TM
CHAITANYA BHARATHI
INSTITUTE OF TECHNOLOGY
స్వయం తేజస్వినా భవ
1979

AIM: Installation, configuration and connection establishment of MongoDB.

DESCRIPTION:

MongoDB is a popular NoSQL database that stores data in JSON-like documents with dynamic schemas, making it highly flexible and scalable. It is widely used for its high performance, scalability, and ease of use.
Some of the key features of MongoDB include:

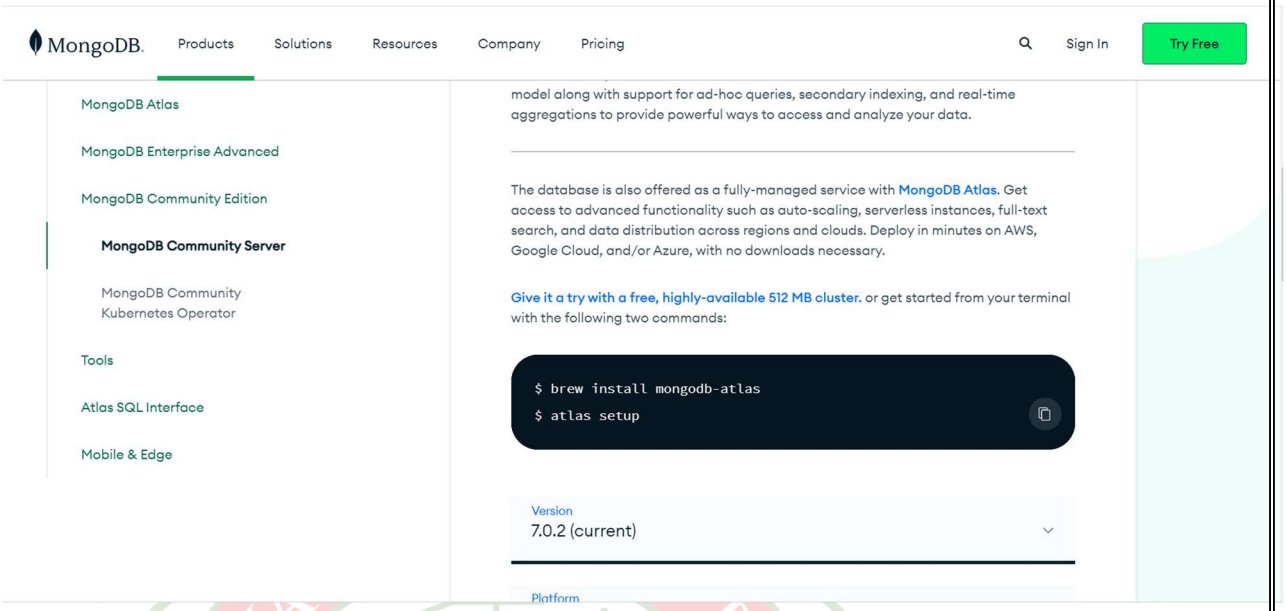
- **Document-oriented:** MongoDB stores data in JSON-like documents, which makes it easy to work with and highly flexible. Documents can have different fields and structures, allowing for dynamic schemas.
- **Scalable:** MongoDB can scale horizontally across multiple servers, allowing it to handle large amounts of data and high traffic loads. It also has built-in sharing and replication features for automatic distribution and redundancy of data.
- **High performance:** MongoDB uses a variety of optimization techniques, including indexing, caching, and native aggregation support, to provide fast read and write performance.
- **Querying and aggregation:** MongoDB supports a powerful query language and aggregation framework, making it easy to search, filter, and aggregate data in real-time.
- **Schema validation:** MongoDB allows you to define validation rules for your documents, ensuring that data is always consistent and valid.
- **Flexible data model:** MongoDB allows you to model complex data structures with ease, including hierarchical data, arrays, and nested documents.

DEMONSTRATION:

Here are the steps to install, configure and establish a connection with MongoDB

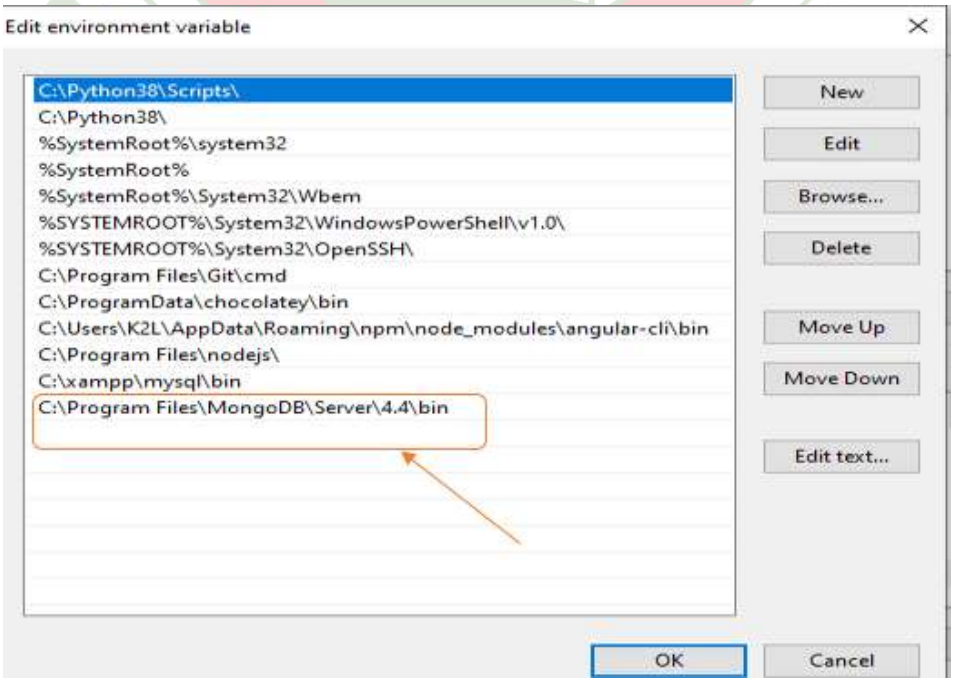
Step 1: Download and Install MongoDB

- Visit the MongoDB download page(https://www.mongodb.com/try/download/community)
- Download the Community server edition of MongoDB for your operating system.
- Follow the installation instructions for your operating system.



Step 2: Configure MongoDB

- Now we go to location where MongoDB installed in step-5 in our system and copy the binpath
- Now, to create an environment variable open system properties << Environment Variable
  << System variable << path << Edit Environment variable and paste the copied link to yourenvironment system and click Ok:



Step 3: Connection Establishment

- Open a terminal or command prompt and start the MongoDB server with the following command: mongosh

INSTITUTE OF TECHNOLOGY
స్వయం తేజస్విన్ భవ
1979

AIM: CRUD operations on MongoDB.

DESCRIPTION:

CRUD operations describe the conventions of a user-interface that let users view, search, and modify partsof the database. MongoDB documents are modified by connecting to a server, querying the proper documents, and thenchanging the setting properties before sending the data back to the database to be updated. CRUD is data-oriented, and it's standardized according to HTTP action verbs.

When it comes to the individual CRUD operations:
The Create operation is used to insert new documents in the MongoDB database.
The Read operation is used to query a document in the database.
The Update operation is used to modify existing documents in the database.
The Delete operation is used to remove documents in the database.

DEMONSTRATION:

Create Operations:

For MongoDB CRUD, if the specified collection doesn't exist, the create operation will create the collection when it's executed. Create operations in MongoDB target a single collection, not multiplecollections. Insert operations in MongoDB are atomic on a single document level. MongoDB provides two different create operations that you can use to insert documents into a collection:

1) db.collection.insertOne()
2) db.collection.insertMany()

insertOne()

It allows you to insert one document into the collection.For this example, we're going to work with a collection called my. We can insert a single entryinto our collection by calling the insertOne() method on my.
We then provide the information we want to insert in the form of key-value pairs, establishing the schema.

```
db.my.insertOne( {name: "Rahul" , age: "18"})
```

OUTPUT:

```
{
  acknowledged: true,
  insertedId: ObjectId("65196951138ad81f0790003b")
}
```

If the create operation is successful, a new document is created. The function will return an object where “acknowledged” is “true” and “insertID” is the newly created “ObjectId.”

insertMany()

It’s possible to insert multiple items at one time by calling the insertMany() method on the desired collection. In this case, we pass multiple items into our chosen collection (my) and separate them by commas. Within the parentheses, we use brackets to indicate that we are passing in a list of multiple entries. This is commonly referred to as a nested method.

```
db.my.insertMany([ {name: "Ravi", age:"20"},
{name:"Ram",age:"24"},
{name:"Sunil",age:"15"}])
```

OUTPUT:

```
{
 acknowledged: true,
 insertedIds: {
  '0': ObjectId("65196a8f138ad81f0790003c"),
  '1': ObjectId("65196a8f138ad81f0790003d"),
  '2': ObjectId("65196a8f138ad81f0790003e")
 }
}
```

Read Operations

The read operations allow you to supply special query filters and criteria that let you specify which documents you want. The MongoDB documentation contains more information on the available query filters. Query modifiers may also be used to change how many results are returned.

MongoDB has two methods of reading documents from a collection:

1) db.collection.find()
2) db.collection.findOne()

find():

In order to get all the documents from a collection, we can simply use the find() method on our chosen collection.

Executing just the find() method with no arguments will return all records currently in the collection

db.my.find()

Here we can see that every record has an assigned “ObjectId” mapped to the “_id” key.
If you want to get more specific with a read operation and find a desired subsection of the records, you can use the previously mentioned filtering criteria to choose what results should be returned. One of the most common ways of filtering the results is to search by value.

OUTPUT:

```
> db.my.find()
< {
    _id: ObjectId("6519692e138ad81f0790003a")
  }
  {
    _id: ObjectId("65196951138ad81f0790003b"),
    name: 'Rahul',
    age: '18'
  }
  {
    _id: ObjectId("65196a8f138ad81f0790003c"),
    name: 'Ravi',
    age: '20'
  }
  {
    _id: ObjectId("65196a8f138ad81f0790003d"),
    name: 'Ram',
    age: '24'
  }
  {
    _id: ObjectId("65196a8f138ad81f0790003e"),
    name: 'Sunil',
    age: '15'
  }
```

findOne()

In order to get one document that satisfies the search criteria, we can simply use the findOne() method on our chosen collection.If multiple documents satisfy the query, this method returns the first document according to the natural order which reflects the order of documents on the disk.If no documents satisfy the search criteria, the function returns null. The function takes the following form of syntax.

db.{collection}.findOne({query}, {projection})

```
> db.my.findOne({name:"Sunil"})
< {
    _id: ObjectId("65196a8f138ad81f0790003e"),
    name: 'Sunil',
    age: '15'
  }
```

Update Operations:

Like create operations, update operations operate on a single collection, and they are atomic at a single document level.
An update operation takes filters and criteria to select the documents you want to update.
You should be careful when updating documents, as updates are permanent and can't be rolled back. This applies to delete operations as well.
For MongoDB CRUD, there are two different methods of updating documents:

1) db.collection.updateOne()
2) db.collection.updateMany()

updateOne()

We can update a currently existing record and change a single document with an update operation.
To do this, we use the updateOne() method on a chosen collection, which here is "my."
To update a document, we provide the method with two arguments: an update filter and an update action.
The update filter defines which items we want to update, and the update action defines how to update those items. We first pass in the update filter.
Then, we use the "$set" key and provide the fields we want to update as a value. This method will update the first record that matches the provided filter.

```
> db.my.updateOne({name: "Ravi"},{$set:{age:"25"}})
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 1,
    modifiedCount: 1,
    upsertedCount: 0
  }
```

updateMany():

updateMany() allows us to update multiple items by passing in a list of items, just as we did when inserting multiple items. This update operation uses the same syntax for updating a single document.

```
> db.my.updateMany({name:"Ravi"},{$set:{age:"24"}})
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 2,
    modifiedCount: 2,
    upsertedCount: 0
  }
```

Delete Operations:

Delete operations operate on a single collection, like update and create operations. Delete operations are also atomic for a single document. You can provide delete operations with filters and criteria in order to specify which documents you would like to delete from a collection. The filter options rely on the same syntax that read operations utilize.

MongoDB has two different methods of deleting records from a collection:

1) db.collection.deleteOne()
2) db.collection.deleteMany()

deleteOne()

deleteOne() is used to remove a document from a specified collection on the MongoDB server. A filter criteria is used to specify the item to delete. It deletes the first record that matches the provided filter.

```
> db.my.deleteOne({name:"Sunil"})
< {
    acknowledged: true,
    deletedCount: 1
  }
```

deleteMany()

deleteMany() is a method used to delete multiple documents from a desired collection with a single delete operation. A list is passed into the method and the individual items are defined with filter criteria as in deleteOne().

```
> db.my.deleteMany({name:"Ravi"})
< {
    acknowledged: true,
    deletedCount: 2
  }
```

AIM: Create Express application

DESCRIPTION:

Express.js is a popular, open-source, minimalist web framework for Node.js. It provides a robust set of features for developing web and mobile applications. Express is designed to be lightweight, flexible, and easy to use, making it a popular choice among developers who want to build scalable, high-performance web applications.

Some of the key features of Express include:

- **Routing:** Express provides a simple and intuitive way to define routes for handling HTTP requests. It allows developers to create custom middleware functions to handle specific requests, and to chain multiple middleware functions together to handle more complex use cases.

- **Middleware:** Express allows developers to use middleware functions to perform various tasks during the request-response cycle, such as logging, authentication, and data validation. It also supports third-party middleware modules that can be easily integrated into an Express application.

- **Templating:** Express supports a variety of templating engines, including EJS, Pug, and Handlebars. These engines make it easy to generate dynamic HTML pages by merging data with pre-defined templates.

- **Error handling:** Express provides robust error handling capabilities, allowing developers to catch and handle errors in a consistent and reliable manner. It also includes built-in error handling middleware that can be used to catch and handle common types of errors.

- **Static file serving**: Express makes it easy to serve static files, such as images, CSS, and JavaScript, from a directory on the server. This allows developers to build front-end applications that can be easily deployed to a web server.

Express has a large and active community of developers who contribute to its ongoing development and support. It is widely used by companies of all sizes, from small startups to large enterprises, to build high- performance web and mobile applications.

DEMONSTRATION:

Step 1: Install Node.js and npm:

- You'll need Node.js and npm (Node Package Manager) installed on your machine to use Express.
- You can download them from the official website: https://nodejs.org/en/download/

Step 2: Create a new directory for your Express application:

- Open up your terminal and create new directory for your application, e.g. mkdir my-express-app.
- Navigate to the directory using cd my-express-app.

```
C:\Users\CBIT\Documents>node -v
v18.14.2

C:\Users\CBIT\Documents>npm -v
9.5.0

C:\Users\CBIT\Documents>mkdir my_express_app

C:\Users\CBIT\Documents>cd my_express_app
```

Step 3: Initialize your project:

- Run npm init in your terminal and follow the prompts to initialize your project. This will create a package.json file in your project directory.

```
C:\Users\CBIT\Documents\my_express_app>npm init -y
Wrote to C:\Users\CBIT\Documents\my_express_app\package.json:

{
  "name": "my_express_app",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "keywords": [],
  "description": ""
}


C:\Users\CBIT\Documents\my_express_app>
```

Step 4: Install Express:

- Run npm install express in your terminal to install Express and add it to your project's dependencies.

```
C:\Users\CBIT\Documents\my_express_app>npm install express

added 57 packages, and audited 58 packages in 12s

7 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

C:\Users\CBIT\Documents\my_express_app>
```

Step 5: Create a new file for your server:

- Create a new file in your project directory, e.g., server.js. This will be the entry point for your application.

Step 6: Set up your Express application:

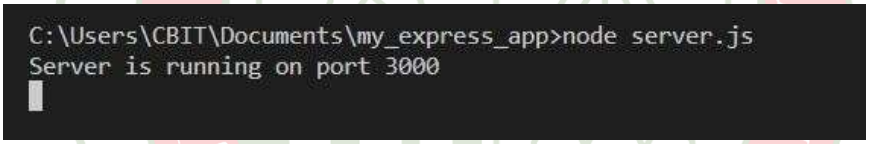
- In server.js, require Express and create an instance of it:
  const express = require('express'); const app = express();

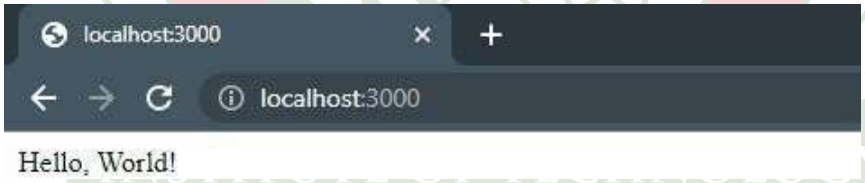
Step 7: Define routes:

- You can define routes using the app.get(), app.post(), app.put(), app.delete(), and other methods.
- For example, here's how you can define a basic route that sends a message to the client:
  app.get('/', (req, res) => { res.send('Hello, World!'); });

Step 8: Start your server:

- Finally, start your server by listening on a port.
  For example, you can listen on port 3000 like this:
  app.listen(3000, () => { console.log('Server is running on port 3000'); });





AIM: To Access Data Using Nodejs

DESCRIPTION: Mongoose: Mongoose is a popular object modeling library for MongoDB. It provides a schema-based solution to model your application data and includes features like validation, query building, and middleware.

Step-1 Create a project folder.
Step-2 install mongoose.
Step-3 Create .js file Step-4 run node file_name.js

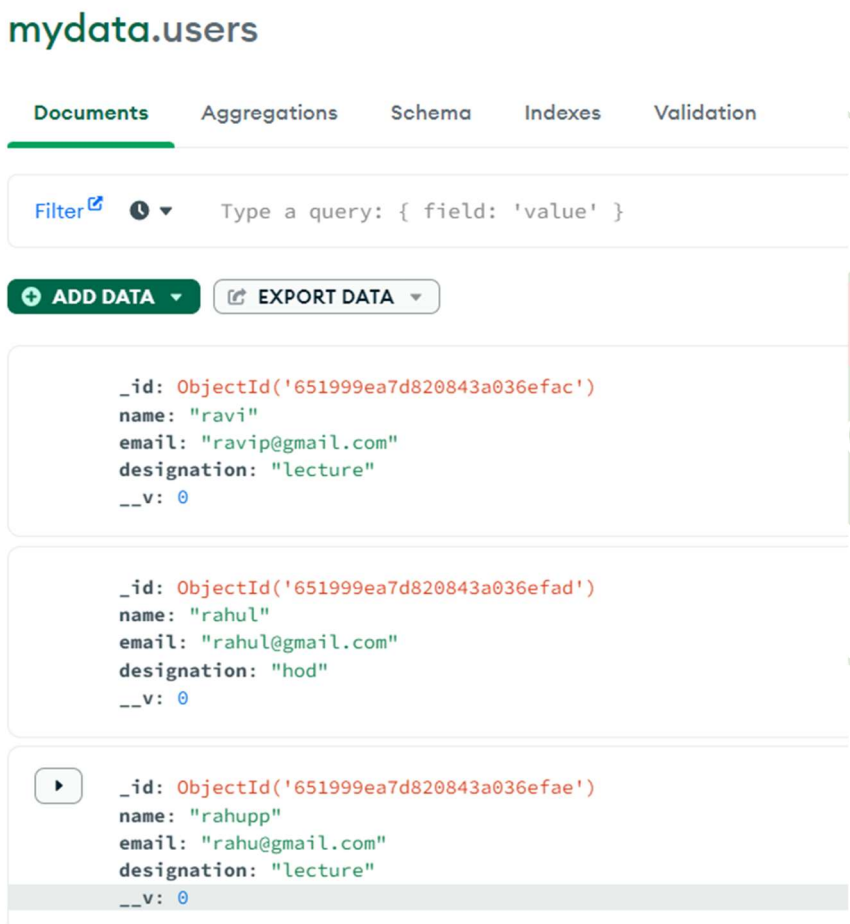
DEMONSTRATION:

```
const mongoose = require('mongoose');
async function main() {

   try {

      await mongoose.connect('mongodb://localhost:27017/mydata', {

         useNewUrlParser: true,

         useUnifiedTopology: true,

      });

console.log("Connected to MongoDB");

      const userSchema = new mongoose.Schema({

         name: String,

         email: String,

         designation: String,

      });

      const User = mongoose.model('User', userSchema);

      // Create and save new users

      const newUsers = [

         {

            name: 'ravi',

            email: 'ravip@gmail.com',
```

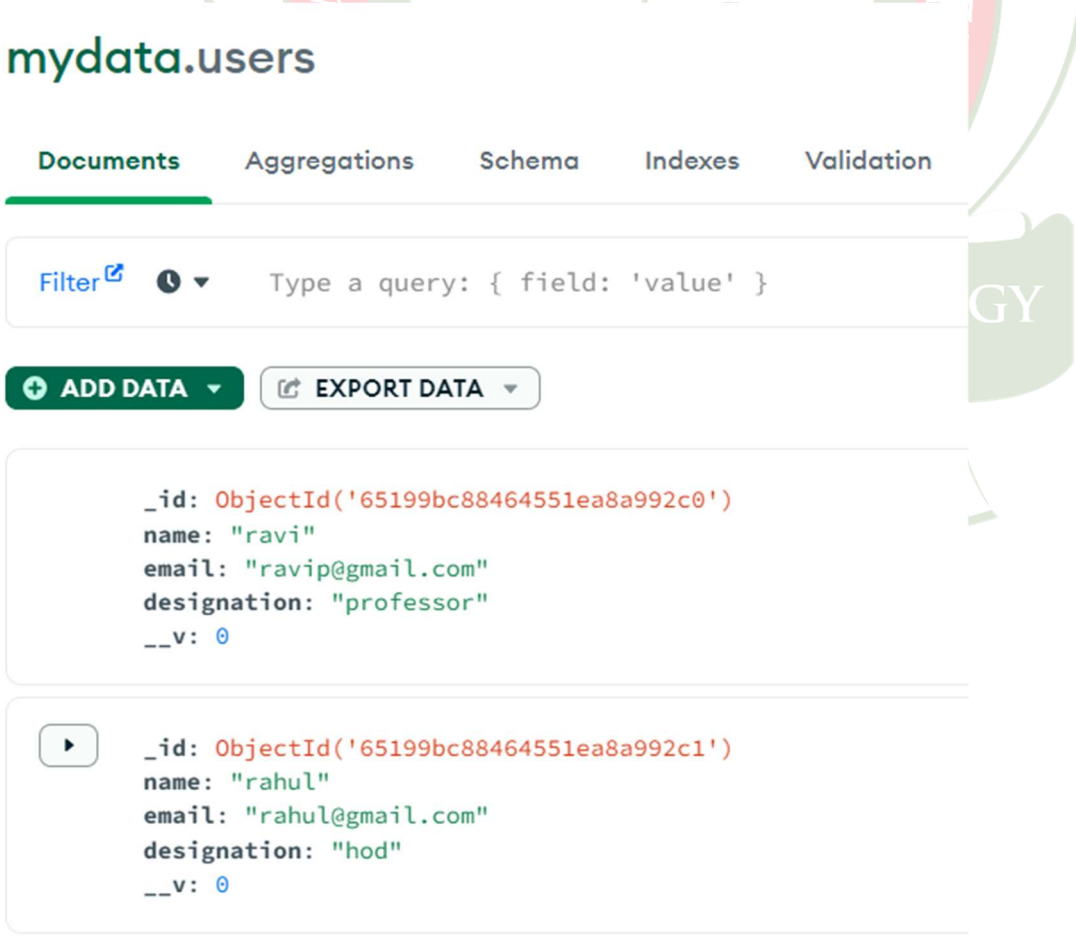
```
        designation: 'lecture'
    },
    {
        name: 'rahul',
        email: 'rahul@gmail.com',
        designation: 'hod'
    },
    {
        name: 'rahupp',
        email: 'rahu@gmail.com',
        designation: 'lecture'
    }
];
await User.insertMany(newUsers);
console.log('Users created');
// Update a user
await User.findOneAndUpdate({ name: 'ravi' }, { designation: 'professor' });
console.log('User updated');
// Delete a user
await User.deleteOne({ name: 'rahupp' });
console.log('User deleted');

// Find all users after deletion
const allUsers = await User.find();
console.log('All users after deletion:', allUsers);
```

```
    } catch (error) {
        console.error('Error:', error);
    } finally {
        mongoose.disconnect(); // Close the MongoDB connection
    }
}

main();
```

OUTPUT:



```
PS C:\Users\ranga\Downloads\EAD> node server.js
Connected to MongoDB
Users created
User updated
User deleted
All users after deletion: [
  {
    _id: new ObjectId("65199bc88464551ea8a992c0"),
    name: 'ravi',
    email: 'ravip@gmail.com',
    designation: 'professor',
    __v: 0
  },
  {
    _id: new ObjectId("65199bc88464551ea8a992c1"),
    name: 'rahul',
    email: 'rahul@gmail.com',
    designation: 'hod',
    __v: 0
  }
]
PS C:\Users\ranga\Downloads\EAD>
```



AIM: To Create a RESTful API that performs CRUD operations on a database, such as MongoDB database.

DESCRIPTION:

Creating a RESTful API that performs CRUD (Create, Read, Update, Delete) operations on a database, like MongoDB, involves designing endpoints and implementing the necessary HTTP methods to interact with the data.

Key Components and Features of the RESTful API:

1) **Project Setup:** The project will be initiated with Node.js as the backend runtime environment. Required dependencies such as Express.js, Mongoose, and any additional packages will be installed.
2) **Database Configuration:** The API will establish a connection with a MongoDB database using Mongoose, a popular ODM (Object-Document Mapping) library. A defined data model/schema will represent the structure of the records to be stored in the database.
3) **Create Operation (POST):** The API will feature an endpoint to handle HTTP POST requests for creating new records. Incoming data will be validated and sanitized to ensure data integrity. New records will be saved to the MongoDB database.
4) **Read Operation (GET):** The API will implement endpoints to handle HTTP GET requests for retrieving records. Two primary routes will be provided: one for fetching all records and another for retrieving a specific record by a unique identifier, typically an ID.
5) **Update Operation (PUT/PATCH)**: HTTP PUT and PATCH requests for updating existing records will be supported. The API will locate the target record by its unique identifier and update it with the provided data.
6) **Delete Operation (DELETE):** An endpoint will be created to manage HTTP DELETE requests for record removal. The API will find and delete records based on their unique identifier.
7) **Error Handling:** Robust error-handling mechanisms will be implemented to address various scenarios, including invalid requests, database errors, or instances where records are not found.
8) **Router Implementation:** Express.js will be utilized to create modular routes or routers for organizing the API endpoints. Each CRUD operation (Create, Read, Update, Delete) will be separated into individual router modules, making the codebase more maintainable and scalable. Routes will be appropriately organized and grouped within these routers, ensuring a clear and structured API design.

DEMONSTRATION:

Server.js:

```
const express= require("express");
const mongoose= require("mongoose")
const url="mongodb://localhost:27017/rkdb"

const app= express()
```

```
app.use(express.json())
app.use(express.urlencoded({
    extended: true
}))
mongoose.connect(url,{
        useNewUrlParser:true,
        useUnifiedTopology: true,
    });
const db=mongoose.connection;
db.on('open',()=>{
    console.log("connected to mongodb");
})
const rkrouter= require('./routers/rk')
app.use('/rk',rkrouter)
app.listen(3000,()=>{
    console.log("server started");
})
```

routers/rk.js:

```
const express= require("express")
const router= express.Router()
const RK= require("../models/rk")
router.get('/', async (req,res)=>{
    try{
        const rk= await RK.find()
        res.json(rk)
    }catch(err)
```

```
    {
        res.send('Error'+err)
    }
})
router.get('/:id',async(req,res) => {
    try{
        const rk = await RK.findById(req.params.id)
        res.json(rk)
    }catch(err)
    {
        res.send("error"+err)
    }
})
router.post('/',async(req,res)=>{
    const rk= new RK(
        {
            name:req.body.name,
            age:req.body.age
        }
    );
    try{
        const a1= await rk.save()
        res.json(a1)
    }catch(err)
    { res.send("error"+err) }
```

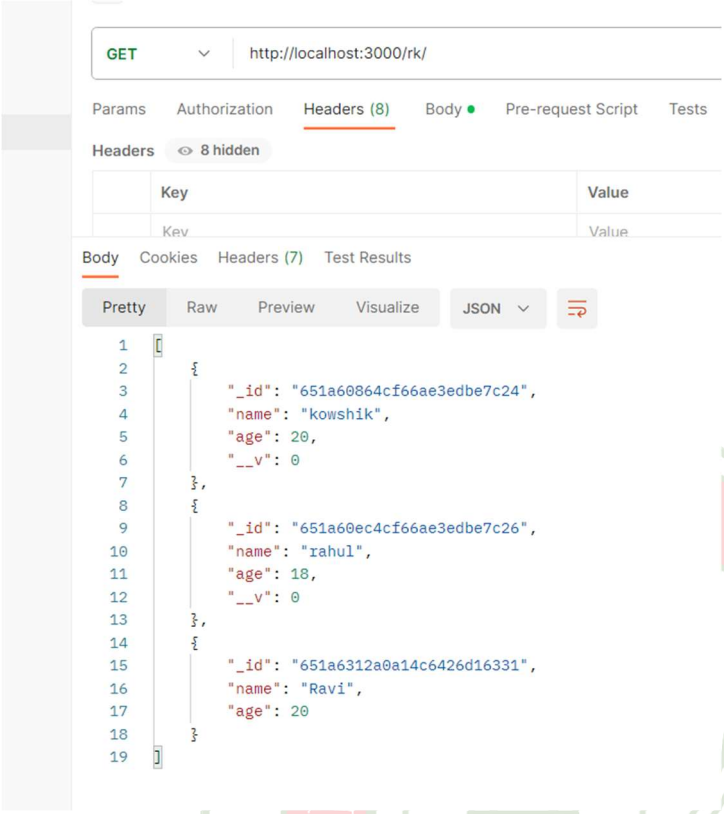
```
})
router.patch('/:id', async(req,res) =>{
    try{
        const rk= await RK.findById(req.params.id)
        rk.age=req.body.age
        const a1= await rk.save()
        res.json(a1)
    }catch(err)
    {
        res.send("error"+err)
    }
})
router.delete('/:id', async(req,res) =>{
    try {
        const rk = await RK.findByIdAndRemove(req.params.id);
        if (!rk) {
          res.json({ message: 'Record not found' });
        } else {
          res.json({ message: 'Record deleted successfully' });
        }
      } catch (err) {
        res.send("error"+err)
      }
})
module.exports=router
```

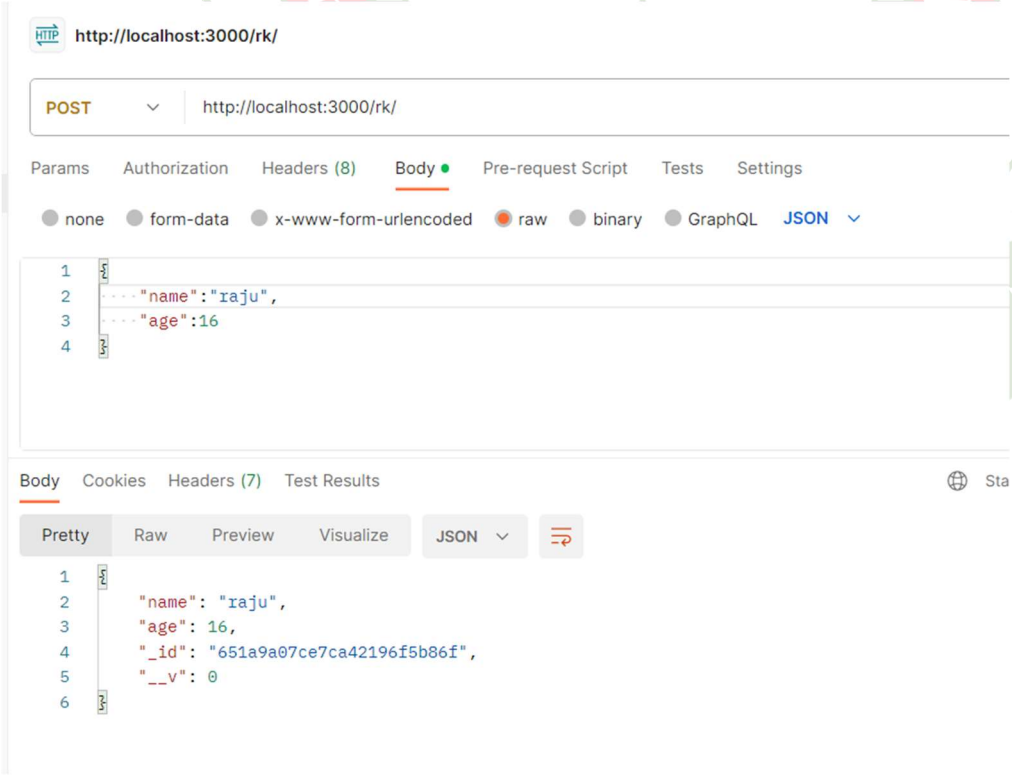
models/rk.js

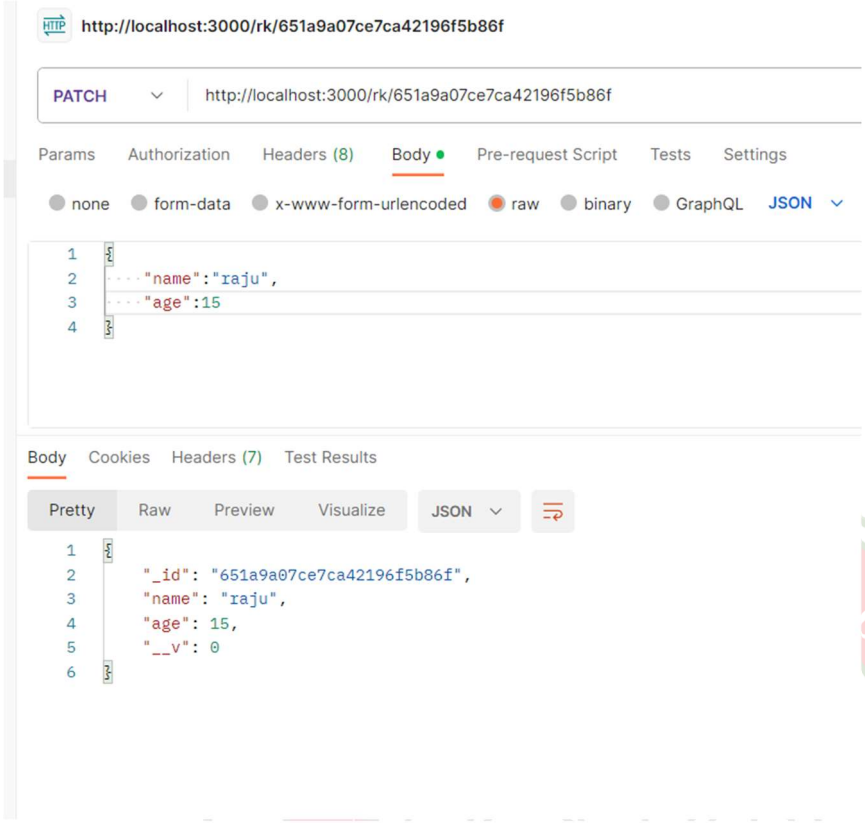
```
const mongoose = require("mongoose");
const schema= new mongoose.Schema(
   {
      name:{
         type:String,
         required: true
      },
      age: {
         type: Number,
         required:true
      }
   }
);

module.exports= mongoose.model('RK',schema)
```

OUTPUT:

CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY స్వయం తేజస్వినా భవ 1979


GET
http://localhost:3000/rk/
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Headers
8 hidden
Key
Value
Body
Cookies
Headers (7)
Test Results
Pretty
Raw
Preview
Visualize
JSON
[
{
"_id": "651a60864cf66ae3edbe7c24",
"name": "kowshik",
"age": 20,
"__v": 0
},
{
"_id": "651a60ec4cf66ae3edbe7c26",
"name": "rahul",
"age": 18,
"__v": 0
},
{
"_id": "651a6312a0a14c6426d16331",
"name": "Ravi",
"age": 20
}
]


http://localhost:3000/rk/
POST
http://localhost:3000/rk/
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"name":"raju",
"age":16
}
Body
Cookies
Headers (7)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"name": "raju",
"age": 16,
"_id": "651a9a07ce7ca42196f5b86f",
"__v": 0
}


http://localhost:3000/rk/651a9a07ce7ca42196f5b86f
PATCH
http://localhost:3000/rk/651a9a07ce7ca42196f5b86f
Params Authorization Headers (8) Body Pre-request Script Tests Settings
none form-data x-www-form-urlencoded raw binary GraphQL JSON
{
"name":"raju",
"age":15
}
Body Cookies Headers (7) Test Results
Pretty Raw Preview Visualize JSON
{
"_id": "651a9a07ce7ca42196f5b86f",
"name": "raju",
"age": 15,
"__v": 0
}


http://localhost:3000/rk/651a9a07ce7ca42196f5b86f
DELETE
http://localhost:3000/rk/651a9a07ce7ca42196f5b86f
Params Authorization Headers (6) Body Pre-request Script Tests Settings
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Cookies Headers (7) Test Results
Pretty Raw Preview Visualize JSON
{
"message": "Record deleted successfully"
}

AIM: To Create a basic calculator that can perform arithmetic operations (addition, subtraction, multiplication and division) through HTTP requests.

DESCRIPTION:

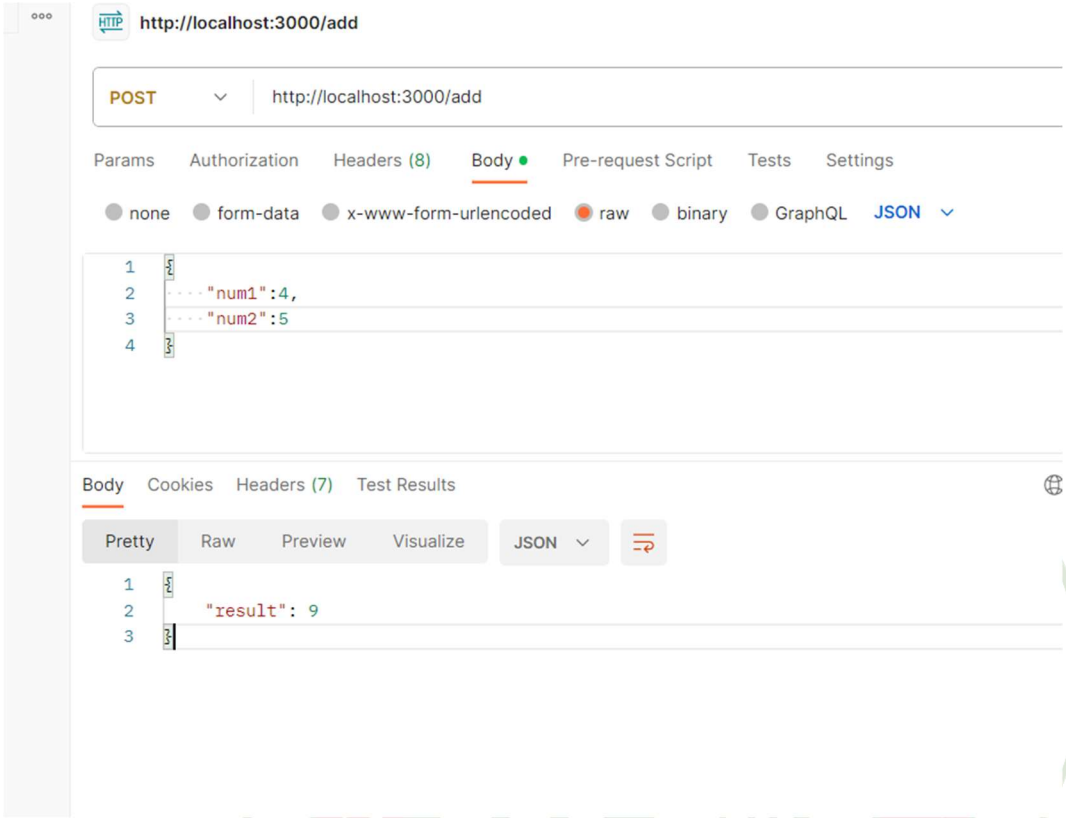
1) Set Up Your Environment: Ensure you have Node.js installed on your system .Use a package manager like npm to install any necessary dependencies, such as Express.js for building the web application.
2) Create an Express.js Application: Initialize an Express.js application to handle HTTP requests and responses.
3) Define app.post Handlers: Use app.post to define handlers for each of the arithmetic operations (addition, subtraction, multiplication, and division). These handlers will be responsible for processing the incoming HTTP requests.
4) Parse Request Data: In each app.post handler, parse the incoming HTTP request to extract the input numbers and the operation to perform. You can use the express.json() middleware to parse JSON data from requests.
5) Perform Arithmetic Operations: Implement the logic for each arithmetic operation based on the parsed data (e.g., perform addition, subtraction, multiplication, or division).
6) Handle Errors: Ensure that your code handles error cases gracefully, such as division by zero or invalid input.
7) Send Response: Send the result of the arithmetic operation as a JSON response to the client using res.json().
8) Run Your Node.js Application: Start your Node.js application to listen for incoming HTTP requests. You can specify a port number (e.g., 3000) on which your application will listen.

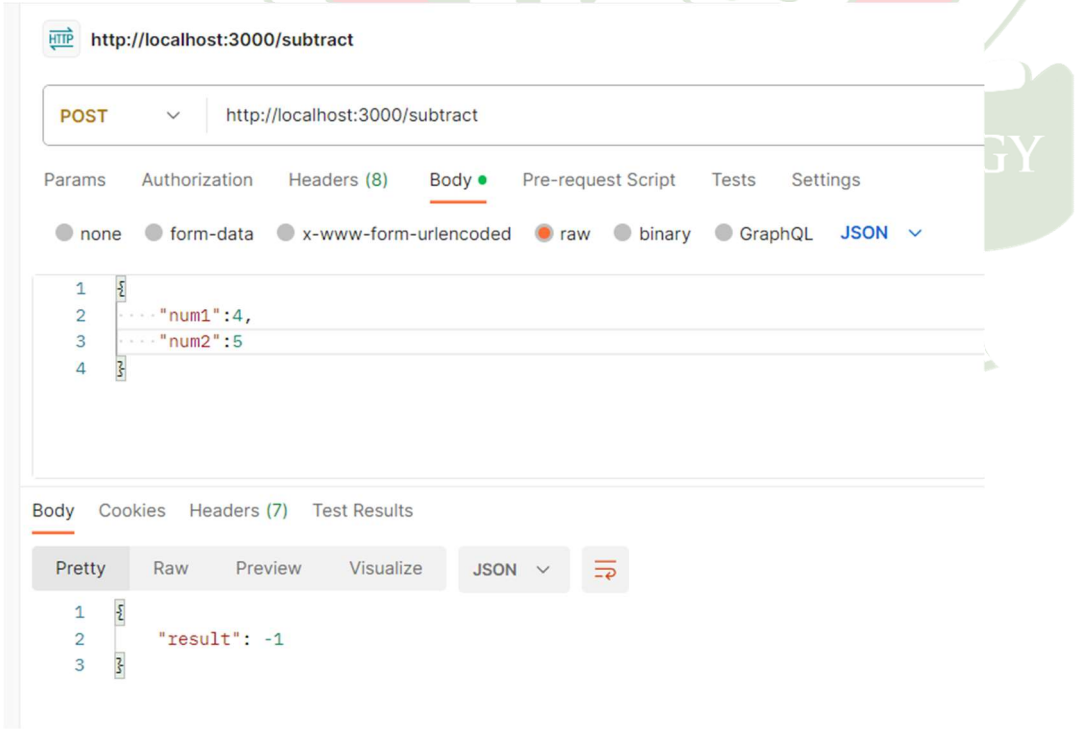
CODE:

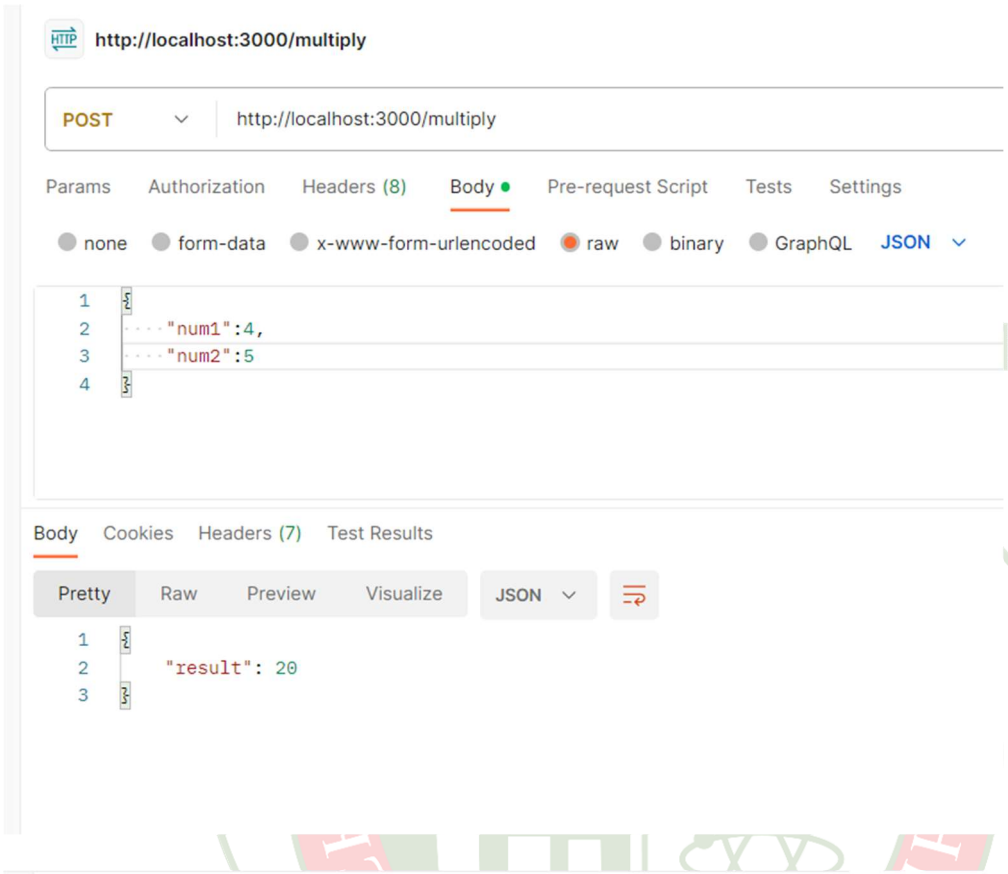
```
const express = require('express');

const bodyParser = require('body-parser');

const app = express();


app.use(bodyParser.json());

app.post('/add', (req, res) => {

  const { num1, num2 } = req.body;

  const result = num1 + num2;

  res.json({ result });

});


app.post('/subtract', (req, res) => {
```

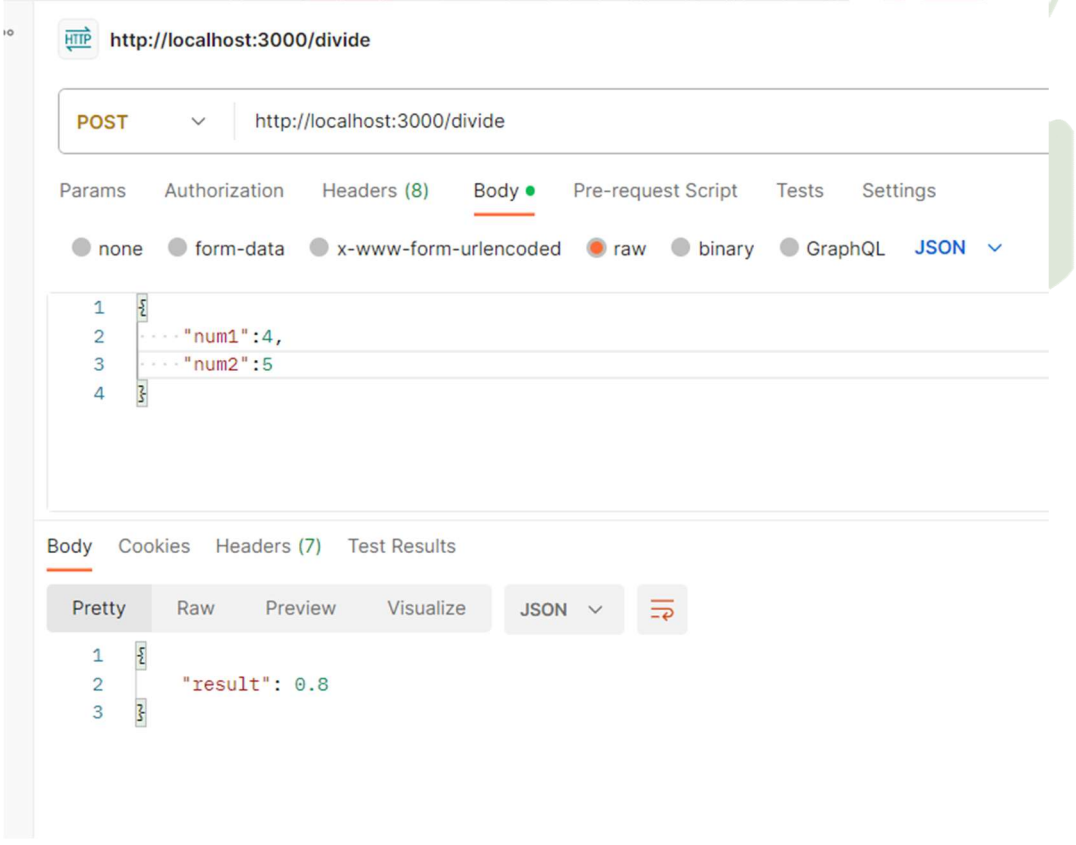
```
  const { num1, num2 } = req.body;
  const result = num1 - num2;
  res.json({ result });
});

app.post('/multiply', (req, res) => {
  const { num1, num2 } = req.body;
  const result = num1 * num2;
  res.json({ result });
});

app.post('/divide', (req, res) => {
  const { num1, num2 } = req.body;
  if (num2 === 0) {
    res.status(400).json({ error: 'Division by zero is not allowed' });
  } else {
    const result = num1 / num2;
    res.json({ result });
  }
});

const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`Calculator API is running on port ${port}`);
});
```

OUTPUT:


http://localhost:3000/add
POST
http://localhost:3000/add
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"num1":4,
"num2":5
}
Body
Cookies
Headers (7)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"result": 9
}


http://localhost:3000/subtract
POST
http://localhost:3000/subtract
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"num1":4,
"num2":5
}
Body
Cookies
Headers (7)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"result": -1
}


http://localhost:3000/multiply
POST
http://localhost:3000/multiply
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"num1":4,
"num2":5
}
Body
Cookies
Headers (7)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"result": 20
}


http://localhost:3000/divide
POST
http://localhost:3000/divide
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"num1":4,
"num2":5
}
Body
Cookies
Headers (7)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"result": 0.8
}

AIM: To Create a to-do list application that stores data in a JSON file, with features such as adding and deleting tasks

DESCRIPTION:

1) Environment Setup: The code starts by importing the required modules, such as fs (File System) and readline. It also creates a readline interface for reading user input.
2) JSON File Initialization: The code attempts to read the tasks from a tasks.json file and initializes the tasks array with the data. If the file doesn't exist or is empty, it defaults to an empty array.Functions for Task Manipulation: Several functions are defined to manipulate the task list:

    - saveTasks(): Writes the tasks array back to the tasks.json file.
    - listTasks(): Lists all the tasks in the console.
    - addTask(taskDescription): Adds a new task to the tasks array and saves it to the JSON file.
    - deleteTask(taskIndex): Deletes a task from the tasks array by its index and updates the JSON file.

3) User Interaction: The askForAction() function displays a menu for users to choose from. Users can:

    - Add a new task by entering a description.
    - List existing tasks.
    - Delete a task by specifying its index.
    - Exit the application.

4) Menu Loop: The application runs in a loop, repeatedly prompting the user for their choice and performing the chosen action.
5) Exit Handling: When the user chooses to exit, the rl.on('close') event handler is triggered. It displays an exit message and terminates the application.
6) Application Initialization: The application starts with a welcome message and immediately calls askForAction() to initiate user interaction.

CODE:

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

const rl = readline.createInterface({

  input: process.stdin,

  output: process.stdout

});

const tasksFile = 'tasks.json';

let tasks = [];
```

```
try {
  const data = fs.readFileSync(tasksFile, 'utf8');
  tasks = JSON.parse(data);
} catch (err) {
    console.log(err);
}
function saveTasks() {
  fs.writeFileSync(tasksFile, JSON.stringify(tasks, null, 2), 'utf8');
}
function listTasks() {
  console.log('To-Do List:');
  tasks.forEach((task, index) => {
    console.log(`${index + 1}. ${task}`);
  });
}
function addTask(taskDescription) {
  tasks.push(taskDescription);
  saveTasks();
  console.log(`Task "${taskDescription}" added.`);
}
function deleteTask(taskIndex) {
  if (taskIndex >= 0 && taskIndex < tasks.length) {
    const deletedTask = tasks.splice(taskIndex, 1);
    saveTasks();
    console.log(`Task "${deletedTask[0]}" deleted.`);
```

```
  } else {
    console.log('Invalid task index.');
  }
}

function askForAction() {
  console.log('\nChoose an action:');
  console.log('1. Add Task');
  console.log('2. Delete Task');
  console.log('3. List Tasks');
  console.log('4. Exit');

  rl.question('Enter the number of your choice: ', (choice) => {
    switch (choice) {
      case '1':
        rl.question('Enter task description: ', (taskDescription) => {
          addTask(taskDescription);
          askForAction();
        });
        break;
      case '2':
        listTasks();
        rl.question('Enter the task number to delete: ', (taskNumber) => {
          const taskIndex = parseInt(taskNumber) - 1;
          deleteTask(taskIndex);
```

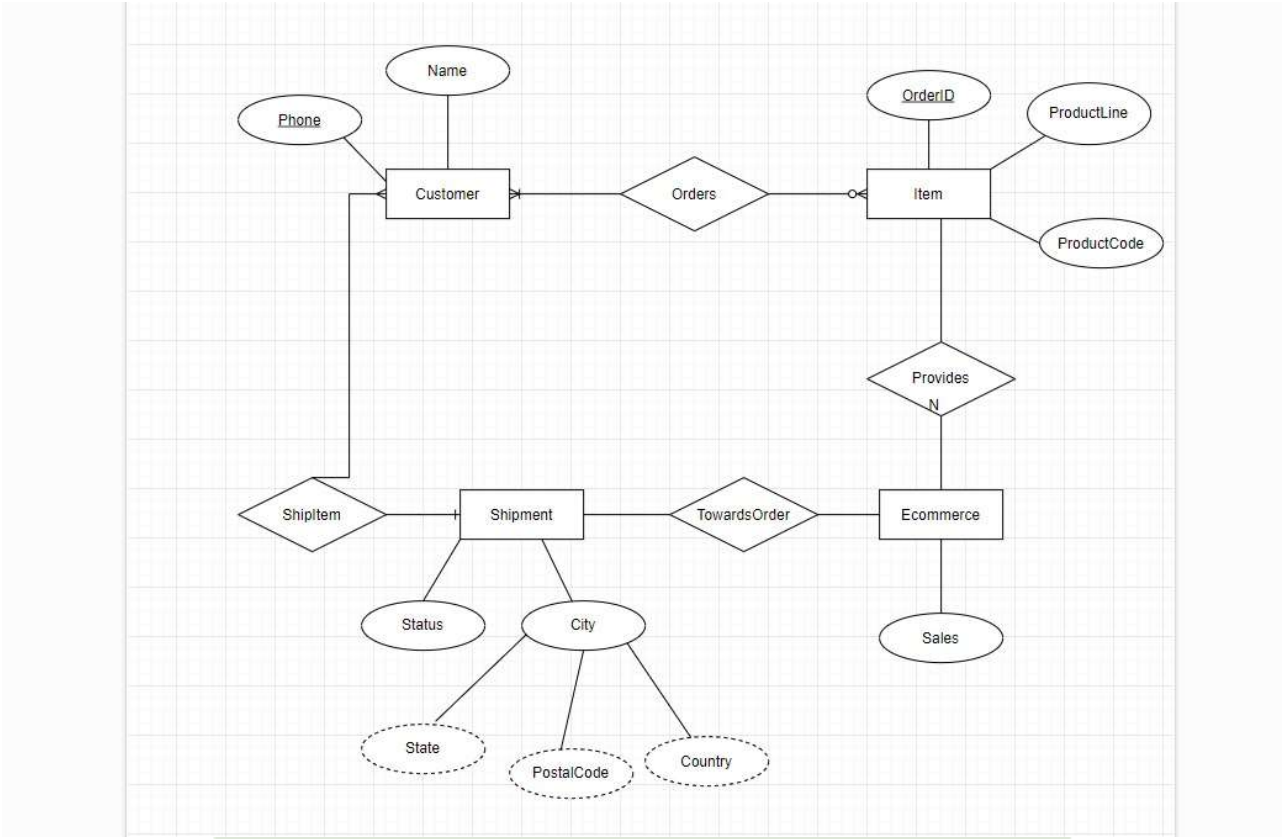
```
          askForAction();
        });
        break;
      case '3':
        listTasks();
        askForAction();
        break;
      case '4':
        rl.close();
        break;
      default:
        console.log('Invalid choice. Please enter a valid number.');
        askForAction();
    }
  });
}
console.log('Welcome to the To-Do List App!');
askForAction();
rl.on('close', () => {
  console.log('Exiting the To-Do List App.');
  process.exit(0);
});
```

CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY TM
స్వయం తేజస్వినా భవ
1979

OUTPUT:

```
Welcome to the To-Do List App!

Choose an action:
1. Add Task
2. Delete Task
3. List Tasks
4. Exit
Enter the number of your choice: 1
Enter task description: read books
Task "read books" added.

Choose an action:
1. Add Task
2. Delete Task
3. List Tasks
4. Exit
Enter the number of your choice: 3
To-Do List:
1. read books

Choose an action:
1. Add Task
2. Delete Task
3. List Tasks
4. Exit
Enter the number of your choice: 2
To-Do List:
1. read books
Enter the task number to delete: 1
Task "read books" deleted.

Choose an action:
1. Add Task
2. Delete Task
3. List Tasks
4. Exit
Enter the number of your choice: 4
Exiting the To-Do List App.
PS C:\Users\ranga\Downloads\Restful\to-do>
```

ER DIAGRAM FOR ECOMMERCE SALES DATASET



INSTITUTE OF TECHNOLOGY
స్వయం తేజస్వినే భవ
1979

AIM: Authentication and Authorization using ExpressJS(JWT)

DESCRIPTION:

Aspect	Authentication	Authorization
Definition	The process of verifying the identity of a user, system, or entity to ensure they are who they claim to be.	The process of granting or denying access to specific resources or actions based on the authenticated user's privileges or permissions.
Main Concern	Establishing identity and trust.	Controlling access and permissions.
Key Question	"Who are you?"	"What are you allowed to do?"
What It Checks	Authenticates users based on their provided credentials, such as username and password.	Determines if a user has the necessary permissions to perform a specific action or access a particular resource.
Outcomes	Successful authentication results in the user being granted access.	Authorization can result in granting or denying access based on the user's permissions and the requested resource or action.
Examples	- Verifying a user's login credentials (e.g., username and password).	- Allowing or denying access to a specific file or database. - Permitting or restricting the ability to perform actions like creating, updating, or deleting records.

Aspect	Authentication	Authorization
Mechanisms	Passwords, biometrics, two-factor authentication (2FA), client certificates, etc.	Role-based access control (RBAC), access control lists (ACLs), attribute-based access control (ABAC), etc

Aspect	Session	Cookie	Token
Purpose	Used for server-side session management.	Used for client-side data storage.	Used for authentication and authorization.
Storage Location	Server-side.	Client-side (usually in a browser).	Client-side (cookie or local storage) and optionally server-side.
Data Storage	Stores user-specific data on the server.	Stores user-specific data on the client.	Contains user information or claims.
Duration	Lives as long as the user's session on the server.	Configurable, can persist beyond a session.	Configurable, can have defined expiration.
Security	Vulnerable to session hijacking if not secured properly.	Vulnerable to theft or tampering if not secured properly.	Secure when implemented correctly; may include security mechanisms like signatures.

Aspect	Session	Cookie	Token
Usage	Used for maintaining stateful sessions in traditional web applications.	Used for various purposes, such as user preferences or tracking.	Used for authentication and authorization in stateless applications and APIs.

Implementing authentication and authorization using Express and JSON Web Tokens (JWT) typically involves several steps. Here's a high-level overview:

1) **Set Up Express:** Create a new Express.js application or use an existing one.
2) **User Model:** Define a user model that represents the users of your application. This model should include fields like username, password, and any other user-specific data
3) **User Registration:** Create an endpoint to allow users to register by providing their credentials. Hash and store their password securely in a database.
4) **User Login:** Implement a login endpoint where users can provide their credentials. Verify the credentials and generate a JWT if they are correct.
5) **JWT Generation:** When a user successfully logs in, create a JWT that includes their user ID or other relevant information as claims. Sign the token with a secret key.
6) **JWT Storage:** Store the JWT on the client-side (usually in a cookie or localStorage) to send it with subsequent requests.
7) **Protected Routes:** Define the routes that require authentication and authorization. Middleware can be used to check if a valid JWT is included in the request headers.
8) **JWT Verification:** Create middleware to verify the JWT in incoming requests. Check the token's signature, expiration, and other relevant claims.
9) **Authorization:** Determine whether the user associated with the JWT has the necessary permissions to access the requested resource. You can use roles or permissions to manage this.
10) **Response Handling:** Return appropriate responses based on the results of authorization and authentication. This may include error messages or the requested data.
11) **Token Refresh (Optional):** Implement a token refresh mechanism, allowing users to obtain a new JWT without needing to re-enter their credentials.

CODE:

app.js:

```
require("dotenv").config();
const express = require("express"),
  app = express(),
  mongoose = require("mongoose"),
  userRoutes = require("./routes/user");
try {
  mongoose.connect("mongodb://localhost:27017/usersdb", {
    useUnifiedTopology: true,
    useNewUrlParser: true });
  console.log("connected to db");
} catch (error) {
  handleError(error); }
process.on('unhandledRejection', error => {
  console.log('unhandledRejection', error.message);
});
app.use(express.json());
app.use(express.urlencoded({
  extended: true
}));
app.use(userRoutes);
app.listen(process.env.PORT || 3000, () => {
  console.log("Server is live on port 3000");})
```

routes/user.js:

```
var express = require("express"), router = express.Router(), verifyToken =
require('../middlewares/authJWT.js'),
  { signup, signin } = require("../controllers/auth.controller.js");
router.post("/register", signup, function (req, res) {
});
router.post("/login", signin, function (req, res) {
});
router.get("/hiddencontent", verifyToken, function (req, res) {
  if (!req.user) {
    res.status(403)
      .send({
        message: "Invalid JWT token"
      }); }
  if (req.user.role  == "admin") {
    res.status(200)
      .send({ message: "Congratulations! but there is no hidden content" });
  } else {
    res.status(403)
      .send({
```

```
        message: "Unauthorised access"
      });
  }
});
module.exports = router;
```

models/user.js

```
var mongoose = require('mongoose'),
  Schema = mongoose.Schema;

/**
 * User Schema
 */
var userSchema = new Schema({
  fullName: {
    type: String,
    required: [true, "fullname not provided "],
  },
  email: {
    type: String,
    unique: [true, "email already exists in database!"],
    lowercase: true,
    trim: true,
    required: [true, "email not provided"],
    validate: {
      validator: function (v) {
        return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(v);
      },
      message: '{VALUE} is not a valid email!'
    }

  },
  role: {
    type: String,
    enum: ["normal", "admin"],
    required: [true, "Please specify user role"]
  },
  password: {
    type: String,
    required: true
  },
  created: {
    type: Date,
    default: Date.now
  }
});

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

middleware/authJWT.js

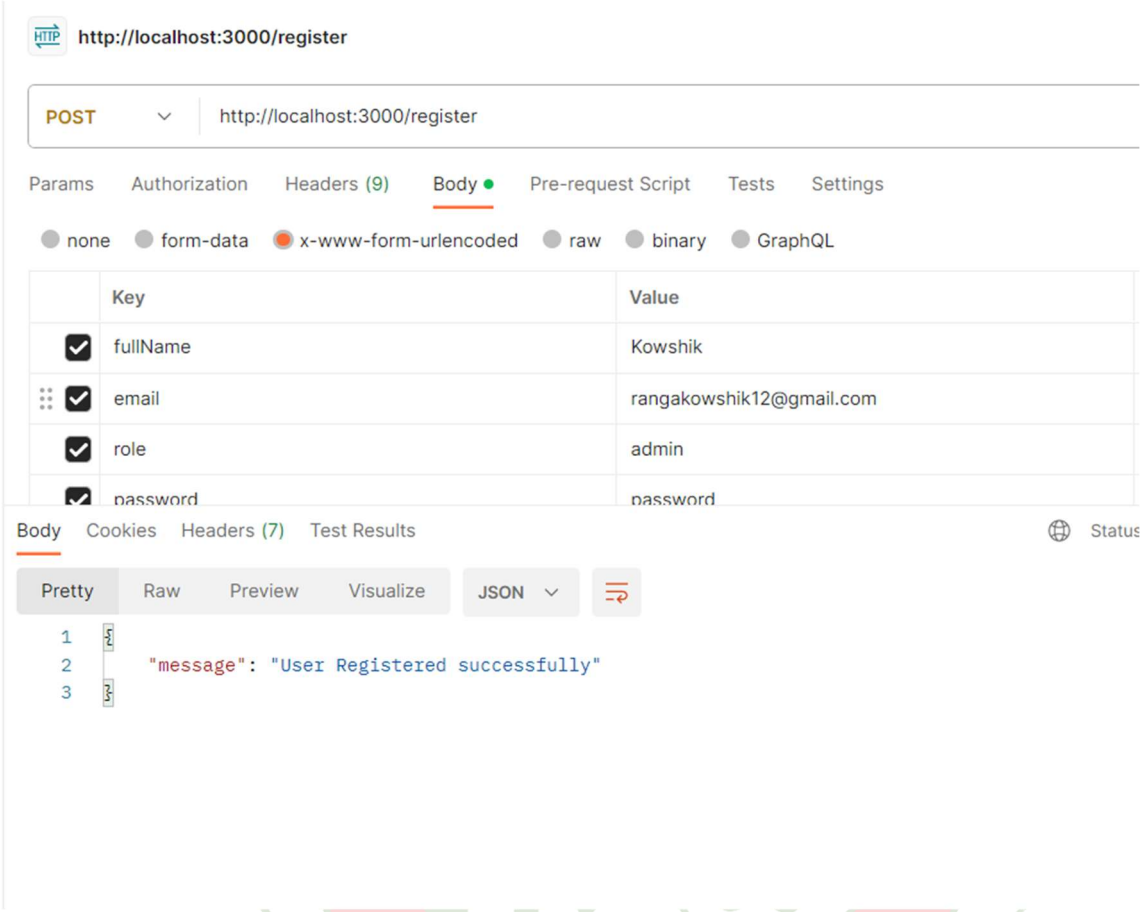
```
const jwt = require("jsonwebtoken");
const User = require("../models/user");
const verifyToken = async (req, res, next) => {
  try {
    if (req.headers && req.headers.authorization && req.headers.authorization.split(' ')[0] ===
'JWT') {
      const token = req.headers.authorization.split(' ')[1];
      const decoded = jwt.verify(token, process.env.API_SECRET);
      const user = await User.findOne({ _id: decoded.id }).exec();
      if (!user) {
        return res.status(404).send({ message: "User Not found." });
      }
req.user = user;
      next();
    } else {
      req.user = undefined;
      next();
    }
  } catch (error) {
    res.status(500).send({ message: error.message });
  }
};

module.exports = verifyToken;
```

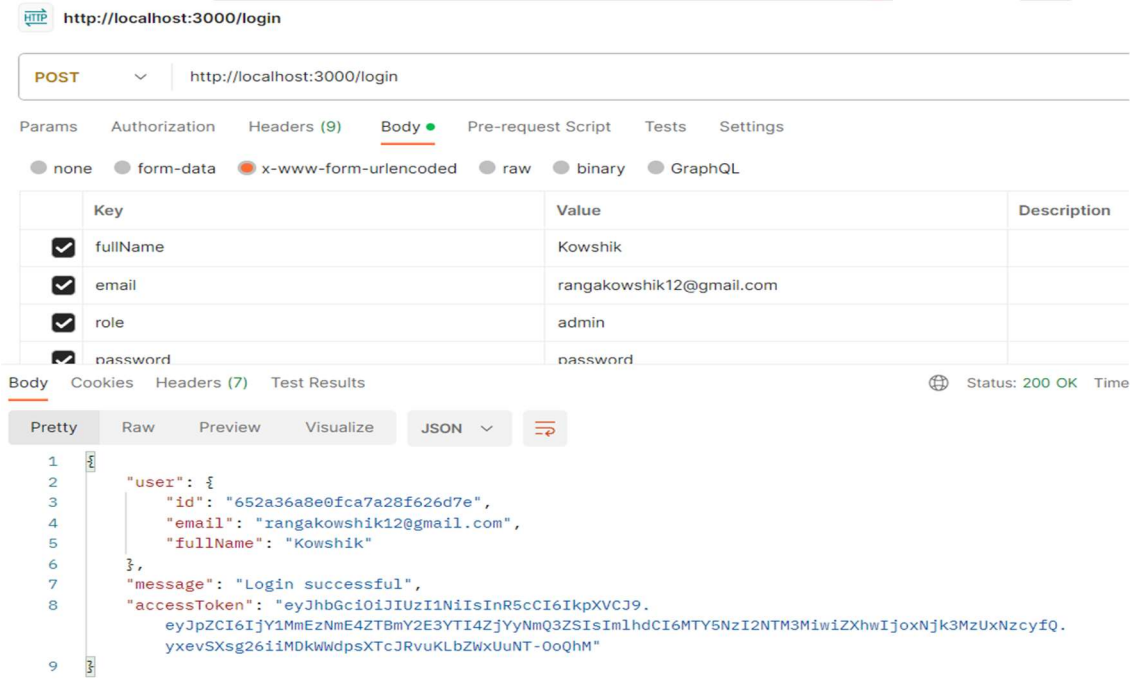
controllers/auth.controller.js

```
const jwt = require("jsonwebtoken");
const bcrypt = require("bcrypt");
const User = require("../models/user");
exports.signup = (req, res) => {
  const { fullName, email, role, password } = req.body;
 bcrypt.hash(password, 8, (err, hashedPassword) => {
    if (err) {
      return res.status(500).send({ message: err.message });
    }

    const user = new User({
      fullName,
      email,
      role,
      password: hashedPassword
    });

    user.save((err) => {
      if (err) {
        res.status(500).send({ message: err.message });
      } else {
```

```
        res.status(200).send({ message: "User Registered successfully" });
      }
    });
  });
};

exports.signin = (req, res) => {
  const { email, password } = req.body;

  User.findOne({ email }, (err, user) => {
    if (err) {
      return res.status(500).send({ message: err.message });
    }
    if (!user) {
      return res.status(404).send({ message: "User Not found." });
    }

    bcrypt.compare(password, user.password, (err, passwordIsValid) => {
      if (err) {
        return res.status(500).send({ message: err.message });
      }

      if (!passwordIsValid) {
        return res.status(401).send({
          accessToken: null,
          message: "Invalid Password!"
        });
      }

      const token = jwt.sign({ id: user.id }, process.env.API_SECRET, {
        expiresIn: 86400
      });

      res.status(200).send({
        user: {
          id: user._id,
          email: user.email,
          fullName: user.fullName
        },
        message: "Login successful",
        accessToken: token
      });
    });
  });
};
```

OUTPUT:


http://localhost:3000/register
POST
http://localhost:3000/register
Params
Authorization
Headers (9)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
Key
Value
fullName
Kowshik
email
rangakowshik12@gmail.com
role
admin
password
password
Body
Cookies
Headers (7)
Test Results
Status
Pretty
Raw
Preview
Visualize
JSON
{
"message": "User Registered successfully"
}


http://localhost:3000/login
POST
http://localhost:3000/login
Params
Authorization
Headers (9)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
Key
Value
Description
fullName
Kowshik
email
rangakowshik12@gmail.com
role
admin
password
password
Body
Cookies
Headers (7)
Test Results
Status: 200 OK
Time
Pretty
Raw
Preview
Visualize
JSON
{
"user": {
"id": "652a36a8e0fca7a28f626d7e",
"email": "rangakowshik12@gmail.com",
"fullName": "Kowshik"
},
"message": "Login successful",
"accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJpZCI6IjY1MmEzNmE4ZTBmY2E3YTI4ZjYyNmQ3ZSIsImlhdCI6MTY5NzI2NTM3MiwiZXhwIjoxNjk3MzUxNzcyfQ.
yxevSXsg26iiMDkWWdpsXTcJRvuKLbZWxUuNT-OoQhM"
}

http://localhost:3000/hiddencontent

GET http://localhost:3000/hiddencontent

Params Authorization Headers (9) Body Pre-request Script Tests Settings

Headers 8 hidden

Key	Value	Descriptio
Authorization	JWT eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6Ij...	
Key	Value	Descriptio

Body Cookies Headers (7) Test Results Status: 200 OK Tin

Pretty Raw Preview Visualize JSON

```
1 {
2     "message": "Congratulations! but there is no hidden content"
3 }
```

AIM: Building and Deploying React App.

DESCRIPTION:

A React app is a web application built using the React JavaScript library. React, developed by Facebook, is a popular open-source library for building user interfaces, particularly for single-page applications (SPAs).

React follows a component-based architecture, where the UI is divided into reusable components that manage their own state and can be composed to build complex user interfaces. These components are written using JavaScript (JSX syntax), which combines HTML-like syntax with JavaScript code.

Key features of React that make it a powerful framework for building web applications include:

Virtual DOM: React uses a virtual representation of the DOM (Document Object Model) called the Virtual DOM. This allows React to efficiently update and render only the necessary parts of the UI, resulting in improved performance.

Component Reusability: React promotes reusability by breaking the UI into independent and reusable components. These components can be composed and combined to build complex interfaces, making the code more modular and maintainable.

One-Way Data Flow: React follows a one-way data flow, where data is passed from parent components to child components through props. This helps to maintain a predictable state and makes it easier to debug and reason about the application.

Declarative Syntax: React uses a declarative approach, where you define how the UI should look based on the current state, and React takes care of updating the actual DOM to reflect the desired state. This simplifies the development process and reduces the amount of manual DOM manipulation.

React Router: React Router is a popular routing library for React applications. It allows developers to implement client-side routing, enabling navigation and rendering of different components based on the URL without requiring a full page reload.

React Context: React Context provides a mechanism for sharing state between components without passing props through multiple levels. It allows data to be accessed by any component within a defined context, making it useful for managing global state or sharing data that is used across different components.

React Hooks: Introduced in React 16.8, Hooks allow developers to use state and other React features in functional components, removing the need to write class components. Hooks provide a simpler and more intuitive way to manage state and lifecycle in React applications.

React can be combined with other libraries and tools, such as Redux for centralized state management, Axios for handling API requests, and many UI libraries and frameworks for styling and component design.

DEMONSTARTION:

Building and deploying a React app involves several steps, including setting up the development environment, creating a production build, and deploying the application to a hosting provider.

- Set up the Development Environment:

  - Install Node.js: Visit the official Node.js website (https://nodejs.org) and download the latest LTS version. Follow the installation instructions for your operating system.
  - Create a new React app: Open a terminal or command prompt and run the following command to create a new React app using Create React App: npx create-react-app my-app
  - Navigate to the app directory: cd my-app
  - Start the development server: npm start

This will launch the app in development mode, and you can view it in your browser at http://localhost:3000.

- **Develop the React App:**

Open your preferred code editor and modify the code in the src directory to develop your React app. Add components, styles, and functionality according to your project requirements.

- **Test the React App:**

Write unit tests for your components using testing frameworks like Jest or React Testing Library. Run the tests to ensure the app functions as expected: npm test

- **Create a Production Build:**

When you're ready to deploy your app, generate a production build by running the following command: npm run build

  - This command creates an optimized and minified version of your app in the build directory.

- **Choose a Hosting Provider:**

Select a hosting provider to deploy your React app. Some popular options include Netlify, Vercel, Firebase, AWS Amplify, and GitHub Pages. Compare their features, pricing, and deployment options to choose the one that suits your needs.

- **Deploy the React App:**

  - Follow the hosting provider's documentation to deploy your React app. Generally, you need to sign up for an account, create a new project, and configure the deployment settings. Most hosting providers allow you to deploy directly from a Git repository or by uploading the build files.
  - Once deployed, the hosting provider will provide you with a URL where your React app is accessible to the public.

AIM: Demonstration of component intercommunication using ReactJS

DESCRIPTION:

Component intercommunication in ReactJS refers to the ability of different components to communicate and share data with each other. It involves passing data, invoking functions, and managing state between components to enable coordination and interaction within the application.

There are several approaches to achieve component intercommunication in ReactJS:

Props:

- Props (short for properties) are used to pass data from a parent component to its child components.
- The parent component can pass data or functions as props, which can then be accessed and used by the child components. This allows for one-way communication from parent to child.

Callbacks:

- Callbacks are functions passed from a parent component to its child components as props.
- Child components can invoke these callbacks, passing data back to the parent component. This enables two-way communication between parent and child components.

Context API:

- React's Context API allows for the creation of a global state that can be accessed by multiple components throughout the component tree.
- It eliminates the need to pass props through intermediate components.
- Context provides a way to share data between components without explicitly passing it down as props.

State Management Libraries:

- State management libraries like Redux or MobX can be used to manage the application state and facilitate intercommunication between components.
- These libraries provide a central store to store and update data, which can be accessed by any component within the application.

Event Emitter/Subscriber Pattern:

- An event emitter/subscriber pattern can be implemented using libraries like EventEmitter3 or PubSubJS.
- Components can emit events or subscribe to events, allowing for communication between unrelated components.

These approaches can be combined or used independently based on the complexity and requirements of the application. The choice depends on factors such as the component hierarchy, the amount of shared data, and the desired level of decoupling between components.

DEMONSTRATION:

Let's walk through a demonstration of component intercommunication using ReactJS.Suppose we have two components: ParentComponent and ChildComponent. The ParentComponent will pass data and a callback function to the ChildComponent, which will then use that data and invoke the callback function to communicate back to the parent.

Here's an example implementation:

ParentComponent.js:

```
import React, { useState } from 'react';
import ChildComponent from './ChildComponent';
const ParentComponent = () => {
 const [message, setMessage] = useState('');
 const handleMessageChange = (newMessage) => {
 setMessage(newMessage);
 };
 return (
 <div>
 <h2>Parent Component</h2>
 <ChildComponent message={message} onMessageChange={handleMessageChange} />
 <p>Message from ChildComponent: {message}</p>
 </div>
 );
};
export default ParentComponent;
```

ChildComponent.js:

```
import React from 'react';
const ChildComponent = ({ message, onMessageChange }) => {
 const handleChange = (event) => {
 const newMessage = event.target.value;
 onMessageChange(newMessage);
 };
 return (
 <div>
 <h3>Child Component</h3>
 <input type="text" value={message} onChange={handleChange} />
 </div>
 );
};
export default ChildComponent
```

- In the ParentComponent, we define the message state using the useState hook. We also define the handleMessageChange callback function, which updates the message state when invoked.
- The ParentComponent renders the ChildComponent and passes the message state and the handleMessageChange callback as props.
- In the ChildComponent, we destructure the message and onMessageChange props. When the input field value changes, we invoke the handleChange function, which extracts the new message from the event and calls the onMessageChange callback with the new message as an argument.
- As a result, when the user types in the input field of the ChildComponent, the handleChange function is triggered, and the updated message is passed back to the ParentComponent through the onMessageChange callback. The ParentComponent updates its message state accordingly and renders the new message value.

This demonstrates how data and intercommunication can flow between parent and child components in ReactJS.

CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY
స్వయం తేజస్వినా భవ
1979

AIM: To Create a form to edit the data using Angular2.

DESCRIPTION:

Angular 2 is a popular open-source, front-end web application framework that is designed to help developers build dynamic, responsive, and scalable web applications. It is a complete rewrite of the original AngularJS framework, with a focus on simplicity, performance, and scalability. Angular 2 is based on a component-based architecture, where each component encapsulates the presentation and logic for a specific feature or functionality. Components can be nested within other components, allowing developers to create complex UI elements with ease.

One of the key features of Angular 2 is its use of declarative templates, which allow developers to define the UI structure and behavior using HTML and a set of directives. These directives provide a way to bind data to the UI, listen to events, and create reusable components.

Angular 2 also includes a powerful set of tools and libraries, such as RxJS for reactive programming, TypeScript for type-checking and compile-time error detection, and the Angular CLI for automating project setup and build processes.

In addition, Angular 2 provides support for internationalization and accessibility, making it easier for developers to create applications that can be used by a global audience. It also has built-in support for testing, with a comprehensive suite of testing tools and libraries.

DEMONSTRATION:

Step 1:

- Set up a new Angular 2 project using the Angular CLI. npm install -g @angular/cli
- Create new project by this command, Choose yes for routing option and, CSS ng new myNewApp
- Go to your project directory cd myNewApp
- Run server and see your application in action ng serve

Step 2:

- Create a new component for your form using the Angular CLI ng generate component edit-form

Step 3:

- In your component's HTML file (edit-form.component.html), create a form using the Angular

ngForm directive:

```
<form #editForm="ngForm" (ngSubmit)="onSubmit()">
```

```
<label for="name">Name:</label>
<input type="text" id="name" name="name" [(ngModel)]="user.name" required>
<label for="email">Email:</label>
<input type="email" id="email" name="email" [(ngModel)]="user.email" required>
<button type="submit" [disabled]="editForm.invalid">Submit</button>
</form>
```

- This form has two input fields for the user's name and email, and a submit button.
- The ngModel directive is used to bind the form fields to a user object in your component's TypeScript file

Step 4:

- In your component's TypeScript file (edit-form.component.ts), define the user object and write a function to handle form submissions:

```
import { Component } from '@angular/core'; @Component({
selector: 'app-edit-form',
templateUrl: './edit-form.component.html', styleUrls: ['./edit-form.component.css']
})
export class EditFormComponent {
user = {
name: 'John Doe',
email: 'johndoe@example.com'
};
onSubmit() {
console.log('User data submitted: ', this.user);  }
}
```

- In this example, the user object is pre-populated with some sample data. When the form is submitted, the onSubmit() function logs the user object to the console.

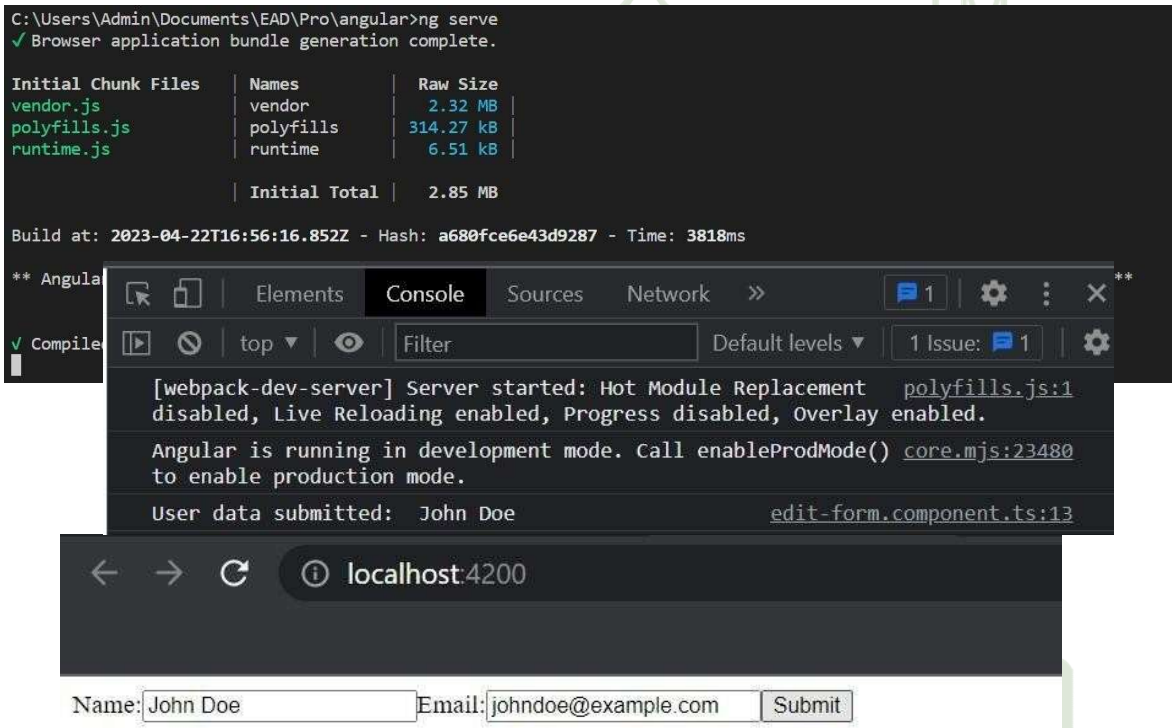
Step 5:

- Add your form component to your application by adding it to your app.module.ts file:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { EditFormComponent } from './edit-form/edit-form.component';@NgModule({
declarations: [
AppComponent, EditFormComponent
],
imports: [ BrowserModule, FormsModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 6:

- Run your application using the Angular CLI: ng serve
- This should start a development server and open your application in a browser. You should now be able to see your form and submit data to it



INSTITUTE OF TECHNOLOGY
స్వయం తేజస్విన్ భవ
1979