# Crypto Alert Backend — Detailed Technical Documentation

**Author Name: Arman Shaikh**
**Email: arman.shaikh050607@gmail.com**

## Overview

This document explains the backend of the Real-Time Cryptocurrency Price Monitoring & Alerting system in full detail — functions, data flows, sockets, caching, alerts, and how the frontend interacts with it. The backend responsibilities: - Periodically fetch and cache prices from CoinGecko. - Emit real-time price updates via Socket.io to subscribed clients. - Evaluate user-defined alerts and notify users via Socket.io and email. - Use Redis to cache prices and to hold a watchlist of coins to poll.

## Architecture and Components

High-level components: -

- Express HTTP server (API endpoints for alerts, auth).
- Socket.io running on same server for real-time messages.
- MongoDB to persist Users, Alerts, Triggered Alerts.
- Redis for caching (latest prices) and for storing the `watched:coins` set.
- Price ingestion service that polls CoinGecko in batches.
- Notification service for Socket.io + email (Nodemailer).

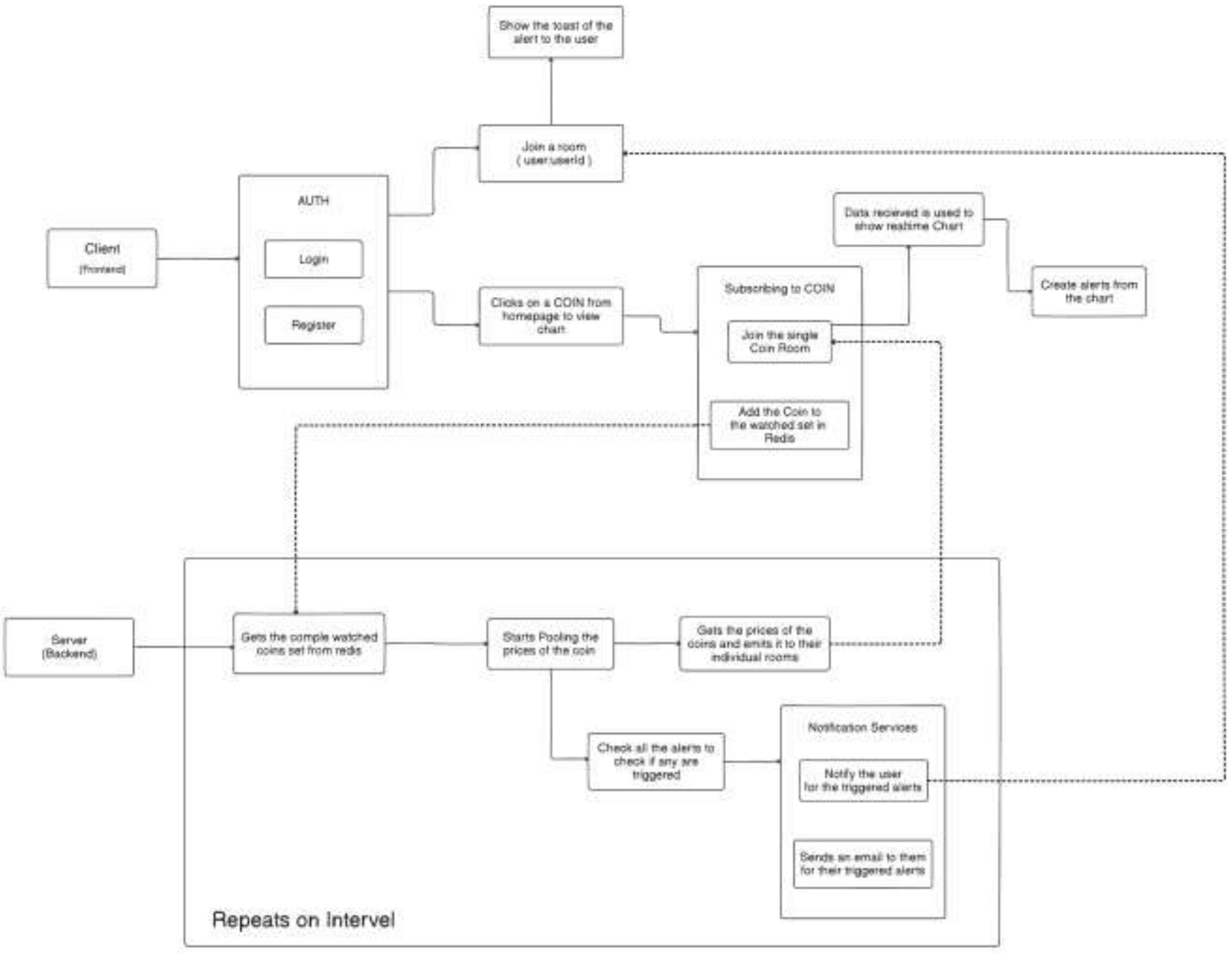
## Environment Variables

Key variables expected (.env):

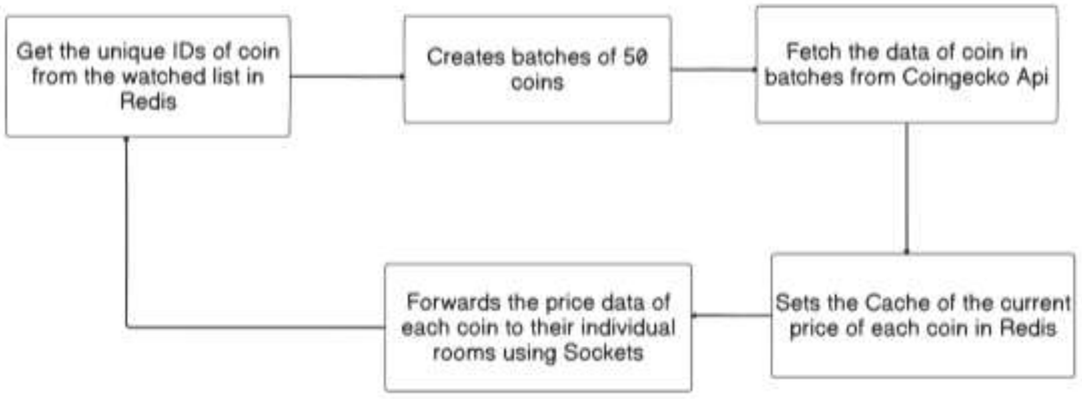
- PORT: HTTP port (e.g., 4000)
- FRONTEND_ROUTE: e.g., http://localhost:5173
- MONGODB_URI: MongoDB connection string
- REDIS_URL: redis://127.0.0.1:6379
- CG_API_BASE: https://api.coingecko.com/api/v3
- CG_API_KEY: optional (pro)
- POLL_INTERVAL_MS: how often to poll (ms)
- EMAIL_HOST / EMAIL_PORT / EMAIL_USER / EMAIL_PASS for Nodemailer

\

## High Level Diagram


Show the toast of the alert to the user
Join a room ( user:userId )
Client (Frontend)
AUTH
Login
Register
Clicks on a COIN from homepage to view chart
Data recieved is used to show realtime Chart
Create alerts from the chart
Subscribing to COIN
Join the single Coin Room
Add the Coin to the watched set in Redis
Server (Backend)
Gets the comple watched coins set from redis
Starts Pooling the prices of the coin
Gets the prices of the coins and emits it to their individual rooms
Check all the alerts to check if any are triggered
Notification Services
Notify the user for the triggered alerts
Sends an email to them for their triggered alerts
Repeats on Intervel

### Price Pooling Mechanism

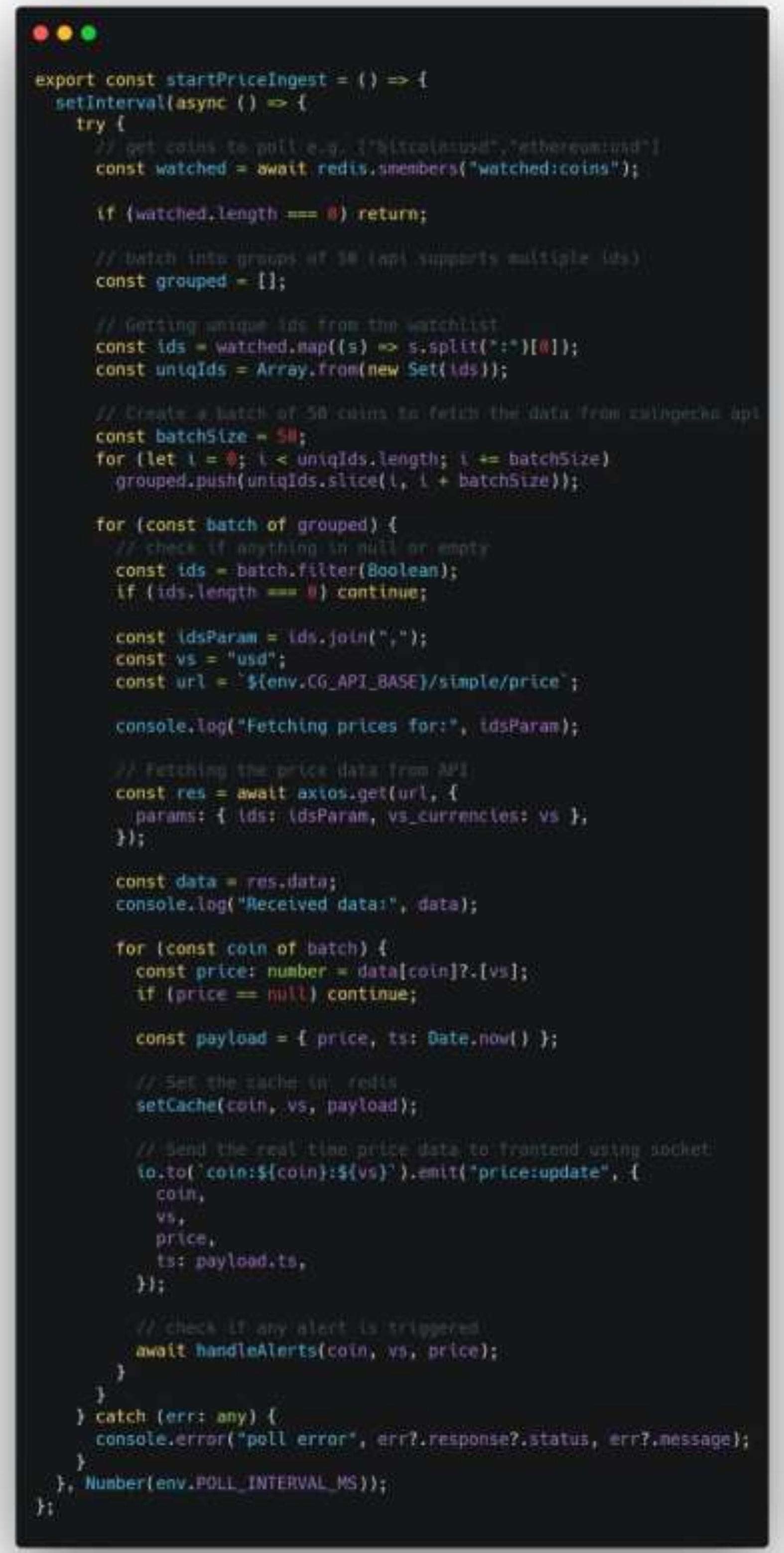

Get the unique IDs of coin from the watched list in Redis
Creates batches of 50 coins
Fetch the data of coin in batches from Coingecko Api
Sets the Cache of the current price of each coin in Redis
Forwards the price data of each coin to their individual rooms using Sockets

# Detailed Function Explanations

## 1) startPriceIngest (priceFetcher.ts)

The startPriceIngest function in priceFetcher.ts is responsible for periodically polling CoinGecko to fetch the latest prices of coins stored in the Redis set watched:coins. It batches coin IDs into groups of up to 50 to reduce the number of API calls. For each price retrieved, it saves the data to Redis using setCache, emits a socket.io update to the corresponding room (coin:{coin}:{vs}), and calls handleAlerts to check if any alerts should be triggered. The function is designed to skip polling if the watched:coins set is empty, remove duplicates and invalid IDs, and gracefully handle API errors by catching them and logging the status and message.

Code Snippet

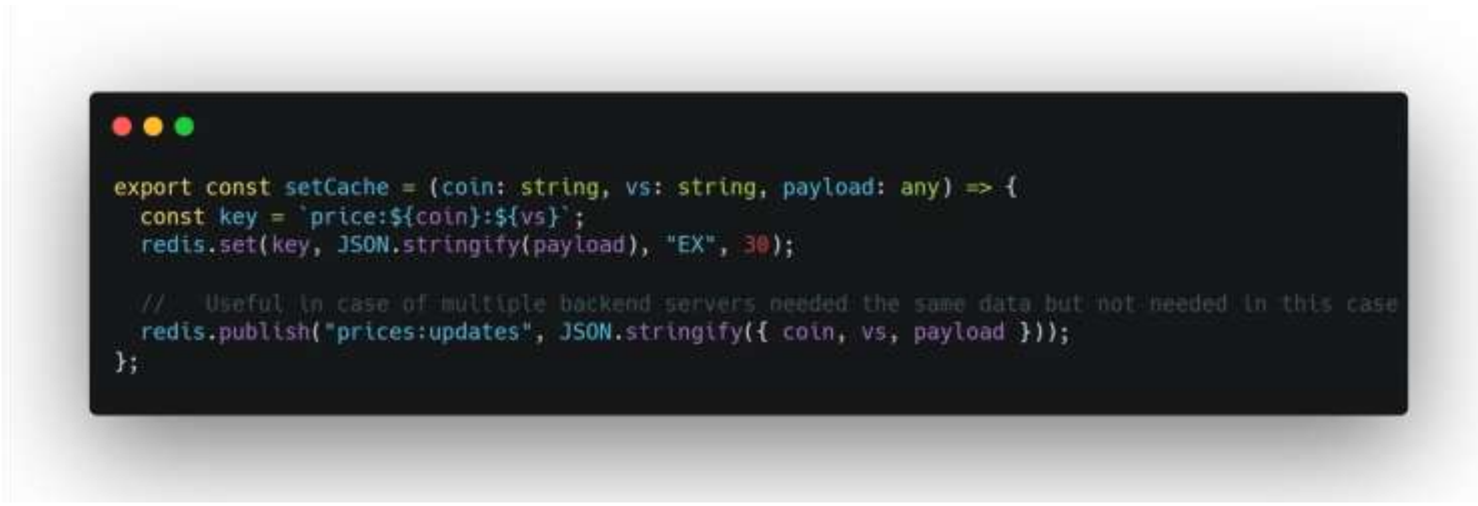


```ts
export const startPriceIngest = () => {
  setInterval(async () => {
    try {
      // get coins to poll e.g. ["bitcoin:usd","ethereum:usd"]
      const watched = await redis.smembers("watched:coins");

      if (watched.length === 0) return;

      // batch into groups of 50 (api supports multiple ids)
      const grouped = [];

      // Getting unique ids from the watchlist
      const ids = watched.map((s) => s.split(":")[0]);
      const uniqIds = Array.from(new Set(ids));

      // Create a batch of 50 coins to fetch the data from coingecko api
      const batchSize = 50;
      for (let i = 0; i < uniqIds.length; i += batchSize)
        grouped.push(uniqIds.slice(i, i + batchSize));

      for (const batch of grouped) {
        // check if anything in null or empty
        const ids = batch.filter(Boolean);
        if (ids.length === 0) continue;

        const idsParam = ids.join(",");
        const vs = "usd";
        const url = `${env.CG_API_BASE}/simple/price`;

        console.log("Fetching prices for:", idsParam);

        // Fetching the price data from API
        const res = await axios.get(url, {
          params: { ids: idsParam, vs_currencies: vs },
        });

        const data = res.data;
        console.log("Received data:", data);

        for (const coin of batch) {
          const price: number = data[coin]?.[vs];
          if (price == null) continue;

          const payload = { price, ts: Date.now() };

          // Set the cache in  redis
          setCache(coin, vs, payload);

          // Send the real time price data to frontend using socket
          io.to(`coin:${coin}:${vs}`).emit("price:update", {
            coin,
            vs,
            price,
            ts: payload.ts,
          });

          // check if any alert is triggered
          await handleAlerts(coin, vs, price);
        }
      }
    } catch (err: any) {
      console.error("poll error", err?.response?.status, err?.message);
    }
  }, Number(env.POLL_INTERVAL_MS));
};
```

## 2) setCache (utils/setCache.ts)

The setCache function in utils/setCache.ts is used to store the latest price data in Redis under a deterministic key format price:{coin}:{vs}, with a short TTL (e.g., 30 seconds) to ensure freshness. Along with saving the data, it also publishes a message to the Redis channel prices:updates containing the coin and its payload. This dual approach serves two purposes: the set operation enables fast reads for API endpoints or as a fallback for the frontend, while the publish operation ensures that other server instances or subscriber services are instantly notified of updates in real time.Code:

Code snippet

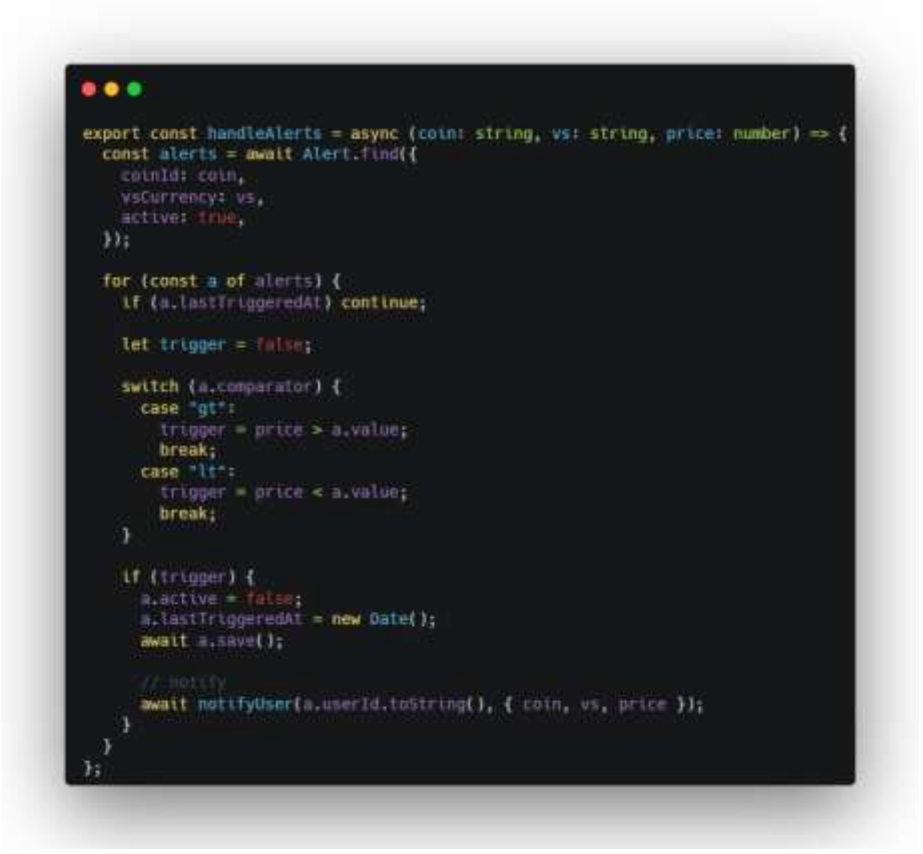


```
export const setCache = (coin: string, vs: string, payload: any) => {
  const key = `price:${coin}:${vs}`;
  redis.set(key, JSON.stringify(payload), "EX", 30);

  //   Useful in case of multiple backend servers needed the same data but not needed in this case
  redis.publish("prices:updates", JSON.stringify({ coin, vs, payload }));
};
```

## 3) handleAlerts (services/handleAlerts.ts)

The handleAlerts function in services/handleAlerts.ts is responsible for checking and processing active alerts for a given coin and vs pair. It queries MongoDB for alerts that match the coin, applies cooldown checks to avoid immediate re-triggering, and compares the current price against the alert's comparator (such as gt, gte, lt, or lte). If an alert is triggered, it either marks the alert as inactive or updates its lastTriggeredAt, optionally persists a TriggeredAlert record, and then calls notifyUser to inform the user. To ensure stability, the function always validates that alert.userId exists before attempting to notify, and it can also use a stored referencePrice when handling percent-based alerts.

Code Snippet

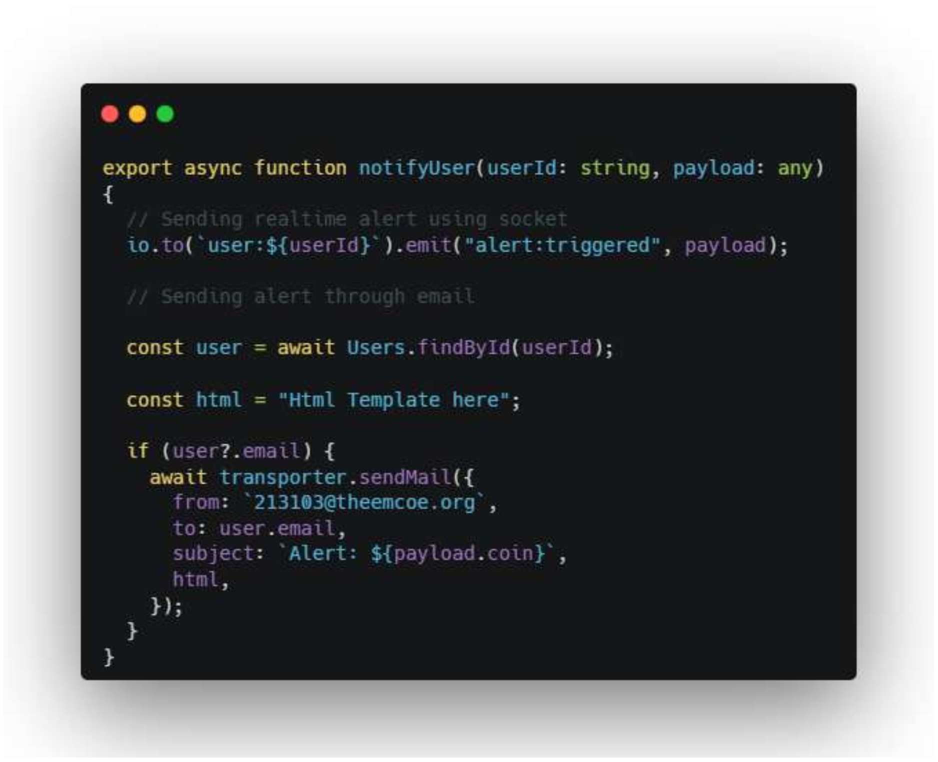


```
export const handleAlerts = async (coin: string, vs: string, price: number) => {
  const alerts = await Alert.find({
    coinId: coin,
    vsCurrency: vs,
    active: true,
  });

  for (const a of alerts) {
    if (a.lastTriggeredAt) continue;

    let trigger = false;

    switch (a.comparator) {
      case "gt":
        trigger = price > a.value;
        break;
      case "lt":
        trigger = price < a.value;
        break;
    }

    if (trigger) {
      a.active = false;
      a.lastTriggeredAt = new Date();
      await a.save();

      // notify
      await notifyUser(a.userId.toString(), { coin, vs, price });
    }
  }
};
```

## 4) notifyUser (services/notifications.ts)

The notifyUser function in services/notifications.ts is designed to deliver alerts and updates directly to users. It emits a Socket.io event to the user's personal room (user:{userId}) so that notifications appear in real time on the frontend, and also sends a formatted HTML email using Nodemailer to the user's email address, if one is available. To work correctly, the frontend socket must join the user:{userId} room (via socket.emit('auth', userId)) after login. The function also includes safety checks to ensure that if the userId is missing, the notification process is skipped to prevent runtime errors.

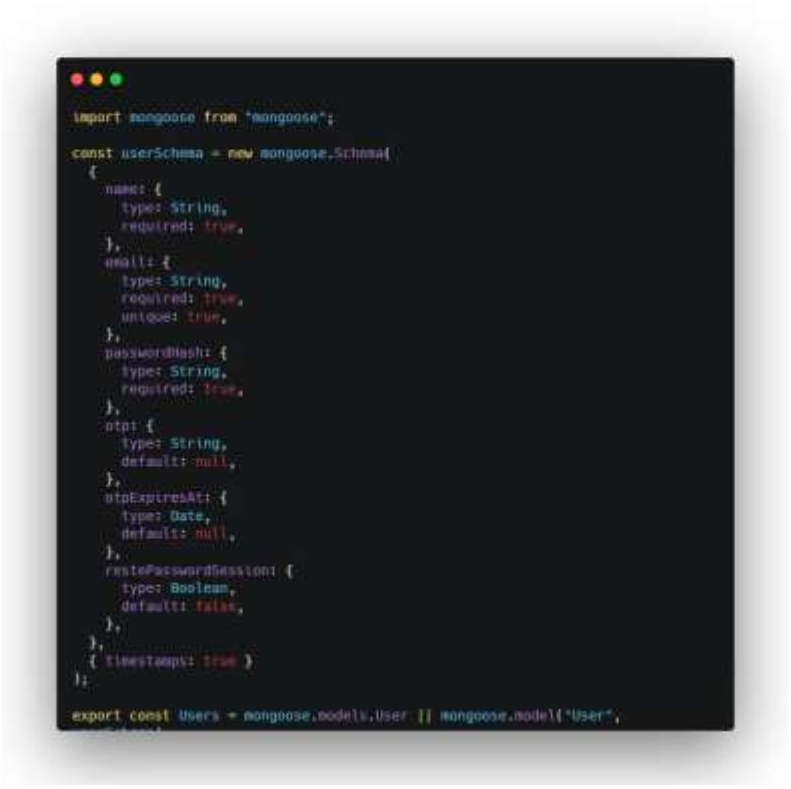
Code snippet



```
export async function notifyUser(userId: string, payload: any)
{
  // Sending realtime alert using socket
  io.to(`user:${userId}`).emit("alert:triggered", payload);

  // Sending alert through email

  const user = await Users.findById(userId);

  const html = "Html Template here";

  if (user?.email) {
    await transporter.sendMail({
      from: `213103@theemcoe.org`,
      to: user.email,
      subject: `Alert: ${payload.coin}`,
      html,
    });
  }
}
```

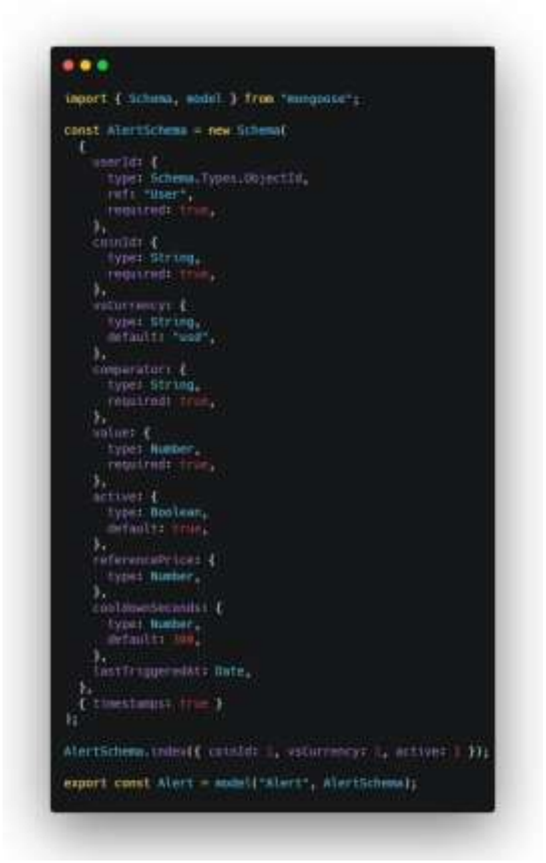
## Data Models & Redis Keys

1. User Model (storing the details of the user)



```
import mongoose from "mongoose";

const userSchema = new mongoose.Schema(
  {
    name: {
      type: String,
      required: true,
    },
    email: {
      type: String,
      required: true,
      unique: true,
    },
    passwordHash: {
      type: String,
      required: true,
    },
    otp: {
      type: String,
      default: null,
    },
    otpExpiresAt: {
      type: Date,
      default: null,
    },
    restePasswordSession: {
      type: Boolean,
      default: false,
    },
  },
  { timestamps: true }
);

export const Users = mongoose.models.User || mongoose.model("User",
```

2. Alerts Model (storing the details of Alert)



```
import { Schema, model } from "mongoose";

const AlertSchema = new Schema(
  {
    userId: {
      type: Schema.Types.ObjectId,
      ref: "User",
      required: true,
    },
    coinId: {
      type: String,
      required: true,
    },
    vsCurrency: {
      type: String,
      default: "usd",
    },
    comparator: {
      type: String,
      required: true,
    },
    value: {
      type: Number,
      required: true,
    },
    active: {
      type: Boolean,
      default: true,
    },
    referencePrice: {
      type: Number,
    },
    cooldownSeconds: {
      type: Number,
      default: 300,
    },
    lastTriggeredAt: Date,
  },
  { timestamps: true }
);

AlertSchema.index({ coinId: 1, vsCurrency: 1, active: 1 });

export const Alert = model("Alert", AlertSchema);
```

## Socket.io Rooms & Events

**Rooms**

- **coin:{coin}:{vs}** → Streams live price updates for a specific coin and vs currency.
- **user:{userId}** → Sends user-specific notifications, such as triggered alerts.

**Client Side Flow ( After connecting to the server )**

- socket.emit('subscribe', { coinId, vs }) → Join a coin price room.
- socket.emit('auth', userId) → Join the user's personal room after login.

**Server Side Usage**

- io.to('coin:bitcoin:usd').emit('price:update', payload) → Broadcasts a price update for Bitcoin in USD.
- io.to('user:123').emit('alert:triggered', payload) → Sends a triggered alert notification to user with ID 123.

## End-to-End Flow (Step-by-step)

This flow explains what happens from a user's action to a real-time alert being delivered.

**1. User Action (Frontend)**

- User clicks on a coin in the frontend.
- The frontend navigates to the coin page and calls **subscribeToCoin(coinId, "usd")**.
- A socket event is emitted: **socket.emit("subscribe", { coinId, vs: "usd" })**.
- The server adds the client to the corresponding **coin room**.

**2. Backend Watchlist**

- The frontend or API adds the coin to the Redis set: **watched:coins (SADD)**.

**3. Price Poller**

- **startPriceIngest** runs periodically every **POLL_INTERVAL_MS**.
- It reads the Redis set **watched:coins**, batches the coin IDs, and queries **CoinGecko /simple/price API**.

**4. Cache & Emit**

- For each returned price:
  - **setCache** stores the price in Redis under the key **price:{coin}:{vs}** with a TTL.
  - Publishes the update to the Redis channel **prices:updates**.
  - Emits a Socket.io event to the corresponding coin room: **io.to("coin:{coin}:{vs}").emit("price:update", { price, ts })**.

**5. Frontend Update**

- Any client subscribed to that room receives the **price:update** event.
- The UI or chart updates in real time.

**6. Alert Evaluation**

- **handleAlerts** checks MongoDB for active alerts related to the coin.
- If the alert condition matches:
  - Marks the alert as inactive (or updates **lastTriggeredAt**).
  - Calls **notifyUser**.

**7. Notify User**

- Emits a Socket.io event to the user's personal room: **io.to("user:{userId}").emit("alert:triggered", payload)**.
- Sends an HTML email notification to the user via **Nodemailer**.

**8. Frontend Notification**

- The frontend receives the **alert:triggered** event.
- A toast notification is displayed, or the user is redirected to the relevant coin page.

## Error Handling & Rate Limits

### Common Errors and Remedies

- **400 Bad Request**
  - Usually caused by a malformed CoinGecko query (e.g., empty IDs).
  - Ensure entries in **watched:coins** are valid and filter out empty strings.
- **401 Unauthorized**
  - If using the Pro API, check the API key header.
  - For the free API, remove any headers related to authentication.
- **429 Too Many Requests**
  - Implement exponential backoff.
  - Increase **POLL_INTERVAL_MS** to reduce request frequency.
  - Batch multiple coin IDs per request to minimize API calls.
- **CORS Errors**
  - Make sure Socket.io and Express CORS settings allow credentials and the correct origin.
- **Missing userId**
  - Always validate alerts for a **userId** before sending notifications to prevent runtime errors.

### Rate Limit Strategy

- Batch multiple coin IDs into a single /simple/price call to reduce API requests.
- Poll only the coins currently present in **watched:coins**.
- Increase **POLL_INTERVAL_MS** (e.g., 15,000 ms) when using the free API for safety.
- On receiving a **429 Too Many Requests**, back off for 30–60 seconds before retrying.

## Frontend Integration Notes

- **Single Socket Instance**
  - Initialize a single socket instance and reuse it throughout the application.
- **User Authentication**
  - After login, emit:
    **socket.emit('auth', userId)**
    once the socket is connected to join the user's personal room.
- **Subscribe to Coin Rooms**
  - To receive live price updates, emit:
    **socket.emit('subscribe', { coinId, vs })**
- **Displaying Alerts**
  - Use a toast library (e.g., **react-hot-toast**) to display alerts received from **alert:triggered** events.
- **Component Mounting Consideration**
  - If navigation causes toast notifications to stop working, ensure the component responsible for joining the user room is mounted globally (e.g., at the top-level **App** component).

## Testing & Debugging Checklist

- Verify server is running and listening on PORT
- Check Redis: SMEMBERS watched:coins contains expected entries
- Test CoinGecko query via curl: /simple/price?ids=bitcoin,ethereum&vs;_currencies=usd
- Verify Redis keys: GET price:bitcoin:usd returns JSON
- Check socket rooms: On connect, server logs join for coin/user room
- Trigger a fake alert emit from server and verify frontend toast
- Monitor logs for 429/401/400 HTTP responses from CoinGecko

## API Endpoints

Auth Endpoints:

Base URL: **/api/v1/auth**
Endpoints

- **Register a New User**
  **POST** /register
  - Registers a new user with the provided details.
- **Login User**
  **POST** /login
  - Authenticates a user and returns session/token.
- **Send OTP for Password Reset**
  **POST** /send-otp
  - Sends a one-time password (OTP) to the user’s email for password reset.
- **Verify OTP**
  **POST** /verify-otp
  - Verifies the OTP sent to the user.
- **Reset Password**
  **POST** /reset-password
  - Resets the user’s password after OTP verification.
- **Logout User**
  **GET** /logout
  - Logs out the authenticated user and clears session/token.

Alert Endpoints

Base URL: **/api/v1/alerts**
Endpoints

- **Create a New Alert**
  **POST** /
  - Creates a new alert for the authenticated user.
  - **Middleware:** useAuth (user must be logged in).
- **Get All Alerts for a User**
  **GET** /user/alerts
  - Retrieves all alerts belonging to the authenticated user.
  - **Middleware:** useAuth.
- **Watch Coins**
  **POST** /coins/watch
  - Adds coins to the user’s watchlist (stored in Redis watched:coins).