Technical Paper: Architectural Enhancements for the Payment Processing API

1. Introduction

The current Payment Processing API successfully fulfills its core requirements: it provides a functional and secure system for transaction management with integrated, rule-based fraud detection. The choice of a modern stack (TypeScript, Prisma) and a portable database (SQLite) makes it an excellent, self-contained solution for evaluation and development.

This paper outlines a strategic roadmap for evolving the application from its current state into a more robust, secure, and scalable production-grade system. The proposed enhancements focus on three key areas: identity and access management, granular authorization, and database architecture.

2. Enhancing Security: From Static Keys to Dynamic Authentication

Current State

The present architecture secures the admin-only endpoints using a static API key passed in an HTTP header. While effective for a simple, internal-use case, this approach presents limitations in a multi-user production environment.

Limitations

- **Single Point of Failure:** A leaked API key compromises the entire admin system.
- **Lack of Accountability:** It is impossible to audit which specific administrator performed an action, only that *an* administrator did.
- **Inflexibility:** It does not scale to accommodate different users or roles.

Proposed Enhancement: JWT-Based Authentication

A more secure and standard approach is to implement a JSON Web Token (JWT) based authentication flow.

The process would be as follows:

1. **User Model:** Introduce a `User` model in the database to store user credentials, including a securely hashed password using a library like `bcrypt`.
2. **Login Endpoint:** Create a `POST /api/auth/login` endpoint. A user submits their credentials, which are validated against the hashed password in the database.
3. **Token Issuance:** Upon successful validation, the server issues a signed, short-lived JWT containing the user's ID and their role.
4. **Authenticated Requests:** The client application would then include this JWT in the `Authorization: Bearer <token>` header for all subsequent requests to protected endpoints. A middleware on the server would verify the token's signature before allowing the request to proceed.

This enhancement shifts the security model from a shared secret to individual, verifiable user identities, laying the groundwork for true accountability and granular control.

3. Implementing Granular Control: Role-Based Access Control (RBAC)

Current State

The current authorization model is binary: a user is either a standard user or an admin. This lacks the nuance required by most real-world applications.

Proposed Enhancement: Role-Based Access Control (RBAC)

Building upon the JWT authentication system, we can implement a flexible RBAC system.

1. **Define Roles:** The `User` model would be updated to include a `role` field, which could be an enum of types such as `MERCHANT_ADMIN`, `SUPPORT_AGENT`, or `SUPER_ADMIN`.
2. **Embed Role in JWT:** The user's role would be included as a claim within the JWT payload upon login.

3. **Authorization Middleware:** A dynamic authorization middleware would be created. This middleware would protect endpoints by checking if the role extracted from the JWT is in a list of allowed roles.

Example Usage:

```
// Example of protecting a route
router.put(
  '/:id/status',
  auth, // JWT authentication middleware
  authorize(['SUPER_ADMIN']), // RBAC middleware
  updateTransactionStatus
);
```

This implementation of the **Principle of Least Privilege** would allow for fine-grained permissions. For instance, a `SUPPORT_AGENT` could be granted read-only access to transactions, while a `MERCHANT_ADMIN` could only view or manage transactions associated with their specific `merchantId`.

4. Scaling the Foundation: Migrating to a Production SQL Database

Current State

The project currently uses SQLite, which is an excellent choice for development and portability due to its serverless, file-based nature.

Limitations

For a high-throughput payment processing system, SQLite presents several challenges in a production environment:

- **Limited Concurrency:** It struggles to handle a high volume of simultaneous write operations, which is common in financial applications.
- **Scalability:** It is not designed to be scaled horizontally across multiple application servers.

- **Lack of Advanced Features:** It lacks the robust management tools, advanced indexing, and performance features of a dedicated client-server database.

Proposed Enhancement: PostgreSQL

The industry-standard solution for applications requiring high data integrity is a relational database like PostgreSQL.

Key benefits include:

- **ACID Compliance:** Guarantees that transactions are processed reliably, a non-negotiable feature for financial systems.
- **High Concurrency:** Can handle thousands of simultaneous connections and write operations efficiently.
- **Scalability and Reliability:** Offers a clear path to scaling through features like read replicas and is renowned for its stability.

Critically, because the project is built with the **Prisma ORM**, this migration is seamless. The application's business logic and query code would remain almost entirely unchanged. The transition would primarily involve updating the `datasource` provider in the `schema.prisma` file and changing the `DATABASE_URL` environment variable.

5. Conclusion

The current API is a robust and well-structured application. By implementing the enhancements outlined in this paper—JWT authentication, Role-Based Access Control, and a migration to PostgreSQL—the project would evolve into a production-ready system capable of meeting the stringent security, scalability, and reliability demands of a real-world financial service.