

## Day 2:

# Technical Planning Documentation

## Overview

This document outlines the technical plan for developing an E-Commerce Marketplace to empower small businesses and individuals by providing a platform to sell their products online. The technical planning follows the brainstorming from Hackathon Day 1 and incorporates the recommendations from the Day 2 guidelines.

## Key Technologies

- **Frontend:** Next.js
- **Content Management System (CMS):** Sanity
- **Order Tracking and Shipment:** ShipEngine
- **Database:** MongoDB (for authentication)
- **Hosting and Deployment:** Vercel (for frontend) and AWS (for backend)
- **Payment Gateway:** Stripe

## Technical Architecture

### System Overview

1. **Frontend (Next.js):**
  - a. Client-side rendering for speed and responsiveness.
  - b. Server-side rendering for SEO and product page preloading.
  - c. Integration with Sanity CMS for dynamic content.
2. **Backend:**
  - a. REST APIs to manage users, products, orders, and delivery zones.
  - b. Handles business logic, data validation, and integration with external services.

3. **Database (MongoDB):**
  - a. NoSQL database to manage flexible and scalable data structures.
  - b. Collections for products, orders, customers, delivery zones, and user authentication.
4. **CMS (Sanity):**
  - a. Manages dynamic content like banners, featured products, and blog posts.
5. **Order Tracking (ShipEngine):**
  - a. Tracks orders in real time.
  - b. Manages shipment and delivery updates.
6. **Authentication (MongoDB):**
  - a. MongoDB stores user credentials securely.
  - b. Passwords encrypted with hashing algorithms (e.g., bcrypt).
7. **Deployment:**
  - a. Frontend deployed on Vercel.
  - b. Backend deployed on AWS Lambda with serverless architecture.

## System Components and Workflow

1. **User Signup/Login:**
  - a. **Input:** User credentials (email, password).
  - b. **Database:** MongoDB for storing user data securely with hashed passwords.
  - c. **API Endpoint:** POST /register, POST /login, and GET /verify-route for handling user authentication and verification.
  - d. **Outcome:** JWT token issued for session management.
2. **Content Management (Sanity CMS):**
  - a. **Admin Role:** Manages product listings, banners, and blog content.
  - b. **API Integration:** GROQ Queries to fetch content dynamically for frontend.
  - c. **Outcome:** Content stored and updated in Sanity is rendered seamlessly on the Next.js frontend.
3. **Product Browsing and Checkout:**
  - a. **Frontend:** Next.js provides server-side rendering for product pages.
  - b. **Database:** MongoDB stores product details (name, price, stock, description, sizes, etc.).

- c. **API Endpoint:** GET /products for listing, GET /products/:id for details, and POST /products to add products (admin/seller role only).
  - d. **Outcome:** Users browse, add products to cart, and proceed to checkout.
- 4. **Order Management:**
  - a. **Database:** MongoDB stores order data (customer ID, product ID, quantity, status).
  - b. **API Endpoint:** POST /orders to create orders (status defaults to "Pending").
  - c. **Outcome:** Order information processed and stored for tracking. Note: Orders cannot be edited once created.
- 5. **Shipment Tracking (ShipEngine):**
  - a. **Integration:** ShipEngine API for real-time shipment tracking.
  - b. **API Endpoint:** GET /shipments/:orderId to fetch delivery status.
  - c. **Outcome:** Users receive real-time updates on their order delivery.
- 6. **Payment Processing (Stripe, Jazz Cash, EasyPaiza, Kuickpay):**
  - a. **Integration:** Secure payment processing with multiple gateways.
  - b. **API Endpoint:** Payment-related endpoints for handling transactions, including Cash on Delivery (COD) option.
  - c. **Outcome:** Orders processed only after successful payment confirmation or COD selection.

## API Endpoints

### User Management

- **POST /api/auth/register:** Register a new user.
- **POST /api/auth/login:** User login.
- **GET /api/users/profile:** Fetch user profile (requires authentication).
- **PUT /api/users/update:** Update user details.

### Product Management

- **GET /api/products:** List all products.
- **GET /api/products/:id:** Fetch product details by ID.

- **POST /api/products:** Add a new product (requires seller role).
- **PUT /api/products/:id:** Update product details (requires seller role).
- **DELETE /api/products/:id:** Delete a product (requires seller role).

### *Order Management*

- **POST /api/orders:** Create a new order.
- **GET /api/orders:** List all orders for the authenticated user.
- **GET /api/orders/:id:** Fetch details of a specific order.

### *Category Management*

- **GET /api/categories:** List all categories.
- **POST /api/categories:** Add a new category (requires admin role).
- **PUT /api/categories/:id:** Update category details (requires admin role).
- **DELETE /api/categories/:id:** Delete a category (requires admin role).

### *Payment Management*

- **POST /api/payments:** Initiate a payment.
- **GET /api/payments/status:** Fetch payment status.

### *Shipment Management*

- **POST /api/shipments:** Create a new shipment.
- **GET /api/shipments/track:** Track shipment status.

### *Component Details and Interactions*

- **Frontend (Next.js):**
  - Handles user interactions and renders data fetched via APIs.
  - Communicates with the backend for authentication, product data, and order processing.
- **Backend APIs:**
  - RESTful endpoints for CRUD operations on users, products, orders, and shipment data.

- Integrated with ShipEngine and multiple payment gateways for third-party functionality.
- **Database (MongoDB):**
  - Stores user, product, and order data.
  - Provides scalable and flexible schema designs for rapid iteration.
- **Sanity CMS:**
  - Manages dynamic content, ensuring marketing and product information stays up-to-date.

## Data Schema Updates

### *Users:*

- **user\_id:** Unique identifier for the user.
- **username:** User's full name. • **email:** User's email address.
- **password\_hash:** Encrypted password.
- **role:** Role of the user (admin, seller, customer).
- **order\_ids:** List of IDs referencing the user's orders.
- **product\_ids:** List of IDs referencing products added by the user (if seller).

### *Products:*

- **product\_id:** Unique identifier for the product.
- **name:** Name of the product.
- **price:** Rental cost per day/hour.
- **stock:** Availability status of the product.
- **description:** Detailed description of the product.
- **image\_url:** URL of the product image.
- **sizes** (optional): Available sizes for the product.
- **user\_id** (mandatory): ID of the seller who listed the product.

### *Orders:*

- **order\_id**: Unique identifier for the order.
- **customer\_id**: Reference to the customer placing the order.
- **product\_id**: Reference to the rented product.
- **quantity**: Number of products rented.
- **status**: Current status (e.g., Pending, Confirmed, Completed).
- **order\_date**: Timestamp of when the order was placed.

### *Delivery Zones:*

- **zone\_id**: Unique identifier for the delivery zone.
- **zone\_name**: Name of the delivery area.
- **coverage\_area**: Geographic coverage of the delivery zone.
- **drivers**: List of drivers assigned to the zone.

### *Sellers:*

- **seller\_id**: Unique identifier for the seller.
- **name**: Full name of the seller.
- **email**: Email address of the seller.
- **products**: List of product IDs listed by the seller.
- **delivery\_zones**: List of delivery zones managed by the seller.

## Relationships

1. **User and Orders:**
  - a. One user can have multiple orders (One-to-Many relationship).
2. **User and Products:**
  - a. One user can list multiple products (One-to-Many relationship).
3. **Orders and Products:**

- a. One order can include multiple products, and each product can be part of multiple orders (Many-to-Many relationship).
- 4. **Seller and Products:**
  - a. One seller can list multiple products (One-to-Many relationship).
- 5. **Seller and Delivery Zones:**
  - a. One seller can manage multiple delivery zones, and one delivery zone can have multiple sellers (Many-to-Many relationship).
- 6. **Payments and Orders:**
  - a. Each payment is associated with exactly one order (One-to-One relationship).
- 7. **Delivery Zones and Drivers:**
  - a. One delivery zone can include multiple drivers (One-to-Many relationship).

## Integration Details

### Sanity CMS

- Used to manage dynamic content such as:
  - Homepage banners.
  - Category highlights.
  - Blog posts for marketing.
- Sanity's GROQ Query API will be used to fetch content dynamically.

### ShipEngine

- API used to:
  - Generate shipping labels.
  - Track shipments.
  - Provide real-time delivery updates.

### Stripe Integration

- Used for:
  - Processing payments securely.
  - Managing subscriptions (if applicable).
  - Handling refunds and payment disputes.

# Deployment Plan

## Frontend (Next.js)

- **Hosting:** Vercel.
- **CI/CD:** Automatically deploy changes from the GitHub repository.

## Backend

- **Hosting:** AWS Lambda with serverless architecture.
- **Scaling:** Automatic scaling based on traffic.

## Database (MongoDB)

- **Hosting:** MongoDB Atlas.
- **Backups:** Automated daily backups.
- **Scaling:** Horizontal scaling for handling high traffic.

# Security Considerations

1. **Data Encryption:**
  - a. Use HTTPS for all communications.
  - b. Encrypt sensitive user data (e.g., passwords).
2. **Authentication and Authorization:**
  - a. MongoDB stores and validates credentials securely.
  - b. Role-based access control for admin and users.
3. **Payment Security:**
  - a. Use PCI-compliant Stripe APIs for payment processing.
4. **API Security:**
  - a. Rate limiting to prevent abuse.
  - b. Input validation to avoid SQL injection and XSS.

# Monitoring and Maintenance

1. **Monitoring Tools:**



- a. New Relic for application performance.
  - b. CloudWatch for serverless logs.
- 2. **Error Tracking:**
  - a. Sentry for real-time error tracking and debugging.
- 3. **Maintenance:**
  - a. Weekly database maintenance and optimization.
  - b. Regular updates for dependencies to fix vulnerabilities.

## Timeline

- 1. Day 3:
  - a. Set up Next.js project structure.
  - b. Configure Sanity CMS.
  - c. Implement user authentication.
- 2. Day 4, 5:
  - a. Develop product listing and detail pages.
  - b. Integrate ShipEngine for order tracking.
  - c. Build backend APIs for orders and products.
- 3. Day 6:
  - a. Finalize payment gateway integration.
  - b. Implement delivery zones management.
  - c. Test and optimize API performance.
- 4. Day 7:
  - a. Perform end-to-end testing.
  - b. Deploy the application.
  - c. Monitor performance and fix any post-deployment issues.

## Conclusion

This technical plan ensures a robust foundation for the marketplace, leveraging modern technologies to deliver a seamless and scalable platform for small businesses and customers alike.

# Entity-Relationship Diagram

