Project Report: Metal Price Tracker Mobile Application

1. Solution Details & Project Overview

The Metal Price Tracker is a cross-platform mobile application built with React Native. Its primary purpose is to provide users with real-time price information for precious metals (Gold, Silver, Platinum, Palladium) and allow them to simulate an investment lifecycle, including buying and selling their holdings.

The application is architected around a component-based, screen-oriented structure, making it scalable and easy to maintain.

Key Features Implemented:

- **Real-Time Price Display:** The landing screen fetches and displays live metal prices from an external API in a user-friendly 2x2 grid.
- **API Integration:** A dedicated service module (metalService.js) handles all communication with the metalpriceapi.com API, abstracting the logic for data fetching, error handling, and data transformation.
- **Investment Flow:** Users can initiate a purchase by selecting a metal, entering an investment amount in INR, and providing personal details on a unified investment screen.
- **Portfolio Management:** All user investments are stored in a global state. A dedicated "Profile" screen lists all purchased assets, showing the original amount invested and the price at the time of purchase.
- **Selling Mechanism:** Users can view their investments for a specific metal and choose to "sell" them. This action removes the investment from their portfolio, simulating a complete transaction lifecycle.

Technical Architecture:

- **Technology Stack:**
    - **React Native (Expo Go / Cli):** A framework for building native mobile apps using JavaScript and React. Expo Go was used for rapid development and testing.
    - **React Navigation:** A library for handling routing and navigation between different screens. A StackNavigator was used to manage the hierarchical flow.
    - **React Context API:** Used for simple, centralized state management. The InvestmentContext holds the user's portfolio data, making it accessible to any component without prop-drilling.
- **Project Structure:**
    - screens/: Contains the main screen components of the app (e.g., LandingScreen.js, BuyScreen.js, ProfileScreen.js).
    - components/: Contains reusable UI components used across multiple screens (e.g., MetalTile.js, Loader.js).

- services/: Holds modules for external communication, specifically metalService.js for API calls.
- context/: Manages the global state of the application (InvestmentContext.js).
- navigation/: Defines the navigation structure and routes (AppNavigator.js).

2. How to Execute the Project

Follow these steps to set up and run the application on your local machine.

Prerequisites:

1. **Node.js and npm:** Ensure you have Node.js (LTS version) and npm installed.
2. **Expo Go App:** Install the "Expo Go" application on your physical Android or iOS device. Alternatively, you can use an Android Studio Emulator or iOS Simulator
3. **Expo CLI:** Install the Expo command-line tool globally:
   npm install -g expo-cli

OR

Community CLI - npx @react-native-community/cli init MetalPriceTrack

Setup and Installation:

1. **Clone the Repository:** Download or clone the project folder to your machine.
2. **Navigate to Project Directory:** Open your terminal and cd into the project's root folder.

3. Install Dependencies: Run the following command to install all the required packages defined in package.json.
npm install & required Dependencies for project

4. Configure API Key: Open services/metalService.js. Although the key is provided, this is where you would replace it if it expires or if you use a different account.

const API_KEY = ' ';

Running the Application:
Start the Metro Server: In the project's root directory, run: - **npx expo start &**

For the cli project to run - npx react-native run-android

1. **Connect Your Device:**
   - The terminal will display a QR code.
   - Open the Expo Go app on your physical device and scan the QR code.
   - If using an emulator, you can press a in the terminal to open it on a running Android emulator, or i for an iOS simulator.
2. The application will now build and launch on your device/emulator.

3. Deployment Notes

To deploy this application to the Apple App Store or Google Play Store, you would move beyond Expo Go and create standalone production builds.

1. **Expo Application Services (EAS):** The modern standard for building and deploying Expo apps is EAS.
    - **EAS CLI:** Install the EAS command-line tool: npm install -g eas-cli.
    - **Configuration:** Log in with your Expo account (eas login) and configure the project (eas build:configure).
    - **Build:** Run eas build --platform android or eas build --platform ios to create a production-ready .apk/.aab or .ipa file.

1.2 **Community Cli -**

    - **Prepare for Android - Prepare KeyStore, then Replace `my-upload-key.keystore` and `my-key-alias` with your desired names.**
    - **Config the KeyStore in Gradle & Generate Release APK**
    - `cd android`
    - `./gradlew bundleRelease # For AAB (recommended for Play Store)`
    - `./gradlew assembleRelease # For APK`
    -
2. **Environment Variables (Crucial for Security):**
    - **Problem:** The API key is currently hardcoded in metalService.js, which is insecure and bad practice for production.
    - **Solution:** Store the API key in an environment variable. Create a file named .env in the project root and add API_KEY=your_actual_api_key. Then, use a library like react-native-dotenv to access it in your code via process.env.API_KEY. This keeps your secret keys out of version control.
3. **App Store Submission:** The build artifacts generated by EAS can be directly uploaded to the Google Play Console and App Store Connect for review and release.

---

4. Approach, Challenges, and Future Improvements

Bullet Points on Approach:

- **Component-First Design:** The UI was broken down into reusable components (MetalTile, Loader) to ensure consistency and maintainability.
- **Centralized API Logic:** All API interactions were isolated in metalService.js. This makes it easy to swap API providers or update endpoints without changing the UI components.
- **Global State for Simplicity:** React Context was chosen for state management as it is built into React and sufficient for managing the simple, shared state of the investment portfolio.

- **User-Centric Flow:** The navigation was designed to be intuitive, guiding the user logically from viewing prices to buying, checking their profile, and selling.

Challenges and Unsolved Notes:

- **Challenge: API Provider Changes:** The project initially used a different API (goldapi.io) which failed due to authentication issues (headers vs. URL parameters). The solution was to pivot to a new provider (metalpriceapi.com) and refactor the metalService.js module entirely, demonstrating adaptability in handling external service dependencies.
- **Challenge: Data Formatting:** The metalpriceapi.com API returns rates as 1 USD = X amount of metal. The service had to be programmed to calculate the inverse (1 / rate) to get the correct price of 1 ounce in USD.
- **Unsolved - Data Persistence:**
    - **Current State:** Investments are lost every time the app is closed because the state is stored in-memory.
    - **Future Solution:** Implement **AsyncStorage** to save the user's investment array to the device's local storage. On app start, the InvestmentContext would read from AsyncStorage to re-populate the state, making the portfolio persistent.
- **Unsolved - User Authentication:**
    - **Current State:** The app is single-user and anonymous.
    - **Future Solution:** Integrate a service like **Firebase Authentication** or Supabase to allow users to sign up and log in. Each user's portfolio would then be saved to a cloud database (like Firestore) associated with their user ID.