COLLEGE CODE:6107

COLLEGE NAME: GCE, BARGUR

DEPARTMENT: CSE

STUDENT NM-ID: 5a22913b2d187cc863d4b453d67408fd

ROLL NO:2303610710422047

DATE:24.09.2025

Completed the project named as Phase 3

TECHNOLOGY PROJECT NAME :

IBM-NJ-E-COMMERCE CART SYSTEM

SUBMITTED BY,

NAME: SHERIN KERENHAP.S
MOBILE NO:9487494054

1.Project Setup:

Technology Stack

- Choose React/Angular (frontend) and Node.js/Django (backend) to build a scalable and modular system.

Database Selection

- Use MongoDB/PostgreSQL for storing users, carts, and products; SQLite can be used for MVP.

Development Tools

- VS Code for coding, Postman for API testing, Docker for environment consistency.

Hosting

Deploy frontend on Vercel/Netlify and backend on Heroku/AWS for global accessibility.

1. Core Objective

- To design and implement a basic, reliable, and scalable cart system that ensures smooth shopping experience, accurate order management, and secure checkout.

2. Key Features in MVP

- Focus only on the essential features needed to test the solution:
- User Management
- Sign up / Login
- Session management
- Cart Operations
- Add / Remove items
- Update product quantity
- Auto calculation of price (with tax & discount logic)

3. Modules Setup

- User Module – registration, login, profile
- Product Module – product listing, details
- Cart Module – add/remove/update items, price calculation
- Order Module – order placement, order tracking

4.Tech Stack (suggested for MVP)

- Frontend: React / Angular (User Interface)
- Backend: Node.js / Spring Boot / Django (API & Business Logic)
- Database: MySQL / PostgreSQL (Product, Cart, Orders)
- Authentication: JWT / OAuth2
- Payment Integration: Stripe / Razorpay sandbox for MVP
- Deployment: Docker + Cloud (AWS / Azure / GCP)

2.Core Features Implementation:

Add/Update/Remove Items

- Customers can manage cart items easily with dynamic updates.

Cart Summary

- Display subtotal, taxes, shipping, discounts, and total in real time.

Persistent Carts

- Logged-in users retain carts across sessions; guests use cookies/localStorage.

Stock & Price Validation

- Backend ensures accuracy to prevent mismatches at checkout.

Checkout & Authentication

- Secure login and checkout process with JWT/session handling.

Implementation

- REST APIs for cart CRUD operations, optimistic UI updates for instant feedback, and error handling for stock issues.

Coupons & Discounts

- Apply promo codes.
- Support percentage-based or fixed-amount discounts.
- Handle restrictions (expiry date, product category, first-time users)

Cart Validation

- Verify stock availability before checkout.
- Check if prices/discounts are still valid (in case of product updates).

- Prevent checkout if invalid items remain.

Security

- Prevent price manipulation (server-side validation of prices).
- CSRF protection on cart updates.
- Sanitize inputs for coupon codes, quantities.

3.Data storage(local state/Database):

Local State (Frontend)

- Use React Context API/Redux for fast cart operations within the browser.
- Guest users (not logged in)
- Faster response time without waiting for server round-trips
- Offline or intermittent network support

LocalStorage/SessionStorage

- Helps guests keep cart items temporarily even after refreshing the page.

Database (Backend)

- Users: user_id, name, email.
- Products: product_id, name, price, stock, image.
- Cart: cart_id, user_id, created_at.
- CartItems: cart_item_id, cart_id, product_id, quantity.

Example:

```
CREATE TABLE carts (
 id UUID PRIMARY KEY,
 user_id UUID,
 status VARCHAR, -- active, completed, abandoned
 created_at TIMESTAMP,
 updated_at TIMESTAMP
```

```
);

CREATE TABLE cart_items (
    id UUID PRIMARY KEY,
    cart_id UUID REFERENCES carts(id),
    product_id UUID,
    quantity INTEGER,
    price NUMERIC,
    variant_id UUID
);
```

Lightweight Persistent Database (Recommended for MVP)

- SQLite – File-based, easy to set up, zero-config.
- MongoDB Atlas Free Tier – Great for JSON-like cart data.
- PostgreSQL/MySQL (Docker or cloud free tier) – If relational schema is needed.

Hybrid MVP Approach

- Use local state + LocalStorage for frontend.
- Mock a backend with JSON Server (a quick REST API over JSON file).
- Later swap JSON file with a real DB (SQLite/Mongo/Postgres).

4.Testing Core Features:

1. Cart Operations

- Add single product to cart
- Add multiple products (different SKUs, quantities)
- Update product quantity (increase/decrease)
- Remove item(s) from cart
- Empty cart entirely
- Cart persistence across sessions (logged in vs guest)

2. Product Validations

- Adding out-of-stock items → should block or show message
- Adding items beyond available stock → correct validation
- Adding items with different variations (size, color)
- Handling discontinued/removed products

3. Price & Discounts

- Correct product price displayed in cart
- Subtotal calculation accuracy
- Tax calculation (location-based if applicable)
- Shipping charges calculation
- Discount codes / vouchers / promo application
- Loyalty points or gift card redemption

4. Checkout Flow

- Proceed to checkout from cart
- Guest checkout vs logged-in checkout
- Address selection / addition / validation
- Payment gateway redirection and return flow
- Order summary correctness before payment

5. User Experience

- Cart updates reflected instantly (AJAX / reload)
- Proper error handling (e.g., “item out of stock”)
- Responsive design on mobile/tablet
- Accessibility (screen readers, keyboard navigation)

6. Persistence & Security

- Cart saved when user logs out and logs in again
- Session timeout handling
- Secure handling of sensitive info (e.g., price manipulation via browser dev tools)
- Prevention of duplicate orders

7. Integration Points

- Inventory system sync (stock reduction after checkout)
- Payment gateway success/failure handling
- Email / SMS notifications for cart abandonment or successful checkout

5.version control(Github)

Repository Setup

- Store frontend and backend code in GitHub for collaboration.

Branching Strategy

- Use main for stable releases, dev for testing, and feature branches for new modules.

Collaboration

- Enable pull requests and code reviews for quality assurance.

CI/CD Pipelines

- Automate builds, testing, and deployment using GitHub Actions.

Issue Tracking

- Use GitHub Issues/Projects to manage tasks, bugs, and enhancements.

1. Version Control in Code (Development Perspective)

This refers to using tools like **Git** to manage changes to the **source code** of the cart system.

Key Practices:

- **Branching:** Feature branches for new cart features (e.g. discount system, multi-currency support).
- **Commits with context:** Clear commit messages (e.g. fix: prevent negative quantity in cart).
- **Pull Requests & Code Reviews:** Ensure quality and collaborative review.
- **Tags/Releases:** Mark versions like v1.0.0, v2.0.0-beta to track production-ready states.
- **Rollback Capability:** Easily revert to a previous working version in case of bu

2. Version Control in Cart Data (Application/Data Perspective)

Here, version control is about **managing the state or changes** of a user's shopping cart — particularly when:

- Multiple sessions/devices are involved
- Cart merges are needed (e.g. guest cart → logged-in cart)
- Price or inventory changes over time
- Integration with backend APIs and third-party services

Strategies:

A. Cart Versioning (State Tracking)

Each modification to a cart (add, remove, update item) increments a **version number** or creates a **new snapshot**.

- Ensures **consistency** during updates.
- Avoids race conditions when multiple clients update the cart simultaneously.
- Enables **conflict resolution** (e.g. price changed while the user was inactive).

B. Event Sourcing (Advanced)

Every change to the cart is stored as an **event**, like:

- Item Added
- Quantity Updated
- Cart Cleared

You can rebuild the cart's state at any time by replaying the events. This offers:

- Full auditability
- Easier debugging
- Better handling of concurrent changes

C. Optimistic Locking

The system assumes no conflicts, but if an update arrives with a stale version, it gets rejected.

E.g.,

1. User fetches cart (version 3)
2. User changes quantity → sends update with version 3
3. Server has version 4 (due to a discount applied elsewhere) → update rejected

D. Snapshotting & Caching

For performance:

- Periodically save **snapshots** of cart state
- Use **caching layers** (e.g. Redis) with version keys to avoid stale writes

E. Versioning Cart APIs

If you're evolving your cart API:

- Keep backward compatibility via API versioning:
    - /api/v1/cart
    - /api/v2/cart with new features (e.g. promotions,bundles)