State Management and Working with Data

Introduction to session control

- A **session** is a temporary data storage mechanism used to **remember user information** across multiple requests in a web application.

- Since **HTTP is a stateless protocol**, each request is independent — meaning the server doesn’t remember who you are between page loads.

- A **session** solves this by storing data on the server side and linking it with a **unique session ID** stored in the client’s browser (usually as a cookie).

Introduction to session control

Session control allows a web application to:

- Keep users **logged in** as they move between pages.
- Store temporary data (like shopping cart items).
- Manage **user preferences**, authentication states, or visit counters.
- Improve **security** by not storing sensitive data directly in cookies.

Introduction to session control

- **Session Control in Node.js** allows app to maintain user state and identity across multiple HTTP requests — essential for logins, carts, and personalization.

- Using packages like express-session, developer can easily create, manage, and destroy sessions safely and efficiently.

Introduction to session control

- Steps in Session Control

a. A user logs in or sends a request to the server.
b. The server creates a **session object** for that user and assigns a **unique session ID**.
c. The **session ID** is sent back to the client as a **cookie**.
d. On each subsequent request, the client sends the cookie (session ID).
e. The server uses this ID to fetch the correct session data.

Session Control in NodeJS

- You can use the express-session middleware to easily manage sessions in NodeJS.

```
npm init –y
npm express express-session
```

- Coding Example

Session vs. Cookie

FEATURE	SESSION	COOOKIE
Storage	Server Memory/ Database	Browser Storage
Security	More Secure	Less Secure
Data Size	Large data can be stored	Limited
Lifespan	Until browser closes	As per expiry date

Session Control in NodeJS

- A **session** allows the server to store information (called *session variables*) about each user between different HTTP requests.
- **Creating / Starting a session:** happens automatically when you first set a session variable.
- **Session variables:** values saved in *req.session* that stay available across pages until the session is destroyed.
- **Destroying a session:** removes all data and ends that user’s session.

Session Control in NodeJS

- A session starts automatically when you assign the first req.session variable.

- Session variables hold user data across multiple requests.

- Use req.session.destroy() to end it.

Session Control in NodeJS

- Note: Before execution of session related code ensure to install associated packages (e.g. express session)

- Coding Example

Session Control in NodeJS

- Key Concepts

ACTION	EXPRESSION	DESCRIPTION
START SESSION	app.use(session({...}))	Initializes session system
CREATE VARIABLE	req.session.username = 'John'	Stores data in current session
READ VARIABLE	req.session.username	Access stored session variable
UPDATE VARIABLE	req.session.username = 'Jane'	Modify stored data
DESTROY VARIABLE	req.session.destroy()	Deletes session data completely

Cookies in NodeJS

- A **cookie** is a small piece of data stored on the client’s (browser’s) side and sent to the server with every request.
- It helps the server **remember information about the user** between page visits.

- Cookies are often used for:
- Storing **user preferences** (like theme or language)
- **Session tracking** (e.g., login state)
- **Analytics or personalization**

Cookies in NodeJS

- **Working of Cookies**

➢The server sends a cookie to the client (browser).

➢The browser stores the cookie.

➢Each time the client makes a request, the cookie is sent back to the server automatically.

Cookies in NodeJS

- Note: Before starting to work with cookies. Developer needs to ensure installation of cookie-parser middleware for easy cookie handling.

- Coding Example showcases setting up of cookies, reading the cookie and deleting the cookie

Query String in NodeJS

- A **Query String** is the part of a URL that carries **data in key–value pairs**, usually after a ?.

- Example

http://localhost:3000/search?name=John&age=25

- Coding Example

Cookie vs Query String

FEATURE	COOKIE	QUERY STRING
STORED	On Client Browser	In URL
VISIBLE TO USER	No	Yes
BEST SUITED FOR	Saving user preference, login info	Passing data between pages
PERSISTANCE	Can last for days/week	Only valid for one request unless reused
SECURITY	More Secure	Less secure

MongoDB : Introduction

- **MongoDB** is an **open-source NoSQL database** that stores data in a **flexible, JSON-like format** called **documents** instead of traditional tables.

- Developed by **MongoDB Inc.**
- Written in **C++**
- Designed to handle **large amounts of data** efficiently.
- Allows you to **store, query, and analyze** data in real time.

MongoDB : Introduction

Feature	Description
Document-Oriented	Stores data as documents (JSON-like structure) instead of rows and columns.
No Fixed Schema	Each document can have different fields; no rigid structure like SQL tables.
High Performance	Fast reads/writes due to its design and indexing.
Scalability	Supports **horizontal scaling** through **sharding** (distributing data across multiple machines).
Flexibility	Ideal for changing or unstructured data.
Rich Query Language	Powerful queries using JSON-style syntax.

MongoDB vs SQL Databases

SQL Database	MongoDB
Data stored in **tables**	Data stored in **collections**
Rows and columns	Documents and fields
Fixed schema	Dynamic schema
Joins are used	Embedded documents (no joins needed)
Example: MySQL, PostgreSQL	Example: MongoDB

MongoDB Terminology

SQL Term	MongoDB Equivalent	Description
Database	Database	A container for collections
Table	Collection	A group of related documents
Row	Document	A single record stored in JSON-like format
Column	Field	A key-value pair inside a document
Primary Key	_id	Unique identifier for each document

Basic MongoDB Commands

Command	Description
show dbs	List all databases
use college	Create or switch to a database
db.createCollection("students")	Create a new collection
db.students.insertOne({name: "Rahul", age: 22})	Insert one document
db.students.find()	View all documents
db.students.find({age: 22})	Query with condition
db.students.updateOne({name: "Rahul"}, {$set: {age: 23}})	Update a document
db.students.deleteOne({name: "Rahul"})	Delete a document
db.dropDatabase()	Delete the current database

Advantages of MongoDB

- Schema-less and flexible.
- Easy to scale horizontally.
- High speed for big data applications.
- Integrates easily with Node.js (via the Mongoose library).
- Ideal for modern web apps, IoT, AI/ML data storage, and real-time analytics.

MongoDB Application Areas

- Social Media Platforms
- E-commerce Product Catalogs
- Content Management Systems (CMS)
- Real-time Analytics Dashboards
- IoT and Sensor Data Storage

Hosting and Authenticating in MongoDB

- **Hosting** means **running your MongoDB database on a server** so it can be accessed online — by your applications or users — rather than only on your local computer.

- MongoDB can be hosted in **two main ways**:

➢**Self-hosted**- You install MongoDB on your own server or local system.

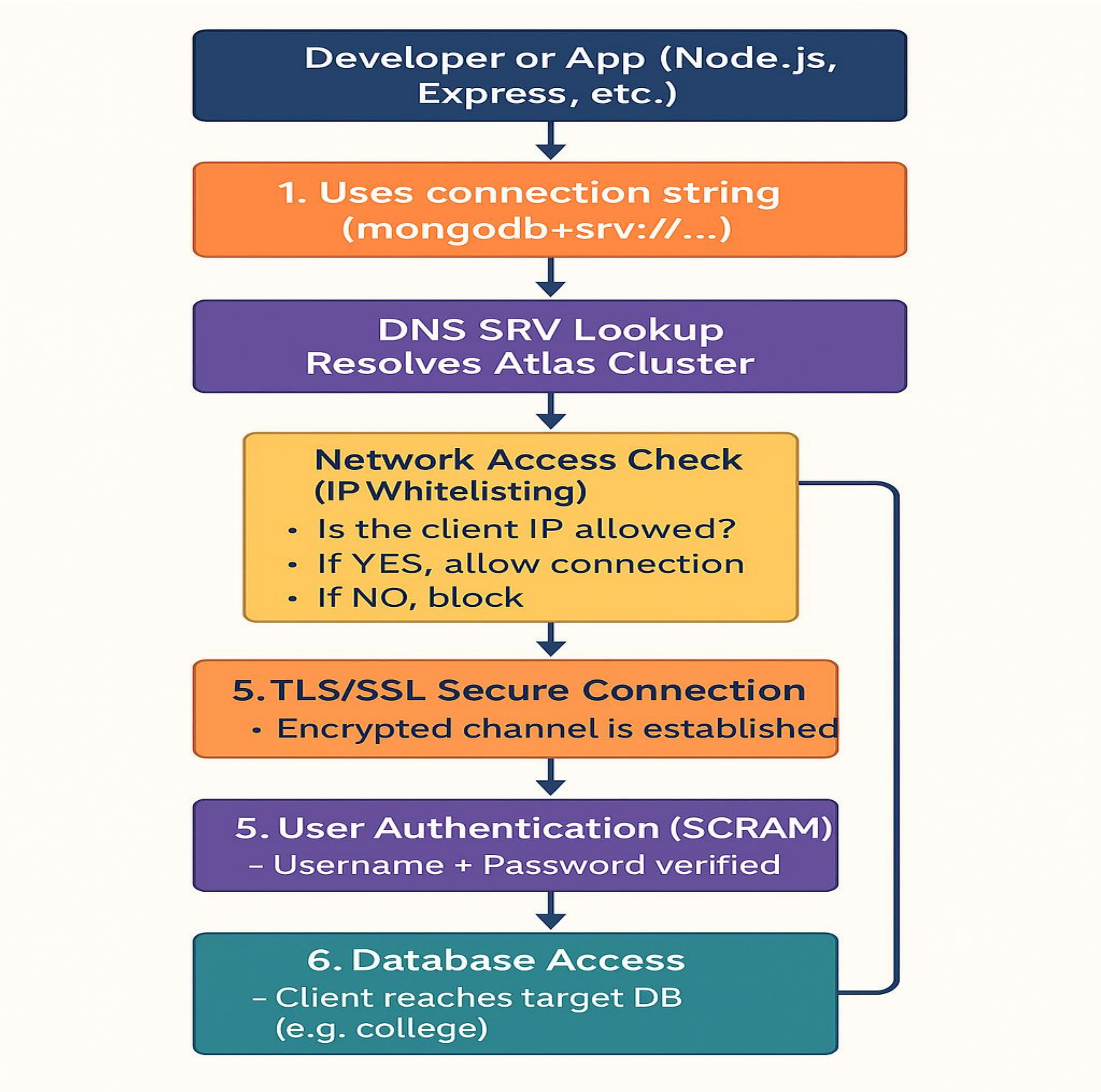
➢**Cloud-hosted** - Managed online service provided by MongoDB Inc. (example MongoDB Atlas)

Authentication in MongoDB

- **Authentication** ensures that only authorized users can access or modify the database.
- MongoDB supports several authentication methods:

Type	Description
SCRAM-SHA-256 (Default)	Username-password based login
x.509 Certificates	For encrypted client–server communication
LDAP / Kerberos	Used in enterprise environments
IAM / Cloud Access	Cloud-based identity access management

MongoDB connection Flow (Atlas)



Developer or App (Node.js, Express, etc.)

1. Uses connection string (mongodb+srv://...)

DNS SRV Lookup
Resolves Atlas Cluster

Network Access Check
(IP Whitelisting)

- Is the client IP allowed?
- If YES, allow connection
- If NO, block

5. TLS/SSL Secure Connection

- Encrypted channel is established

5. User Authentication (SCRAM)
– Username + Password verified

6. Database Access
– Client reaches target DB
(e.g. college)

Best Practices for MongoDB Hosting and Authentication

Category	Best Practice
Security	Never hardcode passwords — use environment variables
Access Control	Give least privileges
IP Whitelisting	Restrict access only to specific IPs
Encryption	Use TLS/SSL for cloud connections
Backups	Schedule regular backups
Monitoring	Enable performance metrics

Model Creation in MongoDB

- In MongoDB (especially when using **Node.js**), we don’t create “models” directly inside MongoDB. Instead, we use **Mongoose**, to define **Schema** and **Models**.

- A **Model** in MongoDB is a **JavaScript representation of a collection**.

- A **Schema** defines the structure of documents.

- A **Model** creates and interacts with the collection.

- Documents are saved based on the model structure.

Model Creation in MongoDB

- Example Schema

```
const studentSchema = new mongoose.Schema({
  name: String,
  age: Number,
  course: String,
  isActive: Boolean
});
```

Model Creation in MongoDB

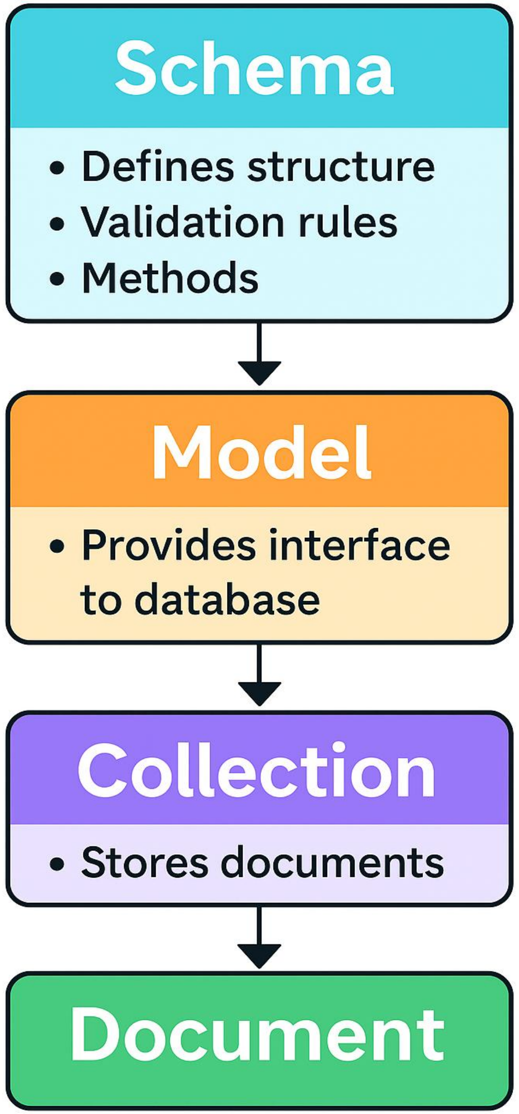
- Example Model

```
const Student = mongoose.model('Student', studentSchema);
```

- Student here becomes the model name.

- Mongoose will create a collection called **student.**

Model Creation in MongoDB



Schema

- Defines structure
- Validation rules
- Methods

Model

- Provides interface to database

Collection

- Stores documents

Document

Managing Database Connections MongoDB

- To manage database connection effectively in MongoDB whilst using NodeJS following points need to be taken into consideration.

➢ **Use one client per process (a singleton)**

- Node.js app should create **only one MongoDB client connection** when it starts.

- The MongoDB driver already manages a connection pool internally.

- Creating multiple clients = multiple pools = unnecessary open connections.

- Too many connections leads to: high memory usage, increased Atlas billings, slower performance etc.

Managing Database Connections MongoDB

➢**Never open/close per request. Open once at startup; reuse.**

- Opening a new MongoDB connection takes ~10–200 ms. If one does this for every request it would lead to app slowing down, MongoDB gets overloaded Latency spikes, crashing the DB.

➢**Tune the pool settings**

- MongoDB drivers automatically create a **pool of connections**.
- Tuning of the pool settings can be done by using “*maxPoolSize” and “minPoolSize”.*
- *Helps define number of sockets to keep open for concurrency control.*

Managing Database Connections MongoDB

➢**Handle app lifecycle**

- Node.js apps can shut down because of SIGINT and SIGTERM(Signals used to terminate a process), rolling updates and or pm2 restart commands. (e.g. Docker and Kubernetes)

- So, one needs to end MongoDB connections gracefully i.e. Implement **shutdown hooks** or **signal handlers** in your application to catch termination signals.

Managing Database Connections MongoDB

- **Test connectivity on boot**

- When the app starts, do a quick **ping** to verify MongoDB is running.

- This ensures that DB is not down even if app is running.

- Code : **`await db.command({ ping: 1 });`**

Managing Database Connections MongoDB

➢**Retries with backoff for initial connection**

- Sometimes Atlas is slow, DNS has not propagated, internet speeds are not up to the mark.

- In such cases retry connecting but with exponential backoff.

- Endless retries lead to hanging up of the app, masking of real failures, results in large amounts of logs.

Managing Database Connections MongoDB

➢**Monitor with MongoDB Atlas or local logs**

- Continuous monitoring is crucial to understand connection pool usage, slow queries, CPU usage, number if active operations and network errors.

- MongoDB Atlas Performance Panel may be used for continuous monitoring.

Basic Operations : MongoDB

- MongoDB stores data in **databases → collections → documents**.

- Basic operations come under CRUD(**C**reate, **R**ead, **U**pdate and **D**elete).

❑**Create (Insert)**

```
db.students.insertOne({
 name: "Arjun",
 age: 22,
 course: "CSE"
});
```

Basic Operations : MongoDB

❑Insert multiple documents

```
db.students.insertMany([
 { name: "Neha", age: 21 },
 { name: "Rohan", age: 23 }
]);
```

Basic Operations : MongoDB

❑Read (Find)

- Find all documents

```
db.students.find();
```

- Find with condition

```
db.students.find({ age: 22 });
```

Basic Operations : MongoDB

❑**Read (Find)**

- Find one document

```
db.students.findOne({ name: "Arjun" });
```

- Select specific fields

```
db.students.find(
 { age: 22 },
 { name: 1, course: 1, _id: 0 }
);
```

Basic Operations : MongoDB

❑**Update**

- Update one document

```
db.students.updateOne(
 { name: "Arjun" },
 { $set: { age: 23 } }
);
```

Basic Operations : MongoDB

❑**Update**

- Update many documents

```
db.students.updateMany(
 { course: "CSE" },
 { $set: { isActive: true } }
);
```

Basic Operations : MongoDB

❑Update

- Replace complete document

```
db.students.replaceOne(
 { name: "Arjun" },
 { name: "Arjun", age: 24, course: "CSE" }
);
```

Basic Operations : MongoDB

❑**Delete**

- Delete one

```
db.students.deleteOne({ name: "Neha" });
```

- Delete many

- **db.students.deleteMany({ course: "CSE" });**

Additional Operations : MongoDB

- **Sort (ascending)**

```
db.students.find().sort({ age: 1 });
```

- **Sort (descending)**

```
db.students.find().sort({ age: -1 });
```

Additional Operations : MongoDB

- **Count documents**

```
db.students.countDocuments();
```

- **Create Indexes**

```
db.students.createIndex({ name: 1 });
```

Additional Operations : MongoDB

- **List indexes**

```
db.students.getIndexes();
```

THANK YOU