E-Commerce Website — React, Node.js, MongoDB

Project Overview

This document is a complete project report and technical guide for building a modern e-commerce web application using **React** (frontend), **Node.js + Express** (backend) and **MongoDB** (database). It includes architecture, data models, API endpoints, setup instructions, sample code snippets, deployment notes, and testing guidance — ready to export as a PDF.

Table of Contents

---

1. Project summary

A scalable e-commerce app that supports product listing, user registration & authentication, shopping cart, checkout, order history, and admin product management. The app is split into a REST API built with Node.js/ Express and a single-page application using React.

2. Features

- User signup / login (JWT)
- Role-based access: user, admin
- Product catalog with categories, search, filters, pagination
- Product details page + images
- Add to cart, update cart quantities
- Checkout flow with shipping address, order summary

- Orders: place, view history, admin can update status
- Admin dashboard: CRUD products, view orders, manage users
- Basic analytics (orders, revenue summary)

3. Tech stack

- Frontend: React, React Router, Context/Redux (optional), Axios, Formik/Yup
- Backend: Node.js, Express
- Database: MongoDB (Atlas or self-hosted)
- Auth: JSON Web Tokens (JWT)
- Storage: Cloud (e.g., AWS S3) or local for product images
- Payments: Stripe (recommended) or PayPal
- Dev tooling: ESLint, Prettier, Jest (tests), Postman (API testing)

4. System architecture

Client (React) <--> REST API (Node/Express) <--> MongoDB

- The client calls REST endpoints to fetch products, manage cart, and perform checkout.
- The server authenticates requests (JWT) and performs CRUD operations on MongoDB.
- Admin routes protected by middleware that checks user role.

(You can include a diagram in the PDF export if desired.)

5. Database design (schemas)

Product schema (Mongoose example)

```
const ProductSchema = new mongoose.Schema({
  name: { type: String, required: true },
  description: String,
  price: { type: Number, required: true },
  category: String,
  images: [String],
  stock: { type: Number, default: 0 },
  rating: { type: Number, default: 0 },
  createdAt: { type: Date, default: Date.now }
});
```

User schema

```
const UserSchema = new mongoose.Schema({
  name: String,
  email: { type: String, required: true, unique: true },
  passwordHash: { type: String, required: true },
```

```
  role: { type: String, enum: ['user','admin'], default: 'user' },
  addresses: [{ label: String, line1: String, city: String, postalCode: String,
country: String }],
  createdAt: { type: Date, default: Date.now }
});
```

Order schema

```
const OrderSchema = new mongoose.Schema({
  user: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  items: [{ product: { type: mongoose.Schema.Types.ObjectId, ref: 'Product' },
qty: Number, priceAtPurchase: Number }],
  shippingAddress: Object,
  totalAmount: Number,
  paymentStatus: { type: String, enum: ['pending','paid','failed'], default:
'pending' },
  orderStatus: { type: String, enum:
['placed','processing','shipped','delivered','cancelled'], default: 'placed' },
  createdAt: { type: Date, default: Date.now }
});
```

6. REST API specification (selected endpoints)

Auth

- `POST /api/auth/register` — register new user
- `POST /api/auth/login` — returns JWT

Products

- `GET /api/products` — list products (query: page, limit, q, category, priceRange)
- `GET /api/products/:id` — get product details
- `POST /api/products` — **admin** create product
- `PUT /api/products/:id` — **admin** update product
- `DELETE /api/products/:id` — **admin** remove product

Cart (client-managed) or server-side

- `POST /api/cart` — (optional) store server-side cart for user

Orders

- `POST /api/orders` — create order (authenticated)
- `GET /api/orders/:id` — get order details (auth + ownership/admin)
- `GET /api/orders` — admin: list all orders; user: list own orders

Payments (webhook)

- `POST /api/webhooks/stripe` — handle payment events from Stripe

7. Frontend structure (React)

Folder layout (example):

```
/src
  /api         // axios instances
  /components  // buttons, inputs, cards
  /pages       // Home, Product, Cart, Checkout, Admin
  /contexts    // auth, cart (or use Redux)
  /hooks       // reusable hooks
  /utils       // helpers (formatCurrency, validators)
  App.js
  index.js
```

Key pages: - Home: product grid, search, pagination - Product Details: images, description, add to cart - Cart: list items, update qty, proceed to checkout - Checkout: shipping, payment, review, place order - Profile: order history, saved addresses - Admin: product & order management

8. Authentication & authorization

- Use bcrypt to hash passwords on signup.
- Issue a JWT signed with a secure secret; store token in httpOnly cookie or local storage (httpOnly cookie recommended for security).
- Middleware on server to verify token and attach `req.user`.
- Authorization middleware for admin routes to check `req.user.role`.

9. Payment integration (overview)

- Use Stripe Checkout or Payment Intents API.
- For card handling, never send raw card data to your server — use Stripe Elements (client) which tokenizes the card and sends a token to your backend.
- Backend uses Stripe secret key to create PaymentIntent, confirm, and listen for webhooks to fulfill orders.

10. Installation & run instructions (local)

Prereqs: Node.js (LTS), npm/yarn, MongoDB (local or Atlas)

1. Clone repo

```
git clone <repo-url>
cd project
```

2. Backend

```
cd backend
cp .env.example .env
# edit .env: MONGODB_URI, JWT_SECRET, STRIPE_KEY etc
npm install
npm run dev
```

3. Frontend

```
cd ../frontend
npm install
npm start
```

API should run on `http://localhost:5000` (example) and React on `http://localhost:3000`.

11. Sample code snippets

Express auth middleware (verify JWT)

```
const jwt = require('jsonwebtoken');
module.exports = function(req,res,next){
  const token = req.headers.authorization?.split(' ')[1];
  if(!token) return res.status(401).json({message:'No token'});
  try{
    const payload = jwt.verify(token, process.env.JWT_SECRET);
    req.user = payload;
    next();
  }catch(err){
    res.status(401).json({message:'Invalid token'});
  }
}
```

Simple React product card (functional)

```
import React from 'react';
export default function ProductCard({p}){
  return (
    <div className="card">
      <img src={p.images?.[0]} alt={p.name} />
```

```
      <h3>{p.name}</h3>
      <p>{p.price.toFixed(2)}</p>
      <button>Add to cart</button>
    </div>
  )
}
```

12. Testing strategy

- Unit tests: Jest for backend functions and React components.
- Integration tests: Supertest for API endpoints.
- End-to-end: Cypress or Playwright for user flows (signup → add to cart → checkout).
- API documentation & testing: OpenAPI/Swagger or Postman collection.

13. Deployment checklist

- Use environment variables for secrets.
- Build React app: `npm run build` and serve via static host (Netlify, Vercel) or serve from Express.
- Host backend: Heroku, Render, Railway, or VPS.
- Use managed MongoDB (Atlas) with IP allowlist and strong user credentials.
- Set up HTTPS (Let's Encrypt or managed provider).
- Configure CORS correctly for client origin.

14. Security & performance considerations

- Password hashing with bcrypt, strong salt rounds.
- Use HTTPS and secure cookies (SameSite, httpOnly) for auth tokens.
- Rate limiting / brute force protection on auth endpoints.
- Input validation and sanitization (prevent NoSQL injection).
- Index frequently queried fields in MongoDB (e.g., category, price).
- Use pagination and limit responses.
- Cache popular product queries with Redis if scale demands.

15. Future improvements

- Wishlist and product reviews
- Inventory syncing and supplier integration
- Multi-language and currency support
- Microservices for high scale (payments, search)
- Recommendation engine (collaborative filtering)

16. References & resources

- React docs — reactjs.org
- Express docs — expressjs.com
- Mongoose docs

- Stripe integration guides

Appendix: Helpful commands

- Start backend (dev): `nodemon server.js` or `npm run dev`
- Start frontend: `npm start`
- Build frontend: `npm run build`
- Run tests: `npm test`

End of document. Customize this with your repo links, screenshots, and any additional sections you want before exporting to PDF.