# Research Report: Recommended Technology Stack and System Architecture for Fresh Roots

**DATE:** 2026-01-22

## Executive Summary

This report outlines the recommended technology stack and system architecture for **Fresh Roots**, a scalable, mobile-first marketplace for fresh produce in Mauritius. The proposed architecture is designed to support rapid development for a one-month prototype, ensure a high-quality user experience, and provide a robust foundation for scaling to millions of users.

The core recommendations are as follows:

- **Frontend:** A **React Native** application built with the **Expo** framework. This choice leverages the developer's preference, enables rapid iteration with web previews, and provides a clear path for future iOS support from a single JavaScript/TypeScript codebase.
- **Backend:** A **Node.js** API server using the **NestJS** framework. This stack is selected for its exceptional performance in handling concurrent I/O-bound operations typical of a marketplace, its native support for real-time features, and its use of TypeScript, which aligns with the frontend for a unified development experience.
- **Database:** A **hybrid database strategy** is recommended. **PostgreSQL** will serve as the primary database for structured, transactional data such as users, product listings, and orders, ensuring high data integrity. **Redis** will be used for caching and real-time messaging.
- **API Architecture:** A **RESTful API** is proposed for its simplicity, widespread adoption, and suitability for the CRUD-centric operations of the initial application.
- **Infrastructure:** The architecture relies on a suite of best-in-class, scalable services: **AWS S3** with **Cloudinary** for optimized image storage and delivery; **JSON Web Tokens (JWT)** for stateless authentication; **Redis** for high-performance caching; **Firebase Cloud Messaging (FCM)** for push notifications; **SendGrid** for transactional emails; and **GitHub Actions** for CI/CD automation.

This technology stack is cost-effective for the initial phase while being inherently scalable. It prioritizes developer experience to meet the aggressive one-month prototype timeline and addresses the specific requirements of the Mauritian market, including provisions for local payment methods and offline tolerance.

## 1. Frontend Technology Stack

The frontend is the primary user interface for Fresh Roots. The choice of technology directly impacts user experience, development speed, and maintenance overhead. The recommendation is for a mobile-first application with an initial focus on Android.

### 1.1. Mobile Framework: React Native with Expo

**Recommendation:** React Native with the Expo framework.

**Justification:**
* **Rapid Development:** Expo provides a managed workflow that abstracts away much of the complexity of native development. Features like Expo Go allow for instant testing on physical devices (including the target Samsung J7 Prime) without building a full APK, and web previews enable live iteration, dramatically accelerating the development cycle.
* **Cross-Platform Capability:** While the initial focus is Android, React Native allows the same codebase to be used to build an iOS application in the future, saving significant time and resources. Expo further simplifies this by managing build configurations for both platforms.
* **Developer Experience:** The developer's stated preference for React Native/Expo is a critical factor. A familiar and productive environment is key to meeting the one-month prototype timeline. The use of JavaScript/TypeScript creates a cohesive development experience across the full stack.
* **Ecosystem:** React Native has a mature and extensive ecosystem of libraries and tools, ensuring that solutions for common problems are readily available.

## 1.2. Navigation

**Recommendation:** React Navigation.

**Justification:**
React Navigation is the de-facto standard for routing and navigation in React Native applications. It is highly customizable, well-documented, and provides all necessary navigation patterns for a marketplace app, including stack navigators (for drilling into product details), tab navigators (for main app sections like Home, Orders, Profile), and drawer navigators.

## 1.3. State Management

**Recommendation:** Redux Toolkit.

**Justification:**
For an application intended to scale, a predictable and robust state management solution is essential. Redux Toolkit is the official, opinionated toolset for efficient Redux development. It simplifies store setup, reduces boilerplate code, and includes powerful tools like Immer for immutable updates and Redux Thunk for asynchronous logic. This will be crucial for managing global state such as user authentication, shopping cart contents, and cached server data.

## 1.4. Offline Support

**Recommendation:** A combination of AsyncStorage and a caching layer.

**Justification:**
To provide a tolerant experience on intermittent networks, the app must cache data.
* **AsyncStorage:** This simple, unencrypted, asynchronous, persistent key-value storage system is bundled with React Native. It is suitable for storing small amounts of non-sensitive data like user settings or session tokens.
* **Application-Level Caching:** For larger datasets like product listings, a more sophisticated caching strategy will be implemented. When data is fetched from the API, it will be stored locally. Subsequent views will load from the cache first for an instant UI response, while a background fetch updates the data. This "stale-while-revalidate" pattern ensures the app feels fast even on slow networks.

## 1.5. UI Component Library

**Recommendation:** React Native Paper.

**Justification:**
React Native Paper is a high-quality, cross-platform component library based on Google's Material

Design. It provides a comprehensive set of pre-built, customizable components (buttons, cards, inputs, modals) that will accelerate UI development, ensure visual consistency, and adhere to modern design standards.

# 2. Backend Technology Stack

The backend is the engine of the marketplace, responsible for business logic, data processing, and communication with the database and third-party services.

## 2.1. API Server: Node.js with NestJS

**Recommendation:** Node.js with the NestJS framework.

**Justification:**
* **Performance and Scalability:** Node.js's non-blocking, event-driven architecture is exceptionally well-suited for I/O-heavy applications like a marketplace, which involves numerous database queries and API calls. It can handle a high number of concurrent connections with low resource usage, making it highly scalable.
* **Unified Language:** Using TypeScript on the backend with NestJS and on the frontend with React Native creates a cohesive development environment. This reduces context-switching for developers and allows for code-sharing of types and interfaces, improving productivity and reducing errors.
* **Framework Maturity:** NestJS is a mature, opinionated framework built on top of Express.js. It imposes a modular architecture inspired by Angular, which is excellent for building maintainable and scalable microservices. This structure is ideal for starting with a monolith for the MVP and cleanly breaking out services (e.g., notifications, payments) as the application grows.
* **Real-time Capabilities:** NestJS has first-class support for WebSockets, which will be essential for implementing real-time notifications for the admin dashboard and users.

**Comparison with Alternatives:**
* **Python/Django:** While excellent for rapid development with its "batteries-included" approach, Django can be less performant for high-concurrency, real-time scenarios compared to Node.js. Its synchronous nature can require more complex setups (like Celery and Channels) to achieve the same level of real-time responsiveness.
* **Go:** Go offers superior raw performance and concurrency. However, its ecosystem for web development is less mature than Node.js's, and it has a steeper learning curve. For this project, the productivity gains and unified ecosystem of the Node.js/NestJS stack outweigh the raw performance benefits of Go.

# 3. Database and Data Management

The choice of database is critical for data integrity, performance, and scalability.

## 3.1. Database System: Hybrid (PostgreSQL + Redis)

**Recommendation:** A hybrid approach utilizing PostgreSQL as the primary database of record and Redis for caching and real-time messaging.

**Justification:**
* **PostgreSQL for Core Data:** A marketplace deals with highly structured and relational data (users, products, orders, payments). PostgreSQL's strict schema, ACID compliance, and support for complex transactions are non-negotiable for ensuring data integrity. It prevents issues like orphaned order records or incorrect inventory counts. Its powerful JSONB support also provides flexibility to store semi-structured data like product tags or metadata within the relational model.

* **Redis for Caching and Ephemeral Data:** Redis is an in-memory data store that provides extremely fast read and write operations. It is the ideal choice for caching API responses to reduce database load and improve app performance. It will also be used to manage real-time components, such as a Pub/Sub system for pushing notifications to the admin dashboard.

**Comparison with Alternatives:**
* **MongoDB-only:** While MongoDB's flexible schema is attractive for rapid prototyping, it lacks the inherent referential integrity of a relational database. For core e-commerce transactions, ensuring consistency through application-level code is risky and complex. A bug could easily lead to data corruption. Therefore, it is not recommended as the sole database for transactional data.

## 3.2. Database Schema Design Considerations

The PostgreSQL schema will be designed with normalization and scalability in mind.

- **`users`:** Stores user information (ID, name, email, hashed password, roles like 'admin' or 'customer').
- **`listings`:** Contains product details (ID, title, description, price, unit, location, `created_by` foreign key to `users`).
- **`listing_images`:** A separate table to allow multiple images per listing (ID, `listing_id` foreign key, image_url, order).
- **`inventory`:** Tracks stock levels (ID, `listing_id` foreign key, quantity, `updated_at`).
- **`orders`:** Represents a purchase request (ID, `user_id` foreign key, status, total_amount, payment_method).
- **`order_items`:** Line items for an order (ID, `order_id` foreign key, `listing_id` foreign key, quantity, price_at_time_of_purchase).
- **`audit_logs`:** Records significant admin actions (ID, `user_id` of admin, `target_entity_id` (e.g., order ID), action, timestamp).

# 4. API Architecture

**Recommendation:** REST (Representational State Transfer).

**Justification:**
REST is a mature, well-understood architectural style that aligns perfectly with the resource-oriented nature of a marketplace. Endpoints will map directly to CRUD (Create, Read, Update, Delete) operations on resources like `/listings`, `/orders`, and `/users`.

- **Simplicity and Familiarity:** REST is simpler to implement for the MVP compared to GraphQL. The ecosystem of tools for testing, documenting (e.g., Swagger/OpenAPI), and caching REST APIs is vast and mature.
- **Performance:** For a mobile client, well-designed REST endpoints can be highly performant. While GraphQL can solve over-fetching, this can also be managed in REST with proper query parameters and tailored endpoints.

GraphQL could be considered in the future if the client's data needs become significantly more complex, requiring fetching deeply nested and varied data structures in a single request. For the prototype, REST provides the fastest path to a robust API.

# 5. Core Infrastructure and Services

## 5.1. Authentication

**Recommendation:** JWT (JSON Web Tokens) with OAuth 2.0 options.

**Justification:**
* **JWT:** JWTs are ideal for securing a stateless API that serves a mobile client. After a user logs in, the server issues a signed JWT. The mobile app stores this token and includes it in the header of subsequent requests. The server can verify the token's signature without needing to query a database, making it highly scalable.
* **OAuth 2.0:** For user convenience, integrating social login providers like Google and Facebook via OAuth 2.0 is recommended. This simplifies the sign-up process and reduces friction for new users.

## 5.2. Caching

**Recommendation:** Redis.

**Justification:**
Redis will be a critical component for performance and real-time functionality.
* **API Caching:** Implementing a **cache-aside** pattern. When a request for data (e.g., a list of products) is received, the backend will first check Redis. If the data exists (a cache hit), it's returned instantly. If not (a cache miss), the backend queries PostgreSQL, stores the result in Redis with a set expiration time (TTL), and then returns it to the client.
* **Real-time Features:** Redis's Pub/Sub capabilities can be used to push real-time updates. For example, when a new order is placed, the API can publish a message to a Redis channel. A WebSocket service listening to that channel can then push a notification to the connected admin dashboard.

## 5.3. File & Image Storage

**Recommendation:** AWS S3 for storage and Cloudinary for delivery and transformation.

**Justification:**
* **AWS S3 (Simple Storage Service):** S3 is an industry-standard, highly durable, and cost-effective solution for storing raw image files uploaded by the admin.
* **Cloudinary (as a CDN and Transformation Layer):** While S3 can serve files, a specialized service like Cloudinary offers significant advantages for a mobile app. Images stored in S3 can be synced to Cloudinary. When the mobile app requests an image, it can request it from Cloudinary's CDN with on-the-fly transformations (e.g., resizing to a specific width for a thumbnail, compressing for mobile networks, converting to modern formats like WebP). This ensures that images are delivered quickly and optimized for the user's device, which is crucial for performance and data usage.

## 5.4. Payment Integration (Mauritius Context)

**Recommendation:** Prioritize Cash on Delivery (COD) for the MVP, with a deep-linking approach for "Juice" by MCB.

**Justification:**
* **Investigation Findings:** Public developer APIs and SDKs for direct integration with the "Juice" by MCB app are not readily available. This is common for local banking applications.
* **Recommended Approach:**
1. **Cash on Delivery (COD):** This should be the primary payment method for the MVP. It is risk-free from a technical integration standpoint and widely accepted.
2. **"Juice" via Deep Link/Redirect:** The app can present "Pay with Juice" as an option. When selected, it would generate a payment request with a unique reference number and either attempt to deep-

link into the Juice app or display instructions for the user to make a manual transfer to a specified MCB account number with the reference. The admin would then have to manually verify the payment receipt.
3. **Future Scalable Option:** Integrate **Stripe** as a fallback. Stripe is available in Mauritius and provides a robust, secure, and well-documented solution for accepting credit/debit card payments, which will be essential for future growth.

### 5.5. Facebook Page Integration

**Recommendation:** A semi-automated import script using the Facebook Graph API, with manual oversight.

**Justification:**
* **Graph API Approach:** It is feasible to access posts from the public Facebook page using the Graph API. This requires a Facebook Developer App and obtaining a Page Public Content Access feature approval. The script would query the page's feed ( `/page-id/posts` ) and parse the post text and images to create draft listings in the Fresh Roots database.
* **Limitations and Fallback:** This approach is subject to rate limits, API changes, and the need for Facebook's app review process. The format of posts may not always be consistent. Therefore, the recommended workflow is a **semi-automated import**. The script runs and creates draft listings, which the admin then reviews, cleans up, and publishes from the admin dashboard. This combines the speed of automation with the reliability of human oversight.

## 6. Notifications and Monitoring

### 6.1. Push Notifications

**Recommendation:** Firebase Cloud Messaging (FCM).

**Justification:**
FCM is Google's free, cross-platform messaging solution. It is the standard for sending push notifications to Android devices and also fully supports iOS. It is reliable, scalable to billions of devices, and integrates well with the Node.js backend via official SDKs.

### 6.2. Email Notifications

**Recommendation:** SendGrid.

**Justification:**
For critical admin alerts (e.g., new purchase requests, low stock warnings), a reliable transactional email service is required. SendGrid offers a generous free tier, excellent deliverability rates, and robust APIs for sending emails from the backend. It provides analytics on email opens and clicks, which can be valuable. Mailgun is a strong alternative with a similar feature set.

### 6.3. Application Monitoring

**Recommendation:** A suite of Sentry, a structured logger, and cloud-native tools.

**Justification:**
* **Error Tracking: Sentry** is an essential tool for both the frontend and backend. It automatically captures errors and exceptions in real-time, providing detailed stack traces and context that are invaluable for debugging.
* **Logging:** The NestJS backend should use a structured logging library like **Winston**. Logs should be output as JSON and forwarded to a centralized logging service like AWS CloudWatch Logs or the ELK

Stack (Elasticsearch, Logstash, Kibana). This allows for powerful searching, filtering, and alerting on log data.
* **Performance Monitoring:** Cloud provider tools (e.g., AWS CloudWatch) should be used to monitor fundamental infrastructure metrics like CPU utilization, memory, and database connections.

## 7. Development and Deployment (CI/CD)

**Recommendation:** GitHub Actions.

**Justification:**
Assuming the project's source code will be hosted on GitHub, GitHub Actions is the most seamless choice for Continuous Integration and Continuous Deployment.
* **Integration:** Workflows are defined in YAML files directly within the repository, versioned alongside the code.
* **Flexibility:** Separate workflows can be created for:
* **Backend:** On every push to the `main` branch, automatically run tests, build a Docker container, and deploy it to the hosting service (e.g., AWS ECS, Google Cloud Run).
* **Frontend:** On every push, run tests and linting. On a new tag/release, trigger a build using Expo's Application Services (EAS) to generate a signed Android APK/AAB for distribution.
* **Cost-Effective:** GitHub Actions has a generous free tier for public and private repositories, which is more than sufficient for the initial stages of the project.

## 8. Conclusion

The proposed technology stack—React Native/Expo, Node.js/NestJS, and PostgreSQL—provides a modern, scalable, and highly productive foundation for building the Fresh Roots marketplace. This architecture is specifically tailored to meet the project's goals: rapid prototyping within one month, long-term scalability to millions of users, and addressing the unique context of the Mauritian market. By leveraging managed services, mature frameworks, and a unified TypeScript ecosystem, the development team can focus on delivering business value and creating a high-quality user experience.

# References


1. Best Tech Stack for Mobile App Development in 2024 - Linkitsoft (https://linkitsoft.com/best-tech-stack-for-mobile-app-development/)
2. How to Choose the Right Tech Stack for a Mobile App in 2024 - Onix Systems (https://onix-systems.com/blog/how-to-choose-right-tech-stack-for-mobile-app)
3. Best Tech Stack for a Startup in 2024 - Squareboat (https://www.squareboat.com/blog/best-tech-stack-for-a-startup)
4. Choosing the Right Tech Stack for Web & Mobile App Development - Bacancy Technology (https://www.bacancytechnology.com/blog/choosing-the-right-tech-stack)
5. React Native vs Flutter in 2024: Detailed Comparison - Fireart Studio (https://fireart.studio/blog/flutter-vs-react-native-cross-platform/)
6. React Native vs Flutter: What to Choose in 2025 - BrowserStack (https://www.browserstack.com/guide/flutter-vs-react-native)
7. Flutter vs React Native: Complete 2025 Framework Comparison Guide - The Droidsonroids (https://www.thedroidsonroids.com/blog/flutter-vs-react-native-comparison)

8. Flutter vs. React Native: Which Framework to Choose for Your Next Mobile App Development? - Netguru (https://www.netguru.com/blog/flutter-vs-react-native)
9. Flutter vs. React Native in 2025 - Nomtek (https://www.nomtek.com/blog/flutter-vs-react-native)
10. Comparing Web Frameworks: Flask, FastAPI, Django, NestJS, Express.js - Medium (https://medium.com/@arif.rahman.rhm/comparing-web-frameworks-flask-fastapi-django-nestjs-express-js-db735f1c6eba)
11. Python Django or Nest.js? Which is best for making a chatting app with media sharing? - DEV Community (https://dev.to/nadim_ch0wdhury/python-django-or-nest-js-which-is-best-for-making-a-chatting-app-with-media-sharing-44jl)
12. Node.js vs Python: Which is Best for Backend Development in 2025? - Kanhasoft (https://kanhasoft.com/blog/node-js-vs-python-which-is-best-for-backend-development-in-2025/)
13. E-commerce shopping cart and checkout process, use relational or NoSQL? - Stack Overflow (https://stackoverflow.com/questions/49700533/ecommerce-shopping-cart-and-checkout-process-use-relational-or-nosql)
14. MongoDB vs PostgreSQL: A Comprehensive Comparison - DevToolsAcademy (https://www.devtoolsacademy.com/blog/mongoDB-vs-postgreSQL/)
15. MongoDB vs. PostgreSQL - MongoDB (https://www.mongodb.com/resources/compare/mongodb-postgresql)
16. PostgreSQL vs. MongoDB: A Full Comparison - Estuary (https://estuary.dev/blog/postgresql-vs-mongodb/)
17. Best practices of using data caching (Redis, Local Storage) in React Native projects - Medium (https://medium.com/@tusharkumar27864/best-practices-of-using-data-caching-redis-local-storage-in-react-native-projects-e151c76b2df0)
18. How Redis Caching in Mobile Apps Can Enhance User Experience - Reintech (https://reintech.io/blog/redis-caching-mobile-apps-user-experience-enhancement)
19. Azure Cache for Redis - Microsoft Azure (https://azure.microsoft.com/en-us/products/cache)
20. Caching with Redis - Redis (https://redis.io/solutions/caching/)
21. Microservices Caching Strategies with Redis - Redis (https://redis.io/learn/howtos/solutions/microservices/caching)