Objective: Q-Commerce (Restaurant Based Website)

1. Technical Plan:

Technology Stack:

- **Frontend:** Next.js for server-side rendering and React.js for the frontend framework.
- **Backend:** Node.js with Express for API creation.
- **Database:** Sanity for data storage.
- **CMS:** Sanity for content management.
- **Authentication:** AithJs Authentication for user login and registration.
- **Payments:** Stripe for processing payments.
- **Shipment:** Ship Engine / Fleet for delivery
- **Hosting:** Vercel for deployment.
- **Version Control:** GitHub for version control.

Infrastructure:

- **APIs:** RESTful APIs for data exchange.
- **CDN:** Cloudflare for content delivery network (CDN) to ensure fast load times.
- **Caching:** Redis for caching frequently accessed data.
- **Monitoring:** New Relic for performance monitoring.
- **CI/CD:** GitHub Actions for continuous integration and deployment.

2. Work Flows:

User Journey:

1. **Landing Page:** Users visit the landing page and see featured restaurants.
2. **Search/Filter:** Users search for restaurants based on location, cuisine, or ratings.
3. **Menu Browsing:** Users browse the menu of the selected restaurant.
4. **Order Placement:** Users select items, add them to the cart, and place an order.
5. **Payment:** Users proceed to the payment gateway to complete the transaction.
6. **Order Tracking:** Users can track their order in real-time.
7. **Delivery:** The order is delivered to the user's specified address.
8. **Feedback:** Users can provide feedback and ratings for the restaurant.

Admin Workflow:

1. **Login:** Admin logs in to the dashboard.
2. **Manage Restaurants:** Admin can add, edit, or remove restaurants.
3. **Manage Orders:** Admin can view and manage all orders.
4. **Manage Users:** Admin can manage user accounts and permissions.
5. **Analytics:** Admin can view analytics and reports on sales and user engagement.

3. API Requirements

Authentication:

- **POST /api/auth/register:** Register a new user.
- **POST /api/auth/login:** Login an existing user.
- **POST /api/auth/logout:** Logout the current user.

Users:

- **GET /api/users:** Get all users.
- **GET /api/users/{id}:** Get user by ID.
- **PUT /api/users/{id}:** Update user details.
- **DELETE /api/users/{id}:** Delete user.

Restaurants:

- **GET /api/restaurants:** Get all restaurants.
- **GET /api/restaurants/{id}:** Get restaurant by ID.
- **POST /api/restaurants:** Add a new restaurant.
- **PUT /api/restaurants/{id}:** Update restaurant details.
- **DELETE /api/restaurants/{id}:** Delete restaurant.

Orders:

- **POST /api/orders:** Place a new order.
- **GET /api/orders:** Get all orders.
- **GET /api/orders/{id}:** Get order by ID.
- **PUT /api/orders/{id}:** Update order status.
- **DELETE /api/orders/{id}:** Delete order.

Payments:

- **POST /api/payments:** Process a payment.

Products:

- **GET /api/products:** Get all products.
- **GET /api/products/{id}:** Get product by ID.
- **POST /api/products:** Add a new product.
- **PUT /api/products/{id}:** Update product details.
- **DELETE /api/products/{id}:** Delete product.

4. Sanity Schema

Create a schema for managing restaurants, products, and orders.

User Schema:

```
export default {
  name: 'user',
  title: 'User',
  type: 'document',
  fields: [
    { name: 'name', title: 'Name', type: 'string' },
    { name: 'phone', title: 'Phone', type: 'number' },
    { name: 'email', title: 'Email', type: 'email' },
    { name: 'location', title: 'Location', type: 'geopoint' },
    { name: 'order', title: 'order', type: 'array', of: [{ type: 'reference', to: { type: 'product' } }] },
  ],
};
```

Product Schema:

```
export default {
  name: 'product',
  title: 'Product',
  type: 'document',
  fields: [
    { name: 'name', title: 'Name', type: 'string' },
    { name: 'description', title: 'Description', type: 'text' },
    { name: 'price', title: 'Price', type: 'number' },
    { name: 'image', title: 'Image', type: 'image' },
    { name: 'category', title: 'Category', type: 'string' },
  ],
};
```

Order Schema:

```
export default {
  name: 'order',
  title: 'Order',
  type: 'document',
  fields: [
    { name: 'userId', title: 'User ID', type: 'string' },
    { name: 'restaurantId', title: 'Restaurant ID', type: 'string' },
    { name: 'products', title: 'Products', type: 'array', of: [{ type: 'reference', to: { type: 'product' } }] },
    { name: 'status', title: 'Status', type: 'string' },
    { name: 'total', title: 'Total', type: 'number' },
  ],
};
```

5. Collaboration Notes:

Team Roles:

- **Project Manager:** Oversees the project and ensures timely delivery.
- **Frontend Developer:** Develops the user interface and integrates APIs.
- **Backend Developer:** Develops the backend services and APIs.
- **Database Administrator:** Manages the database schema and data.
- **QA Engineer:** Tests the application for bugs and issues.
- **Designer:** Designs the UI/UX for the application.
- **Content Manager:** Manages the content on Sanity CMS.

Communication Tools:

- **Slack:** For daily communication and updates.
- **Jira:** For task management and tracking.
- **Zoom:** For meetings and video calls.

Version Control:

- **GitHub:** For version control and code reviews.
- **Branching Strategy:** Use GitFlow for managing branches.

6. Submission from day-2:

Name: Khurram Shahzad

Roll number: 00116699

Class Slot: Friday (9-12)

Teacher: Sir Hamzah Syed

Technical Foundation Plan for Q-Commerce Website (Restaurant-Based) (Tempelate-9)

High-Level Architecture Diagram Description

1. Frontend (Next.js)

- Positioned at the top layer of the diagram.
- Represents the user interface (UI) and user experience (UX) of your Q-commerce website.

Pages to Include:

1. **Home Page**
   - Overview of the restaurant.
   - Highlight popular dishes and categories.
   - Include search functionality.
2. **Product Listing Page**
   - Display a list of menu items or products.
   - Filter and sort functionality (e.g., by cuisine, price, ratings).
3. **Product Detail Page**
   - Dynamic routing for each product.
   - Details of the menu item, including images, description, price, and availability.
   - Option to add items to the cart.
4. **Cart Page**
   - List of selected items with quantities.
   - Option to update or remove items.
   - Display subtotal, taxes, and total price.
5. **Checkout Page**
   - User details (address, contact information).
   - Payment options.
   - Order summary.

6. **Order Confirmation Page**
    - Display order details, payment status, and shipment details.
    - Include a button to track shipment or print receipt.

High-Level Architecture Diagram Description

2. Content Management and Product Data (Sanity CMS)

- Positioned centrally in the diagram, connected directly to the **Frontend**.
- Manages:
    - Product data (menu items, descriptions, prices, images).
    - Content (blogs, promotions, restaurant information).
- Data flows from Sanity CMS to the Frontend for rendering dynamic content.

3. Authentication (Clerk)

- Integrated with the **Frontend**.
- Handles user authentication and authorization.
- Enables features like:
    - User login/signup.
    - Protected routes (e.g., cart and checkout pages).

4. Payment Gateway (Stripe)

- Connected to the **Frontend**.
- Handles payment processing during checkout.
- Data flow:
    - Frontend sends payment details to Stripe.
    - Stripe processes the payment and returns a success/failure response.

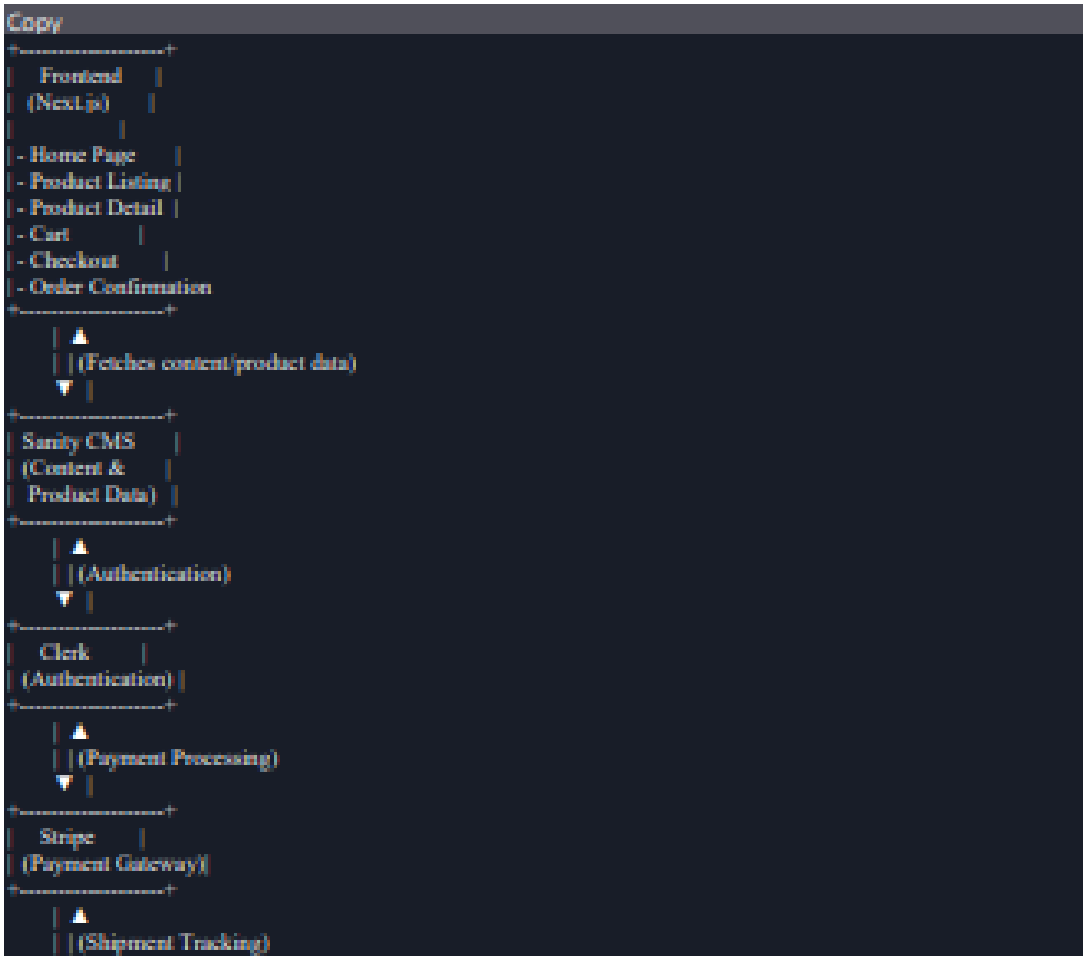
5. Third-Party APIs (ShipEngine)

- Positioned below the **Frontend** layer.
- Used for shipment tracking post-payment.
- Data flow:
    - Frontend sends shipment details to ShipEngine.
    - ShipEngine returns tracking information to the Frontend.

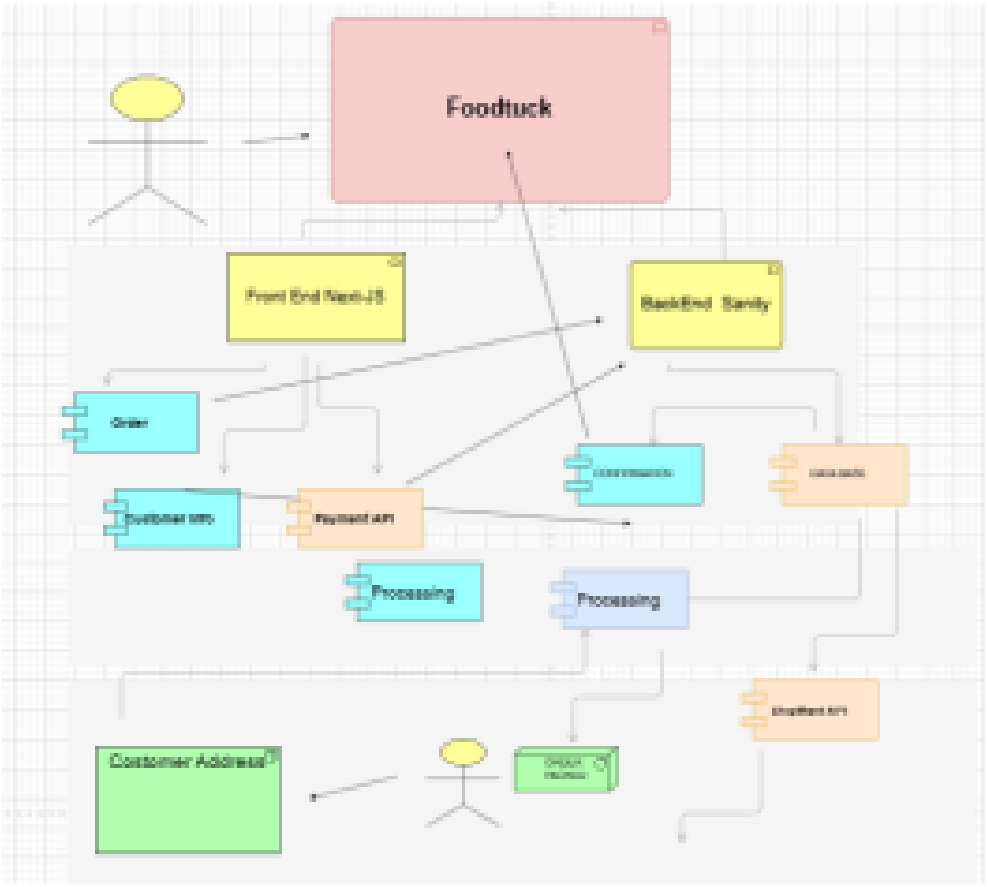
Data Flow

1. **Frontend ↔ Sanity CMS**:
   - Frontend fetches product data and content from Sanity CMS to render pages dynamically.
2. **Frontend ↔ Clerk**:
   - Frontend interacts with Clerk for user authentication and session management.
3. **Frontend ↔ Stripe**:
   - After authentication, the Frontend sends payment details to Stripe for processing.
4. **Frontend ↔ ShipEngine**:
   - Post-payment, the Frontend sends shipment details to ShipEngine and receives tracking information.

Visual Representation (Text-Based Diagram)



```
      ▼ |
+-----------------+
| ShipEngine      |
| (Shipment API)  |
+-----------------+
```



Key Features of the Architecture

1. **Modular Design**: Each component (Frontend, Sanity CMS, Clerk, Stripe, ShipEngine) is independent and scalable.
2. **Dynamic Routing**: Next.js enables dynamic routing for product detail pages (e.g., `/products/[id]`).
3. **Real-Time Updates**: Sanity CMS allows real-time content updates without redeploying the Frontend.
4. **Secure Payments**: Stripe ensures secure and reliable payment processing.
5. **User Management**: Clerk simplifies user authentication and session management.

6. **Shipment Tracking**: ShipEngine provides real-time shipment tracking for a seamless post-purchase experience.

Next Steps

1. **Implement Frontend Pages**:
    - Use Next.js to create the Home, Product Listing, Product Detail, Cart, Checkout, and Order Confirmation pages.
2. **Integrate Sanity CMS**:
    - Set up Sanity CMS for managing product data and content.
3. **Add Authentication**:
    - Integrate Clerk for user authentication.
4. **Set Up Payment Gateway**:
    - Integrate Stripe for payment processing.
5. **Enable Shipment Tracking**:
    - Use ShipEngine API for shipment tracking.
