module.exports = router;

Phase 6: Testing and Best Practices

1. Testing with Jest and Mongoose

javascript

```javascript
// tests/setup.js
const mongoose = require('mongoose');
const { MongoMemoryServer } = require('mongodb-memory-server');

let mongoServer;

// Setup in-memory MongoDB for testing
beforeAll(async () => {
  mongoServer = await MongoMemoryServer.create();
  const mongoUri = mongoServer.getUri();

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

afterAll(async () => {
  await mongoose.disconnect();
  await mongoServer.stop();
});

// Clean up after each test
afterEach(async () => {
  const collections = mongoose.connection.collections;
  for (const key in collections) {
    await collections[key].deleteMany({});
  }
});

module.exports = { mongoServer };
```

javascript

```
// tests/user.test.js
const AdvancedUser = require('../models/AdvancedUser');

describe('AdvancedUser Model', () => {
  describe('Validation', () => {
    test('should create a valid user', async () => {
      const userData = {
        username: 'testuser',
        email: 'test@example.com',
        age: 25,
        profile: {
          firstName: 'Test',
          lastName: 'User'
        }
      };

      const user = new AdvancedUser(userData);
      const savedUser = await user.save();

      expect(savedUser._id).toBeDefined();
      expect(savedUser.username).toBe(userData.username);
      expect(savedUser.email).toBe(userData.email);
      expect(savedUser.fullName).toBe('Test User');
    });

    test('should fail with invalid email', async () => {
      const userData = {
        username: 'testuser',
        email: 'invalid-email',
        profile: { firstName: 'Test', lastName: 'User' }
      };

      const user = new AdvancedUser(userData);

      await expect(user.save()).rejects.toThrow('Please enter a valid email address');
    });

    test('should fail with duplicate username', async () => {
      const userData = {
        username: 'duplicate',
        email: 'test1@example.com',
        profile: { firstName: 'Test', lastName: 'User' }
      };
```

```
        await AdvancedUser.create(userData);

        const duplicateUser = new AdvancedUser({
            ...userData,
            email: 'test2@example.com'
        });

        await expect(duplicateUser.save()).rejects.toThrow();
    });
});

describe('Methods', () => {
    test('should return public profile', async () => {
        const user = await AdvancedUser.create({
            username: 'testuser',
            email: 'test@example.com',
            profile: { firstName: 'Test', lastName: 'User' },
            interests: ['coding', 'reading']
        });

        const publicProfile = user.getPublicProfile();

        expect(publicProfile).toEqual({
            username: 'testuser',
            fullName: 'Test User',
            interests: ['coding', 'reading'],
            bio: undefined
        });
    });

    test('should find users by role', async () => {
        await AdvancedUser.create([
            { username: 'admin1', email: 'admin@example.com', role: 'admin', profile: { firstName: 'Admin', lastName: 'One
            { username: 'user1', email: 'user1@example.com', role: 'user', profile: { firstName: 'User', lastName: 'One' } },
            { username: 'user2', email: 'user2@example.com', role: 'user', profile: { firstName: 'User', lastName: 'Two' } }
        ]);

        const admins = await AdvancedUser.findByRole('admin');
        const users = await AdvancedUser.findByRole('user');

        expect(admins).toHaveLength(1);
        expect(users).toHaveLength(2);
        expect(admins[0].username).toBe('admin1');
        });
    });
});
```

javascript

```
// tests/post.test.js
const Post = require('../models/Post');
const AdvancedUser = require('../models/AdvancedUser');

describe('Post Model and Operations', () => {
  let user;

  beforeEach(async () => {
    user = await AdvancedUser.create({
      username: 'author',
      email: 'author@example.com',
      profile: { firstName: 'Test', lastName: 'Author' }
    });
  });

  test('should create a post with author', async () => {
    const postData = {
      title: 'Test Post',
      content: 'This is a test post content',
      author: user._id,
      tags: ['test', 'example']
    };

    const post = await Post.create(postData);

    expect(post.title).toBe(postData.title);
    expect(post.author.toString()).toBe(user._id.toString());
    expect(post.tags).toEqual(postData.tags);
  });

  test('should populate author information', async () => {
    const post = await Post.create({
      title: 'Test Post',
      content: 'Content',
      author: user._id
    });

    const populatedPost = await Post.findById(post._id)
      .populate('author', 'username profile');

    expect(populatedPost.author.username).toBe('author');
    expect(populatedPost.author.profile.firstName).toBe('Test');
  });
```

```
test('should handle likes correctly', async () => {
  const post = await Post.create({
    title: 'Test Post',
    content: 'Content',
    author: user._id
  });

  // Add like
  post.likes.push({ user: user._id });
  await post.save();

  expect(post.likes).toHaveLength(1);
  expect(post.likes[0].user.toString()).toBe(user._id.toString());

  // Remove like
  post.likes = post.likes.filter(
    like => like.user.toString() !== user._id.toString()
  );
  await post.save();

  expect(post.likes).toHaveLength(0);
});

test('should aggregate post statistics', async () => {
  // Create multiple posts with likes and comments
  const posts = await Post.create([
    {
      title: 'Post 1',
      content: 'Content 1',
      author: user._id,
      likes: [{ user: user._id }],
      comments: [{ user: user._id, text: 'Great post!' }]
    },
    {
      title: 'Post 2',
      content: 'Content 2',
      author: user._id,
      likes: [{ user: user._id }, { user: user._id }],
      comments: []
    }
  ]);

  const stats = await Post.aggregate([
            { $match: { author: user._id } },
            {
                $group: {
                    _id: '$author',
                    totalPosts: { $sum: 1 },
                    totalLikes: { $sum: { $size: '$likes' } },
                    totalComments: { $sum: { $size: '$comments' } }
                }
            }
        ]);

        expect(stats[0].totalPosts).toBe(2);
        expect(stats[0].totalLikes).toBe(3);
        expect(stats[0].totalComments).toBe(1);
    });
});
```

2. Error Handling Best Practices

javascript

```
// utils/errorHandler.js
class AppError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
    this.isOperational = true;

    Error.captureStackTrace(this, this.constructor);
  }
}

const handleCastErrorDB = (err) => {
  const message = `Invalid ${err.path}: ${err.value}`;
  return new AppError(message, 400);
};

const handleDuplicateFieldsDB = (err) => {
  const value = err.errmsg.match(/(["'])(\\?.)*?\1/)[0];
  const message = `Duplicate field value: ${value}. Please use another value!`;
  return new AppError(message, 400);
};

const handleValidationErrorDB = (err) => {
  const errors = Object.values(err.errors).map(val => val.message);
  const message = `Invalid input data. ${errors.join('. ')}`;
  return new AppError(message, 400);
};

const sendErrorDev = (err, res) => {
  res.status(err.statusCode).json({
    status: err.status,
    error: err,
    message: err.message,
    stack: err.stack
  });
};

const sendErrorProd = (err, res) => {
  // Operational, trusted error: send message to client
  if (err.isOperational) {
    res.status(err.statusCode).json({
      status: err.status,
      message: err.message
    });
  } else {
    // Programming or other unknown error: don't leak error details
    console.error('ERROR 💥', err);
    res.status(500).json({
      status: 'error',
      message: 'Something went wrong!'
    });
  }
};

module.exports = (err, req, res, next) => {
  err.statusCode = err.statusCode || 500;
  err.status = err.status || 'error';

  if (process.env.NODE_ENV === 'development') {
    sendErrorDev(err, res);
  } else {
    let error = { ...err };
    error.message = err.message;

    if (error.name === 'CastError') error = handleCastErrorDB(error);
    if (error.code === 11000) error = handleDuplicateFieldsDB(error);
    if (error.name === 'ValidationError') error = handleValidationErrorDB(error);

    sendErrorProd(error, res);
  }
};

module.exports.AppError = AppError;
```

3. Performance Monitoring

```javascript
```

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

// Query performance monitoring
mongoose.set('debug', (collectionName, method, query, doc) => {
  if (process.env.NODE_ENV === 'development') {
    console.log(`${collectionName}.${method}`, JSON.stringify(query), doc);
  }
});

// Connection monitoring
mongoose.connection.on('connected', () => {
  console.log('✅ Mongoose connected to MongoDB');
});

mongoose.connection.on('error', (err) => {
  console.error('❌ Mongoose connection error:', err);
});

mongoose.connection.on('disconnected', () => {
  console.log('⚠️ Mongoose disconnected');
});

// Query performance wrapper
const withQueryPerformance = (modelMethod) => {
  return async function(...args) {
    const startTime = Date.now();
    const result = await modelMethod.apply(this, args);
    const duration = Date.now() - startTime;

    if (duration > 1000) { // Log slow queries (>1s)
      console.warn(`🐌 Slow query detected: ${duration}ms`);
      console.warn('Query:', this.getQuery());
    }

    return result;
  };
};

module.exports = { withQueryPerformance };
```

4. Data Validation and Sanitization

javascript

```
// utils/validators.js
const validator = require('validator');
const mongoose = require('mongoose');

// Custom validators
const customValidators = {
  // Strong password validator
  strongPassword: {
    validator: function(password) {
      return /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$/.test(password);
    },
    message: 'Password must contain at least 8 characters with uppercase, lowercase, number, and special character'
  },

  // URL validator
  validURL: {
    validator: function(url) {
      return validator.isURL(url, {
        protocols: ['http', 'https'],
        require_protocol: true
      });
    },
    message: 'Please provide a valid URL'
  },

  // Phone number validator
  phoneNumber: {
    validator: function(phone) {
      return validator.isMobilePhone(phone, 'any');
    },
    message: 'Please provide a valid phone number'
  },

  // ObjectId validator
  validObjectId: {
    validator: function(id) {
      return mongoose.Types.ObjectId.isValid(id);
    },
    message: 'Invalid ObjectId format'
  },

  // Array length validator
  arrayLength: (min, max) => ({
        validator: function(array) {
            return array.length >= min && array.length <= max;
        },
        message: `Array must have between ${min} and ${max} items`
    }),

    // Date range validator
    dateRange: (startField, endField) => ({
        validator: function() {
            if (this[startField] && this[endField]) {
                return this[startField] < this[endField];
            }
            return true;
        },
        message: `${endField} must be after ${startField}`
    })
};

// Sanitization utilities
const sanitize = {
    html: (str) => validator.escape(str),
    trim: (str) => str.trim(),
    toLowerCase: (str) => str.toLowerCase(),
    removeSpecialChars: (str) => str.replace(/[^a-zA-Z0-9\s]/g, ''),
    normalizeEmail: (email) => validator.normalizeEmail(email),

    // Sanitize object recursively
    object: (obj) => {
        const sanitized = {};
        for (const [key, value] of Object.entries(obj)) {
            if (typeof value === 'string') {
                sanitized[key] = validator.escape(value.trim());
            } else if (typeof value === 'object' && value !== null) {
                sanitized[key] = sanitize.object(value);
            } else {
                sanitized[key] = value;
            }
        }
        return sanitized;
    }
};

module.exports = { customValidators, sanitize };
```

5. Advanced Schema Patterns

javascript

```
// models/AdvancedBlog.js - Complete blog system
const mongoose = require('mongoose');
const { customValidators, sanitize } = require('../utils/validators');

// Category Schema
const categorySchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    unique: true,
    trim: true,
    maxlength: 50
  },
  slug: {
    type: String,
    required: true,
    unique: true,
    lowercase: true
  },
  description: String,
  color: {
    type: String,
    match: /^#[0-9A-F]{6}$/i,
    default: '#007bff'
  },
  isActive: {
    type: Boolean,
    default: true
  }
});

categorySchema.pre('save', function(next) {
  if (this.isModified('name') && !this.slug) {
    this.slug = this.name.toLowerCase().replace(/\s+/g, '-').replace(/[^a-z0-9-]/g, '');
  }
  next();
});

const Category = mongoose.model('Category', categorySchema);

// Enhanced Post Schema with advanced features
const enhancedPostSchema = new mongoose.Schema({
  title: {
    type: String,
    required: [true, 'Title is required'],
    trim: true,
    minlength: [5, 'Title must be at least 5 characters'],
    maxlength: [200, 'Title cannot exceed 200 characters']
  },

  slug: {
    type: String,
    unique: true,
    lowercase: true,
    index: true
  },

  content: {
    type: String,
    required: [true, 'Content is required'],
    minlength: [50, 'Content must be at least 50 characters']
  },

  excerpt: {
    type: String,
    maxlength: 500
  },

  author: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'AdvancedUser',
    required: true
  },

  category: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Category',
    required: true
  },

  tags: [{
    type: String,
    trim: true,
    lowercase: true,
    maxlength: 30
  }],
// SEO fields
seo: {
  metaTitle: {
    type: String,
    maxlength: 60
  },
  metaDescription: {
    type: String,
    maxlength: 160
  },
  keywords: [String],
  canonicalUrl: {
    type: String,
    validate: customValidators.validURL
  }
},

// Media
featuredImage: {
  url: {
    type: String,
    validate: customValidators.validURL
  },
  alt: String,
  caption: String
},

// Status and publishing
status: {
  type: String,
  enum: ['draft', 'published', 'archived'],
  default: 'draft'
},

publishedAt: Date,

// Interaction data
views: {
  type: Number,
  default: 0,
  min: 0
},

likes: [{
        user: {
            type: mongoose.Schema.Types.ObjectId,
            ref: 'AdvancedUser',
            required: true
        },
        createdAt: {
            type: Date,
            default: Date.now
        }
    }],

    comments: [{
        user: {
            type: mongoose.Schema.Types.ObjectId,
            ref: 'AdvancedUser',
            required: true
        },
        content: {
            type: String,
            required: true,
            maxlength: 1000
        },
        isApproved: {
            type: Boolean,
            default: false
        },
        replies: [{
            user: {
                type: mongoose.Schema.Types.ObjectId,
                ref: 'AdvancedUser',
                required: true
            },
            content: {
                type: String,
                required: true,
                maxlength: 1000
            },
            createdAt: {
                type: Date,
                default: Date.now
            }
        }],
        createdAt: {
            type: Date,
            default: Date.now
        }
    }],

    // Analytics
    analytics: {
        totalReadTime: {
            type: Number,
            default: 0
        },
        bounceRate: {
            type: Number,
            min: 0,
            max: 100,
            default: 0
        },
        shareCount: {
            facebook: { type: Number, default: 0 },
            twitter: { type: Number, default: 0 },
            linkedin: { type: Number, default: 0 }
        }
    },

    // Content flags
    isFeatured: {
        type: Boolean,
        default: false
    },

    isSponsored: {
        type: Boolean,
        default: false
    },

    readingTime: {
        type: Number, // in minutes
        default: 1
    }
}, {
    timestamps: true,
    toJSON: { virtuals: true },
    toObject: { virtuals: true }
});
```

```
// Indexes for performance
enhancedPostSchema.index({ status: 1, publishedAt: -1 });
enhancedPostSchema.index({ author: 1, status: 1 });
enhancedPostSchema.index({ category: 1, status: 1 });
enhancedPostSchema.index({ tags: 1 });
enhancedPostSchema.index({ 'seo.keywords': 1 });
enhancedPostSchema.index({ title: 'text', content: 'text', tags: 'text' });

// Virtual fields
enhancedPostSchema.virtual('likeCount').get(function() {
  return this.likes.length;
});

enhancedPostSchema.virtual('commentCount').get(function() {
  return this.comments.filter(comment => comment.isApproved).length;
});

enhancedPostSchema.virtual('engagementScore').get(function() {
  const likes = this.likes.length;
  const comments = this.commentCount;
  const views = this.views;

  if (views === 0) return 0;
  return ((likes * 2 + comments * 3) / views * 100).toFixed(2);
});

enhancedPostSchema.virtual('isPublished').get(function() {
  return this.status === 'published' && this.publishedAt <= new Date();
});

// Pre-save middleware
enhancedPostSchema.pre('save', function(next) {
  // Generate slug from title
  if (this.isModified('title') && !this.slug) {
    this.slug = this.title
      .toLowerCase()
      .replace(/[^a-z0-9\s-]/g, '')
      .replace(/\s+/g, '-')
      .replace(/-+/g, '-')
      .trim('-');
  }

  // Auto-generate excerpt if not provided
  if (this.isModified('content') && !this.excerpt) {
    this.excerpt = this.content.substring(0, 200).replace(/<[^>]*>/g, '') + '...';
  }

  // Calculate reading time
  if (this.isModified('content')) {
    const wordsPerMinute = 200;
    const wordCount = this.content.split(/\s+/).length;
    this.readingTime = Math.ceil(wordCount / wordsPerMinute);
  }

  // Set published date when publishing
  if (this.isModified('status') && this.status === 'published' && !this.publishedAt) {
    this.publishedAt = new Date();
  }

  // Sanitize content
  if (this.isModified('content')) {
    this.content = sanitize.html(this.content);
  }

  next();
});

// Post-save middleware
enhancedPostSchema.post('save', function(doc) {
  if (doc.status === 'published') {
    console.log(`📝 Post "${doc.title}" has been published`);
  }
});

// Instance methods
enhancedPostSchema.methods.incrementViews = function() {
  this.views += 1;
  return this.save();
};

enhancedPostSchema.methods.addLike = function(userId) {
  const existingLike = this.likes.find(like => like.user.toString() === userId.toString());
  if (!existingLike) {
    this.likes.push({ user: userId });
    return this.save();
  }
  return Promise.resolve(this);
};
```

```
enhancedPostSchema.methods.removeLike = function(userId) {
  this.likes = this.likes.filter(like => like.user.toString() !== userId.toString());
  return this.save();
};

enhancedPostSchema.methods.addComment = function(userId, content) {
  this.comments.push({
    user: userId,
    content: sanitize.html(content),
    isApproved: false // Require moderation
  });
  return this.save();
};

// Static methods
enhancedPostSchema.statics.findPublished = function() {
  return this.find({
    status: 'published',
    publishedAt: { $lte: new Date() }
  }).sort({ publishedAt: -1 });
};

enhancedPostSchema.statics.findByTag = function(tag) {
  return this.find({
    tags: { $in: [tag] },
    status: 'published'
  }).sort({ publishedAt: -1 });
};

enhancedPostSchema.statics.getPopularPosts = function(limit = 10) {
  return this.aggregate([
    { $match: { status: 'published' } },
    {
      $addFields: {
        engagementScore: {
          $add: [
            { $multiply: [{ $size: '$likes' }, 2] },
            { $multiply: [{ $size: '$comments' }, 3] },
            { $divide: ['$views', 100] }
          ]
        }
      }
    },
    { $sort: { engagementScore: -1 } },
    { $limit: limit }
  ]);
};

const EnhancedPost = mongoose.model('EnhancedPost', enhancedPostSchema);

module.exports = { EnhancedPost, Category };
```

Phase 7: Production Deployment Best Practices

1. Environment Configuration

```
javascript
```

```javascript
// config/database.js
const mongoose = require('mongoose');

const connectDB = async () => {
  try {
    const options = {
      useNewUrlParser: true,
      useUnifiedTopology: true,

      // Connection pool settings
      maxPoolSize: parseInt(process.env.DB_MAX_POOL_SIZE) || 10,
      minPoolSize: parseInt(process.env.DB_MIN_POOL_SIZE) || 2,

      // Timeout settings
      serverSelectionTimeoutMS: parseInt(process.env.DB_SERVER_TIMEOUT) || 5000,
      socketTimeoutMS: parseInt(process.env.DB_SOCKET_TIMEOUT) || 45000,

      // Heartbeat settings
      heartbeatFrequencyMS: parseInt(process.env.DB_HEARTBEAT_FREQ) || 10000,

      // Buffer settings
      bufferCommands: process.env.NODE_ENV === 'production' ? false : true,
      bufferMaxEntries: 0,

      // SSL settings for production
      ...(process.env.NODE_ENV === 'production' && {
        ssl: true,
        sslValidate: true,
        sslCA: process.env.DB_SSL_CA
      })
    };

    const conn = await mongoose.connect(process.env.MONGODB_URI, options);

    console.log(`✅ MongoDB Connected: ${conn.connection.host}`);

    // Set up connection event listeners
    mongoose.connection.on('error', (err) => {
      console.error('❌ MongoDB connection error:', err);
    });

    mongoose.connection.on('disconnected', () => {
      console.warn('⚠️ MongoDB disconnected');
    });

    // Graceful shutdown
    process.on('SIGINT', async () => {
      try {
        await mongoose.connection.close();
        console.log('📤 MongoDB connection closed through app termination');
        process.exit(0);
      } catch (error) {
        console.error('Error during graceful shutdown:', error);
        process.exit(1);
      }
    });

    return conn;
  } catch (error) {
    console.error('❌ Database connection failed:', error.message);
    process.exit(1);
  }
};

module.exports = connectDB;
```

2. Security Best Practices

```
javascript
```

```
// middleware/security.js
const rateLimit = require('express-rate-limit');
const helmet = require('helmet');
const mongoSanitize = require('express-mongo-sanitize');
const xss = require('xss-clean');
const hpp = require('hpp');

// Rate limiting
const createRateLimit = (windowMs, max, message) => {
  return rateLimit({
    windowMs,
    max,
    message: { error: message },
    standardHeaders: true,
    legacyHeaders: false,
    handler: (req, res) => {
      res.status(429).json({
        error: 'Too many requests',
        retryAfter: Math.round(windowMs / 1000)
      });
    }
  });
};

const securityMiddleware = {
  // General rate limiting
  general: createRateLimit(15 * 60 * 1000, 100, 'Too many requests from this IP'),

  // Strict rate limiting for auth routes
  auth: createRateLimit(15 * 60 * 1000, 5, 'Too many authentication attempts'),

  // API rate limiting
  api: createRateLimit(15 * 60 * 1000, 1000, 'API rate limit exceeded'),

  // Security headers
  helmet: helmet({
    contentSecurityPolicy: {
      directives: {
        defaultSrc: ["'self'"],
        styleSrc: ["'self'", "'unsafe-inline'"],
        scriptSrc: ["'self'"],
        imgSrc: ["'self'", "data:", "https:"],
      },
        },
        hsts: {
            maxAge: 31536000,
            includeSubDomains: true,
            preload: true
        }
    }),

    // Data sanitization
    mongoSanitize: mongoSanitize(),
    xss: xss(),
    hpp: hpp({
        whitelist: ['sort', 'fields', 'page', 'limit']
    })
};

module.exports = securityMiddleware;
```

3. Logging and Monitoring

```javascript
```

```javascript
// utils/logger.js
const winston = require('winston');
const mongoose = require('mongoose');

// Custom log format
const logFormat = winston.format.combine(
  winston.format.timestamp(),
  winston.format.errors({ stack: true }),
  winston.format.json(),
  winston.format.prettyPrint()
);

// Create logger
const logger = winston.createLogger({
  level: process.env.LOG_LEVEL || 'info',
  format: logFormat,
  defaultMeta: { service: 'mongoose-app' },
  transports: [
    // Error logs
    new winston.transports.File({
      filename: 'logs/error.log',
      level: 'error',
      maxsize: 5242880, // 5MB
      maxFiles: 5
    }),

    // Combined logs
    new winston.transports.File({
      filename: 'logs/combined.log',
      maxsize: 5242880,
      maxFiles: 5
    })
  ]
});

// Console logging for development
if (process.env.NODE_ENV !== 'production') {
  logger.add(new winston.transports.Console({
    format: winston.format.combine(
      winston.format.colorize(),
      winston.format.simple()
    )
  }));
}

// Database operation logging
mongoose.set('debug', function(collectionName, method, query, doc, options) {
  logger.debug('Mongoose Operation', {
    collection: collectionName,
    method,
    query: JSON.stringify(query),
    options: JSON.stringify(options)
  });
});

// Performance monitoring
const performanceLogger = {
  logSlowQuery: (operation, duration, query) => {
    if (duration > 1000) {
      logger.warn('Slow Query Detected', {
        operation,
        duration: `${duration}ms`,
        query: JSON.stringify(query)
      });
    }
  },

  logDatabaseStats: async () => {
    try {
      const stats = await mongoose.connection.db.stats();
      logger.info('Database Statistics', {
        collections: stats.collections,
        dataSize: `${(stats.dataSize / 1024 / 1024).toFixed(2)}MB`,
        indexSize: `${(stats.indexSize / 1024 / 1024).toFixed(2)}MB`,
        connections: mongoose.connection.readyState
      });
    } catch (error) {
      logger.error('Failed to get database stats', error);
    }
  }
};

module.exports = { logger, performanceLogger };
```

Conclusion and Next Steps

This comprehensive roadmap covers everything from basic CRUD operations to advanced production-ready patterns. Here's your learning path:

Beginner (Weeks 1-2)

- Master basic connection and schema creation
- Practice all CRUD operations extensively
- Understand validation and basic middleware

Intermediate (Weeks 3-4)

- Learn population and relationships
- Master aggregation pipelines
- Implement advanced schema features

Advanced (Weeks 5-6)

- Build custom plugins and middleware
- Implement transactions and error handling
- Create performance optimizations

Expert (Weeks 7-8)

- Design scalable architectures
- Implement comprehensive testing
- Master production deployment

Continuous Learning

- Stay updated with Mongoose releases
- Study MongoDB best practices
- Contribute to open-source projects
- Build real-world applications

Recommended Resources

- Official Mongoose Documentation
- MongoDB University courses
- Performance tuning guides
- Security best practices documentation

Remember: The key to mastering Mongoose is consistent practice and building real projects. Start with simple applications and gradually increase complexity as you progress through each phase.# Complete Mongoose Learning Roadmap: Beginner to Expert

Prerequisites

- Basic JavaScript knowledge
- Understanding of Node.js
- Basic MongoDB concepts
- Familiarity with async/await and Promises

Phase 1: Foundation (Beginner Level)

1. Setup and Installation

```bash
# Initialize a new Node.js project
npm init -y

# Install Mongoose
npm install mongoose

# Install additional dependencies for examples
npm install express dotenv
```

2. Basic Connection

javascript

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

// Basic connection
async function connectDB() {
  try {
    await mongoose.connect('mongodb://localhost:27017/myapp', {
      useNewUrlParser: true,
      useUnifiedTopology: true
    });
    console.log('MongoDB Connected Successfully');
  } catch (error) {
    console.error('Connection failed:', error.message);
    process.exit(1);
  }
}

module.exports = connectDB;
```

3. Your First Schema and Model

```
javascript
```

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

// Define Schema
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    trim: true
  },
  email: {
    type: String,
    required: true,
    unique: true,
    lowercase: true
  },
  age: {
    type: Number,
    min: 0,
    max: 120
  },
  createdAt: {
    type: Date,
    default: Date.now
  }
});

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

module.exports = User;
```

Phase 2: CRUD Operations (Intermediate Beginner)

1. Creating Documents

javascript

```javascript
// create-operations.js
const User = require('./models/User');

// Method 1: Using constructor and save()
async function createUserMethod1() {
  try {
    const user = new User({
      name: 'John Doe',
      email: 'john@example.com',
      age: 25
    });

    const savedUser = await user.save();
    console.log('User created:', savedUser);
    return savedUser;
  } catch (error) {
    console.error('Error creating user:', error.message);
  }
}

// Method 2: Using create()
async function createUserMethod2() {
  try {
    const user = await User.create({
      name: 'Jane Smith',
      email: 'jane@example.com',
      age: 30
    });
    console.log('User created:', user);
    return user;
  } catch (error) {
    console.error('Error creating user:', error.message);
  }
}

// Method 3: Creating multiple documents
async function createMultipleUsers() {
  try {
    const users = await User.insertMany([
      { name: 'Alice', email: 'alice@example.com', age: 28 },
      { name: 'Bob', email: 'bob@example.com', age: 32 },
      { name: 'Charlie', email: 'charlie@example.com', age: 24 }
    ]);
    console.log('Multiple users created:', users);
    return users;
  } catch (error) {
    console.error('Error creating multiple users:', error.message);
  }
}
```

2. Reading Documents

```
javascript
```

```
// read-operations.js
const User = require('./models/User');

// Find all documents
async function findAllUsers() {
  try {
    const users = await User.find();
    console.log('All users:', users);
    return users;
  } catch (error) {
    console.error('Error finding users:', error.message);
  }
}

// Find by ID
async function findUserById(userId) {
  try {
    const user = await User.findById(userId);
    if (!user) {
      console.log('User not found');
      return null;
    }
    console.log('User found:', user);
    return user;
  } catch (error) {
    console.error('Error finding user by ID:', error.message);
  }
}

// Find with conditions
async function findUsersWithConditions() {
  try {
    // Find users older than 25
    const olderUsers = await User.find({ age: { $gt: 25 } });

    // Find user by email
    const userByEmail = await User.findOne({ email: 'john@example.com' });

    // Find with multiple conditions
    const specificUsers = await User.find({
      age: { $gte: 20, $lte: 35 },
      name: { $regex: /^J/, $options: 'i' } // Names starting with 'J'
    });
```

```
        console.log('Older users:', olderUsers);
        console.log('User by email:', userByEmail);
        console.log('Specific users:', specificUsers);

        return { olderUsers, userByEmail, specificUsers };
    } catch (error) {
        console.error('Error in conditional queries:', error.message);
    }
}

// Advanced querying
async function advancedQueries() {
    try {
        // Limit and skip (pagination)
        const paginatedUsers = await User.find()
            .limit(5)
            .skip(0)
            .sort({ createdAt: -1 }); // Sort by creation date, newest first

        // Select specific fields
        const usersWithSelectedFields = await User.find()
            .select('name email -_id'); // Include name and email, exclude _id

        // Count documents
        const userCount = await User.countDocuments({ age: { $gte: 18 } });

        console.log('Paginated users:', paginatedUsers);
        console.log('Selected fields:', usersWithSelectedFields);
        console.log('Adult user count:', userCount);

        return { paginatedUsers, usersWithSelectedFields, userCount };
    } catch (error) {
        console.error('Error in advanced queries:', error.message);
    }
}
```

3. Updating Documents

```
javascript
```

```
// update-operations.js
const User = require('./models/User');

// Update one document
async function updateUser(userId, updateData) {
  try {
    // Method 1: findByIdAndUpdate
    const updatedUser = await User.findByIdAndUpdate(
      userId,
      updateData,
      {
        new: true, // Return updated document
        runValidators: true // Run schema validators
      }
    );

    if (!updatedUser) {
      console.log('User not found');
      return null;
    }

    console.log('Updated user:', updatedUser);
    return updatedUser;
  } catch (error) {
    console.error('Error updating user:', error.message);
  }
}

// Update with conditions
async function updateUsersWithConditions() {
  try {
    // Update one document matching condition
    const result1 = await User.updateOne(
      { email: 'john@example.com' },
      { $set: { age: 26 } }
    );

    // Update multiple documents
    const result2 = await User.updateMany(
      { age: { $lt: 25 } },
      { $inc: { age: 1 } } // Increment age by 1
    );
    console.log('Single update result:', result1);
    console.log('Multiple update result:', result2);

    return { result1, result2 };
  } catch (error) {
    console.error('Error in conditional updates:', error.message);
  }
}

// Find and update with custom logic
async function findAndUpdate(email) {
  try {
    const user = await User.findOne({ email });

    if (user) {
      user.age += 1;
      user.name = user.name.toUpperCase();
      const savedUser = await user.save();
      console.log('Updated user with custom logic:', savedUser);
      return savedUser;
    } else {
      console.log('User not found');
      return null;
    }
  } catch (error) {
    console.error('Error in find and update:', error.message);
  }
}
```

4. Deleting Documents

```
javascript
```

```
// delete-operations.js
const User = require('./models/User');

// Delete by ID
async function deleteUserById(userId) {
  try {
    const deletedUser = await User.findByIdAndDelete(userId);

    if (!deletedUser) {
      console.log('User not found');
      return null;
    }

    console.log('Deleted user:', deletedUser);
    return deletedUser;
  } catch (error) {
    console.error('Error deleting user:', error.message);
  }
}

// Delete with conditions
async function deleteUsersWithConditions() {
  try {
    // Delete one document
    const result1 = await User.deleteOne({ email: 'test@example.com' });

    // Delete multiple documents
    const result2 = await User.deleteMany({ age: { $lt: 18 } });

    console.log('Single delete result:', result1);
    console.log('Multiple delete result:', result2);

    return { result1, result2 };
  } catch (error) {
    console.error('Error in conditional deletes:', error.message);
  }
}

// Find and delete
async function findAndDelete(email) {
  try {
    const user = await User.findOneAndDelete({ email });
```

```
        if (user) {
            console.log('Found and deleted user:', user);
            return user;
        } else {
            console.log('User not found');
            return null;
        }
    } catch (error) {
        console.error('Error in find and delete:', error.message);
    }
}
```

Phase 3: Intermediate Level

1. Advanced Schema Features

```
javascript
```

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

const advancedUserSchema = new mongoose.Schema({
  // String validations
  username: {
    type: String,
    required: [true, 'Username is required'],
    unique: true,
    minlength: [3, 'Username must be at least 3 characters'],
    maxlength: [20, 'Username cannot exceed 20 characters'],
    match: [/^[a-zA-Z0-9_]+$/, 'Username can only contain letters, numbers, and underscores']
  },

  // Email with custom validator
  email: {
    type: String,
    required: true,
    unique: true,
    lowercase: true,
    validate: {
      validator: function(email) {
        return /^\w+([.-]?\w+)*@\w+([.-]?\w+)*(\.\w{2,3})+$/.test(email);
      },
      message: 'Please enter a valid email address'
    }
  },

  // Number validations
  age: {
    type: Number,
    min: [0, 'Age cannot be negative'],
    max: [120, 'Age cannot exceed 120'],
    validate: {
      validator: Number.isInteger,
      message: 'Age must be an integer'
    }
  },

  // Enum field
  role: {
    type: String,
    enum: {
            values: ['user', 'admin', 'moderator'],
            message: 'Role must be either user, admin, or moderator'
        },
        default: 'user'
    },

    // Array of strings
    interests: [{
        type: String,
        trim: true
    }],

    // Nested object
    profile: {
        firstName: {
            type: String,
            required: true,
            trim: true
        },
        lastName: {
            type: String,
            required: true,
            trim: true
        },
        bio: String,
        avatar: String
    },

    // Reference to another model
    posts: [{
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Post'
    }],

    // Mixed type
    metadata: mongoose.Schema.Types.Mixed,

    // Timestamps
    isActive: {
        type: Boolean,
        default: true
    }
}, {
    timestamps: true, // Adds createdAt and updatedAt
```

```
  versionKey: false // Removes __v field
});

// Virtual field (not stored in database)
advancedUserSchema.virtual('fullName').get(function() {
  return `${this.profile.firstName} ${this.profile.lastName}`;
});

// Virtual populate
advancedUserSchema.virtual('postCount', {
  ref: 'Post',
  localField: '_id',
  foreignField: 'author',
  count: true
});

// Pre-save middleware
advancedUserSchema.pre('save', function(next) {
  if (this.isModified('email')) {
    this.email = this.email.toLowerCase();
  }
  next();
});

// Post-save middleware
advancedUserSchema.post('save', function(doc) {
  console.log(`User ${doc.username} has been saved`);
});

// Instance method
advancedUserSchema.methods.getPublicProfile = function() {
  return {
    username: this.username,
    fullName: this.fullName,
    interests: this.interests,
    bio: this.profile.bio
  };
};

// Static method
advancedUserSchema.statics.findByRole = function(role) {
  return this.find({ role });
};
```

```
const AdvancedUser = mongoose.model('AdvancedUser', advancedUserSchema);
module.exports = AdvancedUser;
```

2. Relationships and Population

```javascript
```

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

const postSchema = new mongoose.Schema({
  title: {
    type: String,
    required: true,
    trim: true
  },
  content: {
    type: String,
    required: true
  },
  author: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'AdvancedUser',
    required: true
  },
  tags: [{
    type: String,
    trim: true
  }],
  likes: [{
    user: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'AdvancedUser'
    },
    createdAt: {
      type: Date,
      default: Date.now
    }
  }],
  comments: [{
    user: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'AdvancedUser',
      required: true
    },
    text: {
      type: String,
      required: true
    },
    createdAt: {
            type: Date,
            default: Date.now
        }
    }]
}, {
    timestamps: true
});

const Post = mongoose.model('Post', postSchema);
module.exports = Post;

// Using relationships
const AdvancedUser = require('./AdvancedUser');

async function createPostWithUser() {
    try {
        // Create a user first
        const user = await AdvancedUser.create({
            username: 'johndoe',
            email: 'john@example.com',
            age: 25,
            profile: {
                firstName: 'John',
                lastName: 'Doe',
                bio: 'Software developer'
            },
            interests: ['programming', 'reading']
        });

        // Create a post
        const post = await Post.create({
            title: 'My First Post',
            content: 'This is my first blog post!',
            author: user._id,
            tags: ['intro', 'blog']
        });

        console.log('Created post:', post);
        return { user, post };
    } catch (error) {
        console.error('Error:', error.message);
    }
}
```

```
// Population examples
async function populationExamples() {
  try {
    // Basic population
    const posts = await Post.find().populate('author');

    // Population with field selection
    const postsWithSelectedFields = await Post.find()
      .populate('author', 'username profile.firstName profile.lastName');

    // Nested population
    const postsWithComments = await Post.find()
      .populate({
        path: 'comments.user',
        select: 'username profile.firstName'
      })
      .populate('author', 'username');

    // Multiple populations
    const fullPosts = await Post.find()
      .populate('author', 'username profile')
      .populate('likes.user', 'username')
      .populate('comments.user', 'username');

    console.log('Posts with basic population:', posts);
    console.log('Posts with selected fields:', postsWithSelectedFields);
    console.log('Posts with nested population:', postsWithComments);

    return { posts, postsWithSelectedFields, postsWithComments, fullPosts };
  } catch (error) {
    console.error('Error in population:', error.message);
  }
}
```

Phase 4: Advanced Level

1. Aggregation Pipeline

```
javascript
```

```
// aggregation-examples.js
const Post = require('./models/Post');
const AdvancedUser = require('./models/AdvancedUser');

async function aggregationExamples() {
  try {
    // Basic aggregation - count posts by author
    const postCountsByAuthor = await Post.aggregate([
      {
        $group: {
          _id: '$author',
          postCount: { $sum: 1 },
          totalLikes: { $sum: { $size: '$likes' } }
        }
      },
      {
        $lookup: {
          from: 'advancedusers',
          localField: '_id',
          foreignField: '_id',
          as: 'author'
        }
      },
      {
        $unwind: '$author'
      },
      {
        $project: {
          _id: 0,
          username: '$author.username',
          fullName: {
            $concat: ['$author.profile.firstName', ' ', '$author.profile.lastName']
          },
          postCount: 1,
          totalLikes: 1,
          avgLikesPerPost: { $divide: ['$totalLikes', '$postCount'] }
        }
      },
      {
        $sort: { postCount: -1 }
      }
    ]);
```

```
// Advanced aggregation - posts with engagement metrics
const postsWithMetrics = await Post.aggregate([
  {
    $addFields: {
      likeCount: { $size: '$likes' },
      commentCount: { $size: '$comments' },
      engagementScore: {
        $add: [
          { $size: '$likes' },
          { $multiply: [{ $size: '$comments' }, 2] }
        ]
      }
    }
  },
  {
    $match: {
      engagementScore: { $gte: 5 }
    }
  },
  {
    $lookup: {
      from: 'advancedusers',
      localField: 'author',
      foreignField: '_id',
      as: 'authorInfo'
    }
  },
  {
    $unwind: '$authorInfo'
  },
  {
    $project: {
      title: 1,
      content: { $substr: ['$content', 0, 100] },
      author: '$authorInfo.username',
      likeCount: 1,
      commentCount: 1,
      engagementScore: 1,
      createdAt: 1
    }
  },
  {
    $sort: { engagementScore: -1 }
  },
        {
          $limit: 10
        }
      ]);

      console.log('Post counts by author:', postCountsByAuthor);
      console.log('High engagement posts:', postsWithMetrics);

      return { postCountsByAuthor, postsWithMetrics };
    } catch (error) {
      console.error('Error in aggregation:', error.message);
    }
}

// Time-based aggregation
async function timeBasedAggregation() {
    try {
      const monthlyStats = await Post.aggregate([
        {
          $group: {
            _id: {
              year: { $year: '$createdAt' },
              month: { $month: '$createdAt' }
            },
            postCount: { $sum: 1 },
            totalLikes: { $sum: { $size: '$likes' } },
            avgLikes: { $avg: { $size: '$likes' } },
            posts: { $push: { title: '$title', likes: { $size: '$likes' } } }
          }
        },
        {
          $sort: { '_id.year': -1, '_id.month': -1 }
        },
        {
          $project: {
            _id: 0,
            period: {
              $dateFromParts: {
                year: '$_id.year',
                month: '$_id.month'
              }
            },
            postCount: 1,
            totalLikes: 1,
            avgLikes: { $round: ['$avgLikes', 2] },
            topPost: {
              $arrayElemAt: [
                {
                  $sortArray: {
                    input: '$posts',
                    sortBy: { likes: -1 }
                  }
                },
                0
              ]
            }
          }
        }
      ]);

      console.log('Monthly statistics:', monthlyStats);
      return monthlyStats;
    } catch (error) {
      console.error('Error in time-based aggregation:', error.message);
    }
  }
```

2. Transactions

javascript

```
// transactions.js
const mongoose = require('mongoose');
const AdvancedUser = require('./models/AdvancedUser');
const Post = require('./models/Post');

async function createUserAndPostTransaction() {
  const session = await mongoose.startSession();

  try {
    await session.withTransaction(async () => {
      // Create user
      const user = new AdvancedUser({
        username: 'transactionuser',
        email: 'transaction@example.com',
        age: 28,
        profile: {
          firstName: 'Transaction',
          lastName: 'User'
        }
      });

      await user.save({ session });

      // Create post
      const post = new Post({
        title: 'Transaction Post',
        content: 'This post was created in a transaction',
        author: user._id,
        tags: ['transaction', 'example']
      });

      await post.save({ session });

      // Update user with post reference
      user.posts.push(post._id);
      await user.save({ session });

      console.log('Transaction completed successfully');
      return { user, post };
    });
  } catch (error) {
    console.error('Transaction failed:', error.message);
    throw error;
    } finally {
        await session.endSession();
    }
}

// Complex transaction with error handling
async function transferLikesTransaction(fromPostId, toPostId, userId) {
    const session = await mongoose.startSession();

    try {
        const result = await session.withTransaction(async () => {
            // Find posts
            const fromPost = await Post.findById(fromPostId).session(session);
            const toPost = await Post.findById(toPostId).session(session);

            if (!fromPost || !toPost) {
                throw new Error('One or both posts not found');
            }

            // Check if user liked the from post
            const likeIndex = fromPost.likes.findIndex(
                like => like.user.toString() === userId.toString()
            );

            if (likeIndex === -1) {
                throw new Error('User has not liked the source post');
            }

            // Remove like from source post
            fromPost.likes.splice(likeIndex, 1);
            await fromPost.save({ session });

            // Add like to destination post
            const existingLike = toPost.likes.find(
                like => like.user.toString() === userId.toString()
            );

            if (!existingLike) {
                toPost.likes.push({ user: userId });
                await toPost.save({ session });
            }

            return { fromPost, toPost };
        });
        console.log('Like transfer completed');
        return result;
    } catch (error) {
        console.error('Like transfer failed:', error.message);
        throw error;
    } finally {
        await session.endSession();
    }
}
```

3. Performance Optimization

```
javascript
```

```
// optimization.js
const AdvancedUser = require('./models/AdvancedUser');
const Post = require('./models/Post');

// Indexing examples
async function createIndexes() {
  try {
    // Single field index
    await AdvancedUser.collection.createIndex({ email: 1 });

    // Compound index
    await Post.collection.createIndex({ author: 1, createdAt: -1 });

    // Text index for search
    await Post.collection.createIndex({
      title: 'text',
      content: 'text',
      tags: 'text'
    });

    // Sparse index (only for documents that have the field)
    await AdvancedUser.collection.createIndex(
      { 'profile.bio': 1 },
      { sparse: true }
    );

    console.log('Indexes created successfully');
  } catch (error) {
    console.error('Error creating indexes:', error.message);
  }
}

// Efficient queries
async function efficientQueries() {
  try {
    // Use projection to limit fields
    const users = await AdvancedUser.find({ role: 'user' })
      .select('username email profile.firstName profile.lastName')
      .lean(); // Returns plain JavaScript objects (faster)

    // Use indexes effectively
    const recentPosts = await Post.find({
      createdAt: { $gte: new Date(Date.now() - 7 * 24 * 60 * 60 * 1000) }
    })
    .sort({ createdAt: -1 })
    .limit(20)
    .populate('author', 'username')
    .lean();

    // Text search
    const searchResults = await Post.find(
      { $text: { $search: 'javascript programming' } },
      { score: { $meta: 'textScore' } }
    )
    .sort({ score: { $meta: 'textScore' } })
    .limit(10);

    console.log('Efficient queries completed');
    return { users, recentPosts, searchResults };
  } catch (error) {
    console.error('Error in efficient queries:', error.message);
  }
}

// Batch operations
async function batchOperations() {
  try {
    // Bulk write operations
    const bulkOps = [
      {
        updateOne: {
          filter: { username: 'user1' },
          update: { $inc: { age: 1 } }
        }
      },
      {
        updateMany: {
          filter: { role: 'user' },
          update: { $set: { isActive: true } }
        }
      },
      {
        deleteOne: {
          filter: { email: 'tobedeleted@example.com' }
        }
      }
    ];
```

```
        const result = await AdvancedUser.bulkWrite(bulkOps);
        console.log('Bulk operations result:', result);

        return result;
    } catch (error) {
        console.error('Error in batch operations:', error.message);
    }
}
```

Phase 5: Expert Level

1. Custom Plugins

javascript

```
// plugins/timestampPlugin.js
function timestampPlugin(schema, options) {
  const { paths = ['createdAt', 'updatedAt'], index = false } = options || {};

  // Add timestamp fields
  if (paths.includes('createdAt')) {
    schema.add({
      createdAt: {
        type: Date,
        default: Date.now,
        immutable: true
      }
    });
  }

  if (paths.includes('updatedAt')) {
    schema.add({
      updatedAt: {
        type: Date,
        default: Date.now
      }
    });
  }

  // Pre-save middleware to update 'updatedAt'
  schema.pre('save', function(next) {
    if (paths.includes('updatedAt') && this.isModified() && !this.isNew) {
      this.updatedAt = new Date();
    }
    next();
  });

  // Pre-update middleware
  schema.pre(['updateOne', 'updateMany', 'findOneAndUpdate'], function() {
    if (paths.includes('updatedAt')) {
      this.set({ updatedAt: new Date() });
    }
  });

  // Create indexes if requested
  if (index) {
    if (paths.includes('createdAt')) {
      schema.index({ createdAt: 1 });
    }
    if (paths.includes('updatedAt')) {
      schema.index({ updatedAt: 1 });
    }
  }
}

module.exports = timestampPlugin;
```

2. Advanced Middleware

```
javascript
```

```
// middleware/auditPlugin.js
function auditPlugin(schema, options) {
  const auditSchema = {
    auditLog: [{
      action: {
        type: String,
        enum: ['create', 'update', 'delete'],
        required: true
      },
      changes: mongoose.Schema.Types.Mixed,
      user: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'AdvancedUser'
      },
      timestamp: {
        type: Date,
        default: Date.now
      },
      ip: String,
      userAgent: String
    }]
  };

  schema.add(auditSchema);

  // Pre-save middleware for auditing
  schema.pre('save', function(next) {
    const doc = this;
    const isNew = doc.isNew;
    const modifiedPaths = doc.modifiedPaths();

    if (isNew) {
      doc.auditLog.push({
        action: 'create',
        changes: doc.toObject(),
        user: doc.$locals.currentUser,
        ip: doc.$locals.ip,
        userAgent: doc.$locals.userAgent
      });
    } else if (modifiedPaths.length > 0) {
      const changes = {};
      modifiedPaths.forEach(path => {
        if (path !== 'auditLog') {
          changes[path] = {
            old: doc.$locals.original?.[path],
            new: doc[path]
          };
        }
      });

      doc.auditLog.push({
        action: 'update',
        changes,
        user: doc.$locals.currentUser,
        ip: doc.$locals.ip,
        userAgent: doc.$locals.userAgent
      });
    }

    next();
  });
}

module.exports = auditPlugin;
```

3. Complete Application Example

javascript

```
// app.js - Complete Express + Mongoose Application
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
require('dotenv').config();

// Import models
const AdvancedUser = require('./models/AdvancedUser');
const Post = require('./models/Post');

// Import plugins
const timestampPlugin = require('./plugins/timestampPlugin');
const auditPlugin = require('./middleware/auditPlugin');

// Apply plugins globally
mongoose.plugin(timestampPlugin, { index: true });
mongoose.plugin(auditPlugin);

const app = express();

// Middleware
app.use(cors());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// Database connection with advanced options
async function connectDB() {
  try {
    await mongoose.connect(process.env.MONGODB_URI || 'mongodb://localhost:27017/advanced_blog', {
      useNewUrlParser: true,
      useUnifiedTopology: true,
      maxPoolSize: 10,
      serverSelectionTimeoutMS: 5000,
      socketTimeoutMS: 45000,
      bufferCommands: false,
      bufferMaxEntries: 0
    });
    console.log('✅ MongoDB Connected');
  } catch (error) {
    console.error('❌ MongoDB connection failed:', error.message);
    process.exit(1);
  }
}
```

```
// Error handling middleware
app.use((error, req, res, next) => {
  if (error.name === 'ValidationError') {
    const errors = Object.values(error.errors).map(err => err.message);
    return res.status(400).json({ error: 'Validation Error', details: errors });
  }

  if (error.code === 11000) {
    const field = Object.keys(error.keyValue)[0];
    return res.status(400).json({ error: `${field} already exists` });
  }

  console.error(error);
  res.status(500).json({ error: 'Internal Server Error' });
});

// Routes
app.use('/api/users', require('./routes/users'));
app.use('/api/posts', require('./routes/posts'));

// Start server
const PORT = process.env.PORT || 3000;
app.listen(PORT, async () => {
  await connectDB();
  console.log(`🚀 Server running on port ${PORT}`);
});

module.exports = app;

// routes/users.js - Complete User Routes
const express = require('express');
const router = express.Router();
const AdvancedUser = require('../models/AdvancedUser');

// GET /api/users - Get all users with filtering and pagination
router.get('/', async (req, res, next) => {
  try {
    const {
      page = 1,
      limit = 10,
      role,
      search,
      sortBy = 'createdAt',
      sortOrder = 'desc'
    } = req.query;

    // Build query
    const query = {};
    if (role) query.role = role;
    if (search) {
      query.$or = [
        { username: { $regex: search, $options: 'i' } },
        { 'profile.firstName': { $regex: search, $options: 'i' } },
        { 'profile.lastName': { $regex: search, $options: 'i' } }
      ];
    }

    // Execute query with pagination
    const skip = (parseInt(page) - 1) * parseInt(limit);
    const sortOptions = { [sortBy]: sortOrder === 'desc' ? -1 : 1 };

    const [users, total] = await Promise.all([
      AdvancedUser.find(query)
        .select('-auditLog')
        .sort(sortOptions)
        .skip(skip)
        .limit(parseInt(limit))
        .populate('posts', 'title createdAt')
        .lean(),
      AdvancedUser.countDocuments(query)
    ]);

    res.json({
      users,
      pagination: {
        page: parseInt(page),
        limit: parseInt(limit),
        total,
        pages: Math.ceil(total / parseInt(limit))
      }
    });
  } catch (error) {
    next(error);
  }
});

// GET /api/users/:id - Get user by ID
```

```
router.get('/:id', async (req, res, next) => {
    try {
        const user = await AdvancedUser.findById(req.params.id)
            .populate('posts')
            .populate('postCount');

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

        res.json(user);
    } catch (error) {
        next(error);
    }
});

// POST /api/users - Create new user
router.post('/', async (req, res, next) => {
    try {
        const user = new AdvancedUser(req.body);

        // Add audit context
        user.$locals = {
            currentUser: req.user?.id,
            ip: req.ip,
            userAgent: req.get('User-Agent')
        };

        await user.save();

        res.status(201).json(user.getPublicProfile());
    } catch (error) {
        next(error);
    }
});

// PUT /api/users/:id - Update user
router.put('/:id', async (req, res, next) => {
    try {
        const user = await AdvancedUser.findById(req.params.id);

        if (!user) {
            return res.status(404).json({ error: 'User not found' });
        }
        // Store original values for audit
        user.$locals = {
            original: user.toObject(),
            currentUser: req.user?.id,
            ip: req.ip,
            userAgent: req.get('User-Agent')
        };

        // Update fields
        Object.keys(req.body).forEach(key => {
            if (key !== 'auditLog') {
                user[key] = req.body[key];
            }
        });

        await user.save();
        res.json(user);
    } catch (error) {
        next(error);
    }
});

// DELETE /api/users/:id - Delete user
router.delete('/:id', async (req, res, next) => {
    try {
        const user = await AdvancedUser.findByIdAndDelete(req.params.id);

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

        res.json({ message: 'User deleted successfully' });
    } catch (error) {
        next(error);
    }
});

// GET /api/users/:id/stats - Get user statistics
router.get('/:id/stats', async (req, res, next) => {
    try {
        const stats = await AdvancedUser.aggregate([
            { $match: { _id: new mongoose.Types.ObjectId(req.params.id) } },
            {
```

```
        $lookup: {
            from: 'posts',
            localField: '_id',
            foreignField: 'author',
            as: 'userPosts'
        }
    },
    {
        $project: {
            username: 1,
            totalPosts: { $size: '$userPosts' },
            totalLikes: {
                $sum: {
                    $map: {
                        input: '$userPosts',
                        as: 'post',
                        in: { $size: '$post.likes' }
                    }
                }
            },
            totalComments: {
                $sum: {
                    $map: {
                        input: '$userPosts',
                        as: 'post',
                        in: { $size: '$post.comments' }
                    }
                }
            },
            avgLikesPerPost: {
                $cond: {
                    if: { $gt: [{ $size: '$userPosts' }, 0] },
                    then: {
                        $divide: [
                            {
                                $sum: {
                                    $map: {
                                        input: '$userPosts',
                                        as: 'post',
                                        in: { $size: '$post.likes' }
                                    }
                                }
                            },
                            { $size: '$userPosts' }
                            ]
                        },
                        else: 0
                    }
                }
            }
        }
    ]);

    if (stats.length === 0) {
        return res.status(404).json({ error: 'User not found' });
    }

    res.json(stats[0]);
  } catch (error) {
    next(error);
  }
});

module.exports = router;

// routes/posts.js - Complete Post Routes
const express = require('express');
const router = express.Router();
const Post = require('../models/Post');
const AdvancedUser = require('../models/AdvancedUser');

// GET /api/posts - Get all posts with advanced filtering
router.get('/', async (req, res, next) => {
  try {
    const {
      page = 1,
      limit = 10,
      author,
      tags,
      search,
      sortBy = 'createdAt',
      sortOrder = 'desc',
      minLikes,
      dateFrom,
      dateTo
    } = req.query;

    // Build aggregation pipeline
```

```
const pipeline = [];

// Match stage
const matchConditions = {};
if (author) matchConditions.author = new mongoose.Types.ObjectId(author);
if (tags) matchConditions.tags = { $in: tags.split(',') };
if (minLikes) matchConditions.$expr = { $gte: [{ $size: '$likes' }, parseInt(minLikes)] };
if (dateFrom || dateTo) {
  matchConditions.createdAt = {};
  if (dateFrom) matchConditions.createdAt.$gte = new Date(dateFrom);
  if (dateTo) matchConditions.createdAt.$lte = new Date(dateTo);
}
if (search) {
  matchConditions.$or = [
    { title: { $regex: search, $options: 'i' } },
    { content: { $regex: search, $options: 'i' } },
    { tags: { $elemMatch: { $regex: search, $options: 'i' } } }
  ];
}

if (Object.keys(matchConditions).length > 0) {
  pipeline.push({ $match: matchConditions });
}

// Add computed fields
pipeline.push({
  $addFields: {
    likeCount: { $size: '$likes' },
    commentCount: { $size: '$comments' },
    engagementScore: {
      $add: [
        { $size: '$likes' },
        { $multiply: [{ $size: '$comments' }, 2] }
      ]
    }
  }
});

// Lookup author
pipeline.push({
  $lookup: {
    from: 'advancedusers',
    localField: 'author',
    foreignField: '_id',
    as: 'authorInfo'
  }
});

pipeline.push({ $unwind: '$authorInfo' });

// Sort
const sortStage = {};
sortStage[sortBy] = sortOrder === 'desc' ? -1 : 1;
pipeline.push({ $sort: sortStage });

// Facet for pagination
pipeline.push({
  $facet: {
    posts: [
      { $skip: (parseInt(page) - 1) * parseInt(limit) },
      { $limit: parseInt(limit) },
      {
        $project: {
          title: 1,
          content: { $substr: ['$content', 0, 200] },
          tags: 1,
          likeCount: 1,
          commentCount: 1,
          engagementScore: 1,
          createdAt: 1,
          updatedAt: 1,
          author: {
            _id: '$authorInfo._id',
            username: '$authorInfo.username',
            fullName: {
              $concat: [
                '$authorInfo.profile.firstName',
                ' ',
                '$authorInfo.profile.lastName'
              ]
            }
          }
        }
      }
    ],
    totalCount: [{ $count: 'count' }]
  }
});
```

```
        const result = await Post.aggregate(pipeline);
        const posts = result[0].posts;
        const total = result[0].totalCount[0]?.count || 0;

        res.json({
            posts,
            pagination: {
                page: parseInt(page),
                limit: parseInt(limit),
                total,
                pages: Math.ceil(total / parseInt(limit))
            }
        });
    } catch (error) {
        next(error);
    }
});

// GET /api/posts/:id - Get post by ID with full details
router.get('/:id', async (req, res, next) => {
    try {
        const post = await Post.findById(req.params.id)
            .populate('author', 'username profile')
            .populate('comments.user', 'username profile.firstName profile.lastName')
            .populate('likes.user', 'username');

        if (!post) {
            return res.status(404).json({ error: 'Post not found' });
        }

        res.json(post);
    } catch (error) {
        next(error);
    }
});

// POST /api/posts - Create new post
router.post('/', async (req, res, next) => {
    try {
        const { title, content, tags, authorId } = req.body;

        // Verify author exists
        const author = await AdvancedUser.findById(authorId);
        if (!author) {
            return res.status(400).json({ error: 'Author not found' });
        }

        const post = new Post({
            title,
            content,
            author: authorId,
            tags: tags || []
        });

        await post.save();

        // Add post reference to user
        author.posts.push(post._id);
        await author.save();

        await post.populate('author', 'username profile');
        res.status(201).json(post);
    } catch (error) {
        next(error);
    }
});

// PUT /api/posts/:id - Update post
router.put('/:id', async (req, res, next) => {
    try {
        const { title, content, tags } = req.body;

        const post = await Post.findByIdAndUpdate(
            req.params.id,
            { title, content, tags },
            { new: true, runValidators: true }
        ).populate('author', 'username profile');

        if (!post) {
            return res.status(404).json({ error: 'Post not found' });
        }

        res.json(post);
    } catch (error) {
        next(error);
    }
});
```

```
// DELETE /api/posts/:id - Delete post
router.delete('/:id', async (req, res, next) => {
  const session = await mongoose.startSession();

  try {
    await session.withTransaction(async () => {
      const post = await Post.findByIdAndDelete(req.params.id).session(session);

      if (!post) {
        throw new Error('Post not found');
      }

      // Remove post reference from user
      await AdvancedUser.updateOne(
        { _id: post.author },
        { $pull: { posts: post._id } }
      ).session(session);
    });

    res.json({ message: 'Post deleted successfully' });
  } catch (error) {
    if (error.message === 'Post not found') {
      return res.status(404).json({ error: 'Post not found' });
    }
    next(error);
  } finally {
    await session.endSession();
  }
});

// POST /api/posts/:id/like - Like/unlike a post
router.post('/:id/like', async (req, res, next) => {
  try {
    const { userId } = req.body;
    const post = await Post.findById(req.params.id);

    if (!post) {
      return res.status(404).json({ error: 'Post not found' });
    }

    const existingLikeIndex = post.likes.findIndex(
      like => like.user.toString() === userId
    );
```

```
        if (existingLikeIndex > -1) {
            // Unlike
            post.likes.splice(existingLikeIndex, 1);
        } else {
            // Like
            post.likes.push({ user: userId });
        }

        await post.save();
        await post.populate('likes.user', 'username');

        res.json({
            liked: existingLikeIndex === -1,
            likeCount: post.likes.length,
            likes: post.likes
        });
    } catch (error) {
        next(error);
    }
});

// POST /api/posts/:id/comments - Add comment to post
router.post('/:id/comments', async (req, res, next) => {
    try {
        const { userId, text } = req.body;
        const post = await Post.findById(req.params.id);

        if (!post) {
            return res.status(404).json({ error: 'Post not found' });
        }

        post.comments.push({
            user: userId,
            text,
            createdAt: new Date()
        });

        await post.save();
        await post.populate('comments.user', 'username profile.firstName profile.lastName');

        res.status(201).json(post.comments[post.comments.length - 1]);
    } catch (error) {
        next(error);
```

```
  }
});

// GET /api/posts/analytics/engagement - Get engagement analytics
router.get('/analytics/engagement', async (req, res, next) => {
  try {
    const analytics = await Post.aggregate([
      {
        $addFields: {
          likeCount: { $size: '$likes' },
          commentCount: { $size: '$comments' },
          engagementScore: {
            $add: [
              { $size: '$likes' },
              { $multiply: [{ $size: '$comments' }, 2] }
            ]
          }
        }
      },
      {
        $group: {
          _id: null,
          totalPosts: { $sum: 1 },
          avgLikes: { $avg: '$likeCount' },
          avgComments: { $avg: '$commentCount' },
          avgEngagement: { $avg: '$engagementScore' },
          maxEngagement: { $max: '$engagementScore' },
          topPosts: {
            $push: {
              $cond: {
                if: { $gte: ['$engagementScore', 10] },
                then: {
                  _id: '$_id',
                  title: '$title',
                  engagementScore: '$engagementScore'
                },
                else: '$REMOVE'
              }
            }
          }
        }
      },
      {
        $project: {
          _id: 0,
          totalPosts: 1,
          avgLikes: { $round: ['$avgLikes', 2] },
          avgComments: { $round: ['$avgComments', 2] },
          avgEngagement: { $round: ['$avgEngagement', 2] },
          maxEngagement: 1,
          topPosts: { $slice: ['$topPosts', 10] }
        }
      }
    ]);

    res.json(analytics[0] || {});
  } catch (error) {
    next(error);
  }
});

module.exports = router;
```