RESEARCH INNOVATION DISCOVERY (RID BHARAT)
A place for coder, developer & Researcher
RUN BY TWKSAA WELFARE FOUNDATION & ISO: 9001:2015

Reg.No: 048884
Foundation Day
05-09-2020

MongoDB

Er. Rajesh Prasad(B.E, M.E)
Founder: TWF & RID Org.

- **RID ORGANIZATION** यानि **Research, Innovation and Discovery** संस्था जिसका मुख्य उदेश्य हैं आने वाले समय में सबसे पहले **NEW (RID, PMS & TLR)** की खोज, प्रकाशन एवं उपयोग भारत की इस पावन धरती से भारतीय संस्कृति, सभ्यता एवं भाषा में ही हो |
- देश, समाज, एवं लोगों की समस्याओं का समाधान **NEW (RID, PMS & TLR)** के माध्यम से किया जाये इसके लिए ही मैं राजेश प्रसाद **इस RID संस्था** की स्थपना किया हूँ|
- Research, Innovation & Discovery में रूचि रखने वाले आप सभी विधार्थियों, शिक्षकों एवं बुधीजिवियो से मैं आवाहनं करता हूँ की आप सभी **इस RID संस्था** से जुड़ें एवं अपने बुध्दि, विवेक एवं प्रतिभा से दुनियां को कुछ नई **(RID, PMS & TLR)** की खोजकर, बनाकर एवं अपनाकर लोगों की समस्याओं का समाधान करें |

त्वक्सा Node JS के इस ई-पुस्तक में आप Node JS से जुड़ी सभी बुनियादी अवधारणाएँ सीखेंगे। मुझे आशा है कि इस ई-पुस्तक को पढ़ने के बाद आपके ज्ञान में वृद्धि होगी और आपको कंप्यूटर विज्ञान के बारे में और अधिक जानने में रुचि होगी

"In this E-Book of TWKSAA Node Js you will learn all the basic concepts related to Node Js. I hope after reading this E-Book your knowledge will be improve and you will get more interest to know more thing about computer Science".

RID हमें क्यों करना चाहिए ?

(Research)

अनुसंधान हमें क्यों करना चाहिए ?

Why should we do research?

1. नई ज्ञान की प्राप्ति (Acquisition of new knowledge)
2. समस्याओं का समाधान(To Solving problems)
3. सामाजिक प्रगति (To Social progress)
4. विकास को बढ़ावा देने(To promote development)
5. तकनीकी और व्यापार में उन्नति(To advances in technology & business)
6. देश विज्ञान और प्रौद्योगिकी के विकास(To develop the country's science & technology)

(Innovation)

नवीनीकरण हमें क्यों करना चाहिए ?

Why should we do Innovation?

1. प्रगति के लिए(To progress)
2. परिवर्तन के लिए(For change)
3. उत्पादन में सुधार(To Improvement in production)
4. समाज को लाभ(To Benefit to society)
5. प्रतिस्पर्धा में अग्रणी (To be ahead of competition)
6. देश विज्ञान और प्रौद्योगिकी के विकास। (To develop the country's science & technology)

(Discovery)

खोज हमें क्यों करना चाहिए?

Why should we do Discovery?

1. नए ज्ञान की प्राप्ति(Acquisition of new knowledge)
2. अविष्कारों की खोज(To Discovery of inventions)
3. समस्याओं का समाधान(To Solving problems)
4. ज्ञान के विकास में योगदान(Contribution to development of knowledge)
5. समाज के उन्नति के लिए (for progress of society)
6. देश विज्ञान और तकनीक के विकास (To develop the country's science & technology)

❖ Research(अनुसंधान):

- अनुसंधान एक प्रणालीकरण कार्य होता है जिसमें विशेष विषय या विषय की नई ज्ञान एवं समझ को प्राप्त करने के लिए सिद्धांतिक जांच और अध्ययन किया जाता है। इसकी प्रक्रिया में डेटा का संग्रह और विश्लेषण, निष्कर्ष निकालना और विशेष क्षेत्र में मौजूदा ज्ञान में योगदान किया जाता है। अनुसंधान के माध्यम से विज्ञान, प्रोधोगिकी, चिकित्सा, सामाजिक विज्ञान, मानविकी, और अन्य क्षेत्रों में विकास किया जाता है। अनुसंधान की प्रक्रिया में अनुसंधान प्रश्न या कल्पनाएँ तैयार की जाती हैं, एक अनुसंधान योजना डिज़ाइन की जाती है, डेटा का संग्रह किया जाता है, विश्लेषण किया जाता है, निष्कर्ष निकाला जाता है और परिणामों को उचित दर्शाने के लिए समाप्ति तक पहुंचाया जाता है।

❖ Innovation(नवीनीकरण): -

- Innovation एक विशेषता या नई विचारधारा की उत्पत्ति या नवीनीकरण है। यह नए और आधुनिक विचारों, तकनीकों, उत्पादों, प्रक्रियाओं, सेवाओं या संगठनात्मक ढंगों का सृजन करने की प्रक्रिया है जिससे समस्याओं का समाधान, प्रतिस्पर्धा में अग्रणी होने, और उपयोगकर्ताओं के अनुकूलता में सुधार किया जा सकता है।

❖ Discovery (आविष्कार):

- Discovery का अर्थ होता है "खोज" या "आविष्कार"। यह एक विशेषता है जो किसी नए ज्ञान, अविष्कार, या तत्व की खोज करने की प्रक्रिया को संदर्भित करता है। खोज विज्ञान, इतिहास, भूगोल, तकनीक, या किसी अन्य क्षेत्र में हो सकती है। इस प्रक्रिया में, व्यक्ति या समूह नए और अज्ञात ज्ञान को खोजकर समझने का प्रयास करते हैं और इससे मानव सभ्यता और विज्ञान-तकनीकी के विकास में योगदान देते हैं।

नोट : अनुसंधान विशेषता या विषय पर नई ज्ञान के प्राप्ति के लिए सिस्टमैटिक अध्ययन है, जबकि आविष्कार नए और अज्ञात ज्ञान की खोज है।

सुविचार:

1.	**समस्याओं का समाधान करने का उतम मार्ग हैं |**	→ शिक्षा ,RID, प्रतिभा, सहयोग, एकता एवं समाजिक-कार्य
2.	**एक इंसान के लिए जरूरी हैं |**	→ रोटी, कपड़ा, मकान, शिक्षा, रोजगार, इज्जत और सम्मान
3.	**एक देश के लिए जरूरी हैं |** -	→ संस्कृति-सभ्यता, भाषा, एकता, आजादी, संविधान एवं अखंडता
4.	**सफलता पाने के लिए होना चाहिए |**	→ लक्ष्य, त्याग, इच्छा-शक्ति, प्रतिबद्धता, प्रतिभा, एवं सतता
5.	**मरने के बाद इंसान छोडकर जाता हैं |**	→ शरीर, अन-धन, घर-परिवार, नाम, कर्म एवं विचार
6.	**मरने के बाद इंसान को इस धरती पर याद किया जाता हैं उनके**	

➔ नाम, काम, दान, विचार, सेवा-समपर्ण एवं कर्मो से...

आशीर्वाद (बड़े भैया जी)

Mr. RAMASHANKAR KUMAR

मार्गदर्शन एवं सहयोग

Mr. GAUTAM KUMAR

.....सोच है जिनकी नये कुछ कर दिखाने की, खोज हैं मुझे आप जैसे इंसान की.....

"अगर आप भी **Research, Innovation and Discovery** के क्षेत्र में रूचि रखतें हैं एवं अपनी प्रतिभा से दुनियां को कुछ नया देना चाहतें एवं अपनी समस्या का समाधान **RID** के माध्यम से करना चाहतें हैं तो **RID ORGANIZATION (रीड संस्था)** से जरुर जुड़ें" || **धन्यवाद** || **Er. Rajesh Prasad (B.E, M.E)**

INTRODUCTION TO MONGODB

MongoDB is a **database** that stores data in document format (**like JSON**) instead of tables, making it easy to handle large and flexible data.

Difference Between NoSQL and SQL Databases		
Feature	**NoSQL Database**	**SQL Database**
Storage	Stores data as documents (JSON), key-value, graph, etc.	Stores data in tables (rows & columns)
Schema	Flexible, no fixed structure	Fixed, predefined columns
Scalability	Horizontally scalable (many servers)	Vertically scalable (powerful server)
Joins	No joins (uses embedding/referencing)	Supports JOIN operations
Query Language	Uses JSON-like queries	Uses SQL language
Performance	Faster for big data	Slower for complex joins
Flexibility	Best for unstructured data	Best for structured data
Transactions	Limited ACID, supports BASE	Fully ACID compliant
Examples	MongoDB, Firebase, Cassandra	MySQL, PostgreSQL, Oracle

Benefits of MongoDB

1. **Flexible –** No fixed structure
2. **Fast –** Quick read/write
3. **Scalable** – Handles big data
4. **Reliable –** Backup and recovery
5. **Smart queries** – Easy data search
6. **JavaScript-friendly –** Works with Node.js
7. **Big** Data & IoT support
8. **Free** and open source
9. **Best** for real-time apps

JSON (JavaScript Object Notation):

JSON is a simple, lightweight format used to store and share data between server and web apps.

Features of JSON:

- Lightweight and easy to read
- Human and machine readable
- Works with all languages
- Supports objects and arrays
- Used in APIs and web communication

Note: Mostly, we send or exchange data from web apps to the server in three formats: JSON, XML, and CSV

JSON Structure (JavaScript Object Notation)

Used in **web APIs** for sending data between client and server.

Example:

```
{
  "name": “Sangam Kumar",
  "age": 27,
  "skills": ["JavaScript", "Node.js"]
  "address": { "city": "Patna",}
}
```

Structure: { "key": "value" } → key-value pairs inside curly braces.

XML Structure (Extensible Markup Language)

Used for **data storage and configurations.**

Example:

```
<student>
  <name>Tanmay</name>
  <age>18</age>
  <course>Python</course>
</student>
```

Structure: <tag>value</tag> → data wrapped in opening and closing tags.

CSV Structure (Comma-Separated Values)

Used for **tabular data** like spreadsheets.

```
name, age, course
Tanmay, 18, Python
```

Structure: Each row represents a record, and columns are separated by commas (,).

How to Download and Install MongoDB on Windows

Step 1: Download MongoDB

1. Go to :- https://www.mongodb.com/try/download/community
2. Choose:
   - **Edition:** Community Server
   - **Platform:** Windows
   - **Package:** .msi
3. Click **Download**.

Step 2: Install MongoDB

1. Double-click the downloaded .msi file.
2. Click **Next → Complete installation**.
3. Check **Install MongoDB as a Service**.
4. Click **Install → Finish**.

Step 3: Verify Installation

1. Open **CMD** and type:
2. mongod --version

❖ How to Add MongoDB to System PATH

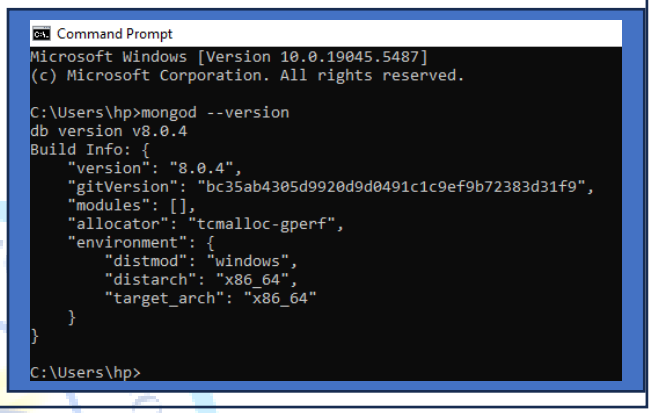
Find Path

Go to: :-> C:\Program Files\MongoDB\Server\<version>\bin

(copy this path)

❖ How to Add to Environment Variables

1. Press **Win + R → type sysdm.cpl → Enter**
2. Go to **Advanced → Environment Variables**
3. Under **System Variables**, select **Path → Edit → New**
4. Paste the copied path → **OK**
5. Restart **CMD or PC**

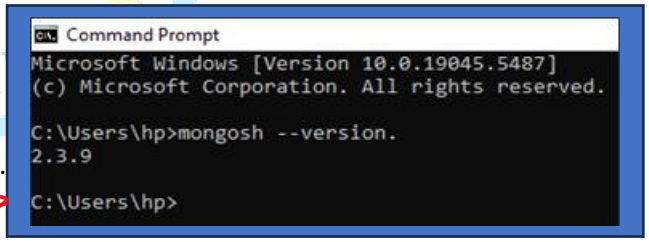


MongoDB Shell (mongosh):- it is a command-line tool used to interact with and manage MongoDB databases.

How to Install MongoDB Shell (mongosh)

1. Download MongoDB Shell

1. Go to https://www.mongodb.com/try/download/shell
2. Select:
   - **OS:** Windows
   - **Package:** .msi
3. Click **Download** and wait.
4. **Set the Path** (add the bin folder to Environment Variables).

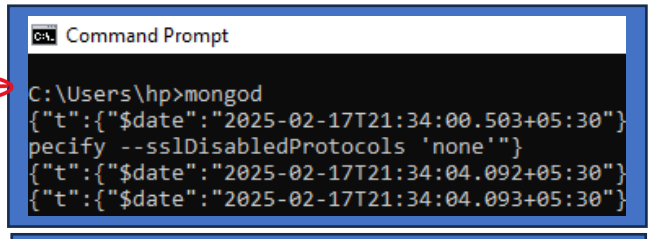


2. Verify Installation

- Open CMD and type:
- mongosh --version

3.How to Open MongoDB Server

1. Open **Command Prompt (CMD)**.
2. Type and run: → mongod



4.How to Open MongoDB Shell

1. Open **CMD**.
2. Type mongosh → press **Enter**.

5.How to check if MongoDB is

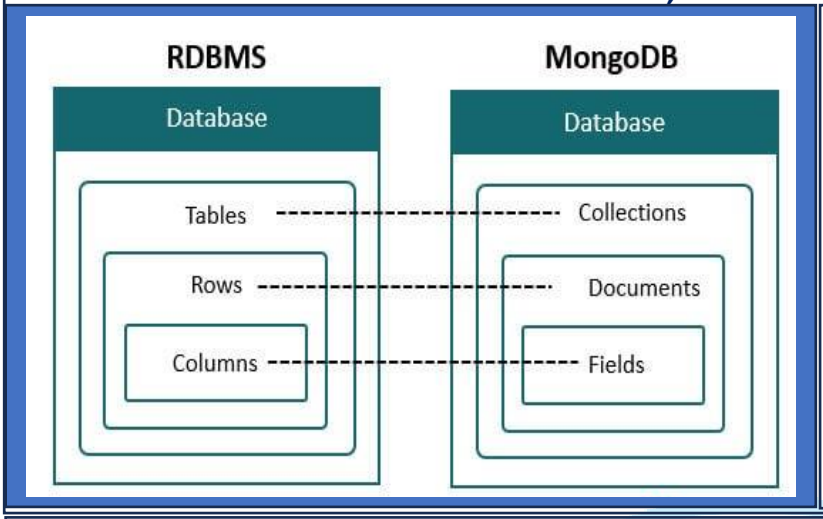
- open a new CMD and type: **sc query MongoDB**

6.Open MongoDB Shell (mongosh

Open a **new Command Prompt** and run this command: **mongosh**

MongoDB Basics

What are Databases, Collections, and Documents?



- **Database:** A database is a container for collections (like a main folder).
- **Collection:** A collection stores multiple documents (like a sub-folder).
- **Document:** A document is a single record stored in **JSON (BSON)** format.

Note: document stores **real values (actual data)** in the form of **key-value pairs** in MongoDB.

- **Database →** Holds collections (e.g., riddb)
- **Collection →** Holds documents (e.g., students)
- **Document →** Holds data in key-value format (e.g., student name, age, branch,etc)

JSON (BSON) format means:

- **JSON** → JavaScript Object Notation (simple text format to store data like key-value pairs).
- **BSON** → Binary JSON (MongoDB's internal format to store JSON data in binary form for faster reading and writing).

Example: Online Shopping App → When you buy something online, your order data is stored like this (in **JSON**):

```
{
  "orderId": 101,
  "customer": "Sangam Kumar",
  "items": ["Laptop", "Mouse"],
  "totalAmount": 55000,
  "paymentStatus": "Paid"
}
```

This JSON data looks simple and readable to humans.
But when MongoDB saves it in the database, it converts it into **BSON (Binary JSON)** — which computers can read and process

Basic MongoDB Queries

1. **How to Create / Switch Database** :**Syntax:** use <database_name> **Example :** use riddb
2. **How to Show all Databases** : **Syntax:** show databases or **Syntax:** show dbs
3. **How to check current DB** : **Syntax:** db :- it will show current used database
4. **How to Create Collection** :**Syntax:** db.createCollection("<collection_name>") **Ex:** db.createCollection("students")
5. **How to Show Collections** :**Syntax:** show collections
6. **How to Insert One Document** :**Syntax:** db.<collection_name>.insertOne({key: value, key: value})
   :Example: db.students.insertOne({ "name": "Sangam Kumar","age": 18})
7. **How to Insert Multiple Documents** :**Syntax:** db.<collection_name>.insertMany([{...}, {...}])
   Example: db.students.insertMany([{"name": "Sangam", "age": 20},
   {"name": "Ankit", "age": 22}])
8. **How to View all Data** **Syntax:** db.<collection_name>.find() **Example:** db.students.find()

Note-1: pretty() method is used to **display output of find() in a more readable (formatted) Ex: db.students.find().pretty()**

Note-2: MongoDB switches to a database named riddb, but it doesn't physically create it yet.
A database is only created after it has at least one collection and one document inside.

Database Command in MongoDB

1. Create / Use Database → Used to create a new database or switch to an existing one.

- **Syntax:** use <database_name>
- **Example:** use skills

2. Show All Databases → **v**Displays all databases that contain data.

- **Syntax:** show dbs
- **Example:** show dbs

3. Check Current Database → Shows the name of the currently active database.

- **Syntax:** db
- **Example:** db

4. Delete Database → Deletes the current database permanently.

- **Syntax:** db.dropDatabase()

Steps to Delete a Database

1. **Switch to the database you want to delete** → **Example:** use skills→ This makes skills the current database
2. **Drop (delete) the current database** →**Example: db.dropDatabase()**

Notes: db.dropDatabase() always deletes **the current database**. If you try show dbs after deleting, you'll see it's gone:

5. How to Rename a Database → In MongoDB, there is no direct command like RENAME DATABASE —

- but you can rename a database manually by copying data to a new database and then deleting the old one.

Step 1: Switch to the Old Database **Example:** use oldDB → Replace oldDB with your current database name.

Step 2: Copy Each Collection to the New Database

Syntax: db.<collection_name>.copyTo("<new_database>.<new_collection_name>")

Example:
```
use oldDB
db.students.copyTo("newDB.students")
db.teachers.copyTo("newDB.teachers")
```

Note: This creates a new database newDB and copies all documents.

Step 3: Verify the New Database → Switch to the new database and check collections:

Example: - use newDB

Example show collections → You'll see all copied collections there

Step 4: Delete the Old Database

Example: use oldDB

Example: db.dropDatabase() → This removes the old database.

Notes:

- There is **no built-in "rename database"** command in MongoDB.
- copyTo() works **only for non-sharded collections**.
- For large databases, use **MongoDB tools**:
  - **mongodump** → backup old database
  - **mongorestore** → restore with new name

Example using tools (optional):

- mongodump --db oldDB --out ./backup/
- mongorestore --db newDB ./backup/oldDB/

Data Types

Data types is a type of data stored in each field of a document, like string, number etc.

No.	Data Type	Used For	Example
1	**String**	Text values	{ "name": "Sangam Kumar" }
2	**Number**	Integers/Decimals	{ "age": 20, "salary": 50000.50 }
3	**Boolean**	True / False	{ "isActive": true }
4	**Array**	List of values	{ "skills": ["Python", "MongoDB"] }
5	**Object**	Nested document	{ "address": { "city": "Delhi", "pin": 110001 } }
6	**ObjectId**	Unique ID	{ "_id": ObjectId("65e1f3b7a1f9c3d8e8a01234") }
7	**Date**	Date & time	{ "createdAt": ISODate("2024-02-17T10:00:00Z") }
8	**Null**	Empty / missing value	{ "deletedAt": null }
9	**Binary Data**	Files or images	{ "profilePic": BinData(0, "image_data") }
10	**Regular Expression**	Pattern matching	{ "email": { "$regex": "@gmail.com$" } }

MongoDB supports **text, numbers, lists, objects, dates, files, and patterns** — all stored as **key-value pairs** in documents

Collection Commands in MongoDB

A collection in MongoDB is a group of related documents stored together within a database, similar to a table in SQL.

1. How to Create Collection :- Used to create a new collection in the current database.

- **Syntax:** db.createCollection("<collection_name>")
- **Example:** db.createCollection("students")

2. How to create the multiple collections :- Used to create **more than one collection** in the same database

- **Syntax:** db.createCollection("<collection_name>")
- **Example:**
```
db.createCollection("students")
db.createCollection("teachers")
db.createCollection("courses")
db.createCollection("departments")
```

Note: - this will create four collections: **students**, **teachers**, **courses**, and **departments** in the current database.

3. How to Show All Collections: - Displays all collections available in the current database.

- **Syntax:** show collections **Example:** show collections

4. How to Drop Collection:- Deletes a specific collection permanently.

- **Syntax:** db.<collection_name>.drop()
- **Example:** db.students.drop()

5. How to Delete Multiple Collections:- MongoDB does **not have a single command** to delete all collections at once, but you can do it in **two simple ways**

Method 1: Delete Collections One by One: - Use the drop() command for each collection.

Example:
```
db.students.drop()
db.teachers.drop()
db.courses.drop()
```
Note: Each command permanently deletes the specified collection.

Method 2: Delete All Collections Using a Loop :- Run this JavaScript command in the MongoDB shell:

```
db.getCollectionNames().forEach(function(coll) {
  db[coll].drop();  });
```

Meaning:

- db.getCollectionNames() → gets all collection names.
- .forEach() → loops through each collection.
- db[coll].drop() → deletes each one. **Note:** This will **delete all collections** in the current database.

Method 3: Drop the Whole Database (Deletes All Collections Together)

- If you want to remove all collections **and** the database:
- db.dropDatabase() This command permanently deletes the **entire database** and all its collections.

4.How to Insert Single Document into Collection

: - Adds a new record/document to a collection.

- **Syntax:** db.<collection_name>.insertOne({key:value})
- **Example:** db.students.insertOne({name: "Raj", age: 20})

5. How to Insert Multiple Documents into a Collection: - Used to add more than one record/document at a time.

Syntax: db.<collection_name>.insertMany([{key:value}, {key:value}, ...])

Example:
```
db.students.insertMany([
{name: "Raj", age: 20},
{name: "Amit", age: 22},
 {name: "Priya", age: 19}
 ])
```

6.How to Display Documents (Find All) :-Shows all documents in a collection.

- **Syntax:** db.<collection_name>.find()
- **Example:** db.students.find()

7. How to Display in Readable Format :-Shows documents in formatted output.

- **Syntax:** db.<collection_name>.find().pretty()
- **Example:** db.students.find().pretty()

8. How to Find Specific Documents (with condition):- Displays only matching documents.

- **Syntax:** db.<collection_name>.find({key:value})
- **Example:** db.students.find({name:"Raj"})

9. How to Update Document:- Updates existing document data.

- **Syntax:** db.<collection_name>.updateOne({filter}, {$set:{key:new_value}})
- **Example:** db.students.updateOne({name:"Raj"}, {$set:{age:21}})

10. How to Update Multiple Documents

- **Syntax:** db.<collection_name>.updateMany({filter}, {$set:{key:value}})
- **Example:** db.students.updateMany({}, {$set:{status:"Active"}})

11. How to Delete Single Document:- Removes one document from a collection.

- **Syntax:** db.<collection_name>.deleteOne({filter})
- **Example:** db.students.deleteOne({name:"Raj"})

12. How to Delete Multiple Documents:- Removes all documents matching the condition.

- **Syntax:** db.<collection_name>.deleteMany({filter})
- **Example:** db.students.deleteMany({age:{$lt:18}})

13. How to Count Documents:- Returns the number of documents in a collection.

- **Syntax:** db.<collection_name>.countDocuments()
- **Example:** db.students.countDocuments()

14. How to Rename Collection:- Renames a collection to a new name.

- **Syntax:** db.<old_name>.renameCollection("<new_name>")
- **Example:** db.students.renameCollection("learners")

CRUD Operations
(Create, Read, Update, Delete)

1.Create

create operation is used to make a **new database** or **collection** for storing data

1. Create and Insert Operations in MongoDB

Step 1: Create/Select Database

- Use or create a database.
  Syntax: use myDatabase

Step 2: Create/Select Collection

- Create or select a collection to store data.
  Syntax: db.createCollection("users") Or simply: db.users

Step 3: Insert Operations

❖ **insertOne() – Insert Single Document** → Used to add one record.

Example:

```
db.users.insertOne({
  name: "Sangam Kumar",
  age: 30,
  address: "SSM Bihar"
})
```

❖ **insertMany() – Insert Multiple Documents**

Used to add many records at once.

Example:

```
db.users.insertMany([
  { name: "Amit Kumar", age: 28, address: "Mumbai" },
  { name: "Priya Sharma", age: 35, address: "Delhi" },
  { name: "Ravi Patel", age: 25, address: "Bangalore" }
])
```

Key Concepts – Create Operation in MongoDB

1. **Purpose:** - The *create operation* is used to make a **new database** or **collection** for storing data.
2. **Automatic Creation:**
   - In MongoDB, databases and collections are **automatically created** when you insert data for the first time.
3. **Manual Creation:** You can also manually create a collection using:
   Example: db.createCollection("collection_name")
4. **Create Database:** Use the use command to create or switch to a database.
   Example: use myDatabase
5. **Structure:** MongoDB stores data as **documents** inside **collections**, and collections inside **databases**.
6. **Flexible Schema:**
   - Collections in MongoDB do **not require a fixed structure** — each document can have different fields.
7. **Verification Commands:**
   - To check databases → show dbs
   - To check collections → show collections

Operators in Mongo dB

- Operators in MongoDB are **special symbols or keywords** used to perform specific actions in queries, such as comparing values, filtering data, updating fields, or performing logical operations.

❖ **Why We Use Operators:** Operators are used to **control and refine queries** in MongoDB.

- They help in **searching**, **comparing**, **updating**, and **filtering** documents based on certain conditions.

❖ **Main Types of Operators:**

- **Comparison Operators** → Compare field values (e.g., $eq, $gt, $lt).
- **Logical Operators** → Combine multiple conditions (e.g., $and, $or, $not).
- **Element Operators** → Check for the presence of fields (e.g., $exists, $type).
- **Update Operators** → Modify field values (e.g., $set, $inc, $unset).
- **Array Operators** → Work with array data (e.g., $push, $pull, $elemMatch).

Operator	Description	Example
$eq	Matches values that are equal to a specified value.	db.college.find({ fee: { $eq: 40000 } })
$ne	Matches values that are not equal to a specified value.	db.college.find({ fee: { $ne: 40000 } })
$gt	Matches values that are greater than a specified value.	db.college.find({ fee: { $gt: 40000 } })
$gte	Matches values that are greater than or equal to a specified value.	db.college.find({ fee: { $gte: 40000 } })
$lt	Matches values that are less than a specified value.	db.college.find({ fee: { $lt: 40000 } })
$lte	Matches values that are less than or equal to a specified value.	db.college.find({ fee: { $lte: 40000 } })
$in	Matches values that are in an array.	db.college.find({ city: { $in: ["Delhi", "Mumbai"] } })
$nin	Matches values that are not in an array.	db.college.find({ city: { $nin: ["Delhi", "Mumbai"] } })
$and	Joins multiple conditions, all of which must be true.	db.college.find({ $and: [{ city: "Delhi" }, { fee: { $gt: 40000 } }] })
$or	Joins multiple conditions, at least one of which must be true.	db.college.find({ $or: [{ city: "Delhi" }, { city: "Mumbai" }] })
$nor	Joins multiple conditions, none of which must be true.	db.college.find({ $nor: [{ city: "Delhi" }, { fee: { $lt: 35000 } }] })
$not	Inverts the effect of another operator.	db.college.find({ fee: { $not: { $gt: 40000 } } })
$exists	Checks if a field is present or absent.	db.college.find({ address: { $exists: true } })
$type	Matches documents where the field's value is of a specific type.	db.college.find({ fee: { $type: "int" } })
$regex	Matches documents where the field value matches a regular expression.	db.college.find({ name: { $regex: "^A" } })
$text	Performs a text search on string content indexed for full-text search.	db.college.find({ $text: { $search: "Computer Science" } })
$ifNull	Returns the first value that is not null or undefined.	db.college.find({ $project: { name: { $ifNull: ["$name", "Unknown"] } } })
$size	Matches arrays with a specific size.	db.college.find({ courses: { $size: 3 } })
$all	Matches arrays that contain all specified values.	db.college.find({ courses: { $all: ["Math", "Science"] } })
$elemMatch	Matches documents where an array field contains at least one element matching a query.	db.college.find({ courses: { $elemMatch: { course_name: "Math", fee: { $gt: 1000 } } } })
$set	Sets the value of a field during an update.	db.college.updateOne({ sid: "S101" }, { $set: { fee: 45000 } })

$inc	Increments a field by a specified value.	db.college.updateOne({ sid: "S101" }, { $inc: { fee: 5000 } })
$push	Adds an element to an array during an update.	db.college.updateOne({ sid: "S101" }, { $push: { courses: "Physics" } })
$pull	Removes an element from an array during an update.	db.college.updateOne({ sid: "S101" }, { $pull: { courses: "Physics" } })
$unset	Removes a field during an update.	db.college.updateOne({ sid: "S101" }, { $unset: { address: "" } })
$near	Finds documents near a specified geo-location.	db.college.find({ location: { $near: [40.7128, 74.0060] } })
$geoWithin	Matches documents within a specified shape (e.g., circle, polygon).	db.college.find({ location: { $geoWithin: { $centerSphere: [[40.7128, 74.0060], 10] } } })

2.Read Operations

2. Read Operations in MongoDB :- Used to **retrieve data** from a collection.

Main Methods:

1. **find()** → Returns **multiple documents**.
2. **findOne()** → Returns **a single document**.
3. **pretty()** → Displays output in a **readable format**.

◇ **find() – Retrieve Multiple Documents**

- Used to fetch all or specific documents.
- **Syntax:** db.collectionName.find(query, projection)
  - query: Condition to filter data (optional).
  - projection: Select specific fields (optional).
- **Example:** db.users.find({ age: { $gt: 18 } }, { name: 1, age: 1 })

→ Shows name and age of users older than 18.

◇ **findOne() – Retrieve Single Document**

- Fetches only **one matching document**.
- **Example:** db.users.findOne({ name: "Amit" })

◇ **pretty() – Format Output** Makes data easier to read.

- **Example:** db.users.find().pretty()

Using 0 and 1 in Projection

- 1 → **Show** the field
- 0 → **Hide** the field
- You **can't mix** 1 and 0 together (except for _id).
- _id can be hidden separately.

Example-1: db.students.find({}, { name: 1, age: 1, _id: 0 }) Note: Shows only **name** and **age**, hides **_id**.

Example 2 – Show selected fields only Example: db.employees.find({}, { name: 1, department: 1, _id: 0 })

Note: Shows only **name** and **department**, hides _id.

Example 3 – Hide specific fields Ex: db.students.find({}, { address: 0, phone: 0 })

Note: Shows all fields **except** address and phone.

Example 3 – Show fields with a condition Ex: db.products.find({ price: { $lt: 1000 } }, { name: 1, price: 1, _id: 0 })

Note: Shows **name** and **price** of products where **price < 1000**, hides _id.

Example: // Connect to the MongoDB database >> mongod & mongosh press in cmd

```
use myDatabase; // Replace with your database name
// Create the collection 'college' and insert 10 data entries with Indian data
db.college.insertMany([
 {
  name: "Ravi Kumar",
  sid: "S101",
  fee: 35000,
  course: "Computer Science",
  city: "Delhi",
  address: "123 Connaught Place"
 },

 {
  name: "Arjun Patel",
  sid: "S103",
  fee: 38000,
  course: "Electrical Engineering",
  city: "Bangalore",
  address: "34 MG Road"
 },

 {
   name: "Priya Sharma",
   sid: "S102",
   fee: 32000,
   course: "Mechanical Engineering",
   city: "Mumbai",
   address: "45 Juhu Beach Road"
 },

{
 name: "Ananya Gupta",
 sid: "S104",
 fee: 34000,
 course: "Civil Engineering",
 city: "Chennai",
 address: "23 Anna Nagar"
},
```

1. Retrieve all documents in the collection:

- This query will retrieve all the documents from the "college" collection.
  Syntax: db.college.find() **Example:** demo> db.college.find(

2. Retrieve students who are enrolled in "Computer Science"

- This query retrieves students whose course is "Computer Science".
  Syntax: db.college.find({ course: "Computer Science" })

3. Retrieve students from "Delhi" or "Mumbai"

- This query retrieves students who are from either Delhi or Mumbai.
- **Syntax:** db.college.find({ city: { $in: ["Delhi", "Mumbai"] } })

4. Retrieve students with a fee greater than ₹35,000

- This query retrieves students whose fee is greater than ₹35,000.
- **Syntax:** db.college.find({ fee: { $gt: 35000 } })

5. Retrieve students whose names start with "A"

- This query retrieves students whose names begin with the letter "A" (case-insensitive).
- **Syntax:** db.college.find({ name: /^A/i }) (i flag (which makes it case-insensitive)
- **Syntax:** db.college.find({ name: /^A/ }) (case-sensitive)

1. Retrieve students who have a fee less than ₹40,000

- This query will return students whose fee is less than ₹40,000.
- **Syntax:** db.college.find({ fee: { $lt: 40000 } })

7. Retrieve students from a specific city, "Bangalore"

- This query retrieves all the students who are from "Bangalore."

- **Syntax:** db.college.find({ city: "Bangalore" })

8.Retrieve students who are enrolled in either "Mechanical Engineering" or "Law"

- this query will return students whose course is either "Mechanical Engineering" or "Law."
- **Example:** db.college.find({ course: { $in: ["Mechanical Engineering", "Law"] } })

9. Retrieve students whose fee is between ₹30,000 and ₹40,000

- This query retrieves students whose fee is between ₹30,000 and ₹40,000.
- **Example:** db.college.find({ fee: { $gte: 30000, $lte: 40000 } })

10. **Retrieve students whose name ends with "a"**

- This query retrieves students whose name ends with the letter "a" (case-insensitive).
- **Example:** db.college.find({ name: /a$/i })

2. findOne() - Retrieve a Single Document

- The findOne() method is used to retrieve a **single document** from a collection. It returns the **first document** that matches the specified condition. **Syntax:** db.collectionName.findOne(query, projection);

Q1. Find a single user by name.

- db.college.findOne({ name: " Rohit Yadav" });

This will return the first user document with the name "Amit Kumar".

Q2. Find the all data Based on name condition.

- db.college.find({name: " Rohit Yadav" })

This will return the all user document with the name "Amit Kumar".

Q3. Find a user and project only specific fields.

- If you want to retrieve the name and address of a user:
- db.college.findOne({ name: "Priya Sharma" }, { name: 1, address: 1 });

This will return the name and address fields for the user named "Priya Sharma".

Output: { _id: ObjectId('67cc3bffb7f27896e74d7943'),
name: 'Priya Sharma',
address: '45 Juhu Beach Road' }

Q4. How to print all name only from college: test3> db.college.find({}, { name: 1, _id: 0 });

Explanation: {} → This means no filter, so it retrieves all documents.
{ name: 1, _id: 0 } → This ensures that only the name field is displayed and _id is excluded.

Q5. How to print the name and fee from the college collection

test3> db.college.find({}, { name: 1, fee: 1, _id: 0 });

3. pretty() - Format the Output for Readability

- The pretty() method is used to make the output of find() more readable. By default, find() returns documents in a compact form. When you chain the pretty() method, it formats the output in a more human-readable format with indentation.

Syntax: db.collectionName.find().pretty();

Example: Find all users and display the results in a readable format.

- db.college.find().pretty();

This will output all documents in the users collection with better formatting.

Example: Use pretty() with a query.

- If you want to find users who are older than 30 and display the results in a readable format:
- db.users.find({ age: { $gt: 30 } }).pretty();

1. **Using findOne() to retrieve a single document for a specific student (e.g., with SID = "S101")**
   - This query returns a single document for the student with SID "S101".
   - **demo>** db.college.findOne({ sid: "S101" })
2. **Using find() to get students with a fee greater than ₹40,000**
   - This query retrieves students whose fee is greater than ₹40,000.
     db.college.find({ fee: { $gt: 40000 } })
3. **Using find() and pretty() to format the output for better readability**
   - retrieves all students from the collection and formats result using pretty() for improved readability.
   - db.college.find().pretty()

6. Using findOne() to retrieve a student with the name "Ravi Kumar"
- This query returns a single document for the student whose name is "Ravi Kumar."
- **Example:** db.college.findOne({ name: "Ravi Kumar" })

7.Find students who are taking a course in either "Computer Science" or "Electronics"
- db.college.find({ course: { $in: ["Computer Science", "Electronics"] } });

8. Find students who are not from "Delhi"
- db.college.find({ city: { $ne: "Delhi" } });

9. Find students whose fee is between 30000 and 50000 (inclusive)
- db.college.find({ fee: { $gte: 30000, $lte: 50000 } });

10. Find students whose sid is missing (i.e., doesn't exist)
- db.college.find({ sid: { $exists: false } });

11. Find students who are from "Mumbai" and have a fee less than 50000
- db.college.find({ $and: [{ city: "Mumbai" }, { fee: { $lt: 50000 } }] });

12. Find students who are either from "Pune" or "Bangalore"
- db.college.find({ $or: [{ city: "Pune" }, { city: "Bangalore" }] });

13. Find students whose name starts with "A"
- db.college.find({ name: { $regex: "^A", $options: "i" } });

($options: "i" makes it case-insensitive)

14. Find students who are not enrolled in "Mechanical Engineering"
- db.college.find({ course: { $not: { $eq: "Mechanical Engineering" } } });

15. Find students who have fee not equal to 40000 and are from "Hyderabad"
- db.college.find({ $and: [{ fee: { $ne: 40000 } }, { city: "Hyderabad" }] });

16. Print only the name of all students
- **db.college.find({}, { name: 1, _id: 0 });**

17. Print name and city of students who have a fee greater than 40,000
- db.college.find({ fee: { $gt: 40000 } }, { name: 1, city: 1, _id: 0 });

18. Print only name of students enrolled in "Computer Science"
- db.college.find({ course: "Computer Science" }, { name: 1, _id: 0 });

19. Print name and city of students from "Mumbai"
- db.college.find({ city: "Mumbai" }, { name: 1, city: 1, _id: 0 });

20. Print only name of students who are not from "Delhi"
- db.college.find({ city: { $ne: "Delhi" } }, { name: 1, _id: 0 });

21. Print name and city of students whose fee is between 30,000 and 50,000
- db.college.find({ fee: { $gte: 30000, $lte: 50000 } }, { name: 1, city: 1, _id: 0 });

3.UPDATE OPERATIONS

(updateOne(), updateMany(), $set, $unset)

1. Creating the Collection and Adding 10 documents

```
db.createCollection("students");
db.students.insertMany([
 { name: "Arjun", age: 20, grade: "A", city: "Delhi", nationality: "Indian" },
 { name: "Priya", age: 22, grade: "B", city: "Mumbai", nationality: "Indian" },
  ..... ]);
```

2. Performing Update Operations

❖ **UpdateOne() Operation:** this method updates **a single document** based on condition.

- You can use $set to modify specific fields.

Syntax:
```
db.<collection_name>.updateOne(
  { <filter_condition> },          // Find the document(s) to update
  { $set: { <field>: <value> } } )  // Specify the field(s) and new value(s)
```

Example: Update the grade of the student named "Arjun" to "A+".

```
db.students.updateOne(
 { name: "Arjun" },          // Filter: Find the student named "Arjun"
 { $set: { grade: "A+" } } )  // Update: Set the grade field to "A+"
```

❖ **updateMany(): This** method updates **multiple documents** that match a given condition.

- **Syntax:**
```
db.<collection_name>.updateMany(
  { <filter_condition> },          // Condition to match multiple documents
  { $set: { <field>: <value> } } )  // Fields to update with new values
```

Example: Update the city of all students who have grade "B" to "New Delhi".

```
db.students.updateMany(
 { grade: "B" },                  // Filter: Find students with grade "B"
 { $set: { city: "New Delhi" } }  // Update: Set the city field to "New Delhi");
```

❖ **$set Operator:** The $set operator is used to **modify or add fields** in a document.

Example: Add a new field status and set it to "Active" for the student "Priya".

```
db.students.updateOne(
 { name: "Priya" },              // Filter: Find the student named "Priya"
 { $set: { status: "Active" } }  // Update: Add the field "status" with value "Active" );
```

❖ **$unset Operator:** this is used to **remove fields** from a document.

Example: Remove the status field from the student "Neha".

```
db.students.updateOne(
 { name: "Neha" },           // Filter: Find the student named "Neha"
 { $unset: { status: "" } } )  // Update: Remove the "status" field
```

❖ **Scenarios Based Example**

- **Scenario 1: Set the Grade of All Students in "Mumbai" to "A"**

```
db.students.updateMany(
 { city: "Mumbai" },              // Filter: Find students from Mumbai
 { $set: { grade: "A" } } );      // Update: Set their grade to "A"
```

- **Scenario 2: Remove the nationality field for students from "Pune"**

```
db.students.updateMany(
  { city: "Pune" },                    // Filter: Find students from Pune
  { $unset: { nationality: "" } } )    // Update: Remove the "nationality" field
```

- **Scenario 3: Update the Age of the Student "Vikram" to 24**

```
db.students.updateOne(
  { name: "Vikram" },                  // Filter: Find the student named "Vikram"
  { $set: { age: 24 } } )              // Update: Set the age to 24
```

- **Scenario 4: Set the status to "Graduated" for all students with Grade "A"**

```
db.students.updateMany(
  { grade: "A" },                      // Filter: Find students with grade "A"
  { $set: { status: "Graduated" } } )  // Update: Set their status to "Graduated"
```

- **Scenario 5: Add a New Field phoneNumber to the Student "Raj"**

```
db.students.updateOne(
  { name: "Raj" },
  { $set: { phoneNumber: "1234567890" } } )
```

- **Scenario 6: Remove the city field for all students with Grade "C"**

```
db.students.updateMany(
  { grade: "C" },                      // Filter: Find students with grade "C"
  { $unset: { city: "" } })            // Update: Remove the "city" field
```

- **Scenario 7: Update the City of All Students in "Kolkata" to "New Kolkata"**

```
db.students.updateMany(
  { city: "Kolkata" },                 // Filter: Find students from Kolkata
  { $set: { city: "New Kolkata" } } )  // Update: Set their city to "New Kolkata"
```

- **Scenario 8: Increase the Age of Students in "Delhi" by 1 Year**

```
db.students.updateMany( { city: "Delhi" }, { $inc: { age: 1 } }  )
```

- **Scenario 9: Update the Grade of "Neha" to "B" and City to "Chennai"**

```
db.students.updateOne( { name: "Neha" }, { $set: { grade: "B", city: "Chennai" } })
```

Scenario 10: Set the Nationality to "Indian" for All Students Who Don't Have a Nationality Field

```
db.students.updateMany(
  { nationality: { $exists: false } }, { $set: { nationality: "Indian" } }  );
```

Note:

- **updateMany()**: Used when you want to apply an update to multiple documents based on a filter.
- **updateOne()**: Used when you want to update a single document based on a filter.
- **$set**: Used to set or update fields with specific values.
- **$unset**: Used to remove fields from a document.
- **$gt**: Used to filter for documents where a field's value is greater than a specified value.
- **$inc**: Used to increment numeric field values by a specified amount.

4.DELETE

Delete One () And DeleteMany() Operation

1.Delete One ()

- deleteOne() method deletes a **single document** that matches the specified filter.
- **Syntax:** db.<collection_name>.deleteOne(
  { <filter_condition> } // Condition to match a single document
  );
- **Scenario 1: Delete a Student with the Name "Raj"**

```
db.students.deleteOne( { name: "Raj" } // Filter: Find the student named "Raj" );
```

Scenario 2: Delete the First Student from "Delhi"

```
db.students.deleteOne( { city: "Delhi" } // Filter: Find the first student from Delhi );
```

2.DeleteMany ()

This method is used to **delete all documents** from a collection that match the given condition (filter).

Syntax:

```
db.<collection_name>.deleteMany(
 { <filter_condition> }  // Condition to match multiple documents
);
```

2. deleteMany() Operation:

- The deleteMany() method deletes **multiple documents** that match the given filter.

Scenario 3: Delete All Students with Grade "C"

```
db.students.deleteMany(
 { grade: "C" } // Filter: Find all students with grade "C" );
```

Scenario 4: Delete All Students from "Bangalore"

```
db.students.deleteMany(
 { city: "Bangalore" } // Filter: Find all students from Bangalore );
```

3. drop() Operation:

- The drop() method is used to **delete an entire collection**.

Scenario 5: Drop the Entire "students" Collection

```
db.students.drop(); // Delete the entire "students" collection
```

4. deleteOne() - Delete a Student with Age Less Than 22

```
db.students.deleteOne({ age: { $lt: 22 } } );
```

5. deleteMany() - Delete All Students Who Do Not Have a phoneNumber Field

```
db.students.deleteMany( { phoneNumber: { $exists: false } } );
```

6. deleteOne() - Delete a Student with a Specific Enrollment Date

```
db.students.deleteOne( { enrollmentDate: "2025-02-21" } enrollment date );
```

7. deleteMany() - Delete All Students with a Grade Lower Than "B"

```
db.students.deleteMany( { grade: { $lt: "B" } } );
```

8.deleteMany() - Delete All Students with the Nationality "Indian" from "Chandigarh"

```
db.students.deleteMany( { nationality: "Indian", city: "Chandigarh" } );
```

9. deleteOne() - Delete the Student with the Name "Manisha" and Age 21

```
db.students.deleteOne( { name: "Manisha", age: 21 } );
```

❖ How to delete a single field from a collection

- To delete a single field (e.g., **`city`**) from all documents in the `student's` collection, use the **`$unset`** operator.

Q1. Query to Remove the city Field from All Documents:

db.students.updateMany({}, { $unset: { city: 1 } })

Q2. Query to Remove the city Field from a Specific Document (e.g., Arjun):

db.students.updateOne({ name: "Arjun" }, { $unset: { city: 1 } })

Q3. Query to Remove the city Field from Multiple Matching Documents (e.g., All Students from "Mumbai" or "Delhi"):

db.students.updateMany({ city: { $in: ["Mumbai", "Delhi"] } }, { $unset: { city: 1 } })

Q4. Query to Remove `city` and `grade` Fields from All Documents.

db.students.updateMany({}, { $unset: { city: 1, grade: 1 } })

Q5. Query to Remove `age` and `nationality` Fields from a Specific Document (e.g., Arjun):

db.students.updateOne({ name: "Arjun" }, { $unset: { age: 1, nationality: 1 } })

Q6. Query to Remove `city`, `age`, and `grade` Fields from Students with `age` Less Than 22

db.students.updateMany({ age: { $lt: 22 } }, { $unset: { city: 1, age: 1, grade: 1 } })

❖ Explanation

- **$unset:** { city: 1 } removes the city field.
- **updateMany**({}, { $unset: { city: 1 } }) **removes city from all documents.**
- **updateOne**({ name: "Arjun" }, { $unset: { city: 1 } }) **removes city only from Arjun's document.**

4.MONGODB QUERY OPERATORS

1. Comparison Operators ($eq, $ne, $gt, $gte, $lt, $lte)
2. Logical Operators ($and, $or, $not, $nor)
3. Element Operators ($exists, $type)
4. Evaluation Operators ($regex, $expr)
5. Array Operators ($all, $size, $elemMatch)
6. Projection ($include, $exclude)

Comparison Operators:

- **$eq:** Matches exact value
- **$ne:** Matches values not equal to a given value
- **$gt:** Finds values greater than a specified number
- **$gte:** Finds values greater than or equal
- **$lt:** Finds values less than a specified number
- **$lte:** Finds values less than or equal

Example:

Step 1: Create a collection and insert data

```
db.indians.insertMany([
{ name: "Amit Sharma", age: 30, city: "Delhi", salary: 50000 },
{ name: "Priya Verma", age: 28, city: "Mumbai", salary: 60000 },
{ name: "Rajesh Kumar", age: 35, city: "Bangalore", salary: 70000 },
{ name: "Anjali Singh", age: 25, city: "Kolkata", salary: 40000 },
{ name: "Suresh Patel", age: 40, city: "Ahmedabad", salary: 80000 } ]);
```

Step 2: Use Comparison Operators

Q1:-$eq (Equal) - Find users aged 30

```
db.indians.find({ age: { $eq: 30 } });
```

Q1:- $ne (Not Equal) - Find users not from Delhi

```
db.indians.find({ city: { $ne: "Delhi" } });
```

Q2: $gt (Greater Than) - Find users with salary greater than 50,000

```
db.indians.find({ salary: { $gt: 50000 } });
```

Q3. $gte (Greater Than or Equal) - Find users aged 35 or older

```
db.indians.find({ age: { $gte: 35 } });
```

Q4. $lt (Less Than) - Find users younger than 30

```
db.indians.find({ age: { $lt: 30 } });
```

Q5. $lte (Less Than or Equal) - Find users with salary 60,000 or less

```
db.indians.find({ salary: { $lte: 60000 } });
```

Q6.: $and (Logical AND) - Find users who are older than 30 AND have a salary greater than 50,000

```
db.indians.find({  $and: [ { age: { $gt: 30 } }, { salary: { $gt: 50000 } } ]  });
```

Q7: $or (Logical OR) - Find users who are from Delhi OR have a salary less than 50,000

```
db.indians.find({  $or: [ { city: "Delhi" }, { salary: { $lt: 50000 } } ]  });
```

Q8: $not (Logical NOT) - Find users who are NOT from Mumbai

```
db.indians.find({  city: { $not: { $eq: "Mumbai" } }  });
```

Q9: $nor (Logical NOR) - Find users who are NOT from Bangalore NOR have a salary greater than 70,000

```
db.indians.find({  $nor: [ { city: "Bangalore" }, { salary: { $gt: 70000 } } ] });
```

Q10.: $and (Logical AND) - Find users who are from Kolkata AND have a salary less than 50,000

```
db.indians.find({  $and: [ { city: "Kolkata" }, { salary: { $lt: 50000 } } ] });
```

Logical Operators ($and, $or, $not, $nor)

- **$and** - Ensures both conditions must be **true**.
- **$or** - Matches if **at least one** condition is **true**.
- **$not** - Negates the condition.
- **$nor** - Ensures **none** of the conditions are **true**.

Q1. $and - Find users who are older than 30 AND have a salary greater than 50,000

```
db.indians.find({   $and: [ { age: { $gt: 30 } }, { salary: { $gt: 50000 } } ]   });
```

➢ **Print only name**

```
db.indians.find({ $and: [ { age: { $gt: 30 } }, { salary: { $gt: 50000 } } ] }, { name: 1, _id: 0 }  );
// Projection: Include name, exclude _id
```

➢ **Print the name and age only**

```
db.indians.find({ $and: [ { age: { $gt: 30 } }, { salary: { $gt: 50000 } } ] }, { name: 1, age:1, _id: 0 }  );
```

Q2. $or - Find users who are from Delhi OR have a salary less than 50,000

```
db.indians.find({  $or: [ { city: "Delhi" }, { salary: { $lt: 50000 } } ]  });
```

Q3. $not - Find users who are NOT from Mumbai

```
db.indians.find({  city: { $not: { $eq: "Mumbai" } } });
```

Q4. $nor - Find users who are NOT from Bangalore NOR have a salary greater than 70,000

```
db.indians.find({  $nor: [ { city: "Bangalore" }, { salary: { $gt: 70000 } } ]  });
```

Q5. $and - Find users who are from Ahmedabad AND have a salary greater than 60,000

```
db.indians.find({   $and: [ { city: "Ahmedabad" }, { salary: { $gt: 60000 } } ]   });
```

Q6. $or - Find users who are from Kolkata OR have an age greater than 35

```
db.indians.find({  $or: [ { city: "Kolkata" }, { age: { $gt: 35 } } ]   });
```

Q7. $not - Find users whose salary is NOT less than 50,000

```
db.indians.find({  salary: { $not: { $lt: 50000 } }   });
```

Q8. $nor - Find users who are NOT older than 40 NOR from Delhi

```
db.indians.find({   $nor: [ { age: { $gt: 40 } }, { city: "Delhi" } ]   });
```

Q9. $and - Find users who are from Mumbai AND have a salary between 50,000 and 70,000

```
db.indians.find({  $and: [ { city: "Mumbai" }, { salary: { $gte: 50000, $lte: 70000 } } ]   });
```

Q10. $or - Find users who are younger than 25 OR have a salary greater than 80,000

```
db.indians.find({   $or: [ { age: { $lt: 25 } }, { salary: { $gt: 80000 } } ]  });
```

- `{ name: 1, age: 1 }` → Includes only the `name` and `age` fields.
- `{ _id: 0 }` → Excludes `_id` (because MongoDB includes it by default).

❖ Element Operators ($exists, $type)

- **$exists** checks if a field is present or missing in a document.
- **$type** checks if a field's data type matches the expected type.

Q1. $exists - Find users who have a salary field

```
db.indians.find({ salary: { $exists: true } });
```

Q2. $exists - Find users who do NOT have a city field

```
db.indians.find({ city: { $exists: false } });
```

Q3. $type - Find users where the salary is stored as a number

```
db.indians.find({ salary: { $type: "number" } });
```

Q4. $type - Find users where the name field is stored as a string

```
db.indians.find({ name: { $type: "string" } });
```

❖Evaluation Operators ($regex, $expr)

Q1. $regex - Find users whose names start with 'A'

```
db.indians.find({ name: { $regex: "^A", $options: "i" } });
```

Explanation:

- **$regex: "^A"** → Finds names that **start** with the letter 'A'.
- **"^"** → Indicates the start of the string.
- **$options: "i"** → Makes the search **case-insensitive** (e.g., matches "Amit", "amit", "Anjali", etc.).

Q2. $regex - Find users whose city name contains 'pur' (e.g., Jaipur, Nagpur)

```
db.indians.find({ city: { $regex: "pur", $options: "i" } });
```

Explanation:

- **$regex: "pur"** → Finds cities that **contain** the text "pur" anywhere.
- **$options: "i"** → Makes it **case-insensitive** (matches "PUR", "Pur", "pur")

Explanation:

- **$expr** → Allows operations on **fields within a document**.
- **$gt: ["$age", 30]** → Checks if age is **greater than 30**.
- **$lt: ["$age", 40]** → Checks if age is **less than 40**.
- **$and** → Ensures **both** conditions are met.

Q3. $expr - Find users where age is greater than 30 and less than 40

```
db.indians.find({ $expr: { $and: [ { $gt: ["$age", 30] }, { $lt: ["$age", 40] } ] } });
```

Explanation:

- **$gt:** ["$salary", { $multiply: ["$age", 2] }]
  → Compares **salary** with **twice the age**.
- **$multiply: ["$age", 2]** → Multiplies age × 2.

Q4. $expr - Find users whose salary is greater than twice their age

```
db.indians.find({ $expr: { $gt: ["$salary", { $multiply: ["$age", 2] }] } });
```

❖ Array Operators ($all, $size, $elemMatch)

❖ Array Operators ($all, $size, $elemMatch)

Step 1: Modify the Collection to Include an Array Field (languages and work Experience)

- Before we perform **array operations**, we need to add some array fields to our collection:

```
db.indians.updateMany({}, {
  $set: {
    languages: ["Hindi", "English"],
    workExperience: [
      { title: "Engineer", salary: 50000 },
      { title: "Manager", salary: 70000 }
    ]  }  });
```

Q1. $all - Find users who speak both "Hindi" and "English"

```
db.indians.find({ languages: { $all: ["Hindi", "English"] } });
```

Explanation: Finds users who **speak both** "Hindi" and "English", irrespective of order.

Q2. $size - Find users who know exactly 2 languages

```
db.indians.find({ languages: { $size: 2 } });
```

Explanation: Matches users where the languages array has **exactly** 2 elements.

Q3. $elemMatch - Find users who have worked as a Manager with a salary greater than 60,000

```
db.indians.find({
  workExperience: {
    $elemMatch: { title: "Manager", salary: { $gt: 60000 } }
  }  });
```

Explanation:

- **$elemMatch** checks for a **specific condition inside an array**.
- Finds users who have at least **one job as "Manager" with a salary > 60,000**.

Projection ($include, $exclude)

1. **$include - Find users from Mumbai and display only name and age (excluding _id)**

```
db.indians.find({ city: "Mumbai" }, { name: 1, age: 1, _id: 0 });
```

Explanation:

- **Includes** only name and age.
- **Excludes** _id field.

Q5. $exclude - Find users but exclude salary field

```
db.indians.find({}, { salary: 0 });
```

Explanation:

Excludes **only** salary field from results. Other fields will be displayed

❖ Summary:

Operator	Purpose
1. **$all**	Matches **all specified values** in an array.
2. **$size**	Matches **arrays of exact length**.
3. **$elemMatch**	Filters **nested arrays** based on conditions.
4. **$include (1)**	**Includes** only selected fields in results.
5. **$exclude (0)**	**Excludes** specified fields from results.

5. Indexing in MongoDB

What is Indexing? Indexing in MongoDB helps to **speed up query performance** by quickly finding documents, just like an index in a book.

Benefits of Indexing

- Faster query execution
- Reduces data scanning
- Improves sorting
- Ensures unique values

Types of Indexes

1. **Single Field Index** – On one field
2. **Compound Index** – On multiple fields
3. **Multikey Index** – On array fields
4. **Text Index** – For full-text search
5. **Hashed Index** – For fast equality search & sharding

Creating Indexes

- **Syntax:**db.collection.createIndex({ field: 1 }) // 1 = Ascending, -1 = Descending
- **Example:**db.students.createIndex({ name: 1 })
- **Check indexes:**db.students.getIndexes()

Examples

1. **Ascending Index:** db.students.createIndex({ name: 1 })
2. **Descending Index:** db.students.createIndex({ name: -1 })
3. **Hashed Index:** db.students.createIndex({ name: "hashed" })
4. **Text Index (Full-Text Search):** db.students.createIndex({ name: "text" })
   db.students.find({ $text: { $search: "Amit" } })

◇ **Compound (Multiple) Index**

Used for multiple fields.

db.students.createIndex({ name: 1, age: -1 })

◇ **Unique Index**

Prevents duplicate values.

db.students.createIndex({ email: 1 }, { unique: true })

◇ **Dropping Indexes**

1. **Remove one index:**
2. db.students.dropIndex("name_1")
3. **Remove all custom indexes:**
4. db.students.dropIndexes()

(Note: _id index cannot be deleted.)

◇ **Text Index Example**

db.articles.createIndex({ content: "text" })

db.articles.find({ $text: { $search: "MongoDB" } })

Note: Indexing = Faster search + Better performance + Prevent duplicates.

6. Aggregation Framework

What is Aggregation?

- Aggregation is used to **process and analyze data** (like filter, group, sort, or transform documents).
- It is similar to **SQL's GROUP BY, JOIN, and HAVING**.
- MongoDB uses an **Aggregation Pipeline**, where data passes through multiple stages.

Aggregation Pipeline Syntax

```
db.collection.aggregate([
  { stage1 },
  { stage2 },
  { stage3 }
])
```

◆ **Common Stages in Aggregation**

Stage	Function	Example
$match	Filters documents	{ $match: { salary: { $gt: 70000 } } }
$group	Groups data and applies functions like $sum, $avg	{ $group: { _id: "$companyId", totalSalary: { $sum: "$salary" } } }
$sort	Sorts results	{ $sort: { salary: -1 } }
$project	Selects specific fields	{ $project: { _id: 0, name: 1, salary: 1 } }
$limit	Limits output	{ $limit: 2 }
$skip	Skips certain documents	{ $skip: 2 }

◆ **$lookup (Joins)**

Used to **join two collections** (like SQL join).

```
db.indians.aggregate([
 {
  $lookup: {
   from: "companies",
   localField: "companyId",
   foreignField: "_id",
   as: "companyDetails"
  }
 }
])
```

from = collection to join
localField = field in main collection
foreignField = field in joined collection
as = name of new field to store joined data

$unwind (For Arrays)

Used to **flatten arrays** into separate documents.

```
db.indians.aggregate([{ $unwind: "$workExperience" }])
```

Each array element becomes a separate document.

$out (For Output)

Used to **save results** into a new collection.

```
db.indians.aggregate([
  { $sort: { salary: -1 } },
  { $limit: 3 },
  { $out: "top_salaries" }
])
```

Creates a new collection named *top_salaries*.

- **MongoDB Relationships**

Type	Description	Example
One-to-One	One document linked to one	User ↔ Passport
One-to-Many	One linked to many	Post ↔ Comments
Many-to-Many	Many linked to many	Students ↔ Courses

(v) $unwind for Arrays

- Flatten work Experience array into separate documents.

```
db.indians.aggregate([
  { $unwind: "$workExperience" }
])
```

Note: Converts each work experience into a separate document.

(vi) $out for Writing Output

- Store top 3 highest-paid employees in top_salaries.

```
db.indians.aggregate([
  { $sort: { salary: -1 } },
  { $limit: 3 },
  { $out: "top_salaries" }
])
```

Note: Saves top 3 employees to top_salaries collection.

7. MongoDB Relationships

- One-to-One Relationship
- One-to-Many Relationship
- Many-to-Many Relationship
- Embedded Documents vs. Referenced Documents

1. One-to-One Relationship

- Each document in one collection is linked to exactly one document in another collection.

Example: A user has one passport.

Users Collection:

➢ { _id: 1, name: "Sangam Kumar", passport_id: 101 }

Passports Collection:

➢ { _id: 101, passport_number: "A1234567", user_id: 1 }

2. One-to-Many Relationship

- One document is linked to multiple documents in another collection.
- **Example:** A blog post has multiple comments.

BlogPosts Collection:

➢ { _id: 1, title: "MongoDB Guide" }

Comments Collection:

➢ { _id: 101, post_id: 1, text: "Great post!" }

➢ { _id: 102, post_id: 1, text: "Very helpful!" }

1. Many-to-Many Relationship

- Multiple documents in one collection are related to multiple documents in another collection.
- **Example:** Students enroll in multiple courses, and courses have multiple students.

4. Embedded vs. Referenced Documents

- Embedded (One-to-Many Example: Blog Post with Comments inside)

```
{  _id: 1,
 title: "MongoDB Guide",
 comments: [
  { text: "Great post!" },
  { text: "Very helpful!" } ] }
```

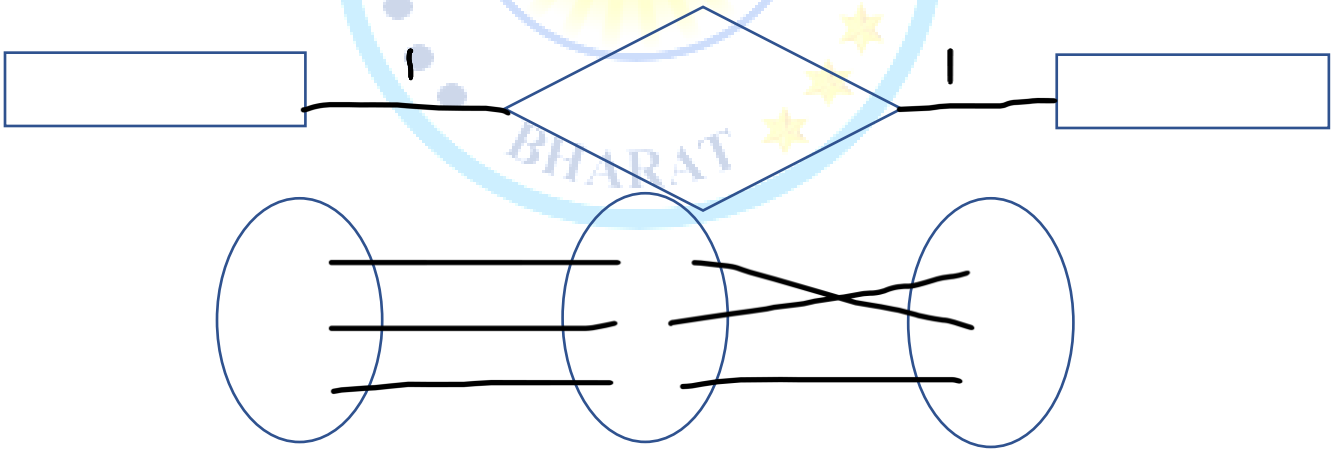
- **Advantage:** Faster reads, fewer queries.

Referenced (One-to-Many Example: Blog Post storing comment IDs)

```
{  _id: 1,
 title: "MongoDB Guide",
 comment_ids: [101, 102]
}
```

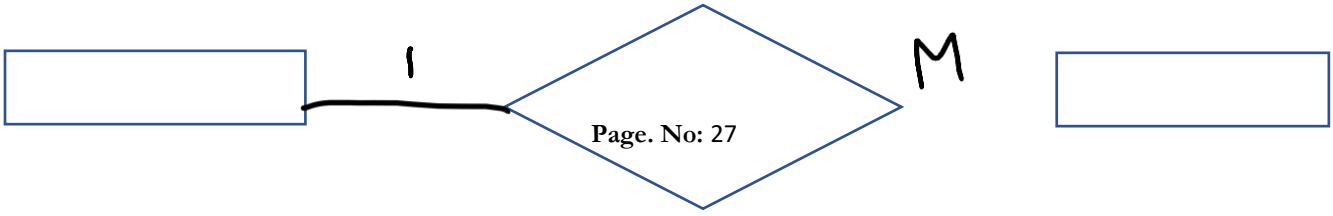
1. One-to-One (1:1) Relationship:

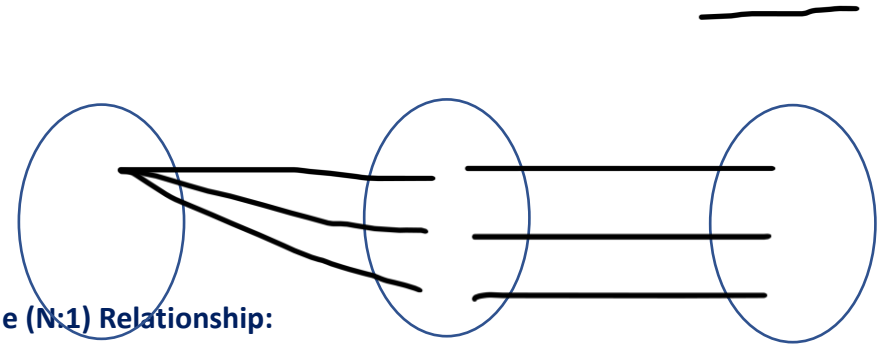
- **Definition:** Each entity in the first set is associated with at most one entity in the second set, and vice versa.
- **Example:** A "Male" entity may have a one-to-one relationship with a "Female" entity, where each person has at most one passport.



2. One-to-Many (1:N) Relationship:

- **Definition:** Each entity in the first set can be associated with multiple entities in the second set, but each entity in the second set is associated with at most one entity in the first set.
- **Example:** A "Scientist" entity may have a one-to-many relationship with an "Invention" entity, where each Invents can have multiple Scientist, but each Scientist is associated with only one Invention.



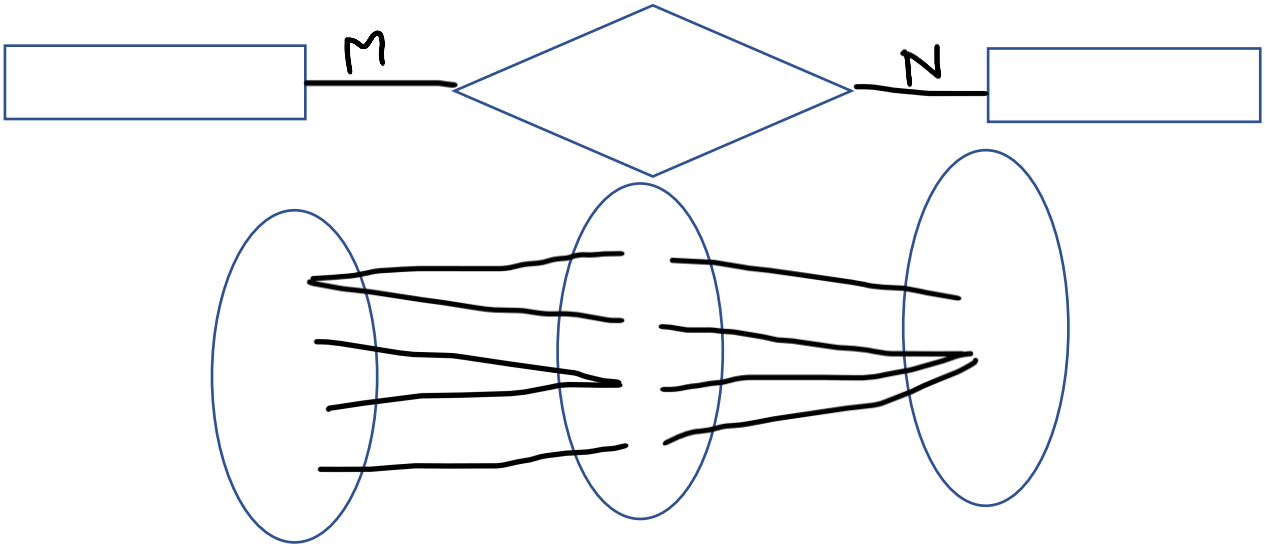


3. Many-to-One (N:1) Relationship:

- **Definition:** The opposite of a one-to-many relationship. Many entities in the first set can be associated with one entity in the second set.
- **Example:** Multiple "students" may have a many-to-one relationship with a single "School" entity, where many students attend the same school.

4. Many-to-Many (N:N) Relationship:

- **Definition:** Entities in both sets can be associated with multiple entities in the opposite set.
- **Example:** A "Student" entity may have a many-to-many relationship with a "Course" entity, indicating that students can enroll in multiple courses, and a course can have multiple students.



8. Transactions in MongoDB

- Introduction to Transactions
- Single and Multi-Document Transactions
- ACID Properties in MongoDB
- Implementing Transactions in MongoDB

1. Introduction to Transactions:

- A transaction in MongoDB is a sequence of operations that are executed as a single unit of work. Transactions ensure data consistency and integrity, even in case of failures or system crashes.

2. Single and Multi-Document Transactions

- Single-Document Transactions: MongoDB automatically ensures atomicity for single-document operations (e.g., insertOne(), updateOne(), deleteOne()).
- Multi-Document Transactions: MongoDB supports multi-document transactions in replica sets (since v4.0) and sharded clusters (since v4.2) for operations affecting multiple documents.

3. ACID Properties in MongoDB

- MongoDB supports ACID properties in transactions:

1. **Atomicity:** All operations in a transaction complete successfully, or none take effect.
2. **Consistency:** The database remains in a valid state before and after the transaction.
3. **Isolation:** Transactions are isolated from each other until committed.
4. **Durability:** Committed transactions are permanently saved.
5. Implementing Transactions in MongoDB

Example: Multi-document transaction using Node.js with MongoDB

```
const { MongoClient } = require("mongodb");
async function runTransaction() {
  const uri = "mongodb://localhost:27017"; // Replace with your MongoDB connection string
  const client = new MongoClient(uri);
  try {
    await client.connect();
    const session = client.startSession();
    session.startTransaction(); // Start the transaction
    const db = client.db("mydatabase");
    const accounts = db.collection("accounts");  // Example: Transferring money between two accounts
    await accounts.updateOne({ name: "Alice" }, { $inc: { balance: -100 } }, { session });
    await accounts.updateOne({ name: "Bob" }, { $inc: { balance: 100 } }, { session });
    await session.commitTransaction(); // Commit the transaction
```

```
    console.log("Transaction committed successfully.");
  } catch (error) {
    console.error("Transaction aborted due to error:", error);
    await session.abortTransaction(); // Rollback in case of error
  } finally {
    await client.close();
  }}
runTransaction();
```

9. Data Modeling in MongoDB

- Schema Design Best Practices
- Normalization vs. Denormalization
- Choosing the Right Storage Model
- Polymorphic vs. Fixed Schemas

1. Schema Design Best Practices

- Design for queries, not just data storage → Structure data based on how it will be accessed.
- Embed related data if frequently accessed together → Use embedded documents instead of joins.
- Reference data if it is large and used independently → Use references (ObjectId) for separate collections.

2. Normalization vs. Denormalization

- **Normalization (Relational-style) →** Data is split into multiple collections, reducing redundancy but requiring joins.
- **Denormalization (NoSQL-style) →** Data is embedded within documents, improving read performance but increasing duplication.

user's collection

- { "_id": 1, "name": "Sangam", "address_id": 101 }

addresses collection

- { "_id": 101, "street": "123 Main St", "city": "NYC" }

Denormalization Example (Embedded address inside user document)

```
{   "_id": 1,
  "name": "Sangam",
  "address": { "street": "123 Main St", "city": "NYC" } }
```

3. Choosing the Right Storage Model

- Embed Data → When data is small and frequently accessed together.
- Reference Data → When data is large and used independently.

Example: E-commerce System

- Embed order items in an order document (denormalized).
- Store customer details separately and reference them in orders (normalized).

4. Polymorphic vs. Fixed Schemas

- Fixed Schema → All documents have the same structure.
- Polymorphic Schema → Documents have different structures based on the data type.

Fixed Schema Example (Consistent fields for all products)

- { "_id": 1, "name": "Laptop", "price": 1000, "category": "Electronics" }

Polymorphic Schema Example (Different fields for different products)

- **Electronics:** { "_id": 1, "name": "Laptop", "price": 1000, "specs": { "RAM": "16GB", "CPU": "i7" } }
- **Clothing:** { "_id": 2, "name": "T-Shirt", "price": 20, "size": "L", "color": "Blue" }

10. MongoDB with Programming Languages

- Connecting MongoDB with Node.js (mongoose ORM)
- Connecting MongoDB with Python (pymongo)
- CRUD Operations with Programming

❖ How to MongoDB with HTML CSS JavaScript and Node.js (mongoose ORM)

Step-by-Step Guide: Connecting MongoDB with HTML, CSS, JavaScript, and Node.js

Full Stack project

- Frontend (HTML, CSS, JavaScript) will collect user data.
- Backend (Node.js, Express, MongoDB - Mongoose ORM) will handle database operations.
- MongoDB will store the data.

❖ Project File Structure

```
mongodb-project/
|— backend/
|   ├— server.js       # Main Node.js server
|   ├— database.js     # MongoDB connection
|   ├— models/User.js  # Mongoose Schema (User Model)
|   ├— routes/user.js  # API Routes
|   ├— package.json    # Node.js dependencies
|
|— frontend/
|   ├— index.html      # HTML Form
|   ├— style.css       # CSS Styling
|   ├— script.js       # JavaScript (Fetch API)
```

Step 1: Setup Backend (Node.js + Express + MongoDB)

1. Initialize a Node.js Project: - Inside your project folder, open a terminal and run.

- mkdir mongodb-project
- cd mongodb-project
- mkdir backend frontend
- cd backend
- npm init -y

This will create a package.json file.

2. Install Required Dependencies

- Run the following command:
- npm install express mongoose cors body-parser

3 Connect MongoDB (database.js

- Inside the `backend/` folder, create **`database.js`** to connect MongoDB:

```
const mongoose = require("mongoose");
const mongoURI = "mongodb://localhost:27017/mydatabase"; // Change 'mydatabase' as needed
const connectDB = async () => {
  try {
    await mongoose.connect(mongoURI, { useNewUrlParser: true, useUnifiedTopology: true });
    console.log("MongoDB Connected Successfully!");
  } catch (err) {
    console.error("MongoDB Connection Failed:", err);
  }
};
module.exports = connectDB;
```

4. Create a Mongoose Schema (User.js)

- Inside backend/models/, create User.js:

```
const mongoose = require("mongoose");
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  message: String
});
const User = mongoose.model("User", userSchema);
module.exports = User;
```

5. Create API Routes (user.js)

- Inside backend/routes/, create user.js:

```
const express = require("express");
const User = require("../models/User");
const router = express.Router();
// Create a User
router.post("/addUser", async (req, res) => {
  try {
    const newUser = new User(req.body);
    await newUser.save();
    res.status(201).json({ message: "User Added Successfully" });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }});
// Get All Users
router.get("/getUsers", async (req, res) => {
```

```
    try {
        const users = await User.find();
        res.status(200).json(users);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }});
module.exports = router;
```

6. Create the Main Server File (server.js)

- Inside backend/, create server.js:

```
const express = require("express");
const cors = require("cors");
const bodyParser = require("body-parser");
const connectDB = require("./database");
const userRoutes = require("./routes/user");
const app = express();
const PORT = 5000;

// Middleware
app.use(cors());
app.use(bodyParser.json());

// Connect to MongoDB
connectDB();

// Routes
app.use("/api/users", userRoutes);

// Start the Server
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

Run the backend server:

node backend/server.js

Your Node.js server is running on http://localhost:5000

Step 2: Setup Frontend (HTML, CSS, JavaScript)

1. Create the HTML Form (index.html)

Inside frontend/, create index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
    <title>MongoDB with JavaScript</title>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1>User Form</h1>
    <form id="userForm">
      <input type="text" id="name" placeholder="Name" required>
      <input type="email" id="email" placeholder="Email" required>
      <textarea id="message" placeholder="Message" required></textarea>
      <button type="submit">Submit</button>
    </form>
    <h2>User List</h2>
    <ul id="usersList"></ul>

    <script src="script.js"></script>
  </body>
  </html>
```

2. Create the CSS Styling (style.css)

- Inside frontend/, create style.css:

```
body {
  font-family: Arial, sans-serif;
  text-align: center;
}
form {
  display: flex;
  flex-direction: column;
  width: 300px;
  margin: 20px auto;
}
input, textarea, button {
  margin: 10px 0;
  padding: 8px;
}
```

3. Create JavaScript to Handle API Calls (script.js)

- Inside frontend/, create script.js

```
document.getElementById("userForm").addEventListener("submit", async function(event) {
  event.preventDefault();

  const name = document.getElementById("name").value;
  const email = document.getElementById("email").value;
  const message = document.getElementById("message").value;

  const response = await fetch("http://localhost:5000/api/users/addUser", {
```

```
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ name, email, message })
    });

    if (response.ok) {
        alert("User added successfully!");
        getUsers();  // Refresh the user list
    }
});

// Fetch users from the database
async function getUsers() {
    const response = await fetch("http://localhost:5000/api/users/getUsers");
    const users = await response.json();

    const usersList = document.getElementById("usersList");
    usersList.innerHTML = "";
    users.forEach(user => {
        const li = document.createElement("li");
        li.textContent = `${user.name} - ${user.email} - ${user.message}`;
        usersList.appendChild(li);
    });
}

getUsers(); // Load users when the page loads
```

Step 3: Run the Full Project:

Start MongoDB Server

```
>> mongod
```

Start the Backend Server

```
>> node backend/server.js
```

Note: Open the frontend/index.html in a browser

- Fill the form and submit.
- The user will be saved in MongoDB and displayed below the form.

Summary

Feature	Technology Used
Frontend	HTML, CSS, JavaScript (Fetch API)
Backend	Node.js, Express.js
Database	MongoDB (Mongoose ORM)
API Routes	`/api/users/addUser`, `/api/users/getUsers`

Step-by-Step Guide:

Connecting MongoDB with Python (pymongo) using Django

- We will set up a Django project that connects to MongoDB using pymongo.

Project File Structure

```
mongodb-django/
|— myproject/          # Django Project
|   ├— myapp/          # Django App
|   |   ├— models.py   # MongoDB Models
|   |   ├— views.py    # Views for API
|   |   ├— urls.py     # API Routes
|   ├— settings.py     # Django Settings
|   ├— urls.py         # Main URL Config
|   ├— wsgi.py         # WSGI Entry Point
|— manage.py           # Django CLI Tool
|— requirements.txt    # Dependencies
```

Step 1: Install Dependencies

- Make sure you have Python and MongoDB installed. Then install Django and pymongo:
- pip install django pymongo dnspython

Create a Django project

shell

```
django-admin startproject myproject
cd myproject
python manage.py startapp myapp
```

❖ Install Django Rest Framework (optional, for APIs)

shell

```
pip install djangorestframework
```

Add myapp and rest_framework to INSTALLED_APPS in settings.py

python

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
```

```
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'myapp',  # Add your app
]
```

Step 2: Configure MongoDB in Django (settings.py)

- Edit myproject/settings.py to connect Django with MongoDB using pymongo:

python

```
from pymongo import MongoClient
# MongoDB Connection
MONGO_URI = "mongodb://localhost:27017/"
MONGO_DB_NAME = "mydatabase"
# Connect to MongoDB
client = MongoClient(MONGO_URI)
db = client[MONGO_DB_NAME]
```

Step 3: Create a Model in models.py (Custom Model using pymongo)

- Since Django does not natively support MongoDB, we will use pymongo directly.

Edit myapp/models.py.

python

```
from django.conf import settings
class User:
    collection = settings.db["users"]  # MongoDB collection
    @staticmethod
    def create_user(name, email, age):
        user_data = {"name": name, "email": email, "age": age}
        return User.collection.insert_one(user_data)
    @staticmethod
    def get_users():
        return list(User.collection.find({}, {"_id": 0}))  # Fetch users, exclude `_id`
```

Step 4: Create API Views (views.py)

- Edit myapp/views.py to define APIs for adding and retrieving users:

python

```
from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt
import json
from .models import User
@csrf_exempt
def add_user(request):
    if request.method == "POST":
```

```
        data = json.loads(request.body)
        User.create_user(data["name"], data["email"], data["age"])
        return JsonResponse({"message": "User added successfully"}, status=201)
    def get_users(request):
        users = User.get_users()
        return JsonResponse(users, safe=False)
```

Step 5: Define Routes (urls.py)

Edit myapp/urls.py :

python

```
    from django.urls import path
    from .views import add_user, get_users
    urlpatterns = [
        path("add-user/", add_user, name="add_user"),
        path("get-users/", get_users, name="get_users"),
    ]
```

Edit myproject/urls.py to include myapp routes.

python

```
    from django.contrib import admin
    from django.urls import path, include

    urlpatterns = [
        path("admin/", admin.site.urls),
        path("api/", include("myapp.urls")),  # Include API routes
    ]
```

Step 6: Run Django Server

Sh>> python manage.py runserver

API Endpoints Available.

POST /api/add-user/ → Add a user (JSON body: { "name": "John", "email": "sangam@example.com", "age": 25 }`)

- GET `/api/get-users/` → Fetch all users from MongoDB

Step 7: Test the API using curl or Postman

Add a User

sh

```
curl -X POST http://127.0.0.1:8000/api/add-user/ \
    -H "Content-Type: application/json" \
    -d '{"name": "Alice", "email": "alice@example.com", "age": 30}'
```

Fetch Users

Sh>> curl http://127.0.0.1:8000/api/get-users/

CRUD Operations in MongoDB with Programming (Node.js & Python)

- CRUD stands for Create, Read, Update, Delete, which are the main database operations.

We will implement CRUD using:

- Node.js (Mongoose - ODM)
- Python (PyMongo - Driver)

1. CRUD Operations in MongoDB using Node.js (Mongoose ORM)

➢ Project File Structure

```
mongodb-nodejs/
|— server.js       # Main Node.js server
|— database.js     # MongoDB connection
|— models/User.js   # User Schema
|— routes/user.js   # API Routes
|— package.json     # Dependencies
```

Step 1: Install Required Packages

```
>> npm install express mongoose cors body-parser
```

Step 2: Connect to MongoDB (`database.js`)

Javascript>>

```
const mongoose = require("mongoose");
const mongoURI = "mongodb://localhost:27017/mydatabase";
const connectDB = async () => {
  try {
    await mongoose.connect(mongoURI, { useNewUrlParser: true, useUnifiedTopology: true });
    console.log("MongoDB Connected Successfully!");
  } catch (err) {
    console.error("MongoDB Connection Failed:", err);
  }
};
module.exports = connectDB;
```

Step 3: Create a User Schema (models/User.js)

Javascript>>

```
const mongoose = require("mongoose");
const userSchema = new mongoose.Schema({
   name: String,
   email: String,
   age: Number
});
const User = mongoose.model("User", userSchema);
module.exports = User;
```

Step 4: Implement CRUD Operations (routes/user.js)

javascript

```
const express = require("express");
const User = require("../models/User");
const router = express.Router();
// Create User
router.post("/addUser", async (req, res) => {
  try {
    const newUser = new User(req.body);
    await newUser.save();
    res.status(201).json({ message: "User Added Successfully", user: newUser });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});// Read Users
router.get("/getUsers", async (req, res) => {
  try {
    const users = await User.find();
    res.status(200).json(users);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});// Update User
router.put("/updateUser/:id", async (req, res) => {
  try {
    const updatedUser = await User.findByIdAndUpdate(req.params.id, req.body, { new: true });
    res.status(200).json(updatedUser);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
//Delete User
router.delete("/deleteUser/:id", async (req, res) => {
  try {
    await User.findByIdAndDelete(req.params.id);
```

```
    res.status(200).json({ message: "User Deleted Successfully" });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
module.exports = router;
```

Step 5: Set Up the Server (server.js)

javascript

```
const express = require("express");
const cors = require("cors");
const bodyParser = require("body-parser");
const connectDB = require("./database");
const userRoutes = require("./routes/user");
const app = express();
const PORT = 5000;
app.use(cors());
app.use(bodyParser.json());
connectDB();
app.use("/api/users", userRoutes);
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

Run the server:>> node server.js

API Routes Available:

Operation	HTTP Method	Endpoint
Create	POST	/api/users/addUser
Read	GET	/api/users/getUsers
Update	PUT	/api/users/updateUser/:id
Delete	DELETE	/api/users/deleteUser/:id

CRUD implementation using Python with PyMongo (MongoDB driver for Python).

❖ Steps to Set Up

1. Install MongoDB and PyMongo

- **Install MongoDB:** [Download MongoDB](https://www.mongodb.com/try/download/community)
- **Install PyMongo:**

```
>> pip install pymongo
```

2. Run MongoDB Server: If MongoDB is installed locally, start it using:

```
>>  mongod
```

CRUD Operations with PyMongo

```
from pymongo import MongoClient
# Connect to MongoDB
client = MongoClient("mongodb://localhost:27017/")  # Default local connection
db = client["test_db"]  # Database name
collection = db["users"]  # Collection name
# CREATE: Insert a new document
def create_user(name, age):
    user = {"name": name, "age": age}
    collection.insert_one(user)
    print("User added successfully!")
# READ: Fetch all users
def read_users():
    users = collection.find()
    for user in users:
        print(user)
# UPDATE: Modify a user's age by name
def update_user(name, new_age):
    collection.update_one({"name": name}, {"$set": {"age": new_age}})
    print("User updated successfully!")
# DELETE: Remove a user by name
def delete_user(name):
    collection.delete_one({"name": name})
    print("User deleted successfully!")
# Example usage
```

```
create_user("Sangam", 25)   # Insert user
read_users()            # Display users
update_user("Sangam", 30)   # Update Sangam's age
delete_user("Sangam")      # Delete Sangam
read_users()            # Show remaining users
```

11. MongoDB Cloud and Atlas

- Introduction to MongoDB Atlas
- Setting Up a Cloud Database

❖ What is MongoDB Atlas?

- MongoDB Atlas is a fully managed cloud database service that allows developers to deploy, manage, and scale MongoDB without manual setup. It runs on AWS, Google Cloud, and Azure, providing automated backups, security, high availability, and scalability.

❖ Key Features.

- **Fully Managed –** No need to install or configure MongoDB manually.
- **Multi-Cloud Support –** Deploy on AWS, GCP, or Azure.
- **High Availability –** Uses replica sets for redundancy.
- **Scalability –** Supports automatic scaling and sharding.
- **Security –** Offers IP whitelisting, authentication, encryption, and access control.
- **Automated Backups –** Provides point-in-time recovery to prevent data loss.
- **Monitoring & Performance Optimization –** Built-in real-time metrics and query optimization tools.

❖ How MongoDB Atlas Works?

- **Create a Cluster –** Atlas automatically sets up a MongoDB cluster.
- **Configure Security –** Set up IP whitelisting and user authentication.
- **Connect Your Application –** Use a connection string to link your app.
- **Manage & Scale –** Atlas automatically scales based on usage.

❖ Pricing Model

- Free Tier (M0) – 512MB storage, ideal for learning and small projects.
- Pay-as-You-Go – Costs depend on cluster size, cloud provider, and usage.

❖ Why Use MongoDB Atlas?

- No server management required
- Scalable for large applications
- Secure & highly available
- Easy cloud deployment

Setting Up a Cloud Database in MongoDB Atlas

Step 1: Sign Up & Log In

- Go to MongoDB Atlas and create an account.
- Log in to the MongoDB Atlas dashboard.

Step 2: Create a New Cluster

- Click "Create a Cluster".
- Choose a Cloud Provider (AWS, GCP, or Azure).
- Select a Region and Cluster Tier (choose M0 free tier for practice).
- Click "Create Cluster" (Takes a few minutes to deploy).

Step 3: Configure Database Access

- Go to Database Access → Add New Database User.
- Choose Password Authentication, set a username & password (save for later).
- Click "Create User".

Step 4: Set Up Network Access

- Go to Network Access → Add IP Address.
- Choose "Allow Access from Anywhere" (0.0.0.0/0) (for development).
- Click Confirm.

Step 5: Get the Connection String

- Go to Clusters → Connect.
- Select "Connect Your Application".
- Copy the MongoDB connection string (e.g., mongodb+srv://yourusername:yourpassword@cluster0.mongodb.net/test).

Step 6: Connect Your Application

- Install PyMongo:
- pip install pymongo

❖ Connect using Python.

```
from pymongo import MongoClient
client = MongoClient("mongodb+srv://yourusername:yourpassword@cluster0.mongodb.net/test")
db = client["mydatabase"]
print("Connected to MongoDB Atlas!")
```

12.MongoDB with Web Applications

- Integrating MongoDB with Express.js and Node.js
- Building a REST API with MongoDB
- Connecting MongoDB with React.js/Angular.js

❖ MongoDB with Web Applications

- MongoDB is commonly used in modern web applications as a NoSQL database due to its flexibility, scalability, and JSON-like document structure. It integrates well with Node.js, Express.js, React.js, and Angular.js to build full-stack applications.

1. Integrating MongoDB with Express.js and Node.js

➢ Why Use MongoDB with Node.js & Express.js?

- JavaScript-based stack (MEAN/MERN) makes integration seamless.
- MongoDB stores data as JSON-like documents, making it easy to use with JavaScript frameworks.
- Fast & scalable for handling large datasets and real-time applications.
- Steps to Integrate MongoDB with Node.js & Express.js

Step-by-Step Guide to Setting Up Node.js with Express and MongoDB

- This guide will take you through setting up a Node.js server with Express.js and connecting it to MongoDB step by step.

Step 1: Install Node.js & npm

1.1 Download Node.js

- Go to the official Node.js website.
- Download the LTS (Long-Term Support) version for your OS (Windows, macOS, or Linux).
- Run the installer and follow the installation steps.

1.2 Verify Installation

- After installation, check if Node.js and npm (Node Package Manager) are installed:

```
>> node -v  # Check Node.js version
>> npm -v   # Check npm version
```

Note: - If both commands return a version number, Node.js is installed successfully.

Step 2: Create a Node.js Project

2.1 Create a New Project Folder

- Open a terminal or command prompt and run:
  >> mkdir myapp **# Create a new folder**
  >> cd myapp **# Navigate into the folder**

2.2 Initialize a Node.js Project

- Run the following command to create a package.json file:
  >> **npm init -y**

Note: - This will create a package.json file, which stores project dependencies and configurations.

❖ Why Use npm init -y?

- npm init initializes a new Node.js project and asks multiple setup questions.
- -y flag skips all questions and generates a default package.json file with preset values.
- This file manages dependencies, project metadata, and scripts.

Step 3: Install Dependencies

3.1 Install Express.js

- Express.js is a web framework for Node.js. Install it with:
  >> npm install express

3.2 Install Mongoose (MongoDB ODM)

- Mongoose is an Object Data Modeling (ODM) library for MongoDB. Install it with:
  >> npm install mongoose

3.3 Install dotenv (for Environment Variables)

- To store database credentials securely, install dotenv:
  >> npm install dotenv

3.4 Install Nodemon (for Auto-restart on Changes)

- Nodemon automatically restarts the server when files change:
  >> npm install -g nodemon

Step 4: Set Up MongoDB Connection

4.1 Create a MongoDB Atlas Database

- Go to MongoDB Atlas and sign up.
- Click "Create Cluster" and choose the M0 (Free Tier) option.
- After cluster setup, go to Database Access and create a new user.
- Under Network Access, allow access from any IP (0.0.0.0/0).
- Copy the MongoDB connection string from the "Connect" section.

4.2 Create a .env File

- Inside your project folder, create a .env file and paste your MongoDB URL:
- MONGO_URI=mongodb+srv://yourusername:yourpassword@cluster0.mongodb.net/mydatabase

PORT=5000

4.3 Create a Database Connection File

- Create a new file db.js to connect to MongoDB:

```
const mongoose = require("mongoose");
require("dotenv").config();
const connectDB = async () => {
```

```
    try {
        await mongoose.connect(process.env.MONGO_URI, {
            useNewUrlParser: true,
            useUnifiedTopology: true
        });
        console.log("MongoDB Connected!");
    } catch (error) {
        console.error("Connection Error:", error);
        process.exit(1);
    }
};
module.exports = connectDB;
```

Step 5: Create an Express Server

5.1 Create server.js File

- Create a server.js file in the project folder and add the following code:

```
const express = require("express");
const dotenv = require("dotenv");
const connectDB = require("./db");

dotenv.config();  // Load environment variables
connectDB();  // Connect to MongoDB

const app = express();
app.use(express.json());  // Middleware to parse JSON data

// Test Route
app.get("/", (req, res) => {
    res.send("API is running...");
});

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

Step 6: Run the Server

6.1 Start the Server with Nodemon

- Run the following command in your terminal:

```
>> nodemon server.js
```

Node: If everything is set up correctly, you should see:

- MongoDB Connected!
- Server running on port 5000

6.2 Test the API

- Open your browser and visit:
  http://localhost:5000/

You should see API is running....

Next Steps: Build a CRUD API with MongoDB

- Now that the server is running, the next steps are:

Would you like a guide on building a CRUD API with MongoDB and Express.js?

13. MongoDB Tools & Utilities

- MongoDB provides various tools and utilities to interact with and manage databases efficiently. Two key tools are MongoDB Compass and mongosh.

1) MongoDB Compass (GUI for MongoDB)

- MongoDB Compass is a Graphical User Interface (GUI) tool that allows users to interact with MongoDB databases visually.
- It helps in exploring data, running queries, and managing collections without using the command line.
- Features:
- Visual representation of documents and schema analysis.
- Query building with a GUI.
- Index and performance monitoring.
- CRUD (Create, Read, Update, Delete) operations on documents.

2) Using mongosh for Advanced Queries

- mongosh (MongoDB Shell) is an interactive command-line interface for executing advanced queries in MongoDB.
- It replaces the older mongo shell and provides better integration with modern JavaScript.
- Features:
- Supports JavaScript-based queries.
- Enables aggregation pipelines and advanced filtering.
- Allows scripting and automation of MongoDB operations.
- Provides debugging tools for database interaction.
- Example Query in mongosh:
- db.students.find({ age: { $gt: 18 } }) // Finds students older than 18
- Both tools are essential for developers and database administrators working with MongoDB.

What is RID Organization (RID संस्था क्या है)?

- **RID Organization** यानि **Research, Innovation and Discovery Organization एक** संस्था हैं जो TWF (TWKSAA WELFARE FOUNDATION) **NGO** द्वारा RUN किया जाता है | जिसका मुख्य उदेश्य हैं आने वाले समय में सबसे पहले **NEW (RID, PMS & TLR)** की खोज, प्रकाशन एवं उपयोग भारत की इस पावन धरती से भारतीय संस्कृति, सभ्यता एवं भाषा में ही हो |
- देश, समाज, एवं लोगों की समस्याओं का समाधान **NEW (RID, PMS & TLR)** के माध्यम से किया जाये इसके लिए ही **इस RID Organization** की स्थपना 30.09.2023 किया गया है | जो TWF द्वारा संचालित किया जाता है |
- TWF (TWKSAA WELFARE FOUNDATION) NGO की स्थपना 26-10-2020 में बिहार की पावन धरती सासाराम में Er. RAJESH PRASAD एवं Er. SUNIL KUMAR द्वारा किया गया था जो की भारत सरकार द्वारा मान्यता प्राप्त संस्था हैं |
- Research, Innovation & Discovery में रूचि रखने वाले आप सभी विधार्थियों, शिक्षकों एवं बुधीजिवियो से मैं आवाहनं करता हूँ की आप सभी **इस RID संस्था** से जुड़ें एवं अपने बुध्दि, विवेक एवं प्रतिभा से दुनियां को कुछ नई **(RID, PMS & TLR)** की खोजकर, बनाकर एवं अपनाकर लोगों की समस्याओं का समाधान करें |

MISSION, VISSION & MOTIVE OF "RID ORGANIZATION"	
मिशन	हर एक ONE भारत के संग
विजन	TALENT WORLD KA SHRESHTM AB AAYEGA भारत में और भारत का TALENT भारत में
मकसद	NEW (RID, PMS, TLR)

MOTIVE OF RID ORGANIZATION NEW (RID, PMS, TLR)		
NEW (RID)		
R	I	D
Research	Innovation	Discovery
NEW (TLR)		
T	L	R
Technology, Theory, Technique	Law	Rule
NEW (PMS)		
P	M	S
Product, Project, Production	Machine	Service

Er. Rajesh Prasad (B.E, M.E)

RID रीड संस्था की मिशन, विजन एवं मकसद को सार्थक हमें बनाना हैं |
भारत के वर्चस्व को हर कोने में फैलना हैं |
कर के नया कार्य एक बदलाव समाज में लाना हैं |
रीड संस्था की कार्य-सिध्दांतों से ही, हमें अपनी पहचान बनाना हैं |

RESEARCH INNOVATION DISCOVERY
RID
BHARAT