GrocARy

AR Grocery Shopping Companion

Kovács Bálint-Hunor

January 2026

Contents

1 Introduction

1.1 Project Overview

GrocARy is an innovative hybrid mobile application designed to enhance the grocery shopping experience using Augmented Reality (AR). By bridging the gap between physical products and digital information, GrocARy allows users to scan product barcodes and instantaneously visualize critical product data, such as Nutri-Score, allergens, and ingredients overlaid directly on the product in the real world.

The core philosophy of GrocARy is simplicity and efficiency. Shoppers often struggle with reading small print on labels or quickly understanding the nutritional value of a product. GrocARy solves this by presenting a clear, high-contrast, standardized "Virtual Label" that floats next to the item.

1.2 The Development Journey

The journey to building GrocARy was iterative and involved significant architectural pivots.

1.2.1 Initial Vision: Full Stack Approach

Initially, the project was conceived as a full-stack application with a custom backend. The plan was to use a **Bun** runtime with a **Hono** web framework for a high-performance backend, serving a React Viro frontend. The goal was to maintain a custom database of products.

1.2.2 The Pivot: Direct API Integration

However, during the planning phase, it became evident that maintaining a comprehensive grocery database is a monumental task for an MVP. To accelerate development and provide immediate value, I decided to pivot away from a custom backend. Instead, GrocARy integrates directly with **OpenFoodFacts (OFF)**, a free, open collaborative database of food products from around the world. This largely simplified the infrastructure, removing the need for server maintenance and database synchronization, allowing me to focus entirely on the mobile experience and AR interactions.

1.2.3 Framework Challenges

The choice of mobile framework also presented challenges. An attempt was made to initialize a fresh React Native project and integrate `viro-react` manually. This led to "dependency hell", a situation where mismatched native library versions and peer dependencies caused build failures.

After a week of troubleshooting configuration conflicts between modern React Native versions and Viro's specific requirements, a strategic decision was made to use a preconfigured Viro template. This provided a stable baseline, which was then modernized and customized to fit the specific needs of GrocARy.

2 Technology Stack

GrocARy leverages a modern, robust stack tailored for hybrid mobile development:

- **React Native (0.73.3)**: The core framework, enabling cross-platform mobile development with a single codebase.
- **TypeScript**: Ensuring type safety and code maintainability across the application.
- **ViroReact (AR)**: A comprehensive developer platform for building AR/VR applications. It creates the 3D scene and handles the tracking of the device in the physical world.
- **React Native Vision Camera**: A high-performance camera library used for the initial barcode scanning phase. It offers superior frame processing capabilities compared to Viro's built-in tracking for this specific use case.
- **OpenFoodFacts API**: The source of truth for product data, accessed via REST API.
- **Jest**: Used for unit testing the logic and ensuring the reliability of data parsing and utility functions.

3 System Architecture

3.1 System Design

The architecture of GrocARy is designed to be lean and event-driven. The following diagram illustrates the high-level system components and their interactions.

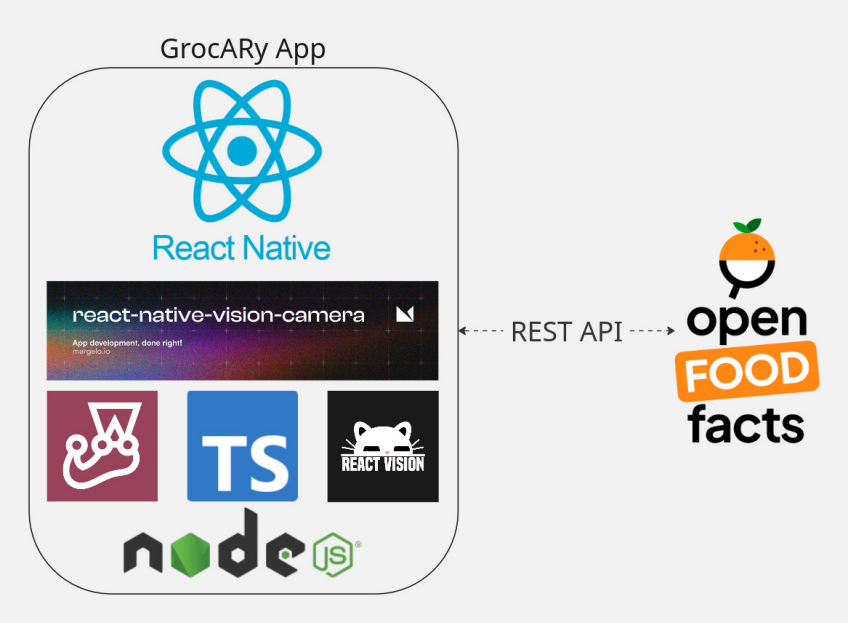


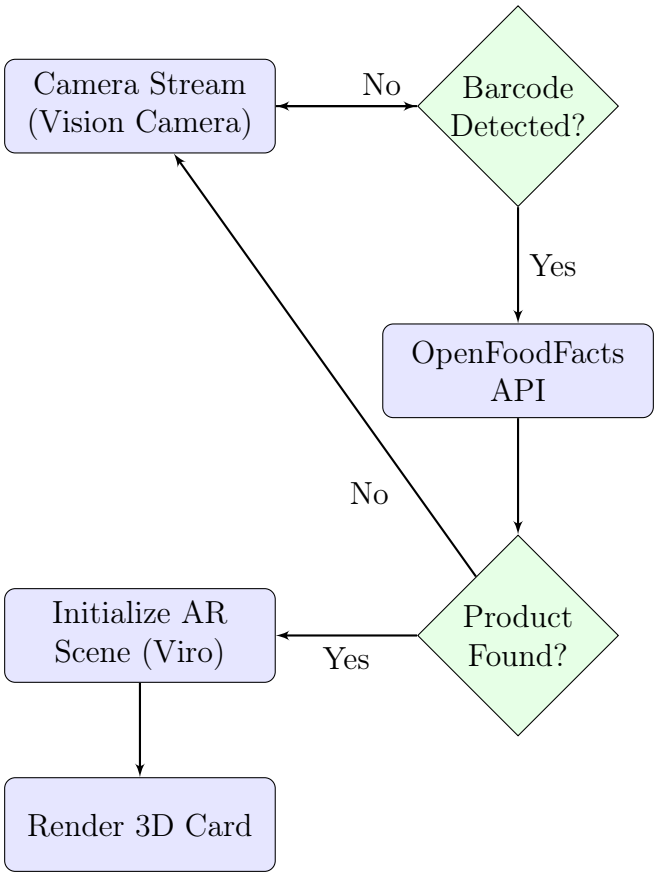
Figure 1: System Architecture Diagram

3.2 Application Flow

The application operates in two distinct modes to optimize performance and resource usage: **Scanning Mode** and **AR View Mode**.

1. **Scanning Mode**: The app starts here. It uses `react-native-vision-camera` to actively search for barcodes (EAN-13, EAN-8). The AR engine is paused or unmounted to save battery.
2. **Data Fetching**: extensive validation is performed on the scanned code. If valid, an asynchronous request is sent to OpenFoodFacts.
3. **AR View Mode**: Upon successful data retrieval, the app switches context. The scanning camera is unmounted, and the Viro AR scene is mounted. The retrieved data is injected into the 3D scene.

3.2.1 Flow Diagram



4 Testing Strategy

To ensure the reliability of the application, I have incorporated **Jest** as the testing framework. Currently, the testing strategy focuses on unit testing the critical logic, specifically the data normalization and parsing functions.

Since the application relies heavily on external data from OpenFoodFacts, it is crucial to verify that the `buildFullName` logic and other utility functions handle various data

shapes and edge cases correctly.

Note: The test suite will be expanded to include snapshot testing for UI components and integration tests for the scanner flow in future iterations.

5 Implementation Details

5.1 Managing the Camera Conflict

One of the most significant technical hurdles was the conflict between `react-native-vision-camera` and `ViroReact`. Both libraries attempt to adhere to the device's camera resource exclusively.

Running them simultaneously results in a race condition where one library fails to initialize or the camera stream freezes.

Solution: I implemented a strict state-machine approach. The `App.tsx` component manages a `appMode` state:

```
type AppMode = "SCANNING" | "AR_VIEW";
const [appMode, setAppMode] = useState<AppMode>("SCANNING");

// Conditional Rendering
{appMode === "SCANNING" ? (
    <Camera ... />
) : (
    <ViroARSceneNavigator ... />
)}
```

By conditionally rendering only one component tree at a time, I ensure that the camera resource is released by one library before being claimed by the other. This "handoff" is managed with a slight delay (`setTimeout`) to allow for proper cleanup of native handles.

5.2 AR Layout Strategy

Positioning text and images in 3D space is fundamentally different from 2D web layout. Flexbox is supported by Viro but behaves inconsistently compared to standard CSS.

To achieve a stable, "glassmorphism" card layout:

- **ViroFlexView**: Used as the main container for the card. It provides a bounded 2D surface within the 3D world.
- **Absolute Sizing**: Instead of relying on percentage-based responsiveness (which is ambiguous in 3D), I use fixed coordinate units for width and height.
- **Materials**: A custom `glassCard` material was created using a transparent black diffuse color (`rgba(20, 20, 20, 0.9)`) combined with constant lighting to ensure readability regardless of the environment's lighting conditions.

5.3 Data Normalization

Data from OpenFoodFacts can be inconsistent. The app implements a robust normalization layer (`buildFullName`, `fetchOpenFoodFacts`) that:

- Prioritizes specific fields (e.g., `image_front_url` over `image_url`).
- Sanitizes allergen tags (removing `en:` prefixes).
- Provides fallbacks for missing data (e.g., "Unknown Product" placeholders).

6 Conclusion

GrocARy demonstrates the viability of high-utility AR applications built with web technologies. By combining the raw performance of native modules for scanning with the immersive capabilities of ViroReact, and backing it with the massive OpenFoodFacts dataset, the application delivers a seamless user experience.

The architecture is designed to be extensible. Future work could re-introduce a lightweight backend for user favorites or caching, but the current serverless architecture serves the MVP goals perfectly: fast, reliable, and maintenance-free.