**Baterly — C2C Marketplace with Negotiation & Analytics**

**By**

**Shreya Dhuri -- AF04957517**

**Vedant More – AF04954640**

**Kajal Jaghtap– AF04955270**

**Index**

|  |  |  |
| --- | --- | --- |
| **Sr.no** | **Topic** | **Page no** |
| 1 | Title of Project | 1 |
| 2 | Acknowledgement | 3 |
| 3 | Abstract | 4 |
| 4 | Introduction | 5 |
| 5 | Objective | 6 |
| 6 | System Analysis | 7 |
| 7 | System Design | 22 |
| 8 | Screenshots | 32 |
| 9 | Coding | 39 |
| 10 | Testing | 46 |
| 11 | Report | 47 |
| 12 | Future Scope | 48 |
| 13 | Conclusion | 48 |
| 14 | Bibliography | 49 |
| 15 | References | 49 |

# Acknowledgement

**The project *“Baterly — C2C Marketplace with Negotiation & Analytics”* is the work carried out by Vedant Gajanan More under the valuable guidance of mentors, and with constant encouragement from family and friends. Their insights, motivation, and support played an essential role in completing this project.**

# Abstract

# Baterly is a Consumer-to-Consumer (C2C) marketplace designed to enhance peer-to-peer trading through structured negotiations and built-in analytics. Unlike traditional platforms where bargaining is unstructured, Baterly enables sellers to list items with condition options (New, Like New, Used), set prices, and define minimum acceptable offers. Buyers can place offers and counter-offers, each recorded with its own contextual message thread.

# The system also includes a complete workflow: Accept → Mark as Paid → Mark as Shipped → Mark as Received, ensuring transparency. Additionally, analytics features such as per-product maximum views, recently viewed items, and top interest categories provide sellers with insights into market demand trends

# Introduction

In the era of digital trade, online marketplaces have become the go-to platforms for buying and selling goods. However, most existing platforms lack structured negotiation mechanisms, forcing buyers and sellers to negotiate externally. This leads to inefficiency and lack of transparency.

Baterly bridges this gap by introducing:

* Structured in-system negotiations,
* A clear transaction workflow, and
* Analytics that give sellers actionable insights about product demand.

By allowing a single user to act as both buyer and seller, Baterly enhances flexibility, usability, and engagement.

## Objective of the Present Work

The objectives of the present work are as follows:

* To build a user-friendly C2C marketplace platform.
* To enable sellers to list items with condition options and set minimum acceptable offers.
* To implement a structured negotiation system with contextual message threads.
* To design a complete transaction workflow ensuring trust and transparency.
* To integrate analytics for product popularity, demand trends, and recently viewed items.
* To ensure scalability, security, and smooth usability.

**System analysis**

**Existing System**

Current online marketplaces allow product listing and purchases but lack structured negotiation. Bargaining happens informally (e.g., via chat apps), making the process inefficient. Sellers also lack insights into buyer interest and market demand.

**Proposed System**

**Baterly introduces:**

* Negotiation threads for each deal,
* Workflow-driven transactions,
* Analytics dashboards for sellers, and
* A dual-role account system (user can be both buyer and seller).

This makes the process transparent, data-driven, and user-centric.

**Feasibility Study**

* **Technical Feasibility:** Built using React.js, Node.js, Express.js, and MongoDB – reliable, scalable, and open-source.
* **Operational Feasibility:** Easy-to-use UI with structured workflows.
* **Economic Feasibility:** Cost-effective as it uses open-source technologies.

**Requirements**

* **Functional:** User authentication, item listing, negotiation, messaging, workflow tracking, analytics.
* **Non-Functional**: Secure, scalable, responsive, and mobile-friendly.

**System Design**

**Modules**

### ****1. User Module****

* **Purpose:**  
  This module handles user management and authentication. Since Baterly is a C2C platform, a single account must support both **buyer** and **seller** roles.
* **Key Features:**
  + User registration with email/phone verification.
  + Secure login with JWT-based authentication.
  + Profile management (name, contact details, address).
  + Role flexibility – the same account can list products (seller role) and make offers (buyer role).
  + Session management and password reset functionality.
* **Technical Implementation:**
  + **Frontend:** React.js forms for login/signup.
  + **Backend:** Node.js & Express.js with authentication middleware.
  + **Database:** MongoDB collection Users storing credentials, profile, and activity logs.

**2. Listing Module**

* **Purpose:**  
  Enables sellers to add, update, and manage product listings. Listings are the core of the marketplace.
* **Key Features:**
  + Item details: title, description, images, category.
  + Condition selection: **New, Like New, Used**.
  + Pricing details: seller sets asking price + minimum acceptable offer.
  + Ability to edit or remove listings.
  + Visibility control (active/inactive status).
* **Technical Implementation:**
  + **Frontend:** Form for listing creation and dashboard for managing items.
  + **Backend:** API endpoints for CRUD operations on listings.
  + **Database:** MongoDB collection Listings storing product metadata and seller ID.

**3. Negotiation Module**

* **Purpose:**  
  Facilitates structured buyer-seller communication with offers and counter-offers.
* **Key Features:**
  + Buyers can submit offers lower than the listed price (but not below seller’s minimum acceptable offer).
  + Sellers can **accept, reject, or counter** an offer.
  + Each negotiation is stored in its own **thread**, showing a history of offers and messages.
  + Built-in messaging for clarification between buyer and seller.
* **Technical Implementation:**
  + **Frontend:** Real-time chat-like interface for offers.
  + **Backend:** WebSocket or polling APIs for instant negotiation updates.
  + **Database:** MongoDB collection Negotiations storing offer details, timestamps, and thread ID.

**4. Deal Workflow Module**

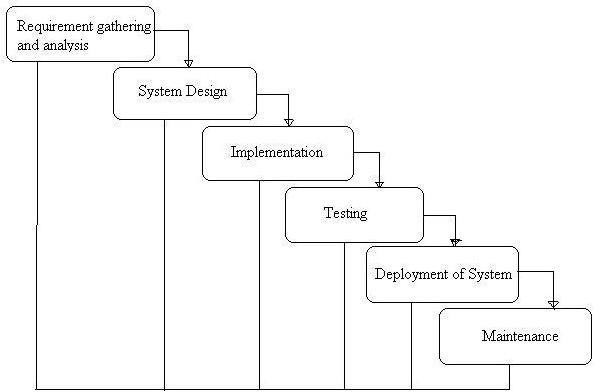
* **Purpose:**  
  Provides a **step-by-step transaction flow** to ensure trust and clarity in deals.
* **Workflow Steps:**
  1. **Accept** – Seller accepts buyer’s offer.
  2. **Mark as Paid** – Buyer confirms payment. (In future: integrated payment gateway).
  3. **Mark as Shipped** – Seller updates shipment status.
  4. **Mark as Received** – Buyer confirms delivery.
* **Key Features:**
  1. Status tracking for every deal.
  2. Both buyer and seller receive notifications at each stage.
  3. Dispute management (future scope – admin intervention if workflow is stuck).
* **Technical Implementation:**
  1. **Frontend:** Progress tracker UI showing each step.
  2. **Backend:** REST APIs for updating workflow states.
  3. **Database:** Deals collection with status logs and linked negotiation ID.

**5. Analytics Module**

* **Purpose:**  
  Provides insights into user behavior and market demand to help sellers make informed decisions.
* **Key Features:**
  + **Per-product views:** Count how many times a listing was opened.
  + **Recently Viewed:** Buyers can revisit their browsing history.
  + **Top Interest Categories:** Aggregated data shows which categories are trending.
  + **Conversion Rate Insights (Future):** Ratio of views to successful deals.
* **Technical Implementation:**
  + **Frontend:** Dashboard with charts and filters.
  + **Backend:** APIs aggregating analytics data using MongoDB queries.
  + **Database:** Analytics collection storing view counts, categories, and user interaction logs.

**6. Admin Module**

* **Purpose:**  
  Allows platform administrators to maintain order, security, and integrity of the marketplace.
* **Key Features:**
  + User management (ban suspicious accounts, reset roles).
  + Category management (add/remove product categories).
  + Monitor active negotiations and deals.
  + Handle disputes between buyers and sellers.
* **Technical Implementation:**
  + **Frontend:** Admin dashboard with restricted access.
  + **Backend:** Role-based access control with admin-only APIs.
  + **Database:** Admin logs stored in AdminActions collection for auditing.



Phases of Development

### Requirement Analysis & System Study

* + Identifying project goals, challenges, and functional specifications.
  + Gathering stakeholder requirements and defining core functionalities.

### System Design

* + Structuring the **database, modules, and architecture**.
  + Designing **user interfaces** for optimal accessibility.

### Implementation (Coding)

* + Backend development using **Node JS**
  + Frontend design using **HTML, CSS, React JS**.
  + Database integration with **MySQL**.

### Testing & Debugging

* + Unit testing, integration testing, and usability checks.
  + Debugging for performance improvements.

### Deployment & Maintenance

* + Hosting on a scalable environment.
  + Continuous updates for feature enhancements.
  1. Data Flow Diagram:

A Data Flow Diagram (DFD) is a traditional visual representation of the information flows within a system. A neat and clear DFD can depict the right amount of the system requirement graphically. It can be manual, automated, or a combination of both.

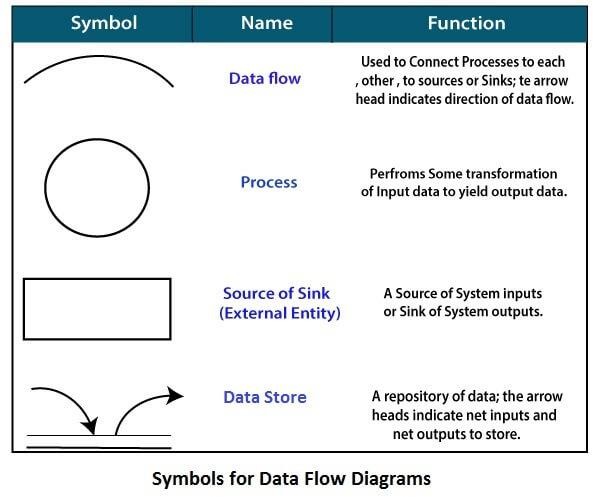
It shows how data enter and leaves the system, what changes the information, and where data is stored.

The objective of a DFD is to show the scope and boundaries of a system as a whole. It may be used as a communication tool between a system analyst and any person who plays a part in the order that acts as a starting point for redesigning a system. The DFD is also called as a data flow graph or bubble chart.

**The following observations about DFDs are essential:**

1. All names should be unique. This makes it easier to refer to elements in the DFD.
2. Remember that DFD is not a flow chart. Arrows is a flow chart that represents the order of events; arrows in DFD represents flowing data. A DFD does not involve any order of events.
3. Suppress logical decisions. If we ever have the urge to draw a diamond-shaped box in a DFD, suppress that urge! A diamond-shaped box is used in flow charts to represents decision points with multiple exists paths of which the only one is taken. This implies an ordering of events, which makes no sense in a DFD.
4. Do not become bogged down with details. Defer error conditions and error handling until the end of the analysis.

Standard symbols for DFDs are derived from the electric circuit diagram analysis and are shown in fig:

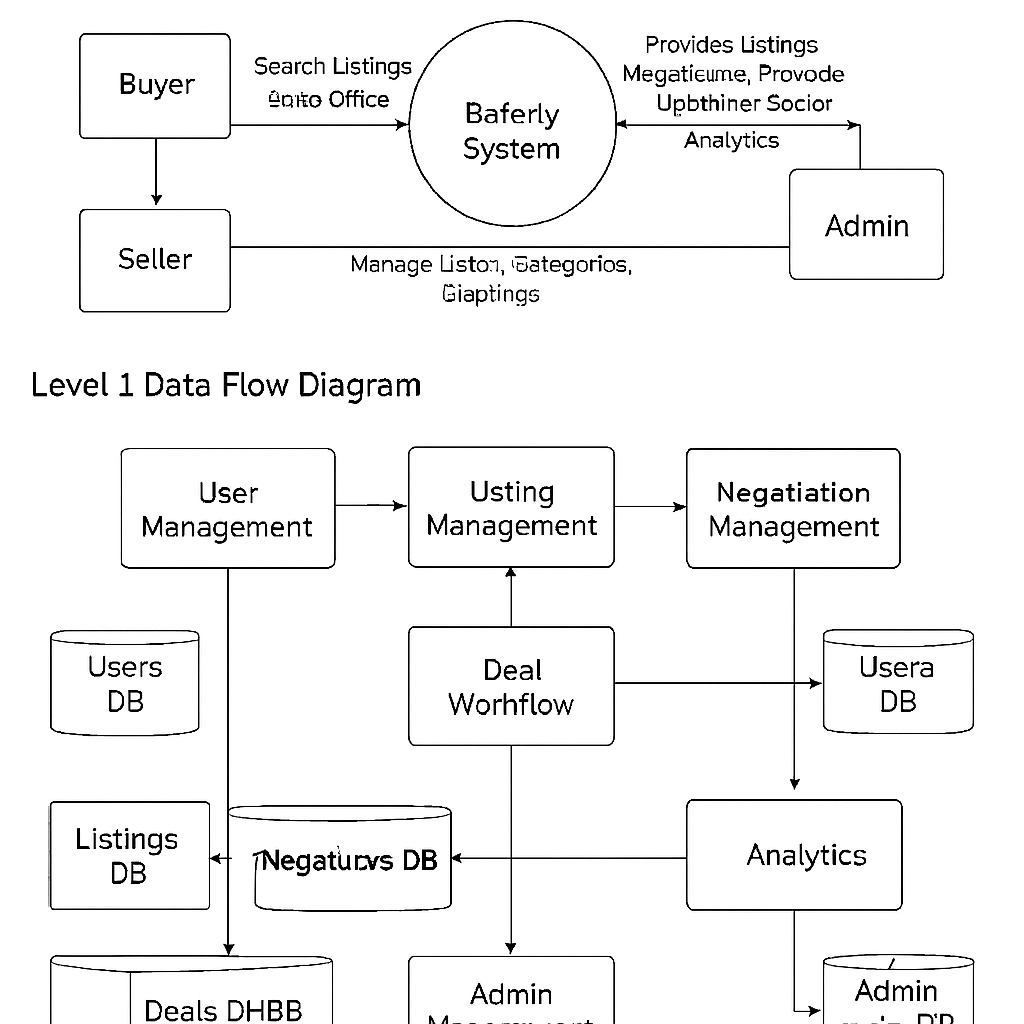


**Circle:** Represents a system function (e.g., “Create Listing”, “Negotiate Offer”).

**Data Flow:** A curved line shows the flow of data into or out of a process or data store.

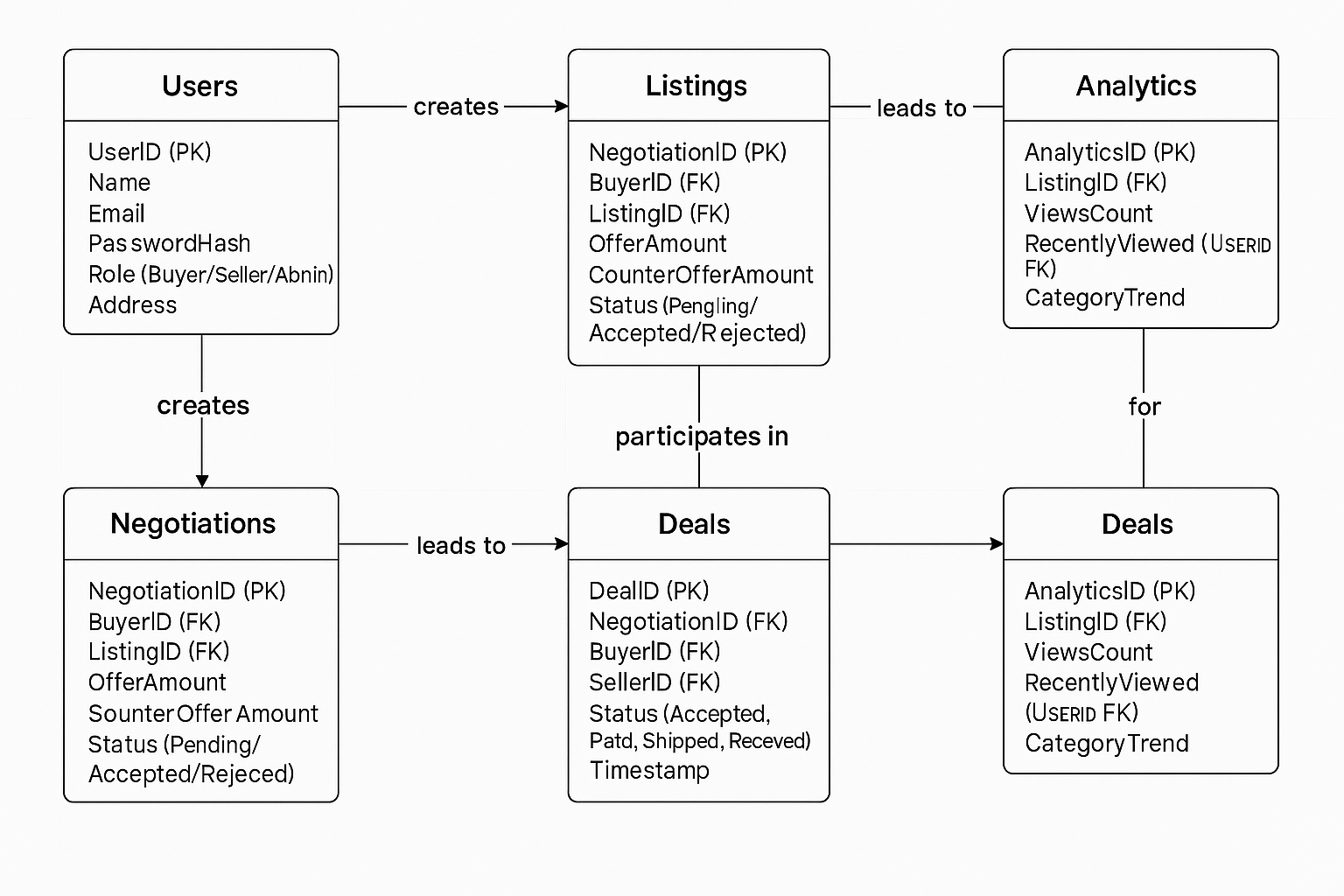
**Data Store:** A set of parallel lines shows a place for the collection of data items. A data store indicates that the data is stored which can be used at a later stage or by the other processes in a different order. The data store can have an element or group of elements.

**Source or Sink**: Source or Sink is an external entity and acts as a source of system inputs or sink of system outputs.



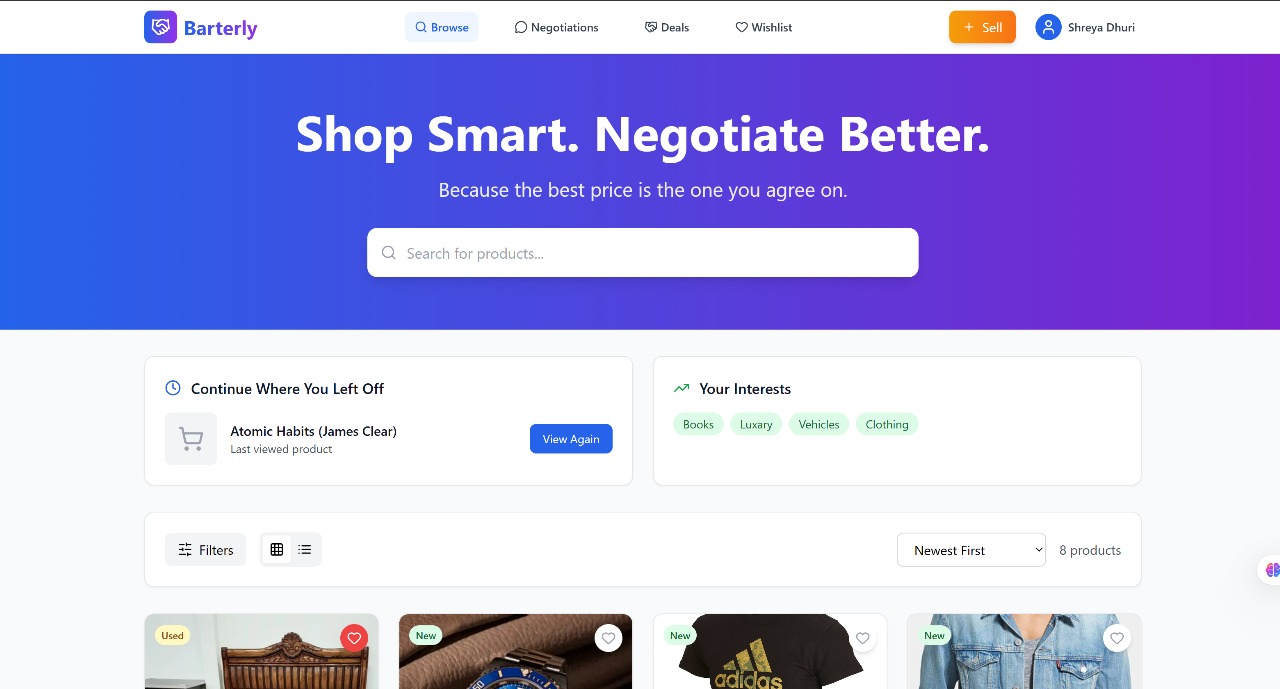
ER diagram

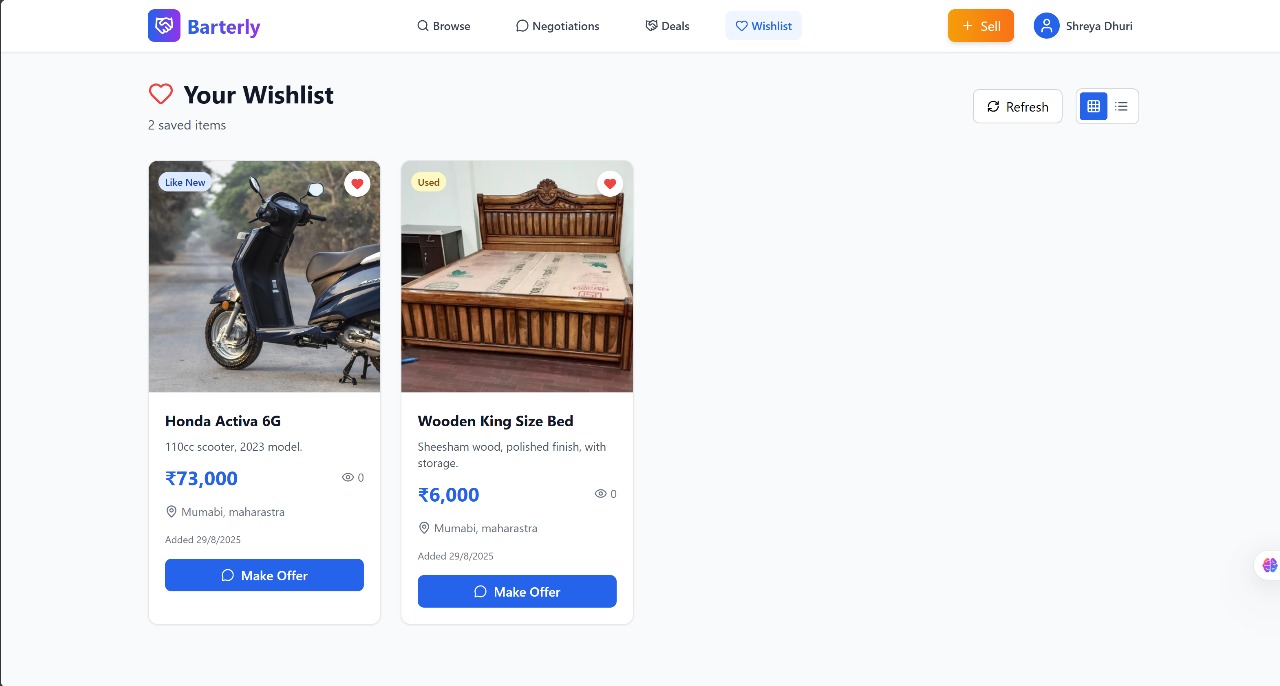
The **ER Diagram** of the **Online News Portal System** shows key relationships between **Users, Admins, News Articles, Categories, Comments, and Sentiment Analysis**, ensuring structured data management and insightful moderation.

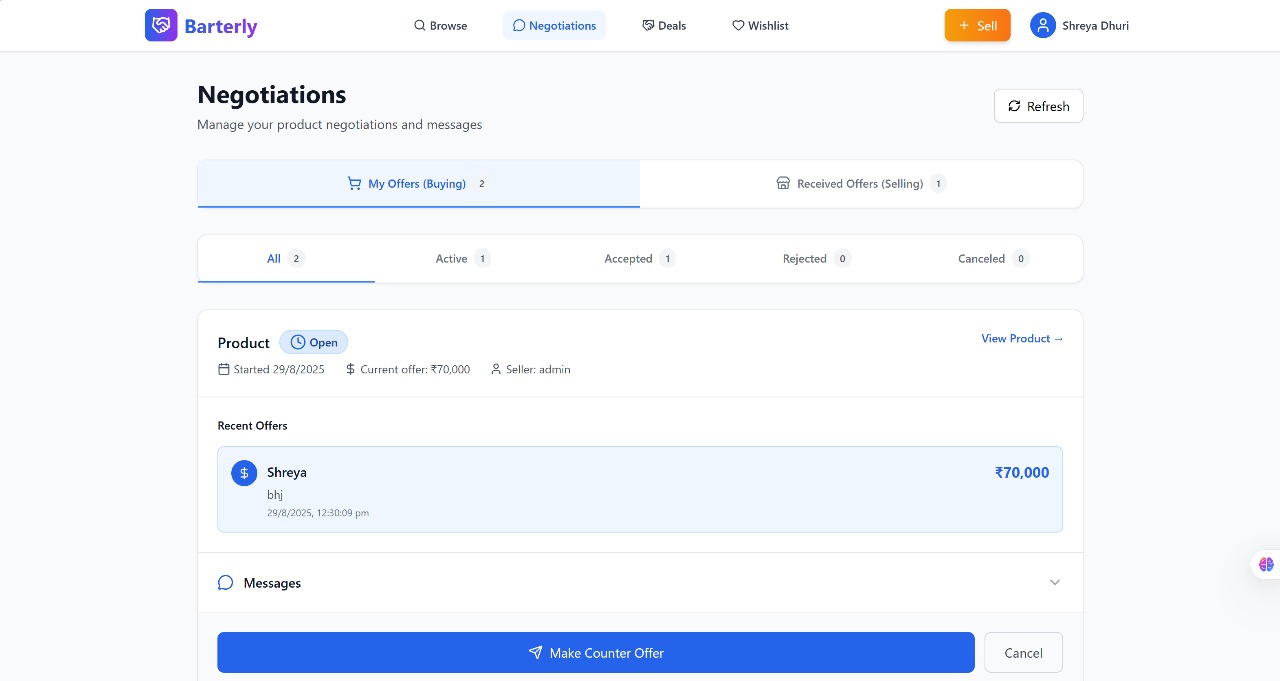


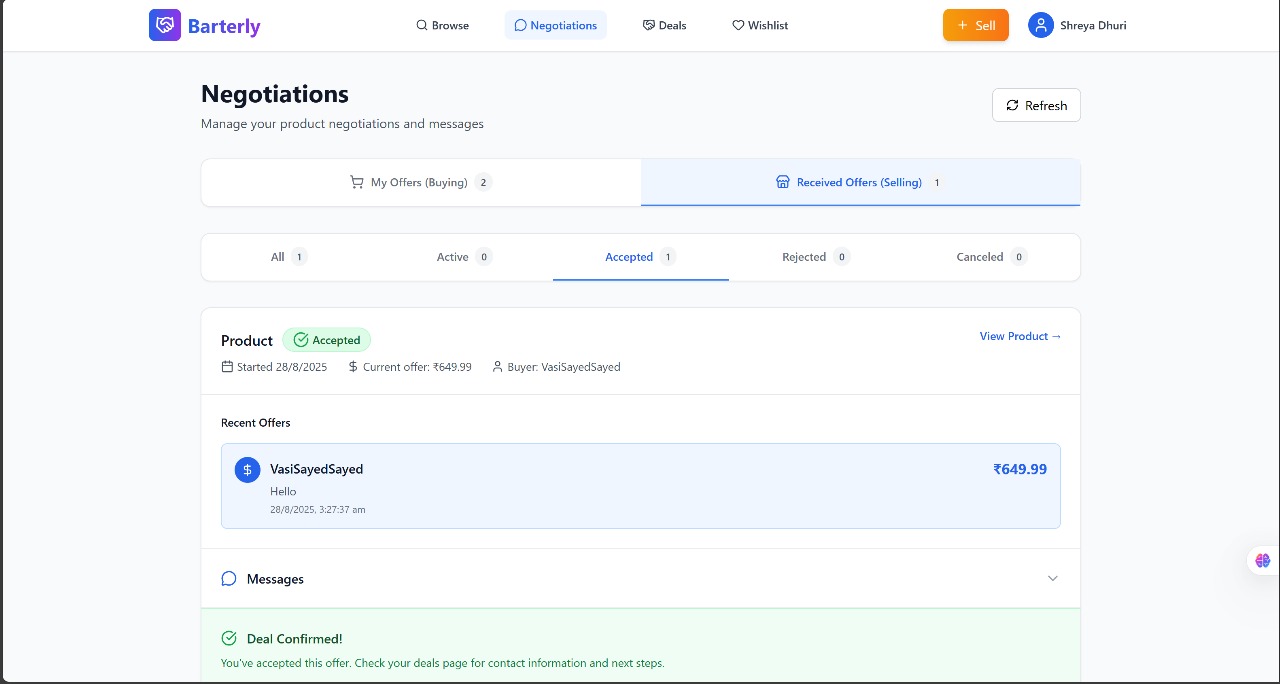
**SCREENSHOTS**

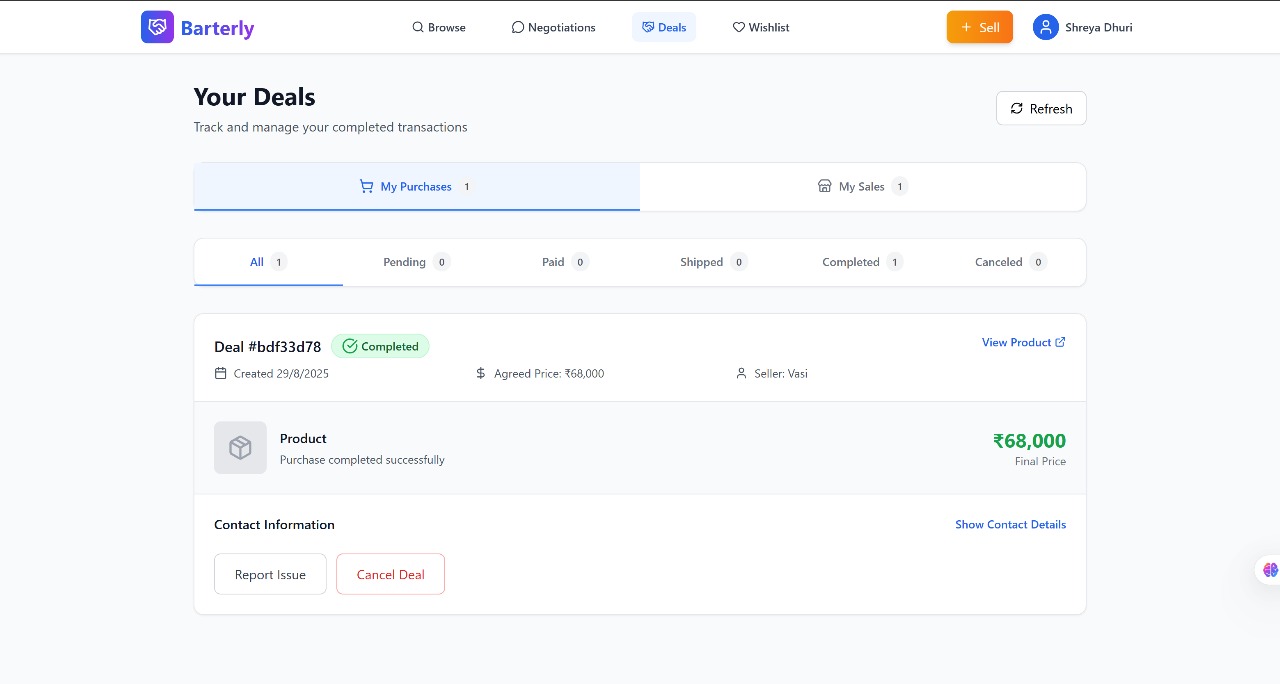
**Home Page**

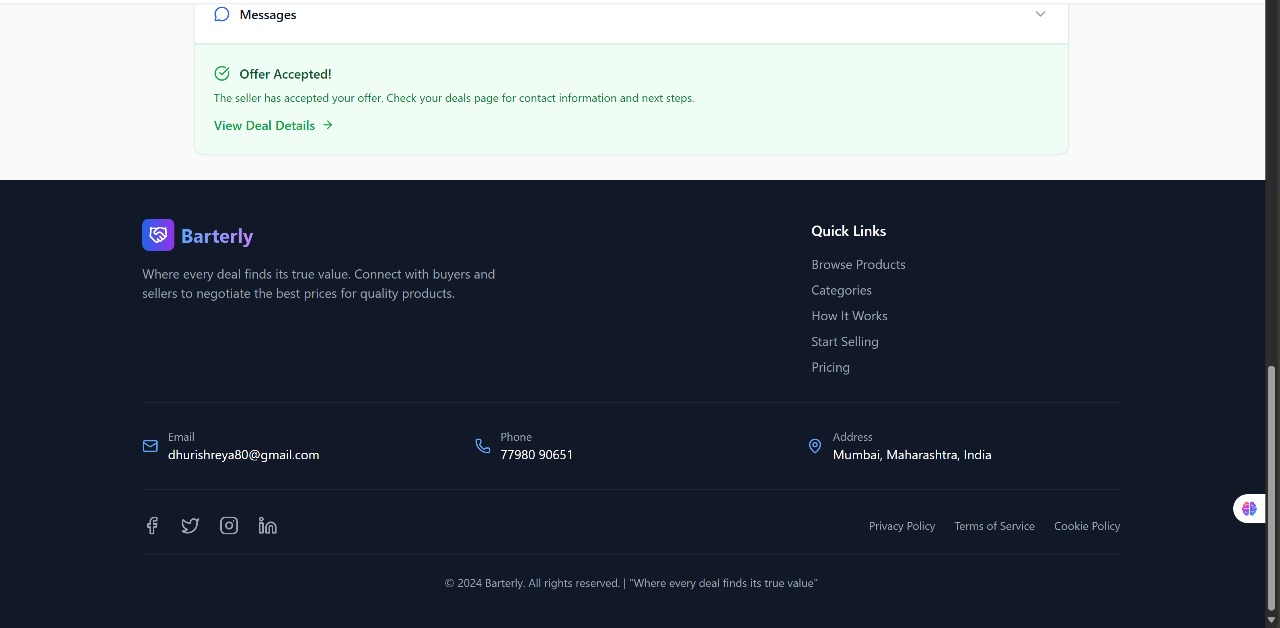
****

****

****

****

****

****

Coding

App.js

import './App.scss';

// react router dom

import {

BrowserRouter,

Routes,

Route

} from "react-router-dom";

// pages

import { Home, MealDetails, Error, Category } from "./pages/index";

// components

import Header from "./components/Header/Header";

import Sidebar from "./components/Sidebar/Sidebar";

function App() {

return (

<BrowserRouter>

<Header />

<Sidebar />

<Routes>

<Route path = "/" element = {<Home />} />

<Route path = "/meal/:id" element = {<MealDetails />} />

<Route path = "/meal/category/:name" element = {<Category />} />

<Route path = "\*" element = {<Error />} />

</Routes>

</BrowserRouter>

);

}

export default App;

Auth.js

import React, { useState } from "react";

import axios from "axios";

const API\_URL = "http://localhost:5000"; // backend server

export default function Auth() {

const [registerData, setRegisterData] = useState({

username: "",

email: "",

password: "",

});

const [loginData, setLoginData] = useState({

email: "",

password: "",

});

const handleRegisterChange = (e) => {

setRegisterData({ ...registerData, [e.target.name]: e.target.value });

};

const handleLoginChange = (e) => {

setLoginData({ ...loginData, [e.target.name]: e.target.value });

};

const register = async () => {

try {

const res = await axios.post(`${API\_URL}/register`, registerData);

alert(res.data.message);

} catch (err) {

alert(err.response?.data?.message || "Registration failed!");

}

};

const login = async () => {

try {

const res = await axios.post(`${API\_URL}/login`, loginData);

alert(res.data.message);

} catch (err) {

alert(err.response?.data?.message || "Login failed!");

}

};

return (

<div className="flex flex-col items-center justify-center min-h-screen bg-gray-100">

{/\* Register Box \*/}

<div className="bg-white p-6 rounded-2xl shadow-md w-80 mb-6">

<h2 className="text-xl font-bold mb-4">Register</h2>

<input

type="text"

name="username"

placeholder="Username"

value={registerData.username}

onChange={handleRegisterChange}

className="w-full mb-3 p-2 border rounded"

/>

<input

type="email"

name="email"

placeholder="Email"

value={registerData.email}

onChange={handleRegisterChange}

className="w-full mb-3 p-2 border rounded"

/>

<input

type="password"

name="password"

placeholder="Password"

value={registerData.password}

onChange={handleRegisterChange}

className="w-full mb-3 p-2 border rounded"

/>

<button

onClick={register}

className="w-full bg-blue-600 text-white py-2 rounded hover:bg-blue-700"

>

Register

</button>

</div>

{/\* Login Box \*/}

<div className="bg-white p-6 rounded-2xl shadow-md w-80">

<h2 className="text-xl font-bold mb-4">Login</h2>

<input

type="email"

name="email"

placeholder="Email"

value={loginData.email}

onChange={handleLoginChange}

className="w-full mb-3 p-2 border rounded"

/>

<input

type="password"

name="password"

placeholder="Password"

value={loginData.password}

onChange={handleLoginChange}

className="w-full mb-3 p-2 border rounded"

/>

<button

onClick={login}

className="w-full bg-green-600 text-white py-2 rounded hover:bg-green-700"

>

Login

</button>

</div>

</div>

);

}

Index.js

import React from 'react';

import ReactDOM from 'react-dom/client';

import App from './App';

import { SidebarProvider } from './context/sidebarContext';

import { MealProvider } from './context/mealContext';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

<SidebarProvider>

<MealProvider>

<App />

</MealProvider>

</SidebarProvider>

);

Mealaction.js

import axios from "../api/axios";

import {

FETCH\_CATEGORY\_BEGIN,

FETCH\_CATEGORY\_ERROR,

FETCH\_CATEGORY\_MEALS\_BEGIN,

FETCH\_CATEGORY\_MEALS\_ERROR,

FETCH\_CATEGORY\_MEALS\_SUCCESS,

FETCH\_CATEGORY\_SUCCESS,

FETCH\_MEALS\_BEGIN,

FETCH\_MEALS\_ERROR,

FETCH\_MEALS\_SUCCESS,

FETCH\_SINGLE\_MEAL\_BEGIN,

FETCH\_SINGLE\_MEAL\_ERROR,

FETCH\_SINGLE\_MEAL\_SUCCESS

} from "./actions";

import { CATEGORIES\_URL, MEAL\_CATEGORIES\_URL, MEAL\_SINGLE\_URL, SEARCH\_URL } from "../utils/constants";

export const startFetchCategories = async(dispatch) => {

try{

dispatch({ type: FETCH\_CATEGORY\_BEGIN});

const response = await axios.get(`${CATEGORIES\_URL}`);

dispatch({type: FETCH\_CATEGORY\_SUCCESS, payload: response.data.categories});

} catch(error){

dispatch({ type: FETCH\_CATEGORY\_ERROR, payload: error.message});

}

}

export const startFetchSingleMeal = async(dispatch, id) => {

try{

dispatch({ type: FETCH\_SINGLE\_MEAL\_BEGIN});

const response = await axios.get(`${MEAL\_SINGLE\_URL}${id}`);

dispatch({type: FETCH\_SINGLE\_MEAL\_SUCCESS, payload: response.data.meals});

} catch(error){

dispatch({ type: FETCH\_SINGLE\_MEAL\_ERROR, payload: error.message});

}

}

export const startFetchMealByCategory = async(dispatch, category) => {

try{

dispatch({type: FETCH\_CATEGORY\_MEALS\_BEGIN});

const response = await axios.get(`${MEAL\_CATEGORIES\_URL}${category}`);

dispatch({type: FETCH\_CATEGORY\_MEALS\_SUCCESS, payload: response.data.meals})

} catch(error){

dispatch({ type: FETCH\_CATEGORY\_MEALS\_ERROR, payload: error.message});

}

}

export const startFetchMealsBySearch = async(dispatch, searchTerm) => {

try{

dispatch({ type: FETCH\_MEALS\_BEGIN});

const response = await axios.get(`${SEARCH\_URL}${searchTerm}`);

dispatch({ type: FETCH\_MEALS\_SUCCESS, payload: response.data.meals});

} catch(error){

dispatch({type: FETCH\_MEALS\_ERROR, payload: error.message});

}

}

Categories.js

import React from 'react';

import { Link } from 'react-router-dom';

import "./Category.scss";

const CategoryList = ({categories}) => {

return (

<div className='section-wrapper bg-whitesmoke'>

<div className='container'>

<div className='sc-title'>categories</div>

<section className='sc-category grid'>

{

categories.map(category => {

const { idCategory: id, strCategory: title, strCategoryThumb: thumbnail} = category;

return (

<Link to = {`/meal/category/${title}`} className = "category-itm align-center justify-center" key = {id}>

<div className='category-itm-img h-100 w-100 flex align-center justify-center'>

<img src = {thumbnail} alt = {title} />

<div className='category-itm-title bg-orange'>

<h3 className='text-white fs-11 fw-6 ls-1 text-uppercase'>{title}</h3>

</div>

</div>

</Link>

)

})

}

</section>

</div>

</div>

)

}

export default CategoryList

MealContext.js

import React, {createContext, useContext, useEffect, useReducer} from "react";

import { mealReducer } from "../reducers/mealReducer";

import { startFetchCategories } from "../actions/mealsActions";

const initialState = {

categories: [],

categoryLoading: false,

categoryError: false,

categoryMeals: [],

categoryMealsLoading: false,

categoryMealsError: false,

meals: [],

mealsLoading: false,

mealsError: false,

meal: [],

mealLoading: false,

mealError: false

}

const MealContext = createContext({});

export const MealProvider = ({children}) => {

const [state, dispatch] = useReducer(mealReducer, initialState);

useEffect(() => {

startFetchCategories(dispatch);

}, []);

return (

<MealContext.Provider value = {{

...state,

dispatch,

startFetchCategories

}}>

{children}

</MealContext.Provider>

)

}

export const useMealContext = () => {

return useContext(MealContext);

}

Testing

**1. Testing**

Testing is a critical phase in ensuring that Baterly functions as intended and provides a smooth user experience. Various levels of testing were performed:

**1.1 Unit Testing**

Unit testing focused on individual components or modules:

* **Negotiation Module:** Verified that buyers and sellers can interact, propose offers, counteroffers, and finalize deals accurately.
* **Workflow Module:** Ensured that task sequences like listing a product, approving it, and updating status work correctly without errors.

**1.2 Module Testing**

Module testing examined the interaction between multiple components:

* **Integration of Listings and Negotiations:** Confirmed that when a product is listed, buyers can negotiate seamlessly and the system records every transaction step correctly.
* Checked data consistency, error handling, and smooth communication between modules.

**1.3 System Testing**

System testing focused on end-to-end functionality:

* Verified the complete workflow from product listing → buyer search → negotiation → purchase completion.
* Ensured system reliability under various scenarios, including simultaneous users and high transaction volumes.

**1.4 Acceptance Testing**

Acceptance testing involved real users:

* Buyers and sellers tested the platform to verify usability, accessibility, and efficiency.
* Feedback was collected to improve user interface, navigation, and workflow intuitiveness

# 6. Future scope

Baterly has a roadmap for advanced features aimed at enhancing user experience and transaction security:

1. **AI-driven Price Recommendations:**
   * Machine learning algorithms can suggest optimal prices based on market trends, product demand, and historical sales data.
2. **Escrow-based Payment Integration:**
   * Secure financial transactions using an escrow system to ensure both parties meet their obligations before payment release.
3. **AR/VR Product Previews:**
   * Augmented and virtual reality features will allow buyers to visualize products in their environment before purchase.
4. **Dedicated Mobile Applications:**
   * Native apps for iOS and Android for faster access, push notifications, and offline functionalities.
5. **Social Media Listing Integration:**
   * Ability to share and sell products directly on platforms like Instagram and Facebook, expanding reach and engagement.

**Conclusion**

Baterly demonstrates how modern C2C marketplaces can be improved through structured negotiation processes and actionable analytics. The platform combines user-friendly workflows, transparency, and efficiency, creating a seamless experience for both buyers and sellers.

With planned AI and AR/VR enhancements, Baterly aims to redefine online trading in the C2C segment by making it more intelligent, interactive, and secure

# REFERENCES

## ****References****

1. **Tailwind CSS Documentation** – [https://tailwindcss.com/docs](https://tailwindcss.com/docs?utm_source=chatgpt.com)
2. **React Documentation** – https://react.dev
3. **React DOM Documentation** – https://react.dev/reference/react-dom
4. **Node.js Documentation** – https://nodejs.org/en/docs
5. **MySQL Documentation** – https://dev.mysql.com/doc

## ****API Reference****

* **TheMealDB API** – https://www.themealdb.com/api.php