ವಿಶ್ವೇಶ್ವರಯ್ಯ ತಾಂತ್ರಿಕ ವಿಶ್ವವಿದ್ಯಾಲಯ, ಬೆಳಗಾವಿ

VISVESVARAYA TECHNOLOGICAL UNIVERSITY - BELAGAVI

BANGALORE INSTITUTE OF TECHNOLOGY

K.R. Road, V.V. Pura, Bengaluru -560 004

Department of CSE (DATA SCIENCE)

BDSL456C

MERN Lab Manual

IV- Semester

Prepared by:

Prof. Mamatha V

Prof. Tejashwini P.S

Prof. Prathik K

Bangalore Institute of Technology

K. R. Road, V. V. Pura, Bengaluru- 560004

Department of CSE (Data Science)

VISION

The vision of CSE Data Science Department is to establish a state of art academic department thattrains the next generation of students as data scientists who will solve complex challenges and innovate through world-class research.

MISSION

M1	To educate students in a field that has ushered in a once-in-a-generation revolution.
M2	To provide an environment for leading edge research that has a strong and rapid impact on the economy.
M3	To establish a "Center of Excellence" to facilitate Data creators, providers, managers, curators, and users.
M4	To inculcate the strong qualities of leadership and Entrepreneurship.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO-1	To provide Graduates with a strong understanding of basic and advanced methods in statistical inference, machine learning, data visualization, data mining, and big data, all of which are essential skills for a high-performing Data Scientist.
PEO-2	The Graduate will have the ability to apply the fundamental and advanced skills to assist with data-based decision making.
PEO-3	Graduates will have the ability to strengthen the level of expertise through higher studies and research.

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO-1	The graduates of the program will have the ability to apply modern artificial intelligence and deep learning methods to complex prediction and recognition tasks.
PSO-2	The graduates of the program will Play an analytical role in a company where one can design, implement, and evaluate advanced statistical models and approaches for application to the company's most complex problem.

PROGRAM OUTCOMES (POs)

The Engineering Graduates will be able to:

1. **Engineering knowledge**:

   Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis**:

   Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. **Design/development of solutions**:

   Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems**:

   Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage**:

   Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society**:

   Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability**:

   Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics**:

   Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work**:

   Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication**:

    Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance**:

    Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Life-long learning**:

    Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

COURSE LEARNING OBJECTIVES (CLO)

CLO1. Understand and apply critical web development languages and tools to create dynamic and responsive web applications.

CLO2. To build server-side applications using Node.js and Express

CLO3. Develop user interfaces with React.js,

CLO4. Manage data using MongoDB, and integrate these technologies to create full stack apps.

CLO5. Understanding APIs and routing.

COURSE OUTCOMES (CO)

At the end of the course the student will be able to:

CO1. Apply the fundamentals of MongoDB, such as data modelling, CRUD operations, and basic queries to solve given problem.

CO2. Use constructs of Express.js, including routing, software and constructing RESTful APIs to solve real world problems.

CO3. Develop scalable and efficient RESTful APIs using NodeJS

CO4. Develop applications using React, including components, state, props, and JSX syntax.

		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
BDSL456C	CO1	3	3	3	2	2	3			3	3	2	2
	CO2	3	3	3	2					3	3	2	2
	CO3	3	3	3	2					3	3	2	2
	CO4	3	3	3	2					3	3	2	2

		PSO1	PSO2
BDSL456C	CO1	2	3
	CO2	2	3
	CO3	2	3
	CO4	2	3

MERN Laboratory

Subject Code : BDSL456C | **CIE Marks : 50**
Hours/Week : 0:0:2 | **Total Hours : 20**

LIST OF PROGRAMS

Sl. No.	Name of Experiment
1.	a.Using MongoDB, create a collection called transactions in database usermanaged (drop if it already exists) and bulk load the data from a json file, transactions.json b. Upsert the record from the new file called transactions_upsert.json in Mongodb.
2.	Query MongoDB with Conditions: [Create appropriate collection with necessary documents to answer the query] a. Find any record where Name is Somu b. Find any record where total payment amount (Payment.Total) is 600. c. Find any record where price (Transaction.price) is between 300 to 500. d. Calculate the total transaction amount by adding up Payment.Total in all records.
3.	a. Write a program to check request header for cookies. b. write node.js program to print the a car object properties, delete the second property and get length of the object.
4.	a. Read the data of a student containing usn, name, sem, year_of_admission from node js and store it in the mongodb b. For a partial name given in node js, search all the names from mongodb student documents created in Question(a) .
5.	Implement all CRUD operations on a File System using Node JS .
6.	Develop the application that sends fruit name and price data from client side to Node.js server using Ajax .
7.	Develop an authentication mechanism with email_id and password using HTML and Express JS (POST method).
8.	Develop two routes: find_prime_100 and find_cube_100 which prints prime numbers less than 100 and cubes less than 100 using Express JS routing mechanism.
9.	Develop a React code to build a simple search filter functionality to display a filtered list based on the search query entered by the user.
10.	Develop a React code to collect data from rest API.

MERN Laboratory

Subject Code : BDS456C **CIE Marks: 50**
Hours/Week : 0:0:2 **Total Hours: 20**

SCHEDULE OF EXPERIMENTS

Sl. No	Name of Experiment	WEEK
1	Sample programs	Week1
2	a. Using MongoDB, create a collection called transactions in database usermanaged (drop if it already exists) and bulk load the data from a json file, transactions.json b. Upsert the record from the new file called transactions_upsert.json in Mongodb.	Week2
3	Query MongoDB with Conditions: [Create appropriate collection with necessary documents to answer the query] a. Find any record where Name is Somu b. Find any record where total payment amount (Payment.Total) is 600. c. Find any record where price (Transaction.price) is between 300 to 500. d. Calculate the total transaction amount by adding up Payment.Total in all records.	Week3
4	a. Write a program to check request header for cookies. b. write node.js program to print the a car object properties, delete the second property and get length of the object.	Week4
5	a. Read the data of a student containing usn, name, sem, year_of_admission from node js and store it in the mongodb b. For a partial name given in node js, search all the names from mongodb student documents created in Question(a) .	Week5
6	Implement all CRUD operations on a File System using Node JS .	Week6
7	Develop the application that sends fruit name and price data from client side to Node.js server using Ajax .	Week7
8	Develop an authentication mechanism with email_id and password using HTML and Express JS (POST method).	Week8
9	Develop two routes: find_prime_100 and find_cube_100 which prints prime numbers less than 100 and cubes less than 100 using Express JS routing mechanism.	Week9
10	Develop a React code to build a simple search filter functionality to display a filtered list based on the search query entered by the user.	Week10

11	Develop a React code to collect data from rest API.	Week11
12	Final Lab Test	Week12

LABORATORY

General Lab Guidelines:

1. Conduct yourself in a responsible manner at all times in the laboratory. Intentional misconduct will lead to exclusion from the lab.
2. Do not wander around, or distract other students, or interfere with the laboratory experiments of other students.
3. Read the handout and procedures before starting the experiments. Follow all written and verbal instructions carefully.
4. If you do not understand the procedures, ask the instructor or teaching assistant. Attendance in all the labs is mandatory, absence permitted only with prior permission from the Class teacher.
5. The workplace has to be tidy before, during and after the experiment.
6. Do not eat food, drink beverages or chew gum in the laboratory.
7. Every student should know the location and operating procedures of all Safety equipment including First Aid Kit and Fire extinguisher.

DO'S:-

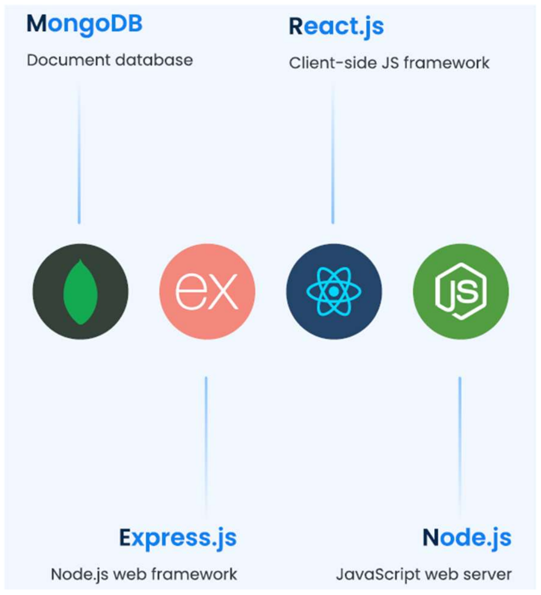
1. An ID card is a must.
2. Keep your belongings in a designated area.
3. Sign the log book when you enter/leave the laboratory.
4. Records have to be submitted every week for evaluation.
5. The program to be executed in the respective lab session has to be written in the lab observation copy beforehand.
6. After the lab session, shut down the computers.
7. Report any problem in system (if any) to the person in-charge

DON'TS:-

1. Do not insert metal objects such as clips, pins and needles into the computer casings(They may cause fire) and should not attempt to repair, open, tamper or interfere with any of the computer, printing, cabling, or other equipment in the laboratory.
2. Do not change the system settings and keyboard keys.
3. Do not upload, delete or alter any software/ system files on laboratory computers.
4. No additional material should be carried by the students during regular labs.

5. Do not open any irrelevant websites in labs.
6. Do not use a flash drive on lab computers without the consent of the lab instructor.
7. Students are not allowed to work in the Laboratory alone or without the presence of the instructor/teaching assistant.

INTRODUCTION TO MERN STACK

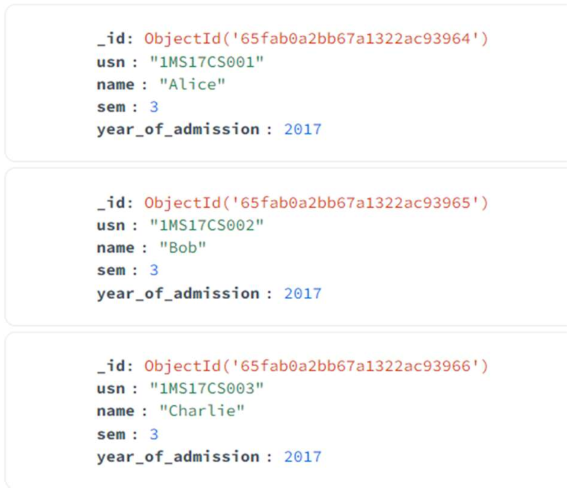


Introduction

MERN Stack is a Javascript Stack that is used for easier and faster deployment of full-stack web applications. MERN Stack comprises of 4 technologies namely: MondoDB , Express, React, and Node.js. It is designed to make the development process smoother and easier.The MERN architecture allows you to easily construct a 3-tier architecture (frontend, backend, database) entirely using JavaScript and JSON.Using these four technologies you can create absolutely any application that you can think of everything that exists in this world today.

MongoDB forms the M of the MERN stack and works pretty well with the JavaScript ecosystem. MongoDB is a NoSQL database in which data is stored in documents that consist of key-value pairs, sharing a lot of resemblance to JSON. The data is not stored in the form of tables and that’s how it differs from other database programs.

This is how the data stored in MongoDB looks like:

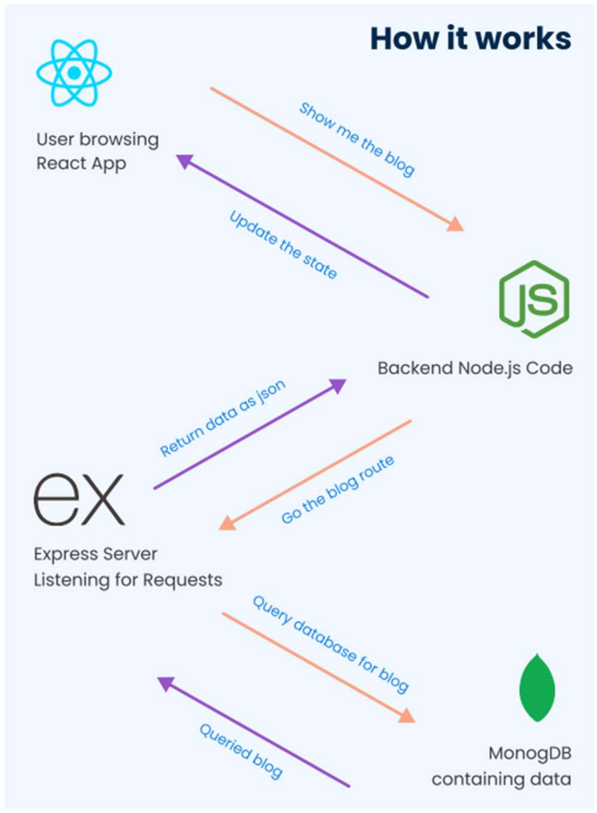


```
_id: ObjectId('65fab0a2bb67a1322ac93964')
usn : "1MS17CS001"
name : "Alice"
sem : 3
year_of_admission : 2017

_id: ObjectId('65fab0a2bb67a1322ac93965')
usn : "1MS17CS002"
name : "Bob"
sem : 3
year_of_admission : 2017

_id: ObjectId('65fab0a2bb67a1322ac93966')
usn : "1MS17CS003"
name : "Charlie"
sem : 3
year_of_admission : 2017
```

Express is a flexible and clean Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node based web applications. In MERN stack, Express will be used as backend API server which interacts with mongoDB database to serve data to client (React) application. Express helps build the backend very easily while staying in JavaScript ecosystem. It is preferred for self-projects as it helps focus on learning development and building projects pretty fast.

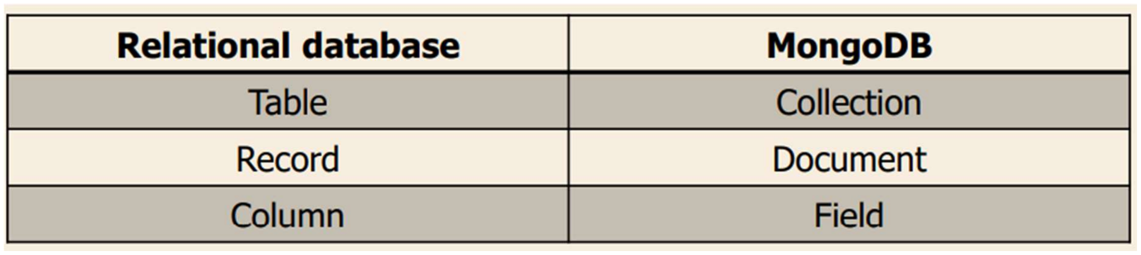
React is an open-source JavaScript library that is used for building user interfaces specifically for single-page applications. It's used for handling the view layer for web and mobile apps. React lets you build up complex interfaces through simple Components, connect them to data on your backend server, and render them as HTML. Almost all the modern tech companies from earlystage startups to the biggest tech companies like Microsoft and Facebook use React. The prime reason why react is used, is for Single Page Applications(SPA). SPA means rendering the entire website on one page rather than different pages of the websites.

NodeJS is a cross-platform JavaScript runtime environment, it's built on Chrome's V8 engine to run JavaScript code outside the browser, for easily building fast and scalable applications. The main purpose of NodeJS is simple, it allows us to write our backend in JavaScript, saving us the trouble of learning a new programming language capable of running the backend. Node.js is the platform for the application layer (logic layer). This will not be visible to the clients. This is where client applications (React) will make requests for data or webpages.



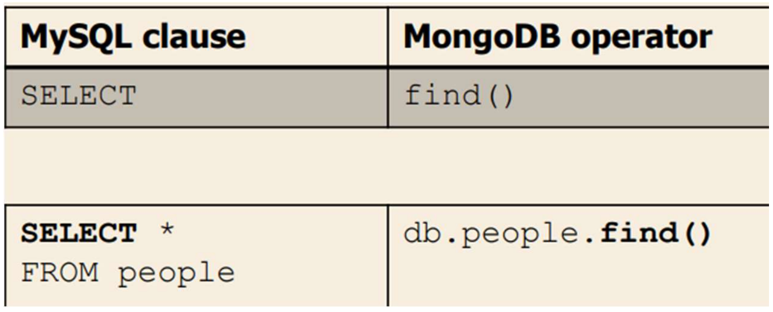
MongoDB stores data records as documents (specifically JSON documents) which are gathered together in collections. A database stores one or more collections of documents.

Terminology – Approximate mapping



Relational database	MongoDB
Table	Collection
Record	Document
Column	Field

Most of the operations available in SQL language can be expressend in MongoDB language



MySQL clause	MongoDB operator
SELECT	find()
SELECT * FROM people	db.people.**find()**

MongoDB use DATABASE_NAME is used to create database. The command will create a new database if it doesn't exist, otherwise it will return the existing database.

Syntax

use DATABASE_NAME

To check your currently selected database, use the command db

```
>db
mydb
```

If you want to check your databases list, use the command show dbs.

```
>show dbs
local    0.78125GB
test     0.23012GB
```

Your created database (mydb) is not present in list. To display database, you need to insert at least one document into it.

```
>db.movie.insert({"name":"tutorials point"})
>show dbs
local     0.78125GB
mydb      0.23012GB
test      0.23012GB
```

In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.

MongoDB db.dropDatabase() command is used to drop a existing database.

db.dropDatabase()

In MongoDB, db.createCollection(name, options) is used to create collection. But usually we dont need to create collection. MongoDB creates collection automatically when you insert some documents.

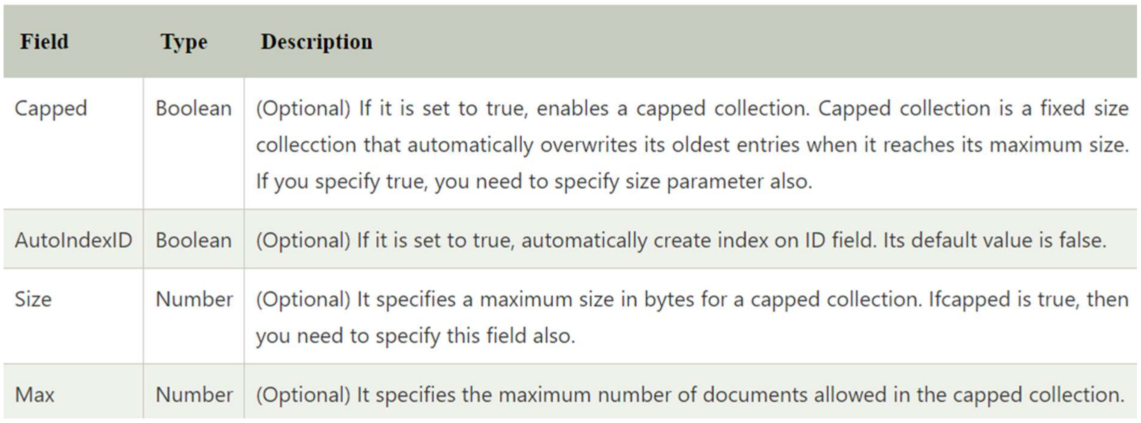
MongoDB db.createCollection(name, options) is used to create collection.

db.createCollection(name, options)

Name: is a string type, specifies the name of the collection to be created.

Options: is a document type, specifies the memory size and indexing of the collection. It is an optional parameter.

Following is the list of options that can be used.



Field	Type	Description
Capped	Boolean	(Optional) If it is set to true, enables a capped collection. Capped collection is a fixed size collecction that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also.
AutoIndexID	Boolean	(Optional) If it is set to true, automatically create index on ID field. Its default value is false.
Size	Number	(Optional) It specifies a maximum size in bytes for a capped collection. Ifcapped is true, then you need to specify this field also.
Max	Number	(Optional) It specifies the maximum number of documents allowed in the capped collection.

Read data from documents

Select documents

db.<collection name>.find({<conditions>},{<fields of interest>});

E.g.,

db.people.find();

Returns all documents contained in the people collection

Select documents

db.<collection name>.find({<conditions>},{<fields of interest>});

Select the documents satisfying the specified conditions and specifically only the fields specified in fields of interest

<conditions> are optional

conditions take a document with the form:

{field1 : <value>, field2 : <value> ... }

Conditions may specify a value or a regular expression

Select the documents satisfying the specified conditions and specifically only the fields specified in fields of interest

- <fields of interest> are optional
- projections take a document with the form:
  {field1 : <value>, field2 : <value> ... }
- 1/true to include the field, 0/false to exclude the field.

E.g.,

db.people.find().pretty();

- No conditions and no fields of interest
- Returns all documents contained in the people collection
- pretty() displays the results in an easy-to-read format
  db.people.find({age:55})
- One condition on the value of age
- Returns all documents having age equal to 55

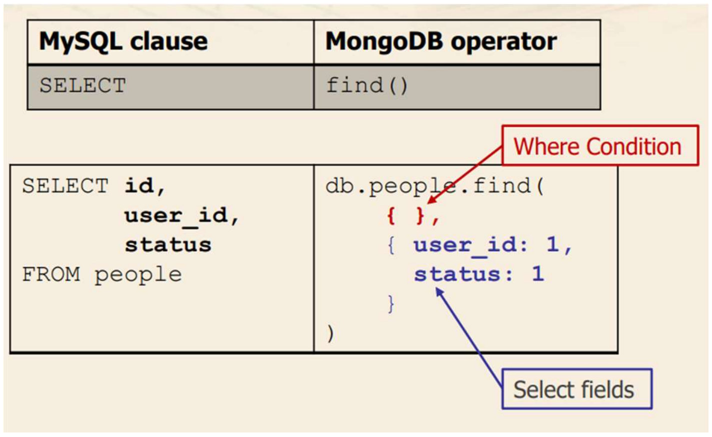
db.people.find({ }, { user_id: 1, status: 1 })

- No conditions, but returns a specific set of fields
  of interest
- Returns only user_id and status of all documents contained in the
  people collection
- Default of fields is false, except for _id

db.people.find({ status: "A", age: 55})

- status = “A” and age = 55
- Returns all documents having status = “A” and age = 55

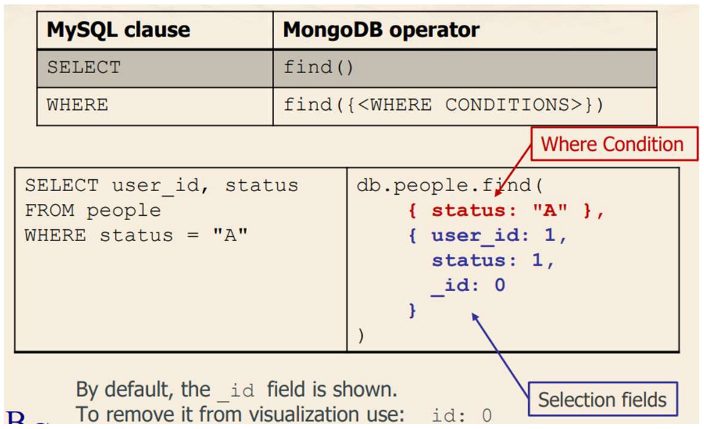
MongoDB: find() operator



MySQL clause	MongoDB operator
SELECT	find()

SELECT **id,**
user_id,
status
FROM people
	db.people.find(
	{ },
	{ **user_id: 1,**
	status: 1
	}
	)

Where Condition
Select fields

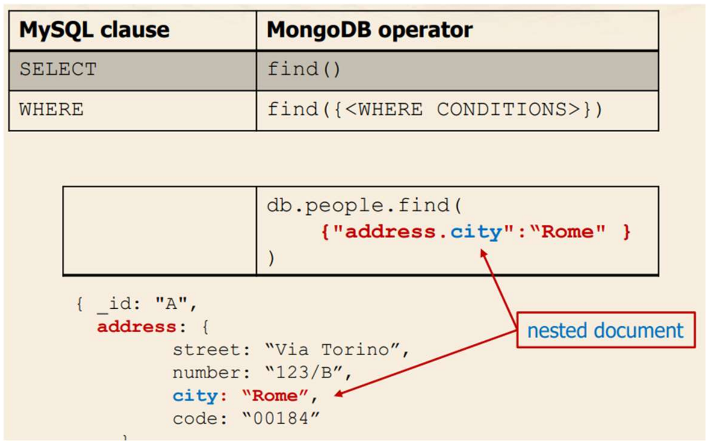


MySQL clause	MongoDB operator
SELECT	find()
WHERE	find({<WHERE CONDITIONS>})

SELECT user_id, status
FROM people
WHERE status = "A"
	db.people.find(
	{ status: "A" },
	{ user_id: 1,
	status: 1,
	_id: 0
	}
	)

Where Condition
Selection fields

By default, the _id field is shown.
To remove it from visualization use: id: 0



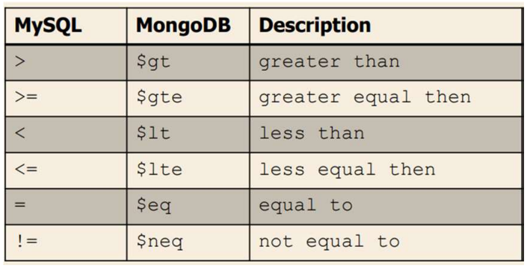
MySQL clause	MongoDB operator
SELECT	find()
WHERE	find({<WHERE CONDITIONS>})

	db.people.find(**{"address.city":"Rome" }**)

nested document

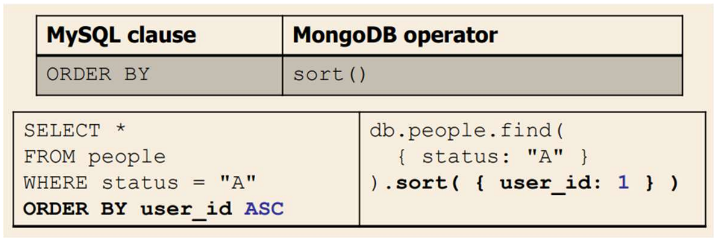
```
{ _id: "A",
  address: {
         street: "Via Torino",
         number: "123/B",
         city: "Rome",
         code: "00184"
```

MongoDB: comparison operators



MySQL	MongoDB	Description
>	$gt	greater than
>=	$gte	greater equal then
<	$lt	less than
<=	$lte	less equal then
=	$eq	equal to
!=	$neq	not equal to

MongoDB: sorting data

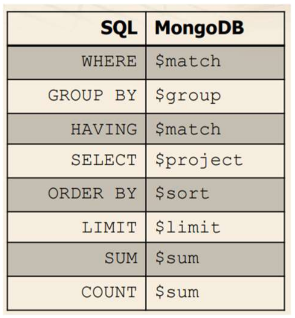


MySQL clause	MongoDB operator
ORDER BY	sort()

```
SELECT *
FROM people
WHERE status = "A"
ORDER BY user_id ASC
```

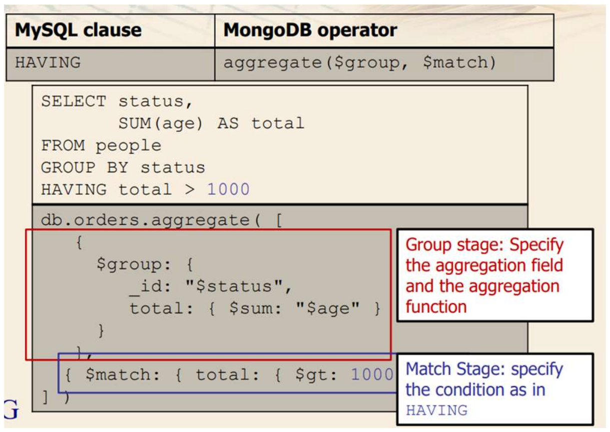
```
db.people.find(
  { status: "A" }
).sort( { user_id: 1 } )
```

MongoDB: Aggregation Frameworks



SQL	MongoDB
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM	$sum
COUNT	$sum

Aggregation in MongoDB: Group By



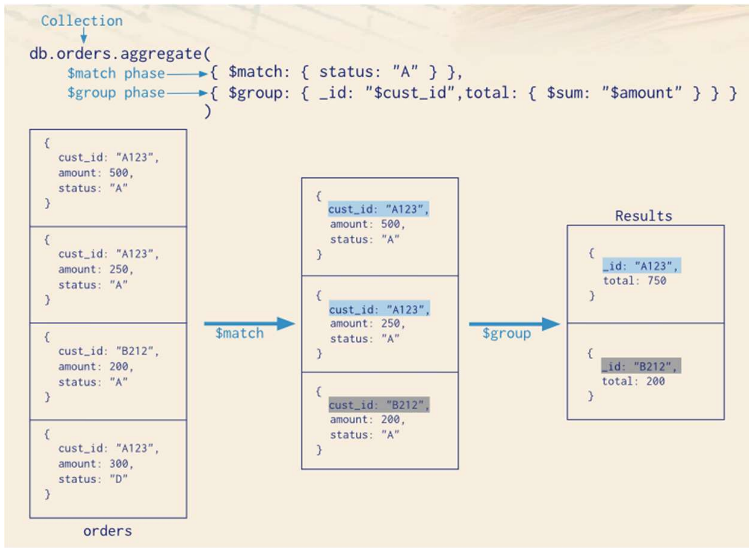
MySQL clause	MongoDB operator
HAVING	aggregate($group, $match)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   },
   { $match: { total: { $gt: 1000
] )
```

Group stage: Specify the aggregation field and the aggregation function

Match Stage: specify the condition as in HAVING



```
db.orders.aggregate(
    { $match: { status: "A" } },
    { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
)
```

Collection
$match phase
$group phase
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }
orders
$match
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
$group
Results
{ _id: "A123", total: 750 }
{ _id: "B212", total: 200 }

Create React App

To learn and test React, you should set up a React Environment on your computer.

This document uses the create-react-app.

The create-react-app tool is an officially supported way to create React applications.

Node.js is required to use create-react-app.

Open your terminal in the directory you would like to create your application.

Run this command to create a React application named my-react-app:

npx create-react-app my-react-app

create-react-app will set up everything you need to run a React application.

Run this command to move to the my-react-app directory:

cd my-react-app

Run this command to execute the React application my-react-app:

npm start

How does React Work?

React creates a VIRTUAL DOM in memory.

Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM.

React only changes what needs to be changed!

React finds out what changes have been made, and changes only what needs to be changed.

Modify the React Application

So far so good, but how do I change the content?

Look in the my-react-app directory, and you will find a src folder. Inside the src folder there is a file called App.js

Try changing the HTML content and save the file.

Replace all the content inside the <div className="App"> with a <h1> element.

```
function App() {
  return (
    <div className="App">
      <h1>Hello World!</h1>
    </div>
  );
}

export default App;
```

See the changes in the browser when you click Save.

SAMPLE PROGRAM QUESTIONS

Structure of the pre requisite database for solving sample programs

Sample of 1 restaurant, similarly create at least 5 different restaurant details.

```
db.collection_name.insertOne({
  "address": {
    "building": "1007",
    "coord": [ -73.856077, 40.848447 ],
    "street": "Morris Park Ave",
    "zipcode": "10462"
  },
  "borough": "Bronx",
  "cuisine": "Bakery",
  "grades": [
    { "date": new Date(1393804800000), "grade": "A", "score": 2 },
    { "date": new Date(1378857600000), "grade": "A", "score": 6 },
    { "date": new Date(1358985600000), "grade": "A", "score": 10 },
    { "date": new Date(1322006400000), "grade": "A", "score": 9 },
    { "date": new Date(1299715200000), "grade": "B", "score": 14 }
  ],
  "name": "Morris Park Bake Shop",
  "restaurant_id": "30075445"
})
```

1. Write a MongoDB query to display all the documents in the collection restaurants.

2. Write a MongoDB query to display the fields restaurant_id, name, borough and cuisine for all the documents in the collection restaurant.

3. Write a MongoDB query to display the field’s restaurant_id, name, borough and cuisine, but exclude the field _id for all the documents in the collection restaurant.

4. Write a MongoDB query to display the first 5 restaurant which is in the borough Bronx.

5. Write a MongoDB query to display the next 5 restaurants after skipping first 5 which are in the borough Bronx.

6. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which contain 'Wil' as first three letters for its name.

7. Write a MongoDB query to find the lowest score for each restaurant.

8. Write a MongoDB query to find the count of restaurants for each cuisine.

LAB PROGRAMS

PROGRAM 1

1a. Using MongoDB, create a collection called transactions in database usermanaged (drop if it already exists) and bulk load the data from a json file, transactions.json

b. Upsert the record from the new file called transactions_upsert.json in Mongodb.

DESCRITPION

"Upsert" is a database operation in MongoDB that combines "update" and "insert" operations. The term "upsert" itself is a portmanteau of "update" and "insert".

When you perform an upsert operation in MongoDB, it attempts to update a document based on a specified query criteria. If the query criteria match an existing document, MongoDB will update that document with the specified update operations. However, if there is no document that matches the query criteria, MongoDB will insert a new document with the specified fields and values.

In summary:

- If the document exists, MongoDB performs an update operation.
- If the document does not exist, MongoDB performs an insert operation.
- Upsert operations are particularly useful when you want to update a document if it exists or insert it if it doesn't exist, without needing to explicitly check for the existence of the document beforehand. This can simplify your code and make it more efficient.

# Step 1: Launch MongoDB shell and switch to the desired database

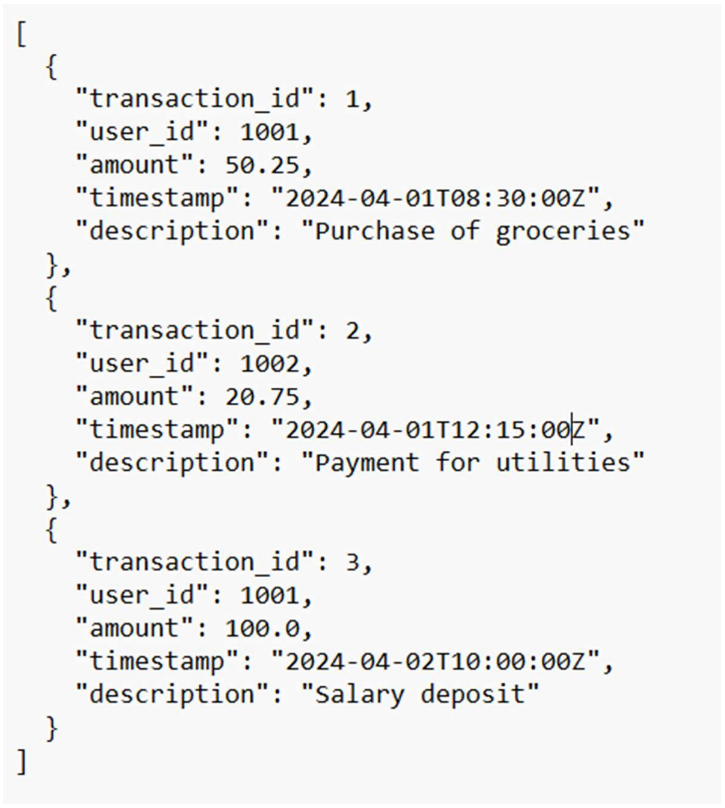
Mongo or mongosh

# Step 2: Switch to the usermanaged database and drop the transactions collection if it exists

use usermanaged

db.transactions.drop()

Create a transaction.json



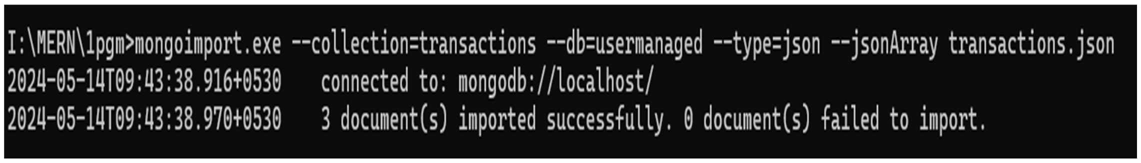
```
[
  {
    "transaction_id": 1,
    "user_id": 1001,
    "amount": 50.25,
    "timestamp": "2024-04-01T08:30:00Z",
    "description": "Purchase of groceries"
  },
  {
    "transaction_id": 2,
    "user_id": 1002,
    "amount": 20.75,
    "timestamp": "2024-04-01T12:15:00Z",
    "description": "Payment for utilities"
  },
  {
    "transaction_id": 3,
    "user_id": 1001,
    "amount": 100.0,
    "timestamp": "2024-04-02T10:00:00Z",
    "description": "Salary deposit"
  }
]
```

# Step 3: Bulk load data from transactions.json into the transactions collection (Open new CMD Terminal and set the path to transactions.json)

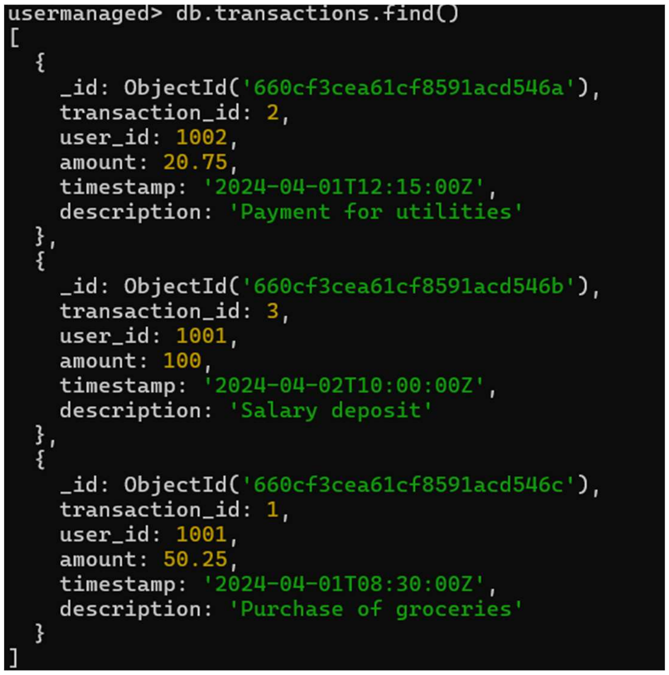
mongoimport.exe --collection=transactions --db=usermanaged --type=json --jsonArray transactions.json

usermanaged> **db.transactions.find()**

OUTPUT:

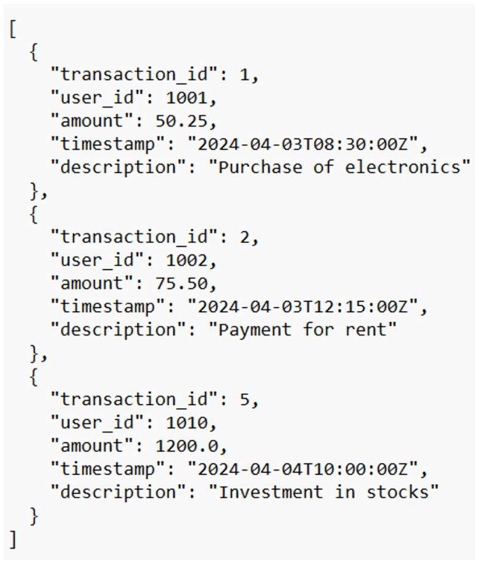


```
I:\MERN\1pgm>mongoimport.exe --collection=transactions --db=usermanaged --type=json --jsonArray transactions.json
2024-05-14T09:43:38.916+0530    connected to: mongodb://localhost/
2024-05-14T09:43:38.970+0530    3 document(s) imported successfully. 0 document(s) failed to import.
```



```
usermanaged> db.transactions.find()
[
  {
    _id: ObjectId('660cf3cea61cf8591acd546a'),
    transaction_id: 2,
    user_id: 1002,
    amount: 20.75,
    timestamp: '2024-04-01T12:15:00Z',
    description: 'Payment for utilities'
  },
  {
    _id: ObjectId('660cf3cea61cf8591acd546b'),
    transaction_id: 3,
    user_id: 1001,
    amount: 100,
    timestamp: '2024-04-02T10:00:00Z',
    description: 'Salary deposit'
  },
  {
    _id: ObjectId('660cf3cea61cf8591acd546c'),
    transaction_id: 1,
    user_id: 1001,
    amount: 50.25,
    timestamp: '2024-04-01T08:30:00Z',
    description: 'Purchase of groceries'
  }
]
```

Create transactions_upsert.json

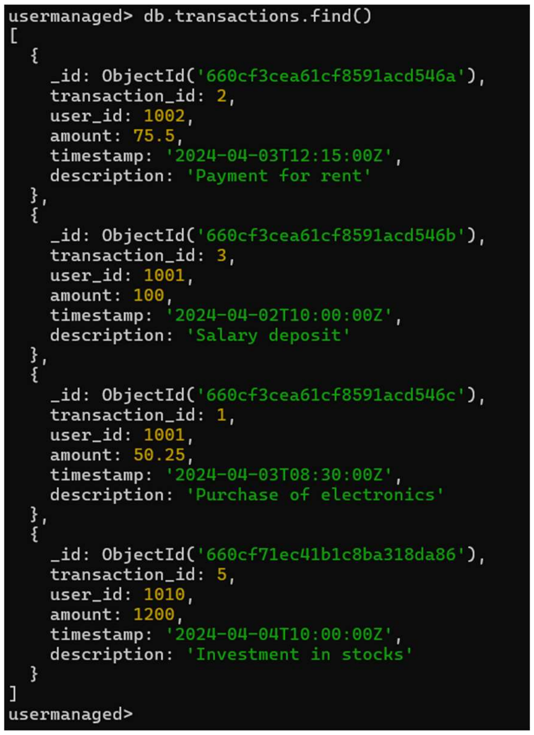


```
[
  {
    "transaction_id": 1,
    "user_id": 1001,
    "amount": 50.25,
    "timestamp": "2024-04-03T08:30:00Z",
    "description": "Purchase of electronics"
  },
  {
    "transaction_id": 2,
    "user_id": 1002,
    "amount": 75.50,
    "timestamp": "2024-04-03T12:15:00Z",
    "description": "Payment for rent"
  },
  {
    "transaction_id": 5,
    "user_id": 1010,
    "amount": 1200.0,
    "timestamp": "2024-04-04T10:00:00Z",
    "description": "Investment in stocks"
  }
]
```

# Step 4: Upsert records from transactions_upsert.json into the transactions collection

mongoimport --db usermanaged --collection transactions --file transactions_upsert.json --jsonArray --upsertFields transaction_id

OUTPUT:



```
usermanaged> db.transactions.find()
[
  {
    _id: ObjectId('660cf3cea61cf8591acd546a'),
    transaction_id: 2,
    user_id: 1002,
    amount: 75.5,
    timestamp: '2024-04-03T12:15:00Z',
    description: 'Payment for rent'
  },
  {
    _id: ObjectId('660cf3cea61cf8591acd546b'),
    transaction_id: 3,
    user_id: 1001,
    amount: 100,
    timestamp: '2024-04-02T10:00:00Z',
    description: 'Salary deposit'
  },
  {
    _id: ObjectId('660cf3cea61cf8591acd546c'),
    transaction_id: 1,
    user_id: 1001,
    amount: 50.25,
    timestamp: '2024-04-03T08:30:00Z',
    description: 'Purchase of electronics'
  },
  {
    _id: ObjectId('660cf71ec41b1c8ba318da86'),
    transaction_id: 5,
    user_id: 1010,
    amount: 1200,
    timestamp: '2024-04-04T10:00:00Z',
    description: 'Investment in stocks'
  }
]
usermanaged>
```

PROGRAM 2

2. Query MongoDB with Conditions: [Create appropriate collection with necessary documents to answer the query]

a. Find any record where Name is Somu

b. Find any record where total payment amount (Payment.Total) is 600.

c. Find any record where price (Transaction.price) is between 300 to 500.

d. Calculate the total transaction amount by adding up Payment.Total in all records.

DESCRIPTION

In a typical business scenario, **"transaction price"** and **"payment total"** are related but represent different aspects of a transaction:

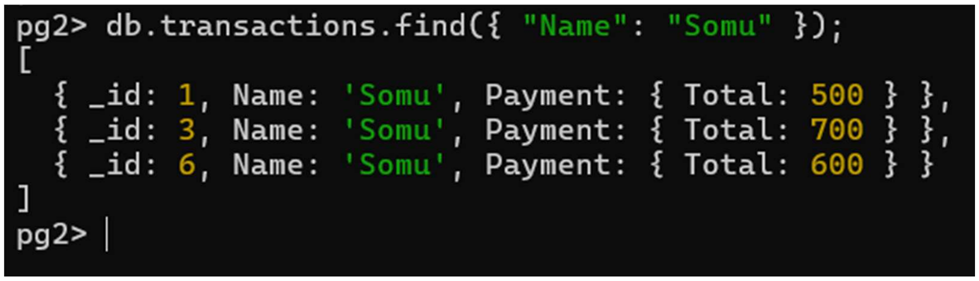
1. **Transaction Price:** This refers to the price or cost associated with a particular transaction. It's the amount of money that is being charged or exchanged for a product or service. For example, in retail setting, if you buy a product for Rs.50, then Rs.50 would be the transaction price.
2. **Payment Total:** This represents the total amount of money paid for a transaction. It includes not only the transaction price but also any additional charges, taxes, or discounts that might be applicable to the transaction. For instance, if you buy a product for Rs.50 but also pay Rs.5 in taxes, then the payment total would be Rs.55.

```
//Create a collection named “transaction”
db.createCollection("transactions")
```

```
// Insert sample documents into the transactions collection

db.transactions.insertMany([
  {
   "_id": 1,
   "Name": "Somu",
   "Payment": { "Total": 500 }
  },
  {
   "_id": 2,
   "Name": "Ravi",
   "Payment": { "Total": 600 }
  },
  {
   "_id": 3,
   "Name": "Somu",
   "Payment": { "Total": 700 }
  },
  {
   "_id": 4,
   "Name": "John",
   "Payment": { "Total": 400 }
  },
  {
   "_id": 5,
   "Name": "David",
   "Payment": { "Total": 800 }
  },
  {
   "_id": 6,
   "Name": "Somu",
   "Payment": { "Total": 600 }
  },
  {
   "_id": 7,
   "Name": "Alex",
   "Payment": { "Total": 450 }
  },
  {
   "_id": 8,
   "Name": "Chris",
   "Transaction": { "price": 350 }
  },
  {
   "_id": 9,
   "Name": "Emma",
   "Transaction": { "price": 400 }
  },
  {
   "_id": 10,
   "Name": "Sophia",
   "Transaction": { "price": 500 }
  },
  {
   "_id": 11,
   "Name": "Olivia",
   "Transaction": { "price": 600 }
  }
]);
```

a. **db.transactions.find({ "Name": "Somu" });**

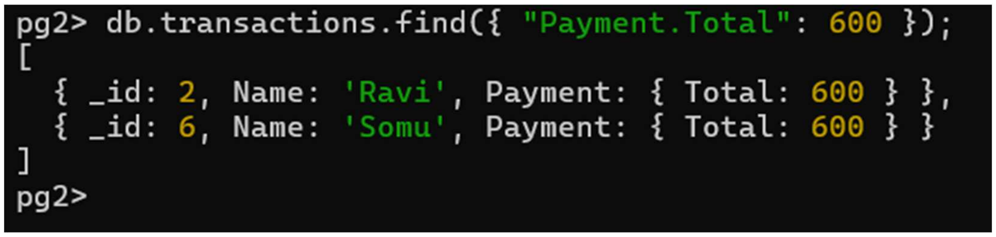
OUTPUT:



```
pg2> db.transactions.find({ "Name": "Somu" });
[
  { _id: 1, Name: 'Somu', Payment: { Total: 500 } },
  { _id: 3, Name: 'Somu', Payment: { Total: 700 } },
  { _id: 6, Name: 'Somu', Payment: { Total: 600 } }
]
pg2> |
```

b. **db.transactions.find({ "Payment.Total": 600 });**

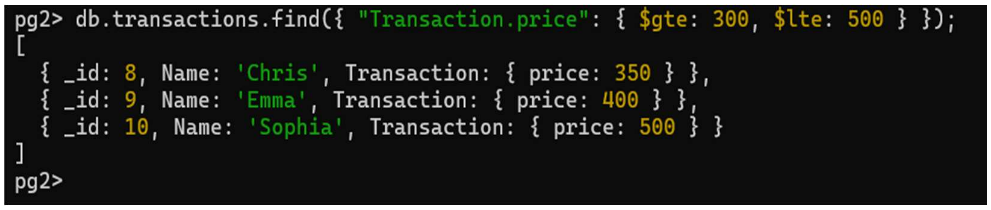
OUTPUT:



```
pg2> db.transactions.find({ "Payment.Total": 600 });
[
  { _id: 2, Name: 'Ravi', Payment: { Total: 600 } },
  { _id: 6, Name: 'Somu', Payment: { Total: 600 } }
]
pg2>
```

c. **db.transactions.find({ "Transaction.price": { $gte: 300, $lte: 500 } });**

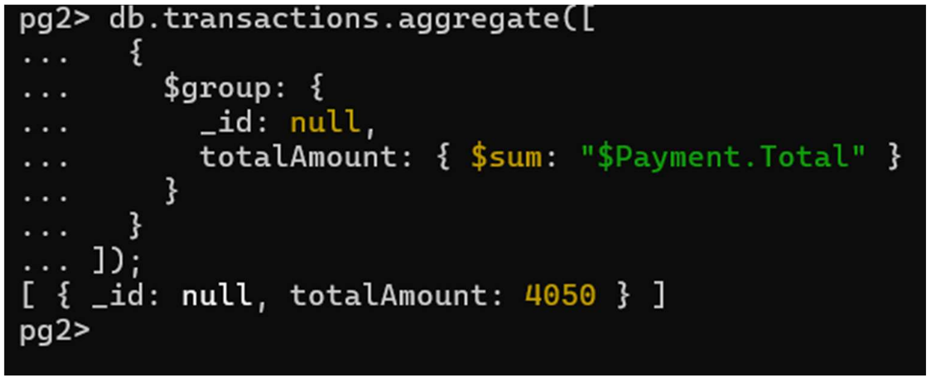
OUTPUT:



```
pg2> db.transactions.find({ "Transaction.price": { $gte: 300, $lte: 500 } });
[
  { _id: 8, Name: 'Chris', Transaction: { price: 350 } },
  { _id: 9, Name: 'Emma', Transaction: { price: 400 } },
  { _id: 10, Name: 'Sophia', Transaction: { price: 500 } }
]
pg2>
```

d. db.transactions.aggregate([
```
 {
  $group: {
    _id: null,
    totalAmount: { $sum: "$Payment.Total" }
  }
 }
]);
```

OUTPUT:



```
pg2> db.transactions.aggregate([
...   {
...     $group: {
...       _id: null,
...       totalAmount: { $sum: "$Payment.Total" }
...     }
...   }
... ]);
[ { _id: null, totalAmount: 4050 } ]
pg2>
```

- **db.transactions.find():** This is a MongoDB query to retrieve documents from the transactions collection.
- **{ "Transaction.price": { $gte: 300, $lte: 500 } }:** This is the query filter criteria. It specifies the condition that documents should meet to be included in the result.
- **"Transaction.price":** This is the field to filter on. It specifies that we're interested in the price field nested within the Transaction subdocument.
- **{ $gte: 300, $lte: 500 }:** This is a range condition using MongoDB query operators.
- **$gte:** Stands for "greater than or equal to". It specifies that the price field must be greater than or equal to 300.
- **$lte:** Stands for "less than or equal to". It specifies that the price field must be less than or equal to 500.
- Combining these, the filter condition means that we want to find documents where the price field within the Transaction subdocument falls within the range from 300 to 500 (inclusive).

PROGRAM 3a

3a. Write a program to check request header for cookies.

DESCRIPTION

The question "Write a program to check request header for cookies" is to create a program that examines incoming HTTP requests and determines whether they contain cookies in their headers.

In the context of web development, cookies are small pieces of data that websites store on a user's computer. When a user visits a website, their browser sends any cookies associated with that website along with the HTTP request. These cookies are typically stored in the request headers.

The task is to write a program, likely using a web framework such as Express.js for Node.js, that intercepts incoming HTTP requests and inspects their headers. Specifically, the program should check if the "Cookie" header is present in the request. If it is, the program should log the cookies to the console or perform any other desired actions based on the presence of cookies.

Overall, the goal is to create a simple server-side application that can detect the presence of cookies in incoming HTTP requests.

1. **Initialize a New Node.js Project:** Open your terminal or command prompt, navigate to the directory you created, and run the following command to initialize a new Node.js project:

   ```
   npm init –y
   ```

2. **Install Express.js:** Next, install the Express.js framework. In your terminal, run the following command:

   ```
   npm install express
   ```

3. **Create Your JavaScript File: Create a new JavaScript file (e.g., app.js) in your project directory and insert the provided code into it.**

```
const express = require('express');
const app = express();

// Middleware to log request headers
app.use((req, res, next) => {
   // Check if 'Cookie' header exists in the request
   if (req.headers.cookie) {
      console.log('Cookies found in the request:');
      console.log(req.headers.cookie);
   } else {
      console.log('No cookies found in the request.');
   }
   next();
});

// Route handler
app.get('/', (req, res) => {
   res.send('Hello World!');
});

// Start the server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
   console.log(`Server is listening on port ${PORT}`);
});
```

This code sets up a basic Express.js server,below is the explanation of the code:

- **Imports**: It imports the express module.
- **App Initialization**: It initializes an Express application by calling express() and assigns it to the variable app.
- **Middleware**: It defines a middleware function using app.use(). This middleware logs the request headers. If the request contains a 'Cookie' header, it prints out the cookie content. Otherwise, it logs that no cookies were found. next() is called to pass control to the next middleware in the stack.
- **Route Handler**: It defines a route handler for the root path ('/'). When a GET request is made to the root path, it sends back the response 'Hello World!'.
- **Server Initialization:** It starts the server listening on either the port specified in the environment variable PORT or port 3000. When the server starts, it logs a message indicating which port it's listening on.

In summary, this code creates a server using Express.js, logs request headers, and responds with 'Hello World!' when a GET request is made to the root path.

Run Your Program: In your terminal, navigate to the directory containing your JavaScript file (e.g., app.js). Then, run the following command to execute your program:

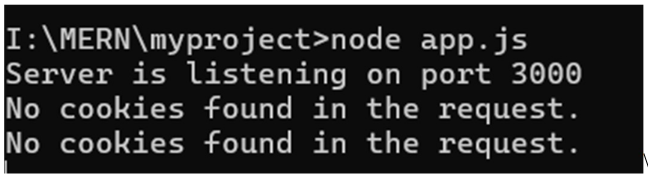
```
node app.js
```

open(**http://localhost:3000)**

- This command starts the Express.js server, and you should see a message in the terminal indicating that the server is listening on a port (by default, port 3000).

- **Test Your Program**: Open a web browser or use a tool like cURL or Postman to make HTTP requests to your server. You can send requests to **http://localhost:3000** (or the port specified in your terminal if different) to trigger the middleware that checks for cookies. The program will log any cookies found in the request headers to the terminal.

OUTPUT:



```
I:\MERN\myproject>node app.js
Server is listening on port 3000
No cookies found in the request.
No cookies found in the request.
```

\\\

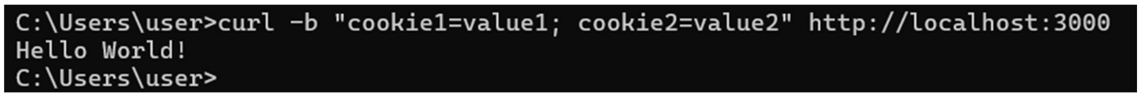
To Add cookies

- **Using cURL:**

If you're using cURL, you can add cookies to the request header using the -b or --cookie option followed by a string containing the cookie(s) you want to include. (Use new terminal window)

curl -b "cookie1=value1; cookie2=value2" http://localhost:3000

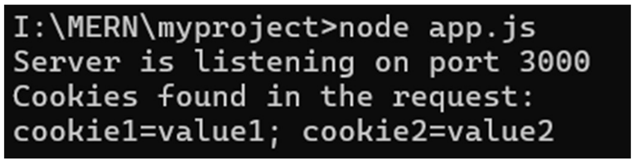
OUTPUT:



```
C:\Users\user>curl -b "cookie1=value1; cookie2=value2" http://localhost:3000
Hello World!
C:\Users\user>
```

Press Enter: After typing the cURL command with the appropriate options, press Enter on your keyboard to execute the command.

OUTPUT:



```
I:\MERN\myproject>node app.js
Server is listening on port 3000
Cookies found in the request:
cookie1=value1; cookie2=value2
```

PROGRAM 3b

3 b. write node.js program to print the a car object properties, delete the second property and get length of the object.

```
// Define an array of car objects
let cars = [
    { make: 'Toyota', model: 'Corolla', year: 2020, color: 'Blue' },
    { make: 'Honda', model: 'Civic', year: 2019, color: 'Red' },
    { make: 'Ford', model: 'Mustang', year: 2021, color: 'Black' }
];

// Function to print the properties of an object
function printProperties(obj) {
    for (let key in obj) {
        console.log(`${key}: ${obj[key]}`);
    }
}

// Function to get the length of an object
function getObjectLength(obj) {
    return Object.keys(obj).length;
}

// Iterate over each car in the cars array
cars.forEach((car, index) => {
    console.log(`Car ${index + 1} properties:`);
    printProperties(car);
    console.log();
    // Delete the second property (in this case, 'model')
    let propertyKeys = Object.keys(car);
    delete car[propertyKeys[1]];



    // Print properties after deletion
    console.log(`Car ${index + 1} properties after deleting the second property:`);
    printProperties(car);
    console.log();

    // Get and print the length of the object
    let length = getObjectLength(car);
    console.log(`Length of car ${index + 1} object: ${length}`);
    console.log('--------------------------');
});
```

Explanation:

Create an array of car objects: The cars array contains multiple car objects, each with properties make, model, year, and color.

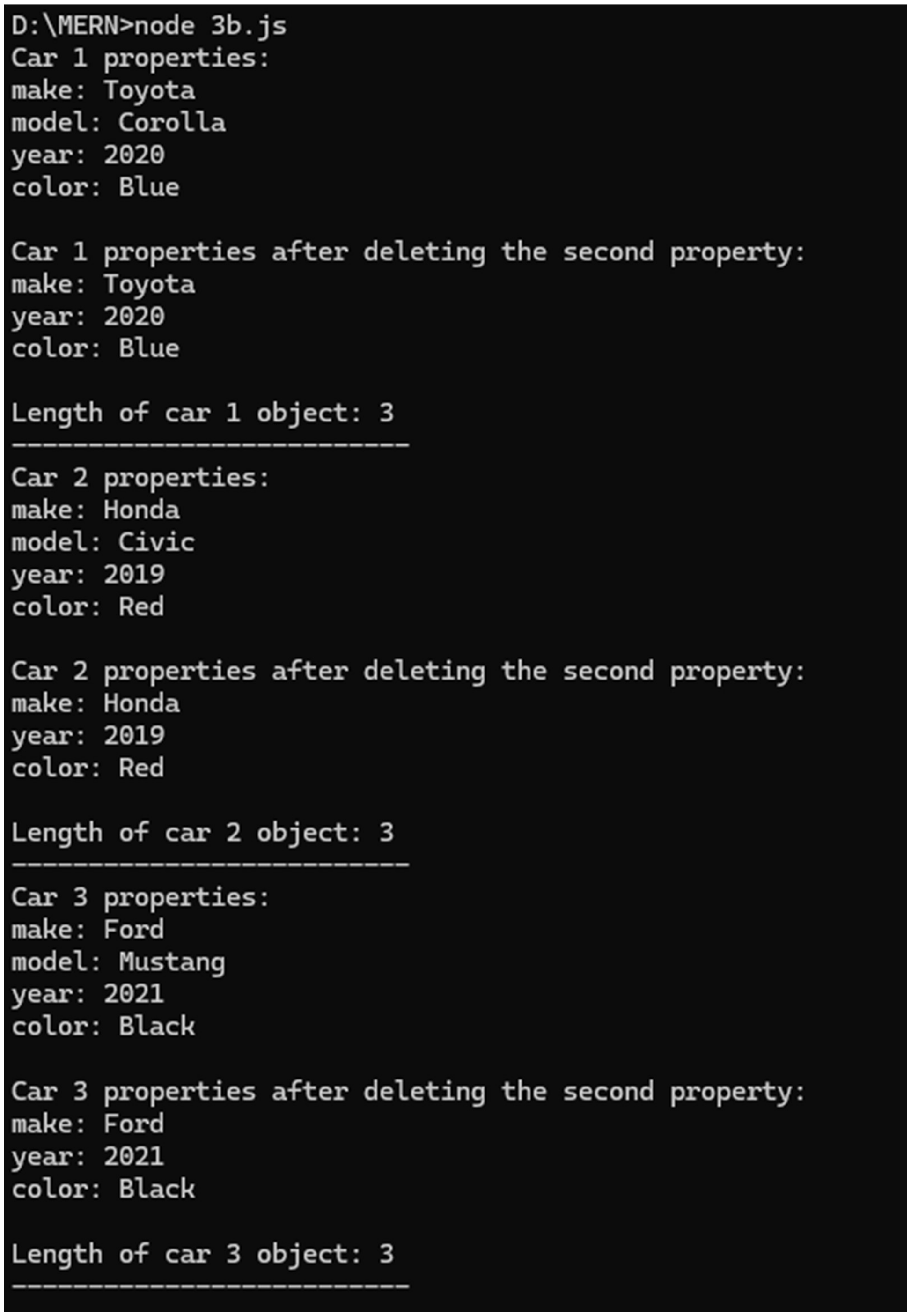
Print the properties: The printProperties function iterates over an object's keys and prints each key-value pair.

Delete the second property of each car: The code iterates over the cars array, and for each car, it gets the property keys using Object.keys(car) and deletes the second property using delete car[propertyKeys[1]].

Print the properties again: After deleting the second property, the properties of each car are printed again to confirm the deletion.

Get the length of each car object: The length (number of properties) is calculated using Object.keys(obj).length and printed for each car.

OUTPUT:



```
D:\MERN>node 3b.js
Car 1 properties:
make: Toyota
model: Corolla
year: 2020
color: Blue

Car 1 properties after deleting the second property:
make: Toyota
year: 2020
color: Blue

Length of car 1 object: 3
--------------------------
Car 2 properties:
make: Honda
model: Civic
year: 2019
color: Red

Car 2 properties after deleting the second property:
make: Honda
year: 2019
color: Red

Length of car 2 object: 3
--------------------------
Car 3 properties:
make: Ford
model: Mustang
year: 2021
color: Black

Car 3 properties after deleting the second property:
make: Ford
year: 2021
color: Black

Length of car 3 object: 3
--------------------------
```

PROGRAM 4a

4a. Read the data of a student containing usn, name, sem, year_of_admission from node js and store it in the mongodb

1. **Set up Node.js:** Create a Node.js project and install the necessary dependencies, such as mongodb package, using npm.

```
npm init
npm install mongodb
npm install readline-sync
```

2. **Write a Node.js script**: Create a Node.js script to read student data from a source (e.g., a file or user input) and store it in MongoDB.

```
const { MongoClient } = require('mongodb');
const readlineSync = require('readline-sync');

// MongoDB connection URL and database name
const url = 'mongodb://localhost:27017';
const dbName = 'schoolDB';

// Function to get student data from the user
function getStudentData() {
   return {
      usn: readlineSync.question('Enter USN: '),
      name: readlineSync.question('Enter Name: '),
      sem: readlineSync.question('Enter Semester: '),
      year_of_admission: readlineSync.question('Enter Year of Admission: ')
   };
}

// Function to insert student data into MongoDB
```

```
async function insertStudentData(studentData) {
   const client = new MongoClient(url, { useNewUrlParser: true,
useUnifiedTopology: true });
   try {
      await client.connect();
      const db = client.db(dbName);
      const result = await db.collection('students').insertOne(studentData);
      console.log('Student data inserted with _id:', result.insertedId);
   } catch (err) {
      console.error('Error inserting data:', err);
   } finally {
      await client.close();
   }
}

// Main function to run the script
(async function() {
   const studentData = getStudentData();
   await insertStudentData(studentData);
})

();
```

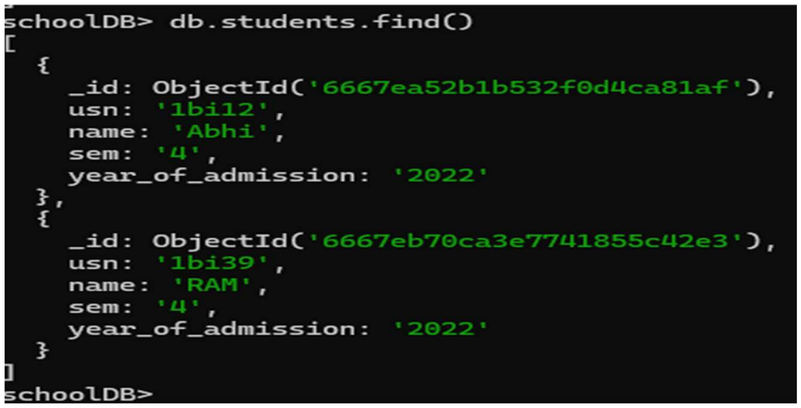
Open another terminal

Connect to MongoDB

mongosh

use studentsDB

db.students.find()



```
schoolDB> db.students.find()
[
  {
    _id: ObjectId('6667ea52b1b532f0d4ca81af'),
    usn: '1bi12',
    name: 'Abhi',
    sem: '4',
    year_of_admission: '2022'
  },
  {
    _id: ObjectId('6667eb70ca3e7741855c42e3'),
    usn: '1bi39',
    name: 'RAM',
    sem: '4',
    year_of_admission: '2022'
  }
]
schoolDB>
```

Program 4b.

4b. For a partial name given in node js, search all the names from mongodb student documents created in Question(a)

```
const { MongoClient } = require('mongodb');
const readline = require('readline');

// Create readline interface for user input
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

// Connection URI
const uri = 'mongodb://localhost:27017';

async function main() {
  const client = new MongoClient(uri);

  try {
    // Connect to MongoDB
    await client.connect();

    // Ask user for partial name
    rl.question('Enter partial name to search: ', async (partialName) => {
      // Select the database
      const db = client.db('schoolDB'); // Replace 'studentsDB' with your database name

      // Get the students collection
      const collection = db.collection('students'); // Replace 'students' with your collection
name

      // Search for documents with names containing the partial name
      const query = { name: { $regex: partialName, $options: 'i' } }; // Case-insensitive
regex
      const students = await collection.find(query).toArray();

      // Print the matching student documents
      console.log(`Matching student documents with partial name "${partialName}":`);
      console.log(students);

      // Close the connection
      await client.close();

      // Close readline interface
      rl.close();
    });
  } catch (error) {
    console.error('Error:', error);
    rl.close();
  }
}

main();
```

PROGRAM EXPLANATION

Imports: Imports mongodb for MongoDB operations and readline for reading user input from the console.

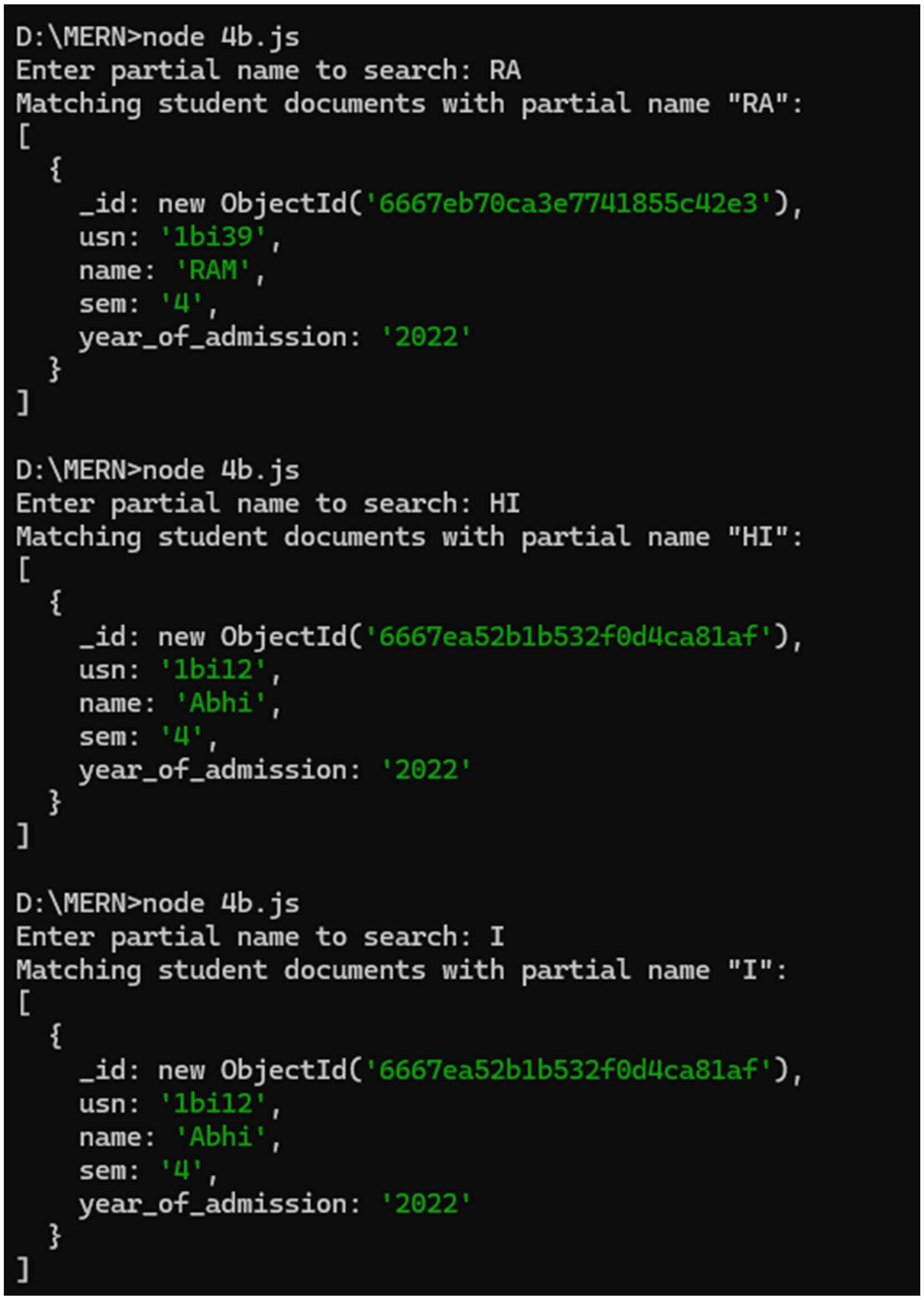
Readline Interface: Sets up a readline interface to handle user input.

MongoDB Connection URI: Defines the URI to connect to the MongoDB server.

Main Function:

- Connects to MongoDB using the MongoClient.
- Prompts the User to enter a partial name to search.
- Selects the Database and Collection (schoolDB and students).
- Constructs a Query to find documents with names containing the partial name using a case-insensitive regex.
- Fetches and Prints Matching Documents.
- Closes the MongoDB Connection and Readline Interface.

OUTPUT:



```
D:\MERN>node 4b.js
Enter partial name to search: RA
Matching student documents with partial name "RA":
[
  {
    _id: new ObjectId('6667eb70ca3e7741855c42e3'),
    usn: '1bi39',
    name: 'RAM',
    sem: '4',
    year_of_admission: '2022'
  }
]

D:\MERN>node 4b.js
Enter partial name to search: HI
Matching student documents with partial name "HI":
[
  {
    _id: new ObjectId('6667ea52b1b532f0d4ca81af'),
    usn: '1bi12',
    name: 'Abhi',
    sem: '4',
    year_of_admission: '2022'
  }
]

D:\MERN>node 4b.js
Enter partial name to search: I
Matching student documents with partial name "I":
[
  {
    _id: new ObjectId('6667ea52b1b532f0d4ca81af'),
    usn: '1bi12',
    name: 'Abhi',
    sem: '4',
    year_of_admission: '2022'
  }
]
```

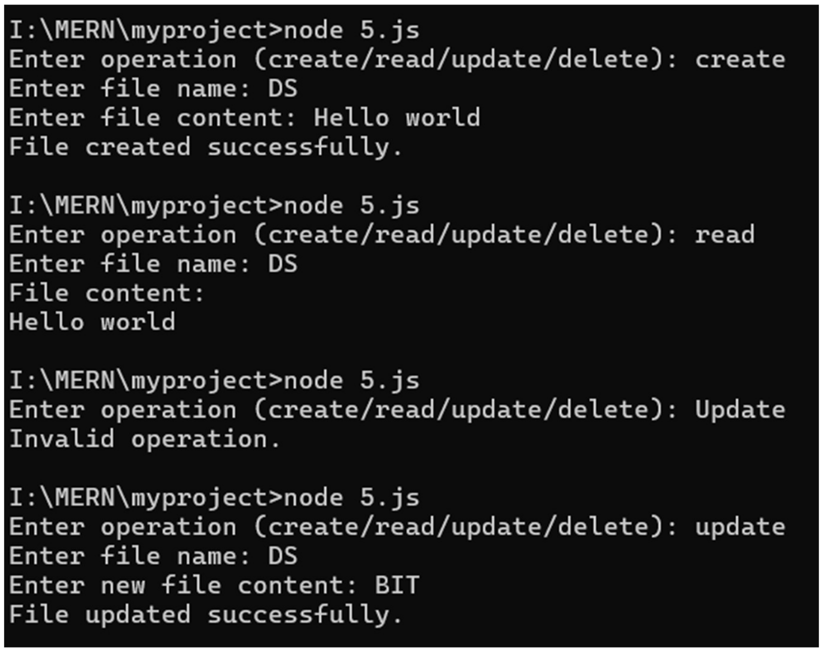
Program 5

5. Implement all CRUD operations on a File System using Node JS

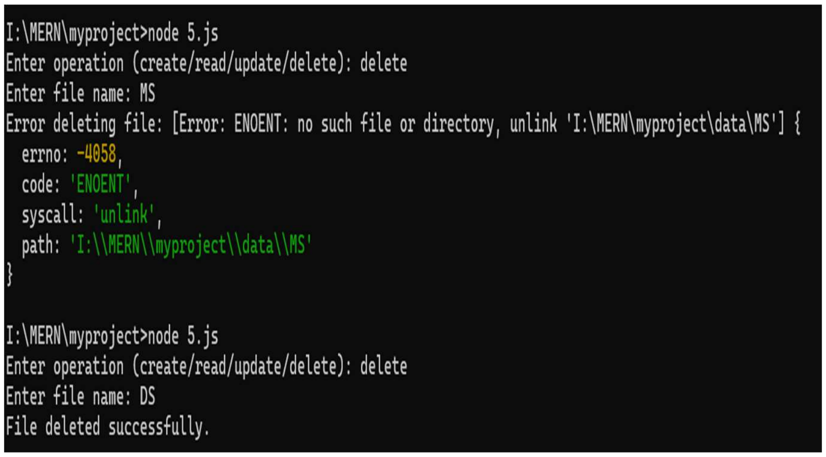
```
const fs = require('fs');
const readline = require('readline');
const path = require('path');

// Create readline interface for user input
const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

// Define the directory path where files will be stored
const directoryPath = './data';

// Create directory if it does not exist
if (!fs.existsSync(directoryPath)) {
    fs.mkdirSync(directoryPath);
}

// Function to create a file
function createFile(fileName, data) {
    const filePath = path.join(directoryPath, fileName);
    fs.writeFile(filePath, data, (err) => {
        if (err) {
            console.error('Error creating file:', err);
            return;
        }
        console.log('File created successfully.');
        rl.close();
    });
}

// Function to read a file
```

```
function readFile(fileName) {
    const filePath = path.join(directoryPath, fileName);
    fs.readFile(filePath, 'utf8', (err, data) => {
        if (err) {
            console.error('Error reading file:', err);
            return;
        }
        console.log('File content:');
        console.log(data);
        rl.close();
    });
}

// Function to update a file
function updateFile(fileName, newData) {
    const filePath = path.join(directoryPath, fileName);
    fs.writeFile(filePath, newData, (err) => {
        if (err) {
            console.error('Error updating file:', err);
            return;
        }
        console.log('File updated successfully.');
        rl.close();
    });
}

// Function to delete a file
function deleteFile(fileName) {
    const filePath = path.join(directoryPath, fileName);
    fs.unlink(filePath, (err) => {
        if (err) {
            console.error('Error deleting file:', err);
            return;
        }
        console.log('File deleted successfully.');
        rl.close();
    });
}

// Prompt user for operation
rl.question('Enter operation (create/read/update/delete): ', (operation) => {
  switch (operation) {
    case 'create':
      rl.question('Enter file name: ', (fileName) => {
        rl.question('Enter file content: ', (data) => {
          createFile(fileName, data);
        });
      });
      break;
    case 'read':
      rl.question('Enter file name: ', (fileName) => {
        readFile(fileName);
      });
      break;
    case 'update':
      rl.question('Enter file name: ', (fileName) => {
        rl.question('Enter new file content: ', (newData) => {
          updateFile(fileName, newData);
        });
      });
      break;
    case 'delete':
      rl.question('Enter file name: ', (fileName) => {
        deleteFile(fileName);
      });
      break;
    default:
      console.log('Invalid operation.');
      rl.close();
  }
});
```

OUTPUT:



```
I:\MERN\myproject>node 5.js
Enter operation (create/read/update/delete): create
Enter file name: DS
Enter file content: Hello world
File created successfully.

I:\MERN\myproject>node 5.js
Enter operation (create/read/update/delete): read
Enter file name: DS
File content:
Hello world

I:\MERN\myproject>node 5.js
Enter operation (create/read/update/delete): Update
Invalid operation.

I:\MERN\myproject>node 5.js
Enter operation (create/read/update/delete): update
Enter file name: DS
Enter new file content: BIT
File updated successfully.
```



```
I:\MERN\myproject>node 5.js
Enter operation (create/read/update/delete): delete
Enter file name: MS
Error deleting file: [Error: ENOENT: no such file or directory, unlink 'I:\MERN\myproject\data\MS'] {
  errno: -4058,
  code: 'ENOENT',
  syscall: 'unlink',
  path: 'I:\\MERN\\myproject\\data\\MS'
}

I:\MERN\myproject>node 5.js
Enter operation (create/read/update/delete): delete
Enter file name: DS
File deleted successfully.
```

PROGRAM 6.

6. Develop the application that sends fruit name and price data from client side to Node.js server using Ajax

server.js

```
// server.js

const express = require('express');
const bodyParser = require('body-parser');
const path = require('path'); // Import path module to handle file paths
const app = express();
const PORT = 3000;

// Middleware
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

// Route to serve index.html when accessing root route
app.get('/', (req, res) => {
   res.sendFile(path.join(__dirname, 'index.html'));
});

// Route to receive data from client
app.post('/fruitData', (req, res) => {
   const { name, price } = req.body;
   console.log(`Received data from client - Name: ${name}, Price: ${price}`);
   // Here you can process the received data as per your requirements
   // For now, let's just send a success response
   res.status(200).send('Data received successfully');
});
```

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

Client-Side (HTML & JavaScript)

1. Create HTML Form: Create an HTML form to capture the fruit name and price.
2. Send Data via Ajax: Use JavaScript to send the captured data to the Node.js server using Ajax.

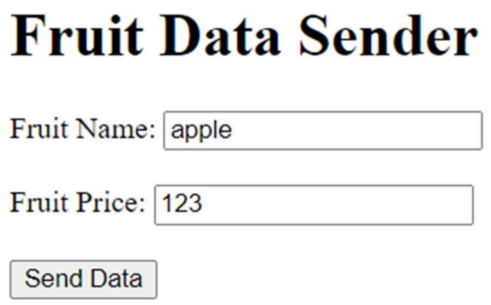
```
<!-- index.html -->

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Fruit Data Sender</title>
</head>
<body>
    <h1>Fruit Data Sender</h1>
    <label for="fruitName">Fruit Name:</label>
    <input type="text" id="fruitName" name="fruitName"><br><br>
    <label for="fruitPrice">Fruit Price:</label>
    <input type="text" id="fruitPrice" name="fruitPrice"><br><br>
    <button onclick="sendData()">Send Data</button>

    <script>
        function sendData() {
            const name = document.getElementById('fruitName').value;
            const price = document.getElementById('fruitPrice').value;

            const data = {
                name: name,
                price: price
            };

            // Using Ajax to send data to server
            const xhr = new XMLHttpRequest();
            xhr.open('POST', '/fruitData', true);
            xhr.setRequestHeader('Content-Type', 'application/json');
            xhr.send(JSON.stringify(data));

            xhr.onreadystatechange = function () {
                if (xhr.readyState === 4 && xhr.status === 200) {
                    console.log('Data sent successfully');
                    // You can handle the success response here
                }
            };
        }
    </script>
</body>
</html>
```

Running the Application

1. Create a Directory.
2. Create server.js in the respective directory.
3. Create index.html in the respective directory.
4. Open CMD and Navigate to your project directory where server.js is located.
5. Run the following command to install Express: **npm install express**
6. Then start your Node.js server: node server.js

7. Open the HTML file in a web browser. You should see a form to input fruit name and price.
8. Enter the fruit details and click the submit button. The data will be sent to the Node.js server using Ajax.
9. Check the server console to verify that the fruit data is received successfully.

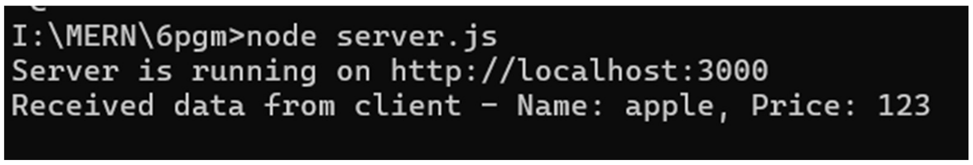
OUTPUT 1:



Fruit Data Sender

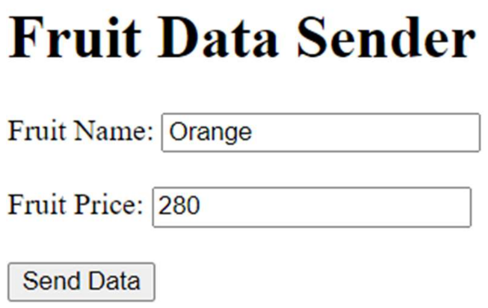
Fruit Name: apple

Fruit Price: 123



```
I:\MERN\6pgm>node server.js
Server is running on http://localhost:3000
Received data from client - Name: apple, Price: 123
```

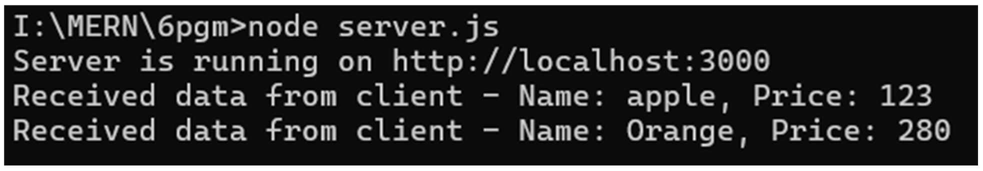
OUTPUT 2:



Fruit Data Sender

Fruit Name: Orange

Fruit Price: 280



```
I:\MERN\6pgm>node server.js
Server is running on http://localhost:3000
Received data from client - Name: apple, Price: 123
Received data from client - Name: Orange, Price: 280
```

PROGRAM 7

7. Develop an authentication mechanism with email_id and password using HTML and Express JS (POST method)

1. **Create a folder for your project and initialize npm:**

```
mkdir authentication-app
cd authentication-app
npm init
```

2. **Install required dependencies:**

```
npm install express body-parser

npm install bcryptjs

npm install mongoose
```

3. **Create your server file (app.js):**
   // app.js

```
const express = require('express');
const bodyParser = require('body-parser');
const bcrypt = require('bcryptjs');
const mongoose = require('mongoose');

const app = express();
const PORT = 3000;

// Connect to MongoDB
mongoose.connect('mongodb://localhost:27017/authentication', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});
const db = mongoose.connection;
db.on('error', console.error.bind(console, 'MongoDB connection error:'));
db.once('open', () => {
  console.log('Connected to MongoDB');
```

```
});

// Define user schema
const userSchema = new mongoose.Schema({
  email: { type: String, unique: true, required: true },
  password: { type: String, required: true },
});
const User = mongoose.model('User', userSchema);

// Middleware
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

// Routes
// Home route
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/index.html');
});

// Signup route (POST method)
app.post('/signup', async (req, res) => {
  const { email, password } = req.body;
  // Simple validation
  if (!email || !password) {
    return res.status(400).send('Email and password are required');
  }
  try {
    // Check if email already exists
    const existingUser = await User.findOne({ email });
    if (existingUser) {
      return res.status(400).send('User already exists');
    }
    // Hash password
    const hashedPassword = await bcrypt.hash(password, 10);
    // Create new user
    const newUser = new User({ email, password: hashedPassword });
    await newUser.save();
    res.status(201).send('User created successfully');
  } catch (error) {
    console.error(error);
    res.status(500).send('Server error');
  }
});

// Login route (POST method)
app.post('/login', async (req, res) => {
  const { email, password } = req.body;
  // Simple validation
  if (!email || !password) {
    return res.status(400).send('Email and password are required');
  }
  try {
    // Find user by email
    const user = await User.findOne({ email });
    if (!user) {
      return res.status(401).send('Invalid credentials');
    }
    // Compare passwords
    const passwordMatch = await bcrypt.compare(password, user.password);
    if (!passwordMatch) {
      return res.status(401).send('Invalid credentials');
    }
    res.status(200).send('Login successful');
  } catch (error) {
    console.error(error);
    res.status(500).send('Server error');
  }
});

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

4. **Create your HTML file (index.html):**

```
<!-- index.html -->

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Authentication App</title>
</head>
<body>
  <h1>Sign Up</h1>
  <form action="/signup" method="POST">
    <input type="email" name="email" placeholder="Email" required><br>
    <input type="password" name="password" placeholder="Password" required><br>
    <button type="submit">Sign Up</button>
  </form>

  <h1>Login</h1>
  <form action="/login" method="POST">
    <input type="email" name="email" placeholder="Email" required><br>
    <input type="password" name="password" placeholder="Password" required><br>
    <button type="submit">Login</button>
  </form>
</body>
</html>
```

3. **Run the code**

```
node app.js
```

4. **Check the database**

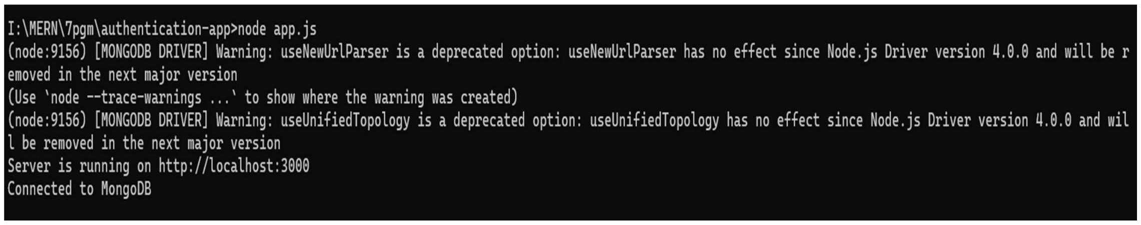
```
Mongosh
```

(if not connected “**net start MongoDB”)**

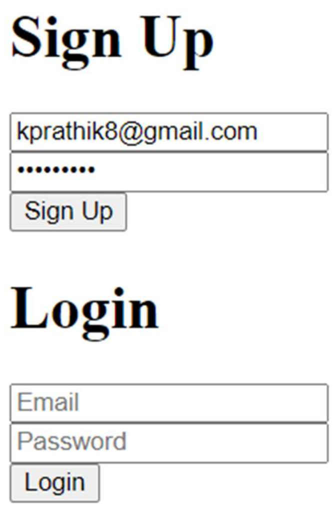
use authentication # Assuming 'authentication' is your database name

db.users.find()

OUTPUT:



```
I:\MERN\7pgm\authentication-app>node app.js
(node:9156) [MONGODB DRIVER] Warning: useNewUrlParser is a deprecated option: useNewUrlParser has no effect since Node.js Driver version 4.0.0 and will be r
emoved in the next major version
(Use `node --trace-warnings ...` to show where the warning was created)
(node:9156) [MONGODB DRIVER] Warning: useUnifiedTopology is a deprecated option: useUnifiedTopology has no effect since Node.js Driver version 4.0.0 and wil
l be removed in the next major version
Server is running on http://localhost:3000
Connected to MongoDB
```



Sign Up

kprathik8@gmail.com

•••••••••

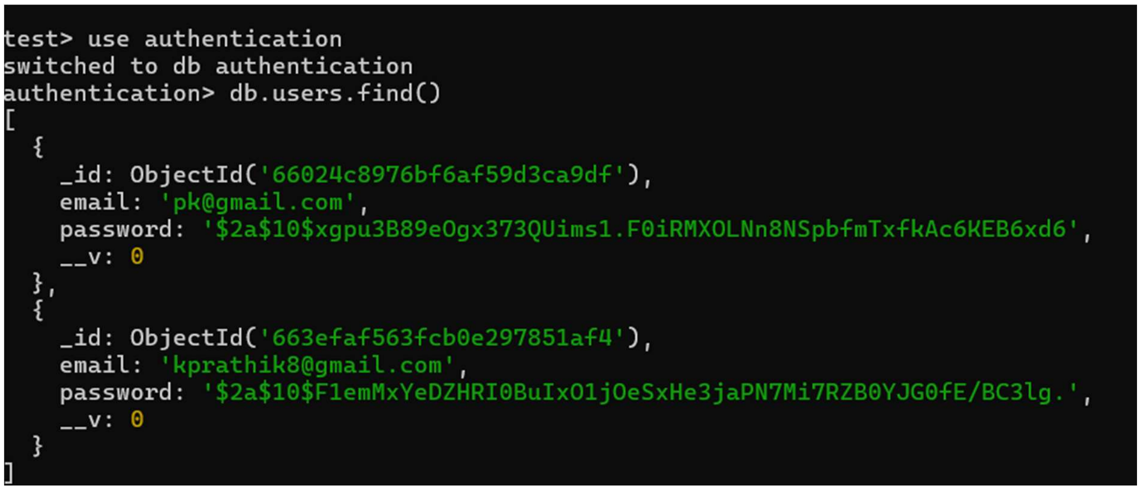
Sign Up

Login

Email

Password

Login



```
test> use authentication
switched to db authentication
authentication> db.users.find()
[
  {
    _id: ObjectId('66024c8976bf6af59d3ca9df'),
    email: 'pk@gmail.com',
    password: '$2a$10$xgpu3B89eOgx373QUims1.F0iRMXOLNn8NSpbfmTxfkAc6KEB6xd6',
    __v: 0
  },
  {
    _id: ObjectId('663efaf563fcb0e297851af4'),
    email: 'kprathik8@gmail.com',
    password: '$2a$10$F1emMxYeDZHRI0BuIxO1jOeSxHe3jaPN7Mi7RZB0YJG0fE/BC3lg.',
    __v: 0
  }
]
```

The code provided is using Express.js, a Node.js web application framework, to handle HTTP requests. Express.js is a popular framework for building web applications and APIs in Node.js.

In the code snippets provided, Express.js is used to define routes, handle HTTP requests (such as POST requests for signup and login), and interact with the MongoDB database using Mongoose, which is an ODM (Object Data Modeling) library for MongoDB and Node.js.

Here's a summary of how Express.js is used in the code:

1. **Defining Routes:** Express.js is used to define routes for different HTTP endpoints (e.g., /signup and /login).
2. **Handling Requests:** Express.js middleware functions are used to handle incoming HTTP requests (e.g., parsing request bodies with body-parser middleware).
3. **Responding to Requests:** Express.js is used to send responses back to clients based on the request (e.g., sending status codes and messages indicating success or failure of operations).
4. **Starting the Server:** Express.js is used to start the HTTP server and listen for incoming requests on a specified port.

Overall, Express.js simplifies the process of building web applications by providing a set of powerful features and utilities for handling HTTP requests and responses.

PROGRAM 8

8. Develop two routes: find_prime_100 and find_cube_100 which prints prime numbers less than 100 and cubes less than 100 using Express JS routing mechanism

DESCRIPTION

The routing mechanism in Express.js defines how your application responds to client requests based on the URL and HTTP method (GET, POST, PUT, DELETE, etc.). Express.js provides a simple and flexible routing system that allows you to define routes for different endpoints of your application.

Here's how the routing mechanism works in Express.js:

1. **Defining Routes:** You define routes using the app.METHOD() functions, where METHOD is the HTTP method of the request (e.g., GET, POST, PUT, DELETE). These functions take two arguments: the URL pattern (or route path) and a callback function that specifies what should happen when a request matches that route.
app.METHOD(path, callback);
2. **Matching Requests:** When a client makes a request to your server, Express.js matches the requested URL and HTTP method against the defined routes. If there's a match, Express.js executes the corresponding callback function.
3. **Executing Middleware:** In Express.js, you can attach one or more middleware functions to a route. Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle. Middleware functions can perform tasks such as data validation, authentication, logging, etc.
4. **Handling Responses:** Inside the route's callback function, you typically send a response back to the client using methods of the res object (e.g., res.send(), res.json(), res.render()). This response can be in various formats, such as HTML, JSON, or others, depending on the client's request and the application's requirements.

Create app.js file

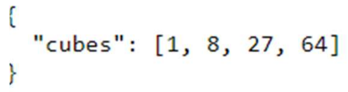
```
const express = require('express');
const app = express();
const PORT = 3000;

// Route to find prime numbers less than 100
app.get('/find_prime_100', (req, res) => {
  const primes = [];
  for (let i = 2; i < 100; i++) {
    let isPrime = true;
    for (let j = 2; j <= Math.sqrt(i); j++) {
      if (i % j === 0) {
        isPrime = false;
        break;
      }
    }
    if (isPrime) {
      primes.push(i);
    }
  }
  res.json({ primeNumbers: primes });
});

// Route to find cubes less than 100
app.get('/find_cube_100', (req, res) => {
  const cubes = [];
  for (let i = 1; i < 100; i++) {
    const cube = i * i * i;
    if (cube < 100) {
      cubes.push(cube);
    } else {
      break; // No need to continue if cube exceeds 100
    }
  }
  res.json({ cubes: cubes });
});

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

Step 1: npm install init

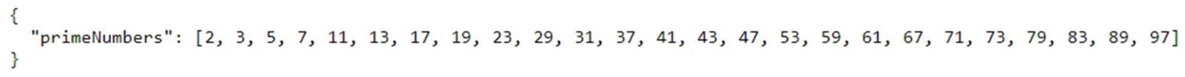
Step 2: npm insall express

- Visiting http://localhost:3000/find_prime_100 will return an array of prime numbers less than 100 in JSON format.
- Visiting http://localhost:3000/find_cube_100 will return an array of cubes less than 100 in JSON format.

OUTPUT:



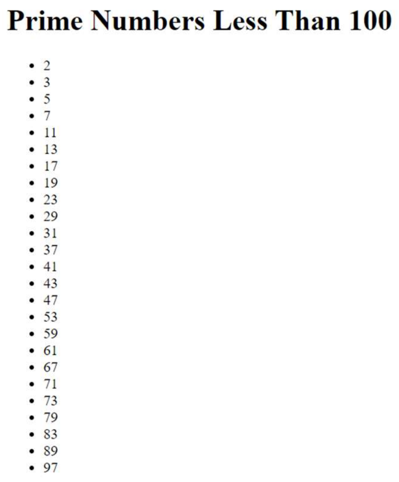
```
{
  "cubes": [1, 8, 27, 64]
}
```



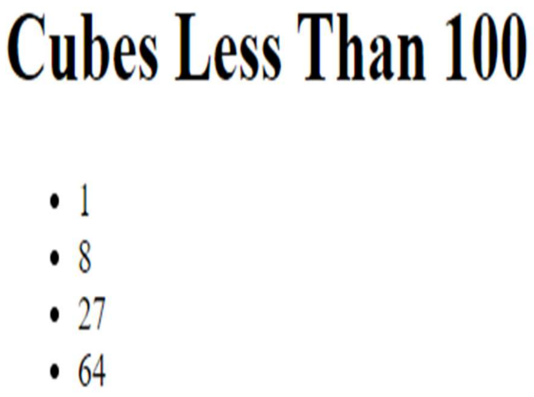
```
{
  "primeNumbers": [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
}
```

```
//HTML return type

const express = require('express');
const app = express();
const PORT = 3000;

// Route to find prime numbers less than 100
app.get('/find_prime_100', (req, res) => {
  let html = '<h1>Prime Numbers Less Than 100</h1>';
  html += '<ul>';
  for (let i = 2; i < 100; i++) {
    let isPrime = true;
    for (let j = 2; j <= Math.sqrt(i); j++) {
      if (i % j === 0) {
        isPrime = false;
        break;
      }
    }
    if (isPrime) {
      html += `<li>${i}</li>`;
    }
  }
  html += '</ul>';
  res.send(html);
});

// Route to find cubes less than 100
app.get('/find_cube_100', (req, res) => {
  let html = '<h1>Cubes Less Than 100</h1>';
  html += '<ul>';
  for (let i = 1; i < 100; i++) {
    const cube = i * i * i;
    if (cube < 100) {
      html += `<li>${cube}</li>`;
    } else {
      break; // No need to continue if cube exceeds 100
    }
  }
  html += '</ul>';
  res.send(html);
});

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

OUTPUT:



```
Prime Numbers Less Than 100

  • 2
  • 3
  • 5
  • 7
  • 11
  • 13
  • 17
  • 19
  • 23
  • 29
  • 31
  • 37
  • 41
  • 43
  • 47
  • 53
  • 59
  • 61
  • 67
  • 71
  • 73
  • 79
  • 83
  • 89
  • 97
```



```
Cubes Less Than 100

  • 1
  • 8
  • 27
  • 64
```

PROGRAM 9

9. Develop a React code to build a simple search filter functionality to display a filtered list based on the search query entered by the user.

Create a Directory:

mkdir mydir

Set Directory :

Cd mydir

Create React Application in the Directory

npx create-react-app my-app

Replace my-app with the name of your application.

Navigate to the Project Directory: After creating the React application, navigate to the project directory using the following command:

cd my-app

Replace Component Code: Replace the contents of the src/App.js file with the code provided for the SearchFilter component.

Start the Development Server: Start the development server to see your React application in action. Run the following command:

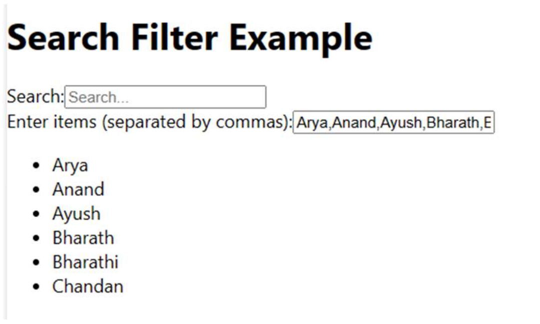
npm start

This command will start the development server, and your default web browser should automatically open the application at http://localhost:3000.

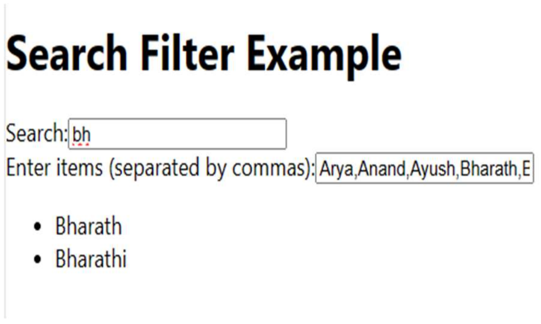
// src/App.js

```
import React, { useState } from 'react';

const SearchFilter = () => {
  // State to hold the search query and list items
  const [searchQuery, setSearchQuery] = useState('');
  const [items, setItems] = useState([]);

  // Function to handle input change for search query
  const handleInputChange = (e) => {
    setSearchQuery(e.target.value);
  };

  // Function to handle input change for list items
  const handleItemsChange = (e) => {
    setItems(e.target.value.split(','));
  };

  // Filter items based on search query
  const filteredItems = items.filter(item =>
    item.toLowerCase().includes(searchQuery.toLowerCase())
  );

  return (
    <div>
      <h1>Search Filter Example</h1>
      <label htmlFor="searchQuery">Search:</label>
      <input
        type="text"
        id="searchQuery"
        placeholder="Search..."
        value={searchQuery}
        onChange={handleInputChange}
      />
      <br />
      <label htmlFor="items">Enter items (separated by commas):</label>
      <input
        type="text"
        id="items"
        placeholder="Enter items..."
        onChange={handleItemsChange}
      />
      <ul>
        {filteredItems.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </div>
  );
};

export default SearchFilter;
```

OUTPUT:



```
Search Filter Example

Search: Search...
Enter items (separated by commas): Arya,Anand,Ayush,Bharath,E

• Arya
• Anand
• Ayush
• Bharath
• Bharathi
• Chandan
```



```
Search Filter Example

Search: bh
Enter items (separated by commas): Arya,Anand,Ayush,Bharath,E

• Bharath
• Bharathi
```

PROGRAM 10

10. Develop a React code to collect data from rest API.

Step 1. Create a React Application

```
npx create-react-app demo
```

Step 2. Change your Project Directory

```
cd demo
```

Step 3. Install the Axios Library with npm or yarn

```
npm install axios
```

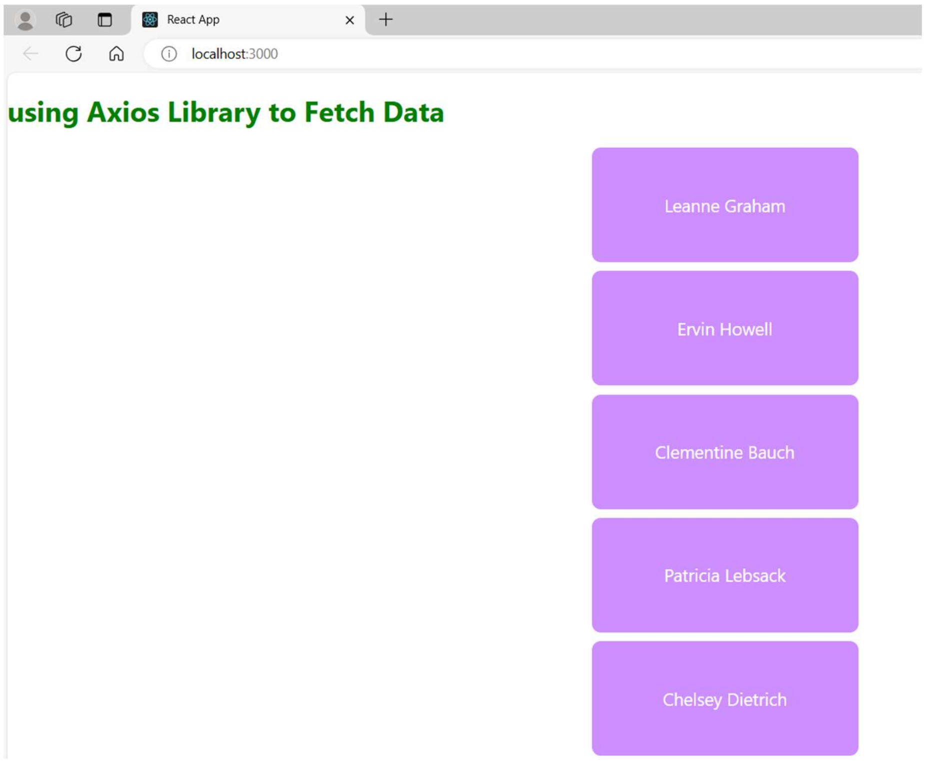
Step 4. Run the Application

```
npm start
```

App.js file:

```
import React, { useState, useEffect } from "react";
import axios from "axios";

function App() {
  const url = "https://jsonplaceholder.typicode.com/users";
  const [data, setData] = useState([]);

  const fetchInfo = async () => {
    try {
      const res = await axios.get(url);
      setData(res.data);
    } catch (error) {
      console.error("Error fetching data: ", error);
    }
  };

  useEffect(() => {
    fetchInfo();
  }, []);

  return (
    <div className="App">
      <h1 style={{ color: "green" }}>using Axios Library to Fetch Data</h1>
      <center>
        {data.map((dataObj) => (
          <div
            key={dataObj.id}
            style={{
              width: "15em",
              backgroundColor: "#CD8FFD",
              padding: "2em",
              borderRadius: "10px",
              marginBlock: "10px",
            }}
          >
            <p style={{ fontSize: "20px", color: "white" }}>{dataObj.name}</p>
          </div>
        ))}
      </center>
    </div>
  );
}

export default App;
```

OUTPUT



using Axios Library to Fetch Data

Leanne Graham

Ervin Howell

Clementine Bauch

Patricia Lebsack

Chelsey Dietrich