1. Define System Architecture

- **Purpose**: Lay the groundwork for how different parts of your system will communicate, ensuring scalability, reliability, and performance.

  Key Components:

  - **Frontend**: Typically, this will be a web or mobile app built with modern frameworks (e.g., React, Vue.js, or Next.js for web apps).
  - **Backend**: This is the engine that powers your marketplace, including user management, payment processing, and data storage. You might use a serverless architecture (e.g., AWS Lambda, Firebase Functions) or a traditional backend (Node.js, Django, Ruby on Rails).
  - **Database**: Choose a database to store user information, marketplace listings, transaction data, etc. Options could include relational databases (PostgreSQL, MySQL) or NoSQL databases (MongoDB, Firebase).
  - **CMS**: Since you mentioned using **Sanity CMS**, you will use it to manage and update content like product listings, user profiles, etc. It integrates well with frontend frameworks.
  - **External APIs**: Integrate third-party APIs for specific functions like payment processing (Stripe, PayPal), shipping (Shippo, EasyPost), or messaging (Twilio).

  Example Architecture:

  - Frontend communicates with backend through RESTful or GraphQL APIs.
  - Backend interacts with the CMS and database for content and transactional data.
  - Third-party APIs are called for payment, shipping, and additional functionality.

2. Define Workflows

- **Purpose**: Outline how the user journey flows through the system, from signup to purchasing a product, ensuring seamless interaction with the system.

  Key User Flows:

  - **User Registration/Login**: Users create an account or log in, either through email/password or social media logins (OAuth with Google/Facebook).
  - **Product Listing**: Sellers can upload and manage their listings using the Sanity CMS. These can include images, descriptions, pricing, and other metadata.
  - **Marketplace Browsing**: Buyers can search for products, filter results, and view details.
  - **Order Placement**: Buyers add items to the cart, proceed to checkout, and select payment options.
  - **Order Fulfillment**: Once an order is placed, the backend triggers the shipping API and sends notifications to the buyer and seller.

- **Payment Processing**: Integrate a payment gateway (e.g., Stripe) to handle transactions securely.
- **Post-Transaction**: Sellers confirm shipment, buyers leave feedback, and both parties interact via messaging (if implemented).

Flow Diagram: Visualize the flow using a tool like Lucidchart or Figma. This will help ensure clarity when developers begin implementing each part.

3. Define API Requirements

- **Purpose**: Determine the technical requirements for the APIs that will allow the frontend and backend to communicate. This will include both internal (backend-to-frontend) and external (third-party integrations) APIs.

  Internal APIs:

  - **User Management API**: To handle user registration, authentication, and profile updates.
  - **Product Management API**: For CRUD operations on product listings (Create, Read, Update, Delete).
  - **Order Management API**: To create, update, and track orders.
  - **Payment API**: To securely process payments (integration with Stripe or PayPal).
  - **Shipping API**: For integrating shipping details, calculating costs, and generating tracking numbers.

  External APIs:

  - **Sanity CMS API**: Use the Sanity API to fetch content like product listings or blog posts.
  - **Stripe/PayPal API**: For handling payments securely, including processing refunds and transactions.
  - **Shipping Provider API**: Integrate with third-party services (like Shippo) to calculate shipping costs and generate tracking info.

  Example API Endpoints:

  - `POST /api/users/register`: User registration.
  - `POST /api/users/login`: User login.
  - `GET /api/products`: Fetch a list of products.
  - `POST /api/orders`: Place a new order.
  - `GET /api/orders/:id`: Get details of a specific order.
  - `POST /api/payment`: Handle payment processing.

4. Scalability & Performance Considerations

- **Purpose**: Ensure your marketplace can scale as users grow and that the performance remains optimal under heavy traffic.

  Key Considerations:

  - **Caching**: Use caching (e.g., Redis, Cloudflare) to reduce load on the database and speed up data retrieval.
  - **Load Balancing**: Ensure your server architecture can handle large amounts of traffic using load balancing techniques (e.g., Nginx, AWS ELB).
  - **Database Optimization**: Use indexing, query optimization, and database sharding to ensure quick data retrieval.
  - **Rate Limiting**: Protect your APIs from abuse by implementing rate limiting (e.g., through API Gateway or custom middleware).

5. Security Considerations

- **Purpose**: Protect your users' data and transactions by adhering to security best practices.

  Key Security Practices:

  - **Authentication**: Use secure authentication methods (JWT tokens, OAuth) and ensure user data is encrypted (e.g., bcrypt for password hashing).
  - **SSL/TLS**: Ensure all communications are encrypted using HTTPS.
  - **Data Validation**: Validate input data on both the frontend and backend to prevent SQL injection and other security vulnerabilities.
  - **Third-Party API Security**: Use API keys securely and avoid exposing sensitive data in the frontend.

6. Timeline & Resources

- **Purpose**: Create a realistic timeline for implementation and identify the necessary resources.

  Example Timeline:

  - **Day 3-4**: Backend API development and integration with Sanity CMS.
  - **Day 5-6**: Frontend development, including UI/UX design and API integration.
  - **Day 7**: Testing and debugging.
  - **Day 8**: Final testing, deployment, and optimization.

  Resources:

  - Backend developer(s)
  - Frontend developer(s)
  - UX/UI designer (if applicable)
  - Cloud infrastructure (AWS, Firebase, etc.)

- Testing tools (e.g., Jest for unit testing, Postman for API testing)

Business Overview:

Your marketplace focuses on **high-quality T-shirts**, with a strong emphasis on affordability and direct manufacturing. The T-shirts are **made in-house**, from fabric production to the final product, which is a unique selling point for your business.

Product Schema for Sanity CMS:

You've set up a structured product schema for your T-shirts in **Sanity CMS** to manage your product data. Here's a quick breakdown of your schema fields:

1. **T-Shirt Name**: The name of each T-shirt product.
2. **Description**: A short description of the T-shirt.
3. **Fabric Type**: The type of fabric used (e.g., cotton, polyester).
4. **Design Type**: Whether the design is classic, modern, or custom.
5. **Price**: The retail price of the T-shirt.
6. **Sizes**: Available sizes (S, M, L, XL, XXL).
7. **Color**: The color of the T-shirt.
8. **Image**: Product image with hotspot options for focusing on important parts.
9. **Availability**: Whether the T-shirt is in stock or available for sale.
10. **Wholesale Price**: Price for bulk purchases.
11. **Origin**: Where the fabric or T-shirt is sourced (e.g., local or from China).
12. **Custom Order Option**: Indicates whether this T-shirt can be customized.

Day 2: Technical Plan for T-Shirt Marketplace

1. System Architecture

As per the plan, I will use **Next.js** and **React** for the frontend development and **Sanity CMS** for managing product data. Below is an outline of the system architecture:

Frontend (Next.js + React):

- **Next.js**: This will handle the server-side rendering (SSR) and routing. It will also manage the dynamic rendering of product pages, category pages, and other content.
- **React**: I will use React to create reusable UI components, such as buttons, product cards, cart elements, and product detail pages.

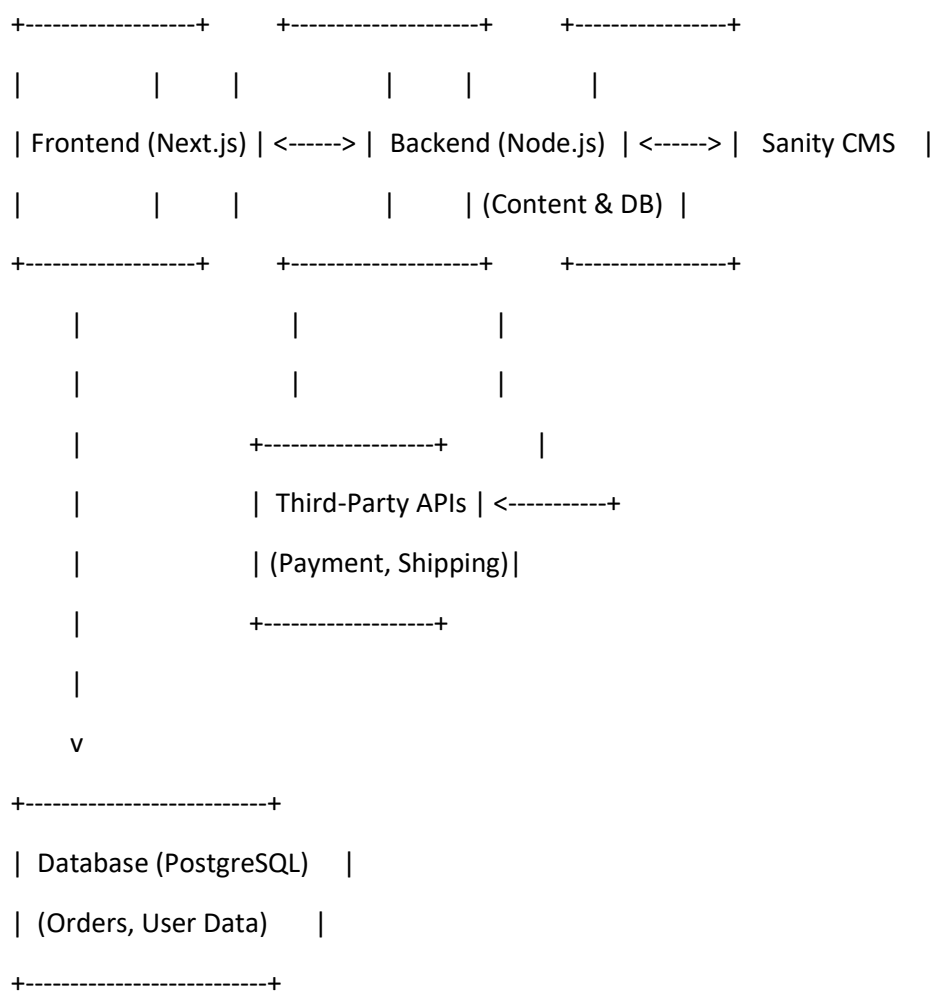
Backend:

- I plan to use **Node.js** (with Express.js) for the backend. It will manage user authentication, order processing, and payment gateway integration.

Database:

- **Sanity CMS**: For managing products, categories, images, and descriptions. Sanity will act as the central content management system.

API Integrations



2. Frontend Development (Next.js + React)

Folder Structure:

my-marketplace/

```
├── components/          # Reusable React components (Product Card, Cart, etc.)
├── pages/               # Next.js pages (Home, Product, Cart, Checkout, etc.)
├── public/              # Static files (Images, Fonts, etc.)
├── styles/              # Global styles (CSS/SCSS)
├── utils/               # Utility functions (API calls, helpers)
├── .env                 # Environment variables (API keys, DB connections)
└── package.json         # Project dependencies and scripts
```

Key Pages:

1. **Home Page (`/pages/index.js`)**:
   - I will fetch products from **Sanity CMS** and display them in a grid layout.
   - The homepage will have filtering options (size, color, design type).
2. **Product Page (`/pages/products/[id].js`)**:
   - This page will show the details of each T-shirt, including description, price, size options, and images.
   - Users can add items to the cart from this page.
3. **Cart Page (`/pages/cart.js`)**:
   - This page will display the selected items in the cart, total price, and provide a checkout option.
4. **Checkout Page (`/pages/checkout.js`)**:
   - I will integrate a payment gateway (Stripe/PayPal) here.
   - Users will enter shipping details before confirming the purchase.
5. **Login/Signup Pages (`/pages/login.js`, `/pages/signup.js`)**:
   - I will implement a secure authentication system using **JWT tokens**.

Fetching Data from Sanity CMS:

- I will set up a connection with **Sanity CMS** to fetch T-shirt product data via **GROQ queries**.

Example of fetching products:

```
// utils/fetchProducts.js
import sanityClient from '@sanity/client';

const client = sanityClient({
  projectId: 'yourProjectId',
  dataset: 'production',
  useCdn: true,
});

export const fetchProducts = async () => {
  const products = await client.fetch('*[_type == "tShirt"]');
  return products;
};
```

Product Listing Page Example:

```
import { fetchProducts } from '../utils/fetchProducts';

const HomePage = ({ products }) => {
  return (
    <div className="product-grid">
      {products.map(product => (
        <ProductCard key={product._id} product={product} />
```

```
      ))}
    </div>
  );
};

export async function getServerSideProps() {
  const products = await fetchProducts();
  return {
    props: { products },
  };
}

export default HomePage;
```

3. Sanity CMS Integration

To integrate **Sanity CMS**, I will follow these steps:

Install Sanity Client:

```
npm install @sanity/client
```

Create a Sanity Client:

```
// utils/sanityClient.js
import sanityClient from '@sanity/client';
```

```
const client = sanityClient({
  projectId: 'yourProjectId', // Replace with your project ID
  dataset: 'production',      // or 'staging' if using a staging dataset
  useCdn: true,
});

export default client;
```

Sanity Schema for Products:

I will ensure the schema for **T-Shirts** in **Sanity** is correctly set up:

```
export default {
  name: 'tShirt',
  title: 'T-Shirt',
  type: 'document',
  fields: [
    { name: 'name', type: 'string', title: 'T-Shirt Name' },
    { name: 'description', type: 'text', title: 'Description' },
    { name: 'price', type: 'number', title: 'Price' },
    { name: 'image', type: 'image', title: 'Image' },
    { name: 'availability', type: 'boolean', title: 'Availability' },
    // Other fields...
  ],
};
```

4. Backend and Database Integration

For the backend, I will use **Node.js** (Express) to manage the user authentication, order processing, and payment gateway integration.

User Model:

```
// backend/models/User.js

const mongoose = require('mongoose');


const userSchema = new mongoose.Schema({

  email: { type: String, required: true, unique: true },

  passwordHash: { type: String, required: true },

  // Additional fields like address, orders, etc.

});


const User = mongoose.model('User', userSchema);

export default User;
```

Order Model:

```
// backend/models/Order.js

const mongoose = require('mongoose');


const orderSchema = new mongoose.Schema({

  userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },

  items: [{ type: mongoose.Schema.Types.ObjectId, ref: 'TShirt' }],
```

```
  totalAmount: { type: Number },
  status: { type: String, enum: ['pending', 'shipped', 'delivered'], default: 'pending' },
});

const Order = mongoose.model('Order', orderSchema);
export default Order;
```

5. Payment Integration (Stripe)

To accept payments securely, I will integrate **Stripe**.

1. **Install Stripe**:

```
npm install stripe
```

2. **Create a Payment Intent**:

```
// backend/payment.js
const stripe = require('stripe')(process.env.STRIPE_SECRET_KEY);

export const createPaymentIntent = async (amount) => {
  const paymentIntent = await stripe.paymentIntents.create({
    amount: amount,
    currency: 'usd',
  });
  return paymentIntent.client_secret;
};
```

3. **Frontend Payment Integration**: I will integrate the Stripe **Checkout** component on the **Checkout page** to securely process payments.

. Security and Best Practices

- **JWT Authentication**: I will use **JWT tokens** for user authentication to secure APIs for user login, order placement, and profile management.
- **Environment Variables**: Sensitive information like **Stripe keys** and **Sanity project ID** will be stored in `.env` files.
- **Data Encryption**: User passwords will be encrypted using **bcryptjs** before being stored in the database.

7. Deployment

Once everything is ready, I will deploy the application as follows:

1. **Frontend (Next.js)**:
   - I will deploy the frontend on **Vercel** or **Netlify** for seamless integration with Next.js.
2. **Backend (Node.js)**:
   - The Node.js backend will be deployed on platforms like **Heroku** or **AWS**.
3. **Database**:
   - I will use **MongoDB Atlas** or **PostgreSQL** for database hosting in the cloud.

Day 2 Activities: Transitioning to Technical Planning

1. Define Technical Requirements

In this phase, I will translate the business goals defined on Day 1 into concrete technical requirements. Here's how I'll approach this:

Frontend Requirements:

- **User Interface**:
  - A user-friendly and intuitive UI for browsing T-shirts.
  - Clear navigation and layout that allows users to filter products based on **size**, **color**, **design type**, and **price**.
- **Responsive Design**:
  - The frontend will be responsive, ensuring a smooth experience across both **mobile** and **desktop** devices.

    - I will implement responsive breakpoints using **CSS Grid** or **Flexbox** to handle various screen sizes effectively.
- **Essential Pages**:
    - **Home Page**: Displays featured products and allows easy navigation to product categories.
    - **Product Listing Page**: Displays a list of products with filtering and sorting options.
    - **Product Details Page**: Provides detailed information about each T-shirt, including images, price, and size options.
    - **Cart Page**: Shows selected products, their quantities, and total price. Users can update their cart or proceed to checkout.
    - **Checkout Page**: Collects user information and payment details. This page will also allow users to select shipping options.
    - **Order Confirmation Page**: Shows the order summary after a successful purchase, including order number and estimated shipping date.

Sanity CMS as Backend:

- **Sanity CMS** will act as the backend for managing product data, customer details, and order records.
    - **Product Data**: Sanity CMS will manage the T-shirt catalog, including details such as name, description, price, image, size options, and availability.
    - **Customer Details**: User details such as name, email, shipping address, and order history will be stored.
    - **Order Records**: Once an order is placed, I will store order details, including customer info, ordered products, and total cost, in Sanity.

Designing Schemas in Sanity:

- I will define schemas in Sanity to ensure data is structured according to business goals. For instance, the **T-shirt** schema will include fields such as `name`, `price`, `image`, `size options`, and `availability`, aligning with the unique selling points of the business (high-quality, customizable T-shirts).

Third-Party APIs:

- **Payment Gateway API**:
    - I will integrate **Stripe** or **PayPal** for payment processing.
    - The API will handle payment authentication, transactions, and confirmation of successful payments.
- **Shipping API**:
    - I will integrate a **Shipping API** (such as **Shippo** or **EasyPost**) for calculating shipping costs, tracking shipments, and providing real-time updates to customers.
- **Product Data API**:
    - **Sanity CMS** will serve as the **Product Data API**, providing product listings, details, and inventory status.

---

2. Design System Architecture

Now, I will create a high-level system architecture diagram that illustrates how the components interact with each other. This will include the **Frontend (Next.js)**, **Sanity CMS**, **Payment Gateway**, and **Shipping APIs**.

System Architecture Overview:

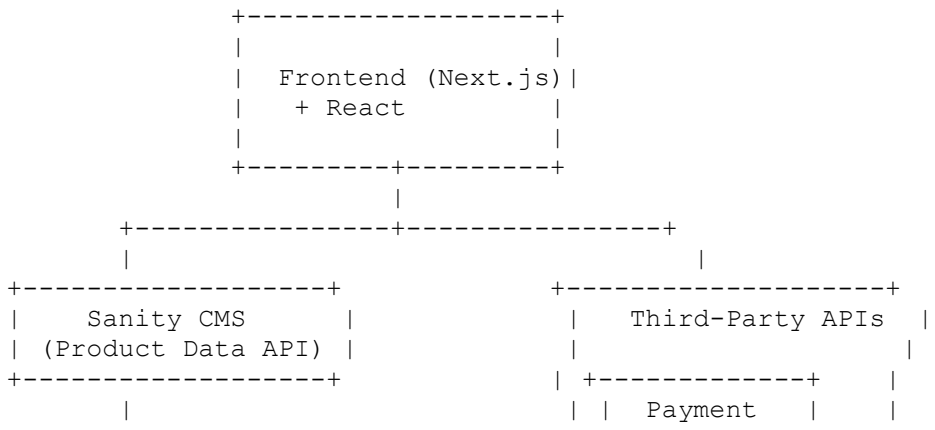
Here's a detailed architecture with the data flow:

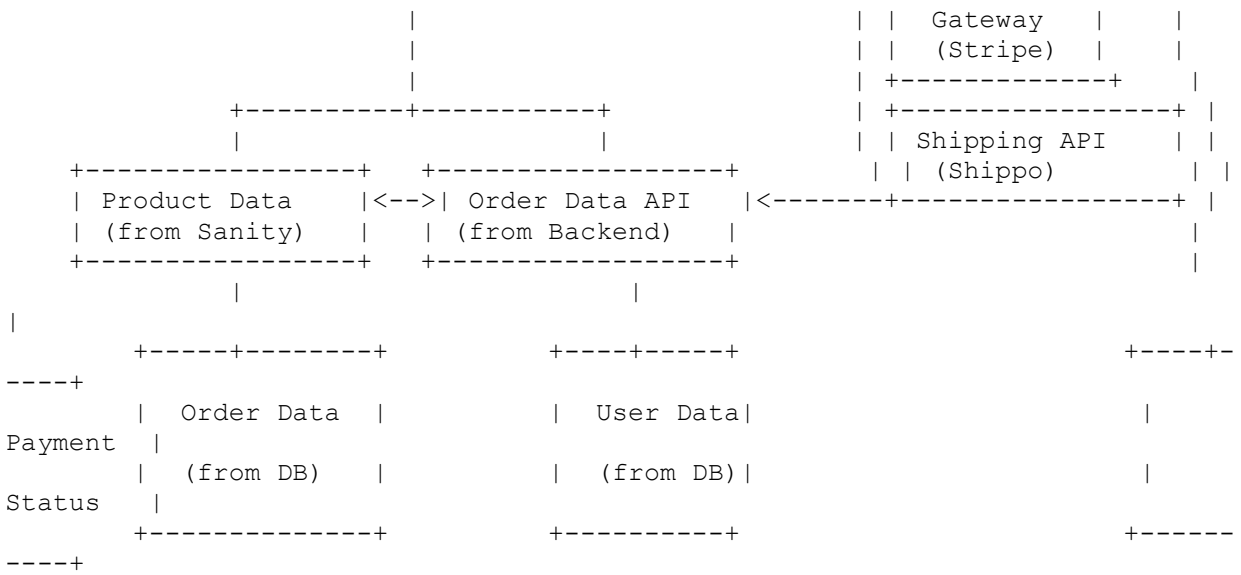
1. **Frontend (Next.js + React)**:
   - **User Actions**: Users interact with the frontend to browse products, view details, add items to their cart, and proceed to checkout.
   - The frontend will make **API requests** to fetch product data from **Sanity CMS** and display it dynamically.
2. **Sanity CMS (Product Data)**:
   - The frontend sends a request to **Sanity's Product Data API** (powered by Sanity CMS) to retrieve product listings, product details, and availability status.
   - Sanity will return structured JSON data that the frontend can use to render the products on the homepage and product pages.
3. **Third-Party APIs**:
   - **Shipping API**: The frontend makes a request to the **Shipping API** for real-time shipping rates and tracking information.
   - **Payment Gateway API**: The frontend interacts with the **Payment Gateway API** (e.g., Stripe) to handle the payment process. Once the payment is successful, the frontend displays an order confirmation page.

Diagram of System Architecture

Below is a simple diagram that shows how the system components interact:

```
sql
Copy
```





Data Flow Breakdown:

1. **Frontend**:
   - When a user visits the homepage, the frontend requests product data from **Sanity CMS** via the **Product Data API**.
   - The frontend displays the product listings dynamically, with options to filter by size, color, and design type.
2. **Product Data API (Sanity CMS)**:
   - The frontend sends a request to **Sanity CMS** for product information. Sanity returns structured product data such as name, description, images, price, sizes, and availability.
3. **Order and Payment**:
   - Once the user adds items to their cart and proceeds to checkout, the frontend collects shipping details and calculates the total price.
   - The frontend interacts with the **Payment Gateway API** (e.g., **Stripe**) to handle the payment transaction.
   - Upon successful payment, the frontend receives confirmation and shows the order confirmation page.
4. **Shipping API**:
   - The frontend will use a **Shipping API** to calculate shipping costs based on the user's location and order details.
   - After payment, users will also be able to track their shipments in real-time through the **Shipping API**.

---

3. Next Steps

With the technical requirements and system architecture outlined, the next steps are:

1. **Set up Sanity CMS**: Define schemas for the product catalog, orders, and customer data.

2. **Develop the Frontend**: Build the pages and UI components (Product Listing, Product Details, Cart, Checkout, etc.) in **Next.js** and **React**.
3. **Integrate Third-Party APIs**: Integrate **Stripe** for payment and **Shippo** (or another shipping API) for shipping tracking.
4. **Build Backend**: Develop a simple Node.js backend to handle order processing, customer authentication, and interaction with the frontend.

Key Workflows and API Requirements

---

1. User Registration

Workflow:

- **User signs up**: The user provides necessary details like email, password, and other required information.
- **Data storage in Sanity**: After successful registration, the user's data is saved in **Sanity CMS**.
- **Confirmation sent to the user**: A confirmation email or message is sent to the user.

API Endpoint:

- **Endpoint Name**: `/api/register-user`
- **Method**: `POST`
- **Description**: Registers a new user in the system.
- **Request Payload**:

```
json
Copy
{
  "email": "user@example.com",
  "password": "password123",
  "name": "John Doe",
  "address": "123 Main Street, City, Country",
  "phone": "+1234567890"
}
```

- **Response**:
  - **Success**:

```
json
Copy
{
  "message": "User registered successfully",
  "userId": "unique_user_id"
```

```
}
```

- **Error**:

```
json
Copy
{
  "error": "Email already exists"
}
```

---

2. Product Browsing

Workflow:

- **User views product categories**: The user can browse products by category.
- **Sanity CMS API fetches data**: Data from Sanity CMS is fetched dynamically, displaying products with details like name, price, and size options.

API Endpoint:

- **Endpoint Name**: `/api/products`
- **Method**: `GET`
- **Description**: Fetches a list of products from **Sanity CMS**.
- **Query Parameters**:
  - `category` (optional): To filter products by category (e.g., "T-Shirts", "Hoodies").
  - `size` (optional): To filter by available sizes (e.g., "M", "L").
  - `color` (optional): To filter by color (e.g., "Red", "Blue").
- **Response**:

```
json
Copy
{
  "products": [
    {
      "id": "product_001",
      "name": "Classic White T-Shirt",
      "price": 499,
      "sizes": ["S", "M", "L", "XL"],
      "color": "White",
      "image": "https://image.url",
      "description": "High-quality white T-shirt made from premium
cotton fabric."
    },
    {
      "id": "product_002",
      "name": "Modern Black T-Shirt",
      "price": 599,
      "sizes": ["M", "L", "XL"],
      "color": "Black",
      "image": "https://image.url",
```

```
      "description": "Stylish black T-shirt with a unique design."
    }
  ]
}
```

---

3. Order Placement

Workflow:

- **User adds items to the cart**: User selects products and adds them to the cart.
- **User proceeds to checkout**: After reviewing the cart, the user proceeds to checkout.
- **Order details saved in Sanity CMS**: Once payment is confirmed, the order details are saved in **Sanity CMS**.

API Endpoint:

- **Endpoint Name**: `/api/place-order`
- **Method**: `POST`
- **Description**: Saves the order details (user info, ordered items, payment status) in **Sanity CMS**.
- **Request Payload**:

```
json
Copy
{
  "userId": "unique_user_id",
  "products": [
    {
      "productId": "product_001",
      "quantity": 2,
      "price": 499
    },
    {
      "productId": "product_002",
      "quantity": 1,
      "price": 599
    }
  ],
  "shippingAddress": "123 Main Street, City, Country",
  "paymentStatus": "success",
  "orderStatus": "pending"
}
```

- **Response**:
  - **Success**:

```
json
Copy
{
  "message": "Order placed successfully",
```

```
  "orderId": "unique_order_id"
}
```

- **Error**:

```
json
Copy
{
  "error": "Order processing failed"
}
```

---

4. Shipment Tracking

Workflow:

- **User tracks their order**: After the order is shipped, the user can track the shipment status in real-time.
- **Shipment tracking fetched via Third-Party API**: The tracking information is fetched from a **third-party API** (e.g., **Shippo**).
- **Real-time updates displayed to the user**: The frontend fetches the latest shipment status and displays it on the order details page.

API Endpoint:

- **Endpoint Name**: `/api/track-shipment`
- **Method**: `GET`
- **Description**: Fetches the shipment status of an order from a third-party API (e.g., **Shippo**).
- **Query Parameters**:
  - `orderId`: The unique order ID to fetch shipment details.
- **Response**:

```
json
Copy
{
  "orderId": "unique_order_id",
  "trackingNumber": "TRACK12345678",
  "status": "In Transit",
  "estimatedDelivery": "2025-01-20",
  "shipmentDetails": {
    "carrier": "Shippo",
    "trackingUrl": "https://shippo.com/track/TRACK12345678"
  }
}
```

---

5. Payment Confirmation

Workflow:

- **Payment details processed**: After checkout, the payment is processed securely using a payment gateway (e.g., **Stripe**).
- **Confirmation sent to user**: A payment confirmation is displayed to the user, and the transaction is recorded in Sanity CMS.

API Endpoint:

- **Endpoint Name**: `/api/payment-confirmation`
- **Method**: `POST`
- **Description**: Records payment details after successful payment and updates the order status.
- **Request Payload**:

```
json
Copy
{
  "orderId": "unique_order_id",
  "paymentStatus": "success",
  "paymentTransactionId": "TRANSACTION12345678",
  "paymentAmount": 1597,
  "paymentMethod": "Stripe"
}
```

- **Response**:
  - **Success**:

```
json
Copy
{
  "message": "Payment processed successfully",
  "orderStatus": "confirmed"
}
```

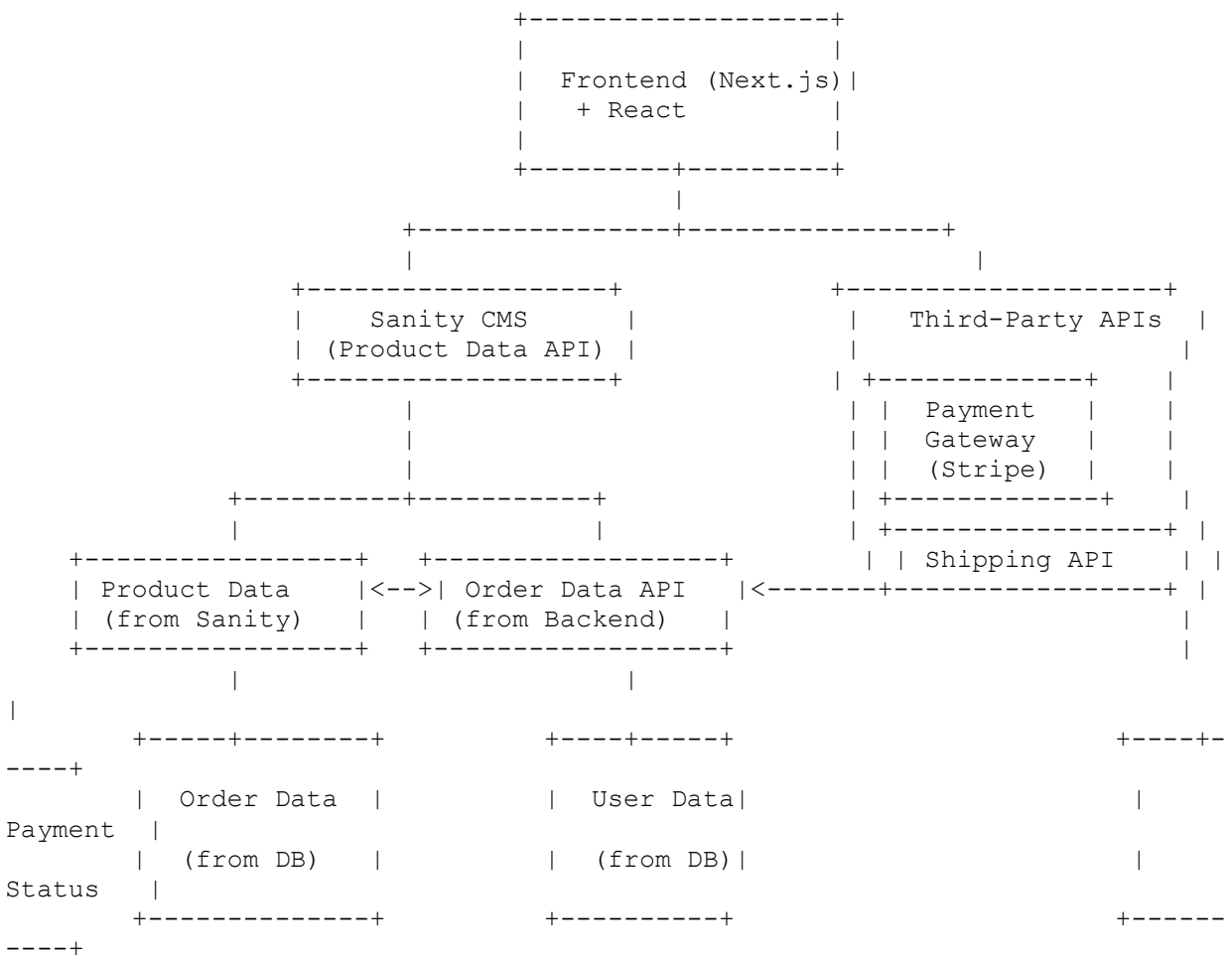
  - **Error**:

```
json
Copy
{
  "error": "Payment processing failed"
}
```

Example System Architecture and Data Flow

Here's how the components interact in this marketplace system:

```
sql
Copy
```



API Integration Plan

- **Sanity CMS** will serve as the central hub for managing product data and orders.
- **Payment Gateway (Stripe)** will handle payment processing, securely storing transaction details.
- **Shipping API (Shippo)** will be responsible for calculating shipping costs and providing real-time shipment tracking.

By defining these API endpoints and workflows, I can ensure that the technical solution is both functional and scalable, supporting the key features of the marketplace such as user registration, product browsing, order placement, shipment tracking, and payment confirmation.

API Endpoints Documentation

---

1. /products

Method: `GET`

Description:

Fetches all available product details from Sanity CMS. This endpoint is used to retrieve information about products in the marketplace, such as product name, price, available stock, and product images.

Request:

- No query parameters are needed.

Response Example:

```
json
Copy
{
  "products": [
    {
      "id": "product_001",
      "name": "Classic White T-Shirt",
      "price": 499,
      "stock": 150,
      "image": "https://example.com/images/product_001.jpg",
      "description": "High-quality white T-shirt made from premium cotton
fabric."
    },
    {
      "id": "product_002",
      "name": "Modern Black T-Shirt",
      "price": 599,
      "stock": 200,
      "image": "https://example.com/images/product_002.jpg",
      "description": "Stylish black T-shirt with a unique design."
    }
  ]
}
```

---

2. /orders

Method: `POST`

Description:

Creates a new order in Sanity CMS. When a user proceeds to checkout and confirms their order, this endpoint saves the order details, including customer information, selected products, payment status, and shipping address.

Request Payload:

```
json
Copy
{
```

```
  "userId": "unique_user_id",
  "products": [
    {
      "productId": "product_001",
      "quantity": 2,
      "price": 499
    },
    {
      "productId": "product_002",
      "quantity": 1,
      "price": 599
    }
  ],
  "shippingAddress": "123 Main Street, City, Country",
  "paymentStatus": "success",
  "orderStatus": "pending"
}
```

Response Example:

```
json
Copy
{
  "message": "Order placed successfully",
  "orderId": "unique_order_id"
}
```

---

3. /shipment

Method: `GET`

Description:

Fetches the real-time shipment status of an order using a third-party API (e.g., Shippo). This endpoint allows customers to track the status of their shipments after the order has been processed and dispatched.

Request Query Parameters:

- `orderId`: The unique ID of the order for which the shipment status is to be tracked.

Response Example:

```
json
Copy
{
  "orderId": "unique_order_id",
  "trackingNumber": "TRACK12345678",
  "status": "In Transit",
  "ETA": "2025-01-20",
  "shipmentDetails": {
    "carrier": "Shippo",
    "trackingUrl": "https://shippo.com/track/TRACK12345678"
  }
}
```

---

4. /express-delivery-status

Method: `GET`

Description:

Fetch real-time delivery updates specifically for **perishable items**. This endpoint is particularly relevant for tracking the status of perishable goods that require expedited delivery.

Response Example:

```
json
Copy
{
  "orderId": 123,
  "status": "In Transit",
  "ETA": "15 mins"
}
```

---

5. /rental-duration

Method: `POST`

Description:

This endpoint is designed for a **rental eCommerce** flow. It adds rental details for a specific product, including rental duration and any deposits required.

Request Payload:

```
json
Copy
{
  "productId": 456,
  "duration": "7 days",
  "deposit": 500
}
```

Response Example:

```
json
Copy
{
  "confirmationId": 789,
  "status": "Success"
}
```

Key API Workflow Overview

1. **Product Browsing (GET /products)**:
   The frontend makes a request to **Sanity CMS** to retrieve the available products and their

details. Users can browse categories, view product information, and select items to add to their cart.

2. **Order Placement (POST /orders)**:
   After the user adds items to the cart and proceeds to checkout, the order details, including customer information, ordered products, and payment status, are sent to **Sanity CMS** to create a new order.
3. **Shipment Tracking (GET /shipment)**:
   After the order is placed and shipped, the user can track the status of their order via the third-party shipment tracking API. This API provides real-time updates on the shipment's location and estimated delivery time.
4. **Perishable Item Delivery (GET /express-delivery-status)**:
   This endpoint is used to track **perishable goods** and ensures that real-time updates are provided, indicating critical information such as expected time of arrival (ETA).
5. **Rental Product Details (POST /rental-duration)**:
   For rental eCommerce items, this endpoint allows the marketplace to capture rental details such as the duration of rental and any required deposits. This is useful for items that are rented out for specific periods.

API Integration Plan

Sanity CMS Integration:

- **Products**: All product data, including details like name, price, size, and stock, will be managed through **Sanity CMS**. The `GET /products` API will be used to fetch this data.
- **Orders**: When an order is placed, the `POST /orders` API will interact with **Sanity CMS** to store the order data.

Third-Party API Integration:

- **Shipment Tracking**: A third-party API like **Shippo** will be used to fetch shipment status updates and track delivery in real time. The `GET /shipment` endpoint will handle this integration.
- **Payment Gateway**: The payment confirmation process will involve the payment gateway (e.g., **Stripe**) to process and confirm transactions. Payment status will be recorded in Sanity via the `POST /orders` endpoint.

Real-Time Updates:

- For perishable items, the **real-time delivery status** (`GET /express-delivery-status`) will provide updates specific to the delivery speed and condition of items that need expedited handling.

Rental System:

- The rental-specific details (e.g., rental duration and deposit) will be handled by the `POST /rental-duration` endpoint, ensuring that rental-specific products are properly processed.

System Architecture and Data Flow

Here is how the components interact within this system:

```
css
Copy
[Frontend (Next.js)]
   |
   | --> [Sanity CMS] <--> [Products API]
   |                        [Orders API]
   |
   | --> [Third-Party APIs] --> [Shipment API] --> [Shippo]
                               --> [Payment API] --> [Stripe]
                               --> [Express Delivery API]
```

By defining these **API endpoints** and the corresponding workflows, I ensure the marketplace operates smoothly, from product browsing and order creation to shipment tracking and payment processing. This comprehensive API documentation will provide a clear structure for developers to implement the required functionality efficiently.

Marketplace Technical Foundation - T-Shirt Marketplace

---

1. System Architecture Overview

System Architecture Diagram:

```
plaintext
Copy
   [Frontend (Next.js)]
          |
   [Sanity CMS (Backend)]
          |
[Products API] --- [Orders API] --- [User Data API]
          |
    [Third-Party APIs]
       |          |
  [Shipment API] [Payment Gateway API]
```

Description of Components:

- **Frontend (Next.js)**: The frontend is developed using **Next.js** and **React**. It handles user interactions, displays product data, and integrates with APIs to manage the cart, orders, and checkout.
- **Sanity CMS**: Serves as the content management system (CMS) and acts as the backend to manage product data, order details, and other essential marketplace information. It handles:
  - Product information (name, description, price, stock, images).
  - Order details (user info, products, shipping details).
  - Customer data (user profile, preferences).
- **Products API**: A **GET** endpoint to fetch product details from Sanity CMS to display products to the user.
- **Orders API**: A **POST** endpoint to handle order placement. It saves the order data in Sanity CMS and interacts with other services like payment gateways and shipment tracking.
- **User Data API**: An endpoint to manage user registrations, sign-ins, and profile information.
- **Third-Party APIs**:
  - **Shipment API**: Allows users to track their orders in real-time using external shipment tracking services like Shippo.
  - **Payment Gateway API**: Integrates with services like **Stripe** for secure payment processing during checkout.

---

2. Key Workflows

1. User Browses Products

- **Step 1**: User lands on the homepage and browses available product categories.
- **Step 2**: The frontend calls the **Products API** (`GET /products`), which fetches product details from Sanity CMS.
- **Step 3**: The system dynamically displays the list of products (name, price, image, description).

2. User Adds Products to Cart

- **Step 1**: User selects a product and adds it to the cart.
- **Step 2**: Cart data is stored in the frontend's state (e.g., using React's context API or Redux).
- **Step 3**: The cart is updated with product quantity, price, and total value.

3. User Proceeds to Checkout

- **Step 1**: User reviews the cart and clicks on the checkout button.
- **Step 2**: The frontend collects user details (shipping address, payment method) and sends the order request to the **Orders API** (`POST /orders`).
- **Step 3**: The order data is saved in Sanity CMS, and a confirmation ID is returned.

4. Payment Processing

- **Step 1**: The user is redirected to the **Payment Gateway API** (e.g., Stripe) for secure payment processing.
- **Step 2**: Once payment is successful, the **Payment Gateway API** sends a confirmation back to the frontend.
- **Step 3**: The order is marked as "Paid" in Sanity CMS.

5. Shipment Tracking

- **Step 1**: After payment confirmation, the system fetches shipment details using the **Shipment API** (`GET /shipment`).
- **Step 2**: The user is able to track their order's delivery status in real-time.

---

3. Category-Specific Instructions

Q-Commerce (Quick Commerce):

- **Key Focus**: Real-time inventory updates, delivery SLA (Service Level Agreement) tracking, and express delivery.

  Example Endpoint: `/express-delivery-status`

  - **Method**: `GET`
  - **Description**: Fetch real-time delivery updates for perishable or high-priority items.
  - **Response Example**:

```
json
Copy
{
  "orderId": 123,
  "status": "In Transit",
  "ETA": "15 mins"
}
```

Rental eCommerce:

- **Key Focus**: Rental duration management, condition reports, and item return handling.

  Example Schema Fields:

  - `rentalDuration`: Defines how long the rental item is booked (e.g., 7 days).
  - `depositAmount`: The refundable deposit for the rental item.
  - `conditionStatus`: Defines the condition of the rented product (e.g., New, Used, Damaged).

  Example API Endpoint: `/rental-duration`

- **Method**: `POST`
- **Description**: Handles rental-specific details for a product.
- **Request Payload**:

```
json
Copy
{
  "productId": 456,
  "duration": "7 days",
  "deposit": 500
}
```

General eCommerce:

- **Key Focus**: Standard eCommerce workflows for product browsing, cart management, and order placement.

  Example Endpoint: `/products`

  - **Method**: `GET`
  - **Description**: Fetch all available product details from Sanity.
  - **Response Example**:

```
json
Copy
{
  "id": 1,
  "name": "Classic White T-Shirt",
  "price": 499,
  "stock": 150,
  "image": "https://example.com/images/product_001.jpg"
}
```

---

4. Data Schema Design

In order to maintain consistency and structure for the marketplace, the **Sanity CMS** will use the following schema for **products** and **orders**.

T-Shirt Product Schema:

```
js
Copy
export default {
  name: 'tShirt',
  title: 'T-Shirt',
  type: 'document',
  fields: [
    { name: 'name', type: 'string' },
    { name: 'description', type: 'text' },
    { name: 'fabricType', type: 'string' },
    { name: 'designType', type: 'string' },
    { name: 'price', type: 'number' },
    { name: 'sizes', type: 'array', of: [{ type: 'string' }] },
```

```
    { name: 'color', type: 'string' },
    { name: 'image', type: 'image' },
    { name: 'availability', type: 'boolean' },
    { name: 'wholesalePrice', type: 'number' },
    { name: 'origin', type: 'string' },
    { name: 'customOrder', type: 'boolean' }
  ]
};
```

Order Schema:

```
js
Copy
export default {
  name: 'order',
  title: 'Order',
  type: 'document',
  fields: [
    { name: 'userId', type: 'string' },
    { name: 'products', type: 'array', of: [{ type: 'reference', to: [{ type:
'tShirt' }] }] },
    { name: 'shippingAddress', type: 'string' },
    { name: 'paymentStatus', type: 'string' },
    { name: 'orderStatus', type: 'string' },
    { name: 'orderDate', type: 'datetime' }
  ]
};
```

---

5. Technical Roadmap

Step 1: Initial Setup

- Set up **Next.js** project for frontend.
- Integrate **Sanity CMS** as the backend for product management.

Step 2: Develop Product and Order APIs

- Develop endpoints for products and orders (`GET /products`, `POST /orders`).

Step 3: Payment and Shipment API Integration

- Integrate **Stripe** for payments.
- Integrate third-party **shipment tracking** API for order tracking.

Step 4: Testing and Debugging

- Test the integration between frontend, backend (Sanity CMS), and third-party APIs.
- Test user workflows, such as browsing products, placing orders, and tracking shipments.

Step 5: Deployment and Launch

- Deploy the solution using **Vercel** or another cloud platform.

- Ensure all APIs and services are functioning as expected.

This **technical documentation** outlines the system architecture, key workflows, API specifications, and roadmap for building the marketplace. By following these instructions, I ensure that the marketplace is built to industry standards, with a clear structure for development and easy integration of all components.

Marketplace Technical Foundation - T-Shirt Marketplace

API Endpoints

Endpoint	Method	Purpose	Response Example
`/products`	GET	Fetches all product details	`{ "id": 1, "name": "Product A", "price": 100, "stock": 150, "image": "image_url" }`
`/orders`	POST	Creates a new order	`{ "orderId": 123, "status": "Success", "paymentStatus": "Paid" }`
`/shipment`	GET	Fetches shipment tracking status	`{ "orderId": 123, "status": "In Transit", "ETA": "15 mins" }`
`/checkout`	POST	Processes checkout and payment	`{ "orderId": 123, "paymentStatus": "Paid", "totalAmount": 499 }`

Sanity Schema Example:

Product Schema

```
js
Copy
export default {
  name: 'product',
  title: 'Product',
  type: 'document',
  fields: [
    {
      name: 'name',
      type: 'string',
      title: 'Product Name'
    },
    {
```

```
      name: 'price',
      type: 'number',
      title: 'Price'
    },
    {
      name: 'stock',
      type: 'number',
      title: 'Stock Level'
    },
    {
      name: 'description',
      type: 'text',
      title: 'Description'
    },
    {
      name: 'image',
      type: 'image',
      title: 'Product Image'
    },
    {
      name: 'availability',
      type: 'boolean',
      title: 'Is Available?'
    },
    {
      name: 'wholesalePrice',
      type: 'number',
      title: 'Wholesale Price'
    },
    {
      name: 'origin',
      type: 'string',
      title: 'Fabric Origin'
    }
  ]
};
```

Order Schema

```
js
Copy
export default {
  name: 'order',
  title: 'Order',
  type: 'document',
  fields: [
    {
      name: 'userId',
      type: 'string',
      title: 'User ID'
    },
    {
      name: 'products',
      type: 'array',
      of: [{ type: 'reference', to: [{ type: 'product' }] }]
    },
    {
      name: 'shippingAddress',
```

```
      type: 'string',
      title: 'Shipping Address'
    },
    {
      name: 'paymentStatus',
      type: 'string',
      title: 'Payment Status'
    },
    {
      name: 'orderStatus',
      type: 'string',
      title: 'Order Status'
    },
    {
      name: 'orderDate',
      type: 'datetime',
      title: 'Order Date'
    }
  ]
};
```

Collaborate and Refine

1. Group Discussions:

- Organize brainstorming sessions with your peers to exchange ideas on the **system architecture** and **API design**. Tools like **Slack**, **Discord**, or **Google Meet** will facilitate easy communication and sharing of ideas.
- Focus on **innovative approaches** to your workflows, such as improving how you fetch products or handle complex data relationships like **user orders** and **product availability**.
- Discuss how **real-time data updates** and **scalable architecture** can benefit your project in the long term.

2. Peer Review:

- Share your **technical documentation** with teammates and mentors for **constructive feedback**. This will help identify any potential gaps in your approach.
- Review each other's **workflows**, **data schemas**, and **API endpoints**. Provide suggestions for improvement on data handling, API efficiency, and security (e.g., payment data).
- Ensure your **technical roadmap** is clear and achievable within the timeline. Regular review sessions can help adjust the project's scope if necessary.

Next Steps

- **Refining Architecture:** Collaborate on refining the system architecture based on feedback and suggestions, ensuring that it is both scalable and secure.

- **Testing API Endpoints:** Test your APIs for functionality, security, and performance before integrating them into the frontend.
- **UI/UX Feedback:** Work closely with the frontend developers to align the technical structure with user interface requirements, ensuring a smooth user experience

Version Control and Collaboration Guidelines

Version Control:

- **Platform:** Use **GitHub** (or similar platforms like GitLab, Bitbucket) for tracking changes, sharing documents, and collaborating effectively.
    - Create a **separate repository** for your project where all team members can push their updates.
    - Commit changes regularly and use descriptive commit messages to maintain a clear history of changes. Example commit messages:
        - "Created Product Schema in Sanity."
        - "Updated API endpoints for order management."
        - "Added checkout flow and order API integration."
    - Create **branches** for each major feature or task (e.g., `feature/product-schema`, `feature/order-api`) to allow parallel development without conflicts.

Divide and Conquer:

- **Individual Submissions:** Although collaboration is encouraged for brainstorming, **each team member** must submit their **individual technical documentation**. This helps maintain personal accountability and ensures the uniqueness of each approach.
    - **Work Independently:** After group discussions and brainstorming, focus on creating your own technical documents, designs, and schemas.
    - **Allow Creativity:** While adhering to the framework provided (e.g., system architecture, workflows, API endpoints), you can bring your **individual perspectives** and **innovative ideas** to the project.

Submission Requirements:

- **Individual Approach:** Ensure that your **final document** reflects **your own understanding and approach**, even if you collaborated with others. This should highlight your technical decision-making and problem-solving skills.

- **Clarity and Structure:** Organize your document with clear headings, diagrams, and bullet points to ensure it's easy to read and follow.

Key Outcome of Day 2:

By the end of **Day 2**, the goal is to have:

1. **Technical Plan Aligned with Business Goals:**
    - A comprehensive **technical plan** that clearly ties the **technical requirements** to the **business goals** defined on **Day 1**. This plan will include system components, APIs, and workflows that are customized for your **marketplace type** (Q-Commerce, Rental eCommerce, or General eCommerce).
2. **System Architecture Visualized:**
    - A **clear diagram** showcasing how the **frontend** (built using **Next.js** and **React**) communicates with **Sanity CMS** for content management and third-party APIs for tasks like **shipment tracking** and **payment processing**.
    - The architecture should represent **specific workflows** tailored to the nature of your marketplace (whether it's focused on **real-time inventory**, **product rentals**, or **general e-commerce**).

System Architecture Example:

For a **General E-commerce Marketplace** (like your T-shirt marketplace):

- **Frontend (Next.js)** communicates with **Sanity CMS** to fetch product details.
- **Third-party APIs** such as **payment gateways** (Stripe, PayPal) and **shipment tracking APIs** (ShipEngine, Aftership) interact with the backend.
- **Product Management** via **Sanity CMS** to handle inventory, descriptions, images, and pricing.

Example Workflow for Your T-Shirt Marketplace:

1. **User Registration/Sign-Up:**
    - User registers → Data is saved to **Sanity CMS** → Confirmation email sent to user.
2. **Product Browsing:**
    - User browses products → **Next.js** frontend calls **Sanity CMS** API to fetch product details.
3. **Order Placement:**
    - User adds items to the cart → Proceeds to checkout → Order data sent to **Sanity CMS** and saved.
    - Payment is processed via **third-party payment gateway (e.g., Stripe)** → Payment status updated in **Sanity CMS**.
4. **Shipment Tracking:**
    - Shipment details fetched from **Third-party API** → Shipment status displayed to the user.

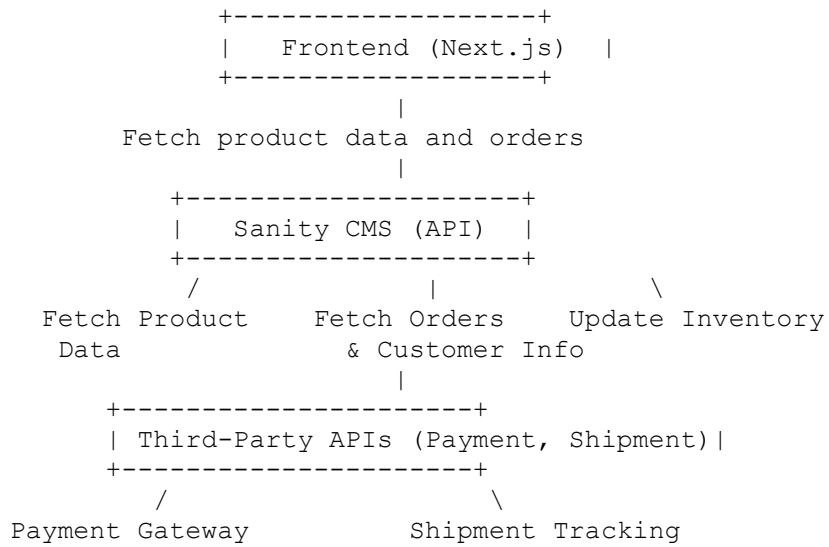
API Endpoints:

Endpoint	Method	Purpose	Response Example
/products	GET	Fetches all product details	{ "id": 1, "name": "Product A", "price": 100, "stock": 150, "image": "image_url" }
/orders	POST	Creates a new order	{ "orderId": 123, "status": "Success", "paymentStatus": "Paid" }
/shipment	GET	Fetches shipment tracking status	{ "orderId": 123, "status": "In Transit", "ETA": "15 mins" }
/checkout	POST	Processes checkout and payment	{ "orderId": 123, "paymentStatus": "Paid", "totalAmount": 499 }

System Architecture Diagram:

Here's a **basic layout** for the architecture diagram (to be created with a tool like Lucidchart, Figma, or Excalidraw):

sql
Copy



Collaboration Tips:

1. **Group Brainstorming:** Hold frequent group sessions to refine your understanding of the **marketplace requirements**. Discuss how to efficiently structure your **data schemas** and ensure your **frontend** aligns with your **backend** design.
2. **Peer Review:** Encourage a culture of **constructive feedback**. After completing your sections (e.g., API endpoints, system architecture), share them with teammates for review and suggestions on improvement.
3. **Iterate:** Based on feedback, revise and optimize your designs, workflows, and APIs to ensure better scalability, security, and user experience.

By working together and refining each component of my project, I will ensure the final product is robust and aligned with both business goals and technical requirements.

1. **Q-Commerce:**
    - **Real-time inventory updates** to ensure users can see available stock in real time.
    - **SLA tracking** to display the estimated time of delivery and ensure that all orders are handled swiftly and efficiently.
2. **Rental eCommerce:**
    - Not directly applicable, but if the concept were to extend to **renting out T-shirts** (e.g., event-based rentals), I would manage features like **rental duration**, **deposit handling**, and **condition tracking**.
3. **General eCommerce:**
    - Typical workflows such as **product browsing**, **cart management**, and **order placement** will be the core focus. I will design these workflows to create a smooth and intuitive customer journey from browsing to checkout.

Detailed API Requirements

To streamline the interaction between the frontend and backend, I've outlined the following key **API endpoints**, methods, and expected responses based on my marketplace needs:

Endpoint	Method	Purpose	Response Example
`/products`	GET	Fetches all product details	`{ "id": 1, "name": "T-Shirt", "price": 100, "stock": 150, "image": "image_url" }`
`/orders`	POST	Creates a new order	`{ "orderId": 123, "status": "Success", "paymentStatus": "Paid" }`
`/shipment`	GET	Fetches shipment tracking status	`{ "orderId": 123, "status": "In Transit", "ETA": "15 mins" }`

Endpoint	Method	Purpose	Response Example
/checkout	POST	Processes checkout and payment	{ "orderId": 123, "paymentStatus": "Paid", "totalAmount": 499 }

These endpoints will ensure smooth operations, from managing product listings to tracking shipments and handling payments.

Sanity Schema Draft

For **Sanity CMS**, I've designed the following schemas to handle key data entities like products, orders, and customers. This structure aligns with the business model I've defined in Day 1.

Product Schema:

```
js
Copy
export default {
  name: 'product',
  type: 'document',
  fields: [
    { name: 'name', type: 'string', title: 'Product Name' },
    { name: 'price', type: 'number', title: 'Price' },
    { name: 'stock', type: 'number', title: 'Stock Level' },
    { name: 'description', type: 'text', title: 'Product Description' },
    { name: 'image', type: 'image', title: 'Product Image' },
    { name: 'availability', type: 'boolean', title: 'Is Available?' },
  ]
};
```

Order Schema:

```
js
Copy
export default {
  name: 'order',
  type: 'document',
  fields: [
    { name: 'orderId', type: 'string', title: 'Order ID' },
    { name: 'products', type: 'array', of: [{ type: 'reference', to: [{ type:
'product' }] }] },
    { name: 'customer', type: 'reference', to: [{ type: 'customer' }] },
    { name: 'totalAmount', type: 'number', title: 'Total Order Amount' },
    { name: 'status', type: 'string', title: 'Order Status' },
    { name: 'paymentStatus', type: 'string', title: 'Payment Status' },
  ]
};
```

This structure will help me manage product listings, order placements, and customer information seamlessly through **Sanity CMS**.

Collaborative Feedback and Refinement

I understand that effective collaboration is essential for ensuring the quality of my technical documentation and system design. Here's how I plan to incorporate feedback:

1. **Brainstorming:** I will engage in group discussions with peers to share insights and solutions for common challenges, such as API design and workflow improvements.
2. **Peer Reviews:** I'll share my technical plans with teammates and mentors to receive constructive feedback, refine my system architecture, and make any necessary adjustments.
3. **Iterative Refinement:** Based on the feedback, I will refine my **system architecture diagrams**, **API documentation**, and **data schemas** to ensure they are optimal and scalable.

Portfolio-Ready Submission

At the end of Day 2, I will have:

- A polished, professional document outlining the technical foundation of my marketplace.
- Clear and detailed **API documentation**, **system architecture diagrams**, and **Sanity CMS schemas**.
- A fully aligned **technical plan** that reflects the business goals established on Day 1 and is ready for implementation.

This document will not only serve as the blueprint for the actual implementation phase but will also be a valuable addition to my **portfolio**, showcasing my ability to design and implement a complete, full-stack e-commerce solution.