Node.js CRUD for User model — Flowchart & Full Code

> This document contains a clear **flowchart** and ready-to-run **Node.js (Express + Mongoose)** implementation of CRUD operations for the `User` model you provided. The implementation uses a simple service → controller → route structure and includes DB connection, error handling, and examples for testing.

✅What you get in this doc

- A visual **flowchart** (Mermaid) showing request flow
- Recommended **file structure**
- `.env` and `package.json` examples
- **DB connection** (`db.js`) code
- **Mongoose model** for `User` (keeps the same field names you provided)
- **Service layer** (`userService.js`) with all CRUD logic and password hashing
- **Controller** (`userController.js`) that uses the service and returns HTTP responses
- **Routes** (`userRoutes.js`) for REST endpoints
- **Server** (`server.js`) to bootstrap the app
- **Error handler** middleware
- `curl` / Postman examples

Prerequisites

- Node.js (>=14)
- MongoDB URI (local or Atlas)

Install dependencies used here:

```
npm install express mongoose dotenv bcryptjs cors helmet morgan
```

File structure (recommended)

```
project-root/
├─ package.json
├─ .env
├─ server.js
├─ db.js
```

```
├─ routes/userRoutes.js
├─ controllers/userController.js
├─ services/userService.js
├─ models/User.js
├─ middlewares/errorHandler.js
└─ README.md
```

Flowchart (request lifecycle)

```
flowchart LR
  A[Client (Postman / Browser / Frontend)] -->|HTTP request| B[Express Router (/
api/users)]
  B --> C[Controller]
  C --> D[Service Layer]
  D --> E[Mongoose Model]
  E --> F[(MongoDB)]
  D --> G[Validation & Business Logic]
  C --> H[Response / Error Handler]
```

Sequence explanation: Client → Router receives request → Controller extracts data and calls Service → Service performs business logic and talks to DB via Mongoose Model → DB returns result → Controller sends response or forwards error to middleware.

`.env` example

```
PORT=5000
MONGO_URI=mongodb://localhost:27017/myapp
# or your MongoDB Atlas connection string
SALT_ROUNDS=10
```

`package.json` (minimal)

```
{
  "name": "user-crud-app",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
```

```
    "dev": "nodemon server.js"
  },
  "dependencies": {
    "bcryptjs": "^2.4.3",
    "cors": "^2.8.5",
    "dotenv": "^16.0.0",
    "express": "^4.18.2",
    "helmet": "^6.0.1",
    "mongoose": "^7.0.0",
    "morgan": "^1.10.0"
  }
}
```

`db.js` — MongoDB connection (mongoose)

```
// db.js
const mongoose = require('mongoose');

const connectDB = async () => {
  try {
    const uri = process.env.MONGO_URI;
    if (!uri) throw new Error('MONGO_URI not set in .env');

    await mongoose.connect(uri, {
      useNewUrlParser: true,
      useUnifiedTopology: true
    });

    console.log('MongoDB connected');
  } catch (err) {
    console.error('MongoDB connection error:', err.message);
    process.exit(1);
  }
};

module.exports = connectDB;
```

`models/User.js` — Mongoose model

I kept your field names exactly as you provided (capitalized fields). Fields marked required (`!`) in your schema are set as `required: true` in Mongoose.

```
// models/User.js
const mongoose = require('mongoose');

const IconsSchema = new mongoose.Schema({
  name: { type: String },
  icon: { type: String }
}, { _id: false });

const KycSchema = new mongoose.Schema({
  pastHealthConditions: String,
  pastHealthConditionsDescription: String,
  foodAllergies: String,
  foodAllergiesDescription: String,
  CurrentMedications: String,
  CurrentMedicationsDescription: String,
  ChronicIllnesses: String,
  ChronicIllnessesDescription: String,
  SmokingStatus: String,
  AlcoholConsumption: String,
  heardFrom: String,
  sugar: String,
  bloodPressure: String,
  heartAttack: String,
  pregnacy: String
}, { _id: false });

const UserSchema = new mongoose.Schema({
  FirstName: { type: String, required: true },
  LastName: { type: String, required: true },
  Specialization: { type: String, required: true },
  Gender: { type: String },
  Age: { type: String },
  Status: { type: String },
  Role: { type: String, required: true },
  PhoneNumber: { type: String, required: true },
  EmailId: { type: String, required: true, unique: true, lowercase: true },
  Password: { type: String, required: true },
  countryCode: { type: String, required: true },
  userProfile: { type: String },
  Otp: { type: String },
  ShowInTeam: { type: String },
  TimeSlot: { type: String },
  Description: { type: String },
  SocialIcons: [IconsSchema],
  Address: { type: String },
  KYC: [KycSchema],
  Slug: { type: String, required: true, unique: true },
```

```
  City: { type: String },
  Street: { type: String },
  pinCode: { type: String },
  DOB: { type: String },
  HouseNumber: { type: String },
  MaritalStatus: { type: String },
  qualification: { type: String },
  userType: { type: String },
  resetPWD: { type: String }
}, { timestamps: true });

// Index to speed up lookups
UserSchema.index({ EmailId: 1 });

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

`services/userService.js` — business logic (CRUD)

```
// services/userService.js
const User = require('../models/User');
const bcrypt = require('bcryptjs');

const SALT_ROUNDS = Number(process.env.SALT_ROUNDS) || 10;

async function hashPassword(password) {
  return await bcrypt.hash(password, SALT_ROUNDS);
}

async function createUser(data) {
  // check unique email
  const existing = await User.findOne({ EmailId: data.EmailId });
  if (existing) throw { status: 409, message: 'Email already in use' };

  // generate slug if not provided
  if (!data.Slug) {
    const base = `${data.FirstName || 'user'}-${data.LastName || ''}`.replace(/
\s+/g, '-').toLowerCase();
    data.Slug = `${base}-${Date.now()}`;
  }

  // hash password
  if (data.Password) data.Password = await hashPassword(data.Password);

  const user = new User(data);
```

```
  await user.save();
  return user;
}

async function getUsers(filter = {}, options = {}) {
  // pagination
  const page = parseInt(options.page, 10) || 1;
  const limit = parseInt(options.limit, 10) || 20;
  const skip = (page - 1) * limit;

  const query = { ...filter };

  const [items, total] = await Promise.all([
    User.find(query).skip(skip).limit(limit).sort({ createdAt: -1 }),
    User.countDocuments(query)
  ]);

  return {
    items,
    meta: {
      total,
      page,
      limit,
      pages: Math.ceil(total / limit)
    }
  };
}

async function getUserById(id) {
  const user = await User.findById(id);
  if (!user) throw { status: 404, message: 'User not found' };
  return user;
}

async function updateUser(id, update) {
  // if changing email, ensure uniqueness
  if (update.EmailId) {
    const other = await User.findOne({ EmailId: update.EmailId, _id: { $ne:
id } });
    if (other) throw { status: 409, message: 'Email already in use' };
  }

  // hash password if provided
  if (update.Password) update.Password = await hashPassword(update.Password);

  const updated = await User.findByIdAndUpdate(id, update, { new: true });
  if (!updated) throw { status: 404, message: 'User not found' };
  return updated;
```

```
}

async function deleteUser(id) {
  const deleted = await User.findByIdAndDelete(id);
  if (!deleted) throw { status: 404, message: 'User not found' };
  return deleted;
}

module.exports = {
  createUser,
  getUsers,
  getUserById,
  updateUser,
  deleteUser
};
```

`controllers/userController.js` — HTTP handlers

```
// controllers/userController.js
const userService = require('../services/userService');

async function createUser(req, res, next) {
  try {
    const user = await userService.createUser(req.body);
    // do not send password back in response
    const userObj = user.toObject();
    delete userObj.Password;
    res.status(201).json({ success: true, data: userObj });
  } catch (err) {
    next(err);
  }
}

async function getUsers(req, res, next) {
  try {
    const filter = {};
    // support search by query params (e.g. ?role=Doctor)
    if (req.query.Role) filter.Role = req.query.Role;
    if (req.query.EmailId) filter.EmailId = req.query.EmailId;

    const options = { page: req.query.page, limit: req.query.limit };
    const result = await userService.getUsers(filter, options);
    // remove passwords
    result.items = result.items.map(u => { const o = u.toObject(); delete
```

```
o.Password; return o; });
    res.json({ success: true, ...result });
  } catch (err) {
    next(err);
  }
}

async function getUser(req, res, next) {
  try {
    const user = await userService.getUserById(req.params.id);
    const userObj = user.toObject();
    delete userObj.Password;
    res.json({ success: true, data: userObj });
  } catch (err) {
    next(err);
  }
}

async function updateUser(req, res, next) {
  try {
    const updated = await userService.updateUser(req.params.id, req.body);
    const userObj = updated.toObject();
    delete userObj.Password;
    res.json({ success: true, data: userObj });
  } catch (err) {
    next(err);
  }
}

async function deleteUser(req, res, next) {
  try {
    await userService.deleteUser(req.params.id);
    res.json({ success: true, message: 'User deleted' });
  } catch (err) {
    next(err);
  }
}

module.exports = {
  createUser,
  getUsers,
  getUser,
  updateUser,
  deleteUser
};
```

`routes/userRoutes.js` — Router

```
// routes/userRoutes.js
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');

// POST   /api/users       -> create a new user
// GET    /api/users       -> list users (supports ?page=&limit=&Role=)
// GET    /api/users/:id   -> get user by id
// PUT    /api/users/:id   -> update user
// DELETE /api/users/:id   -> delete user

router.post('/', userController.createUser);
router.get('/', userController.getUsers);
router.get('/:id', userController.getUser);
router.put('/:id', userController.updateUser);
router.delete('/:id', userController.deleteUser);

module.exports = router;
```

`middlewares/errorHandler.js` — centralized error handling

```
// middlewares/errorHandler.js
function errorHandler(err, req, res, next) {
  console.error(err);
  const status = err.status || 500;
  const message = err.message || 'Internal Server Error';
  res.status(status).json({ success: false, message });
}

module.exports = errorHandler;
```

`server.js` — bootstrapping the app

```
// server.js
require('dotenv').config();
const express = require('express');
const helmet = require('helmet');
const cors = require('cors');
```

```
const morgan = require('morgan');
const connectDB = require('./db');
const userRoutes = require('./routes/userRoutes');
const errorHandler = require('./middlewares/errorHandler');

const app = express();

// connect to DB
connectDB();

// middlewares
app.use(helmet());
app.use(cors());
app.use(express.json());
app.use(morgan('dev'));

// routes
app.use('/api/users', userRoutes);

// health
app.get('/', (req, res) => res.send('API running'));

// error handler (last)
app.use(errorHandler);

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`Server listening on port ${PORT}`));
```

Quick testing (curl / Postman)

Create user (required fields shown)

```
curl -X POST http://localhost:5000/api/users
  -H "Content-Type: application/json"
  -d '{
    "FirstName":"John",
    "LastName":"Doe",
    "Specialization":"Cardiology",
    "Role":"Doctor",
    "PhoneNumber":"1234567890",
    "EmailId":"john.doe@example.com",
    "Password":"secret123",
    "countryCode":"+91",
```

```
    "Slug":"john-doe"
  }'
```

List users

```
GET http://localhost:5000/api/users?page=1&limit=10
```

Get single user

```
GET http://localhost:5000/api/users/<userId>
```

Update user

```
PUT http://localhost:5000/api/users/<userId>
Body (JSON): { "City": "Bengaluru" }
```

Delete user

```
DELETE http://localhost:5000/api/users/<userId>
```

Notes & next steps (recommended)

- Add input validation (e.g., `express-validator`) for robust request checks.
- Add authentication (JWT) and restrict some endpoints (e.g., update/delete) to authorized users only.
- Add file uploading for `userProfile` if you plan to store images (use `multer` + cloud storage / S3).
- Add tests (Jest / Supertest) to cover controllers and services.

If you'd like, I can now: - generate `express-validator` validators and wire them up, - add JWT authentication middleware, - produce Dockerfile and a docker-compose that includes a MongoDB service, - or convert these files to ES modules (import/export) instead of CommonJS.

Tell me which follow-up you'd like and I'll add it directly into this project.