SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJS22ITL604 **DATE: 21-02-2025**
COURSE NAME: Full Stack Web Development Laboratory **CLASS: TYBTech**

NAME: Anish Sharma **ROLL: I011** **DIV: IT**

EXPERIMENT NO. 04

CO/LO: CO1-Develop a full stack web application.

AIM / OBJECTIVE: Connecting Express.js with MongoDB Set up database connections, define schemas, and execute CRUD operations using Mongoose.

THEORY:

Express.js is a web application framework for Node.js that simplifies building server-side applications. MongoDB is a NoSQL database that stores data in JSON-like documents. Mongoose is an Object Data Modeling (ODM) library that provides a structured way to interact with MongoDB.

The process of connecting Express.js with MongoDB using Mongoose involves **setting up a database connection, defining schemas, and performing CRUD operations**.

Setting Up the Database Connection

To connect Express.js with MongoDB, we need to establish a connection between the application and the database. This is typically done using Mongoose, which provides a simple and efficient way to manage MongoDB interactions. The database connection is often configured using **environment variables** to ensure security and flexibility.

Once the connection is established, the application can communicate with the MongoDB database to store, retrieve, update, and delete data.

Defining Schemas and Models

A **schema** in Mongoose defines the structure of documents within a MongoDB collection. It specifies the fields, their data types, and any validation rules.

A **model** is created based on the schema and serves as an interface to interact with the MongoDB collection. Models allow us to perform operations like inserting, retrieving, updating, and deleting documents.

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

Schemas can also include timestamps, default values, and data validation rules to ensure data consistency. **Performing CRUD Operations**
CRUD operations—**Create, Read, Update, and Delete**—are essential for managing data in a database. Mongoose simplifies these operations with built-in methods.

- **Create:** Adding new documents to the database.
- **Read:** Retrieving data, either all documents or specific ones based on conditions.
- **Update:** Modifying existing records using filtering criteria.
- **Delete:** Removing documents from the database.

Each of these operations is typically handled within API routes in an Express.js application, allowing seamless interaction between the frontend and the database.

Middleware and Error Handling

Mongoose supports middleware functions that allow pre- and post-processing of database operations. Middleware can be used for **data validation, logging, and automatic updates** before saving or retrieving data.

Error handling is an important part of database interactions. Common errors include **connection failures, validation errors, and duplicate entries**. Proper error handling ensures the application remains stable and provides meaningful feedback to users.

- **Use Environment Variables:** Store database credentials securely in `.env` files.
- **Modularize Code:** Separate database connection logic, model definitions, and API routes for better maintainability.
- **Enable Validation:** Use Mongoose's schema validation to ensure data integrity.
- **Handle Errors Gracefully:** Implement proper error handling to avoid application crashes.
- **Use Indexing for Performance:** Optimize database queries using indexing when dealing with large datasets.

PROCEDURE:

Step 1: Install MongoDB and Mongoose

Ensure MongoDB is installed and running on your system or create a free cluster on MongoDB Atlas.

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

Install Mongoose in your Node.js project using:

npm install mongoose

Step 2: Connect to MongoDB

Modify server.js to connect Express.js with MongoDB using Mongoose:

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

mongoose.connect('mongodb://localhost:27017/expressdb', {
   useNewUrlParser: true,
useUnifiedTopology: true
})
.then(() => console.log('MongoDB connected successfully'))
.catch(err => console.error('MongoDB connection error:', err));
```

Step 3: Design Schema and Model using Mongoose Create

a new folder named models and add a file Item.js:

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

const itemSchema = new
mongoose.Schema({     name: String,
description: String,     price: Number
});

const Item = mongoose.model('Item', itemSchema); module.exports
= Item;
```

Step 4: Implement CRUD Operations

Update server.js to include routes for CRUD operations:

1. Create (POST) - Add a new item to the database const Item = require('./models/Item');

```
app.post('/items', async (req, res) => {
try {
      const newItem = new Item(req.body);
      await newItem.save();
```

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

```
    res.status(201).json({ message: 'Item created successfully', data: newItem });
  } catch (err) {
    res.status(500).json({ message: 'Error creating item', error: err });
  }
});
2. Read (GET) - Retrieve all items from the database
app.get('/items', async (req, res) => {
try {
    const items = await Item.find();
    res.json({ message: 'List of all items', data: items });
  } catch (err) {
    res.status(500).json({ message: 'Error retrieving items', error: err });
  }
});
3. Update (PUT) - Modify an existing item
app.put('/items/:id', async (req, res) => {
try {
    const updatedItem = await Item.findByIdAndUpdate(req.params.id, req.body, { new: true });
res.json({ message: 'Item updated successfully', data: updatedItem });
  } catch (err) {
    res.status(500).json({ message: 'Error updating item', error: err });
  }
});
4. Delete (DELETE) - Remove an item
app.delete('/items/:id', async (req, res) => {
try {
    await Item.findByIdAndDelete(req.params.id);
res.json({ message: 'Item deleted successfully' });
  } catch (err) {
    res.status(500).json({ message: 'Error deleting item', error: err });
  }
});
```

Step 5: Testing APIs using Postman

1. Open Postman and create a new request.
2. Use the following API endpoints to test the CRUD operations: ○ **GET**: http://localhost:3000/items

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

- **POST**: http://localhost:3000/items (with JSON body)
- **PUT**: http://localhost:3000/items/{id}
- **DELETE**: http://localhost:3000/items/{id}

3. Observe the responses and ensure that the database is updating correctly.

Key Concepts:

1. Middleware in Express.js

Middleware functions are functions that execute during the request-response cycle in Express.js. They can modify the request, response, or terminate the request.

In Express.js, **middleware** refers to functions that have access to the request (`req`), response (`res`), and the next middleware function in the application's request-response cycle. Middleware functions are executed sequentially, and they can perform various tasks, such as:

1. **Modifying request or response objects**: Middleware can add properties to the request and response objects, such as adding data or modifying headers.
2. **Executing code**: Middleware can execute code before passing control to the next middleware.
3. **Ending the request-response cycle**: Middleware can end the request-response cycle by sending a response back to the client, bypassing the need for further middleware.
4. **Passing control to the next middleware**: If a middleware doesn't end the cycle, it calls `next()` to pass control to the next middleware function.

Example of logging middleware:

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

// 1. Middleware to log requests app.use((req,
res, next) => {
  console.log(`${req.method} request made to ${req.url}`);
next();  // Pass control to the next middleware
});

// 2. Middleware to check if the user is authenticated const
checkAuth = (req, res, next) => {
  const isAuthenticated = req.headers['authorization'] === 'Bearer secretToken';
// Mock check   if
(isAuthenticated) {
    next();  // If authenticated, pass to next middleware or route handler
```

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

```
  } else {
    res.status(401).send('Unauthorized');  // If not authenticated, send error
  }
};

// 3. Route that uses authentication middleware
app.get('/protected', checkAuth, (req, res) => {
res.send('This is a protected route');
});

// 4. Route that doesn't need authentication app.get('/',
(req, res) => {
  res.send('Welcome to the public route!');
});

// 5. Error-handling middleware (optional)
app.use((err, req, res, next) => {
console.error(err);
  res.status(500).send('Something went wrong!'); });

// Start the server app.listen(3000,
() => {
  console.log('Server running on port 3000');
});
```

 Test using Postman

2. Field Validation in Mongoose

Mongoose provides **schema validation** to ensure that data stored in a MongoDB database meets certain requirements. You can define validation rules when creating a **Schema**, ensuring data integrity before saving documents.

1. Basic Field Validation

You can define validation rules inside the schema by using options like:

- `required` → Makes the field mandatory.
- `unique` → Ensures the field value is unique across all documents.
- `minlength` / `maxlength` → Limits string length. `min` / `max` → Limits number values.
- `enum` → Restricts field values to a predefined list.

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

- `match` → Validates string fields using regex patterns.
- `default` → Assigns a default value if none is provided.

Example Schema with Field Validation

```
const mongoose = require('mongoose');
 const userSchema = new
mongoose.Schema({   name: {     type:
String,
    required: [true, 'Name is required'],  // Custom error message
minlength: [3, 'Name must be at least 3 characters'],     maxlength:
[50, 'Name must not exceed 50 characters']
  },   email: {
type: String,
required: true,
    unique: true,  // Ensures no duplicate emails
    match: [/.+@.+\..+/, 'Please enter a valid email']  // Regex for basic email
validation   },   age: {     type: Number,
    min: [18, 'Age must be at least 18'],  // Minimum age
    max: [100, 'Age cannot be more than 100']
  },   role: {
type: String,
    enum: ['admin', 'user', 'guest'],  // Only allow these values
default: 'user'
  },
createdAt: {
type: Date,
    default: Date.now  // Assigns current date if not provided
  }
});
const User = mongoose.model('User', userSchema);
```

2. Custom Validation with a Function

You can define **custom validation functions** if built-in validators don't meet your requirements.

Example: Custom Validator for Password Strength

```
const userSchema = new mongoose.Schema({
password: {     type: String,
required: true,     validate: {
      validator: function(value) {
        return /^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{6,}$/.test(value);
      },
      message: 'Password must be at least 6 characters long and include
uppercase, lowercase, and a number'
    }
```

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

```
  }
});
```

3. Async Validation (e.g., Checking Database Before Saving)

If validation requires checking the database (like ensuring a username isn't already taken), you can use **asynchronous validation**.

Example: Custom Async Validator

```
const userSchema = new mongoose.Schema({
username: {     type: String,
required: true,     validate: {
      validator: async function(value) {
        const user = await mongoose.model('User').findOne({ username: value });
return !user; // If user exists, return false
      },
      message: 'Username already exists'
    }
  }
});
```

4. Handling Validation Errors

Mongoose throws a `ValidationError` when validation fails. You can catch these errors when saving a document.

Example: Handling Validation Errors

```
const newUser = new User({   name:
'Jo',  // Too short (will fail)
  email: 'invalidemail',  // Invalid format (will fail)
age: 15  // Below minimum age (will fail)
});
newUser.save()
  .then(() => console.log('User saved successfully'))
  .catch(err => {
    if (err.name === 'ValidationError') {
      console.error('Validation Error:', err.message);
    } else {
      console.error(err);
    }
  });
```

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

5. Disabling Mongoose Validation

If needed, you can disable validation using:

```
user.save({ validateBeforeSave: false });
```

3. Pagination in MongoDB

Pagination helps manage large datasets by retrieving a limited number of documents per request. Pagination is the process of retrieving a subset of data (or "pages") from a larger dataset to improve performance and user experience. In MongoDB, pagination is commonly implemented using .skip() and .limit(), cursors, or aggregation pipelines.

Example implementation:

1. Basic Pagination Using **.skip()** and **.limit()**

The `.skip(n)` method skips `n` documents, and `.limit(m)` restricts the number of documents retrieved.

Example: Implementing Pagination

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

const app = express();
mongoose.connect('mongodb://localhost:27017/mydatabase');

const Product = mongoose.model('Product', new mongoose.Schema({ name: String,
price: Number }));

app.get('/products', async (req, res) => {
  const page = parseInt(req.query.page) || 1;  // Default to page 1
const limit = parseInt(req.query.limit) || 10; // Default limit is 10
const skip = (page - 1) * limit;
   try
{
    const products = await Product.find().skip(skip).limit(limit);
res.json(products);   } catch (error) {
    res.status(500).json({ error: 'Internal Server Error' });
  }
```

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

```
});   app.listen(3000, () => console.log('Server running on port
3000'));
```

How It Works:

?page=2&limit=10 fetches page 2 with 10 products per page.
.skip((page - 1) * limit) ensures that previous pages are skipped.
.limit(limit) fetches only the required number of documents.

2. Using Cursors for Efficient Pagination

MongoDB provides **cursors** to fetch paginated data more efficiently.

Example: Using Cursors

```
app.get('/products', async (req, res) => {   const
limit = parseInt(req.query.limit) || 10;   let
cursor = Product.find().limit(limit).cursor();

  let results = [];
  for await (const doc of cursor) {
results.push(doc);
  }

  res.json(results);
});
```

Benefit: Cursors use streaming, consuming **less** memory.

3. Pagination Using **sort()** with **_id** for Better Performance

When dealing with large datasets, **using `_id` instead of `.skip()` is more efficient**.

Example: Paginating with _id

```
app.get('/products', async (req, res) => {
const limit = parseInt(req.query.limit) || 10;
  const lastId = req.query.lastId; // The _id of the last document from the
previous page

  let query = lastId ? { _id: { $gt: lastId } } : {};

  const products = await Product.find(query).sort({ _id: 1 }).limit(limit);

  res.json(products);
});
```

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

How It Works:
Fetch the next page by passing the _id of the last document in lastId.
Sorting by _id ensures consistent pagination.
Faster than .skip(), especially for large datasets.

4. Pagination with **countDocuments()** for Total Pages You

may need to calculate the **total number of pages**.

```
app.get('/products', async (req, res) => {
const page = parseInt(req.query.page) || 1;
const limit = parseInt(req.query.limit) || 10;
const skip = (page - 1) * limit;

  const total = await Product.countDocuments();
  const products = await Product.find().skip(skip).limit(limit);

res.json({
total,
    totalPages: Math.ceil(total / limit),
currentPage: page,     products,
  });
});
```

4. Soft Delete Implementation

Soft delete is a technique where a record **is not actually deleted** from the database but is instead **marked as deleted** using a field like isDeleted: true. This allows you to **restore** data later if needed..
Example:

Step 1: Define Mongoose Schema const

```
mongoose = require('mongoose');


const productSchema = new mongoose.Schema({
 name: String,   price: Number,   isDeleted: { type: Boolean,
default: false }, // Soft delete flag }, { timestamps: true });
```

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

```
const Product = mongoose.model('Product', productSchema);
```

Step 2: Create a Soft Delete Function

Instead of actually deleting the document, we **update isDeleted to true**.

```
app.delete('/products/:id', async (req, res) => {
  try {
    const product = await Product.findByIdAndUpdate(req.params.id, { isDeleted: true });
    if (!product) {
      return res.status(404).json({ message: 'Product not found' });
    }
    res.json({ message: 'Product soft deleted successfully' });
  } catch (error) {    res.status(500).json({ error: 'Internal
Server Error' });   }

});
```

This does not remove the product but marks it as deleted.

Step 3: Fetch Only Active Products (Ignore Deleted Ones) By default, we should **hide deleted products** from queries.

```
app.get('/products', async (req, res) => {   const products =
await Product.find({ isDeleted: false }); // Fetch only non-deleted products
res.json(products);
});
```

Deleted products are hidden but still exist in the database.

Step 4: Restore a Soft-Deleted Product

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

We can create an API to **restore** soft-deleted records. app.put('/products/restore/:id',

async (req, res) => {

try {

const product = await Product.findByIdAndUpdate(req.params.id, { isDeleted: false });

if (!product) { return res.status(404).json({ message:

'Product not found' });

}

res.json({ message: 'Product restored successfully' });

} catch (error) { res.status(500).json({ error:

'Internal Server Error' });

}

});

Now, deleted records can be recovered!

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY TASK:

Inventory Management System Extend the `Item` schema to include properties like `quantity` and `category`.

- Implement additional routes to manage inventory, such as decreasing stock on order placement.
- Develop a React frontend to interact with the backend and visualize inventory data.

Write-up questions:

1. Explain the importance of schema design in MongoDB.
2. What are the benefits of using Mongoose over the native MongoDB driver?

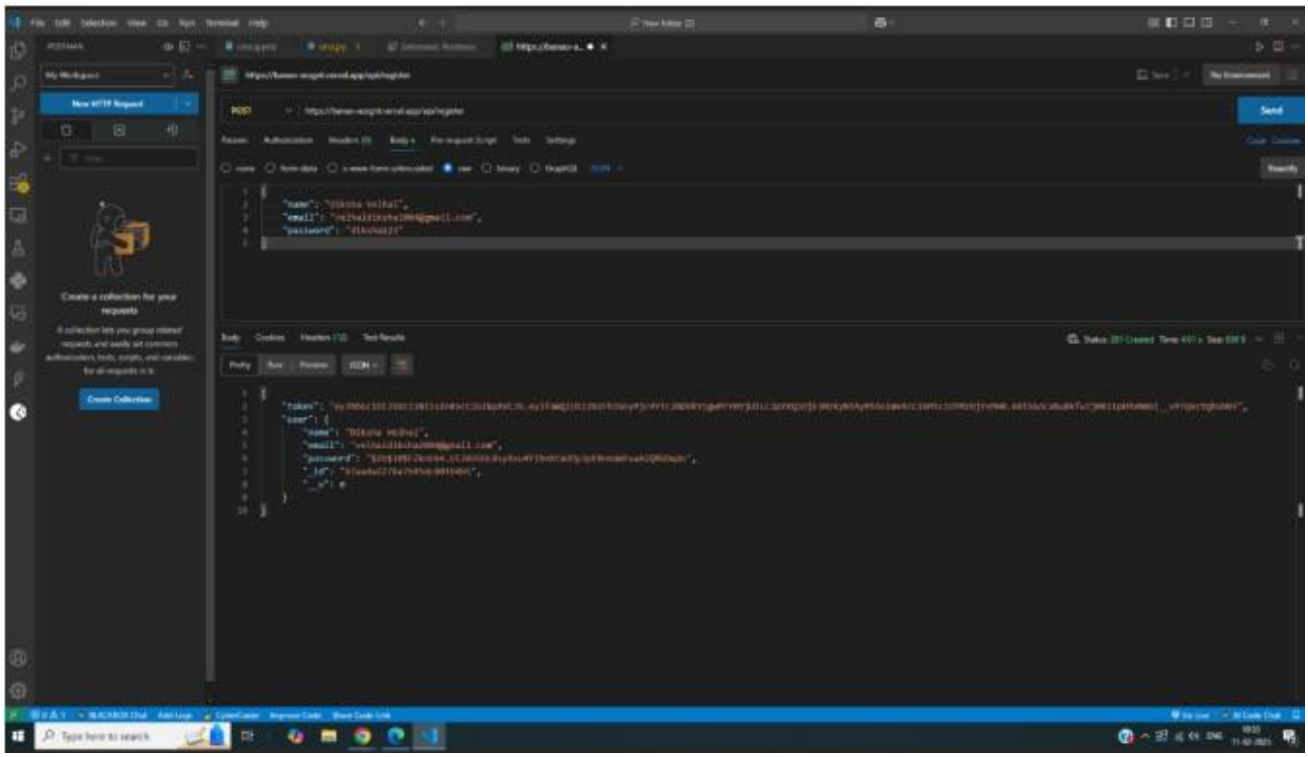
Practice Questions:

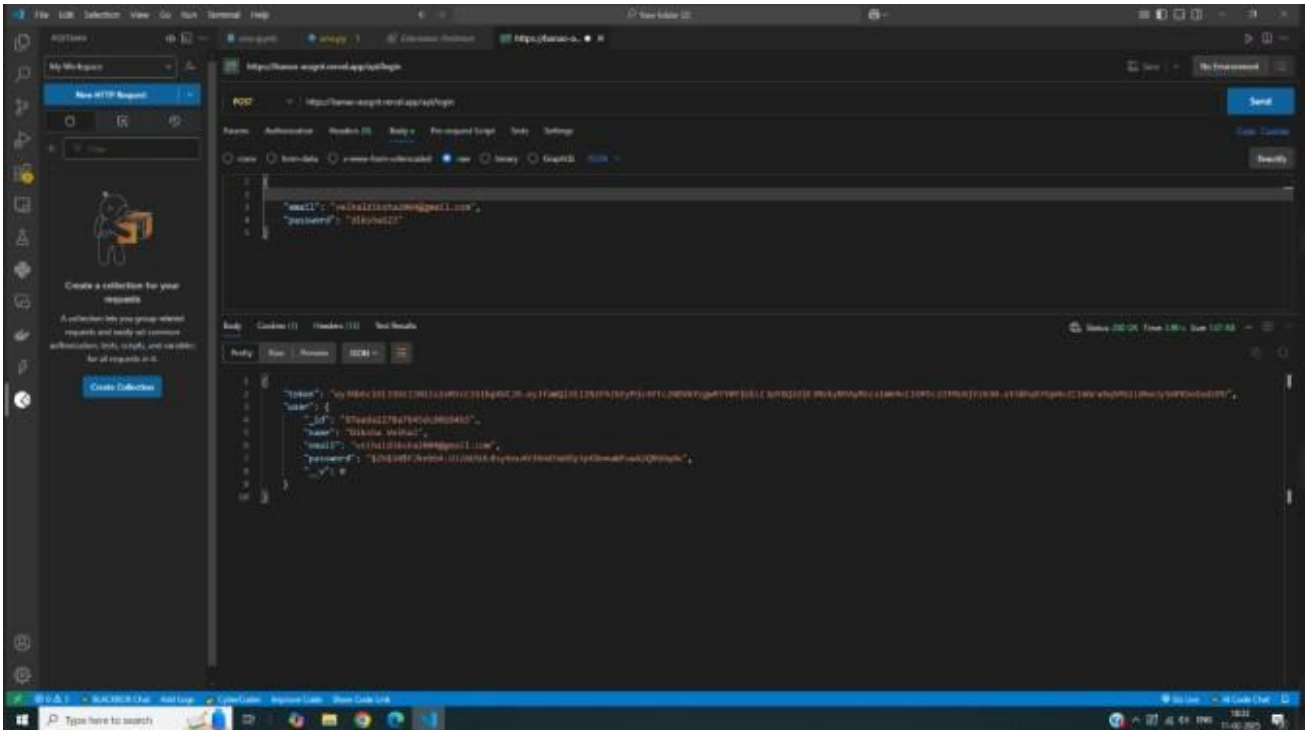
1. Write a Mongoose schema for a user authentication system that includes fields for username, password (hashed), email, and role.
2. Create an Express.js route to search for items in the database based on a keyword in their name or description.
3. Implement pagination in the GET `/items` endpoint to return results in batches of 5 items per page.
4. Write a middleware function in Express.js that logs all incoming requests with their method and URL.
5. Modify the DELETE endpoint to implement a soft delete mechanism instead of permanently removing items.
6. Implement field validation in the POST `/items` route using Mongoose validators.
7. Create a route that retrieves all items from a specific category using query parameters.

OUTPUT:

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

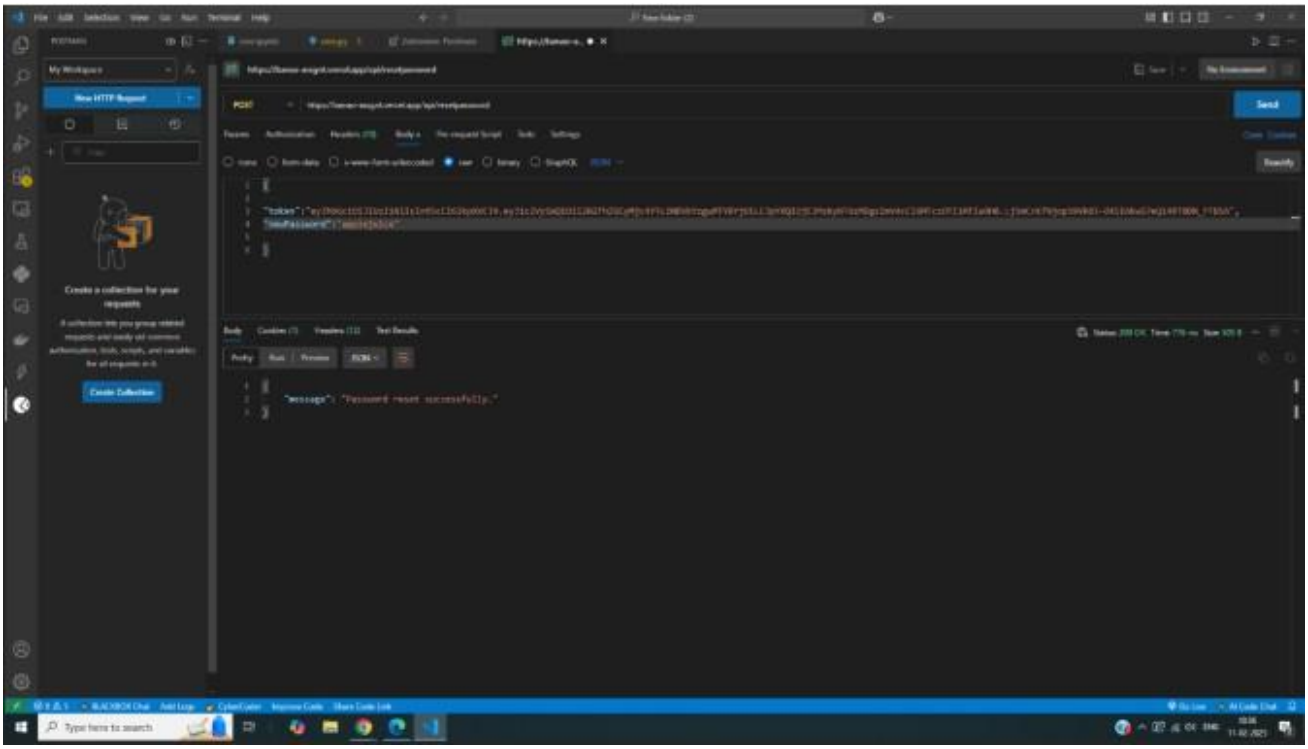
DEPARTMENT OF INFORMATION TECHNOLOGY



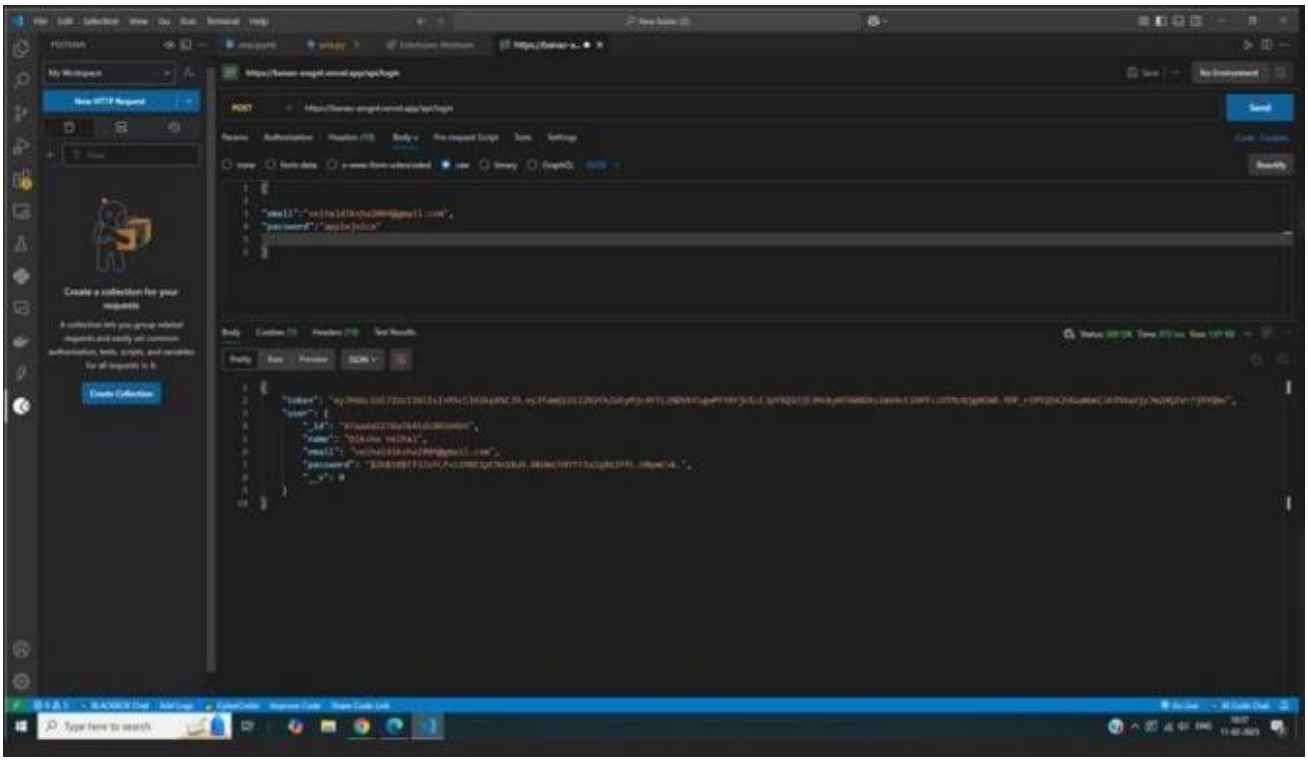


SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY



SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)



DEPARTMENT OF INFORMATION TECHNOLOGY

CONCLUSION: Connected Express.js with MongoDB Set up database connections, define schemas, and execute CRUD operations using Mongoose.

BOOKS AND WEB RESOURCES:

[1] MDN Web Docs - https://developer.mozilla.org/en-US/docs/Learn/Serverside/Express_Nodejs
[2] MongoDB Documentation - https://www.mongodb.com/docs/manual/
[3] Express.js Guide - https://expressjs.com/en/guide/routing.html