Batch: A1	**Roll No.: 16010123012**
Experiment / assignment / tutorial No.: 05	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

Title: Implementation of CRUD Operations using MongoDB and Mongoose.

AIM: Demonstrate the use of Mongoose with CRUD operation.

Problem Definition:

1) Generate Database model

2) Create RESTful API

3) Demonstrate the Endpoints.

Resources used: https://www.mongodb.com/docs/

Expected OUTCOME of Experiment:
CO 2:

Books/ Journals/ Websites referred:

1. Shelly Powers Learning Node O' Reilly 2nd Edition, 2016.

Pre Lab/ Prior Concepts:

Write details about the following content

- **Mongoose CRUD operation**

  Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a schema-based solution to model application data and includes built-in type casting, validation, query building, and business logic hooks.

  CREATE (C)

  Model.create() - Creates a new document

  new Model().save() - Instantiate and save

  await User.create({ name: "John", email: "john@example.com" });

READ (R)

Model.find() - Finds all matching documents

Model.findOne() - Finds single document

Model.findById() - Finds by ID

await User.findById(userId);

await User.find({ email: "john@example.com" });

UPDATE (U)

Model.updateOne() - Updates first matching document

Model.findByIdAndUpdate() - Finds by ID and updates

Model.findOneAndUpdate() - Finds and updates

await User.findByIdAndUpdate(userId, { name: "John Doe" });

DELETE (D)

Model.deleteOne() - Deletes first matching document

Model.findByIdAndDelete() - Finds by ID and deletes

Model.deleteMany() - Deletes multiple documents

await User.findByIdAndDelete(userId);

- **RESTFul API**

  REST (Representational State Transfer) is an architectural style for designing networked applications. A RESTful API uses HTTP requests to perform CRUD operations.

  REST Principles:

  1. Stateless: Each request contains all information needed to process it
  2. Client-Server Architecture: Separation of concerns
  3. Cacheable: Responses must define themselves as cacheable or not
  4. Uniform Interface: Standardized way of communicating
  5. Layered System: Client cannot tell if connected directly to server

Methodology:

1. Database Modeling - A User schema was created using Mongoose to define the structure of the documents in MongoDB. The schema included fields such as name, email, password, favorites, and preferences. Validation rules and password hashing were implemented using Mongoose middleware and the bcryptjs library.
2. API Design and Development - A RESTful API was designed to handle CRUD operations. Endpoints were created for user registration (/signup), login (/login), profile retrieval (/profile), and profile update (/profile using PUT method). Each endpoint interacts with the MongoDB database through Mongoose functions.
3. Authentication and Security - JSON Web Tokens (JWT) were used for authentication to secure protected routes such as profile management. The bcryptjs library ensured password security through hashing.
4. Server and Database Configuration - The server was built using Node.js and Express.js. MongoDB was connected using Mongoose with environment variables to manage credentials securely.
5. Testing and Verification - Postman was used to test all API endpoints. CRUD operations (Create, Read, Update, Delete) were verified to ensure proper interaction between the backend and MongoDB.
6. Execution Environment - The application was executed locally with MongoDB running in the background. Logs were checked to ensure successful operations and error handling.

Implementation Details:

1. User.js

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

const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, 'Name is required'],
    trim: true,
    minlength: [2, 'Name must be at least 2 characters long']
  },
  email: {
    type: String,
    required: [true, 'Email is required'],
    unique: true,
    lowercase: true,
    trim: true,
    match: [/^\S+@\S+\.\S+$/, 'Please provide a valid email']
  },
  password: {
    type: String,
    required: [true, 'Password is required'],
```

```
    minlength: [6, 'Password must be at least 6 characters long']
  },
  favorites: {
    type: [String],
    default: []
  },
  preferences: {
    type: Object,
    default: {}
  }
}, {
  timestamps: true
});

userSchema.pre('save', async function(next) {
  if (!this.isModified('password')) return next();

  const salt = await bcrypt.genSalt(10);
  this.password = await bcrypt.hash(this.password, salt);
  next();
});

userSchema.methods.comparePassword = async function(candidatePassword) {
  return await bcrypt.compare(candidatePassword, this.password);
};

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

2. User.js

```
router.post('/signup', async (req, res) => {
  try {
    const { name, email, password } = req.body;

    const existingUser = await User.findOne({ email });
    if (existingUser) {
      return res.status(400).json({ message: 'User already exists' });
    }

    const user = new User({ name, email, password });
    await user.save();
    const token = generateToken(user._id);

    res.status(201).json({
```

```
      success: true,
      message: 'User registered successfully',
      token,
      user: { id: user._id, name: user.name, email: user.email }
    });
  } catch (error) {
    res.status(500).json({ message: 'Server error' });
  }
});

router.get('/profile', auth, async (req, res) => {
  try {
    const user = await User.findById(req.user.id).select('-password');

    if (!user) {
      return res.status(404).json({ message: 'User not found' });
    }

    res.json({
      success: true,
      user: {
        id: user._id,
        name: user.name,
        email: user.email,
        favorites: user.favorites || [],
        preferences: user.preferences || {},
        createdAt: user.createdAt
      }
    });
  } catch (error) {
    res.status(500).json({ message: 'Server error' });
  }
});

router.post('/login', async (req, res) => {
  try {
    const { email, password } = req.body;
    const user = await User.findOne({ email });
    if (!user) {
      return res.status(401).json({ message: 'Invalid credentials' });
    }

    const isMatch = await user.comparePassword(password);
```

```
    if (!isMatch) {
      return res.status(401).json({ message: 'Invalid credentials' });
    }
    const token = generateToken(user._id);

    res.json({
      success: true,
      token,
      user: { id: user._id, name: user.name, email: user.email }
    });
  } catch (error) {
    res.status(500).json({ message: 'Server error' });
  }
});


router.put('/profile', auth, async (req, res) => {
  try {
    const { name, email } = req.body;
    const user = await User.findById(req.user.id);

    if (!user) {
      return res.status(404).json({ message: 'User not found' });
    }
    if (email && email !== user.email) {
      const emailExists = await User.findOne({ email });
      if (emailExists) {
        return res.status(400).json({ message: 'Email already in use' });
      }
      user.email = email;
    }

    if (name) user.name = name;
    await user.save();

    res.json({
      success: true,
      message: 'Profile updated successfully',
      user: {
        id: user._id,
        name: user.name,
        email: user.email
      }
```

```
    });
  } catch (error) {
    res.status(500).json({ message: 'Server error' });
  }
});
```

Steps for execution:
1. Install Dependencies
2. Configure Environment Variables
3. Install MongoDB
4. Start the Server
5. Test CRUD Operations
6. Verify in MongoDB

Conclusion:
This experiment successfully demonstrated the implementation of CRUD operations using MongoDB and Mongoose in a RESTful API architecture.