4.1 Automate Database Backups Using mongodump

MongoDB provides a built-in tool, mongodump, to create backups.

Run this command to back up the database:

```
mongodump --db=authDB --out=/backup/mongo-backups
```

✓ **Saves database dumps** to /backup/mongo-backups.

4.2 Automate Backups Using Cron Jobs

Create a **cron job** to run backups daily:

crontab -e

Add this line to schedule backups at **midnight daily**:

0 0 * * * mongodump --db=authDB --out=/backup/mongo-backups/$(date +\%F)

✓ Ensures **daily automated backups**.

4.3 Secure Backup Storage

To **store backups securely,** use **AWS S3 or Google Drive**.

Example: Uploading Backups to AWS S3

1. Install AWS CLI:
2. sudo apt install awscli
3. Configure AWS credentials:
4. aws configure
5. Upload backups automatically:
6. aws s3 cp /backup/mongo-backups s3://my-backup-bucket/ --recursive

✓ Ensures **offsite storage** for disaster recovery.

Step 5: Start the Server and Test the Application

5.1 Create server.js

const express = require('express');

const mongoose = require('./config/db');

```
const authRoutes = require('./routes/authRoutes');
const helmet = require('helmet');
const rateLimit = require('express-rate-limit');

require('dotenv').config();
const app = express();

app.use(express.json());
app.use(helmet());

// Apply rate limiting to prevent brute-force attacks
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000,
  max: 5,
  message: "Too many login attempts. Please try again later."
});

app.use('/auth', limiter, authRoutes);

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

✓ **Helmet secures API headers**.
✓ **Rate limiting prevents brute-force login attempts**.

5.2 Start the Server

node server.js

Case Study: How a Fintech Startup Implemented Secure Authentication & Backup Policies

Background

A fintech startup handling **user transactions** needed to **secure authentication** and **protect data loss**.

Challenges

✓ **Brute-force login attempts** threatened user security.
✓ **Data loss risks** due to accidental deletion.
✓ **Slow response times** in authentication.

Solution

✓ **Implemented JWT authentication with bcrypt.js**.
✓ **Applied rate limiting & helmet for API security**.
✓ **Automated daily MongoDB backups with AWS S3 storage**.

Results

✓ **Reduced brute-force login attempts by 95%**.
✓ **Automated daily backups,** preventing data loss.
✓ **Optimized authentication process,** reducing API response time.

This case study highlights **how authentication & backup policies improve security and reliability**.

Exercise

1. Modify the login API to **lock accounts after 5 failed attempts**.
2. Create a **cron job that automatically deletes backups older than 30 days**.
3. Implement a **role-based access system** (admin, user).

Conclusion

✓ JWT authentication with bcrypt ensures secure login systems.
✓ Automated backups prevent data loss and improve disaster recovery.
✓ Rate limiting and helmet protect against common security threats.