MY MARKETPLACE BUSINESS PLAN DAY 2.

Marketplace Technical Foundation

[Foodtuck]

On the first day of my previous hackathon, I documented my business objectives. I began by defining my E-Commerce business, explaining the reasons behind choosing this field, and outlining the goals for my marketplace. Through careful planning and brainstorming, I determined the steps required to launch my marketplace business and identified the challenges I might encounter while establishing it. I also devised a strategy for my E-Commerce marketplace and considered the factors that would make it appealing to the industry. Furthermore, I emphasized the unique features of my marketplace. Now, I have a solid understanding of how a beginner can initiate their own E-Commerce marketplace business. Moving forward, I will concentrate on brainstorming the next phase, which involves building the technical foundation.

My Next Steps

My upcoming tasks involve setting clear goals and creating a detailed technical plan for my E-Commerce marketplace. This includes designing the system architecture, APIs, and other essential components.

Transitioning to Technical Planning

Based on my understanding, I'll outline the technical requirements necessary for effective planning. Here's a breakdown of my approach:

1. **Frontend Requirements**
    - My first priority is to design a responsive and user-friendly interface for seamless browsing. I plan to include the following essential pages in my hackathon:
        - **Home Page:** Display featured products and categories that I intend to sell.
        - **Product Listing Page:** Showcase products with detailed information for each item.
        - **Product Details Page:** Provide comprehensive details about individual products.
        - **Cart Page:** Enable users to add items to their cart, view selected products, and make modifications.
        - **Checkout Page:** Facilitate users in completing their purchases smoothly.
        - **Order Confirmation Page:** Summarize completed orders, including relevant details for users.
2. **Backend Requirements with Sanity CMS**

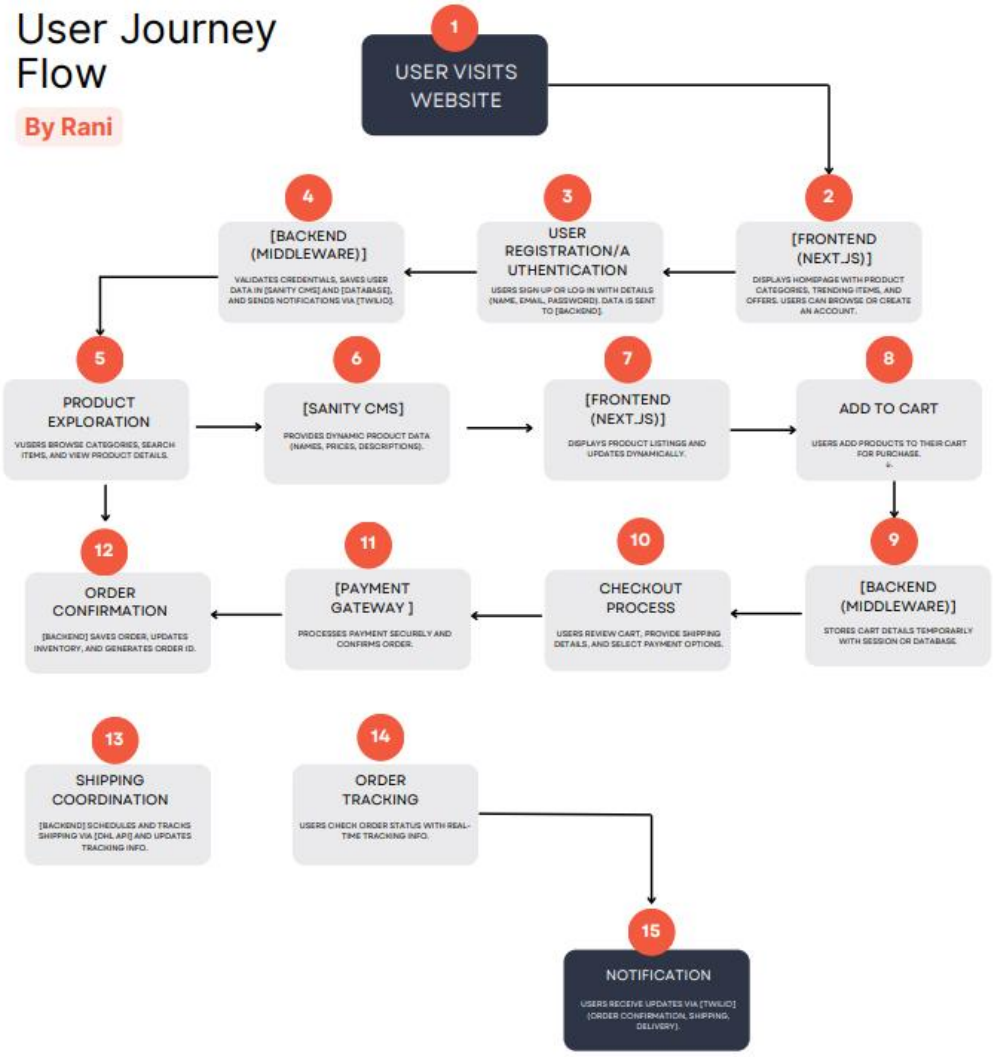
- Next, I'll focus on backend requirements. For my hackathon, I plan to use Sanity CMS, which I've successfully worked with on previous projects. My backend requirements include:
  - **Managing Product Data:** Utilize Sanity CMS to handle product details such as name, price, description, and images.
  - **Storing Customer Details:** Build a backend system to store customer information like names, email addresses, and delivery addresses.
  - **Recording Order Information:** Create schemas to manage order details, including items, total price, and order status.

3. **Third-Party APIs**
   - I'll integrate third-party APIs to enhance the functionality of my hackathon project. Key APIs to include are:
     - **Shipment Tracking API:** Use APIs like Shippo or ShipEngine to track and fetch real-time updates for shipments.
     - **Payment Gateway APIs:** Implement secure payment processing using services like PayPal, Stripe, or similar, to handle transactions safely.

At the end of this process, I'll ensure that all APIs deliver the necessary data to support seamless frontend functionality.

Designing the System Architecture

The system will begin with users visiting the website, where the frontend delivers an interactive and dynamic interface. Users can register or log in, with their details securely stored in the backend and CMS. Products fetched from Sanity CMS will be displayed for browsing, and users can add items to their cart. During checkout, the backend will process orders, update the database and inventory, and handle payments securely through a gateway like Stripe. Shipping details will be managed via third-party APIs (e.g., Shippo or ShipEngine), providing users with tracking updates. Notifications, such as order confirmations and updates, will be sent using services like Twilio, ensuring a smooth and efficient user experience from start to finish.



Summary of My Design System Architecture:

1. **User Visits Website:**
   - The process begins when a user lands on the website. The frontend, built with Next.js, delivers a responsive and dynamic interface.
2. **User Signup/Login:**

    - Users sign up or log in, with the backend validating their details and securely storing them in the CMS and database. A confirmation notification is sent to the user.
3. **Product Browsing:**
    - Users explore the product catalog, with data dynamically fetched from Sanity CMS to ensure up-to-date product information is displayed.
4. **Add to Cart and Checkout:**
    - Users add items to their cart and proceed to checkout, where they provide payment and shipping details.
5. **Payment Processing:**
    - Secure transactions are handled by a payment gateway (e.g., Stripe), and payment confirmation is sent to the backend.
6. **Order Placement:**
    - The backend records the order details in Sanity CMS and updates inventory data in the database.
7. **Shipping and Tracking:**
    - The backend communicates with shipping services (e.g., DHL API) to arrange and track shipments. Tracking details are updated in the database.
8. **Order Tracking:**
    - Users can check their order status on the frontend, with tracking information retrieved from the backend.
9. **Notifications:**
    - Updates such as order confirmations and shipping statuses are sent to users via notification services like Twilio or email.

API Requirements Plan:

Step 1: Understanding API Requirements

To manage data flow between the frontend, backend (Sanity CMS), and third-party services, the main goals for the API include:

1. Fetching data for the frontend (e.g., products, categories).
2. Recording user actions (e.g., placing orders).
3. Integrating with third-party services (e.g., shipment tracking, payment gateways).

Step 2: API Documentation Structure

1. **Product API**
    - **Endpoint:** `/products`
    - **Method:** GET
    - Fetches all available products with details such as name, price, stock, description, and images.
      Response Example:

```
 [
    {
 "id": 1,
 "name": "Product A",
 "price": 200,
 "stock": 40,
"image": "Product1.jpg"
},
    {
"id": 2,
 "name": "Product B",
 "price": 200,
 "stock": 20,
"image": "ProductB.jpg"
 }
]
```

2. **Single Product API**
   - **Endpoint:** `/products/{id}`
   - **Method:** GET
   - Retrieves detailed information about a specific product using its unique ID.
     Response Example:

```
{
    "id": 1,
    "name": "Product A",
    "description": "High-quality product A",
    "price": 100,
    "stock": 50,
    "image": "ProductA.jpg",
    "categories": ["New Arrival", "Top Selling"]
}
```

3. **Add to Cart API**
   - **Endpoint:** `/cart`
   - **Method:** POST
   - Adds a product to the user's cart.
     Request Payload:

```
{
"userId": 456,
 "productId": 1,
 "quantity": 4
}
```

Response Example:

```
{
 "cartId": 456,
 "message": "Item added to cart successfully"
 }
```

4. **View Cart API**
    - **Endpoint:** `/cart/{userId}`
    - **Method:** GET
    - Retrieves all items in a user's cart.
      Response Example:

```
{
    "cartId": 456,
    "userId": 123,
    "items": [
        {
"productId": 1,
 "name": "Product A",
 "price": 100,
"quantity": 2,
 "total": 200
 },
        {
"productId": 2,
 "name": "Product B",
 "price": 200,
 "quantity": 1,
 "total": 200
 }
    ],

    "cartTotal": 400
}
```

5. **Place Order API**
    - **Endpoint:** `/orders`
    - **Method:** POST
    - Submits the user's order and saves the details in Sanity CMS.
      Request Payload:

```
{
 "userId": 123,
 "paymentStatus": "Paid",
 "deliveryAddress": "123 Main Street"
}
```

Response Example:

```
{
"orderId": 789,
 "status": "Order Placed",
 "estimatedDelivery": "2025-01-18"
 }
```

6. **Order Details API**
    - **Endpoint:** `/orders/{orderId}`
    - **Method:** GET

- Retrieves specific order details, including products, total cost, and delivery status.
  Response Example:

```
{
    "orderId": 789,
    "userId": 123,
    "items": [
        {
 "productId": 1,
 "name": "Product A",
"price": 100,
"quantity": 2
},
        {
 "productId": 2,
 "name": "Product B",
 "price": 200,
 "quantity": 1
 }
    ],
    "orderTotal": 400,
    "paymentStatus": "Paid",
    "deliveryStatus": "Shipped",
    "deliveryDate": "2025-01-18"
}
```

7. **Shipment Tracking API**
   - **Endpoint:** `/shipment/{orderId}`
   - **Method:** GET
   - Fetches the shipping status of an order.
     Response Example:

```
{
"shipmentId": 12345,
"orderId": 789,
 "status": "In Transit",
 "expectedDeliveryDate": "2025-01-18"
}
```

8. **Payment Processing API**
   - **Endpoint:** `/payment`
   - **Method:** POST
   - Processes secure payments using a third-party gateway.
     Request Payload:

```
{
"orderId": 789,
 "amount": 400,
"paymentMethod":
 "Credit Card"
}
```

Response Example:

```
{
"paymentId": 56789,
"status": "Payment Successful",
"transactionDate": "2025-01-16T10:30:00Z"
}
```

9. **User Registration API**
    - **Endpoint:** `/users/register`
    - **Method:** POST
    - Registers a new user.
      Request Payload:

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

Response Example:

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

10. **User Login API**
    - **Endpoint:** `/users/login`
    - **Method:** POST
    - Authenticates a user and generates a token.
      Request Payload:

```
{
"email": "john.doe@example.com",
 "password": "securepassword"
 }
```

Response Example:

```
{
"token": "abc123xyz456",
"status": "Login Successful"
}
```

Design System Architecture

This diagram represents the workflow cycle in our technical journey, designed by me. Below is a breakdown of its components:

Frontend (React/Next.js):

- Users interact with the interface, performing actions like browsing products, adding items to the cart, and placing orders.
- The frontend communicates with the backend to request data and trigger actions.

Backend (Middleware):

- Serves as a connection point between the frontend and other system components.
- Handles business logic, such as validating orders and calculating totals.
- Facilitates data exchange with the database, CMS, and third-party APIs.

Database:

- Stores critical data, including:
    1. User information (e.g., accounts, preferences).
    2. Product details (e.g., inventory, pricing).
    3. Order records (e.g., history, status).

Sanity CMS:

- Manages dynamic content like product details, categories, and promotions.
- Enables non-technical users to update content without modifying the codebase.

Third-Party APIs:

- Ensures secure payment processing and provides confirmation notifications.
- Monitors shipments and offers real-time delivery updates.

DESIGN SYSTEM ARCHITECTURE