GoBazar Backend Development Workflow

Project Overview

This document outlines the complete step-by-step workflow for building a Node.js/Express backend for the GoBazar e-commerce platform, migrating from the current Next.js mock API implementation to a robust PostgreSQL-based backend system.

Technology Stack

- **Runtime**: Node.js (v18+)
- **Framework**: Express.js
- **Database**: PostgreSQL
- **ORM**: Prisma
- **Authentication**: JWT + Mail-based OTP verification
- **Validation**: Joi
- **File Upload**: Multer + Cloudinary
- **OTP Service**: Nodemailer + SMTP
- **Payment Gateways**: RMC, SabPaisa, PayU Money
- **Testing**: Jest + Supertest
- **Documentation**: Swagger/OpenAPI

Current Frontend Analysis

Based on the existing frontend codebase, we have identified the following data models and API endpoints:

Data Models

- **Users**: Authentication, profiles, addresses, roles (user/admin)
- **Categories**: Hierarchical category structure with subcategories
- **Products**: Complex product catalog with variants, pricing, inventory
- **Orders**: Order management with status tracking, delivery slots
- **Cart**: Shopping cart functionality
- **Coupons**: Discount and promotional codes

Current Mock API Endpoints

- `/api/auth/send-otp` - Send OTP to email for authentication
- `/api/auth/verify-otp` - Verify OTP and authenticate user
- `/api/products` - Product catalog
- `/api/categories` - Category management
- `/api/orders` - Order management
- `/api/subcategories` - Subcategory management
- `/api/recommendations` - Product recommendations

Development Phases

Phase 1: Project Setup & Infrastructure (Week 1)

Step 1.1: Initialize Backend Project

```
# Create new directory
mkdir gobazar-backend
cd gobazar-backend

# Initialize Node.js project
npm init -y

# Install core dependencies
npm install express cors helmet morgan compression
npm install dotenv jsonwebtoken
npm install prisma @prisma/client
npm install joi express-rate-limit express-validator
npm install multer cloudinary
npm install nodemailer
npm install axios

# Install development dependencies
npm install -D nodemon @types/node @types/express
npm install -D @types/jsonwebtoken @types/nodemailer
npm install -D @types/cors @types/morgan @types/compression
npm install -D jest supertest @types/jest @types/supertest
npm install -D eslint prettier husky lint-staged
```

Step 1.2: Project Structure

```
gobazar-backend/
  src/
      controllers/     # Route handlers
      middleware/      # Custom middleware
      routes/          # API routes
      services/        # Business logic
      utils/           # Utility functions
      config/          # Configuration files
      types/           # TypeScript definitions
      tests/           # Test files
  prisma/              # Database schema and migrations
  uploads/             # File uploads (temporary)
  docs/                # API documentation
  .env                 # Environment variables
  .env.example         # Environment template
  .gitignore
```

```
package.json
tsconfig.json
README.md
```

Step 1.3: Environment Configuration Create `.env.example`:

```
# Server Configuration
PORT=5000
NODE_ENV=development

# Database Configuration
DATABASE_URL="postgresql://username:password@localhost:5432/gobazar_db"

# JWT Configuration
JWT_SECRET=your-super-secret-jwt-key
JWT_EXPIRES_IN=7d

# File Upload Configuration
CLOUDINARY_CLOUD_NAME=your-cloud-name
CLOUDINARY_API_KEY=your-api-key
CLOUDINARY_API_SECRET=your-api-secret

# Email Configuration
SMTP_HOST=smtp.gmail.com
SMTP_PORT=587
SMTP_USER=your-email@gmail.com
SMTP_PASS=your-app-password

# OTP Configuration
OTP_EXPIRY_MINUTES=5
OTP_LENGTH=6

# Payment Gateway Configuration
# RMC Payment Gateway
RMC_MERCHANT_ID=your-rmc-merchant-id
RMC_SECRET_KEY=your-rmc-secret-key
RMC_API_URL=https://api.rmc.com

# SabPaisa Payment Gateway
SABPAISA_MERCHANT_ID=your-sabpaisa-merchant-id
SABPAISA_SECRET_KEY=your-sabpaisa-secret-key
SABPAISA_API_URL=https://api.sabpaisa.com

# PayU Money Payment Gateway
PAYU_MERCHANT_KEY=your-payu-merchant-key
PAYU_MERCHANT_SALT=your-payu-merchant-salt
```

```
PAYU_API_URL=https://api.payu.in
```

Step 1.4: Basic Express Server Create `src/app.ts`:

```typescript
import express from 'express';
import cors from 'cors';
import helmet from 'helmet';
import morgan from 'morgan';
import compression from 'compression';
import rateLimit from 'express-rate-limit';

const app = express();

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

// Rate limiting
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100 // limit each IP to 100 requests per windowMs
});
app.use(limiter);

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

// Compression middleware
app.use(compression());

// Logging middleware
app.use(morgan('combined'));

// Health check endpoint
app.get('/health', (req, res) => {
  res.status(200).json({ status: 'OK', timestamp: new Date().toISOString() });
});

export default app;
```

Phase 2: Database Design & Setup (Week 1-2)

Step 2.1: PostgreSQL Database Setup

```
# Install PostgreSQL (Ubuntu/Debian)
sudo apt update
sudo apt install postgresql postgresql-contrib

# Create database and user
sudo -u postgres psql
CREATE DATABASE gobazar_db;
CREATE USER gobazar_user WITH PASSWORD 'secure_password';
GRANT ALL PRIVILEGES ON DATABASE gobazar_db TO gobazar_user;
\q
```

Step 2.2: Prisma Schema Design Create `prisma/schema.prisma`:

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id         String   @id @default(cuid())
  email      String   @unique
  name       String?
  phone      String?
  role       Role     @default(USER)
  isVerified Boolean @default(false)
  addresses Address[]
  orders    Order[]
  cartItems CartItem[]
  wishlist  WishlistItem[]
  otpCodes  OTPCode[]
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  @@map("users")
}

model OTPCode {
  id         String   @id @default(cuid())
  userId     String?
```

```
  email     String
  code      String
  type      OTPType  @default(LOGIN)
  isUsed    Boolean  @default(false)
  expiresAt DateTime
  createdAt DateTime @default(now())
  user      User?    @relation(fields: [userId], references: [id], onDelete: Cascade)

  @@map("otp_codes")
}

model Address {
  id        String  @id @default(cuid())
  userId    String
  type      AddressType
  street    String
  city      String
  state     String
  pincode   String
  landmark  String?
  isDefault Boolean @default(false)
  user      User    @relation(fields: [userId], references: [id], onDelete: Cascade)

  @@map("addresses")
}

model Category {
  id           String        @id @default(cuid())
  name         String
  slug         String        @unique
  image        String?
  order        Int
  subcategories SubCategory[]
  products     Product[]
  createdAt    DateTime      @default(now())
  updatedAt    DateTime      @updatedAt

  @@map("categories")
}

model SubCategory {
  id         String    @id @default(cuid())
  categoryId String
  name       String
  slug       String    @unique
  description String?
```

```
  order      Int
  category   Category  @relation(fields: [categoryId], references: [id], onDelete: Cascade)
  products   Product[]
  createdAt  DateTime  @default(now())
  updatedAt  DateTime  @updatedAt

  @@map("subcategories")
}

model Product {
  id              String           @id @default(cuid())
  name            String
  brand           String
  categoryId      String
  subcategoryId   String?
  price           Decimal          @db.Decimal(10, 2)
  mrp             Decimal          @db.Decimal(10, 2)
  discountPercent Int              @default(0)
  images          String[]
  unit            String
  stock           Int              @default(0)
  description     String
  highlights      String[]
  rating          Decimal          @db.Decimal(2, 1) @default(0)
  reviewCount     Int              @default(0)
  tags            String[]
  nutritionalInfo String?
  ingredients     String?
  benefits        String?
  variants        ProductVariant[]
  cartItems       CartItem[]
  orderItems      OrderItem[]
  wishlistItems   WishlistItem[]
  category        Category         @relation(fields: [categoryId], references: [id])
  subcategory     SubCategory?     @relation(fields: [subcategoryId], references: [id])
  createdAt       DateTime         @default(now())
  updatedAt       DateTime         @updatedAt

  @@map("products")
}

model ProductVariant {
  id        String  @id @default(cuid())
  productId String
  name      String
  unit      String
```

```
  price     Decimal @db.Decimal(10, 2)
  mrp       Decimal @db.Decimal(10, 2)
  stock     Int     @default(0)
  size      String?
  weight    String?
  product   Product @relation(fields: [productId], references: [id], onDelete: Cascade)

  @@map("product_variants")
}

model CartItem {
  id        String  @id @default(cuid())
  userId    String
  productId String
  variantId String?
  quantity  Int
  user      User    @relation(fields: [userId], references: [id], onDelete: Cascade)
  product   Product @relation(fields: [productId], references: [id], onDelete: Cascade)
  variant   ProductVariant? @relation(fields: [variantId], references: [id], onDelete: Casca

  @@unique([userId, productId, variantId])
  @@map("cart_items")
}

model Order {
  id           String      @id @default(cuid())
  userId       String
  orderNumber   String      @unique
  status       OrderStatus @default(RECEIVED)
  total        Decimal     @db.Decimal(10, 2)
  subtotal     Decimal     @db.Decimal(10, 2)
  discount     Decimal     @db.Decimal(10, 2) @default(0)
  deliveryFee  Decimal     @db.Decimal(10, 2) @default(0)
  taxes        Decimal     @db.Decimal(10, 2) @default(0)
  deliverySlot String
  couponCode   String?
  address      Json        // Store address as JSON
  paymentMethod PaymentMethod?
  paymentStatus PaymentStatus @default(PENDING)
  paymentGateway String?
  paymentId    String?
  items        OrderItem[]
  user         User        @relation(fields: [userId], references: [id])
  createdAt    DateTime    @default(now())
  updatedAt    DateTime    @updatedAt
```

```
  @@map("orders")
}

model OrderItem {
  id        String  @id @default(cuid())
  orderId   String
  productId String
  variantId String?
  quantity  Int
  price     Decimal @db.Decimal(10, 2)
  name      String
  image     String
  unit      String
  order     Order   @relation(fields: [orderId], references: [id], onDelete: Cascade)
  product   Product @relation(fields: [productId], references: [id])

  @@map("order_items")
}

model Coupon {
  id            String      @id @default(cuid())
  code          String      @unique
  description   String
  discountType  DiscountType
  discountValue Decimal     @db.Decimal(10, 2)
  minOrderValue Decimal     @db.Decimal(10, 2)
  maxDiscount   Decimal?    @db.Decimal(10, 2)
  validFrom     DateTime
  validTo       DateTime
  isActive      Boolean     @default(true)
  createdAt     DateTime    @default(now())
  updatedAt     DateTime    @updatedAt

  @@map("coupons")
}

model WishlistItem {
  id        String  @id @default(cuid())
  userId    String
  productId String
  user      User    @relation(fields: [userId], references: [id], onDelete: Cascade)
  product   Product @relation(fields: [productId], references: [id], onDelete: Cascade)

  @@unique([userId, productId])
  @@map("wishlist_items")
}
```

```
enum Role {
  USER
  ADMIN
}

enum AddressType {
  HOME
  WORK
  OTHER
}

enum OrderStatus {
  RECEIVED
  PACKING
  ON_THE_WAY
  DELIVERED
  CANCELED
}

enum DiscountType {
  PERCENTAGE
  FIXED
}

enum OTPType {
  LOGIN
  VERIFICATION
}

enum PaymentMethod {
  CARD
  UPI
  NET_BANKING
  WALLET
  COD
}

enum PaymentStatus {
  PENDING
  SUCCESS
  FAILED
  CANCELLED
  REFUNDED
}
```

Step 2.3: Database Migration & Seeding

```
# Generate Prisma client
npx prisma generate

# Create and run migration
npx prisma migrate dev --name init

# Seed database with initial data
npx prisma db seed
```

Phase 3: Authentication & Authorization (Week 2)

Step 3.1: Authentication Middleware Create `src/middleware/auth.ts`:

```
import { Request, Response, NextFunction } from 'express';
import jwt from 'jsonwebtoken';
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

interface AuthRequest extends Request {
  user?: any;
}

export const authenticateToken = async (
  req: AuthRequest,
  res: Response,
  next: NextFunction
) => {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1];

  if (!token) {
    return res.status(401).json({ message: 'Access token required' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET!) as any;
    const user = await prisma.user.findUnique({
      where: { id: decoded.userId },
      select: { id: true, email: true, name: true, role: true }
    });

    if (!user) {
      return res.status(401).json({ message: 'Invalid token' });
    }
```

```
    req.user = user;
    next();
  } catch (error) {
    return res.status(403).json({ message: 'Invalid or expired token' });
  }
};

export const requireAdmin = (req: AuthRequest, res: Response, next: NextFunction) => {
  if (req.user?.role !== 'ADMIN') {
    return res.status(403).json({ message: 'Admin access required' });
  }
  next();
};
```

Step 3.2: OTP Service Create `src/services/otpService.ts`:

```
import nodemailer from 'nodemailer';
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

const transporter = nodemailer.createTransporter({
  host: process.env.SMTP_HOST,
  port: Number(process.env.SMTP_PORT),
  secure: false,
  auth: {
    user: process.env.SMTP_USER,
    pass: process.env.SMTP_PASS,
  },
});

export const generateOTP = (): string => {
  const length = Number(process.env.OTP_LENGTH) || 6;
  return Math.floor(100000 + Math.random() * 900000).toString().slice(0, length);
};

export const sendOTP = async (email: string, type: 'LOGIN' | 'VERIFICATION' = 'LOGIN') => {
  try {
    const code = generateOTP();
    const expiryMinutes = Number(process.env.OTP_EXPIRY_MINUTES) || 5;
    const expiresAt = new Date(Date.now() + expiryMinutes * 60 * 1000);

    // Store OTP in database
    await prisma.oTPCode.create({
      data: {
```

```
        email,
        code,
        type,
        expiresAt,
      },
    });

    // Send email
    const mailOptions = {
      from: process.env.SMTP_USER,
      to: email,
      subject: 'GoBazar - Your OTP Code',
      html: `
        <div style="font-family: Arial, sans-serif; max-width: 600px; margin: 0 auto;">
          <h2 style="color: #2563eb;">GoBazar Authentication</h2>
          <p>Your OTP code is: <strong style="font-size: 24px; color: #059669;">${code}</str
          <p>This code will expire in ${expiryMinutes} minutes.</p>
          <p>If you didn't request this code, please ignore this email.</p>
        </div>
      `,
    };

    await transporter.sendMail(mailOptions);
    return { success: true, message: 'OTP sent successfully' };
  } catch (error) {
    console.error('Send OTP error:', error);
    throw new Error('Failed to send OTP');
  }
};

export const verifyOTP = async (email: string, code: string) => {
  try {
    const otpRecord = await prisma.oTPCode.findFirst({
      where: {
        email,
        code,
        isUsed: false,
        expiresAt: { gt: new Date() },
      },
      orderBy: { createdAt: 'desc' },
    });

    if (!otpRecord) {
      return { success: false, message: 'Invalid or expired OTP' };
    }
```

```
    // Mark OTP as used
    await prisma.oTPCode.update({
      where: { id: otpRecord.id },
      data: { isUsed: true },
    });

    return { success: true, message: 'OTP verified successfully' };
  } catch (error) {
    console.error('Verify OTP error:', error);
    throw new Error('Failed to verify OTP');
  }
};
```

Step 3.3: Auth Controller Create `src/controllers/authController.ts`:

```
import { Request, Response } from 'express';
import jwt from 'jsonwebtoken';
import { PrismaClient } from '@prisma/client';
import Joi from 'joi';
import { sendOTP, verifyOTP } from '../services/otpService';

const prisma = new PrismaClient();

const sendOTPSchema = Joi.object({
  email: Joi.string().email().required(),
});

const verifyOTPSchema = Joi.object({
  email: Joi.string().email().required(),
  code: Joi.string().length(6).pattern(/^\d+$/).required(),
  name: Joi.string().min(2).max(50).optional(),
  phone: Joi.string().pattern(/^[6-9]\d{9}$/).optional(),
});

export const sendOTPForLogin = async (req: Request, res: Response) => {
  try {
    const { error, value } = sendOTPSchema.validate(req.body);
    if (error) {
      return res.status(400).json({ message: error.details[0].message });
    }

    const { email } = value;

    // Send OTP
    await sendOTP(email, 'LOGIN');
```

```
    res.json({
      success: true,
      message: 'OTP sent to your email address',
    });
  } catch (error) {
    console.error('Send OTP error:', error);
    res.status(500).json({ message: 'Failed to send OTP' });
  }
};

export const verifyOTPAndLogin = async (req: Request, res: Response) => {
  try {
    const { error, value } = verifyOTPSchema.validate(req.body);
    if (error) {
      return res.status(400).json({ message: error.details[0].message });
    }

    const { email, code, name, phone } = value;

    // Verify OTP
    const otpResult = await verifyOTP(email, code);
    if (!otpResult.success) {
      return res.status(400).json({ message: otpResult.message });
    }

    // Find or create user
    let user = await prisma.user.findUnique({
      where: { email },
      select: {
        id: true,
        name: true,
        email: true,
        phone: true,
        role: true,
        isVerified: true,
        createdAt: true,
      },
    });

    if (!user) {
      // Create new user if doesn't exist
      user = await prisma.user.create({
        data: {
          email,
          name: name || null,
          phone: phone || null,
```

```
          role: 'USER',
          isVerified: true,
        },
        select: {
          id: true,
          name: true,
          email: true,
          phone: true,
          role: true,
          isVerified: true,
          createdAt: true,
        },
      });
    } else {
      // Update user verification status
      await prisma.user.update({
        where: { id: user.id },
        data: { isVerified: true },
      });
    }

    // Generate JWT token
    const token = jwt.sign(
      { userId: user.id, email: user.email },
      process.env.JWT_SECRET!,
      { expiresIn: process.env.JWT_EXPIRES_IN || '7d' }
    );

    res.json({
      success: true,
      user,
      token,
      message: 'Login successful',
    });
  } catch (error) {
    console.error('Verify OTP error:', error);
    res.status(500).json({ message: 'Internal server error' });
  }
};
```

Phase 4: Core API Development (Week 3-4)

Step 4.1: Product Management APIs Create `src/controllers/productController.ts`:

```
import { Request, Response } from 'express';
import { PrismaClient } from '@prisma/client';
```

```
import Joi from 'joi';

const prisma = new PrismaClient();

export const getProducts = async (req: Request, res: Response) => {
  try {
    const { page = 1, limit = 20, category, search, sort = 'createdAt', order = 'desc' } = r

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

    const where: any = {};

    if (category) {
      where.categoryId = category;
    }

    if (search) {
      where.OR = [
        { name: { contains: search as string, mode: 'insensitive' } },
        { brand: { contains: search as string, mode: 'insensitive' } },
        { tags: { has: search as string } }
      ];
    }

    const [products, total] = await Promise.all([
      prisma.product.findMany({
        where,
        include: {
          category: true,
          subcategory: true,
          variants: true
        },
        skip,
        take: Number(limit),
        orderBy: { [sort as string]: order }
      }),
      prisma.product.count({ where })
    ]);

    res.json({
      success: true,
      data: products,
      pagination: {
        page: Number(page),
        limit: Number(limit),
        total,
```

```
        pages: Math.ceil(total / Number(limit))
      }
    });
  } catch (error) {
    console.error('Get products error:', error);
    res.status(500).json({ message: 'Internal server error' });
  }
};

export const getProductById = async (req: Request, res: Response) => {
  try {
    const { id } = req.params;

    const product = await prisma.product.findUnique({
      where: { id },
      include: {
        category: true,
        subcategory: true,
        variants: true
      }
    });

    if (!product) {
      return res.status(404).json({ message: 'Product not found' });
    }

    res.json({
      success: true,
      data: product
    });
  } catch (error) {
    console.error('Get product error:', error);
    res.status(500).json({ message: 'Internal server error' });
  }
};
```

Step 4.2: Order Management APIs Create `src/controllers/orderController.ts`:

```
import { Request, Response } from 'express';
import { PrismaClient } from '@prisma/client';
import Joi from 'joi';

const prisma = new PrismaClient();

export const createOrder = async (req: Request, res: Response) => {
  try {
```

```
const orderSchema = Joi.object({
  items: Joi.array().items(Joi.object({
    productId: Joi.string().required(),
    variantId: Joi.string().optional(),
    quantity: Joi.number().integer().min(1).required()
  })).min(1).required(),
  address: Joi.object().required(),
  deliverySlot: Joi.string().required(),
  couponCode: Joi.string().optional()
});

const { error, value } = orderSchema.validate(req.body);
if (error) {
  return res.status(400).json({ message: error.details[0].message });
}

const userId = req.user.id;
const { items, address, deliverySlot, couponCode } = value;

// Calculate totals
let subtotal = 0;
const orderItems = [];

for (const item of items) {
  const product = await prisma.product.findUnique({
    where: { id: item.productId },
    include: { variants: true }
  });

  if (!product) {
    return res.status(400).json({ message: `Product ${item.productId} not found` });
  }

  let price = product.price;
  if (item.variantId) {
    const variant = product.variants.find(v => v.id === item.variantId);
    if (!variant) {
      return res.status(400).json({ message: `Variant ${item.variantId} not found` });
    }
    price = variant.price;
  }

  const itemTotal = price * item.quantity;
  subtotal += itemTotal;

  orderItems.push({
```

```
    productId: item.productId,
    variantId: item.variantId,
    quantity: item.quantity,
    price,
    name: product.name,
    image: product.images[0],
    unit: product.unit
  });
}

// Apply coupon if provided
let discount = 0;
if (couponCode) {
  const coupon = await prisma.coupon.findUnique({
    where: { code: couponCode }
  });

  if (coupon && coupon.isActive && new Date() >= coupon.validFrom && new Date() <= coupo
    if (subtotal >= coupon.minOrderValue) {
      if (coupon.discountType === 'PERCENTAGE') {
        discount = (subtotal * coupon.discountValue) / 100;
        if (coupon.maxDiscount) {
          discount = Math.min(discount, coupon.maxDiscount);
        }
      } else {
        discount = coupon.discountValue;
      }
    }
  }
}

const deliveryFee = subtotal >= 500 ? 0 : 50; // Free delivery above 500
const taxes = (subtotal - discount) * 0.18; // 18% GST
const total = subtotal - discount + deliveryFee + taxes;

// Generate order number
const orderNumber = `GB${Date.now()}${Math.random().toString(36).substr(2, 5).toUpperCas

// Create order
const order = await prisma.order.create({
  data: {
    userId,
    orderNumber,
    status: 'RECEIVED',
    total,
    subtotal,
```

```
        discount,
        deliveryFee,
        taxes,
        deliverySlot,
        couponCode,
        address: address as any,
        items: {
          create: orderItems
        }
      },
      include: {
        items: true,
        user: {
          select: { id: true, name: true, email: true, phone: true }
        }
      }
    });

    // Clear user's cart
    await prisma.cartItem.deleteMany({
      where: { userId }
    });

    res.status(201).json({
      success: true,
      data: order
    });
  } catch (error) {
    console.error('Create order error:', error);
    res.status(500).json({ message: 'Internal server error' });
  }
};
```

Phase 5: Payment Gateway Integration (Week 5)

Step 5.1: Payment Gateway Service Create `src/services/paymentService.ts`:

```
import axios from 'axios';
import crypto from 'crypto';

interface PaymentRequest {
  orderId: string;
  amount: number;
  customerEmail: string;
  customerPhone: string;
  customerName: string;
```

```
  returnUrl: string;
}

interface PaymentResponse {
  success: boolean;
  paymentUrl?: string;
  paymentId?: string;
  message?: string;
}

export class RMCPaymentService {
  private merchantId: string;
  private secretKey: string;
  private apiUrl: string;

  constructor() {
    this.merchantId = process.env.RMC_MERCHANT_ID!;
    this.secretKey = process.env.RMC_SECRET_KEY!;
    this.apiUrl = process.env.RMC_API_URL!;
  }

  async initiatePayment(paymentData: PaymentRequest): Promise<PaymentResponse> {
    try {
      const payload = {
        merchant_id: this.merchantId,
        order_id: paymentData.orderId,
        amount: paymentData.amount,
        currency: 'INR',
        customer_email: paymentData.customerEmail,
        customer_phone: paymentData.customerPhone,
        customer_name: paymentData.customerName,
        return_url: paymentData.returnUrl,
        timestamp: Date.now().toString(),
      };

      // Generate signature
      const signature = this.generateSignature(payload);
      payload.signature = signature;

      const response = await axios.post(`${this.apiUrl}/initiate`, payload);

      return {
        success: true,
        paymentUrl: response.data.payment_url,
        paymentId: response.data.payment_id,
      };
```

```
    } catch (error) {
      console.error('RMC Payment initiation error:', error);
      return {
        success: false,
        message: 'Failed to initiate RMC payment',
      };
    }
  }

  private generateSignature(payload: any): string {
    const sortedKeys = Object.keys(payload).sort();
    const queryString = sortedKeys
      .map(key => `${key}=${payload[key]}`)
      .join('&');

    return crypto
      .createHmac('sha256', this.secretKey)
      .update(queryString)
      .digest('hex');
  }

  async verifyPayment(paymentId: string): Promise<boolean> {
    try {
      const response = await axios.get(`${this.apiUrl}/verify/${paymentId}`);
      return response.data.status === 'success';
    } catch (error) {
      console.error('RMC Payment verification error:', error);
      return false;
    }
  }
}

export class SabPaisaPaymentService {
  private merchantId: string;
  private secretKey: string;
  private apiUrl: string;

  constructor() {
    this.merchantId = process.env.SABPAISA_MERCHANT_ID!;
    this.secretKey = process.env.SABPAISA_SECRET_KEY!;
    this.apiUrl = process.env.SABPAISA_API_URL!;
  }

  async initiatePayment(paymentData: PaymentRequest): Promise<PaymentResponse> {
    try {
      const payload = {
```

```
      merchantId: this.merchantId,
      orderId: paymentData.orderId,
      amount: paymentData.amount,
      currency: 'INR',
      customerEmail: paymentData.customerEmail,
      customerPhone: paymentData.customerPhone,
      customerName: paymentData.customerName,
      returnUrl: paymentData.returnUrl,
      timestamp: Date.now().toString(),
    };

    // Generate checksum
    const checksum = this.generateChecksum(payload);
    payload.checksum = checksum;

    const response = await axios.post(`${this.apiUrl}/payment/initiate`, payload);

    return {
      success: true,
      paymentUrl: response.data.paymentUrl,
      paymentId: response.data.paymentId,
    };
  } catch (error) {
    console.error('SabPaisa Payment initiation error:', error);
    return {
      success: false,
      message: 'Failed to initiate SabPaisa payment',
    };
  }
}

private generateChecksum(payload: any): string {
  const sortedKeys = Object.keys(payload).sort();
  const queryString = sortedKeys
    .map(key => `${key}=${payload[key]}`)
    .join('&');

  return crypto
    .createHash('sha256')
    .update(queryString + this.secretKey)
    .digest('hex');
}

async verifyPayment(paymentId: string): Promise<boolean> {
  try {
    const response = await axios.get(`${this.apiUrl}/payment/verify/${paymentId}`);
```

```
      return response.data.status === 'SUCCESS';
    } catch (error) {
      console.error('SabPaisa Payment verification error:', error);
      return false;
    }
  }
}

export class PayUPaymentService {
  private merchantKey: string;
  private merchantSalt: string;
  private apiUrl: string;

  constructor() {
    this.merchantKey = process.env.PAYU_MERCHANT_KEY!;
    this.merchantSalt = process.env.PAYU_MERCHANT_SALT!;
    this.apiUrl = process.env.PAYU_API_URL!;
  }

  async initiatePayment(paymentData: PaymentRequest): Promise<PaymentResponse> {
    try {
      const payload = {
        key: this.merchantKey,
        txnid: paymentData.orderId,
        amount: paymentData.amount,
        productinfo: 'GoBazar Order',
        firstname: paymentData.customerName,
        email: paymentData.customerEmail,
        phone: paymentData.customerPhone,
        surl: paymentData.returnUrl,
        furl: paymentData.returnUrl,
        hash: '',
      };

      // Generate hash
      const hashString = `${this.merchantKey}|${payload.txnid}|${payload.amount}|${payload.p
      payload.hash = crypto.createHash('sha512').update(hashString).digest('hex');

      const response = await axios.post(`${this.apiUrl}/_payment`, payload);

      return {
        success: true,
        paymentUrl: response.data.payment_url,
        paymentId: payload.txnid,
      };
    } catch (error) {
```

```
      console.error('PayU Payment initiation error:', error);
      return {
        success: false,
        message: 'Failed to initiate PayU payment',
      };
    }
  }

  async verifyPayment(paymentData: any): Promise<boolean> {
    try {
      const hashString = `${this.merchantSalt}|${paymentData.status}||||||||||${paymentData
      const hash = crypto.createHash('sha512').update(hashString).digest('hex');

      return hash === paymentData.hash && paymentData.status === 'success';
    } catch (error) {
      console.error('PayU Payment verification error:', error);
      return false;
    }
  }
}
```

Step 5.2: Payment Controller Create `src/controllers/paymentController.ts`:

```
import { Request, Response } from 'express';
import { PrismaClient } from '@prisma/client';
import Joi from 'joi';
import { RMCPaymentService, SabPaisaPaymentService, PayUPaymentService } from '../services/p

const prisma = new PrismaClient();
const rmcService = new RMCPaymentService();
const sabPaisaService = new SabPaisaPaymentService();
const payUService = new PayUPaymentService();

const initiatePaymentSchema = Joi.object({
  orderId: Joi.string().required(),
  gateway: Joi.string().valid('rmc', 'sabpaisa', 'payu').required(),
});

export const initiatePayment = async (req: Request, res: Response) => {
  try {
    const { error, value } = initiatePaymentSchema.validate(req.body);
    if (error) {
      return res.status(400).json({ message: error.details[0].message });
    }

    const { orderId, gateway } = value;
```

```
// Get order details
const order = await prisma.order.findUnique({
  where: { id: orderId },
  include: {
    user: {
      select: { name: true, email: true, phone: true }
    }
  }
});

if (!order) {
  return res.status(404).json({ message: 'Order not found' });
}

if (order.paymentStatus !== 'PENDING') {
  return res.status(400).json({ message: 'Payment already processed' });
}

const paymentData = {
  orderId: order.orderNumber,
  amount: Number(order.total),
  customerEmail: order.user.email,
  customerPhone: order.user.phone || '',
  customerName: order.user.name || '',
  returnUrl: `${process.env.FRONTEND_URL}/payment/callback/${gateway}`,
};

let paymentResponse;

switch (gateway) {
  case 'rmc':
    paymentResponse = await rmcService.initiatePayment(paymentData);
    break;
  case 'sabpaisa':
    paymentResponse = await sabPaisaService.initiatePayment(paymentData);
    break;
  case 'payu':
    paymentResponse = await payUService.initiatePayment(paymentData);
    break;
  default:
    return res.status(400).json({ message: 'Invalid payment gateway' });
}

if (!paymentResponse.success) {
  return res.status(400).json({ message: paymentResponse.message });
```

```
    }

    // Update order with payment details
    await prisma.order.update({
      where: { id: orderId },
      data: {
        paymentGateway: gateway.toUpperCase(),
        paymentId: paymentResponse.paymentId,
      }
    });

    res.json({
      success: true,
      paymentUrl: paymentResponse.paymentUrl,
      paymentId: paymentResponse.paymentId,
    });
  } catch (error) {
    console.error('Initiate payment error:', error);
    res.status(500).json({ message: 'Internal server error' });
  }
};

export const handlePaymentCallback = async (req: Request, res: Response) => {
  try {
    const { gateway } = req.params;
    const callbackData = req.body;

    let isPaymentSuccess = false;

    switch (gateway) {
      case 'rmc':
        isPaymentSuccess = await rmcService.verifyPayment(callbackData.payment_id);
        break;
      case 'sabpaisa':
        isPaymentSuccess = await sabPaisaService.verifyPayment(callbackData.paymentId);
        break;
      case 'payu':
        isPaymentSuccess = await payUService.verifyPayment(callbackData);
        break;
      default:
        return res.status(400).json({ message: 'Invalid payment gateway' });
    }

    // Update order payment status
    const orderId = callbackData.order_id || callbackData.txnid;
    await prisma.order.update({
```

```
      where: { orderNumber: orderId },
      data: {
        paymentStatus: isPaymentSuccess ? 'SUCCESS' : 'FAILED',
      }
    });

    if (isPaymentSuccess) {
      res.redirect(`${process.env.FRONTEND_URL}/orders/success`);
    } else {
      res.redirect(`${process.env.FRONTEND_URL}/orders/failed`);
    }
  } catch (error) {
    console.error('Payment callback error:', error);
    res.redirect(`${process.env.FRONTEND_URL}/orders/failed`);
  }
};

export const verifyPayment = async (req: Request, res: Response) => {
  try {
    const { orderId } = req.params;

    const order = await prisma.order.findUnique({
      where: { id: orderId },
      select: { paymentStatus: true, paymentGateway: true, paymentId: true }
    });

    if (!order) {
      return res.status(404).json({ message: 'Order not found' });
    }

    res.json({
      success: true,
      paymentStatus: order.paymentStatus,
      paymentGateway: order.paymentGateway,
      paymentId: order.paymentId,
    });
  } catch (error) {
    console.error('Verify payment error:', error);
    res.status(500).json({ message: 'Internal server error' });
  }
};
```

Phase 6: Advanced Features (Week 6)

Step 6.1: Cart Management APIs Create `src/controllers/cartController.ts`:

```typescript
import { Request, Response } from 'express';
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

export const getCart = async (req: Request, res: Response) => {
  try {
    const userId = req.user.id;

    const cartItems = await prisma.cartItem.findMany({
      where: { userId },
      include: {
        product: {
          include: {
            variants: true,
            category: true
          }
        },
        variant: true
      }
    });

    res.json({
      success: true,
      data: cartItems
    });
  } catch (error) {
    console.error('Get cart error:', error);
    res.status(500).json({ message: 'Internal server error' });
  }
};

export const addToCart = async (req: Request, res: Response) => {
  try {
    const userId = req.user.id;
    const { productId, variantId, quantity = 1 } = req.body;

    // Check if item already exists in cart
    const existingItem = await prisma.cartItem.findFirst({
      where: {
        userId,
        productId,
        variantId: variantId || null
      }
    });
```

```
    if (existingItem) {
      // Update quantity
      const updatedItem = await prisma.cartItem.update({
        where: { id: existingItem.id },
        data: { quantity: existingItem.quantity + quantity },
        include: {
          product: {
            include: { variants: true }
          },
          variant: true
        }
      });

      return res.json({
        success: true,
        data: updatedItem
      });
    }

    // Add new item to cart
    const cartItem = await prisma.cartItem.create({
      data: {
        userId,
        productId,
        variantId,
        quantity
      },
      include: {
        product: {
          include: { variants: true }
        },
        variant: true
      }
    });

    res.status(201).json({
      success: true,
      data: cartItem
    });
  } catch (error) {
    console.error('Add to cart error:', error);
    res.status(500).json({ message: 'Internal server error' });
  }
};
```

Step 6.2: Search & Recommendations Create `src/controllers/searchController.ts`:

```
import { Request, Response } from 'express';
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

export const searchProducts = async (req: Request, res: Response) => {
  try {
    const { q, category, minPrice, maxPrice, sort = 'relevance' } = req.query;

    if (!q || (q as string).length < 2) {
      return res.status(400).json({ message: 'Search query must be at least 2 characters' });
    }

    const where: any = {
      OR: [
        { name: { contains: q as string, mode: 'insensitive' } },
        { brand: { contains: q as string, mode: 'insensitive' } },
        { description: { contains: q as string, mode: 'insensitive' } },
        { tags: { has: q as string } }
      ]
    };

    if (category) {
      where.categoryId = category;
    }

    if (minPrice || maxPrice) {
      where.price = {};
      if (minPrice) where.price.gte = Number(minPrice);
      if (maxPrice) where.price.lte = Number(maxPrice);
    }

    let orderBy: any = { createdAt: 'desc' };
    if (sort === 'price_asc') orderBy = { price: 'asc' };
    if (sort === 'price_desc') orderBy = { price: 'desc' };
    if (sort === 'rating') orderBy = { rating: 'desc' };

    const products = await prisma.product.findMany({
      where,
      include: {
        category: true,
        subcategory: true,
        variants: true
      },
```

```
      orderBy,
      take: 50
    });

    res.json({
      success: true,
      data: products,
      query: q
    });
  } catch (error) {
    console.error('Search products error:', error);
    res.status(500).json({ message: 'Internal server error' });
  }
};
```

Phase 7: Admin Panel APIs (Week 7)

Step 7.1: Admin Dashboard APIs Create `src/controllers/adminController.ts`:

```
import { Request, Response } from 'express';
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

export const getDashboardStats = async (req: Request, res: Response) => {
  try {
    const [
      totalUsers,
      totalProducts,
      totalOrders,
      totalRevenue,
      recentOrders,
      topProducts
    ] = await Promise.all([
      prisma.user.count({ where: { role: 'USER' } }),
      prisma.product.count(),
      prisma.order.count(),
      prisma.order.aggregate({
        _sum: { total: true },
        where: { status: 'DELIVERED' }
      }),
      prisma.order.findMany({
        include: {
          user: { select: { name: true, email: true } },
          items: true
        },
```

```
        orderBy: { createdAt: 'desc' },
        take: 10
      }),
      prisma.orderItem.groupBy({
        by: ['productId'],
        _sum: { quantity: true },
        _count: { productId: true },
        orderBy: { _sum: { quantity: 'desc' } },
        take: 10
      })
    ]);

    // Get product details for top products
    const topProductIds = topProducts.map(item => item.productId);
    const topProductDetails = await prisma.product.findMany({
      where: { id: { in: topProductIds } },
      select: { id: true, name: true, brand: true, images: true }
    });

    const topProductsWithDetails = topProducts.map(item => {
      const product = topProductDetails.find(p => p.id === item.productId);
      return {
        ...item,
        product
      };
    });

    res.json({
      success: true,
      data: {
        stats: {
          totalUsers,
          totalProducts,
          totalOrders,
          totalRevenue: totalRevenue._sum.total || 0
        },
        recentOrders,
        topProducts: topProductsWithDetails
      }
    });
  } catch (error) {
    console.error('Get dashboard stats error:', error);
    res.status(500).json({ message: 'Internal server error' });
  }
};
```

Phase 8: Testing & Documentation (Week 8)

Step 8.1: Unit Testing Setup Create `src/tests/auth.test.ts`:

```
import request from 'supertest';
import app from '../app';
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

describe('Authentication', () => {
  beforeEach(async () => {
    // Clean up test data
    await prisma.user.deleteMany({
      where: { email: { contains: 'test' } }
    });
  });

  afterAll(async () => {
    await prisma.$disconnect();
  });

  describe('POST /api/auth/send-otp', () => {
    it('should send OTP to email', async () => {
      const emailData = {
        email: 'test@example.com',
      };

      const response = await request(app)
        .post('/api/auth/send-otp')
        .send(emailData)
        .expect(200);

      expect(response.body.success).toBe(true);
      expect(response.body.message).toContain('OTP sent');
    });
  });

  describe('POST /api/auth/verify-otp', () => {
    it('should verify OTP and authenticate user', async () => {
      const otpData = {
        email: 'test@example.com',
        code: '123456',
        name: 'Test User',
        phone: '9876543210',
      };
```

```
      const response = await request(app)
        .post('/api/auth/verify-otp')
        .send(otpData)
        .expect(200);

      expect(response.body.success).toBe(true);
      expect(response.body.user.email).toBe(otpData.email);
      expect(response.body.token).toBeDefined();
    });
  });
});
```

Step 8.2: API Documentation with Swagger Install Swagger dependencies:

```
npm install swagger-jsdoc swagger-ui-express
npm install -D @types/swagger-jsdoc @types/swagger-ui-express
```

Create `src/config/swagger.ts`:

```
import swaggerJsdoc from 'swagger-jsdoc';
import swaggerUi from 'swagger-ui-express';

const options = {
  definition: {
    openapi: '3.0.0',
    info: {
      title: 'GoBazar API',
      version: '1.0.0',
      description: 'API documentation for GoBazar e-commerce platform',
    },
    servers: [
      {
        url: 'http://localhost:5000',
        description: 'Development server',
      },
    ],
    components: {
      securitySchemes: {
        bearerAuth: {
          type: 'http',
          scheme: 'bearer',
          bearerFormat: 'JWT',
        },
      },
    },
```

```
  },
  apis: ['./src/routes/*.ts', './src/controllers/*.ts'],
};

const specs = swaggerJsdoc(options);

export { specs, swaggerUi };
```

Phase 9: Deployment & Production Setup (Week 9)

Step 9.1: Production Configuration Create `src/config/database.ts`:

```
import { PrismaClient } from '@prisma/client';

const globalForPrisma = globalThis as unknown as {
  prisma: PrismaClient | undefined;
};

export const prisma = globalForPrisma.prisma ?? new PrismaClient();

if (process.env.NODE_ENV !== 'production') globalForPrisma.prisma = prisma;
```

Step 9.2: Docker Configuration Create `Dockerfile`:

```
FROM node:18-alpine

WORKDIR /app

COPY package*.json ./
RUN npm ci --only=production

COPY . .
RUN npm run build

EXPOSE 5000

CMD ["npm", "start"]
```

Create `docker-compose.yml`:

```
version: '3.8'

services:
  app:
    build: .
    ports:
      - "5000:5000"
    environment:
```

```
      - NODE_ENV=production
      - DATABASE_URL=postgresql://gobazar_user:secure_password@db:5432/gobazar_db
    depends_on:
      - db

  db:
    image: postgres:15-alpine
    environment:
      - POSTGRES_DB=gobazar_db
      - POSTGRES_USER=gobazar_user
      - POSTGRES_PASSWORD=secure_password
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

volumes:
  postgres_data:
```

Step 9.3: Environment Setup Scripts Create `scripts/setup.sh`:

```
#!/bin/bash

# Install dependencies
npm install

# Setup environment
cp .env.example .env

# Generate Prisma client
npx prisma generate

# Run migrations
npx prisma migrate deploy

# Seed database
npx prisma db seed

echo "Setup complete!"
```

Step 9.4: Production Deployment Checklist

- ☐ Set up PostgreSQL database
- ☐ Configure environment variables
- ☐ Set up SSL certificates
- ☐ Configure reverse proxy (Nginx)

- ☐ Set up monitoring (PM2)
- ☐ Configure logging
- ☐ Set up backup strategy
- ☐ Configure CI/CD pipeline

API Endpoints Summary

Authentication

- `POST /api/auth/send-otp` - Send OTP to email for authentication
- `POST /api/auth/verify-otp` - Verify OTP and authenticate user
- `POST /api/auth/logout` - User logout

Products

- `GET /api/products` - Get all products (with pagination, filters)
- `GET /api/products/:id` - Get product by ID
- `POST /api/products` - Create product (Admin)
- `PUT /api/products/:id` - Update product (Admin)
- `DELETE /api/products/:id` - Delete product (Admin)

Categories

- `GET /api/categories` - Get all categories
- `GET /api/categories/:id` - Get category by ID
- `POST /api/categories` - Create category (Admin)
- `PUT /api/categories/:id` - Update category (Admin)

Orders

- `GET /api/orders` - Get user orders
- `GET /api/orders/:id` - Get order by ID
- `POST /api/orders` - Create new order
- `PUT /api/orders/:id/status` - Update order status (Admin)
- `POST /api/orders/:id/payment` - Process payment for order

Cart

- `GET /api/cart` - Get user cart
- `POST /api/cart` - Add item to cart
- `PUT /api/cart/:id` - Update cart item
- `DELETE /api/cart/:id` - Remove item from cart

Search & Recommendations

- `GET /api/search` - Search products
- `GET /api/recommendations/:productId` - Get product recommendations

Payment

- `POST /api/payment/rmc/initiate` - Initiate RMC payment
- `POST /api/payment/sabpaisa/initiate` - Initiate SabPaisa payment
- `POST /api/payment/payu/initiate` - Initiate PayU Money payment
- `POST /api/payment/:gateway/callback` - Payment callback handler
- `POST /api/payment/:gateway/verify` - Verify payment status

Admin

- `GET /api/admin/dashboard` - Get dashboard stats
- `GET /api/admin/orders` - Get all orders
- `GET /api/admin/users` - Get all users
- `GET /api/admin/products` - Get all products with admin data

Development Commands

```
# Development
npm run dev          # Start development server
npm run build        # Build for production
npm run start        # Start production server

# Database
npx prisma generate  # Generate Prisma client
npx prisma migrate dev # Run migrations
npx prisma db seed   # Seed database
npx prisma studio    # Open Prisma Studio

# Testing
npm test             # Run tests
npm run test:watch   # Run tests in watch mode
npm run test:coverage # Run tests with coverage

# Linting
npm run lint         # Run ESLint
npm run lint:fix     # Fix ESLint issues
npm run format       # Format code with Prettier
```

Security Considerations

1. **Authentication**: JWT tokens with secure secrets
2. **Authorization**: Role-based access control
3. **Input Validation**: Joi schema validation
4. **Rate Limiting**: Express rate limiting
5. **CORS**: Configured for specific origins
6. **Helmet**: Security headers
7. **SQL Injection**: Prisma ORM protection

8. **Password Hashing**: bcrypt with salt rounds

Performance Optimizations

1. **Database Indexing**: Proper indexes on frequently queried fields
2. **Pagination**: Implemented for all list endpoints
3. **Caching**: Redis for frequently accessed data
4. **Compression**: Gzip compression enabled
5. **Connection Pooling**: Prisma connection pooling
6. **Query Optimization**: Efficient Prisma queries

Monitoring & Logging

1. **Winston**: Structured logging
2. **Morgan**: HTTP request logging
3. **Health Checks**: `/health` endpoint
4. **Error Tracking**: Centralized error handling
5. **Performance Monitoring**: Response time tracking

Backup Strategy

1. **Database Backups**: Daily automated backups
2. **File Backups**: Cloudinary asset backups
3. **Configuration Backups**: Environment and config files
4. **Disaster Recovery**: Multi-region deployment plan

This comprehensive workflow provides a complete roadmap for building a production-ready backend for the GoBazar e-commerce platform. Each phase builds upon the previous one, ensuring a systematic and thorough development process.