Complete Node.js and Express.js Backend Development Guide - ES6 Module Edition

Table of Contents

Introduction

ES6 modules provide a standardized way to organize and share JavaScript code. Unlike CommonJS (require/module.exports), ES6 modules use `import` and `export` statements, offering better static analysis, tree shaking, and cleaner syntax.

Benefits of ES6 Modules:

- **Static Analysis**: Imports/exports are determined at compile time
- **Tree Shaking**: Dead code elimination in bundlers
- **Cleaner Syntax**: More readable import/export statements
- **Standardized**: Part of the JavaScript specification
- **Better IDE Support**: Enhanced autocomplete and refactoring

Environment Setup

Package.json Configuration

```
{
  "name": "express-es6-backend",
  "version": "1.0.0",
  "type": "module",
  "description": "Express.js backend with ES6 modules",
  "main": "src/app.js",
  "scripts": {
    "start": "node src/app.js",
    "dev": "nodemon src/app.js",
    "test": "jest --experimental-vm-modules"
  },
  "dependencies": {
    "express": "^4.18.2",
    "mongoose": "^7.5.0",
    "bcryptjs": "^2.4.3",
    "jsonwebtoken": "^9.0.2",
    "dotenv": "^16.3.1",
    "helmet": "^7.0.0",
    "cors": "^2.8.5",
    "joi": "^17.9.2",
    "redis": "^4.6.7"
  },
  "devDependencies": {
    "nodemon": "^3.0.1",
    "jest": "^29.6.2",
    "supertest": "^6.3.3"
  }
}
```

Environment Variables (.env)

```
NODE_ENV=development
PORT=3000
MONGODB_URI=mongodb://localhost:27017/expressdb
JWT_SECRET=your-super-secret-jwt-key
JWT_EXPIRE=7d
REDIS_URL=redis://localhost:6379
```

Basic Express Server

App Configuration (src/app.js)

```
import express from 'express';
import cors from 'cors';
import helmet from 'helmet';
import dotenv from 'dotenv';
import { connectDB } from './config/database.js';
import { globalErrorHandler } from './middleware/errorHandler.js';
```

```
import { notFound } from './middleware/notFound.js';
import authRoutes from './routes/auth.js';
import userRoutes from './routes/users.js';
import productRoutes from './routes/products.js';

// Load environment variables
dotenv.config();

const app = express();
const PORT = process.env.PORT || 3000;

// Connect to database
connectDB();

// Security middleware
app.use(helmet());
app.use(cors({
    origin: process.env.CORS_ORIGIN || 'http://localhost:3000',
    credentials: true
}));

// Body parsing middleware
app.use(express.json({ limit: '10mb' }));
app.use(express.urlencoded({ extended: true }));

// Routes
app.use('/api/auth', authRoutes);
app.use('/api/users', userRoutes);
app.use('/api/products', productRoutes);

// Health check
app.get('/health', (req, res) => {
    res.json({
        status: 'OK',
        timestamp: new Date().toISOString(),
        uptime: process.uptime()
    });
});

// Error handling middleware
app.use(notFound);
app.use(globalErrorHandler);

app.listen(PORT, () => {
    console.log(`🚀 Server running on port ${PORT}`);
    console.log(`📝 Environment: ${process.env.NODE_ENV}`);
});

export default app;
```

Project Structure

```
src/
├── app.js                 # Main application file
├── config/
│   ├── database.js       # Database configuration
│   ├── redis.js          # Redis configuration
│   └── config.js         # App configuration
├── models/
│   ├── User.js           # User model
│   ├── Product.js        # Product model
│   └── Order.js          # Order model
├── controllers/
│   ├── authController.js  # Authentication controller
│   ├── userController.js  # User controller
│   └── productController.js # Product controller
├── services/
│   ├── authService.js     # Authentication service
│   ├── userService.js     # User service
│   ├── emailService.js    # Email service
│   └── cacheService.js    # Cache service
├── middleware/
│   ├── auth.js           # Authentication middleware
│   ├── validation.js     # Validation middleware
│   ├── errorHandler.js   # Error handling middleware
│   └── rateLimiter.js    # Rate limiting middleware
├── routes/
│   ├── auth.js           # Authentication routes
│   ├── users.js          # User routes
│   └── products.js       # Product routes
├── utils/
│   ├── AppError.js       # Custom error class
│   ├── catchAsync.js     # Async error catcher
│   ├── logger.js         # Logging utility
│   └── helpers.js        # Helper functions
└── validators/
    ├── authValidator.js   # Auth validation schemas
    ├── userValidator.js   # User validation schemas
    └── productValidator.js # Product validation schemas
```

Database Integration

Database Configuration (src/config/database.js)

```
import mongoose from 'mongoose';

export const connectDB = async () => {
    try {
        const conn = await mongoose.connect(process.env.MONGODB_URI, {
            useNewUrlParser: true,
            useUnifiedTopology: true,
        });
```

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

        // Handle connection events
        mongoose.connection.on('error', (err) => {
            console.error('MongoDB connection error:', err);
        });

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

        // Graceful shutdown
        process.on('SIGINT', async () => {
            await mongoose.connection.close();
            console.log('MongoDB connection closed.');
            process.exit(0);
        });

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

export default connectDB;
```

User Model (src/models/User.js)

```
import mongoose from 'mongoose';
import bcrypt from 'bcryptjs';
import jwt from 'jsonwebtoken';

const { Schema } = mongoose;

const userSchema = new Schema({
    firstName: {
        type: String,
        required: [true, 'First name is required'],
        trim: true,
        maxlength: [50, 'First name cannot exceed 50 characters']
    },
    lastName: {
        type: String,
        required: [true, 'Last name is required'],
        trim: true,
        maxlength: [50, 'Last name cannot exceed 50 characters']
    },
    email: {
        type: String,
        required: [true, 'Email is required'],
        unique: true,
        lowercase: true,
        match: [/^\S+@\S+\.\S+$/, 'Please enter a valid email']
    },
```

```
    password: {
        type: String,
        required: [true, 'Password is required'],
        minlength: [6, 'Password must be at least 6 characters'],
        select: false
    },
    role: {
        type: String,
        enum: ['user', 'admin', 'moderator'],
        default: 'user'
    },
    isActive: {
        type: Boolean,
        default: true
    },
    emailVerified: {
        type: Boolean,
        default: false
    },
    avatar: {
        type: String,
        default: ''
    },
    lastLoginAt: {
        type: Date
    },
    profile: {
        bio: String,
        website: String,
        location: String,
        birthday: Date
    }
}, {
    timestamps: true,
    toJSON: { virtuals: true },
    toObject: { virtuals: true }
});

// Virtual for full name
userSchema.virtual('fullName').get(function() {
    return `${this.firstName} ${this.lastName}`;
});

// Indexes for performance
userSchema.index({ email: 1 });
userSchema.index({ role: 1, isActive: 1 });
userSchema.index({ createdAt: -1 });

// Pre-save middleware to hash password
userSchema.pre('save', async function(next) {
    if (!this.isModified('password')) return next();

    try {
        const salt = await bcrypt.genSalt(12);
        this.password = await bcrypt.hash(this.password, salt);
        next();
```

```
    } catch (error) {
        next(error);
    }
});

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

userSchema.methods.generateAuthToken = function() {
    return jwt.sign(
        {
            id: this._id,
            email: this.email,
            role: this.role
        },
        process.env.JWT_SECRET,
        { expiresIn: process.env.JWT_EXPIRE }
    );
};

userSchema.methods.toPublicJSON = function() {
    const user = this.toObject();
    delete user.password;
    delete user.__v;
    return user;
};

// Static methods
userSchema.statics.findByEmail = function(email) {
    return this.findOne({ email }).select('+password');
};

userSchema.statics.findActiveUsers = function() {
    return this.find({ isActive: true });
};

export default mongoose.model('User', userSchema);
```

Product Model (src/models/Product.js)

```
import mongoose from 'mongoose';

const { Schema } = mongoose;

const reviewSchema = new Schema({
    user: {
        type: Schema.Types.ObjectId,
        ref: 'User',
        required: true
    },
    rating: {
        type: Number,
        required: true,
```

```
        min: 1,
        max: 5
    },
    comment: {
        type: String,
        required: true,
        maxlength: 500
    }
}, {
    timestamps: true
});

const productSchema = new Schema({
    name: {
        type: String,
        required: [true, 'Product name is required'],
        trim: true,
        maxlength: [100, 'Product name cannot exceed 100 characters']
    },
    description: {
        type: String,
        required: [true, 'Product description is required'],
        maxlength: [2000, 'Description cannot exceed 2000 characters']
    },
    price: {
        type: Number,
        required: [true, 'Product price is required'],
        min: [0, 'Price cannot be negative']
    },
    category: {
        type: String,
        required: [true, 'Product category is required'],
        enum: ['electronics', 'clothing', 'books', 'home', 'sports', 'beauty', 'toys']
    },
    brand: {
        type: String,
        required: true,
        trim: true
    },
    stock: {
        type: Number,
        required: true,
        min: [0, 'Stock cannot be negative'],
        default: 0
    },
    images: [{
        type: String,
        required: true
    }],
    reviews: [reviewSchema],
    rating: {
        type: Number,
        default: 0,
        min: 0,
        max: 5
    },
```

```
    numReviews: {
        type: Number,
        default: 0
    },
    isActive: {
        type: Boolean,
        default: true
    },
    tags: [{
        type: String,
        trim: true
    }],
    specifications: {
        type: Map,
        of: String
    }
}, {
    timestamps: true,
    toJSON: { virtuals: true },
    toObject: { virtuals: true }
});

// Virtual for availability
productSchema.virtual('isAvailable').get(function() {
    return this.stock > 0 && this.isActive;
});

// Indexes
productSchema.index({ name: 'text', description: 'text' });
productSchema.index({ category: 1, isActive: 1 });
productSchema.index({ price: 1 });
productSchema.index({ rating: -1 });
productSchema.index({ createdAt: -1 });

// Update rating when reviews change
productSchema.methods.updateRating = function() {
    if (this.reviews.length === 0) {
        this.rating = 0;
        this.numReviews = 0;
    } else {
        const totalRating = this.reviews.reduce((sum, review) => sum + review.rating, 0);
        this.rating = Math.round((totalRating / this.reviews.length) * 10) / 10;
        this.numReviews = this.reviews.length;
    }
};

// Pre-save middleware to update rating
productSchema.pre('save', function(next) {
    if (this.isModified('reviews')) {
        this.updateRating();
    }
    next();
});

export default mongoose.model('Product', productSchema);
```

RESTful API with ES6 Modules

User Controller (src/controllers/userController.js)

```
import User from '../models/User.js';
import { catchAsync } from '../utils/catchAsync.js';
import { AppError } from '../utils/AppError.js';
import { createResponse } from '../utils/responseHelper.js';
import { userService } from '../services/userService.js';

// @desc    Get all users
// @route   GET /api/users
// @access  Admin
export const getAllUsers = catchAsync(async (req, res) => {
    const {
        page = 1,
        limit = 10,
        sort = '-createdAt',
        role,
        isActive,
        search
    } = req.query;

    const result = await userService.getAllUsers({
        page: parseInt(page),
        limit: parseInt(limit),
        sort,
        role,
        isActive: isActive === 'true',
        search
    });

    res.status(200).json(createResponse(
        true,
        'Users retrieved successfully',
        result.users,
        result.pagination
    ));
});

// @desc    Get user by ID
// @route   GET /api/users/:id
// @access  Public
export const getUserById = catchAsync(async (req, res) => {
    const user = await userService.getUserById(req.params.id);

    if (!user) {
        throw new AppError('User not found', 404);
    }

    res.status(200).json(createResponse(
        true,
        'User retrieved successfully',
        user.toPublicJSON()
    ));
```

```
});

// @desc    Update user
// @route   PUT /api/users/:id
// @access  Private
export const updateUser = catchAsync(async (req, res) => {
    const user = await userService.updateUser(req.params.id, req.body);

    res.status(200).json(createResponse(
        true,
        'User updated successfully',
        user.toPublicJSON()
    ));
});

// @desc    Delete user
// @route   DELETE /api/users/:id
// @access  Admin
export const deleteUser = catchAsync(async (req, res) => {
    await userService.deleteUser(req.params.id);

    res.status(200).json(createResponse(
        true,
        'User deleted successfully'
    ));
});

// @desc    Get user profile
// @route   GET /api/users/profile
// @access  Private
export const getProfile = catchAsync(async (req, res) => {
    const user = await User.findById(req.user.id);

    res.status(200).json(createResponse(
        true,
        'Profile retrieved successfully',
        user.toPublicJSON()
    ));
});

// @desc    Update user profile
// @route   PUT /api/users/profile
// @access  Private
export const updateProfile = catchAsync(async (req, res) => {
    const user = await userService.updateUser(req.user.id, req.body);

    res.status(200).json(createResponse(
        true,
        'Profile updated successfully',
        user.toPublicJSON()
    ));
});

// @desc    Get user statistics
// @route   GET /api/users/stats
// @access  Admin
```

```
export const getUserStats = catchAsync(async (req, res) => {
    const stats = await userService.getUserStats();

    res.status(200).json(createResponse(
        true,
        'User statistics retrieved successfully',
        stats
    ));
});
```

Product Controller (src/controllers/productController.js)

```
import Product from '../models/Product.js';
import { catchAsync } from '../utils/catchAsync.js';
import { AppError } from '../utils/AppError.js';
import { createResponse } from '../utils/responseHelper.js';
import { productService } from '../services/productService.js';

// @desc    Get all products
// @route   GET /api/products
// @access  Public
export const getAllProducts = catchAsync(async (req, res) => {
    const {
        page = 1,
        limit = 12,
        sort = '-createdAt',
        category,
        minPrice,
        maxPrice,
        search,
        inStock
    } = req.query;

    const filters = {};
    if (category) filters.category = category;
    if (minPrice || maxPrice) {
        filters.price = {};
        if (minPrice) filters.price.$gte = parseFloat(minPrice);
        if (maxPrice) filters.price.$lte = parseFloat(maxPrice);
    }
    if (search) {
        filters.$text = { $search: search };
    }
    if (inStock === 'true') {
        filters.stock = { $gt: 0 };
    }

    const result = await productService.getAllProducts({
        page: parseInt(page),
        limit: parseInt(limit),
        sort,
        filters
    });

    res.status(200).json(createResponse(
```

```
        true,
        'Products retrieved successfully',
        result.products,
        result.pagination
    ));
});

// @desc    Get product by ID
// @route   GET /api/products/:id
// @access  Public
export const getProductById = catchAsync(async (req, res) => {
    const product = await Product.findById(req.params.id)
        .populate('reviews.user', 'firstName lastName avatar');

    if (!product) {
        throw new AppError('Product not found', 404);
    }

    res.status(200).json(createResponse(
        true,
        'Product retrieved successfully',
        product
    ));
});

// @desc    Create product
// @route   POST /api/products
// @access  Admin
export const createProduct = catchAsync(async (req, res) => {
    const product = await productService.createProduct(req.body);

    res.status(201).json(createResponse(
        true,
        'Product created successfully',
        product
    ));
});

// @desc    Update product
// @route   PUT /api/products/:id
// @access  Admin
export const updateProduct = catchAsync(async (req, res) => {
    const product = await productService.updateProduct(req.params.id, req.body);

    res.status(200).json(createResponse(
        true,
        'Product updated successfully',
        product
    ));
});

// @desc    Delete product
// @route   DELETE /api/products/:id
// @access  Admin
export const deleteProduct = catchAsync(async (req, res) => {
    await productService.deleteProduct(req.params.id);
```

```
    res.status(200).json(createResponse(
        true,
        'Product deleted successfully'
    ));
});

// @desc    Add product review
// @route   POST /api/products/:id/reviews
// @access  Private
export const addReview = catchAsync(async (req, res) => {
    const { rating, comment } = req.body;

    const product = await Product.findById(req.params.id);

    if (!product) {
        throw new AppError('Product not found', 404);
    }

    // Check if user already reviewed
    const existingReview = product.reviews.find(
        review => review.user.toString() === req.user.id
    );

    if (existingReview) {
        throw new AppError('Product already reviewed', 400);
    }

    const review = {
        user: req.user.id,
        rating: Number(rating),
        comment
    };

    product.reviews.push(review);
    await product.save();

    res.status(201).json(createResponse(
        true,
        'Review added successfully',
        review
    ));
});

// @desc    Get product categories
// @route   GET /api/products/categories
// @access  Public
export const getCategories = catchAsync(async (req, res) => {
    const categories = await Product.distinct('category');

    res.status(200).json(createResponse(
        true,
        'Categories retrieved successfully',
        categories
    ));
});
```

User Service (src/services/userService.js)

```
import User from '../models/User.js';
import { AppError } from '../utils/AppError.js';
import { cacheService } from './cacheService.js';

class UserService {
    async getAllUsers(options = {}) {
        const {
            page = 1,
            limit = 10,
            sort = '-createdAt',
            role,
            isActive,
            search
        } = options;

        const skip = (page - 1) * limit;
        const filters = {};

        if (role) filters.role = role;
        if (isActive !== undefined) filters.isActive = isActive;
        if (search) {
            filters.$or = [
                { firstName: { $regex: search, $options: 'i' } },
                { lastName: { $regex: search, $options: 'i' } },
                { email: { $regex: search, $options: 'i' } }
            ];
        }

        // Try to get from cache first
        const cacheKey = `users:${JSON.stringify({ filters, sort, page, limit })}`;
        const cached = await cacheService.get(cacheKey);

        if (cached) {
            return cached;
        }

        const users = await User.find(filters)
            .select('-password')
            .sort(sort)
            .skip(skip)
            .limit(limit)
            .lean();

        const total = await User.countDocuments(filters);

        const result = {
            users,
            pagination: {
                current: page,
                pages: Math.ceil(total / limit),
                total,
                hasNext: page < Math.ceil(total / limit),
                hasPrev: page > 1
            }
```

```
        };

        // Cache the result for 5 minutes
        await cacheService.set(cacheKey, result, 300);

        return result;
    }

    async getUserById(id) {
        const cacheKey = `user:${id}`;
        let user = await cacheService.get(cacheKey);

        if (!user) {
            user = await User.findById(id).select('-password');
            if (user) {
                await cacheService.set(cacheKey, user, 300);
            }
        }

        return user;
    }

    async createUser(userData) {
        // Check if user already exists
        const existingUser = await User.findOne({ email: userData.email });
        if (existingUser) {
            throw new AppError('User with this email already exists', 400);
        }

        const user = new User(userData);
        await user.save();

        // Clear related cache
        await cacheService.del('users:*');

        return user;
    }

    async updateUser(id, updateData) {
        // Remove sensitive fields that shouldn't be updated directly
        delete updateData.password;
        delete updateData.role;

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

        if (!user) {
            throw new AppError('User not found', 404);
        }

        // Update cache
        const cacheKey = `user:${id}`;
        await cacheService.set(cacheKey, user, 300);
```

```
        await cacheService.del('users:*');

        return user;
    }

    async deleteUser(id) {
        const user = await User.findByIdAndUpdate(
            id,
            { isActive: false },
            { new: true }
        );

        if (!user) {
            throw new AppError('User not found', 404);
        }

        // Clear cache
        await cacheService.del(`user:${id}`);
        await cacheService.del('users:*');

        return user;
    }

    async getUserStats() {
        const cacheKey = 'user:stats';
        let stats = await cacheService.get(cacheKey);

        if (!stats) {
            stats = await User.aggregate([
                {
                    $group: {
                        _id: '$role',
                        count: { $sum: 1 },
                        active: {
                            $sum: { $cond: ['$isActive', 1, 0] }
                        },
                        verified: {
                            $sum: { $cond: ['$emailVerified', 1, 0] }
                        }
                    }
                },
                { $sort: { count: -1 } }
            ]);

            await cacheService.set(cacheKey, stats, 600); // Cache for 10 minutes
        }

        return stats;
    }

    async searchUsers(query, options = {}) {
        const {
            page = 1,
            limit = 10,
            sort = '-createdAt'
        } = options;
```

```
        const skip = (page - 1) * limit;

        const users = await User.find({
            $or: [
                { firstName: { $regex: query, $options: 'i' } },
                { lastName: { $regex: query, $options: 'i' } },
                { email: { $regex: query, $options: 'i' } }
            ],
            isActive: true
        })
        .select('-password')
        .sort(sort)
        .skip(skip)
        .limit(limit);

        const total = await User.countDocuments({
            $or: [
                { firstName: { $regex: query, $options: 'i' } },
                { lastName: { $regex: query, $options: 'i' } },
                { email: { $regex: query, $options: 'i' } }
            ],
            isActive: true
        });

        return {
            users,
            pagination: {
                current: page,
                pages: Math.ceil(total / limit),
                total
            }
        };
    }
}

export const userService = new UserService();
export default userService;
```

Authentication & Authorization

Authentication Service (src/services/authService.js)

```
import User from '../models/User.js';
import { AppError } from '../utils/AppError.js';
import { emailService } from './emailService.js';
import { cacheService } from './cacheService.js';

class AuthService {
    async register(userData) {
        const { firstName, lastName, email, password } = userData;

        // Check if user exists
```

```
        const existingUser = await User.findOne({ email });
        if (existingUser) {
            throw new AppError('User already exists with this email', 400);
        }

        // Create user
        const user = new User({
            firstName,
            lastName,
            email,
            password
        });

        await user.save();

        // Generate token
        const token = user.generateAuthToken();

        // Send welcome email (async)
        emailService.sendWelcomeEmail(user.email, user.fullName)
            .catch(err => console.error('Welcome email failed:', err));

        return {
            user: user.toPublicJSON(),
            token
        };
    }

    async login(email, password) {
        // Find user with password
        const user = await User.findByEmail(email);

        if (!user || !(await user.comparePassword(password))) {
            throw new AppError('Invalid email or password', 401);
        }

        if (!user.isActive) {
            throw new AppError('Account is deactivated', 401);
        }

        // Update last login
        user.lastLoginAt = new Date();
        await user.save({ validateBeforeSave: false });

        // Generate token
        const token = user.generateAuthToken();

        return {
            user: user.toPublicJSON(),
            token
        };
    }

    async changePassword(userId, currentPassword, newPassword) {
        const user = await User.findById(userId).select('+password');
```

```
        if (!user) {
            throw new AppError('User not found', 404);
        }

        // Verify current password
        if (!(await user.comparePassword(currentPassword))) {
            throw new AppError('Current password is incorrect', 400);
        }

        // Update password
        user.password = newPassword;
        await user.save();

        return { message: 'Password updated successfully' };
    }

    async forgotPassword(email) {
        const user = await User.findOne({ email });

        if (!user) {
            throw new AppError('User not found with this email', 404);
        }

        // Generate reset token (you might want to use crypto.randomBytes)
        const resetToken = Math.random().toString(36).substr(2, 15);

        // Store reset token in cache (expires in 1 hour)
        await cacheService.set(
            `reset_token:${resetToken}`,
            user._id.toString(),
            3600
        );

        // Send reset email
        await emailService.sendPasswordResetEmail(user.email, resetToken);

        return { message: 'Password reset email sent' };
    }

    async resetPassword(resetToken, newPassword) {
        // Get user ID from cache
        const userId = await cacheService.get(`reset_token:${resetToken}`);

        if (!userId) {
            throw new AppError('Invalid or expired reset token', 400);
        }

        // Find user and update password
        const user = await User.findById(userId);
        if (!user) {
            throw new AppError('User not found', 404);
        }

        user.password = newPassword;
        await user.save();
```

```
        // Delete reset token from cache
        await cacheService.del(`reset_token:${resetToken}`);

        return { message: 'Password reset successful' };
    }

    async verifyEmail(userId, token) {
        const user = await User.findById(userId);

        if (!user) {
            throw new AppError('User not found', 404);
        }

        // Verify token (implement your verification logic)
        const isValidToken = await this.verifyEmailToken(token, userId);

        if (!isValidToken) {
            throw new AppError('Invalid verification token', 400);
        }

        user.emailVerified = true;
        await user.save({ validateBeforeSave: false });

        return { message: 'Email verified successfully' };
    }

    async verifyEmailToken(token, userId) {
        // Implement your email verification token logic
        const cachedToken = await cacheService.get(`email_verify:${userId}`);
        return cachedToken === token;
    }

    async refreshToken(userId) {
        const user = await User.findById(userId);

        if (!user || !user.isActive) {
            throw new AppError('User not found or inactive', 404);
        }

        const token = user.generateAuthToken();

        return {
            user: user.toPublicJSON(),
            token
        };
    }
}

export const authService = new AuthService();
export default authService;
```

Authentication Controller (src/controllers/authController.js)

```
import { authService } from '../services/authService.js';
import { catchAsync } from '../utils/catchAsync.js';
import { createResponse } from '../utils/responseHelper.js';

// @desc    Register user
// @route   POST /api/auth/register
// @access  Public
export const register = catchAsync(async (req, res) => {
    const { user, token } = await authService.register(req.body);

    // Set token in httpOnly cookie
    const cookieOptions = {
        expires: new Date(Date.now() + 7 * 24 * 60 * 60 * 1000), // 7 days
        httpOnly: true,
        secure: process.env.NODE_ENV === 'production',
        sameSite: 'strict'
    };

    res.cookie('token', token, cookieOptions);

    res.status(201).json(createResponse(
        true,
        'User registered successfully',
        { user, token }
    ));
});

// @desc    Login user
// @route   POST /api/auth/login
// @access  Public
export const login = catchAsync(async (req, res) => {
    const { email, password } = req.body;

    const { user, token } = await authService.login(email, password);

    // Set token in httpOnly cookie
    const cookieOptions = {
        expires: new Date(Date.now() + 7 * 24 * 60 * 60 * 1000), // 7 days
        httpOnly: true,
        secure: process.env.NODE_ENV === 'production',
        sameSite: 'strict'
    };

    res.cookie('token', token, cookieOptions);

    res.status(200).json(createResponse(
        true,
        'Login successful',
        { user, token }
    ));
});

// @desc    Logout user
// @route   POST /api/auth/logout
```

```
// @access  Private
export const logout = catchAsync(async (req, res) => {
    res.clearCookie('token');

    res.status(200).json(createResponse(
        true,
        'Logout successful'
    ));
});

// @desc    Get current user
// @route   GET /api/auth/me
// @access  Private
export const getCurrentUser = catchAsync(async (req, res) => {
    res.status(200).json(createResponse(
        true,
        'User profile retrieved',
        req.user.toPublicJSON()
    ));
});

// @desc    Change password
// @route   PUT /api/auth/change-password
// @access  Private
export const changePassword = catchAsync(async (req, res) => {
    const { currentPassword, newPassword } = req.body;

    await authService.changePassword(req.user.id, currentPassword, newPassword);

    res.status(200).json(createResponse(
        true,
        'Password changed successfully'
    ));
});

// @desc    Forgot password
// @route   POST /api/auth/forgot-password
// @access  Public
export const forgotPassword = catchAsync(async (req, res) => {
    const { email } = req.body;

    await authService.forgotPassword(email);

    res.status(200).json(createResponse(
        true,
        'Password reset email sent'
    ));
});

// @desc    Reset password
// @route   PUT /api/auth/reset-password/:token
// @access  Public
export const resetPassword = catchAsync(async (req, res) => {
    const { token } = req.params;
    const { password } = req.body;
```

```
    await authService.resetPassword(token, password);

    res.status(200).json(createResponse(
        true,
        'Password reset successful'
    ));
});

// @desc    Verify email
// @route   GET /api/auth/verify-email/:token
// @access  Public
export const verifyEmail = catchAsync(async (req, res) => {
    const { token } = req.params;
    const { userId } = req.query;

    await authService.verifyEmail(userId, token);

    res.status(200).json(createResponse(
        true,
        'Email verified successfully'
    ));
});

// @desc    Refresh token
// @route   POST /api/auth/refresh-token
// @access  Private
export const refreshToken = catchAsync(async (req, res) => {
    const { user, token } = await authService.refreshToken(req.user.id);

    res.status(200).json(createResponse(
        true,
        'Token refreshed successfully',
        { user, token }
    ));
});
```

Authentication Middleware (src/middleware/auth.js)

```
import jwt from 'jsonwebtoken';
import User from '../models/User.js';
import { AppError } from '../utils/AppError.js';
import { catchAsync } from '../utils/catchAsync.js';

// Protect routes - authentication required
export const protect = catchAsync(async (req, res, next) => {
    let token;

    // 1) Get token from header or cookie
    if (req.headers.authorization && req.headers.authorization.startsWith('Bearer')) {
        token = req.headers.authorization.split(' ')[1];
    } else if (req.cookies.token) {
        token = req.cookies.token;
    }

    // 2) Check if token exists
```

```
    if (!token) {
        throw new AppError('You are not logged in! Please log in to get access.', 401);
    }

    try {
        // 3) Verify token
        const decoded = jwt.verify(token, process.env.JWT_SECRET);

        // 4) Check if user still exists
        const currentUser = await User.findById(decoded.id);
        if (!currentUser) {
            throw new AppError('The user belonging to this token no longer exists.', 401)
        }

        // 5) Check if user is active
        if (!currentUser.isActive) {
            throw new AppError('Your account has been deactivated.', 401);
        }

        // Grant access to protected route
        req.user = currentUser;
        next();
    } catch (error) {
        if (error.name === 'JsonWebTokenError') {
            throw new AppError('Invalid token. Please log in again!', 401);
        } else if (error.name === 'TokenExpiredError') {
            throw new AppError('Your token has expired! Please log in again.', 401);
        }
        throw error;
    }
});

// Authorize specific roles
export const authorize = (...roles) => {
    return (req, res, next) => {
        if (!roles.includes(req.user.role)) {
            throw new AppError(
                'You do not have permission to perform this action',
                403
            );
        }
        next();
    };
};

// Optional authentication - doesn't throw error if no token
export const optionalAuth = catchAsync(async (req, res, next) => {
    let token;

    if (req.headers.authorization && req.headers.authorization.startsWith('Bearer')) {
        token = req.headers.authorization.split(' ')[1];
    } else if (req.cookies.token) {
        token = req.cookies.token;
    }

    if (token) {
```

```
        try {
            const decoded = jwt.verify(token, process.env.JWT_SECRET);
            const currentUser = await User.findById(decoded.id);

            if (currentUser && currentUser.isActive) {
                req.user = currentUser;
            }
        } catch (error) {
            // Ignore token errors for optional auth
        }
    }

    next();
});

export default { protect, authorize, optionalAuth };
```

Middleware Development

Validation Middleware (src/middleware/validation.js)

```
import Joi from 'joi';
import { AppError } from '../utils/AppError.js';

export const validate = (schema) => {
    return (req, res, next) => {
        const { error } = schema.validate(req.body, { abortEarly: false });

        if (error) {
            const errors = error.details.map(detail => ({
                field: detail.path.join('.'),
                message: detail.message
            }));

            throw new AppError('Validation Error', 400, errors);
        }

        next();
    };
};

export const validateQuery = (schema) => {
    return (req, res, next) => {
        const { error } = schema.validate(req.query, { abortEarly: false });

        if (error) {
            const errors = error.details.map(detail => ({
                field: detail.path.join('.'),
                message: detail.message
            }));

            throw new AppError('Query Validation Error', 400, errors);
        }
```

```
        next();
    };
};

export const validateParams = (schema) => {
    return (req, res, next) => {
        const { error } = schema.validate(req.params, { abortEarly: false });

        if (error) {
            const errors = error.details.map(detail => ({
                field: detail.path.join('.'),
                message: detail.message
            }));

            throw new AppError('Parameter Validation Error', 400, errors);
        }

        next();
    };
};

export default { validate, validateQuery, validateParams };
```

Rate Limiting Middleware (src/middleware/rateLimiter.js)

```
import { cacheService } from '../services/cacheService.js';
import { AppError } from '../utils/AppError.js';

export const createRateLimiter = (options = {}) => {
    const {
        windowMs = 15 * 60 * 1000, // 15 minutes
        max = 100, // limit each IP to 100 requests per windowMs
        message = 'Too many requests from this IP, please try again later.',
        keyGenerator = (req) => req.ip,
        skip = () => false
    } = options;

    return async (req, res, next) => {
        if (skip(req)) {
            return next();
        }

        const key = `rate_limit:${keyGenerator(req)}`;
        const now = Date.now();
        const windowStart = now - windowMs;

        try {
            // Get current requests from cache
            let requests = await cacheService.get(key) || [];

            // Filter out requests outside the window
            requests = requests.filter(time => time > windowStart);

            // Check if limit exceeded
```

```
            if (requests.length >= max) {
                const oldestRequest = Math.min(...requests);
                const resetTime = oldestRequest + windowMs;

                res.set({
                    'X-RateLimit-Limit': max,
                    'X-RateLimit-Remaining': 0,
                    'X-RateLimit-Reset': Math.ceil(resetTime / 1000)
                });

                throw new AppError(message, 429);
            }

            // Add current request
            requests.push(now);

            // Store in cache
            await cacheService.set(key, requests, Math.ceil(windowMs / 1000));

            // Set rate limit headers
            res.set({
                'X-RateLimit-Limit': max,
                'X-RateLimit-Remaining': max - requests.length,
                'X-RateLimit-Reset': Math.ceil((now + windowMs) / 1000)
            });

            next();
        } catch (error) {
            if (error instanceof AppError) {
                throw error;
            }
            // If cache fails, allow the request
            next();
        }
    };
};

// Predefined rate limiters
export const authLimiter = createRateLimiter({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 5, // limit each IP to 5 requests per windowMs for auth routes
    message: 'Too many authentication attempts, please try again later.'
});

export const apiLimiter = createRateLimiter({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 100 // limit each IP to 100 requests per windowMs
});

export const strictLimiter = createRateLimiter({
    windowMs: 60 * 1000, // 1 minute
    max: 10, // limit each IP to 10 requests per minute
    message: 'Rate limit exceeded for this endpoint.'
});

export default { createRateLimiter, authLimiter, apiLimiter, strictLimiter };
```

Request Logging Middleware (src/middleware/logger.js)

```
import { logger } from '../utils/logger.js';

export const requestLogger = (req, res, next) => {
    const start = Date.now();
    const { method, url, ip, headers } = req;

    // Log request
    logger.info(`${method} ${url}`, {
        ip,
        userAgent: headers['user-agent'],
        timestamp: new Date().toISOString()
    });

    // Capture response
    const originalSend = res.send;
    const originalJson = res.json;

    res.send = function(body) {
        logResponse();
        return originalSend.call(this, body);
    };

    res.json = function(obj) {
        logResponse();
        return originalJson.call(this, obj);
    };

    function logResponse() {
        const duration = Date.now() - start;
        const { statusCode } = res;

        const logLevel = statusCode >= 500 ? 'error' :
                         statusCode >= 400 ? 'warn' : 'info';

        logger[logLevel](`${method} ${url} ${statusCode}`, {
            duration: `${duration}ms`,
            statusCode,
            ip,
            timestamp: new Date().toISOString()
        });
    }

    next();
};

export default requestLogger;
```

Error Handling

Custom Error Class (src/utils/AppError.js)

```
export class AppError extends Error {
    constructor(message, statusCode, errors = null) {
        super(message);

        this.statusCode = statusCode;
        this.status = `${statusCode}`.startsWith('4') ? 'fail' : 'error';
        this.isOperational = true;
        this.errors = errors;

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

export default AppError;
```

Async Error Catcher (src/utils/catchAsync.js)

```
export const catchAsync = (fn) => {
    return (req, res, next) => {
        Promise.resolve(fn(req, res, next)).catch(next);
    };
};

export default catchAsync;
```

Global Error Handler (src/middleware/errorHandler.js)

```
import { AppError } from '../utils/AppError.js';
import { logger } from '../utils/logger.js';

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(el => ({
        field: el.path,
        message: el.message
    }));
    return new AppError('Invalid input data', 400, errors);
```

```
};

const handleJWTError = () =>
    new AppError('Invalid token. Please log in again!', 401);

const handleJWTExpiredError = () =>
    new AppError('Your token has expired! Please log in again.', 401);

const sendErrorDev = (err, res) => {
    logger.error('Development Error:', {
        error: err.message,
        stack: err.stack,
        statusCode: err.statusCode
    });

    res.status(err.statusCode).json({
        success: false,
        error: err.message,
        errors: err.errors,
        stack: err.stack,
        statusCode: err.statusCode
    });
};

const sendErrorProd = (err, res) => {
    // Operational, trusted error: send message to client
    if (err.isOperational) {
        res.status(err.statusCode).json({
            success: false,
            message: err.message,
            errors: err.errors
        });
    } else {
        // Programming or other unknown error: don't leak error details
        logger.error('Production Error:', {
            error: err.message,
            stack: err.stack
        });

        res.status(500).json({
            success: false,
            message: 'Something went wrong!'
        });
    }
};

export const globalErrorHandler = (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);
        if (error.name === 'JsonWebTokenError') error = handleJWTError();
        if (error.name === 'TokenExpiredError') error = handleJWTExpiredError();

        sendErrorProd(error, res);
    }
};

export default globalErrorHandler;
```

Not Found Middleware (src/middleware/notFound.js)

```
import { AppError } from '../utils/AppError.js';

export const notFound = (req, res, next) => {
    const message = `Route ${req.originalUrl} not found`;
    next(new AppError(message, 404));
};

export default notFound;
```

Advanced Patterns with ES6

Repository Pattern (src/repositories/BaseRepository.js)

```
export class BaseRepository {
    constructor(model) {
        this.model = model;
    }

    async findAll(filter = {}, options = {}) {
        const {
            page = 1,
            limit = 10,
            sort = '-createdAt',
            populate = '',
            select = ''
        } = options;

        const skip = (page - 1) * limit;

        let query = this.model.find(filter);

        if (select) query = query.select(select);
        if (populate) query = query.populate(populate);

        query = query.sort(sort).skip(skip).limit(limit);

        const [data, total] = await Promise.all([
```

```
            query.exec(),
            this.model.countDocuments(filter)
        ]);

        return {
            data,
            pagination: {
                current: page,
                pages: Math.ceil(total / limit),
                total,
                hasNext: page < Math.ceil(total / limit),
                hasPrev: page > 1
            }
        };
    }

    async findById(id, populate = '') {
        let query = this.model.findById(id);
        if (populate) query = query.populate(populate);
        return await query.exec();
    }

    async findOne(filter, populate = '') {
        let query = this.model.findOne(filter);
        if (populate) query = query.populate(populate);
        return await query.exec();
    }

    async create(data) {
        const document = new this.model(data);
        return await document.save();
    }

    async createMany(dataArray) {
        return await this.model.insertMany(dataArray);
    }

    async update(id, data) {
        return await this.model.findByIdAndUpdate(
            id,
            data,
            { new: true, runValidators: true }
        );
    }

    async updateMany(filter, data) {
        return await this.model.updateMany(filter, data);
    }

    async delete(id) {
        return await this.model.findByIdAndDelete(id);
    }

    async deleteMany(filter) {
        return await this.model.deleteMany(filter);
    }
```

```
    async count(filter = {}) {
        return await this.model.countDocuments(filter);
    }

    async exists(filter) {
        const doc = await this.model.findOne(filter).select('_id');
        return !!doc;
    }

    async aggregate(pipeline) {
        return await this.model.aggregate(pipeline);
    }
}

export default BaseRepository;
```

User Repository (src/repositories/UserRepository.js)

```
import { BaseRepository } from './BaseRepository.js';
import User from '../models/User.js';

export class UserRepository extends BaseRepository {
    constructor() {
        super(User);
    }

    async findByEmail(email) {
        return await this.model.findOne({ email }).select('+password');
    }

    async findActiveUsers(filter = {}) {
        return await this.model.find({ ...filter, isActive: true });
    }

    async updateLastLogin(userId) {
        return await this.model.findByIdAndUpdate(
            userId,
            { lastLoginAt: new Date() },
            { new: true }
        );
    }

    async getUsersStats() {
        return await this.model.aggregate([
            {
                $group: {
                    _id: '$role',
                    count: { $sum: 1 },
                    active: { $sum: { $cond: ['$isActive', 1, 0] } },
                    verified: { $sum: { $cond: ['$emailVerified', 1, 0] } }
                }
            },
            { $sort: { count: -1 } }
        ]);
```

```
    }

    async searchUsers(searchTerm, options = {}) {
        const searchFilter = {
            $or: [
                { firstName: { $regex: searchTerm, $options: 'i' } },
                { lastName: { $regex: searchTerm, $options: 'i' } },
                { email: { $regex: searchTerm, $options: 'i' } }
            ],
            isActive: true
        };

        return await this.findAll(searchFilter, options);
    }

    async deactivateUser(userId) {
        return await this.update(userId, { isActive: false });
    }
}

export const userRepository = new UserRepository();
export default userRepository;
```

Event Emitter Pattern (src/events/EventEmitter.js)

```
import { EventEmitter } from 'events';
import { logger } from '../utils/logger.js';

class AppEventEmitter extends EventEmitter {
    emit(event, data) {
        logger.info(`Event emitted: ${event}`, { data });
        return super.emit(event, data);
    }

    onAny(listener) {
        this.originalEmit = this.emit;
        this.emit = (event, ...args) => {
            listener(event, ...args);
            return this.originalEmit(event, ...args);
        };
    }
}

export const eventEmitter = new AppEventEmitter();

// Set max listeners to prevent memory leak warnings
eventEmitter.setMaxListeners(20);

export default eventEmitter;
```

Event Handlers (src/events/handlers.js)

```
import { eventEmitter } from './EventEmitter.js';
import { emailService } from '../services/emailService.js';
import { logger } from '../utils/logger.js';

// User events
eventEmitter.on('user.created', async (userData) => {
    try {
        await emailService.sendWelcomeEmail(userData.email, userData.fullName);
        logger.info(`Welcome email sent to ${userData.email}`);
    } catch (error) {
        logger.error('Error sending welcome email:', error);
    }
});

eventEmitter.on('user.updated', async (userData) => {
    try {
        logger.info(`User ${userData.id} profile updated`);
        // Add other user update logic here
    } catch (error) {
        logger.error('Error handling user update:', error);
    }
});

eventEmitter.on('user.deleted', async (userData) => {
    try {
        logger.info(`User ${userData.id} deactivated`);
        // Add cleanup logic here
    } catch (error) {
        logger.error('Error handling user deletion:', error);
    }
});

// Product events
eventEmitter.on('product.created', async (productData) => {
    try {
        logger.info(`Product ${productData.name} created`);
        // Add product creation logic here
    } catch (error) {
        logger.error('Error handling product creation:', error);
    }
});

eventEmitter.on('product.outOfStock', async (productData) => {
    try {
        logger.warn(`Product ${productData.name} is out of stock`);
        // Send notification to admin
    } catch (error) {
        logger.error('Error handling out of stock event:', error);
    }
});

// Order events
eventEmitter.on('order.created', async (orderData) => {
    try {
```

```
        await emailService.sendOrderConfirmation(orderData.userEmail, orderData);
        logger.info(`Order confirmation sent for order ${orderData.id}`);
    } catch (error) {
        logger.error('Error sending order confirmation:', error);
    }
});

export default eventEmitter;
```

Complete Real-World Example

Main Routes (src/routes/users.js)

```
import express from 'express';
import {
    getAllUsers,
    getUserById,
    updateUser,
    deleteUser,
    getProfile,
    updateProfile,
    getUserStats
} from '../controllers/userController.js';
import { protect, authorize } from '../middleware/auth.js';
import { validate, validateParams, validateQuery } from '../middleware/validation.js';
import { userUpdateSchema, userQuerySchema, idParamSchema } from '../validators/userValid
import { apiLimiter } from '../middleware/rateLimiter.js';

const router = express.Router();

// Apply rate limiting to all routes
router.use(apiLimiter);

// Public routes
router.get('/:id', validateParams(idParamSchema), getUserById);

// Protected routes
router.use(protect); // All routes after this middleware require authentication

router.get('/profile/me', getProfile);
router.put('/profile/me', validate(userUpdateSchema), updateProfile);

// Admin routes
router.get(
    '/',
    authorize('admin'),
    validateQuery(userQuerySchema),
    getAllUsers
);

router.put(
    '/:id',
    authorize('admin'),
```

```
    validateParams(idParamSchema),
    validate(userUpdateSchema),
    updateUser
);

router.delete(
    '/:id',
    authorize('admin'),
    validateParams(idParamSchema),
    deleteUser
);

router.get(
    '/stats/overview',
    authorize('admin'),
    getUserStats
);

export default router;
```

Authentication Routes (src/routes/auth.js)

```
import express from 'express';
import {
    register,
    login,
    logout,
    getCurrentUser,
    changePassword,
    forgotPassword,
    resetPassword,
    verifyEmail,
    refreshToken
} from '../controllers/authController.js';
import { protect } from '../middleware/auth.js';
import { validate, validateParams } from '../middleware/validation.js';
import {
    registerSchema,
    loginSchema,
    changePasswordSchema,
    forgotPasswordSchema,
    resetPasswordSchema,
    tokenParamSchema
} from '../validators/authValidator.js';
import { authLimiter } from '../middleware/rateLimiter.js';

const router = express.Router();

// Apply stricter rate limiting to auth routes
router.use(authLimiter);

// Public routes
router.post('/register', validate(registerSchema), register);
router.post('/login', validate(loginSchema), login);
router.post('/forgot-password', validate(forgotPasswordSchema), forgotPassword);
```

```
router.put(
    '/reset-password/:token',
    validateParams(tokenParamSchema),
    validate(resetPasswordSchema),
    resetPassword
);
router.get(
    '/verify-email/:token',
    validateParams(tokenParamSchema),
    verifyEmail
);

// Protected routes
router.use(protect);

router.get('/me', getCurrentUser);
router.post('/logout', logout);
router.put('/change-password', validate(changePasswordSchema), changePassword);
router.post('/refresh-token', refreshToken);

export default router;
```

Validation Schemas (src/validators/userValidator.js)

```
import Joi from 'joi';

export const userUpdateSchema = Joi.object({
    firstName: Joi.string().trim().min(2).max(50),
    lastName: Joi.string().trim().min(2).max(50),
    email: Joi.string().email().lowercase(),
    profile: Joi.object({
        bio: Joi.string().max(500).allow(''),
        website: Joi.string().uri().allow(''),
        location: Joi.string().max(100).allow(''),
        birthday: Joi.date().max('now')
    })
}).min(1);

export const userQuerySchema = Joi.object({
    page: Joi.number().integer().min(1).default(1),
    limit: Joi.number().integer().min(1).max(100).default(10),
    sort: Joi.string().valid('createdAt', '-createdAt', 'firstName', '-firstName', 'email
    role: Joi.string().valid('user', 'admin', 'moderator'),
    isActive: Joi.boolean(),
    search: Joi.string().trim().min(1).max(100)
});

export const idParamSchema = Joi.object({
    id: Joi.string().regex(/^[0-9a-fA-F]{24}$/).required().messages({
        'string.pattern.base': 'Invalid ID format'
    })
});

export default {
    userUpdateSchema,
```

```
    userQuerySchema,
    idParamSchema
};
```

Validation Schemas (src/validators/authValidator.js)

```
import Joi from 'joi';

export const registerSchema = Joi.object({
    firstName: Joi.string().trim().min(2).max(50).required(),
    lastName: Joi.string().trim().min(2).max(50).required(),
    email: Joi.string().email().lowercase().required(),
    password: Joi.string().min(6).max(128).required()
        .pattern(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/)
        .messages({
            'string.pattern.base': 'Password must contain at least one lowercase letter,
        })
});

export const loginSchema = Joi.object({
    email: Joi.string().email().required(),
    password: Joi.string().required()
});

export const changePasswordSchema = Joi.object({
    currentPassword: Joi.string().required(),
    newPassword: Joi.string().min(6).max(128).required()
        .pattern(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/)
        .messages({
            'string.pattern.base': 'Password must contain at least one lowercase letter,
        })
});

export const forgotPasswordSchema = Joi.object({
    email: Joi.string().email().required()
});

export const resetPasswordSchema = Joi.object({
    password: Joi.string().min(6).max(128).required()
        .pattern(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/)
        .messages({
            'string.pattern.base': 'Password must contain at least one lowercase letter,
        })
});

export const tokenParamSchema = Joi.object({
    token: Joi.string().required()
});

export default {
    registerSchema,
    loginSchema,
    changePasswordSchema,
    forgotPasswordSchema,
    resetPasswordSchema,
```

```
    tokenParamSchema
};
```

Utility Functions (src/utils/responseHelper.js)

```
export const createResponse = (success, message, data = null, pagination = null, errors =
    const response = {
        success,
        message,
        timestamp: new Date().toISOString()
    };

    if (data !== null) response.data = data;
    if (pagination !== null) response.pagination = pagination;
    if (errors !== null) response.errors = errors;

    return response;
};

export const createErrorResponse = (message, errors = null) => {
    return createResponse(false, message, null, null, errors);
};

export const createSuccessResponse = (message, data = null, pagination = null) => {
    return createResponse(true, message, data, pagination);
};

export default {
    createResponse,
    createErrorResponse,
    createSuccessResponse
};
```

Logger Utility (src/utils/logger.js)

```
import fs from 'fs';
import path from 'path';

class Logger {
    constructor() {
        this.logDir = 'logs';
        this.ensureLogDirectory();
    }

    ensureLogDirectory() {
        if (!fs.existsSync(this.logDir)) {
            fs.mkdirSync(this.logDir, { recursive: true });
        }
    }

    formatMessage(level, message, meta = {}) {
        return JSON.stringify({
            timestamp: new Date().toISOString(),
```

```
            level: level.toUpperCase(),
            message,
            ...meta
        }) + '\n';
    }

    writeToFile(filename, content) {
        const filePath = path.join(this.logDir, filename);
        fs.appendFileSync(filePath, content);
    }

    info(message, meta = {}) {
        const formatted = this.formatMessage('info', message, meta);
        console.log(formatted.trim());
        this.writeToFile('app.log', formatted);
    }

    error(message, meta = {}) {
        const formatted = this.formatMessage('error', message, meta);
        console.error(formatted.trim());
        this.writeToFile('error.log', formatted);
        this.writeToFile('app.log', formatted);
    }

    warn(message, meta = {}) {
        const formatted = this.formatMessage('warn', message, meta);
        console.warn(formatted.trim());
        this.writeToFile('app.log', formatted);
    }

    debug(message, meta = {}) {
        if (process.env.NODE_ENV === 'development') {
            const formatted = this.formatMessage('debug', message, meta);
            console.log(formatted.trim());
            this.writeToFile('debug.log', formatted);
        }
    }
}

export const logger = new Logger();
export default logger;
```

Running the Application

Start Script (package.json scripts)

```
{
  "scripts": {
    "start": "node src/app.js",
    "dev": "nodemon src/app.js",
    "test": "NODE_ENV=test jest --detectOpenHandles",
    "test:watch": "NODE_ENV=test jest --watchAll --detectOpenHandles",
    "test:coverage": "NODE_ENV=test jest --coverage --detectOpenHandles"
```

```
  }
}
```

Development Server

```
# Install dependencies
npm install

# Start development server
npm run dev

# The server will start on http://localhost:3000
```

Key Benefits of ES6 Modules

1. **Static Analysis**: Import/export statements are analyzed at compile time
2. **Tree Shaking**: Unused code can be eliminated by bundlers
3. **Better IDE Support**: Enhanced autocomplete and refactoring
4. **Cleaner Syntax**: More readable than CommonJS
5. **Standard Compliance**: Part of the JavaScript specification
6. **Circular Dependencies**: Better handling than CommonJS
7. **Named Exports**: Multiple exports from a single module
8. **Default Exports**: Single main export per module

This complete ES6 modules example demonstrates a production-ready Express.js backend with modern JavaScript features, proper error handling, authentication, validation, caching, and more. The modular structure makes the codebase maintainable and scalable.