**Project Documentation: Shopify Data Ingestion & Insights Application**

**Author:** Aditya Goja  
**Date:** 14 September 2025

**1. Assumptions Made**

This project was developed with a specific set of assumptions to meet the core requirements of the assignment within a realistic timeframe.

* **Single-Tenant Focus:** While the database schema is designed to be multi-tenant (with tenantId foreign keys), the application logic and user management are built for a single, pre-defined tenant and user. The primary goal was to demonstrate the data ingestion and dashboard functionality for one specific Shopify store.
* **Admin-Level User:** The application assumes a single administrative user who is created manually via Postman or a database seed script. A full user signup and management flow was considered out of scope.
* **Webhook-First Data Ingestion:** The primary mechanism for data ingestion is real-time Shopify webhooks. A feature to perform a one-time, historical bulk import of all past orders, products, and customers was not implemented but is a logical next step.
* **Free-Tier Infrastructure:** The entire stack is deployed on free tiers (Render for frontend and backend, TiDB Cloud for the database). This is suitable for a demo but would require scaling for a production load.
* **Security Foundation:** Security is based on JWT (JSON Web Token) authentication for API requests. More advanced security measures like rate-limiting, comprehensive input validation, and a full Shopify OAuth2 installation flow are considered next steps for production.
* **Data Consistency:** It is assumed that the data provided by Shopify webhooks is accurate and in the expected format. The application does not perform deep validation on the content of the webhook payloads themselves, only on the webhook's authenticity via Shopify's signature (if implemented).

**2. High-Level Architecture Diagram**

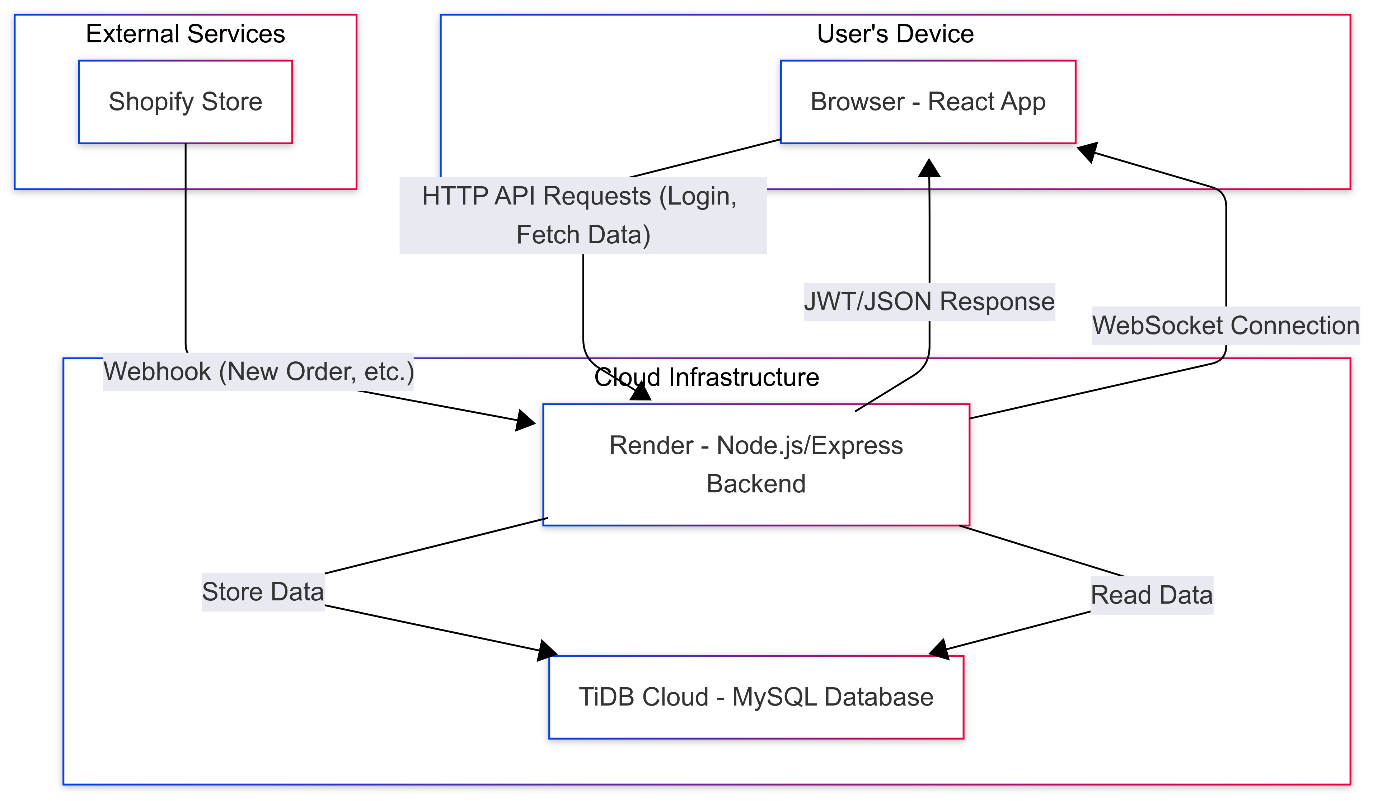
The application follows a modern, decoupled frontend/backend architecture, ensuring a separation of concerns and scalability.

**Components:**

* **User's Browser (Frontend):** A React Single Page Application (SPA) built with Create React App and hosted on Render. It handles the user interface, real-time updates via WebSockets, and API communication.
* **Render Web Service (Backend):** A Node.js/Express server responsible for handling API requests, authenticating users, processing Shopify webhooks, and interacting with the database. It is also hosted on Render.
* **TiDB Cloud (Database):** A MySQL-compatible, serverless SQL database that stores all application data, including tenants, users, and ingested Shopify data.
* **Shopify Store:** The external service that acts as the source of truth. It sends real-time data to our backend via webhooks when events like "order created" occur.

**Data Flow:**

1. **Webhook Ingestion:** A Shopify store sends a webhook (e.g., a new order) to a public endpoint on the Render backend. The backend processes this data and stores it in the TiDB Cloud database.
2. **User Authentication:** A user logs in from the React frontend. The request goes to the backend, which validates credentials against the database and returns a JWT.
3. **Dashboard Data:** The authenticated frontend requests dashboard data. The backend queries the database, aggregates the information, and returns it to the frontend.
4. **Real-Time Updates:** The frontend establishes a persistent WebSocket connection to the backend. When a new webhook is received (Step 1), the backend pushes a dashboardUpdated event through the WebSocket to all connected clients, ensuring the dashboard updates in real-time without needing a refresh.

**Diagram**

**3. APIs and Data Models Used**

**API Endpoints (Backend):**

* POST /api/auth/signup: Creates a new user record. Used manually via Postman for initial setup.
* POST /api/auth/signin: Authenticates a user with email/password, returning a JWT and user data.
* POST /api/webhooks/shopify: Public endpoint to receive and process incoming webhooks from Shopify.
* GET /api/insights/summary: (Authenticated) Returns key metrics for the dashboard (total revenue, orders, etc.).
* GET /api/insights/orders-by-date: (Authenticated) Returns aggregated order data for time-series charts.
* GET /api/insights/top-customers: (Authenticated) Returns a list of top customers by revenue.
* **WebSocket Events:**
  + connect: Establishes a WebSocket connection.
  + dashboardUpdated: Server-sent event to notify clients of new data.

**Data Models (Sequelize/MySQL):**

* **Tenant:** Represents a Shopify store that has installed the application.
  + id (Primary Key)
  + shopifyShopDomain (string, unique): e.g., "your-store.myshopify.com"
  + shopifyAccessToken (string): The token needed for API calls back to Shopify.
* **User:** Represents a user who can log into the dashboard.
  + id (Primary Key)
  + email (string, unique)
  + password (string, hashed)
  + tenantId (Foreign Key to Tenant)
* **Customer:** Ingested from Shopify.
  + id (Primary Key)
  + shopifyCustomerId (string)
  + email (string)
  + totalSpent (decimal)
  + tenantId (Foreign Key to Tenant)
* **Product:** Ingested from Shopify.
  + id (Primary Key)
  + shopifyProductId (string)
  + title (string)
  + vendor (string)
  + tenantId (Foreign Key to Tenant)
* **Order:** Ingested from Shopify.
  + id (Primary Key)
  + shopifyOrderId (string)
  + totalPrice (decimal)
  + createdAt (datetime)
  + tenantId (Foreign Key to Tenant)

**4. Next Steps to Productionize Your Solution**

To transition this project from a functional prototype to a production-ready, multi-tenant SaaS application, the following steps would be critical:

1. **Implement Full Shopify OAuth Flow:**
   * Replace the manual user/tenant creation process with a proper Shopify OAuth 2.0 handshake. This is the standard, secure way to install Shopify apps. The backend would handle the OAuth callback, exchange the temporary code for a permanent access token, and create the Tenant and User records automatically.
2. **Enhance Security and Reliability:**
   * **Webhook Verification:** Implement robust verification for all incoming Shopify webhooks using their HMAC signature to ensure they are authentic.
   * **Webhook Resiliency:** Move webhook processing from an immediate request-response cycle to a background job queue (e.g., using BullMQ with Redis). This prevents timeouts on large webhooks and allows for automatic retries if processing fails, ensuring no data is lost.
   * **Rate Limiting:** Add rate limiting to all public and authenticated endpoints to prevent abuse.
3. **Robust Data Handling:**
   * **Historical Data Sync:** Create a one-time "sync" job that runs after a store installs the app. This job would use the Shopify REST or GraphQL API to pull in all historical customers, products, and orders, providing immediate value instead of starting from zero.
   * **Data Validation:** Implement comprehensive validation on all incoming data, both from API requests and webhook payloads, to ensure data integrity.
4. **Scalability and Monitoring:**
   * **Infrastructure Upgrade:** Move from the free tiers of Render and TiDB Cloud to paid plans that can handle increased traffic and data storage.
   * **Logging and Error Monitoring:** Integrate a third-party service like Sentry or Datadog to capture, report, and alert on any backend errors in real-time.
   * **CI/CD Pipeline:** Implement a full CI/CD pipeline with automated testing (unit, integration) that runs on every commit, ensuring code quality and preventing regressions from reaching production.