***Avion Furniture project :***

**1. Project Understanding**

* **Objective**: Define the purpose of your project. For example:
  + *Avion Furniture is an e-commerce platform for selling furniture online with features like product browsing, cart management, checkout, and payment processing.*
* **Core Features**:
  + User Authentication (Sign Up, Login)
  + Product Catalog (List, Filter, Search)
  + Shopping Cart
  + Checkout & Payment
  + Order Tracking
  + Admin Panel for Managing Products and Orders

**2. Identify Components**

Create a list of the components and technologies you're using:

* **Frontend**:
  + Framework: React.js (Next.js if applicable)
  + UI Library: Tailwind CSS / Chakra UI
  + Components: Reusable UI components (e.g., Card, Button, Input, Modal)
* **Backend**:
  + Framework: Node.js (with Express.js)
  + Database: MongoDB / PostgreSQL / MySQL
  + APIs: RESTful or GraphQL APIs for communication
* **Storage**:
  + LocalStorage/SessionStorage (for cart and session management)
  + Cloud Storage (e.g., AWS S3 for product images)
* **Payment Gateway**:
  + Integration with Stripe, Razorpay, or PayPal
* **Authentication**:
  + JWT-based authentication or OAuth (e.g., Google Login)
* **Deployment**:
  + Vercel/Netlify for frontend
  + AWS/DigitalOcean/Heroku for backend

**3. Break Down User Flows**

Identify key workflows in your project and divide them into smaller steps. Common workflows include:

**User Flow for Shopping:**

1. **Home Page**:
   * User lands on the homepage.
   * Browse categories or featured products.
2. **Product Page**:
   * View product details, images, and price.
   * Add item to cart.
3. **Cart Page**:
   * View selected items with quantity and total price.
   * Proceed to checkout.
4. **Checkout Page**:
   * Enter shipping details.
   * Review the order summary and total amount.
   * Select payment method.
5. **Payment**:
   * Process payment via gateway.
   * Receive confirmation and redirect to the order confirmation page.
6. **Order Confirmation**:
   * Show order ID and tracking details.

**4. Create Flow Diagram**

A flow diagram visualizes the steps and their connections. Use a tool like Figma, Lucidchart, or Draw.io to create it.

**Example Flow Diagram:**

1. **Start** → Home Page → Product Listing → Product Details → Add to Cart
2. **Cart** → Checkout → Shipping Details → Payment Gateway → Order Confirmation → **End**

Below is how you can structure it in detail:

* **Actors**: User, Admin, Backend, Payment Gateway
* **Processes**: Browsing, Adding to Cart, Checkout, Payment, Order Management

**5. Document Architecture**

Include an architecture diagram to explain the technologies and how they interact:

**Frontend-Backend Communication:**

1. **Frontend**:
   * User sends a request (e.g., Add to Cart) → API Call → Backend
2. **Backend**:
   * Processes request → Updates Database → Sends Response to Frontend
3. **Database**:
   * Stores user, product, cart, and order data.
4. **Payment Gateway**:
   * Handles secure payment processing.
5. **Cloud Services**:
   * Hosts static files, images, and deploys the application.

**6. Add a Detailed Explanation of Tools/Technologies**

Explain each tool and why you chose it. For example:

* **React.js**: For building a dynamic and responsive user interface.
* **Node.js with Express**: For creating robust backend APIs.
* **MongoDB**: For storing product, user, and order data.
* **Stripe**: For secure and flexible payment processing.

**7. Write a Summary**

Conclude your document with a summary of:

* Project Purpose
* Features
* Technologies Used
* Future Enhancements

# **Simple Flow Diagram for Cart Functionality**

## **Flow Description**

### 1. User Interaction

* **Add to Cart**:
  + User clicks "Add to Cart" on a product page.
  + Sends product details (name, price, quantity, and image) to the frontend.
* **View Cart**:
  + User navigates to the cart page.
  + Frontend fetches the cart items stored locally or in a database.
* **Update Quantity**:
  + User clicks "+" or "-" to update product quantity.
  + Updates are reflected dynamically in the UI and stored locally or in the database.
* **Remove Item**:
  + User clicks "Remove" on a product.
  + The item is removed from the cart.

### 2. Backend Actions

* On "Add to Cart" or "Update Quantity":
  + Request sent to backend API to store or update the cart.
  + Backend updates the cart in the database and sends confirmation.
* On "View Cart":
  + Backend retrieves cart items for the user.

### 3. Checkout

* User clicks "Checkout".
* Backend calculates subtotal, shipping cost, and total.
* Redirect to payment gateway.

### 4. Payment Processing

* User completes payment via the payment gateway.
* Payment confirmation received by the backend.
* Order details saved in the database.

### 5. Order Confirmation

* Backend sends a confirmation email to the user.
* Frontend redirects to the order confirmation page.

## **Components and Technologies Used**

### Frontend

* **React/Next.js**: For the UI and interaction logic.
* **State Management**: LocalStorage or Redux for managing cart state.

### Backend

* **Node.js/Express**: For handling API requests.
* **Database**: Storing cart items, product details, and order data.

### Other Tools

* **Sanity CMS**: For managing product data.
* **Payment Gateway (e.g., Stripe)**: For secure payment processing.

## Diagram Structure

**1. Add to Cart Flow**

* User clicks "Add to Cart"
  + Frontend triggers API request to backend.
  + Backend saves item to the cart in the database.
  + Response sent to the frontend to update UI.

**2. View Cart**

* User opens cart page.
  + Frontend fetches cart items from the backend or LocalStorage.

**3. Update/Remove Item**

* User updates/removes items.
  + Frontend updates UI.
  + API call to backend to save changes.

**4. Checkout Flow**

* User clicks "Checkout".
  + Backend calculates totals.
  + Payment gateway is initialized.

**5. Payment Processing and Order Confirmation**

* Payment is processed.
  + Backend saves order.
  + Frontend redirects to confirmation page.

## Notes

* Ensure robust error handling in all stages.
* Optimize API calls to reduce latency.
* Implement authentication for secure transactions.

**Defining Technical Requirements**

The **first step** in building a system is to **translate business goals into technical requirements**. This involves outlining what the system should do and how it will work in terms of technology.

**Frontend Requirements:**

* **User-Friendly Interface for Browsing Products**: The frontend should be designed with an emphasis on ease of use and navigation. It should make it easy for customers to find products, explore categories, and make purchases. This often includes:
  + Intuitive navigation (search, categories, filters)
  + Clear product details (images, prices, descriptions)
  + Smooth user experience (fast page loading, easy-to-understand buttons).
* **Responsive Design for Mobile and Desktop**: Since customers use a variety of devices (phones, tablets, desktops), your application must be **responsive**. This means it should adjust seamlessly to different screen sizes using tools like **CSS media queries**, flexible grid layouts, and frameworks like **Tailwind CSS** or **Bootstrap**.
* **Essential Pages**:
  + **Home Page**: An overview of the marketplace, including featured products and promotions.
  + **Product Listing Page**: Displays a list of products with sorting and filtering options.
  + **Product Details Page**: Shows detailed information about a specific product.
  + **Cart**: Allows users to review and modify their selections before checkout.
  + **Checkout Page**: Where the user completes the purchase, providing shipping and payment information.
  + **Order Confirmation**: Confirms the purchase, often includes an order ID and estimated delivery details.

**Backend Requirements using Sanity CMS:**

* **Sanity CMS** will serve as the **Content Management System (CMS)** and **backend** for storing product data, customer information, and order records.
  + **Sanity CMS** allows you to create customizable schemas for different types of data like products, categories, customer profiles, and orders.
  + You need to design **schemas** that represent each of these entities in a way that makes it easy to manage them via the Sanity platform.
  + These schemas help to store product data (e.g., name, description, price, images) and other crucial marketplace details in an organized way.

**Third-Party APIs Integration:**

* **Shipment Tracking API**: To provide real-time shipping details and tracking updates, you will integrate a third-party shipment tracking API (e.g., UPS, FedEx).
* **Payment Gateway API**: You will need to integrate APIs from payment providers (e.g., Stripe, PayPal) for secure payment processing.
* **Other Backend Services**: You may also need APIs for other services such as authentication (e.g., OAuth), email notifications, or inventory management.

These integrations ensure that the **frontend** (user interface) has all the necessary data and functionality to support browsing, purchasing, and order management.

**2. Design System Architecture**

The **System Architecture** represents the high-level structure of how your application's components work together. It’s a **blueprint** of how all systems will communicate, and it shows the data flow and interactions between each part.

**High-Level System Architecture Overview:**

* **Frontend (Next.js)**:
  + The **Next.js** framework is used for building the frontend of the application. It's a popular framework built on top of React, known for its server-side rendering and static site generation capabilities, making it ideal for SEO and performance.
  + The frontend will send requests to various APIs to fetch data dynamically (e.g., product data, order status).
* **Sanity CMS** (Backend):
  + Sanity CMS acts as the **content database**. It stores structured content (products, customers, orders) and exposes this data through APIs.
  + The **Product Data API** is responsible for fetching product listings, product details, categories, and prices.
  + The frontend will make API requests to Sanity CMS to retrieve this information and display it on the site.
* **Third-Party APIs**:
  + **Shipment Tracking API**: When an order is placed, shipment tracking information is fetched from a third-party API to provide real-time updates to the user about where their order is.
  + **Payment Gateway**: When a user places an order and proceeds to checkout, payment details are sent to the payment gateway API (e.g., Stripe). The gateway processes the payment and returns a success or failure response.

**Typical Data Flow:**

1. **User Browses Products**:
   * The user visits the frontend of the marketplace, which loads the **product listing** page.
   * The frontend makes a request to the **Product Data API** (powered by Sanity CMS) to fetch a list of products and display them dynamically.
2. **Placing an Order**:
   * When the user adds items to their cart and proceeds to checkout, the **order details** are sent to the **Sanity CMS** via an API request.
   * Sanity CMS stores the order data (e.g., products, quantities, customer information).
3. **Shipment Tracking**:
   * After the order is processed, the **shipment tracking information** is fetched from a third-party API and displayed to the user in real-time.
   * This could include updates such as "shipped," "out for delivery," and "delivered" status.
4. **Payment Processing**:
   * The user enters their payment details, which are securely processed through a **payment gateway** (e.g., Stripe or PayPal).
   * Once the payment is confirmed, the frontend receives the confirmation and displays an **order confirmation page** to the user.
5. **Order Confirmation and Data Storage**:
   * The frontend displays the order confirmation and sends the final confirmation to **Sanity CMS** for record-keeping.
   * This ensures that the system has up-to-date data on the user’s order, including payment and shipping status.

**How This Architecture Supports Business Goals**

By following this structure, the system achieves the following business goals:

* **Scalability**: The use of Sanity CMS for data management ensures that the system can scale as the product catalog and customer base grow.
* **Efficiency**: By using third-party APIs, the system offloads complex tasks (e.g., payment processing, shipment tracking) to reliable external services, ensuring smooth operations.
* **User Experience**: The responsive frontend built on Next.js ensures a seamless experience across devices, while dynamic product listings and real-time tracking information help keep users informed.

**Final Thought**

In technical planning, it’s crucial to **visualize** how each part of the system communicates and functions, which helps you identify potential bottlenecks, integrations, and dependencies before moving to implementation. Using tools like **Lucidchart**, **Figma**, or **Excalidraw** helps in creating system diagrams that clarify these interactions.

This architectural overview ensures your application is not only functional but also optimized for scalability, usability, and maintainability.

Bottom of Form

### +-----------------+

### | Frontend |

### | (Next.js UI) |

### +-----------------+

### |

### |

### +-------------+-------------+

### | |

### +------------------+ +-------------------+

### | Sanity CMS | | 3rd Party APIs |

### | (Product Data) | | (Payment, |

### | | | Shipment) |

### +------------------+ +-------------------+

### | |

### +---------------------+ +--------------------+

### | Product Data API | | Payment Gateway |

### | (Sanity API) | | API (Stripe/PayPal)|

### +---------------------+ +--------------------+