HACKATHON:02

Technical Planning Document

Introduction

This document provides a detailed technical framework for building an E-Commerce Marketplace designed to support small businesses and entrepreneurs by offering a platform to showcase and sell their products online. This plan incorporates ideas developed during Hackathon Day 1 and feedback from Day 2, aligning with the project's goals.

Core Technologies

- **Frontend Framework:** Next.js
- **Content Management System (CMS):** Sanity
- **Order and Shipping Solution:** ShipEngine
- **Database:** MongoDB (for managing user authentication)
- **Hosting and Deployment:** Vercel (frontend) and AWS (backend)
- **Payment Integration:** Stripe

System Design

1. Frontend (Next.js):

- Optimized for client-side rendering to enhance speed and user experience.
- Implements server-side rendering to boost SEO and enable preloading for product pages.
- Seamlessly integrates with Sanity CMS for managing and displaying dynamic content.

2. Backend:

- Provides RESTful APIs for handling users, products, orders, and delivery zones.
- Ensures robust business logic, data validation, and smooth communication with external services.

System Architecture

3. Database (MongoDB):

- Utilizes a NoSQL database for scalability and flexibility in managing data.
- Stores collections for products, orders, customers, delivery zones, and user authentication details.

4. CMS (Sanity):

- Handles dynamic content such as banners, featured items, and blog posts.

5. Order Management (ShipEngine):

- Provides real-time order tracking.
- Manages shipping and delivery status updates.

6. Authentication (MongoDB):

- Securely stores user credentials.
- Implements encrypted passwords using hashing algorithms (e.g., bcrypt).

7. Deployment:

- Frontend hosted on Vercel for seamless delivery.
- Backend deployed using AWS Lambda with a serverless architecture for scalability and reliability.

System Workflow and Components

1. User Registration and Login:

- **Input:** Captures user credentials (e.g., email and password).
- **Database:** Utilizes MongoDB to store securely hashed passwords.
- **API Endpoints:**
  - `POST /register`: Handles user registration.
  - `POST /login`: Manages login authentication.
  - `GET /verify-route`: Verifies authentication tokens.
- **Outcome:** Issues JWT tokens for secure session handling.

2. Content Management (Sanity CMS):

- **Admin Role:** Facilitates management of product listings, banners, and blog posts.
- **API Integration:** Fetches dynamic content using GROQ queries for the frontend.
- **Outcome:** Dynamic content is rendered seamlessly in the frontend powered by Next.js.

3. Product Browsing and Checkout:

- **Frontend:** Utilizes Next.js for server-side rendering of product pages.
- **Database:** MongoDB stores comprehensive product data, including names, descriptions, prices, and sizes.

System Workflow and Features

3. Product Management and Browsing:

- **API Endpoints:**
  - `GET /products`: Fetches a list of products for browsing.
  - `GET /products/:id`: Retrieves detailed information about a specific product.
  - `POST /products`: Allows sellers or admins to add new products to the catalog.
- **Outcome:** Enables users to explore products, add them to the cart, and proceed to checkout.

4. Order Processing:

- **Database:** Utilizes MongoDB to manage order details, including customer ID, product ID, quantity, and order status.
- **API Endpoint:**
  - `POST /orders`: Creates a new order with the default status set to "Pending."
- **Outcome:** Processes and stores order data for tracking purposes. *Note: Once an order is placed, it cannot be modified*.

5. Shipment Tracking (ShipEngine):

- **Integration:** Leverages the ShipEngine API for real-time shipment updates.
- **API Endpoint:**

    - `GET /shipments/:orderId`: Retrieves the current delivery status of an order.
- **Outcome:** Provides users with up-to-date shipment and delivery information.

6. Payment Processing:

- **Integration:** Supports secure payment processing through multiple gateways (e.g., Stripe, Jazz Cash, EasyPaisa, and Kwickpay).
- **API Endpoint:** Handles transaction-related operations, including support for Cash on Delivery (COD).
- **Outcome:** Orders are confirmed and processed only after successful payment verification or COD selection.

Data System Architecture Documentation Based on Requirements:

1. Frontend Requirements:

- **Update the Browser Section**:
    - Add specific frontend modules like:
        - **Home Page**
        - **Product Listing Page**
        - **Product Details Page**
        - **Cart Page**
        - **Checkout Page**
        - **Order Confirmation Page**
    - Clearly differentiate the admin panel (for CMS management) and the customer storefront UI.
- **Responsive Design**:
    - Indicate that both the Admin and Storefront gateways will be optimized for mobile and desktop.

2. Sanity CMS as Backend:

- Replace **Admin Gateway** with **Sanity CMS Gateway**:
    - Show how Sanity CMS interacts with the Catalog, Customer, and Order services by managing the product data, customer details, and order records.

    - Label the **Catalog Service API** to reflect that Sanity CMS is used to fetch product data.
- Add a **Schema Design Box**:
    - Highlight that schemas in Sanity are designed to:
        - Handle product categories, variants, and inventory.
        - Track customer profiles and their orders.
        - Support order records, promotions, and shipping details.

3. Third-Party APIs:

- In the **Integrations Section**:
    - Add modules for:
        - **Payment Gateway API** (e.g., Stripe, PayPal).
        - **Shipment Tracking API**.
        - Highlight these integrations connecting to the **Checkout Service** and **Order Service** APIs.
- Modify **Checkout Service**:
    - Label it to include "Payment Integration" and "Cart Management."
- Modify **Order Service**:
    - Include "Shipment Tracking" in its responsibilities.

4. Database Layer:

- Indicate that Sanity CMS acts as the primary database for the e-commerce system:
    - Highlight **Product Data**, **Customer Data**, and **Order Records** stored and managed by Sanity.

5. Aggregation Section:

- Ensure the aggregation layer consolidates data from the CMS, APIs, and services to provide:
    - Reports for admin users.
    - Insights for frontend dashboards.

Key Workflows

1. **User Registration:**
    a. The user submits their registration details via the frontend.
    b. A POST request is sent to Sanity CMS (or an optional database) to store the user details.
    c. A confirmation email is sent to the user via an Email API.
2. **Product Browsing:**
    a. The user selects a product category.
    b. The frontend sends a GET request to Sanity CMS to retrieve products for the selected category.
    c. The products are displayed on the user interface.
3. **Order Placement:**
    a. The user adds items to their shopping cart.
    b. At checkout, the order details are sent to Sanity CMS.
    c. Payments are securely processed using a payment gateway (e.g., Stripe or PayPal).
    d. The order is confirmed, and the details are stored in Sanity CMS.
    e. A confirmation message is displayed to the user.
4. **Shipment Tracking:**
    a. The order's tracking ID is passed to the Shipment Tracking API.
    b. The shipment status is retrieved and displayed on the frontend in real time.

Example Product Data Structure:

```
json
{
  "id": 1,
  "name": "Product A",
  "price": 100,
  "description": "Detailed product description here.",
  "stock": 50,
  "image": "https://example.com/images/product-a.jpg"
}
```

1. Fetch All Available Products

- **Endpoint Name:** `/products`
- **Method:** `GET`
- **Description:** Retrieves all available products from Sanity CMS.
- **Response Example:**

```
json
CopyEdit
[
  {
    "id": 1,
    "name": "Product A",
    "price": 100,
    "stock": 50,
    "image": "https://example.com/images/product-a.jpg"
  },
  {
    "id": 2,
    "name": "Product B",
    "price": 200,
    "stock": 30,
    "image": "https://example.com/images/product-b.jpg"
  }
]
```

2. Create New Order

- **Endpoint Name:** `/orders`
- **Method:** `POST`
- **Description:** Creates a new order in Sanity CMS.
- **Payload Example:**

```
json
CopyEdit
{
  "userId": 123,
  "items": [
    {
```

```
      "productId": 1,
      "quantity": 2
    },
    {
      "productId": 2,
      "quantity": 1
    }
  ],
  "totalPrice": 400,
  "status": "pending"
}
```

1. Create New Order

- **Payload Example:**

```
json
CopyEdit
{
  "customerId": 123,
  "products": [
    { "productId": 1, "quantity": 2 },
    { "productId": 2, "quantity": 1 }
  ],
  "paymentStatus": "Pending",
  "shippingAddress": "123 Main St, City, Country"
}
```

- **Response Example:**

```
json
CopyEdit
{
  "orderId": 456,
  "status": "Order Created"
}
```

2. Track Order Status

- **Endpoint Name:** `/shipment`
- **Method:** `GET`
- **Description:** Tracks the status of an order using a third-party Shipment Tracking API.
- **Response Example:**

```
json
CopyEdit
{
  "shipmentId": "789",
  "orderId": 456,
  "status": "In Transit",
  "expectedDeliveryDate": "2025-01-20"
}
```

3. User Registration (Optional)

- **Endpoint Name:** `/register`
- **Method:** `POST`
- **Description:** Registers a new user and stores their details.
- **Payload Example:**

```
json
CopyEdit

{
  "name": "John Doe",
  "email": "johndoe@example.com",
  "password": "securePassword123"
}
```

User Registration (Optional)

Endpoint Name: `/register`

Method: `POST`

Description: Registers a new user in the system.

Payload:

```
json
CopyEdit
{
  "name": "John Doe",
  "email": "john.doe@example.com",
  "password": "password123"
}
```

Response Example:

```
json
CopyEdit
{
  "userId": 123,
  "status": "Registration Successful"
}
```

6. User Login (Optional)

Endpoint Name: /login

Method: POST

Description: Authenticates a user login.

Payload:

```
json
CopyEdit
{
  "email": "john.doe@example.com",
  "password": "password123"
}
```

Response Example:

```
json
CopyEdit
{
  "userId": 123,
  "status": "Login Successful",
  "token": "jwt-token"
}
```

7. Fetch Product Details (Optional)

Endpoint Name: /products/{id}

Method: GET

Description: Retrieves details of a product by its ID.

Data Schema Updates

Users:

- **user_id**: Unique identifier for the user.
- **username**: Full name of the user.
- **email**: User's email address.
- **password_hash**: Encrypted password for security.
- **role**: Role of the user (e.g., admin, seller, customer).
- **order_ids**: List of IDs referencing the user's orders.
- **product_ids**: List of IDs referencing products added by the user (applicable if the user is a seller).

Products:

- **product_id**: Unique identifier for the product.
- **name**: Name of the product.
- **price**: Rental cost per day or per hour.

- **stock**: Availability status of the product.
- **description**: Detailed description of the product.
- **image_url**: URL of the product's image.
- **sizes** (optional): Available sizes for 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 of the order (e.g., Pending, Confirmed, Completed).
- **order_date**: Timestamp of when the order was placed.
- **user_id** (mandatory): ID of the seller who listed the product.

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 delivery 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:** Defines the connection between users (customers) and the orders they place.

1. **User and Products**
    a. A user can list multiple products for sale (One-to-Many relationship).
2. **Orders and Products**
    a. An order can contain multiple products, and each product can belong to multiple orders (Many-to-Many relationship).
3. **Seller and Products**
    a. A seller can list multiple products (One-to-Many relationship).
4. **Seller and Delivery Zones**
    a. A seller can manage multiple delivery zones, and a delivery zone can include multiple sellers (Many-to-Many relationship).
5. **Payments and Orders**
    a. Each payment is linked to one specific order (One-to-One relationship).
6. **Delivery Zones and Drivers**
    a. A delivery zone can be assigned to multiple drivers (One-to-Many relationship).

Integration Details

Sanity CMS

- Utilized for managing dynamic content such as:
    - Homepage banners
    - Category highlights
    - Blog posts for marketing purposes
- Content is dynamically fetched using Sanity's GROQ Query API.

ShipEngine

- Provides API functionalities for:
    - Generating shipping labels
    - Tracking shipments
    - Delivering real-time shipment updates

Stripe Integration

- Utilized for:
    - Secure payment processing

- Subscription management (if applicable)
- Handling refunds and resolving payment disputes

Deployment Plan

Frontend (Next.js)

- **Hosting:** Vercel
- **CI/CD:** Automatic deployment from the GitHub repository

Backend

- **Hosting:** AWS Lambda using a serverless architecture
- **Scaling:** Automatic scaling based on traffic

Database (MongoDB)

- **Hosting:** MongoDB Atlas
- **Backups:** Automated daily backups
- **Scaling:** Horizontal scaling to handle 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. Secure credential storage and validation with MongoDB
   b. Implement role-based access control for admin and users
3. **Payment Security**
   a. Ensure PCI-compliant Stripe APIs are used for payment processing

API Security

- **Rate Limiting:** Prevent abuse by controlling request rates.

- **Input Validation:** Ensure protection against SQL injection and cross-site scripting (XSS).

Monitoring and Maintenance

1. **Monitoring Tools**
    a. **New Relic:** For tracking application performance.
    b. **CloudWatch:** For monitoring serverless logs.
2. **Error Tracking**
    a. **Sentry:** Enables real-time error tracking and debugging.
3. **Maintenance**
    a. Perform weekly database maintenance and optimization.
    b. Regularly update dependencies to address vulnerabilities.