# File Paths and Contents

File Path: .\.env

Content:

NODE\_ENV=development  
PORT=5000  
MONGO\_URI=mongodb+srv://shonizharidy:shonizharidy@cluster0.48zf33d.mongodb.net/perfume\_store?retryWrites=true&w=majority&appName=Cluster0  
JWT\_SECRET=your\_jwt\_secret\_here  
JWT\_EXPIRES\_IN=7d

File Path: .\index.js

Content:

const express = require('express');  
const mongoose = require('mongoose');  
const cors = require('cors');  
const dotenv = require('dotenv');  
const path = require('path');  
const cookieParser = require('cookie-parser');  
const multer = require('multer');  
const { v4: uuidv4 } = require('uuid');  
  
// Load environment variables  
dotenv.config();  
  
// Initialize Express app  
const app = express();  
  
// Middleware  
app.use(cors());  
app.use(express.json());  
app.use(cookieParser());  
app.use(express.urlencoded({ extended: true }));  
  
// Configure multer for file uploads  
const storage = multer.diskStorage({  
 destination: (req, file, cb) => {  
 cb(null, 'uploads/');  
 },  
 filename: (req, file, cb) => {  
 const uniqueName = `${uuidv4()}-${file.originalname}`;  
 cb(null, uniqueName);  
 }  
});  
  
const upload = multer({ storage });  
  
// Serve static files  
app.use('/uploads', express.static(path.join(\_\_dirname, 'uploads')));  
  
// Add a global middleware to transform image URLs in API responses  
app.use((req, res, next) => {  
 // Store the original res.json function  
 const originalJson = res.json;  
   
 // Create a function to convert image paths to full URLs  
 const convertImagePaths = (obj) => {  
 if (!obj) return obj;  
   
 // If it's a Mongoose document, convert to a plain object first  
 if (obj.constructor && obj.constructor.name === 'model') {  
 obj = obj.toObject();  
 }  
   
 if (Array.isArray(obj)) {  
 return obj.map(item => {  
 // If array item is a Mongoose document, convert to plain object  
 if (item && item.constructor && item.constructor.name === 'model') {  
 return convertImagePaths(item.toObject());  
 }  
 return convertImagePaths(item);  
 });  
 }  
   
 if (typeof obj === 'object' && obj !== null) {  
 const newObj = { ...obj };  
 for (const key in newObj) {  
 if (key === 'image' || key === 'boxImage' || key === 'avatar' || key.endsWith('Image')) {  
 if (typeof newObj[key] === 'string' && newObj[key].startsWith('/uploads')) {  
 // Get the server base URL  
 const baseUrl = `${req.protocol}://${req.get('host')}`;  
 newObj[key] = `${baseUrl}${newObj[key]}`;  
 }  
 } else if (typeof newObj[key] === 'object' && newObj[key] !== null) {  
 newObj[key] = convertImagePaths(newObj[key]);  
 }  
 }  
 return newObj;  
 }  
   
 return obj;  
 };  
   
 // Override res.json  
 res.json = function(data) {  
 // Apply conversion to the response data  
 const convertedData = convertImagePaths(data);  
 // Call the original json method with the converted data  
 return originalJson.call(this, convertedData);  
 };  
   
 next();  
});  
  
// MongoDB connection  
mongoose.connect(process.env.MONGO\_URI || 'mongodb://localhost:27017/perfume-store')  
 .then(() => console.log('Connected to MongoDB'))  
 .catch(err => console.error('MongoDB connection error:', err));  
  
// Import routes  
const productRoutes = require('./routes/productRoutes');  
const authRoutes = require('./routes/authRoutes');  
const orderRoutes = require('./routes/orderRoutes');  
const adminRoutes = require('./routes/adminRoutes');  
  
// Use routes  
app.use('/api/products', productRoutes);  
app.use('/api/auth', authRoutes);  
app.use('/api/orders', orderRoutes);  
app.use('/api/admin', adminRoutes);  
  
// Serve frontend in production  
if (process.env.NODE\_ENV === 'production') {  
 app.use(express.static(path.join(\_\_dirname, '../dist')));  
   
 app.get('\*', (req, res) => {  
 res.sendFile(path.join(\_\_dirname, '../dist/index.html'));  
 });  
}  
  
// Error handling middleware  
app.use((err, req, res, next) => {  
 console.error(err.stack);  
 res.status(500).json({  
 success: false,  
 message: 'Server error',  
 error: process.env.NODE\_ENV === 'development' ? err.message : undefined  
 });  
});  
  
// Start server  
const PORT = process.env.PORT || 5000;  
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));

File Path: .\package.json

Content:

{  
 "dependencies": {  
 "bcrypt": "^5.1.1",  
 "cookie-parser": "^1.4.7",  
 "cors": "^2.8.5",  
 "dotenv": "^16.5.0",  
 "express": "^5.1.0",  
 "jsonwebtoken": "^9.0.2",  
 "mongoose": "^8.14.1",  
 "multer": "^1.4.5-lts.2",  
 "uuid": "^11.1.0"  
 }  
}

File Path: .\middleware\adminAuth.js

Content:

const auth = require('./auth');  
  
module.exports = (req, res, next) => {  
 auth(req, res, () => {  
 if (req.user && req.user.role === 'admin') {  
 next();  
 } else {  
 res.status(403).json({ message: 'Admin access required' });  
 }  
 });  
};

File Path: .\middleware\auth.js

Content:

const jwt = require('jsonwebtoken');  
const User = require('../models/User');  
  
module.exports = async (req, res, next) => {  
 try {  
 // Get token from request headers, cookies, or query params  
 const token =   
 req.cookies.token ||   
 req.header('Authorization')?.replace('Bearer ', '') ||   
 req.query.token;  
   
 if (!token) {  
 return res.status(401).json({ message: 'Authentication required' });  
 }  
   
 // Verify token  
 const decoded = jwt.verify(token, process.env.JWT\_SECRET);  
   
 // Find user by id  
 const user = await User.findById(decoded.userId);  
   
 if (!user) {  
 return res.status(401).json({ message: 'User not found' });  
 }  
   
 // Attach user to request object  
 req.user = user;  
 next();  
 } catch (err) {  
 res.status(401).json({ message: 'Invalid token' });  
 }  
};

File Path: .\models\Order.js

Content:

const mongoose = require('mongoose');  
  
const orderSchema = new mongoose.Schema({  
 orderNumber: {  
 type: String,  
 required: true,  
 unique: true  
 },  
 user: {  
 type: mongoose.Schema.Types.ObjectId,  
 ref: 'User',  
 required: true  
 },  
 items: [  
 {  
 product: {  
 type: mongoose.Schema.Types.ObjectId,  
 ref: 'Product',  
 required: true  
 },  
 quantity: {  
 type: Number,  
 required: true,  
 min: 1  
 },  
 price: {  
 type: Number,  
 required: true,  
 min: 0  
 }  
 }  
 ],  
 totalAmount: {  
 type: Number,  
 required: true,  
 min: 0  
 },  
 status: {  
 type: String,  
 enum: ['pending', 'processing', 'shipped', 'completed', 'cancelled'],  
 default: 'pending'  
 },  
 shippingAddress: {  
 fullName: String,  
 address: String,  
 city: String,  
 postalCode: String,  
 country: String,  
 phone: String  
 },  
 paymentMethod: {  
 type: String,  
 enum: ['credit\_card', 'cash\_on\_delivery'],  
 required: true  
 },  
 paymentStatus: {  
 type: String,  
 enum: ['pending', 'paid', 'failed'],  
 default: 'pending'  
 },  
 paymentDetails: {  
 transactionId: String,  
 paymentDate: Date  
 },  
 createdAt: {  
 type: Date,  
 default: Date.now  
 },  
 updatedAt: {  
 type: Date,  
 default: Date.now  
 }  
});  
  
// Update timestamp on save  
orderSchema.pre('save', function(next) {  
 this.updatedAt = Date.now();  
 next();  
});  
  
// Generate order number  
orderSchema.pre('save', async function(next) {  
 if (!this.orderNumber) {  
 // Format: ORD-YYYYMMDD-XXXX (XXXX is a random 4-digit number)  
 const date = new Date();  
 const dateStr = date.getFullYear().toString() +  
 (date.getMonth() + 1).toString().padStart(2, '0') +  
 date.getDate().toString().padStart(2, '0');  
   
 const randomNum = Math.floor(1000 + Math.random() \* 9000); // 4-digit random number  
 this.orderNumber = `ORD-${dateStr}-${randomNum}`;  
 }  
 next();  
});  
  
module.exports = mongoose.model('Order', orderSchema);

File Path: .\models\Product.js

Content:

const mongoose = require('mongoose');  
  
const productSchema = new mongoose.Schema({  
 name: {  
 type: String,  
 required: true,  
 trim: true  
 },  
 nameAr: {  
 type: String,  
 required: true,  
 trim: true  
 },  
 price: {  
 type: Number,  
 required: true,  
 min: 0  
 },  
 currency: {  
 type: String,  
 default: '﷼'  
 },  
 description: {  
 type: String,  
 trim: true  
 },  
 descriptionAr: {  
 type: String,  
 trim: true  
 },  
 category: {  
 type: String,  
 required: true,  
 enum: ['spray', 'perfume', 'candle', 'gift']  
 },  
 image: {  
 type: String,  
 required: true  
 },  
 boxImage: {  
 type: String,  
 required: true  
 },  
 stock: {  
 type: Number,  
 default: 0  
 },  
 featured: {  
 type: Boolean,  
 default: false  
 },  
 createdAt: {  
 type: Date,  
 default: Date.now  
 },  
 updatedAt: {  
 type: Date,  
 default: Date.now  
 }  
});  
  
// Update timestamp on save  
productSchema.pre('save', function(next) {  
 this.updatedAt = Date.now();  
 next();  
});  
  
module.exports = mongoose.model('Product', productSchema);

File Path: .\models\User.js

Content:

const mongoose = require('mongoose');  
const bcrypt = require('bcrypt');  
  
const userSchema = new mongoose.Schema({  
 name: {  
 type: String,  
 required: true,  
 trim: true  
 },  
 email: {  
 type: String,  
 required: true,  
 unique: true,  
 trim: true,  
 lowercase: true  
 },  
 password: {  
 type: String,  
 required: true,  
 minlength: 6  
 },  
 role: {  
 type: String,  
 enum: ['user', 'admin'],  
 default: 'user'  
 },  
 phone: {  
 type: String,  
 trim: true  
 },  
 addresses: [  
 {  
 title: {  
 type: String,  
 default: 'Default Address'  
 },  
 fullName: String,  
 address: String,  
 city: String,  
 postalCode: String,  
 country: String,  
 phone: String,  
 isDefault: {  
 type: Boolean,  
 default: false  
 }  
 }  
 ],  
 createdAt: {  
 type: Date,  
 default: Date.now  
 }  
});  
  
// Hash password before saving  
userSchema.pre('save', async function(next) {  
 if (!this.isModified('password')) return next();  
   
 try {  
 const salt = await bcrypt.genSalt(10);  
 this.password = await bcrypt.hash(this.password, salt);  
 next();  
 } catch (err) {  
 next(err);  
 }  
});  
  
// // Method to compare passwords  
// userSchema.methods.comparePassword = async function(candidatePassword) {  
// console.log('Comparing password:', candidatePassword, 'with hashed password:', this.password);  
// return await bcrypt.compare(candidatePassword, this.password);  
// };  
  
userSchema.methods.comparePassword = async function(candidatePassword) {  
 try {  
 const storedHash = this.password;  
 console.log('Stored hash:', storedHash);  
  
 // Extract salt from stored hash  
 const parts = storedHash.split('$');  
 if (parts.length < 4) {  
 throw new Error('Invalid stored hash format');  
 }  
 const saltFromStored = parts[3].substring(0, 22);  
 const fullSalt = `\$${parts[1]}\$${parts[2]}\$${saltFromStored}`;  
  
 // Hash the candidate password with the extracted salt  
 const candidateHash = await bcrypt.hash(candidatePassword, fullSalt);  
 console.log('Candidate hash (using stored salt):', candidateHash);  
  
 // Compare the candidate password with stored hash  
// In your comparePassword method  
const isMatch = await bcrypt.compare(candidatePassword, storedHash);  
console.log('Do they match?', isMatch); // Should be true   
return isMatch;  
 } catch (err) {  
 console.error('Error comparing passwords:', err);  
 throw err;  
 }  
};  
  
module.exports = mongoose.model('User', userSchema);

File Path: .\routes\adminRoutes.js

Content:

const express = require('express');  
const router = express.Router();  
const Order = require('../models/Order');  
const Product = require('../models/Product');  
const User = require('../models/User');  
const adminAuth = require('../middleware/adminAuth');  
  
// Get dashboard stats  
router.get('/dashboard', adminAuth, async (req, res) => {  
 try {  
 const totalProducts = await Product.countDocuments();  
 const totalOrders = await Order.countDocuments();  
 const totalCustomers = await User.countDocuments({ role: 'user' });  
   
 // Calculate total revenue from completed orders  
 const completedOrders = await Order.find({ status: 'completed' });  
 const totalRevenue = completedOrders.reduce((sum, order) => sum + order.totalAmount, 0);  
   
 // Get recent orders  
 const recentOrders = await Order.find()  
 .sort({ createdAt: -1 })  
 .limit(5)  
 .populate('user', 'name email')  
 .lean();  
   
 // Format recent orders  
 const formattedRecentOrders = recentOrders.map(order => ({  
 id: order.\_id,  
 orderNumber: order.orderNumber,  
 customer: order.user.name,  
 date: order.createdAt,  
 total: order.totalAmount,  
 status: order.status  
 }));  
   
 res.json({  
 totalProducts,  
 totalOrders,  
 totalCustomers,  
 totalRevenue,  
 recentOrders: formattedRecentOrders  
 });  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
// Get all orders with pagination  
router.get('/orders', adminAuth, async (req, res) => {  
 try {  
 const page = parseInt(req.query.page) || 1;  
 const limit = parseInt(req.query.limit) || 10;  
 const skip = (page - 1) \* limit;  
   
 let query = {};  
   
 // Filter by status if provided  
 if (req.query.status && req.query.status !== 'all') {  
 query.status = req.query.status;  
 }  
   
 // Search functionality  
 if (req.query.search) {  
 const searchRegex = new RegExp(req.query.search, 'i');  
   
 // Get users matching the search query  
 const users = await User.find({   
 $or: [  
 { name: searchRegex },  
 { email: searchRegex }  
 ]   
 }).select('\_id');  
   
 const userIds = users.map(user => user.\_id);  
   
 query.$or = [  
 { orderNumber: searchRegex },  
 { user: { $in: userIds } }  
 ];  
 }  
   
 // Count total orders matching the query  
 const totalOrders = await Order.countDocuments(query);  
 const totalPages = Math.ceil(totalOrders / limit);  
   
 // Get paginated orders  
 const orders = await Order.find(query)  
 .sort({ createdAt: -1 })  
 .skip(skip)  
 .limit(limit)  
 .populate('user', 'name email phone')  
 .populate('items.product', 'name price image')  
 .lean();  
   
 // Format orders for response  
 const formattedOrders = orders.map(order => ({  
 \_id: order.\_id,  
 orderNumber: order.orderNumber,  
 customer: {  
 name: order.user.name,  
 email: order.user.email,  
 phone: order.user.phone  
 },  
 items: order.items.map(item => ({  
 product: {  
 \_id: item.product.\_id,  
 name: item.product.name,  
 price: item.price,  
 image: item.product.image  
 },  
 quantity: item.quantity,  
 price: item.price  
 })),  
 totalAmount: order.totalAmount,  
 status: order.status,  
 shippingAddress: order.shippingAddress,  
 paymentMethod: order.paymentMethod,  
 createdAt: order.createdAt,  
 updatedAt: order.updatedAt  
 }));  
   
 res.json({  
 orders: formattedOrders,  
 page,  
 totalPages,  
 totalOrders  
 });  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
// Update order status  
router.put('/orders/:id/status', adminAuth, async (req, res) => {  
 try {  
 const { status } = req.body;  
   
 if (!['pending', 'processing', 'shipped', 'completed', 'cancelled'].includes(status)) {  
 return res.status(400).json({ message: 'Invalid status' });  
 }  
   
 const order = await Order.findById(req.params.id);  
   
 if (!order) {  
 return res.status(404).json({ message: 'Order not found' });  
 }  
   
 // If changing to cancelled and was not cancelled before, restore stock  
 if (status === 'cancelled' && order.status !== 'cancelled') {  
 for (const item of order.items) {  
 await Product.findByIdAndUpdate(  
 item.product,  
 { $inc: { stock: item.quantity } }  
 );  
 }  
 }  
   
 // If changing from cancelled to another status, reduce stock again  
 if (order.status === 'cancelled' && status !== 'cancelled') {  
 for (const item of order.items) {  
 await Product.findByIdAndUpdate(  
 item.product,  
 { $inc: { stock: -item.quantity } }  
 );  
 }  
 }  
   
 order.status = status;  
 await order.save();  
   
 res.json(order);  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
// Get all customers  
router.get('/customers', adminAuth, async (req, res) => {  
 try {  
 const page = parseInt(req.query.page) || 1;  
 const limit = parseInt(req.query.limit) || 10;  
 const skip = (page - 1) \* limit;  
   
 let query = { role: 'user' };  
   
 // Search functionality  
 if (req.query.search) {  
 const searchRegex = new RegExp(req.query.search, 'i');  
 query.$or = [  
 { name: searchRegex },  
 { email: searchRegex },  
 { phone: searchRegex }  
 ];  
 }  
   
 // Count total customers matching the query  
 const totalCustomers = await User.countDocuments(query);  
 const totalPages = Math.ceil(totalCustomers / limit);  
   
 // Get paginated customers  
 const customers = await User.find(query)  
 .select('-password')  
 .sort({ createdAt: -1 })  
 .skip(skip)  
 .limit(limit);  
   
 res.json({  
 customers,  
 page,  
 totalPages,  
 totalCustomers  
 });  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
// Generate sales report  
router.get('/reports/sales', adminAuth, async (req, res) => {  
 try {  
 const { startDate, endDate } = req.query;  
   
 // Validate dates  
 const start = startDate ? new Date(startDate) : new Date(new Date().setMonth(new Date().getMonth() - 1));  
 const end = endDate ? new Date(endDate) : new Date();  
   
 // Ensure end date is the end of the day  
 end.setHours(23, 59, 59, 999);  
   
 // Query completed orders within date range  
 const orders = await Order.find({  
 status: 'completed',  
 createdAt: { $gte: start, $lte: end }  
 }).populate('items.product', 'name category');  
   
 // Calculate total revenue  
 const totalRevenue = orders.reduce((sum, order) => sum + order.totalAmount, 0);  
   
 // Calculate sales by category  
 const salesByCategory = {};  
   
 orders.forEach(order => {  
 order.items.forEach(item => {  
 const category = item.product.category;  
 if (!salesByCategory[category]) {  
 salesByCategory[category] = 0;  
 }  
 salesByCategory[category] += item.price \* item.quantity;  
 });  
 });  
   
 // Format sales by date (daily)  
 const salesByDate = {};  
   
 orders.forEach(order => {  
 const dateStr = order.createdAt.toISOString().split('T')[0];  
 if (!salesByDate[dateStr]) {  
 salesByDate[dateStr] = 0;  
 }  
 salesByDate[dateStr] += order.totalAmount;  
 });  
   
 res.json({  
 period: {  
 startDate: start,  
 endDate: end  
 },  
 totalOrders: orders.length,  
 totalRevenue,  
 salesByCategory,  
 salesByDate  
 });  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
module.exports = router;

File Path: .\routes\authRoutes.js

Content:

const express = require('express');  
const router = express.Router();  
const bcrypt = require('bcrypt');  
const jwt = require('jsonwebtoken');  
const User = require('../models/User');  
const auth = require('../middleware/auth');  
  
// Register a new user  
router.post('/register', async (req, res) => {  
 try {  
 const { name, email, password, phone } = req.body;  
   
 // Check if user already exists  
 const existingUser = await User.findOne({ email });  
 if (existingUser) {  
 return res.status(400).json({ message: 'User already exists with this email' });  
 }  
   
 // Create new user  
 const user = new User({  
 name,  
 email,  
 password, // Will be hashed in the model's pre-save hook  
 phone  
 });  
   
 await user.save();  
   
 // Generate JWT token  
 const token = jwt.sign(  
 { userId: user.\_id },  
 process.env.JWT\_SECRET,  
 { expiresIn: process.env.JWT\_EXPIRES\_IN || '7d' }  
 );  
   
 // Return user info (excluding password) and token  
 res.status(201).json({  
 token,  
 user: {  
 \_id: user.\_id,  
 name: user.name,  
 email: user.email,  
 role: user.role  
 }  
 });  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
// Login user  
router.post('/login', async (req, res) => {  
 try {  
 const { email, password } = req.body;  
   
 // Find user by email  
 const user = await User.findOne({ email });  
 if (!user) {  
 return res.status(400).json({ message: 'Invalid email or password' });  
 }  
   
 // Check password  
 const isPasswordValid = await user.comparePassword(password);  
 if (!isPasswordValid) {  
 return res.status(400).json({ message: 'Invalid email or password' });  
 }  
   
 // Generate JWT token  
 const token = jwt.sign(  
 { userId: user.\_id },  
 process.env.JWT\_SECRET,  
 { expiresIn: process.env.JWT\_EXPIRES\_IN || '7d' }  
 );  
   
 // Set cookie (optional)  
 res.cookie('token', token, {  
 httpOnly: true,  
 maxAge: 7 \* 24 \* 60 \* 60 \* 1000, // 7 days  
 sameSite: 'strict'  
 });  
   
 // Return user info and token  
 res.json({  
 token,  
 user: {  
 \_id: user.\_id,  
 name: user.name,  
 email: user.email,  
 role: user.role  
 }  
 });  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
// Get current user  
router.get('/me', auth, async (req, res) => {  
 try {  
 const user = await User.findById(req.user.\_id).select('-password');  
 if (!user) {  
 return res.status(404).json({ message: 'User not found' });  
 }  
   
 res.json(user);  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
// Update user profile  
router.put('/profile', auth, async (req, res) => {  
 try {  
 const { name, email, phone, currentPassword, newPassword } = req.body;  
   
 // Find user  
 const user = await User.findById(req.user.\_id);  
 if (!user) {  
 return res.status(404).json({ message: 'User not found' });  
 }  
   
 // Update basic info  
 if (name) user.name = name;  
 if (email && email !== user.email) {  
 // Check if email already in use by another user  
 const emailExists = await User.findOne({ email, \_id: { $ne: user.\_id } });  
 if (emailExists) {  
 return res.status(400).json({ message: 'Email already in use' });  
 }  
 user.email = email;  
 }  
 if (phone) user.phone = phone;  
   
 // Update password if provided  
 if (currentPassword && newPassword) {  
 const isPasswordValid = await user.comparePassword(currentPassword);  
 if (!isPasswordValid) {  
 return res.status(400).json({ message: 'Current password is incorrect' });  
 }  
   
 user.password = newPassword; // Will be hashed in pre-save hook  
 }  
   
 await user.save();  
   
 // Return updated user (excluding password)  
 res.json({  
 \_id: user.\_id,  
 name: user.name,  
 email: user.email,  
 phone: user.phone,  
 role: user.role  
 });  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
// Logout (clear cookie)  
router.post('/logout', (req, res) => {  
 res.clearCookie('token');  
 res.json({ message: 'Logged out successfully' });  
});  
  
module.exports = router;

File Path: .\routes\orderRoutes.js

Content:

const express = require('express');  
const router = express.Router();  
const Order = require('../models/Order');  
const Product = require('../models/Product');  
const auth = require('../middleware/auth');  
  
// Create a new order  
router.post('/', auth, async (req, res) => {  
 try {  
 const { items, totalAmount, shippingAddress, paymentMethod } = req.body;  
   
 // Validate items and check stock  
 for (const item of items) {  
 const product = await Product.findById(item.product);  
   
 if (!product) {  
 return res.status(404).json({ message: `Product ${item.product} not found` });  
 }  
   
 if (product.stock < item.quantity) {  
 return res.status(400).json({ message: `Insufficient stock for ${product.name}` });  
 }  
 }  
   
 // Create new order  
 const order = new Order({  
 user: req.user.\_id,  
 items,  
 totalAmount,  
 shippingAddress,  
 paymentMethod  
 });  
   
 await order.save();  
   
 // Update product stock  
 for (const item of items) {  
 await Product.findByIdAndUpdate(  
 item.product,  
 { $inc: { stock: -item.quantity } }  
 );  
 }  
   
 // Return created order  
 res.status(201).json(order);  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
// Get orders for current user  
router.get('/', auth, async (req, res) => {  
 try {  
 const orders = await Order.find({ user: req.user.\_id })  
 .sort({ createdAt: -1 })  
 .populate('items.product', 'name nameAr price image');  
   
 res.json(orders);  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
// Get order details  
router.get('/:id', auth, async (req, res) => {  
 try {  
 const order = await Order.findById(req.params.id)  
 .populate('items.product', 'name nameAr price image')  
 .populate('user', 'name email phone');  
   
 if (!order) {  
 return res.status(404).json({ message: 'Order not found' });  
 }  
   
 // Check if the order belongs to the current user or if the user is an admin  
 if (order.user.\_id.toString() !== req.user.\_id.toString() && req.user.role !== 'admin') {  
 return res.status(403).json({ message: 'Not authorized to access this order' });  
 }  
   
 res.json(order);  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
// Cancel an order  
router.put('/:id/cancel', auth, async (req, res) => {  
 try {  
 const order = await Order.findById(req.params.id);  
   
 if (!order) {  
 return res.status(404).json({ message: 'Order not found' });  
 }  
   
 // Check if the order belongs to the current user  
 if (order.user.toString() !== req.user.\_id.toString() && req.user.role !== 'admin') {  
 return res.status(403).json({ message: 'Not authorized to cancel this order' });  
 }  
   
 // Check if order can be cancelled (only pending or processing orders)  
 if (!['pending', 'processing'].includes(order.status)) {  
 return res.status(400).json({ message: 'Order cannot be cancelled at this stage' });  
 }  
   
 // Update order status  
 order.status = 'cancelled';  
 await order.save();  
   
 // Restore product stock  
 for (const item of order.items) {  
 await Product.findByIdAndUpdate(  
 item.product,  
 { $inc: { stock: item.quantity } }  
 );  
 }  
   
 res.json(order);  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
module.exports = router;

File Path: .\routes\productRoutes.js

Content:

const express = require('express');  
const router = express.Router();  
const multer = require('multer');  
const { v4: uuidv4 } = require('uuid');  
const path = require('path');  
const Product = require('../models/Product');  
const auth = require('../middleware/auth');  
const adminAuth = require('../middleware/adminAuth');  
  
// Configure multer for file uploads  
const storage = multer.diskStorage({  
 destination: (req, file, cb) => {  
 cb(null, 'uploads/products/');  
 },  
 filename: (req, file, cb) => {  
 const uniqueName = `${uuidv4()}${path.extname(file.originalname)}`;  
 cb(null, uniqueName);  
 }  
});  
  
const upload = multer({   
 storage,  
 limits: { fileSize: 5 \* 1024 \* 1024 }, // 5MB limit  
 fileFilter: (req, file, cb) => {  
 const filetypes = /jpeg|jpg|png|webp|jfif/;  
 const mimetype = filetypes.test(file.mimetype);  
 const extname = filetypes.test(path.extname(file.originalname).toLowerCase());  
   
 if (mimetype && extname) {  
 return cb(null, true);  
 }  
   
 cb(new Error('Only image files are allowed!'));  
 }  
});  
  
// Get all products  
router.get('/', async (req, res) => {  
 try {  
 const { category, search, featured } = req.query;  
 const query = {};  
   
 if (category && category !== 'all') {  
 query.category = category;  
 }  
   
 if (search) {  
 query.$or = [  
 { name: { $regex: search, $options: 'i' } },  
 { nameAr: { $regex: search, $options: 'i' } },  
 { description: { $regex: search, $options: 'i' } },  
 { descriptionAr: { $regex: search, $options: 'i' } }  
 ];  
 }  
   
 if (featured) {  
 query.featured = true;  
 }  
   
 const products = await Product.find(query).sort({ createdAt: -1 });  
 res.json(products);  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
// Get single product  
router.get('/:id', async (req, res) => {  
 try {  
 const product = await Product.findById(req.params.id);  
   
 if (!product) {  
 return res.status(404).json({ message: 'Product not found' });  
 }  
   
 res.json(product);  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
// Create product (admin only)  
router.post('/', adminAuth, upload.fields([  
 { name: 'image', maxCount: 1 },  
 { name: 'boxImage', maxCount: 1 }  
]), async (req, res) => {  
 try {  
 const { name, nameAr, price, category, description, descriptionAr, stock } = req.body;  
   
 if (!req.files.image || !req.files.boxImage) {  
 return res.status(400).json({ message: 'Product image and box image are required' });  
 }  
   
 const newProduct = new Product({  
 name,  
 nameAr,  
 price,  
 category,  
 description,  
 descriptionAr,  
 stock: stock || 0,  
 image: `/uploads/products/${req.files.image[0].filename}`,  
 boxImage: `/uploads/products/${req.files.boxImage[0].filename}`  
 });  
   
 const savedProduct = await newProduct.save();  
 res.status(201).json(savedProduct);  
 } catch (err) {  
 res.status(400).json({ message: err.message });  
 }  
});  
  
// Update product (admin only)  
router.put('/:id', adminAuth, upload.fields([  
 { name: 'image', maxCount: 1 },  
 { name: 'boxImage', maxCount: 1 }  
]), async (req, res) => {  
 try {  
 const { name, nameAr, price, category, description, descriptionAr, stock, featured } = req.body;  
 const updates = {  
 name,  
 nameAr,  
 price,  
 category,  
 description,  
 descriptionAr,  
 updatedAt: Date.now()  
 };  
   
 if (stock !== undefined) {  
 updates.stock = stock;  
 }  
   
 if (featured !== undefined) {  
 updates.featured = featured;  
 }  
   
 if (req.files.image) {  
 updates.image = `/uploads/products/${req.files.image[0].filename}`;  
 }  
   
 if (req.files.boxImage) {  
 updates.boxImage = `/uploads/products/${req.files.boxImage[0].filename}`;  
 }  
   
 const updatedProduct = await Product.findByIdAndUpdate(  
 req.params.id,  
 updates,  
 { new: true }  
 );  
   
 if (!updatedProduct) {  
 return res.status(404).json({ message: 'Product not found' });  
 }  
   
 res.json(updatedProduct);  
 } catch (err) {  
 res.status(400).json({ message: err.message });  
 }  
});  
  
// Delete product (admin only)  
router.delete('/:id', adminAuth, async (req, res) => {  
 try {  
 const deletedProduct = await Product.findByIdAndDelete(req.params.id);  
   
 if (!deletedProduct) {  
 return res.status(404).json({ message: 'Product not found' });  
 }  
   
 res.json({ message: 'Product deleted successfully' });  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
module.exports = router;

File Path: .\scripts\create-admin.js

Content:

const mongoose = require('mongoose');  
const User = require('../models/User');  
require('dotenv').config();  
  
mongoose.connect(process.env.MONGO\_URI || 'mongodb://localhost:27017/perfume-store')  
 .then(() => console.log('MongoDB connected'))  
 .catch(err => {  
 console.error('MongoDB connection error:', err);  
 process.exit(1);  
 });  
  
const adminData = {  
 name: 'Admin User',  
 email: 'admin@example.com',  
 password: 'Admin@123', // Plain password - will be hashed by model  
 role: 'admin',  
 phone: '+123456789'  
};  
  
async function createAdmin() {  
 try {  
 // Delete existing admin if any (to start fresh)  
 await User.deleteOne({ email: adminData.email });  
 console.log('Cleaned up any existing admin user');  
   
 // Create new admin user (let the model handle hashing)  
 const admin = new User(adminData);  
 await admin.save();  
   
 console.log('Admin user created successfully!');  
 console.log(`Email: ${adminData.email}`);  
 console.log(`Password: ${adminData.password}`);  
 } catch (err) {  
 console.error('Error creating admin user:', err);  
 } finally {  
 mongoose.disconnect();  
 }  
}  
  
createAdmin();

File Path: .\uploads\products\7a1cfcd0-a618-42fe-8c10-233094697ce8.jfif

Could not read file .\uploads\products\7a1cfcd0-a618-42fe-8c10-233094697ce8.jfif: 'utf-8' codec can't decode byte 0xff in position 0: invalid start byte

File Path: .\uploads\products\dbe2a02d-3fbd-45ae-b297-0f85b8507110.jfif

Could not read file .\uploads\products\dbe2a02d-3fbd-45ae-b297-0f85b8507110.jfif: 'utf-8' codec can't decode byte 0xff in position 0: invalid start byte