MongoDB Collections Setup Guide

1. MongoDB Compass Setup

Creating the Database

1. Open MongoDB Compass
2. Connect to your MongoDB instance (usually `mongodb://localhost:27017`)
3. Click "Create Database"
4. Database name: `8conedge_db` (or your preferred name)
5. Collection name: `users` (we'll create this first)

Collections to Create

Collection 1: `users`

This collection stores all user information including students, teachers, and admins.

Sample Document Structure:

json

```json
{
  "_id": "ObjectId",
  "email": "student@example.com",
  "password": "hashed_password_here",
  "username": "john_doe",
  "name": "John Doe",
  "roles": "student", // "student", "teacher", "admin"

  // Profile Information
  "avatar": "https://example.com/avatar.jpg",
  "phone": "+1234567890",
  "location": "New York, USA",
  "bio": "Computer Science student passionate about learning",

  // Academic Information (for students)
  "studentId": "2024-001234",
  "course": "Bachelor of Science in Computer Science",
  "yearLevel": "3rd Year",
  "department": "Computer Science",

  // Professional Information (for faculty/staff)
  "employeeId": "EMP-2024-001",
  "position": "Assistant Professor",

  // Account Information
  "status": "active", // "active", "inactive", "suspended"
  "isVerified": true,
  "preferences": {
    "theme": "light",
    "notifications": true,
    "language": "en"
  },
  "permissions": ["read_profile", "edit_profile"],

  // Timestamps
  "createdAt": "2024-01-01T00:00:00.000Z",
  "updatedAt": "2024-01-01T00:00:00.000Z",
  "lastLogin": "2024-01-01T00:00:00.000Z"
}
```

Collection 2: `profiles`

Extended profile information (optional - can be embedded in users collection)

```json
{
  "_id": "ObjectId",
  "userId": "ObjectId (reference to users collection)",
  "personalInfo": {
    "firstName": "John",
    "lastName": "Doe",
    "middleName": "Michael",
    "dateOfBirth": "1995-05-15",
    "gender": "Male",
    "nationality": "American"
  },
  "contactInfo": {
    "primaryEmail": "john@example.com",
    "alternateEmail": "john.doe@gmail.com",
    "primaryPhone": "+1234567890",
    "alternatePhone": "+0987654321",
    "address": {
      "street": "123 Main St",
      "city": "New York",
      "state": "NY",
      "zipCode": "10001",
      "country": "USA"
    }
  },
  "academicInfo": {
    "enrollmentDate": "2021-08-15",
    "expectedGraduation": "2025-05-15",
    "gpa": 3.75,
    "credits": 90,
    "advisor": "Dr. Jane Smith"
  },
  "socialLinks": {
    "linkedin": "https://linkedin.com/in/johndoe",
    "github": "https://github.com/johndoe",
    "portfolio": "https://johndoe.dev"
  },
  "createdAt": "2024-01-01T00:00:00.000Z",
  "updatedAt": "2024-01-01T00:00:00.000Z"
}
```

Collection 3: `sessions`

For managing user sessions (optional if using JWT)

```json
{
  "_id": "ObjectId",
  "userId": "ObjectId (reference to users)",
  "sessionToken": "unique_session_token",
  "deviceInfo": {
    "userAgent": "Mozilla/5.0...",
    "ip": "192.168.1.1",
    "device": "Desktop"
  },
  "isActive": true,
  "expiresAt": "2024-01-02T00:00:00.000Z",
  "createdAt": "2024-01-01T00:00:00.000Z"
}
```

2. Creating Collections in MongoDB Compass

Step-by-Step Instructions:

1. **Create Users Collection:**
   - In your database, click "Create Collection"
   - Name: `users`
   - Click "Create Collection"
2. **Add Sample User Document:**
   - Click on the `users` collection
   - Click "Insert Document"
   - Use the JSON view and paste a sample document
   - Click "Insert"
3. **Create Indexes for Performance:**
   - In the `users` collection, go to "Indexes" tab
   - Create indexes for frequently queried fields:
     - Email: `{ "email": 1 }` (unique)
     - Username: `{ "username": 1 }` (unique)

- Student ID: `{ "studentId": 1 }` (unique, sparse)
- Employee ID: `{ "employeeId": 1 }` (unique, sparse)

3. Backend API Endpoints

Express.js Server Setup

javascript

```
// server.js or app.js
const express = require('express');
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');
const cors = require('cors');
const multer = require('multer');
const path = require('path');

const app = express();

// Middleware
app.use(cors({
  origin: 'http://localhost:3000', // Your React app URL
  credentials: true
}));
app.use(express.json());
app.use('/uploads', express.static('uploads')); // For serving uploaded files

// MongoDB Connection
mongoose.connect('mongodb://localhost:27017/8conedge_db', {
  useNewUrlParser: true,
  useUnifiedTopology: true
});

// User Schema
const userSchema = new mongoose.Schema({
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  username: { type: String, unique: true },
  name: String,
  roles: { type: String, enum: ['student', 'teacher', 'admin'], default: 'student' },

  // Profile Information
  avatar: String,
  phone: String,
  location: String,
  bio: String,

  // Academic Information
  studentId: { type: String, unique: true, sparse: true },
  course: String,
  yearLevel: String,
  department: String,
```

```
  // Professional Information
  employeeId: { type: String, unique: true, sparse: true },
  position: String,

  // Account Information
  status: { type: String, enum: ['active', 'inactive', 'suspended'], default: 'active' },
  isVerified: { type: Boolean, default: false },
  preferences: {
    theme: { type: String, default: 'light' },
    notifications: { type: Boolean, default: true },
    language: { type: String, default: 'en' }
  },
  permissions: [String],

  // Timestamps
  lastLogin: Date
}, {
  timestamps: true // Automatically adds createdAt and updatedAt
});

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

// File upload configuration
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/avatars/');
  },
  filename: (req, file, cb) => {
    cb(null, Date.now() + '-' + Math.round(Math.random() * 1E9) + path.extname(file.origina
  }
});
const upload = multer({ storage: storage });

// Routes

// Login Route
app.post('/api/login', async (req, res) => {
  try {
    const { email, password } = req.body;

    // Find user by email
    const user = await User.findOne({ email });
    if (!user) {
```

```
      return res.status(401).json({ success: false, error: 'Invalid credentials' });
    }

    // Check password
    const isValidPassword = await bcrypt.compare(password, user.password);
    if (!isValidPassword) {
      return res.status(401).json({ success: false, error: 'Invalid credentials' });
    }

    // Update last login
    user.lastLogin = new Date();
    await user.save();

    // Remove password from response
    const userResponse = user.toObject();
    delete userResponse.password;

    res.json({
      success: true,
      user: userResponse,
      message: 'Login successful'
    });

  } catch (error) {
    console.error('Login error:', error);
    res.status(500).json({ success: false, error: 'Server error' });
  }
});

// Get User Profile Route
app.get('/api/user/profile', async (req, res) => {
  try {
    // In a real app, you'd get userId from JWT token or session
    const { userId } = req.query;

    const user = await User.findById(userId).select('-password');
    if (!user) {
      return res.status(404).json({ success: false, error: 'User not found' });
    }

    res.json({ success: true, user });
  } catch (error) {
    console.error('Get profile error:', error);
    res.status(500).json({ success: false, error: 'Server error' });
```

```
  }
});

// Update User Profile Route
app.put('/api/user/profile', async (req, res) => {
  try {
    // In a real app, you'd get userId from JWT token or session
    const { userId, ...updateData } = req.body;

    // Remove sensitive fields that shouldn't be updated via this route
    delete updateData.password;
    delete updateData.email; // Email changes should be handled separately
    delete updateData._id;

    const user = await User.findByIdAndUpdate(
      userId,
      { ...updateData, updatedAt: new Date() },
      { new: true, runValidators: true }
    ).select('-password');

    if (!user) {
      return res.status(404).json({ success: false, error: 'User not found' });
    }

    res.json({ success: true, user });
  } catch (error) {
    console.error('Update profile error:', error);
    res.status(500).json({ success: false, error: 'Server error' });
  }
});

// Upload Avatar Route
app.post('/api/user/avatar', upload.single('avatar'), async (req, res) => {
  try {
    if (!req.file) {
      return res.status(400).json({ success: false, error: 'No file uploaded' });
    }

    // In a real app, you'd get userId from JWT token or session
    const { userId } = req.body;

    const avatarUrl = `/uploads/avatars/${req.file.filename}`;

    const user = await User.findByIdAndUpdate(
```

```
      userId,
      { avatar: avatarUrl },
      { new: true }
    ).select('-password');

    res.json({
      success: true,
      avatarUrl,
      user
    });
  } catch (error) {
    console.error('Upload avatar error:', error);
    res.status(500).json({ success: false, error: 'Server error' });
  }
});

// Create User Route (Registration)
app.post('/api/register', async (req, res) => {
  try {
    const { email, password, username, name, roles = 'student' } = req.body;

    // Check if user already exists
    const existingUser = await User.findOne({
      $or: [{ email }, { username }]
    });

    if (existingUser) {
      return res.status(400).json({
        success: false,
        error: 'User with this email or username already exists'
      });
    }

    // Hash password
    const hashedPassword = await bcrypt.hash(password, 10);

    // Create user
    const user = new User({
      email,
      password: hashedPassword,
      username,
      name,
      roles
    });
```

```
    await user.save();

    // Remove password from response
    const userResponse = user.toObject();
    delete userResponse.password;

    res.status(201).json({
      success: true,
      user: userResponse,
      message: 'User created successfully'
    });

  } catch (error) {
    console.error('Registration error:', error);
    res.status(500).json({ success: false, error: 'Server error' });
  }
});

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

4. Environment Setup

Install Required Dependencies:

bash

```
npm install express mongoose bcryptjs cors multer jsonwebtoken
```

Create Folder Structure:

```
your-project/
├── server/
│   ├── server.js
│   └── uploads/
│       └── avatars/
├── client/
│   └── src/
│       ├── components/
│       │   ├── LoginSignupPage.jsx
│       │   └── ProfilePage.jsx
│       └── App.js
```

5. Sample Data for Testing

Here's some sample data you can insert into your `users` collection:

```json
[
  {
    "email": "student@test.com",
    "password": "$2a$10$example_hashed_password",
    "username": "student1",
    "name": "John Doe",
    "roles": "student",
    "studentId": "2024-001",
    "course": "Computer Science",
    "yearLevel": "3rd Year",
    "department": "IT",
    "phone": "+1234567890",
    "location": "New York, USA",
    "status": "active",
    "isVerified": true,
    "createdAt": "2024-01-01T00:00:00.000Z",
    "updatedAt": "2024-01-01T00:00:00.000Z"
  },
  {
    "email": "teacher@test.com",
    "password": "$2a$10$example_hashed_password",
    "username": "prof_smith",
    "name": "Dr. Jane Smith",
    "roles": "teacher",
    "employeeId": "EMP-2024-001",
    "position": "Assistant Professor",
    "department": "Computer Science",
    "phone": "+1234567891",
    "location": "Boston, USA",
    "status": "active",
    "isVerified": true,
    "createdAt": "2024-01-01T00:00:00.000Z",
    "updatedAt": "2024-01-01T00:00:00.000Z"
  }
]
```

6. Next Steps

1. Set up the Express server with the provided code
2. Create the MongoDB collections using Compass
3. Test the API endpoints using Postman or your React app

4. Implement proper authentication (JWT tokens)
5. Add validation and error handling
6. Set up file upload for avatars
7. Add more profile features as needed

Remember to:

- Hash passwords before storing them
- Implement proper authentication middleware
- Add input validation
- Set up proper CORS configuration
- Add rate limiting and security headers
- Implement proper error handling and logging