Documentation: Multi-Tenant Shopify Data Ingestion & Insights Service

Prepared by: Aditi Shrivastava

Date: September 15, 2025

1. Overview

This document outlines the architecture, assumptions, and technical specifications of the multi-tenant Shopify Data Ingestion & Insights Service. The project's goal is to simulate how an enterprise service would onboard Shopify retailers, ingest their core business data (customers, orders, products), and provide them with a simple analytics dashboard. The solution is built with a focus on multi-tenancy, ensuring that data for each retailer (tenant) is logically isolated and secure.

2. Assumptions Made

During the development of this prototype, several assumptions were made to manage the scope and focus on the core requirements of the assignment.

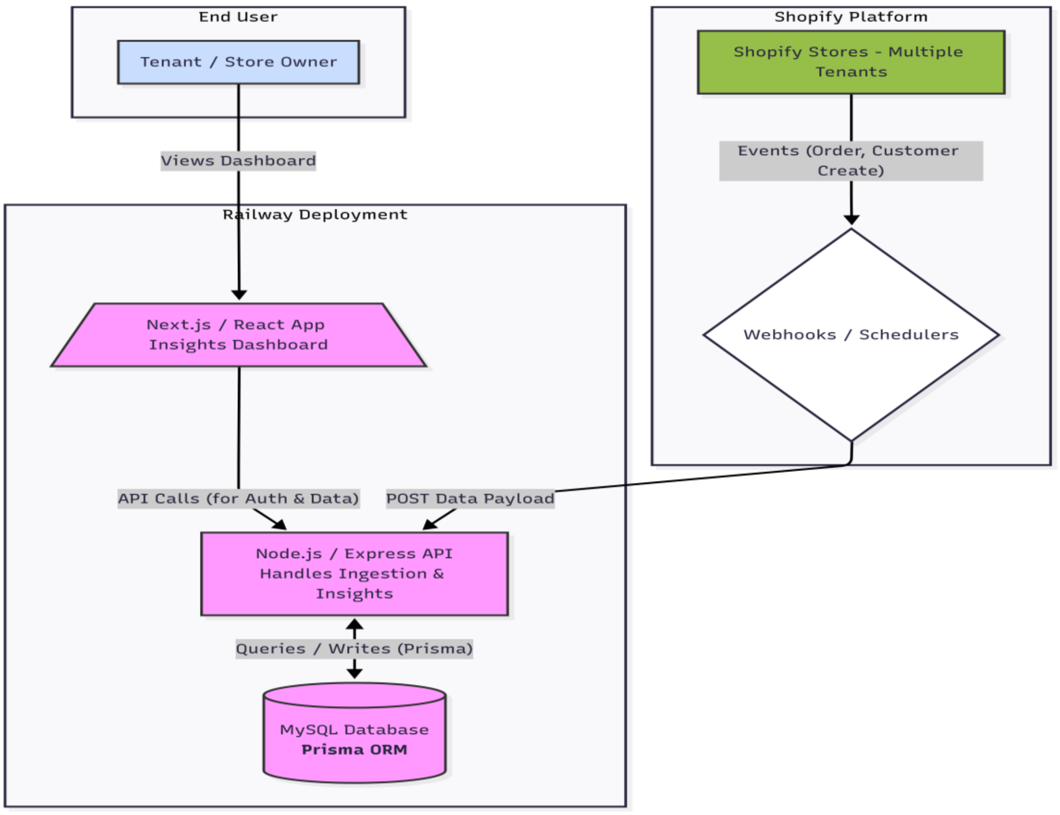
- **Development Environment:** The entire setup and testing process was conducted in a local development environment. A public URL required for Shopify's OAuth callback was created using ngrok, a tunneling service. For a production environment, this would be replaced by a registered domain name hosted on a cloud provider.
- **Authentication Scope:**
  - **Tenant Onboarding:** The service successfully implements Shopify's OAuth 2.0 flow for authenticating and authorizing a Shopify store (a "tenant").
  - **Dashboard Authentication:** The prompt mentioned dashboard authentication (backed by email). This has been architected for (see User model) but not yet implemented. The dashboard is currently public, with the assumption that data is fetched for a pre-determined tenant.
- **Data Ingestion Trigger:** The current implementation focuses on the initial onboarding. The data ingestion logic is designed to be triggered once, immediately after a successful OAuth callback. A continuous, real-time data sync via webhooks is planned as a next step for productionization.
- **Data Scope:** The ingestion is focused on the three core Shopify objects: Products, Orders, and Customers. Bonus objectives like custom events (cart abandoned, checkout started) were not implemented in this phase.
- **Security:** Sensitive data, such as the Shopify accessToken for each tenant, is currently stored in plaintext in the database. This is acceptable for a

development prototype, but in a production environment, these tokens would be encrypted at rest.

- **Scalability:** The architecture uses a single Node.js server instance and a single database. This is sufficient for the assignment's scope but would require a more distributed architecture (e.g., message queues, read replicas) to handle a large number of tenants and high data volume.

3. High-Level Architecture Diagram

The system is composed of four main components: a Frontend Dashboard (Next.js), a Backend Service (Node.js/Express), a Database (MySQL on Railway), and the external Shopify Platform.


End User
Tenant / Store Owner
Views Dashboard
Shopify Platform
Shopify Stores - Multiple Tenants
Events (Order, Customer Create)
Webhooks / Schedulers
Railway Deployment
Next.js / React App Insights Dashboard
API Calls (for Auth & Data)
POST Data Payload
Node.js / Express API Handles Ingestion & Insights
Queries / Writes (Prisma)
MySQL Database Prisma ORM

4. APIs and Data Models Used

Data Models (Prisma Schema)

The database schema is designed with a multi-tenant architecture at its core. A central Tenant table represents each connected Shopify store. All other data tables (e.g., Product, Customer, Order) contain a tenantId foreign key, ensuring that all data is logically partitioned by the store it belongs to.

API Endpoints

The backend exposes a set of RESTful API endpoints to handle authentication and provide data to the frontend.

Method	Endpoint	Description
GET	/api/auth	Initiates the Shopify OAuth 2.0 flow. Expects a shop query parameter (e.g., ?shop=my-store.myshopify.com).
GET	/api/auth/callback	The callback URL that Shopify redirects to after authorization. Handles token exchange and tenant creation.
GET	/api/insights/summary	*(Planned)* Fetches summary metrics (total revenue, orders, customers) for the authenticated tenant.
GET	/api/insights/orders	*(Planned)* Fetches order data within a date range for trend charts.
GET	/api/insights/top-customers	*(Planned)* Fetches the top 5 customers by total spend for the authenticated tenant.

5. Next Steps to Productionize

To move this solution from a prototype to a production-ready service, the following steps would be necessary:

1. **Robust Infrastructure & Deployment:**
    - Deploy the frontend and backend services to a reliable cloud provider (e.g., Vercel for frontend, Railway/Render for backend).
    - Replace ngrok with a registered domain name and configure DNS and SSL certificates.
    - Provision a production-grade managed database with automated backups and scaling capabilities.
2. **Asynchronous Data Ingestion:**
    - Implement a message queue (e.g., RabbitMQ, AWS SQS) to handle the initial data sync. The /api/auth/callback endpoint should only create a "sync job" and add it to the queue, returning a response to the user immediately. A separate worker service would then process the job asynchronously, preventing timeouts and improving the user onboarding experience.

3. **Enhanced Security:**
    - **Encrypt Access Tokens:** All Shopify access tokens must be encrypted at rest in the database using a key management service (e.g., AWS KMS).
    - **Implement Dashboard Authentication:** Build a complete user authentication and authorization system for the frontend dashboard using a library like NextAuth.js or a service like Auth0.
    - **Webhook Verification:** All incoming webhooks from Shopify must have their HMAC signatures verified to ensure they are authentic.
4. **Real-Time Data Synchronization:**
    - Implement webhook handlers for key Shopify events (orders/create, products/update, customers/update, etc.). This would allow the service to maintain near real-time data synchronization instead of relying on periodic full syncs.
5. **Comprehensive Monitoring and Logging:**
    - Integrate structured logging (e.g., using Pino or Winston) to capture application events, errors, and performance metrics.
    - Set up a monitoring and alerting service (e.g., Datadog, Grafana, Sentry) to track application health, API latency, and database performance, with alerts for critical issues.
6. **Full-Fledged Insights API:**
    - Develop the planned insights API endpoints with efficient, tenant-scoped SQL queries to power the dashboard.
    - Implement caching strategies (e.g., with Redis) for frequently accessed dashboard data to reduce database load and improve response times.