COLLEGE CODE : 8203

COLLEGE NAME : A.V.C College Of Engineering

DEPARTMENT : B.Tech-Information Technology

STUDENT NM_ID : 6758371BD883BE19DBB421292A03628A

ROLL NO : 820323205079

DATE : 15-09-2025

Completed the project named as Phase II technology

project name : E-Commerce Cart System

SUBMITTED BY,

NAME : Priyadharshan A

MOBILE NO:8220863295

Tech Stack Selection :

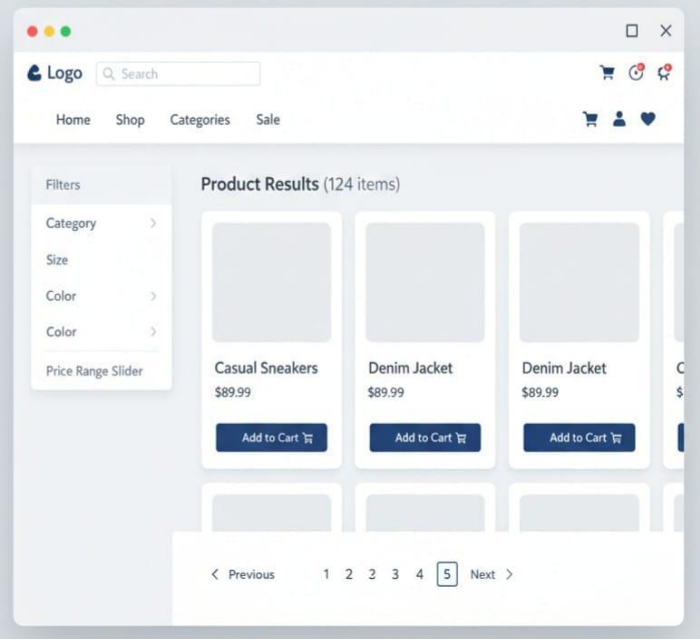
The following technologies have been selected for the development of the E-Commerce Cart System API:

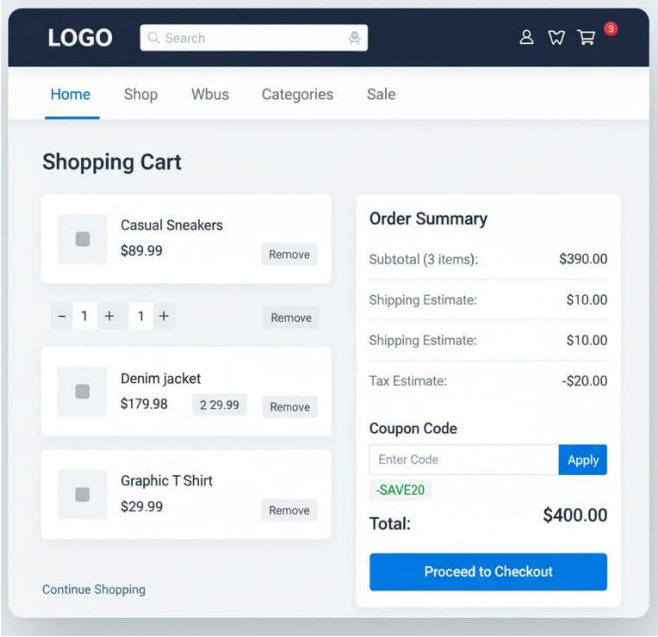
- **Backend Framework**: **Node.js with Express.js**
    - **Reasoning**: Node.js is an open-source, cross-platform JavaScript runtime environment that allows for fast, scalable, and efficient server-side development. Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications, making it ideal for building RESTful APIs. Its non-blocking I/O model is well-suited for handling concurrent requests, which is crucial for an e-commerce application.
- **Database**: **MongoDB (NoSQL Document Database)**
    - **Reasoning**: MongoDB is a highly scalable, flexible NoSQL database that stores data in JSON-like BSON documents. This schema-less approach is particularly advantageous for e-commerce carts, as item details can vary, and the structure can evolve without complex migrations. It offers high performance and horizontal scalability, essential for handling potentially large volumes of cart data and user traffic.
- **Authentication/Authorization**: **JSON Web Tokens (JWT) or Express-Sessions**
    - **Reasoning**:
        - **JWT**: A compact, URL-safe means of representing claims to be transferred between two parties. It's excellent for stateless authentication, where tokens are sent with each request, reducing server load for session management. Ideal for microservices or mobile apps.
        - **Express-Sessions**: A traditional session-based approach where session data is stored on the server (e.g., in MongoDB) and a session ID (cookie) is sent to the client. This is simpler to implement for server-rendered applications or if full server-side session control is desired.
        - *Decision for this project*: We will initially lean towards **JWT** for its stateless nature and suitability for API-first development, providing more flexibility for frontend integration (web, mobile).
- **Other Libraries/Tools**:
    - mongoose: An elegant MongoDB object modeling tool for Node.js, simplifying data interaction and schema definition.
    - bcryptjs: For hashing passwords (if user authentication is extended beyond just cart association).
    - dotenv: For managing environment variables securely.

UI Structure / API Schema Design :

Overall Structure (API-Centric)

The project will be structured as a RESTful API backend. The UI (Frontend) is considered a separate client application that consumes these APIs. This promotes separation of concerns and allows for multiple client applications (web, mobile, etc.) to use the same backend.





API Schema Design (Key Data Models)

1. Product Schema (Simplified)

This schema defines the structure for individual products available for purchase.

- **_id**: ObjectId (MongoDB's default unique ID)
- **name**: String, required: true (e.g., "Laptop Pro")
- **description**: String (e.g., "High-performance laptop...")
- **price**: Number, required: true (e.g., 1200.00)
- **imageUrl**: String (e.g., "http://example.com/laptop.jpg")
- **stock**: Number, default: 0 (for optional inventory check)

2. User Schema (Simplified for Cart Association)

This schema defines basic user information, primarily for associating carts with authenticated users.

- **_id**: ObjectId
- **email**: String, required: true, unique: true
- **password**: String, required: true (hashed)
- **createdAt**: Date, default: Date.now

3. Cart Schema

This is the core schema, representing a user's shopping cart.

- **_id**: ObjectId
- **userId**: ObjectId, ref: 'User', required: true, unique: true (Each user has one cart)
- **items**: Array of Objects
    - **productId**: ObjectId, ref: 'Product', required: true
    - **quantity**: Number, required: true, min: 1
    - **price**: Number, required: true (Price at the time of adding to cart)
- **createdAt**: Date, default: Date.now
- **updatedAt**: Date, default: Date.now

4. Order Schema (For Checkout)

This schema defines the structure for a completed order.

- **_id**: ObjectId
- **userId**: ObjectId, ref: 'User', required: true
- **items**: Array of Objects (Snapshot of cart items at time of purchase)
    - **productId**: ObjectId, ref: 'Product'
    - **name**: String
    - **quantity**: Number
    - **price**: Number
- **totalAmount**: Number, required: true
- **status**: String, enum: ['pending', 'completed', 'shipped', 'cancelled'], default: 'pending'
- **orderDate**: Date, default: Date.now

Data Handling Approach :

1. Data Storage (MongoDB)

- All persistent data (Products, Users, Carts, Orders) will be stored in a MongoDB database.
- mongoose will be used as an ODM (Object Data Modeling) library to define schemas, validate data, and interact with MongoDB from Node.js.

2. Data Flow

- **Client to Server**: Frontend applications will send HTTP requests (GET, POST, PUT, DELETE) to the Express API endpoints. Request bodies will typically be JSON.
- **Server Processing**: Express routes will handle incoming requests, validate input, interact with MongoDB via mongoose models, apply business logic (e.g., updating cart quantities, calculating totals), and manage user sessions/tokens.
- **Server to Database**: mongoose queries will be used to create, read, update, and delete documents in MongoDB collections.
- **Server to Client**: The API will respond with JSON data (e.g., the updated cart, a list of products, an order confirmation) and appropriate HTTP status codes (200 OK, 201 Created, 400 Bad Request, 401 Unauthorized, 404 Not Found, 500 Internal Server Error).

3. Error Handling

- Global error handling middleware will be implemented in Express to catch synchronous and asynchronous errors.
- Specific error classes will be used for different error types (e.g., NotFoundError, BadRequestError) to provide consistent error responses to the client.
- Validation errors (e.g., missing required fields, invalid quantities) will be caught and returned as 400 Bad Request.

4. Security Considerations

- **Authentication**: JWTs will be used. Upon successful login (or potentially automatically for guest carts tied to a temporary JWT), a token will be issued. This token will be sent with every subsequent request in the Authorization header.
- **Authorization**: Middleware will check the validity of the JWT and ensure the user associated with the token has permission to perform the requested action (e.g., only access their own cart).
- **Input Validation**: All incoming API request data will be rigorously validated to prevent injection attacks and ensure data integrity.
- **Sensitive Data**: Passwords (if implemented) will be hashed using bcryptjs before being stored in the database.

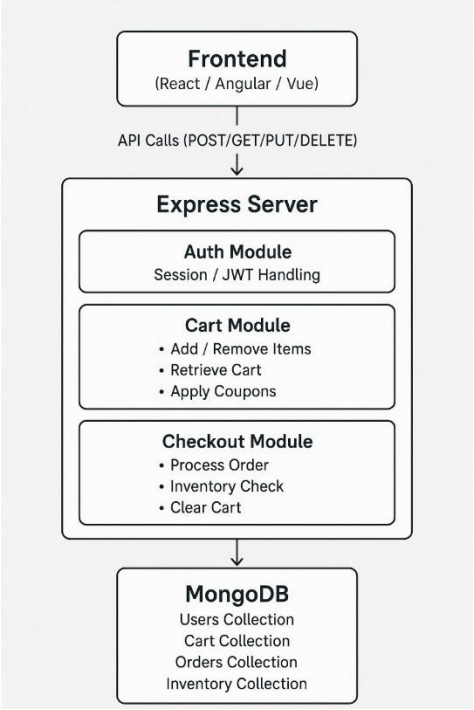
- **Environment Variables**: Sensitive configuration data (e.g., database connection strings, JWT secrets) will be stored in environment variables and accessed using dotenv.

Component / Module Diagram

This diagram illustrates the main components of the backend application and how they interact.

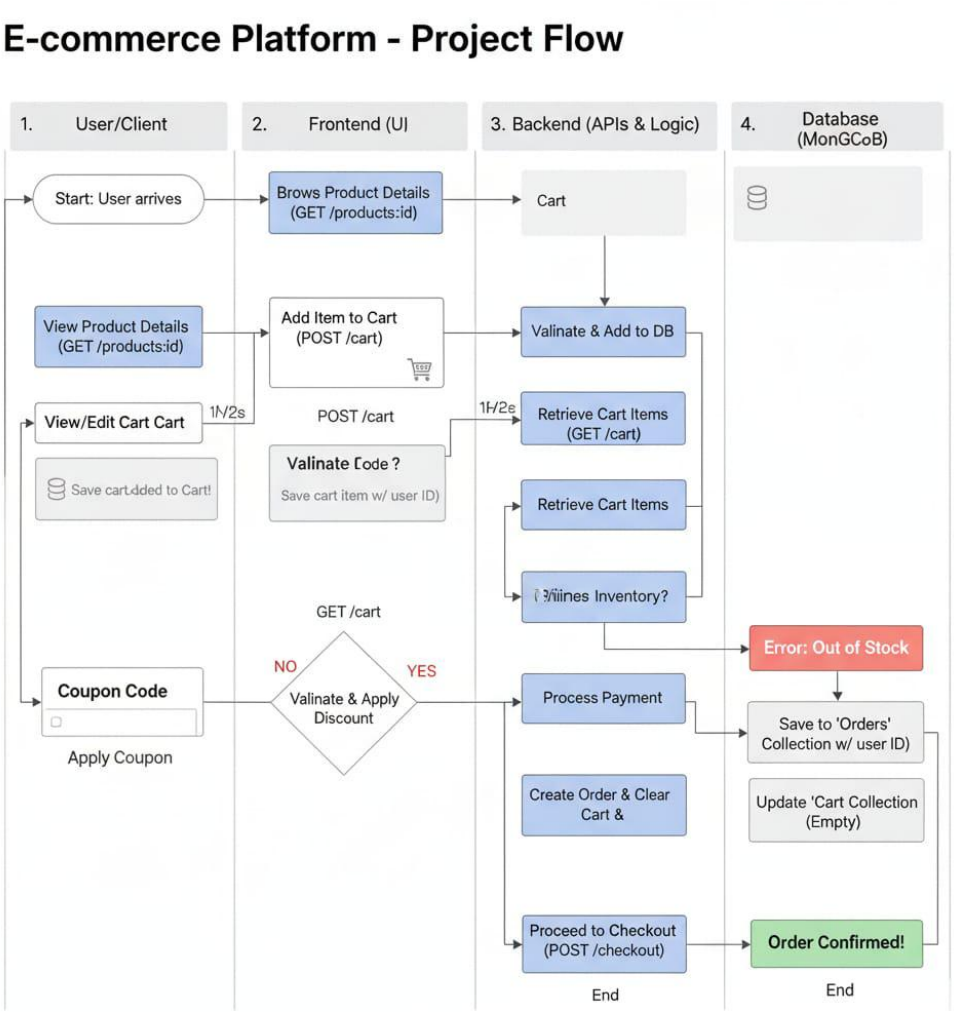
Explanation:

- **Client/Frontend App**: Any application (web browser, mobile app) that consumes the API.
- **API Gateway/Load Balancer (Optional but good practice)**: Manages incoming requests, routes them to the correct service, and can provide caching, security, and load distribution.
- **Express.js Application**: The core backend.
    - **Middleware**: Functions that run before the main route handlers, typically for authentication, authorization, logging, etc.
    - **Routes**: Define the API endpoints (e.g., /cart, /checkout).
    - **Controllers**: Functions associated with routes that parse requests, apply business logic, and orchestrate responses.
    - **Services/Models**: Abstractions for interacting with the database. Mongoose models would reside here.
- **MongoDB Database**: Stores all application data.
- **Error Handling Middleware**: Catches errors from any part of the application and sends a standardized error response to the client.



Flow Diagram :

This diagram illustrates the typical flow for adding an item to the cart, retrieving the cart, and the checkout process.



Explanation:

The flow diagrams break down the interaction between the frontend, backend (Express), and database (MongoDB) for the core functionalities. Each step highlights authentication, data manipulation, and expected responses.