CRYPTWIRE SYSTEM DESIGN

cryptwire.net

1 INTRODUCTION

1.1 Purpose and Scope

This document provides an in-depth breakdown of how Cryptwire was built and how it works to provide analytical services to its users. It is not an exhaustive document with regards to every detail of the system as you'd find in other system design documents but rather aimed at providing a solid understanding of the architecture of the platform.

1.2 Summary

Cryptwire is an analytics and optimisation platform for cryptocurrency traders using a Cryptohopper trading bot. It provides various charts, performance history over time as well as live candle charts of your entire portfolio.

1.2.1 System Overview

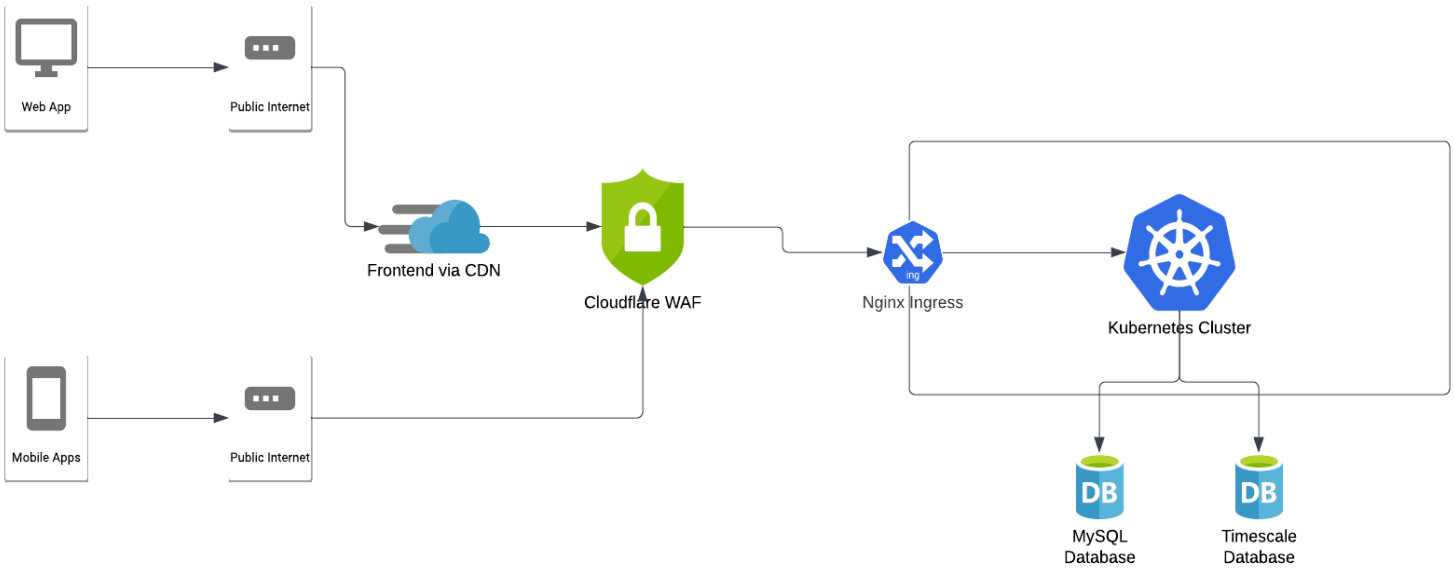

Figure 1: High-level overview

The platform is built on top of Kubernetes with a front end built in Angular using the Ionic Framework that allows the same codebase to be used for web, mobile and desktop apps.

Information is ingested into the system from Cryptohopper, CoinMarketCap, cryptocurrency exchanges and exchange rate services. Once the data is ingested it is transformed into actionable insights to traders.

The primary requirement of the platform is to provide more in-depth analytics to the trader in order to make better decisions regarding their bot setup. Further, the platform provides ways to optimise your trading and protect your balance from sudden dips in the market.

1.2.2 Design Constraints

Cryptwire has no direct access to a user's exchange account, it only has access to the user's Cryptohopper account and is limited with what can be retrieved via their API. Even though Cryptohoppper's API does return the most important information, it is not real-time but rather updated on a schedule ranging from 60 seconds to 10 minutes – depending on your Cryptohopper subscription level.

It should be noted that we have requested various upgrades and extensions to the Cryptohopper API and they have all been implemented by their team.

1.2.3 Future Contingencies

We do have a great working relationship with Cryptohopper and have operated within the requirements for their API. However, we are still at risk should they decide to cut us off from their API permanently or should their API be discontinued in future.

If either of these should happen in future, we'll integrate directly with the cryptocurrency exchanges we support and extract the information from their platforms. This will ensure we can continue to provide the same level of service that users have come to expect of us.

1.3 Points of Contact

All question, queries or other requests may be directed at Donovan Solms, Founder, ds@donovansolms.com

2 SYSTEM ARCHITECTURE

2.1 System Hardware Architecture

Cryptwire is built to be cloud-native and as such has no requirements with regards to specialised hardware. The minimum hardware required to run the service in high-availability mode with low usage is:

A Kubernetes cluster with 3 Linux nodes with at least 8GB of RAM, 128GB SSD and 4-core CPU.

2.2 System Software Architecture

Cryptwire follows a microservices architecture with a multitude of small services working together or individually to accomplish tasks. A vast majority of these services are written in Go with the front end and primary API being the exceptions, written in Angular and Nodejs respectively.

The diagram below shows a general layout of the services and how they interact.

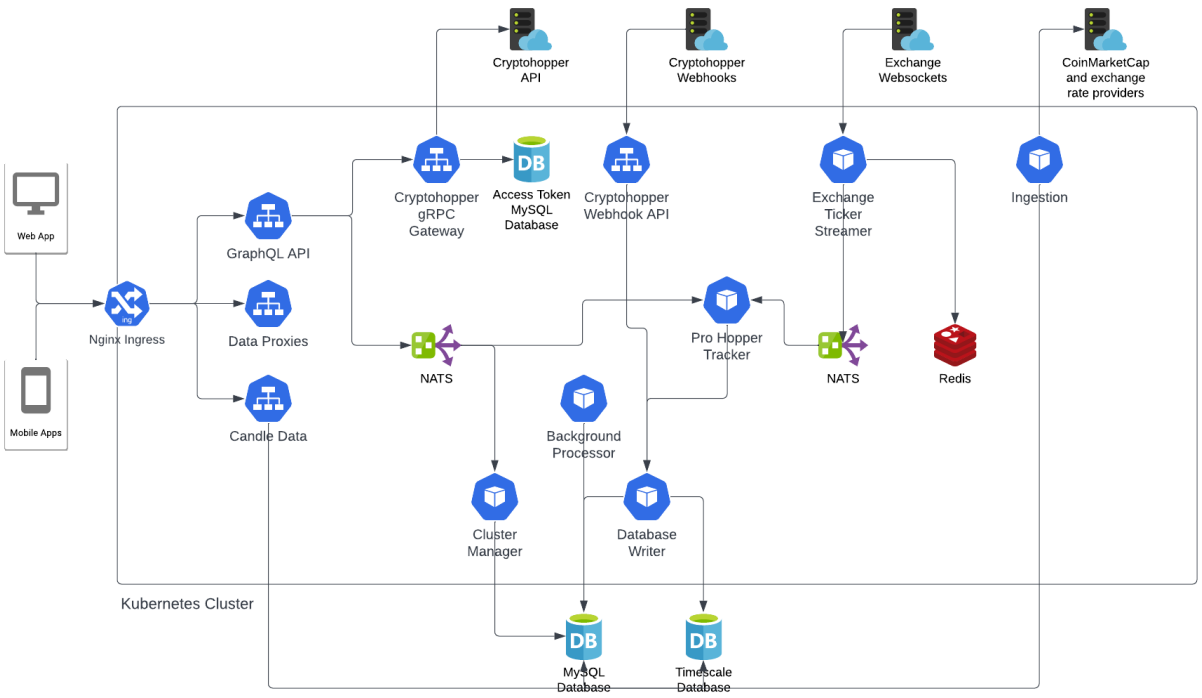

Figure 2: General cluster overview

Front end

Angular, Ionic, TypeScript

The front end is written in TypeScript using Angular and the Ionic Framework. It is hosted on a CDN with additional Cloudflare caching enabled. The Ionic Framework allows us to use the exact same codebase for web, mobile and desktop apps. Some notable parts of the front end are highlighted below.

Authentication and authorisation

Once authenticated the GraphQL API issues the front end with a JSON Web Token (JWT) containing permission information as well as the user ID and session ID. We make use of the CASL authorisation library that allows us to use the same permission structure and validation across back end and front end services.

Within templates and Angular services you can use the simple CASL check
`this.ability.can("view", "funds-usage")`
This provides a powerful, yet simple, approach to limiting access to various front end sections.

Datafeeds

We implement TradingView's Charting Library for all candlestick charts, including your portfolio overview, dashboard, and trading history. The TradingView Charting Library is a top-tier charting tool, however, you need to bring your own candle data – TradingView does not supply it.

To realise our goal of real-time and historic charts we implemented the Datafeed API provided by TradingView for each exchange we support. We further implemented it for our own candle data service to provide real-time candlestick charts of your portfolio.

Formatting pipes

In providing a user-friendly experience we implemented several formatting pipes that are influenced by the user's preferences.

- Currency formatting – When signing up for Cryptwire you specify your local currency and have the option of displaying crypto values in your local currency
- Cryptocurrency formatting – While most users are happy with the default amount of decimals being displayed, they do have the option to select a different amount of decimals for stablecoins and other cryptocurrencies
- Balance masking – Many users like to share their trades on social media and Discord, by allowing them to mask their balance and trade sizes, they can share the information without the concern of other users knowing their financial situation

Dashboard widgets

To create a truly custom experience for our Pro users we implemented 14 different dashboard widgets ranging from candlesticks to top trades and fund usage charts. Each of these widgets can be customised in size and data to fit in with the user's preference.

GraphQL API

Nodejs, Apollo, Hapi, TypeScript

The primary API is built with TypeScript using Nodejs. Hapi is used as the webserver library with Apollo handling GraphQL queries. Logging is managed by Winston, nconf handles configuration and Sequelize was chosen as the ORM. We utilise GraphQL queries, mutations as well as subscriptions. Notable parts of the API are detailed below.

Authentication and authorisation

As noted in the front end section, we use the CASL authorisation library across front- and back end to provide a unified developer experience with regards to permission management.

Within any part of a GraphQL resolver, you may use the following to check a permission

```
currentUser.abilities.can("list", "users")
```

This provides a powerful, yet simple, approach to restricting any action in a granular way.

Background task processing

Long-running tasks are not handled by the API directly, instead, it is handed off to the Background Processor service using RSMQ (Redis Simple Message Queue) to pass on the task that needs to be processed.

Within the context of the API, the only task handed off is the importing of a user's information from Cryptohopper. The process is as follows:

1. A user signs up and connects their Cryptohopper account
2. A task is queued and the user is shown an 'importing' page
3. The user can't continue until the data is imported and is free to leave as we'll notify them via email. If they decide to stay, we provide progress updates to the front end via a GraphQL subscription.
4. Once the import completes, the user is taken to their dashboard.

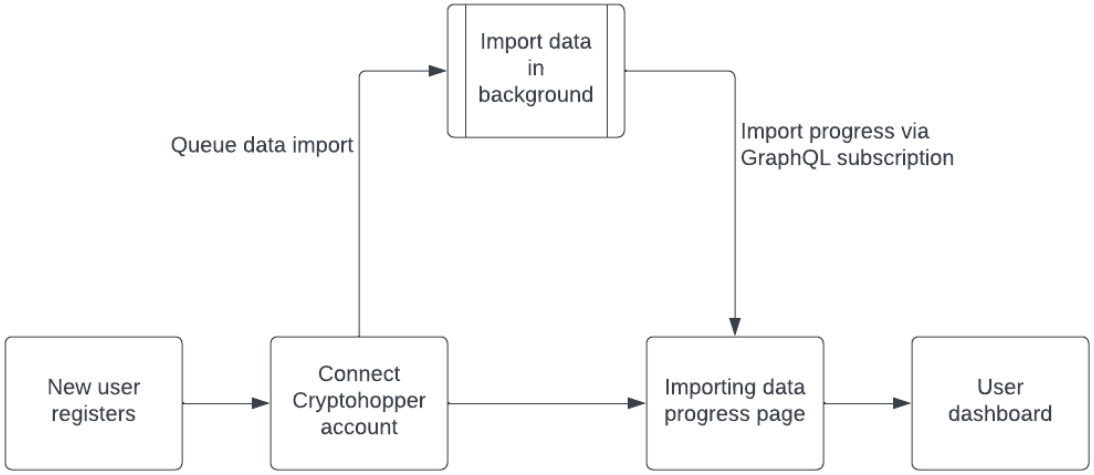

Figure 3: Registration and data import process

gRPC communication

To provide the most accurate information to the user, we update their portfolio balance whenever they log in. This requires a quick API call to Cryptohopper through the Cryptohopper gRPC API Gateway. In this case the gRPC service definition was compiled to JavaScript with TypeScript definitions that allowed us to use it immediately without writing JSON-based API calls.

Deep back end communication

Some of the features provided to Pro users require us to communicate with services running on the user's account which provide no direct communication interface, like an API.

In these cases we can communicate with the service using the NATS Request-Reply pattern. In such a request, the query is routed through NATS to the service listening with the response routed back to the original requesting client. In theory we could run thousands of API endpoints for these services, but NATS provided a more elegant solution.

Timescale Database

Timescale enhances the capabilities of PostgreSQL by implementing various time-series and analytical functions.

We use these features to build up candlestick charts for user portfolios for a variety of timeframes from the base 15-minute interval the data is captured in. Timescale makes it very easy to build up 1 hour, 4 hour, daily or even weekly candles.

These features are further used to calculate trading activity for custom date ranges with custom resolutions. For a granular view, you can query the past week's data in hour resolution or for a higher-level view you can query the past year in month resolution.

Below is a diagram showing the communication between the back end API and other services.

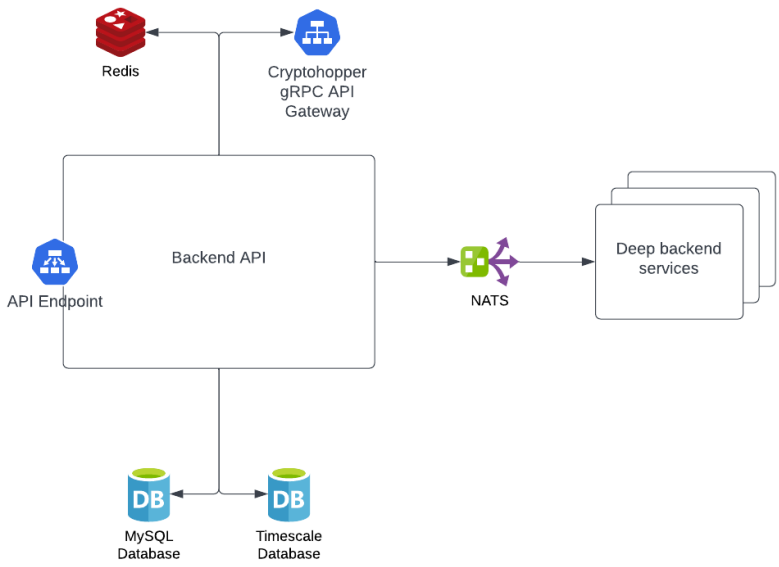

Figure 4: Back end API communication

Cryptohopper gRPC API Gateway

gRPC, Protocol Buffers, Go

To interact with the Cryptohopper API we created a standalone service using Go that can be communicated with over gRPC. gRPC was chosen as the framework as the service definitions can be compiled down to multiple languages providing an instant client library that can be used with little to no issue.

Even though any service could theoretically implement the Cryptohopper API directly, we made a security decision to keep all Cryptohopper access tokens separate from the main infrastructure.

This gateway is the only service that has access to a user's API access tokens and will only inject it at the moment the API call is made. This ensures there is no way to retrieve a user's access token and perform malicious operations on their account. As an additional security layer, the service only implements the specific requests that are used on the platform instead of all the functionality provided, further reducing the attack surface.

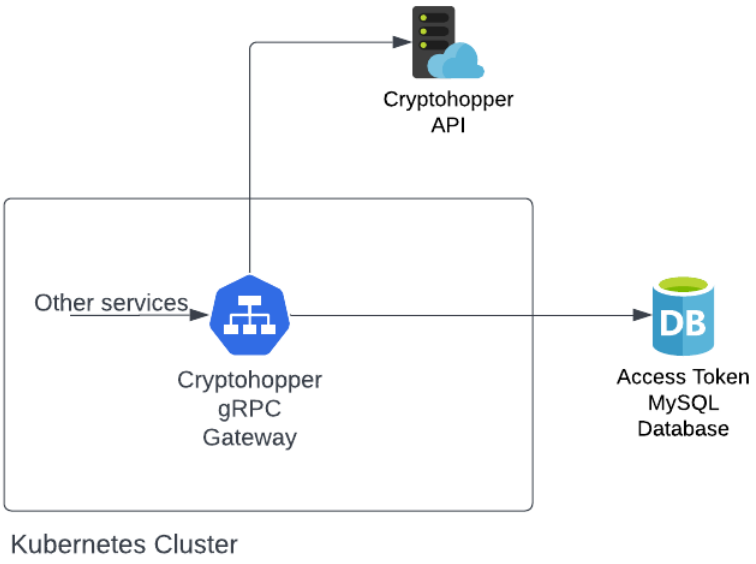

Figure 5: Overview of the gRPC Gateway service

Cryptohopper Webhook API

Fiber HTTP server, Go

Cryptohopper provides real-time webhooks for trades. All buys and sells are captured by the webhook API, logged in the Timescale database for analysis as well as sent to NATS to be distributed to all services listening.

For Pro users, the Hopper Tracker listens for these trades using NATS and they are incorporated into the user's portfolio candles as volume bars.

For all users, trades are logged in Timescale for analysis and review at a later stage.

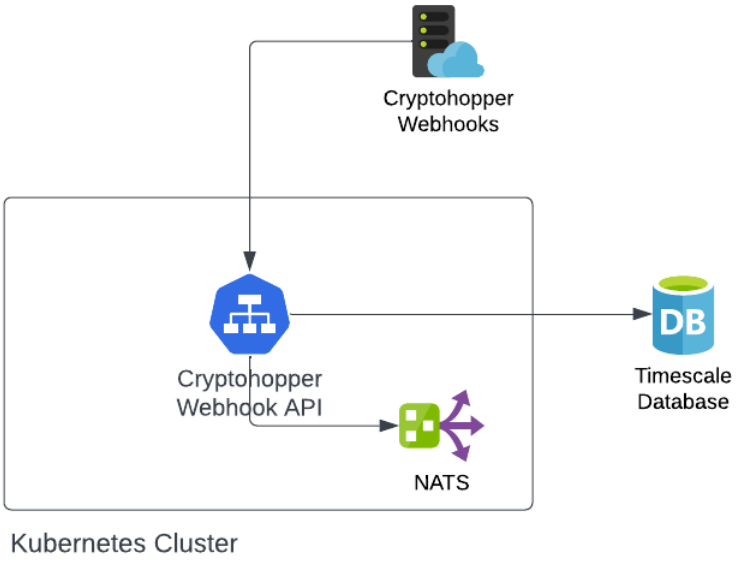

Figure 6: The flow of webhooks

Data Proxy (Exchanges)

Nodejs, off-the-shelf open-source service

To provide a unique trading history experience, where you can see your buys and sell of a specific position on a candlestick chart, we require the OHLC (Open, High, Low and Close) data from the exchange for the given timeframe.

While most exchanges provide that data for free they do not provide Cross-Origin Resource Sharing (CORS) headers to be able to query their data directly from the JavaScript front end. The data proxy for these services are handled by an off-the-shelf component "CORS Anywhere" which is available under the MIT licence at https://github.com/Rob--W/cors-anywhere

Candle Data Service

Gorilla WebSockets, Go

The candle data service implements the required API for the TradingView Charting Library to be able to fetch and display correct candles based on the user's timezone and preferences.

This service only handles the candle information for a user's portfolio values, ie. the candles generated by Cryptwire, and not candles for specific assets such as Bitcoin. It implements two main components:

1. Fetching and constructing candles for a given period using a specified resolution
   For these candles, the Timescale database is queried directly to construct the candles for the request

2. Real-time WebSocket feed for candles in a specified resolution
   For real-time candles, the service listens for information being pushed by the Hopper Tracker over NATS. Using the real-time 15-minute candle information provided by the tracker the service then combines the information with data from Timescale to construct live candles for any resolution

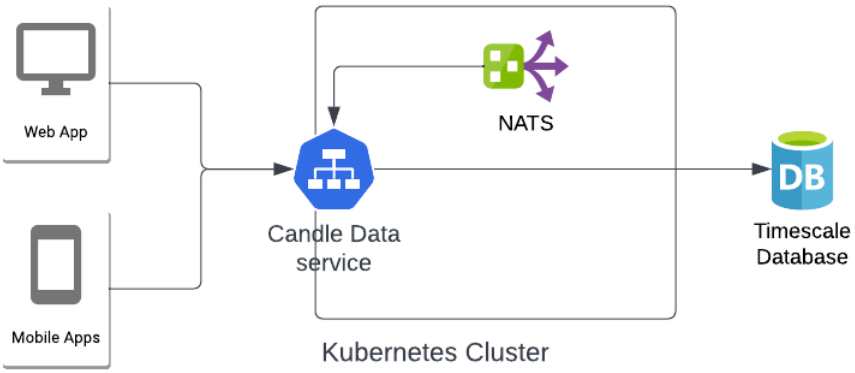

Figure 7: Flow within the candle data service

Exchange Ticker Streamer

Go

The ticker streamer is responsible for caching and providing pricing information for cryptocurrencies. It is built in Go and uses WebSockets to connect with various exchanges for pricing information. If an exchange doesn't support WebSockets, the service falls back to polling all prices at an interval.

Prices are cached in Redis at a specified interval to provide the front end with up-to-date pricing information for calculations.

Prices are also pushed to NATS on predefined subjects. Any service relying on real-time pricing information, such as the Hopper Tracker, can subscribe to the subject and receive the pair prices it is interested in.

As an example, for pairs on Binance using USDT as the quote currency, the prices are pushed to the subject `ticker.binance.usdt.[currency]`. Any service that wishes to receive USDT pair prices can subscribe to `ticker.binance.usdt.*` which will allow it to receive any USDT pair's price. Alternatively, subscribing to `ticker.binance.usdt.btc` allows the service to only listen for BTC/USDT price updates.

Pro subscriber services

Go

Every user on our Pro paid package gets access to more advanced features such as live candle charts of their portfolio, additional charts, as well as automated actions that can be taken on their bots. To enable these features we built additional services that operate on a specific user's account – the Tracker, Analyser and Controller. All running within the same Kubernetes Pod as separate containers.

Below is an overview of how these services interact.

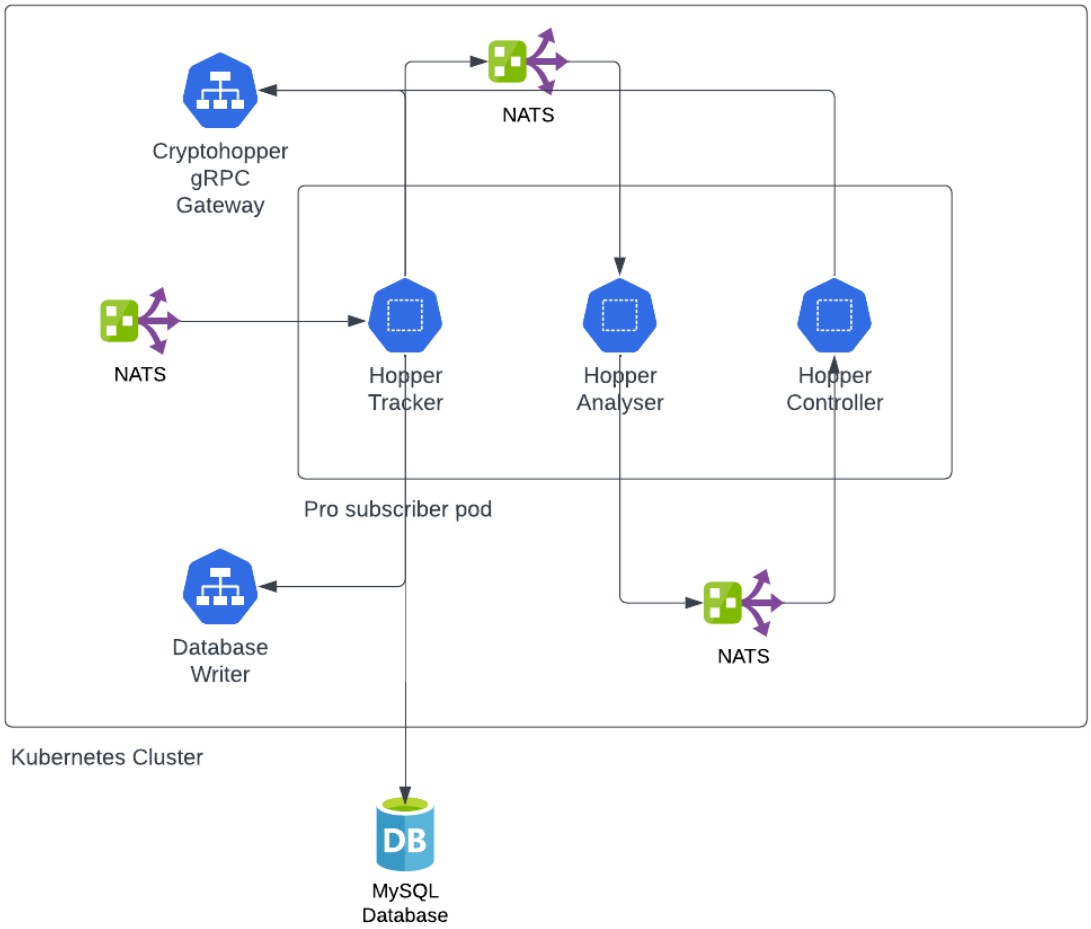

Figure 8: Pod containing Pro subscriber services

Tracker

Go

The Hopper Tracker service is written in Go and handles the tracking of a user's assets, their trades, and pricing of each asset to calculate their exact portfolio value.

Assets are retrieved in regular intervals as well as updated via the information received from webhooks. Trades are only received via the webhook service and asset prices are received via NATS from the exchange ticker streamer services.

The current portfolio value and current 15-minute candle is sent to the Analyser service every two seconds using NATS.

Analyser

Go

The analyser implements various technical analysis (TA) functions. It receives candle and balance information from the Tracker every two seconds. Based on the active TA functions, it calculates the current value for those functions using live and historic candles.

It uses the Techan Go library for the math behind technical analysis formulas. Within the service, all calculations are treated as technical analysis – even the current balance. By treating everything as the result of technical analysis we end up with a unified output format that is sent to the Controller whenever a new value becomes available.

Controller

Go

The Controller is the only service that is capable of executing an action on a user's trading bot. Actions, in terms of a trading bot, include things like panic selling all positions, reducing the amount used in buys, disabling the bot or any other functionality exposed by Cryptohopper.

How these actions are implemented make for a very flexible and extensible framework to work within. The structure is as follows:

The Controller holds a collection of RuleSets. A RuleSet is a collection of Rules. If all Rules within a RuleSet returns true, then all the Actions must be executed in sequence for the RuleSet. The diagram below represents this structure.

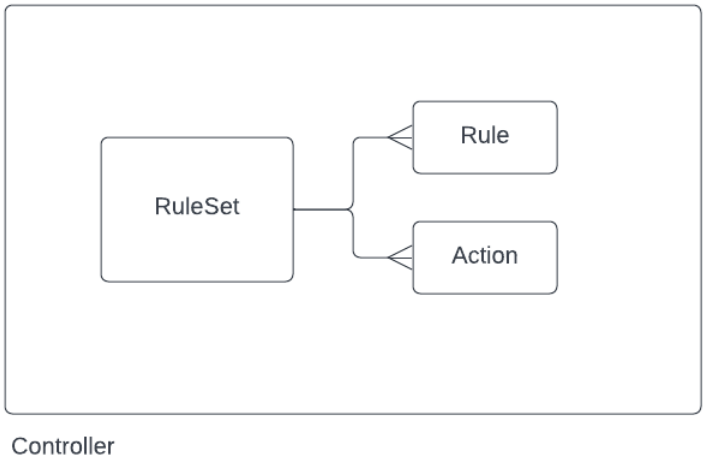

Figure 9: RuleSet, Rule and Action relationship

How is this structure used for the Portfolio Trailing Stop-Loss feature?

A typical trailing stop-loss tracks a position, once the position reaches a certain percentage increase it is armed. Once armed, if the position drop by a certain percentage, the position is sold. The difference between trailing stop-loss and a normal stop-loss is that the trailing version will keep increasing the 'sell at' value as the position's value increases – locking in more profit.

In terms of the Controller, the Portfolio Trailing Stop-Loss works similarly. It uses a RuleSet with a single Rule that implements the logic described above *on the portfolio balance*. When the Rule, and in this case RuleSet, returns true, it will execute the selling of all the bot's open positions.

With this, you can set up any number of Rules with a long sequence of actions to optimise your trading experience.

Cluster Manager

Go

The Cluster Manager handles the addition and removal of Pro subscriber Kubernetes Pods. It is responsible for ensuring that all the bots for all our paying Pro users are being tracked at all times. This includes adding pods for new users that start their Pro trial as well as removing the pods for users whose trial has expired, additionally, if users create or delete bots, the service ensures they are tracked as well.

The manager serves as a management interface to the pods as well, it can receive commands via NATS from an admin user to verify the current running state together with manual addition and removal of pods.

Background Processor

Go

The Background processor is responsible for two things, importing a user's information from Cryptohopper and logging their daily performance stats.

<u>Importing user information</u>

The following was already described in the GraphQL API section and is repeated here for clarity and completeness.

We import existing data from a user's account when they register for Cryptwire. We handle this process in the background as to not lock up the interface for an extended period. The process is as follows:

5. A user signs up and connects their Cryptohopper account
6. A task is queued and the user is shown an 'importing' page

7. The user can't continue until the data is imported and is free to leave as we'll notify them via email. If they decide to stay, we provide progress updates to the front end via a GraphQL subscription.
8. Once the import completes, the user it taken to their dashboard.

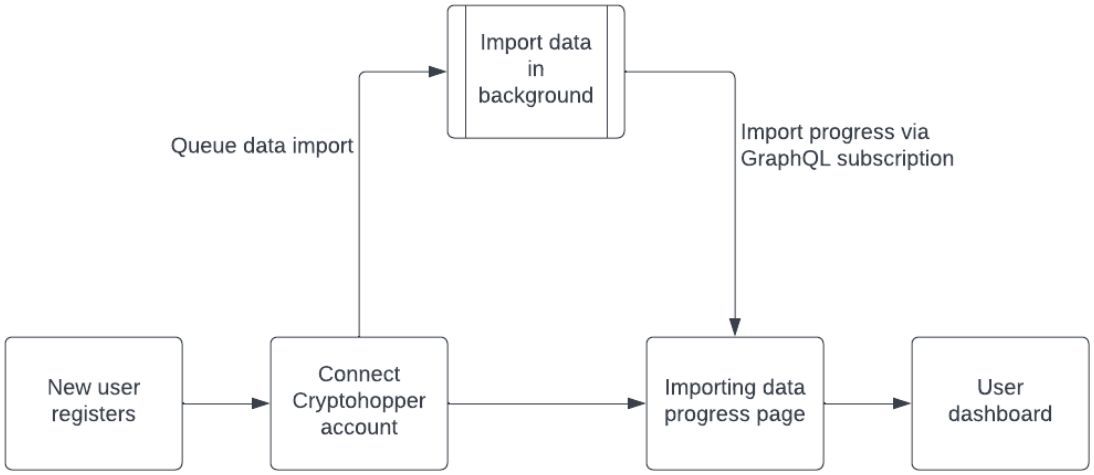

Figure 10: Registration and data import process

Logging performance stats

To understand your bot's performance we log as much as we can at midnight in the user's timezone. This allows us to build charts and insights that make sense to a user with regards to the concept of a 'day'.

During registration you select your timezone and we set your update on a specific schedule in GMT+0. All of our services operate exclusively in GMT+0 and convert the dates and times to a user's local timezone when querying data.

The daily stats import is a simple task, however, it achieves much more. To analyse trading information and provide reliable insights we need information that is captured at the same time, every day.

Technically the schedule is handled by a simple timer that runs every 30 minutes. When it runs, it finds all the users that have been scheduled for the current timeframe and updates their performance stats.

Database Writer

Go

This Go service is responsible for writing custom candle data to the Timescale database. We track thousands of trading bots at any given time and writing candle data for every bot, every 15 minutes, creates a thundering herd problem in terms of database connections.

A typical solution to the thundering herd problem is to add some randomness to the problematic calls, ie. make the call at a random time between 8 and 12 minutes instead of everyone at exactly 10 minutes. This helps spread the load and can, at least up to a point, avoid the herd problem.

However, this type of solution was not an option for candle data. These candles must be captured at exact, or within milliseconds, 15 minute intervals to produce accurate candles. Being off by a minute in these captures will cause Timescale to average out the values and produce incorrect candles.

To solve this we do three things:

1. Write all candles to a Redis list instead of directly to the database
2. Include the exact open and close timestamps for the candle
3. Write all the candle information to the database using only a few threads that don't drastically increase the connection count

By using this technique we lower the resources required by the database server and avoid running out of connections for other parts of the system.

Ingestion services (CoinMarketCap and Exchange rates)

Go

We deploy two services that ingest external information into the system, one for currency exchange rates and another for market cap information of the top 1000 coins and tokens.

1. Exchange rate service
   To provide exchange rate information between a user's local currency and cryptocurrencies we fetch hundreds of rates every hour from OpenExchangeRates
2. CoinMarketCap
   On an hourly basis, we import the market cap and all-time-high data of the top 1000 coins and tokens from CoinMarketCap. This information is used to calculate whether your current positions are well balanced based on market cap and potential

2.3 Internal Communications Architecture

Within the cluster we employ three methods of communication:

1. gRPC
   gRPC is a remote procedure call framework that uses the Protocol Buffers format developed at Google.
2. Redis lists
   Redis lists are used to queue work to be processed by a different service. Messages written to Redis may be in Protocol Buffer format, standard JSON, or built-in Go data types.
3. NATS
   NATS provide an efficient message bus for service-to-service communication where gRPC would not be a good fit. All the messages sent and received over NATS is in the Protocol Buffer format.