Crypto Bubbles Web App: Responses and Project Architecture

Yashvardhan

January 21, 2025

Questions and Responses

Q1 Where should we host the application?

The application can be hosted as follows:

- **Frontend:** Host the frontend using **Vercel** or **Netlify** for efficient static file hosting and serverless functions.
- **Backend:** Deploy the backend on platforms like **AWS**, **Render**, or **Heroku**. For an all-in-one approach, consider using **Firebase Hosting** combined with Firestore for the database.

Q2 Which Blockchain API should we use to return the price, market cap, and volume?

I recommend using the **CoinGecko API**, as it offers reliable, real-time data for market cap, price, and volume with generous free-tier limits. Alternative APIs include:

- **CoinMarketCap API** (paid plans for advanced features)
- **CryptoCompare API**
- **Messari API** (detailed metrics but costlier)

Q3 How would the scraper work for https://cookie.fun?

The scraper will:

(a) Inspect the HTML and network calls to understand the data structure.

(b) Use Python libraries like **BeautifulSoup** and **Requests** for static content, or **Selenium/Playwright** for dynamic JavaScript content.

(c) Extract and filter tokens that are at least 13 days old.

(d) Automate scraping at regular intervals using **Cron Jobs** or **Celery**.

Q4 We should implement caching, so we reduce the number of API calls. How would you do it?

To reduce API calls, caching will be implemented as follows:

- **Server-Side Caching:** Use **Redis** or **Node.js LRU Cache** to temporarily store API responses, with a TTL of 15-30 minutes.
- **Client-Side Caching:** Use **React Query** to cache API responses in the frontend.
- **Conditional Requests:** Utilize API headers like `ETag` or `Last-Modified` to avoid unnecessary re-fetching of data.
- **Database Caching:** Store scraped data in **MongoDB/PostgreSQL**, and update it via background tasks.

Q5 I have the Buy notification script ready in Python. Should you integrate it in the code or access it via Webhook/API from the other script?

The Python script should be accessed via a **Webhook/API** for better modularity. Here's why:

- The Python script can remain independent, making it reusable for other services or platforms.
- Easier maintenance and debugging, as updates to the script will not affect the main application.
- Host the script as a service using **FastAPI**, **Flask**, or **Django**, and expose endpoints for fetching notifications.

Project Architecture

Step 1 Frontend:

- Built using **Next.js** for Server-Side Rendering (SSR) and API routes.
- Charts and bubbles implemented with **D3.js** or **Chart.js**.
- Styling with **Tailwind CSS** for responsiveness.
- Includes a modal for detailed token information (price chart, links to TradingView/Dexscreener, market cap, volume, etc.).

Step 2 Backend:

- Developed using **Node.js** with **Express.js**.
- Data caching handled by **Redis**.
- API integration with **CoinGecko** for token data.
- Database: **MongoDB** for storing token data, user subscriptions, and buy notifications.

Step 3 Scraper:

- Built with Python (**BeautifulSoup**, **Requests**, or **Selenium** for dynamic content).
- Periodic scraping automated with **Celery** or **Cron Jobs**.
- Filters tokens to include only those older than 13 days.

Step 4 Notifications:

- The Python buy notification script runs independently as a microservice.
- Notifications are fetched via a Webhook/API and displayed on the right-hand panel.

Step 5 Payment Integration:

- Use **Coinbase Commerce** or **BitPay** for crypto payments.
- Use **Stripe API** for fiat currency subscriptions.

Step 6 Mobile App:

- Develop a Progressive Web App (PWA) for cross-platform usage.
- Optionally, extend functionality into a React Native app for mobile devices.