TracStock App: Technical Documentation

This document provides a concise overview of the TracStock Android application, detailing its architecture, key features, implementation practices, and data flow.

1. Overview

TracStock is an Android application designed as a stock/ETFs broking platform. It provides users with functionalities to explore top-performing stocks, search for specific tickers, view detailed product information with historical charts, and manage personalized watchlists.

Key Requirements (from PDF):

- **Explore Screen:** Displays Top Gainers and Top Losers in a grid of cards.
- **Watchlist Screen:** Manages multiple watchlists, showing an empty state if none are present.
- **Product Screen (Details):** Shows basic stock information, a line graph of prices, and allows adding/removing stocks from watchlists. The "Add to Watchlist" button dynamically changes based on watchlist status.
- **Add to Watchlist Popup:** Allows creating new watchlists or adding stocks to existing ones.
- **View All Screen:** Displays a paginated list of stocks from specific sections (e.g., all top gainers).
- **Data API:** Utilizes Alpha Vantage API for all data (TOP_GAINERS_LOSERS, OVERVIEW, TIME_SERIES_INTRADAY, SYMBOL_SEARCH).
- **Technical Practices:** API Key management, Loading/Error/Empty states, well-defined folder structure, third-party library for graphs (MPAndroidChart), **caching API responses with expiration (60 minutes)**, Dependency Injection (Hilt), Kotlin-first development.

2. Architecture and Practices

The application adheres to modern Android development best practices, primarily following the **MVVM (Model-View-ViewModel)** architectural pattern. It emphasizes separation of concerns, testability, and maintainability.

- **MVVM:**
    - **View (Fragments/Activities):** Responsible for observing LiveData/StateFlow from ViewModels and updating the UI. It handles user interactions and delegates events to the ViewModel.

    - **ViewModel:** Holds and manages UI-related data in a lifecycle-conscious way. It exposes data to the View and handles business logic, interacting with Use Cases.
    - **Model (Repository, Use Cases, Data Sources):** Encapsulates business logic and data operations.
        - **Use Cases (Interactors):** Contain specific business logic operations (e.g., GetCompanyOverview, AddStockToWatchlist). They act as intermediaries between ViewModels and Repositories.
        - **Repository:** Abstracts data sources (network, local database). It decides whether to fetch data from the network or local cache.
        - **Data Sources (API Service, Room DAOs):** Directly interact with external data (Alpha Vantage API via Retrofit) or local persistence (Room Database).

- **Dependency Injection (Hilt):** Hilt is used to manage dependencies, providing a robust, scalable, and testable codebase. It simplifies constructor injection and provides application-wide singletons.
- **Asynchronous Operations (Kotlin Coroutines & Flow):** All network and database operations are performed asynchronously using Kotlin Coroutines. Flow is extensively used in the data layer for reactive streams of data, especially for caching and real-time updates. LiveData is used in ViewModels to expose data to the UI.
- **Data Persistence (Room Database):** Room Persistence Library is used for local caching of API responses (Top Gainers/Losers, Company Overview, Historical Data) and for managing watchlists.
- **Networking (Retrofit & OkHttp):** Retrofit is used for making API calls to Alpha Vantage.
- **JSON Parsing (Gson):** Gson is used for serializing and deserializing JSON data from API responses.
- **Charting (MPAndroidChart):** A third-party library is integrated for rendering interactive line graphs on the Product Detail screen.
- **Error, Loading, Empty States:** A Resource sealed class is used across the data and domain layers to represent Loading, Success, and Error states, enabling robust UI feedback.

3. Feature Implementation & Data Flow

3.1. Explore Screen (Top Gainers & Top Losers)

- **Purpose:** Displays a curated list of top-performing and worst-performing stocks.
- **Implementation:**
    - **UI:** ExploreFragment contains two RecyclerViews (rvTopGainers, rvTopLosers) configured with a GridLayoutManager to display stock items in a grid using item_stock.xml.
    - **Data Fetching:** ExploreViewModel uses getTopGainers() and getTopLosers() use cases.
    - **Caching:**
        - StockRepositoryImpl uses a fetchDataAndCache helper function.
        - It first checks StockCacheDao for cached "top_gainers" or "top_losers" data.
        - If valid cache exists (within 5 minutes), it's returned. If expired, it's returned with a "stale" message, and a network refresh is attempted in the background.
        - If no cache or network fails, appropriate error/loading states are managed.
        - Successful network responses are cached in StockCacheEntity with a timestamp.

3.2. Search Functionality

- **Purpose:** Allows users to search for specific stock tickers.
- **Implementation:**

- **UI:** ExploreFragment has a TextInputEditText for search input and a RecyclerView (searchResultsRecyclerView) for displaying results using item_search_stock.xml and SearchStockAdapter.
- **Search Logic:** ExploreViewModel has onSearchQueryChanged which debounces input and calls searchStocks use case.
- **API Call:** StockRepositoryImpl.searchStocks() directly calls AlphaVantageApiService.searchSymbol().
- **Data Flow:** User input in SearchEditText -> ExploreFragment (TextWatcher) -> ExploreViewModel.onSearchQueryChanged (debounced) -> SearchStocks use case -> StockRepositoryImpl.searchStocks -> AlphaVantageApiService.searchSymbol -> Resource<List<Stock>> emitted back to ViewModel -> View updates SearchStockAdapter.

3.3. Product Screen (Details & Chart)

- **Purpose:** Displays detailed information about a selected stock, including a price chart and watchlist management.
- **Implementation:**
    - **UI:** ProductDetailFragment displays company overview details and a LineChart (MPAndroidChart).
    - **Data Fetching:** ProductDetailViewModel uses GetCompanyOverview and GetHistoricalData use cases.
    - **Caching (CompanyOverview & HistoricalData):**
        - StockRepositoryImpl.getCompanyOverview():
            - Checks CompanyOverviewDao for cached CompanyOverviewCacheEntity.
            - If valid (within 60 minutes), returns cached data.
            - If expired or not found, fetches from ApiService.
            - On successful network fetch, saves the raw DTO to CompanyOverviewCacheEntity.
            - Handles network/API errors with fallback to expired cache if available.
        - StockRepositoryImpl.getHistoricalDailyAdjusted():
            - Checks HistoricalDataDao for cached HistoricalDataCacheEntity.
            - If valid (within 60 minutes), returns cached data.
            - If expired or not found, fetches from AlphaVantageApiService using TIME_SERIES_INTRADAY with a 60min interval.
            - On successful network fetch, saves the raw DTO to HistoricalDataCacheEntity.
            - Handles errors with fallback to expired cache.
    - **Chart X-Axis:** A custom DateAxisValueFormatter is used with MPAndroidChart. It displays the dates from the responses as the endpoint returns the data of multiple dates.

- **Data Flow:** View (ProductDetailFragment) observes ProductDetailViewModel.companyOverview and historicalData -> ProductDetailViewModel calls GetCompanyOverview and GetHistoricalData use cases -> StockRepositoryImpl (checks CompanyOverviewDao/HistoricalDataDao -> calls AlphaVantageApiService if needed -> caches in CompanyOverviewCacheEntity/HistoricalDataCacheEntity -> **updates StockEntity name for Company Overview**) -> Resource<CompanyOverview/List<HistoricalPrice>> emitted back to ViewModel -> View updates UI and chart.

3.4. Watchlist Management

- **Purpose:** Allows users to create, view, add stocks to, and remove stocks from personalized watchlists.
- **Implementation:**
    - **UI:**
        - WatchlistFragment: Displays a list of watchlists.
        - WatchlistDetailFragment: Shows stocks within a specific watchlist.
        - AddWatchlistDialogFragment: A dialog for creating new watchlists or adding stocks to existing ones.
    - **Persistence:** Watchlists and Watchlist items are stored in the Room database using WatchlistEntity and WatchlistItemEntity via WatchlistDao.
    - **Add/Remove Flow:**
        - **Add:** From ProductDetailFragment, clicking "Add to Watchlist" navigates to AddWatchlistDialogFragment. This dialog allows selecting an existing watchlist or creating a new one. WatchlistViewModel.onAddStockToWatchlist() handles the database insertion. Upon successful addition, AddWatchlistDialogFragment sets a savedStateHandle result ("stockAddedToWatchlist": true) back to ProductDetailFragment.
        - **Button State Update (Add):** ProductDetailFragment observes this savedStateHandle result. When true, it calls ProductDetailViewModel.onStockAddedToWatchlist(), which in turn triggers checkWatchlistStatus(). This updates _isStockInAnyWatchlist LiveData, changing the button to "Remove from Watchlist".

        - **Remove:** From ProductDetailFragment, clicking "Remove from Watchlist" calls ProductDetailViewModel.removeStockFromAnyWatchlist(). This function iterates through all watchlists to find and remove the stock.
        - **Button State Update (Remove):** After removal, ProductDetailViewModel.checkWatchlistStatus() is explicitly called, refreshing _isStockInAnyWatchlist and updating the button to "Add to Watchlist".
- **Data Flow:**
    - **Watchlist List:** View (WatchlistFragment) observes WatchlistViewModel.watchlists (Flow from WatchlistDao) -> WatchlistViewModel -> WatchlistRepositoryImpl -> WatchlistDao.
    - **Add Stock:** ProductDetailFragment -> AddWatchlistDialogFragment -> WatchlistViewModel.onAddStockToWatchlist -> WatchlistRepositoryImpl -> WatchlistDao (insert WatchlistItemEntity).
    - **Remove Stock:** ProductDetailFragment -> ProductDetailViewModel.removeStockFromAnyWatchlist -> WatchlistRepositoryImpl -> WatchlistDao (delete WatchlistItemEntity).
    - **Button State:** AddWatchlistDialogFragment sets savedStateHandle result -> ProductDetailFragment observes -> ProductDetailViewModel.onStockAddedToWatchlist -> ProductDetailViewModel.checkWatchlistStatus -> ProductDetailViewModel.isStockInAnyWatchlist LiveData updates ProductDetailFragment UI.

3.5. View All Screen

- **Purpose:** Displays a comprehensive list of stocks from a specific category (e.g., all Top Gainers).
- **Implementation:**
    - **UI:** ViewAllFragment uses a RecyclerView with a GridLayoutManager (similar to Explore screen) and potentially a loading indicator for pagination.
    - **Data Fetching:** ViewAllViewModel fetches the full list of Top Gainers or Top Losers.

Data Flow: View (ViewAllFragment) observes ViewAllViewModel.stocks -> ViewAllViewModel calls GetTopGainers/GetTopLosers use cases -> StockRepositoryImpl (same caching logic as Explore screen) -> Resource<List<Stock>> emitted back to ViewModel -> View updates StockAdapter.

ADDITIONAL FEATURES –

1. Implemented a secure way to keep the API key secured at the client side. Used a .properties file and included it in .gitignore. Althought the API key is free but It demonstrated the thought of keeping the API Key securely.
2. Swipe to delete Watchlist Items implemented.

4. Conclusion

The TracStock app demonstrates a robust Android application built with modern architecture and best practices. It effectively leverages MVVM, Hilt for dependency injection, Kotlin Coroutines for asynchronous operations, Room for persistent caching with expiration, and Retrofit for network communication. The implementation addresses key requirements, including dynamic UI updates, efficient data handling, and a user-friendly experience.