CraftCart

Artisan Marketplace Platform

Technical Documentation

Complete Architecture, Deployment & Development Guide

Project Overview

Framework: Nuxt 4 (Vue 3)
Backend: Node.js with Nitro Server
Database: MongoDB with Mongoose ODM
Deployment: Vercel
Repository: `https://github.com/Black-Lights/craftcart-website`

Last Updated: November 23, 2025

Version 1.0

Contents

1 Introduction

1.1 Project Overview

CraftCart is a full-stack marketplace platform designed to connect skilled Indian artisans with customers across India. The platform empowers traditional craftspeople to showcase and sell their handcrafted products while supporting UN SDG 8 (Decent Work and Economic Growth).

1.2 Key Features

- **Dual Authentication System:** Separate seller and customer roles with JWT-based security
- **Product Management:** Full CRUD operations with 8 distinct categories
- **Shopping Experience:** Cart management, checkout, and order tracking
- **Seller Dashboard:** Real-time analytics, inventory, and order management
- **Responsive Design:** Mobile-first UI using Tailwind CSS
- **58+ Seeded Products:** Pre-populated catalog for demonstration

1.3 Target Audience

- **Sellers:** Indian artisans and craftspeople
- **Customers:** Individuals seeking authentic handcrafted products
- **Categories:** Handicrafts, Textiles, Pottery, Jewelry, Home Decor, Paintings, Woodwork, Metalwork

2 Technology Stack

2.1 Frontend Technologies

Technology	Version	Purpose
Vue.js	3.5.24	Progressive JavaScript framework
Nuxt	4.2.1	Full-stack framework for Vue
TypeScript	5.9.3	Type-safe development
Pinia	0.11.3	State management
Tailwind CSS	3.4.15	Utility-first CSS framework
Nuxt UI	4.2.0	Pre-built UI components
Vue Router	4.6.3	Client-side routing

Table 1: Frontend Technology Stack

2.2 Backend Technologies

Technology	Version	Purpose
Node.js	20+	Server runtime environment
Nitro	(via Nuxt)	Server engine
MongoDB	5.0+	NoSQL database
Mongoose	8.20.0	MongoDB object modeling
JWT	9.0.2	Authentication tokens
bcryptjs	3.0.3	Password hashing
Firebase	12.6.0	Cloud storage (planned)

Table 2: Backend Technology Stack

2.3 Development Tools

- **Package Manager:** npm
- **Version Control:** Git & GitHub
- **Code Editor:** Visual Studio Code
- **Deployment Platform:** Vercel
- **Database Hosting:** MongoDB Atlas
- **CI/CD:** GitHub Actions (sync workflow)

3 System Architecture

3.1 Application Architecture

CraftCart follows a modern full-stack architecture pattern:

Architecture Layers

1. Presentation Layer (Client)

- Vue 3 components with Composition API
- Pinia stores for state management
- Tailwind CSS for styling
- Client-side routing with Vue Router

2. Application Layer (Server)

- Nitro server engine
- API routes (/api/*)
- Server middleware for authentication
- SSR (Server-Side Rendering) capabilities

3. Data Layer

- MongoDB database
- Mongoose schemas and models
- Database connection pooling

3.2 Project Structure

Listing 1: CraftCart Directory Structure

```
CraftCart_Website/
|-- .github/
|   `-- workflows/
|       `-- sync-fork.yml          # GitHub Actions sync workflow
|-- assets/
|   `-- css/
|       `-- main.css              # Global styles
|-- components/
|   |-- AppButton.vue             # Reusable button component
|   |-- AppCard.vue               # Card wrapper component
|   |-- AuthModal.vue             # Authentication modal
|   |-- ProductCard.vue           # Product display card
|   `-- ToastNotification.vue    # Toast notification system
|-- layouts/
|   |-- auth.vue                  # Authentication pages layout
|   `-- default.vue               # Main application layout
|-- middleware/
|   `-- auth.ts                   # Client-side auth middleware
|-- pages/
|   |-- index.vue                 # Home page
|    |-- about.vue                # About page
|    |-- cart.vue                 # Shopping cart
|    |-- checkout.vue             # Checkout page
|    |-- contact.vue              # Contact page
|    |-- faq.vue                  # FAQ page
|    |-- privacy.vue              # Privacy policy
|    |-- terms.vue                # Terms and conditions
|    |-- auth/
|    |    |-- login.vue           # Login page
|    |    `-- register.vue        # Registration page
|    |-- customer/
|    |    `-- orders.vue          # Customer order history
|    |-- products/
|    |    |-- index.vue           # Product listing
|    |    |-- [id].vue            # Product details
|    |    |-- create.vue          # Create product (seller)
|    |    `-- edit/
|    |        `-- [id].vue        # Edit product (seller)
|    |-- seller/
|    |    |-- dashboard.vue       # Seller dashboard
|    |    |-- products.vue        # Seller product management
|    |    |-- orders.vue          # Seller order management
|    |    `-- help.vue            # Seller help center
|    `-- order-success/
|         `-- [id].vue            # Order confirmation
|-- plugins/
|    `-- auth.client.ts           # Client-side auth plugin
|-- public/
|    `-- robots.txt               # SEO configuration
|-- scripts/
|    |-- seed-products.mjs        # Database seeding script
|    |-- seed-more-products.mjs   # Additional products seeding
|    |-- remove-duplicates.mjs    # Duplicate removal utility
|    `-- fix-categories.mjs       # Category cleanup script
|-- server/
|    |-- api/
|    |    |-- auth/
|    |    |    |-- login.post.ts     # Login endpoint
|    |    |    |-- logout.post.ts    # Logout endpoint
|    |    |    |-- me.get.ts         # Get current user
|    |    |    `-- register.post.ts # Registration endpoint
|    |    |-- orders/
|    |    |    |-- create.post.ts    # Create order
|    |    |    |-- my-orders.get.ts # Get user orders
|    |    |    |-- [id]/
|    |    |    |    `-- status.patch.ts  # Update order status
|    |    |    `-- seller/
|    |    |         `-- my-orders.get.ts # Get seller orders
|    |    `-- products/
|    |         |-- index.get.ts      # List products
|    |         |-- create.post.ts    # Create product
|   |       |-- [id].get.ts      # Get product
|   |       |-- [id].put.ts      # Update product
|   |       |-- [id].patch.ts    # Partial update
|   |       |-- [id].delete.ts   # Delete product
|   |       `-- seller/
|   |           `-- my-products.get.ts  # Get seller products
|   |-- middleware/
|   |   `-- auth.ts              # Server-side auth middleware
|   |-- models/
|   |   |-- User.ts              # User schema
|   |   |-- Product.ts           # Product schema
|   |   `-- Order.ts             # Order schema
|   `-- plugins/
|       `-- mongoose.ts          # MongoDB connection
|-- stores/
|   |-- authStore.ts             # Authentication state
|   |-- cartStore.ts             # Shopping cart state
|   |-- productStore.ts          # Product state
|   `-- toastStore.ts            # Notification state
|-- types/
|   |-- user.ts                  # User TypeScript types
|   |-- product.ts               # Product TypeScript types
|   |-- order.ts                 # Order TypeScript types
|   `-- api.ts                   # API response types
|-- .env                         # Environment variables (
    gitignored)
|-- .env.example                 # Environment template
|-- .gitignore                   # Git ignore rules
|-- app.config.ts                # App configuration
|-- app.vue                      # Root Vue component
|-- nuxt.config.ts               # Nuxt configuration
|-- package.json                 # Dependencies
|-- tsconfig.json                # TypeScript configuration
|-- README.md                    # Project documentation
|-- MVP-Implementation.md        # Implementation guide
`-- SETUP_COMPLETE.md            # Setup instructions
```

4 Frontend Architecture

4.1 Vue 3 with Composition API

CraftCart utilizes Vue 3's Composition API for all components, providing:

- Better TypeScript integration
- Improved code organization
- Enhanced reusability through composables
- Better performance with reactivity system

4.2 Nuxt 4 Framework

Nuxt 4 provides the foundation with:

- **File-based Routing:** Automatic route generation from pages/ directory
- **Auto-imports:** Components, composables, and utilities
- **Layouts System:** Reusable page layouts (default, auth)
- **SSR Support:** Server-Side Rendering for better SEO
- **API Routes:** server/api/ directory for backend endpoints
- **Middleware:** Client and server-side route guards

4.3 State Management with Pinia

Four main stores manage application state:

Listing 2: Auth Store Example

```
// stores/authStore.ts
import { defineStore } from 'pinia'

export const useAuthStore = defineStore('auth', () => {
  // State
  const user = ref<User | null>(null)
  const loading = ref<boolean>(false)
  const error = ref<string | null>(null)

  // Getters
  const isAuthenticated = computed(() => !!user.value)
  const isSeller = computed(() => user.value?.userType === 'seller
     ')
  const isCustomer = computed(() => user.value?.userType === '
     customer')

  // Actions
  const login = async (credentials: UserLogin) => {
    loading.value = true
    const response = await $fetch('/api/auth/login', {
      method: 'POST',
      body: credentials,
    })
    user.value = response.data.user
    loading.value = false
  }

  return { user, loading, error, isAuthenticated,
           isSeller, isCustomer, login }
})
```

Store Responsibilities:

- **authStore:** User authentication and session management
- **cartStore:** Shopping cart items and calculations
- **productStore:** Product listing, filtering, pagination
- **toastStore:** Global notification system

4.4 UI Components

4.4.1 Nuxt UI Integration

Pre-built components from Nuxt UI:

- UButton, UInput, UForm
- UCard, UBadge, UAvatar
- UModal, UDropdown, UPagination
- Built on Tailwind CSS

4.4.2 Custom Components

1. **ProductCard:** Displays product with image, price, seller info
2. **AppButton:** Reusable button with loading states
3. **ToastNotification:** Success/error message display
4. **AuthModal:** Login/register modal dialog

5 Backend Architecture

5.1 Nitro Server Engine

Nuxt 4's Nitro server provides:

- Fast, lightweight server runtime
- API route handling
- Server middleware support
- Edge deployment compatibility
- Built-in caching capabilities

5.2 API Routes Structure

All API endpoints follow RESTful conventions:

Endpoint	Method	Description
/api/auth/register	POST	Register new user
/api/auth/login	POST	Authenticate user
/api/auth/logout	POST	Clear session
/api/auth/me	GET	Get current user
/api/products	GET	List products with filters
/api/products/create	POST	Create new product
/api/products/[id]	GET	Get product details
/api/products/[id]	PUT	Update product
/api/products/[id]	DELETE	Delete product
/api/products/seller/my-products	GET	Get seller's products
/api/orders/create	POST	Create new order
/api/orders/my-orders	GET	Get customer orders
/api/orders/seller/my-orders	GET	Get seller orders
/api/orders/[id]/status	PATCH	Update order status

Table 3: API Endpoints Overview

5.3 Database Models

5.3.1 User Model

Listing 3: User Schema

```
// server/models/User.ts
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  phone: { type: String, required: true },
  password: { type: String, required: true, select: false },
  userType: {
    type: String,
    enum: ['seller', 'customer'],
    required: true
  },
  profileImage: String,
  isVerified: { type: Boolean, default: false },
  location: {
    city: String,
    state: String,
  },
  rating: {
    average: { type: Number, default: 0 },
    count: { type: Number, default: 0 },
  },
}, { timestamps: true })
```

5.3.2 Product Model

Listing 4: Product Schema

```
// server/models/Product.ts
const productSchema = new mongoose.Schema({
  sellerId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true
  },
  title: { type: String, required: true },
  description: { type: String, required: true },
  price: { type: Number, required: true },
  category: {
    type: String,
    enum: ['Handicrafts', 'Textiles', 'Pottery',
           'Jewelry', 'Home Decor', 'Paintings',
           'Woodwork', 'Metalwork'],
    required: true,
  },
  images: [{ type: String, required: true }],
  stock: { type: Number, default: 1 },
  location: {
    city: String,
    state: String,
  },
  isActive: { type: Boolean, default: true },
}, { timestamps: true })
```

5.3.3 Order Model

Listing 5: Order Schema

```
// server/models/Order.ts
const orderSchema = new mongoose.Schema({
  orderNumber: { type: String, unique: true, required: true },
  customerId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true
  },
  sellerId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true
  },
  productId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Product',
    required: true
  },
  quantity: { type: Number, required: true },
  totalAmount: { type: Number, required: true },
  deliveryAddress: {
    street: String,
    city: String,
    state: String,
    pincode: String,
    phone: String,
  },
  status: {
    type: String,
    enum: ['pending', 'confirmed', 'shipped',
           'delivered', 'cancelled'],
    default: 'pending',
  },
}, { timestamps: true })
```

5.4 Authentication System

5.4.1 JWT Implementation

1. User submits credentials via POST /api/auth/login
2. Server validates credentials against database
3. bcryptjs compares hashed passwords
4. JWT token generated with user ID and role
5. Token stored in httpOnly cookie (XSS protection)
6. Cookie expires in 7 days
7. Subsequent requests include cookie automatically

5.4.2 Password Security

Listing 6: Password Hashing

```
// Pre-save hook in User model
userSchema.pre('save', async function(next) {
  if (!this.isModified('password')) return next()
  this.password = await bcrypt.hash(this.password, 12)
  next()
})

// Password comparison method
userSchema.methods.comparePassword = async function(
   candidatePassword) {
  return await bcrypt.compare(candidatePassword, this.password)
}
```

5.5 Database Connection

MongoDB connection is established via Nitro plugin:

Listing 7: MongoDB Plugin

```
// server/plugins/mongoose.ts
export default defineNitroPlugin(async () => {
  const config = useRuntimeConfig()

  if (!config.mongodbUri) {
    console.error('MONGODB_URI not set!')
    return
  }

  const options = {
    serverSelectionTimeoutMS: 10000,
    socketTimeoutMS: 45000,
    maxPoolSize: 10,
  }

  await mongoose.connect(config.mongodbUri, options)
  console.log('MongoDB connected successfully')
})
```

6 Key Features Implementation

6.1 Authentication Flow

Registration Process

Client Side:

1. User fills registration form (pages/auth/register.vue)
2. Form data validated client-side
3. authStore.register() called
4. POST request to /api/auth/register

Server Side:

1. Validate request body
2. Check if email already exists
3. Hash password with bcrypt
4. Create user in database
5. Generate JWT token
6. Set httpOnly cookie
7. Return user data (without password)

Post-Registration:

1. Update authStore state
2. Redirect to home page
3. Display welcome toast notification

6.2 Product Listing & Filtering

6.2.1 Query Parameters

Products can be filtered using:

- `category`: Filter by product category
- `minPrice`, `maxPrice`: Price range
- `search`: Text search in title/description
- `sortBy`: Field to sort by (price, createdAt, etc.)
- `sortOrder`: asc or desc
- `page`, `limit`: Pagination

6.2.2 Implementation

Listing 8: Product Filtering API

```
// server/api/products/index.get.ts
export default defineEventHandler(async (event) => {
  const query = getQuery(event)
  const filter = { isActive: true }

  // Category filter
  if (query.category && query.category !== 'all') {
    filter.category = query.category
  }

  // Price range filter
  if (query.minPrice || query.maxPrice) {
    filter.price = {}
    if (query.minPrice) filter.price.$gte = Number(query.minPrice)
    if (query.maxPrice) filter.price.$lte = Number(query.maxPrice)
  }

  // Text search
  if (query.search) {
    filter.$or = [
      { title: { $regex: query.search, $options: 'i' } },
      { description: { $regex: query.search, $options: 'i' } },
    ]
  }

  const products = await Product.find(filter)
    .populate('sellerId', 'name rating')
    .sort({ [query.sortBy]: query.sortOrder === 'desc' ? -1 : 1 })
    .skip((query.page - 1) * query.limit)
    .limit(Number(query.limit))

  return { success: true, data: products }
})
```

6.3 Shopping Cart

Cart is managed client-side with localStorage persistence:

Listing 9: Cart Store

```
// stores/cartStore.ts
export const useCartStore = defineStore('cart', {
  state: () => ({
    items: [] as CartItem[],
  }),

  getters: {
    cartCount: (state) =>
```

```
      state.items.reduce((total, item) => total + item.quantity,
         0),

    cartTotal: (state) =>
      state.items.reduce((total, item) =>
        total + (item.price * item.quantity), 0),
  },

  actions: {
    addToCart(product) {
      const existingItem = this.items.find(
        item => item.productId === product.id
      )

      if (existingItem) {
        existingItem.quantity++
      } else {
        this.items.push({
          productId: product.id,
          title: product.title,
          price: product.price,
          quantity: 1,
          // ... other fields
        })
      }

      this.saveToLocalStorage()
    },

    saveToLocalStorage() {
      localStorage.setItem('cart', JSON.stringify(this.items))
    },
  },

  // Persist to localStorage
  persist: true,
})
```

6.4 Order Management

6.4.1 Order Creation

1. User clicks "Checkout" in cart
2. Enters delivery address
3. System generates unique order number: ORD-timestamp-random
4. Creates order documents (one per seller)
5. Reduces product stock

6. Clears cart
7. Shows order confirmation

6.4.2 Order Status Workflow

[node distance=2cm, auto]

Order Lifecycle:

1. **Pending:** Order created, awaiting seller confirmation
2. **Confirmed:** Seller confirms order
3. **Shipped:** Order dispatched for delivery
4. **Delivered:** Order completed successfully
5. **Cancelled:** Order cancelled (stock restored)

6.5 Seller Dashboard

Analytics displayed on seller dashboard:

- Total Orders
- Total Revenue (excluding cancelled)
- Active Products
- Order Completion Rate
- Recent Orders
- Top Selling Products
- Order Status Distribution

Listing 10: Dashboard Analytics

```
// Calculated in seller/dashboard.vue
const analytics = computed(() => {
  const totalOrders = orders.value.length
  const totalRevenue = orders.value
    .filter(o => o.status !== 'cancelled')
    .reduce((sum, o) => sum + o.totalAmount, 0)

  const completedOrders = orders.value
    .filter(o => o.status === 'delivered').length

  const completionRate = totalOrders > 0
    ? (completedOrders / totalOrders * 100).toFixed(1)
    : 0

  return { totalOrders, totalRevenue, completionRate }
})
```

7 Deployment on Vercel

7.1 Why Vercel?

Vercel is ideal for Nuxt applications:

- **Zero Configuration:** Automatic Nuxt detection
- **Edge Network:** Global CDN for fast delivery
- **Serverless Functions:** API routes as serverless functions
- **Environment Variables:** Secure configuration management
- **Automatic Deployments:** GitHub integration
- **Free Tier:** Generous free plan for personal projects

7.2 Deployment Configuration

Nuxt 4 is automatically configured for Vercel deployment. No additional configuration file needed.

7.3 Deployment Steps

1. **Push to GitHub**

```
git add .
git commit -m "Ready for deployment"
git push origin main
```

2. **Connect to Vercel**
   - Visit `https://vercel.com`
   - Sign in with GitHub
   - Click "New Project"
   - Import repository: Black-Lights/craftcart-website

3. **Configure Environment Variables**

   In Vercel Project Settings → Environment Variables, add:

   - `MONGODB_URI`: Your MongoDB Atlas connection string
   - `JWT_SECRET`: Strong random secret key
   - `RAZORPAY_KEY_ID`: Payment gateway key (if using)
   - `RAZORPAY_KEY_SECRET`: Payment gateway secret
   - `FIREBASE_*`: Firebase configuration (if using)

4. **Deploy**
   - Click "Deploy"

- Vercel automatically builds and deploys
- Deployment completes in 2-3 minutes
- Accessible at: `https://craftcart-website.vercel.app`

5. **Automatic Updates**

   - Every push to main branch triggers deployment
   - Pull requests create preview deployments
   - Roll back to any previous deployment

7.4 MongoDB Atlas Configuration

For Vercel deployment, MongoDB Atlas must allow connections:

1. Login to MongoDB Atlas
2. Navigate to Network Access
3. Click "Add IP Address"
4. Select "Allow Access from Anywhere" (0.0.0.0/0)
5. This allows Vercel's serverless functions to connect

Security Note: Use strong database passwords and enable authentication.

7.5 GitHub Actions Sync Workflow

Repository includes automatic sync to deployment fork:

Listing 11: .github/workflows/sync-fork.yml

```
name: Sync to Deployment Repository

on:
  push:
    branches:
      - main

jobs:
  sync:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout source repository
        uses: actions/checkout@v3
        with:
          fetch-depth: 0

      - name: Push to deployment repository
        env:
          SYNC_TOKEN: ${{ secrets.SYNC_TOKEN }}
        run: |
          git remote add deployment \
            https://x-access-token:${SYNC_TOKEN}@github.com/\
            AliRehman7065/craftcart-website.git
          git push deployment main:main --force
```

8 Development Workflow

8.1 Local Development Setup

1. **Clone Repository**

```
git clone https://github.com/Black-Lights/craftcart-website.git
cd craftcart-website
```

2. **Install Dependencies**

```
npm install
```

3. **Configure Environment**

   Create .env file:

```
MONGODB_URI=mongodb://localhost:27017/craftcart
JWT_SECRET=your-secret-key-change-in-production
```

4. **Start Development Server**

```
npm run dev
# Server runs at http://localhost:3000
```

5. **Seed Database (Optional)**

```
node scripts/seed-products.mjs
node scripts/seed-more-products.mjs
```

8.2 Development Commands

Command	Purpose
`npm run dev`	Start development server
`npm run build`	Build for production
`npm run generate`	Generate static site
`npm run preview`	Preview production build
`npm run postinstall`	Prepare Nuxt

Table 4: Available NPM Commands

8.3 Git Workflow

1. Create feature branch

```
git checkout -b feature/new-feature
```

2. Make changes and commit

```
git add .
git commit -m "feat: add new feature"
```

3. Push to GitHub

```
git push origin feature/new-feature
```

4. Create Pull Request on GitHub
5. Merge to main (triggers deployment)

8.4 Code Organization Best Practices

- **Components:** Reusable UI elements in components/
- **Pages:** Route-based components in pages/
- **Stores:** State management in stores/
- **Types:** TypeScript definitions in types/
- **API Routes:** Server endpoints in server/api/
- **Models:** Database schemas in server/models/
- **Utilities:** Helper functions in utils/ (auto-imported)

9 Security Measures

9.1 Authentication Security

- **Password Hashing:** bcrypt with 12 salt rounds
- **JWT Tokens:** Signed with secret key
- **httpOnly Cookies:** Prevents XSS attacks
- **SameSite Cookie:** CSRF protection
- **Token Expiration:** 7-day validity
- **Selective Field Retrieval:** Password excluded from queries

9.2 Environment Variables

Sensitive data stored in environment variables:

- Database connection strings
- JWT secret keys
- Payment gateway credentials
- Firebase service accounts

Never commit .env file to version control!

9.3 Input Validation

9.3.1 Client-Side

- Form validation with Nuxt UI
- Type checking with TypeScript
- Required field validation
- Email format validation
- Password strength requirements

9.3.2 Server-Side

- Request body validation
- Type coercion and sanitization
- MongoDB injection prevention (Mongoose)
- Authorization checks

9.4 Data Protection

1. **MongoDB Atlas:** Network access control
2. **Mongoose Schemas:** Field validation and defaults
3. **HTTPS:** Enforced in production (Vercel)
4. **CORS:** Configured for same-origin requests

10 Testing & Debugging

10.1 Development Tools

- **Vue Devtools:** Browser extension for Vue debugging
- **Nuxt DevTools:** Built-in Nuxt development tools
- **MongoDB Compass:** GUI for database inspection
- **Postman/Insomnia:** API endpoint testing
- **Browser DevTools:** Network, console, and performance debugging

10.2 Logging

Server-side logging for debugging:

Listing 12: Server Logging Example

```
// server/plugins/mongoose.ts
console.log('MongoDB connected successfully')
console.log(`Database: ${mongoose.connection.name}`)
console.log(`Connection state: ${mongoose.connection.readyState}`)

// Error logging
console.error('MongoDB connection error:', error.message)
console.error('Error details:', {
  name: error.name,
  code: error.code,
  codeName: error.codeName,
})
```

10.3 Common Issues & Solutions

11 Future Enhancements

11.1 Phase 2 Features

1. **Payment Integration**
   - Razorpay payment gateway
   - Multiple payment methods
   - Payment verification
   - Refund handling
2. **Rating & Reviews**
   - Product reviews
   - Seller ratings

Issue	Solution
MongoDB connection failed	Check MONGODB_URI in .env, ensure MongoDB is running, verify network access in Atlas
Port 3000 already in use	Use different port: `PORT=3001 npm run dev`
JWT authentication fails	Verify JWT_SECRET is set, check cookie settings, clear browser cookies
Module not found errors	Run `npm install` to reinstall dependencies
Build fails on Vercel	Check environment variables are set, review build logs, verify Node version

Table 5: Common Issues and Solutions

- Review moderation
- Helpful votes

3. **Real-time Chat**
   - Buyer-seller messaging
   - WebSocket integration
   - Message notifications
   - Chat history
4. **Email Notifications**
   - Order confirmations
   - Status updates
   - Welcome emails
   - Password reset

11.2 Phase 3 Features

1. **Image Upload**
   - Firebase Storage integration
   - Image optimization
   - Multi-image upload
   - Drag-and-drop interface
2. **Advanced Analytics**
   - Sales charts
   - Traffic analytics

- Revenue trends
- Customer insights

3. **Search Enhancement**
   - Elasticsearch integration
   - Autocomplete
   - Advanced filters
   - Sort options
4. **Social Features**
   - Share products
   - Follow sellers
   - Wishlist
   - Product recommendations

12 Performance Optimization

12.1 Current Optimizations

- **Server-Side Rendering:** Faster initial page load
- **Code Splitting:** Automatic by Nuxt/Vite
- **Image Optimization:** Lazy loading for product images
- **Database Indexing:** MongoDB indexes on frequently queried fields
- **Connection Pooling:** MongoDB connection pool (maxPoolSize: 10)
- **Caching:** Browser caching via Vercel CDN

12.2 Planned Optimizations

1. **Redis Caching:** Cache frequently accessed data
2. **Image CDN:** Cloudinary or Firebase Storage
3. **API Rate Limiting:** Prevent abuse
4. **Query Optimization:** MongoDB aggregation pipelines
5. **Lazy Loading:** Component-level lazy loading

13 Conclusion

CraftCart represents a complete, production-ready marketplace platform built with modern web technologies. The combination of Nuxt 4, Vue 3, MongoDB, and Vercel provides:

- **Developer Experience:** TypeScript, auto-imports, hot reload
- **User Experience:** Fast, responsive, accessible
- **Scalability:** Serverless architecture, global CDN
- **Maintainability:** Clear structure, type safety, documentation
- **Security:** Industry-standard authentication and encryption

13.1 Project Statistics

Metric	Value
Total Files	100+
Lines of Code	5,000+
Components	15+
API Endpoints	20+
Database Models	3
Seeded Products	58
Dependencies	20+
Build Time	~2 minutes

Table 6: Project Statistics

13.2 Repository Information

GitHub Repository

Primary Repository:
https://github.com/Black-Lights/craftcart-website
Deployment Repository:
https://github.com/AliRehman7065/craftcart-website
Live Demo:
https://craftcart-website.vercel.app (if deployed)
Documentation:
See README.md, MVP-Implementation.md, SETUP_COMPLETE.md

13.3 Support & Contributing

For issues, questions, or contributions:

- Open an issue on GitHub

- Submit pull requests for improvements
- Follow coding standards and conventions
- Update documentation for new features

Built with for Indian Artisans

A Environment Variables Reference

Variable	Description
MONGODB_URI	MongoDB connection string
JWT_SECRET	Secret key for JWT signing
RAZORPAY_KEY_ID	Razorpay publishable key
RAZORPAY_KEY_SECRET	Razorpay secret key
FIREBASE_API_KEY	Firebase API key
FIREBASE_AUTH_DOMAIN	Firebase auth domain
FIREBASE_DATABASE_URL	Firebase database URL
FIREBASE_PROJECT_ID	Firebase project ID
FIREBASE_STORAGE_BUCKET	Firebase storage bucket
FIREBASE_MESSAGING_SENDER_ID	Firebase messaging sender ID
FIREBASE_APP_ID	Firebase application ID
FIREBASE_SERVICE_ACCOUNT	Firebase admin SDK credentials

Table 7: Complete Environment Variables List

B API Response Format

All API endpoints return consistent JSON responses:

Listing 13: Success Response

```
{
  "success": true,
  "data": {
    // Actual response data
  }
}
```

Listing 14: Error Response

```
{
  "success": false,
  "statusCode": 400,
  "message": "Error description"
}
```

C Database Seeding Commands

Listing 15: Database Management

```
# Seed initial products
node scripts/seed-products.mjs

# Seed additional products
node scripts/seed-more-products.mjs

# Remove duplicate products
node scripts/remove-duplicates.mjs

# Fix category names
node scripts/fix-categories.mjs
```

D Useful Resources

- **Nuxt 3 Documentation:** https://nuxt.com
- **Vue 3 Documentation:** https://vuejs.org
- **Pinia Documentation:** https://pinia.vuejs.org
- **Tailwind CSS:** https://tailwindcss.com
- **MongoDB Documentation:** https://www.mongodb.com/docs
- **Mongoose Documentation:** https://mongoosejs.com
- **Vercel Documentation:** https://vercel.com/docs