Botchi Documentation

Table of Contents

Introduction

Botchi is a **free and open-source** desktop (Mac, Windows and Linux) **bot suite** developed by a single hobbyist programmer. There are no associated tokens, fees or user accounts. I have attempted to make Botchi accessible for non-programmers while providing a powerful platform for other developers to learn from, use and extend.

What can you do with it? Quite a lot! It can be used to build different trading strategies using **modular components** that you can incorporate however you like- from simple trail bots and grid-based strategies to bots with complicated stop-loss logic or even bots managed by *other* bots in a hierarchy of control.

But you could also, for example, run your own **auto-compounder** that performs a certain lineage of tasks at a specific time each day. You can even communicate with defi contracts directly to **play or test interactive contracts** without needing a specific web3.0 app.

Besides empowering users with the ability to create a wide range of automatons, there are other less obvious- but nonetheless very practical- benefits to using Botchi. Since all bot logic is handled *in Botchi*, you can implement **limit orders without locking up your funds** as you would placing such orders on a CEX. Quantities for trades can also be given as percentages of whatever your wallet balance is at the time a swap is triggered. In fact, you could run bots for different token pairs using the same wallet principle with a first-in-best-dressed approach. You can also use Botchi as an **accounting abstraction layer** over a single wallet or account to run multiple “virtual wallets” with their own principle and compounding pools.

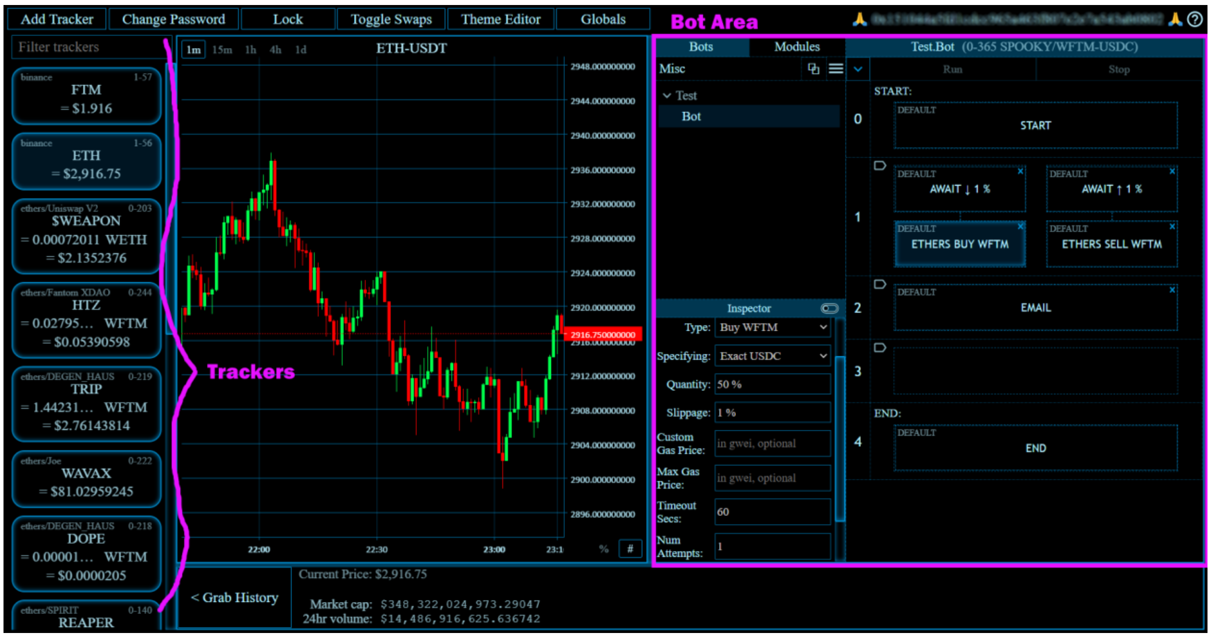
I have tried to make a versatile tool which anyone can use by dragging and dropping prebuilt modules. More than that, the ability to **write and share custom modules and import/export bots** will hopefully result in use cases and strategies I haven’t even though of!

Although- rather, *because*- Botchi is designed to work without me as a middleman, there are a few **considerations** to keep in mind:

1) To make use of Binance modules **you will need access to the Binance API** (these are free).
2) To make use of ethers modules (covering a broad range of blockchains such as Ethereum, Fantom and Binance Smart Chain) **you will need an API endpoint to talk to the blockchain**. Specifically, for now, you need a HTTPS RPC endpoint, with the access token embedded in the URL. I have been using a free plan through ANKR and it works fine- there are also paid plans that come with less restrictions.
3) Unlike dedicated websites that store price data even when no one is viewing a chart, **Botchi is responsible for getting price data itself each time it runs, as it runs.** To make up for this, there is a *Get History* button (available for Binance trackers only at the moment), which retrieves a few hours of price data to backfill the chart.
4) You can run botchi on your home computer, but **if you want to run 24/7 strategies I would advise renting a basic Virtual Machine from an established cloud provider.** Something powerful enough just for Botchi would be right on the cheap end of the scale. I’m *hoping* to run an adequate Azure VM for less than $15/month- I will update this paragraph.

I want you to know there are strong philosophical positions that have guided Botchi's design and distribution. In some ways it is for me, in its entirety, only a statement and manifestation of certain principles and as such… leaves nothing more to be said here.

High Level Overview



Add Tracker
Change Password
Lock
Toggle Swaps
Theme Editor
Globals
Bot Area
Filter trackers
binance 1-57 FTM = $1.916
binance 1-56 ETH = $2,916.75
ethers/Uniswap V2 0-203 $WEAPON = 0.00072011 WETH = $2.1352376
ethers/Fantom XDAO 0-244 HTZ = 0.02795… WFTM = $0.05390598
ethers/DEGEN_HAUS 0-219 TRIP = 1.44231… WFTM = $2.76143814
ethers/Joe 0-222 WAVAX = $81.02959245
ethers/DEGEN_HAUS 0-218 DOPE = 0.00001… WFTM = $0.0000205
ethers/SPIRIT 0-140 REAPER
Trackers
1m 15m 1h 4h 1d
ETH-USDT
22:00 22:30 23:00 23:1
< Grab History
Current Price: $2,916.75
Market cap: $348,322,024,973.29047
24hr volume: $14,486,916,625.636742
Bots
Modules
Misc
Test
Bot
Inspector
Type: Buy WFTM
Specifying: Exact USDC
Quantity: 50 %
Slippage: 1 %
Custom Gas Price: in gwei, optional
Max Gas Price: in gwei, optional
Timeout Secs: 60
Num Attempts: 1
Test.Bot (0-365 SPOOKY/WFTM-USDC)
Run
Stop
START:
START
AWAIT ↓ 1 %
AWAIT ↑ 1 %
ETHERS BUY WFTM
ETHERS SELL WFTM
EMAIL
END:
END

Figure 1: Botchi showing a workspace "Misc" that contains one bot group "Test" which has one bot named "Bot"

Botchi consists of **trading pair trackers** whose backends do whatever they need to, to provide a simple price stream for the chart and bot modules to make use of. **Modules are atomic units of logic** that either perform a specific function or wait for specific conditions. These modules can be dragged and dropped onto cells in a bot's logical structure, through which execution travels. **A bot's logical structure is a set of sequential rows**- either single cells or *race blocks* that consist of two internal rows, with the first module to complete in the first row determining which module gets executed in the second (see Figure 2). **Custom modules** can be written in Javascript and imported as plugins (these are written the same way and have the same accesses as the built-in modules, so *please be very careful about where you get them from!*).

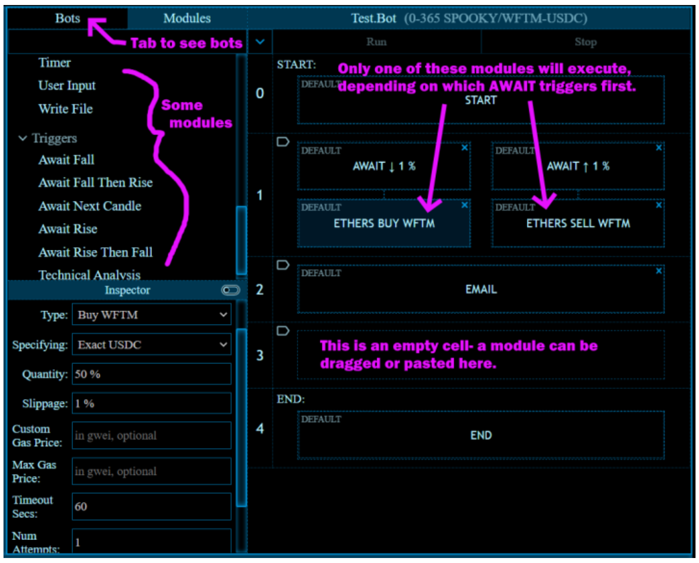


Figure 2: High overview of a bot being built

Bots are assigned to ***bot groups***: all bots in a bot group mirror the same logical structure and modules, but can differ in parameters (making it easier to manage the same *kinds* of bots for different trading pairs). Bot groups can themselves optionally be given a **workspace tag**. Selecting a workspace shows only the relevant bot groups- this makes it easier to manage sets of heterogenous bots that work together to achieve a specific goal.

User Data Files

Botchi stores a number of files specific to the user, which are here detailed. Note: no telemetry is gathered; nothing in these files is sent anywhere or used to generate data that is sent anywhere. By the way, because of this I don't know if you're experiencing bugs or even generally like or are making use of what I've built here so please, feel free to email me. I will have some indication from github of the popularity of Botchi, but that's about it!

Where Are They?

Windows: C:/Users/<USER>/AppData/Roaming/botchi/default_user/
Mac: ~/Library/Application Support/botchi/default_user/
Linux: ~/.config/botchi/default_user/

Note that other files and folders in the botchi folder are created by the electron framework upon which Botchi was built.

Directories:

- **bot_groups**: You might have been bot groups to be divided into workspaces in the file system but workspaces are just tags given to bot groups. Each bot group is a JSON file that defines the bots within that group.
- **logs**: These are partitioned by date Botchi was started, then by session (starting Botchi, then stopping Botchi, is a session), then by bot, then by the run of that bot. So if you run the same bot twice in one session you will have two folders under that bot for the two runs. The logs are HTML snippets by the way- literally just the trace pages- so they could be mined for data with some simple parsing if you'd like.
- **plugins**: JavaScript files of custom modules. This is where you place custom modules. Botchi will ask you to review and approve each new plugin when it first starts after a plugin is added and also after a plugin's code has changed.
- **themes**: Colour themes. These are simple JSON files, which you can edit yourself if you'd like.

Files:

- **binance-token-database.json**: Defines the trackers added that use the Binance backend. Note that this file contains any API keys used to authorize Botchi to read data from Binance.
- **config.json**: A simple JSON file containing variables kept between sessions to improve the user experience. These are:
    - trackerTopFrameHeightProportion
    - trackerChartWidthProportion
    - trackerUriStringsInOrder
    - lastTrackerType
    - showingSwaps
    - lastWorkspaceTag
- **ethers-token-database.json**: Defines the trackers added that use the Ethers backend. Note that this file contains the HTTPS RPC endpoints, with the access token embedded in the URL.
- **globals.json**: Stores global variables.

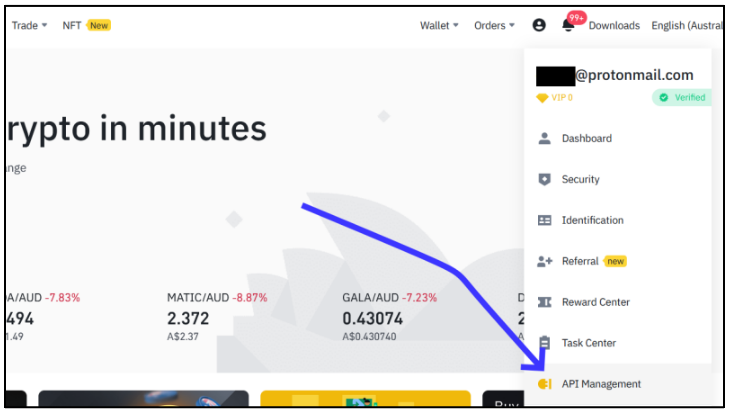
- **plugin-hashes.json**: contains all approved plugin hashes. This is the mechanism by which new plugins and alterations to existing plugins are detected.
- **script-modules.json**: Contains module authorizations (e.g. email authorizations, Binance account authorizations, blockchain wallet keys), which makes this the most sensitive file.

Getting API Access

APIs allow Botchi to communicate to different entities- be they CEX backends or a blockchain.

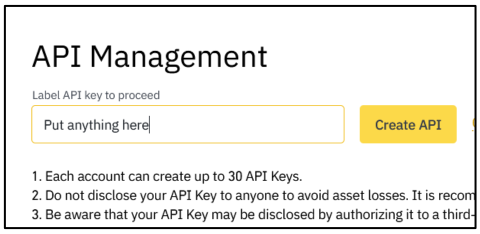
Getting a Binance API

Step 1) Log on to Binance and, via the dropdown menu attached to your account logo, select *API Management.*



Step 1: To go the API management page

Step 2) Enter a label for your API. Note: although it might be a good idea to use the same label for a given API in both Binance and Botchi, there is no requirement to do so.



API Management

Label API key to proceed

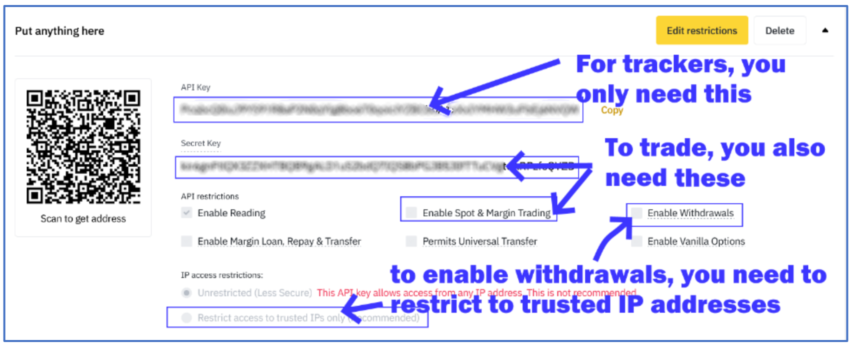
Put anything here

Create API

1. Each account can create up to 30 API Keys.
2. Do not disclose your API Key to anyone to avoid asset losses. It is recom
3. Be aware that your API Key may be disclosed by authorizing it to a third-

Step 2: Enter a label for your API

Step 3) Note the API key and the Secret Key. There are two places you use this information in Botchi. To create trackers for trading pairs you only need the API Key, but swapping and withdrawing through a bot module require the API Key *and* the Secret Key.



Put anything here

API Key
[redacted]

Secret Key
[redacted]

API restrictions

- [✓] Enable Reading
- [ ] Enable Spot & Margin Trading
- [ ] Enable Withdrawals
- [ ] Enable Margin Loan, Repay & Transfer
- [ ] Permits Universal Transfer
- [ ] Enable Vanilla Options

IP access restrictions:

- [✓ Unrestricted (Less Secure)] This API key allows access from any IP address. This is not recommended.
- Restrict access to trusted IPs only (Recommended)

For trackers, you only need this

To trade, you also need these

to enable withdrawals, you need to restrict to trusted IP addresses

Figure 3: Take note of the two keys to give to Botchi

Getting a Blockchain API (ANKR)

First it is important to note that ANKR is not the only company to offer such API access. They just happen to be the first one I tried and it's been good enough for my purposes.

Step 1) Go to https://app.ankr.com/auth/sign-up and sign up for an account.

Step 2) Sign in, head to https://app.ankr.com/api and click to create a new API in the top left (see Figure 4).

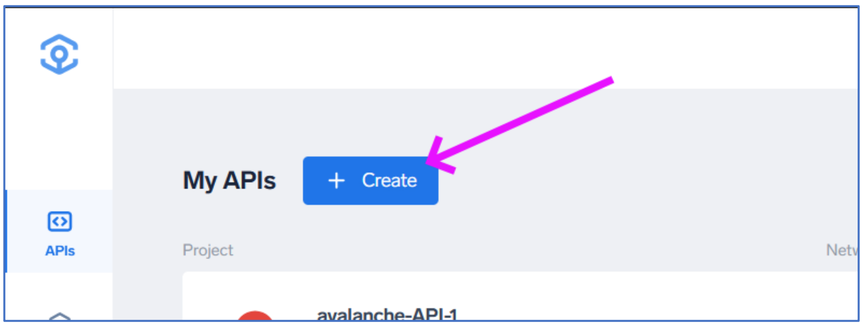


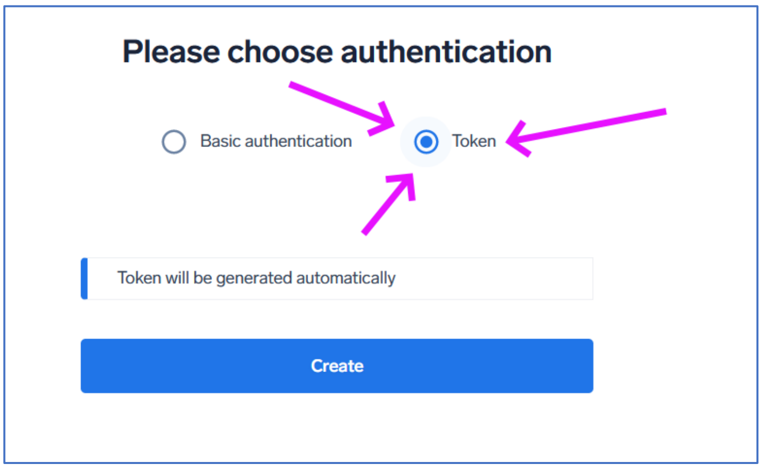
Figure 4: Click to create an API

Step 3) Choose an API from the list provided and click on it to *Deploy*. It should be pretty obvious which one to pick- it depends on what blockchain you're interested in (I have tested Ethereum, Fantom and Avalanche- and I *think* Binance Smart Chain at some point).

Step 4) Choose your API plan (free is basically good enough).

Step 5) In the next page you can name your project and choose which location to base it in. Note: although it might be a good idea to label your APIs the same in both ANKR and Botchi, there is no requirement to do so.

Step 6) This is really the only possible gotcha: make sure you select *Token* as your authentication method- this is *not* the default (see Figure 5). This isn't an area I have focussed on too much; if there's demand I'll have a look at supporting different authentication methods.


Please choose authentication

Basic authentication [✓ Token]

Token will be generated automatically

Figure 5: Select Token as the authentication method

Step 7) Once created, click on the project to see an overview for it, click on the Settings tab, and note the HTTPS endpoint (see Figure 6). This is what you will need to feed into Botchi.

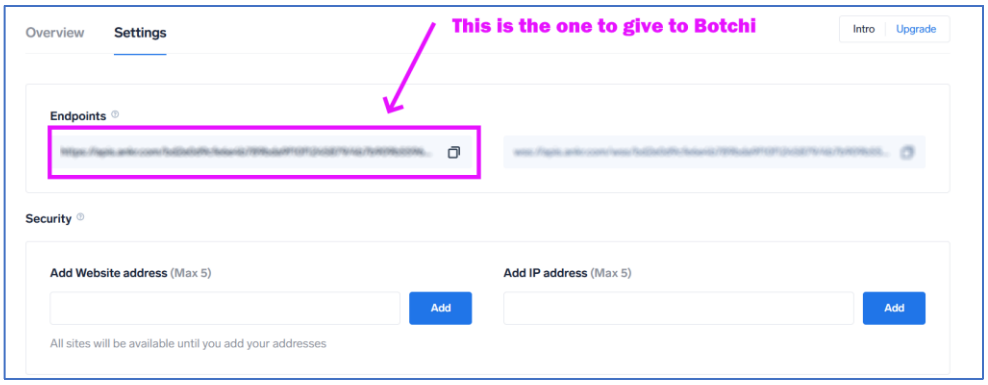

Overview
Settings
This is the one to give to Botchi
Intro
Upgrade
Endpoints
Security
Add Website address (Max 5)
Add
All sites will be available until you add your addresses
Add IP address (Max 5)
Add

Figure 6: Take note of the HTTPS endpoint

Unlike Binance, your authentication for making transactions using a blockchain wallet is completely separate to the authentication used for *talking* with the blockchain.

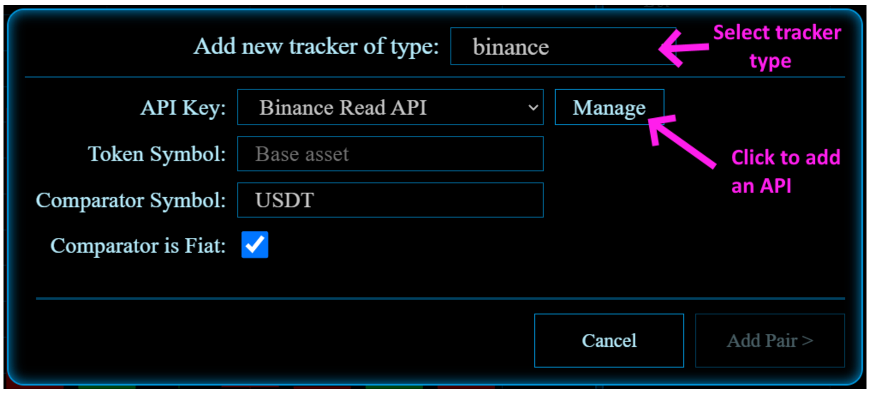
Adding Trackers

Trackers represent a trading pair. They are fed price data from the relevant backend which is then passed on to the chart and active bot modules. You may deactivate a bot module at any time without deleting it, so that it is longer listening for or processing data, by right-clicking on the tracker in the *Trackers* pane to the left (See Figure 1).

To add a tracker, click the *Add Tracker* button in the toolbar at the top of the app. From the drop-down box in the resulting popup you can select which type of tracker to add.

Adding a Binance Tracker

Step 1) To add an API that trackers can use to request price data from Binance, click the *Manage* button (see Figure 7).



Add new tracker of type: binance

API Key: Binance Read API

Token Symbol: Base asset

Comparator Symbol: USDT

Comparator is Fiat: ✓

Figure 7: Adding a Binance tracker type

Give the new API a name (which doesn't have to correspond to anything you entered on Binance) and paste in your API Key. Note that a tracker doesn't need to perform any actions that would require the API *Secret*, so there is no space to provide it here.

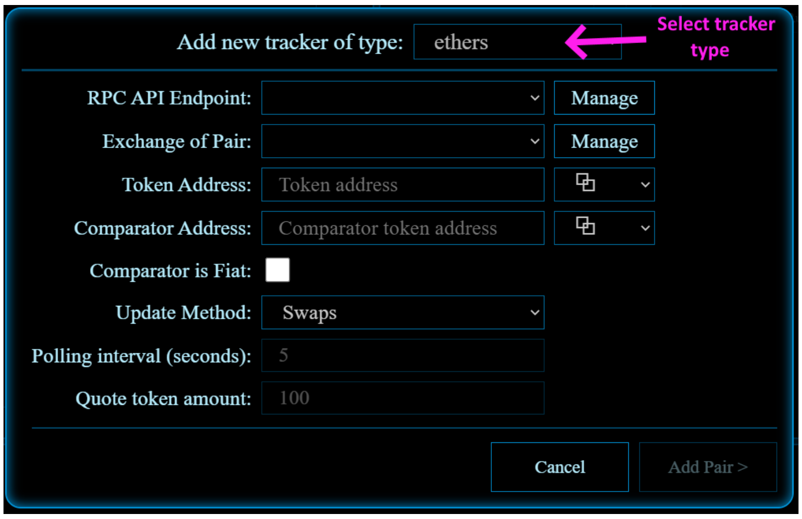
Step 2) All you need to do now is fill in the token symbol (e.g. FTM, ETH, BTC) and the comparator symbol (default USDT). Throughout Botchi, *token* refers to the token of interest, and *comparator* refers to the thing the token is being priced in (*token* is the base, *comparator* is the quote).

If the comparator is fiat, checking the relevant checkbox will let Botchi know this. Botchi tries to create a conversion chain from a given token to something measured in fiat in order to automatically give the price of that token in fiat as well as its value in *comparator*. For example, if you have one tracker tracking ETH-BTC and another tracking BTC-USDT, Botchi automatically calculates and shows the price of ETH in USDT.

Step 3) Click *Add Pair* to add the pair and start tracking trades. If you get an error stating that the request failed with status code 400, it is likely you have entered non-existent or incompatible symbols.

Adding an Ethers tracker

An ethers tracker requires more information than other types because of the decentralized nature of blockchain liquidity pools. A lot of this information comes in the form of *contract addresses* which are the primary means of identifying entities on a blockchain. You can usually hunt for these on the typical blockchain explorers (e.g. etherscan.io, ftmscan.com).

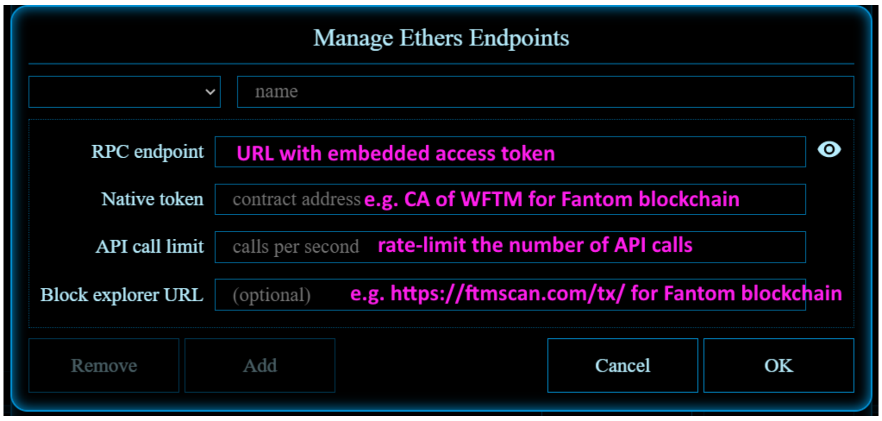


Add new tracker of type: ethers

RPC API Endpoint:	
Exchange of Pair:	
Token Address:	Token address
Comparator Address:	Comparator token address
Comparator is Fiat:	
Update Method:	Swaps
Polling interval (seconds):	5
Quote token amount:	100

Figure 8: Adding an ethers tracker type

Step 1) To add an API that trackers can use to communicate with a blockchain, click the first *Manage* button. This will show a popup similar to Figure 9. The first field- the RPC endpoint- is the HTTPS URL received from, for example, ANKR. The native token will be the contract address of ETHER for the Ethereum blockchain, FTM for the Fantom blockchain, BNB for the Binance Smart chain, and so on (in fact, we are really using the *wrapped* versions of those native tokens). These contracts can be a little difficult to find, so I will provide a list of common ones in Appendix A. Providing the blockchain explorer URL will enable Botchi to link to a summary of the transaction from each swap it reads in.



Manage Ethers Endpoints

name

RPC endpoint	URL with embedded access token
Native token	contract address e.g. CA of WFTM for Fantom blockchain
API call limit	calls per second rate-limit the number of API calls
Block explorer URL	(optional) e.g. https://ftmscan.com/tx/ for Fantom blockchain

Figure 9: Adding an ethers endpoint

Step 2) Next we will need to add an exchange. An exchange is an entity like Uniswap on Ethereum or SpookySwap on Fantom. An exchange will have its own liquidity pools which people trade through, so different exchanges can have slightly different prices for the same trading pair. To add an exchange, you will need the exchange's *factory contract address*, and its *router contract address*.

Again, these can be found using an explorer but I will provide a list of some common ones in Appendix A.

Note that there is currently no support for Uniswap V3.

Step 3) Throughout Botchi, *token* refers to the token of interest, and *comparator* refers to the thing the token is being priced in (*token* is the base, *comparator* is the quote). Clicking on the dropdown button beside these fields will allow you to auto-paste in contracts you've used in the past (you might find yourself reusing ETHER as the comparator quite often, for example).

If the comparator is fiat, checking the relevant checkbox will let Botchi know this. Botchi tries to create a conversion chain from a given token to something measured in fiat in order to automatically give the price of that token in fiat as well as its value in *comparator*. For example, if you have one tracker tracking $WEAPON-ETH and another tracking ETH-USDT, Botchi automatically calculates and shows the price of $WEAPON in USDT.

Step 4) There are two methods that Botchi can use to track ethers-based tokes. The *Swaps* method will track each swap transaction and take the price last swapped at as the current price. The *Poll* method regularly retrieves a quote from the liquidity pool directly, using a specific token amount to estimate the cost of that many tokens.

Common errors are incorrect (or accidentally swapped around) router and factory addresses, incorrect token addresses (the FTM token address differs between Ethereum and Fantom blockchains), and there being no liquidity pool for the token-comparator pair on the given exchange.

Bot Organization

Bot Groups

Bot groups adhere to a single *logical structure*. Changing the logical structure of one bot (by adding and removing rows) in a group propagates the change to all bots in that group. Similarly, the addition and removal of modules from *cells* in that logical structure will be mirrored. However, the *parameters* for these bots are independent. In this way, the same general strategies can be implemented for different trackers in an easily manageable way.

To create a bot group, make sure *Bots* is selected in the *Bots/Modules* tab and press the hamburger just below that same tab.

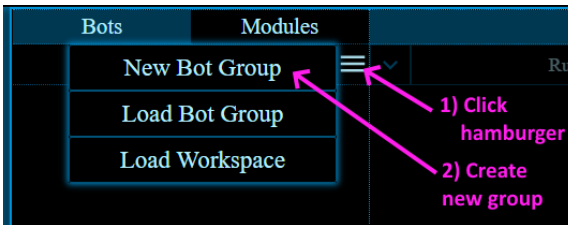


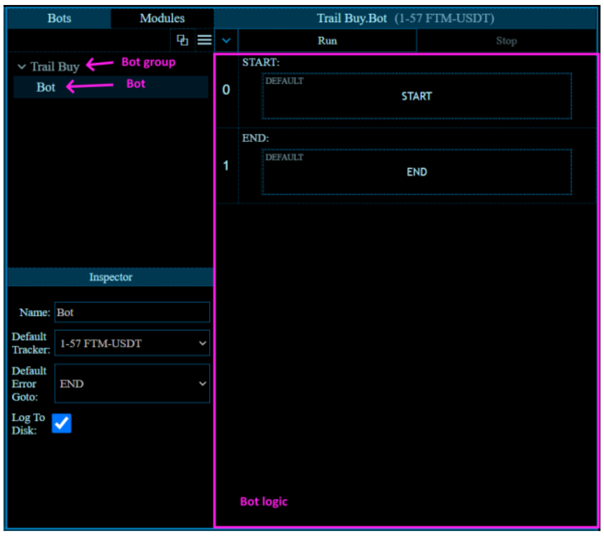
Figure 10: Creating a bot group

Give a name for your bot group and press *Create*. A new bot group will be created with one bot inside it called *Bot*. Clicking on the bot group will show some parameters that can be changed- namely, its **Name** and its **Workgroup Tag** (workgroups will be discussed later).

To add more bots to the bot group, right click on the bot group's name and select *Add Bot*.

Bots

Clicking on a bot’s name will show some editable parameters for it and display its logical structure to the right (see Figure 11). The displayed bot is independent to whatever tracker might be selected and showing on the chart.



Name: Bot
Default Tracker: 1-57 FTM-USDT
Default Error Goto: END
Log To Disk: ✓

Figure 11: Showing bot details

The editable bot parameters are

- **Name**: the name of the bot.
- **Default Tracker**: each module can have a different tracker as its subject. Most of the time within a single bot this tracker is the same across all modules. If a module is configured to use the default tracker, it will defer to this setting.
- **Default Error Goto**: if an error occurs within a module, it jumps to whatever row the **Error Goto** is set to for that module. If the module is configured to use the default error goto, it will defer to this setting.
- **Log to Disk**: A running bot generates a *trace*, divided into pages of 50 rows as execution passes through the rows of the bot logic. This can be useful to refer to if either an error occurs or the bot seems to have misbehaved. It can also be useful data to process to determine how a bot has been performing. This option determines whether the trace log is saved out to disk or not. In either case, a live trace is viewable for a running bot. Note that for long-running bots, logging traces to disk can greatly reduce RAM usage (Botchi will only cache some trace pages in RAM and retrieve others from disk if the user navigates to those pages). It is safe to remove trace pages cached to disk while a bot is running. At the end of a run, if this parameter is checked, Botchi will also log a copy of the bot as a JSON object. Logs are kept with other user data files (see the section on User Data Files).

Running bots will have an active tag to the left of their name, and bots with running bots below them in the spawn tree will have an icon displayed to the right of their name (see Figure 12). Right clicking on a bot will give you the option of viewing the spawn tree, which updates in real time.

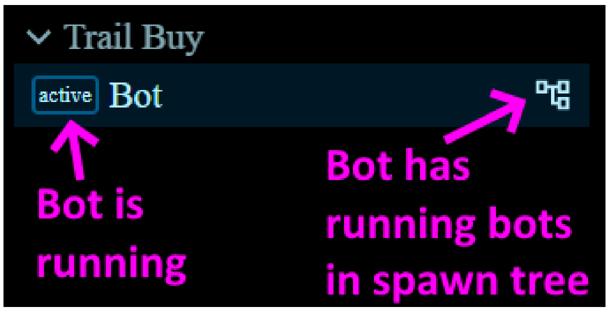


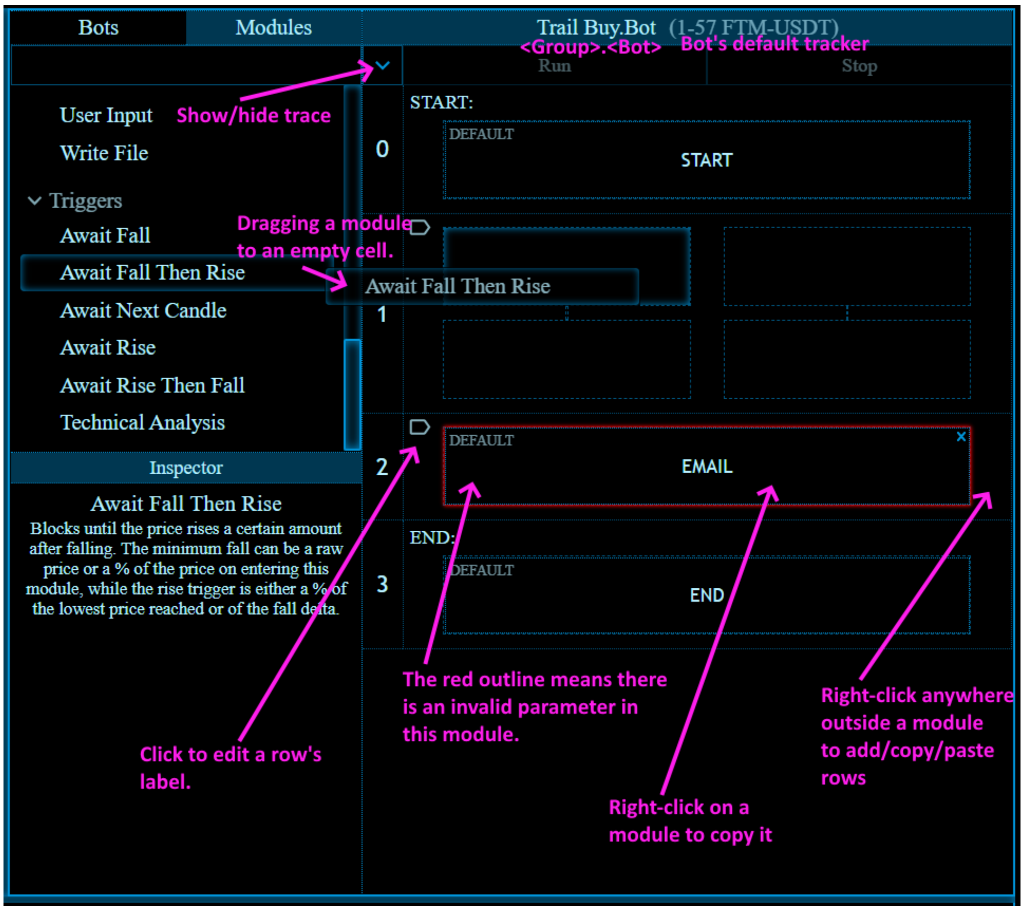
Figure 12: Bot icons

Workspaces

Groups of bots may work together to achieve a certain goal (this is particularly true for bots that spawn other bots or read and write to the same global variables). To assist with managing these conceptual “groups of groups”, each bot group has a **Workspace Tag** parameter that can be set, after which it will only appear when either that workspace is selected, or the *<All>* workspace is selected. A workspace can be selected using the dropdown button to the left of the hamburger, and the name of the current workspace is displayed in a label to the left of *that*. To change the name of the current workspace (and the workspace tag of all bot groups within it), simply click on the workspace label and enter a new name.

Building a Bot

Building a bot consists of adding sequential rows of two possible types, filling in the cell(s) of those rows with modules, and editing the parameters of those modules in the inspector.



Await Fall Then Rise

Blocks until the price rises a certain amount after falling. The minimum fall can be a raw price or a % of the price on entering this module, while the rise trigger is either a % of the lowest price reached or of the fall delta.

Figure 13: Basic bot-building interface

Rows

A bot comes with Start and End rows, with corresponding modules filled in. There is no inherent functionality to these modules.

Right-clicking anywhere in a row (apart from on an actual module) will bring up a context menu allowing you to insert one of the two types of rows. Note that you cannot insert rows after the End row, so no context menu will appear for it.

A *Single* (Figure 14) is a row that holds a single module, which is activated when execution reaches that row. Execution waits until that module is completed before moving to the next row.

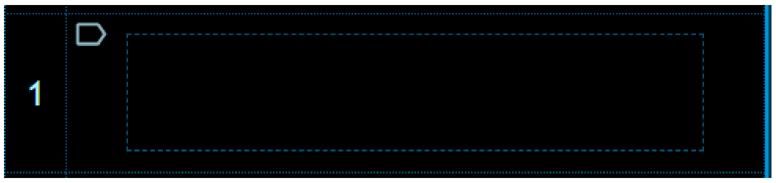


Figure 14: Row of type Single

A *Race Block* (Figure 15) consists of four cells in 2x2 formation. When execution reaches this block, it activates both modules in the first *internal* row and waits for one of those modules to complete. Once the first module completes, the module directly beneath it is activated and the module *next* to it is deactivated. In this way, only one of the modules in the *second* internal row is activated. Execution then waits until the active module in that second row is completed before moving on to the next logical row. NOTE: if the module in the first column of the first internal row completes immediately (e.g. the Goto module), the module in the second column will not get activated.

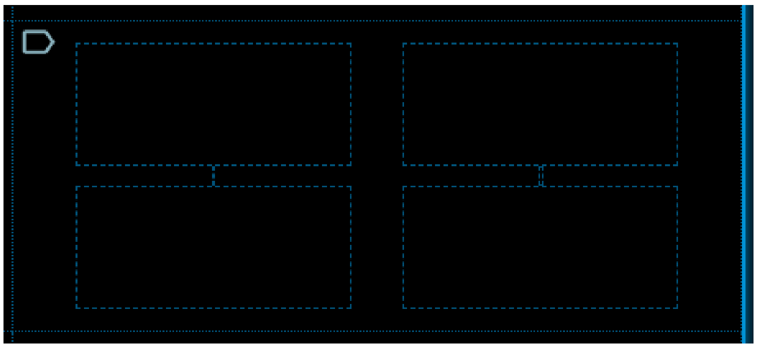


Figure 15: Row of type Race Block

There are special modules, such as Statements and User Input, that output one of two possible values. These modules can be dragged into the top internal row of a Race Block such that it fills that whole row. In these cases, it is the output of the module that determines which module in the second internal row is activated.

Rows can be given labels, which can be referred to in places where a row number might be given. For example, the Go To module lists row labels as well as row numbers. Also, instead of referring to the result of a row in a parameter that accepts expressions by referring to its number (e.g.

`$r.4.result`), you can refer to it by label (e.g. `$r.get_epoch.result`). Labels are case-insensitive.

Since execution can jump to arbitrary rows, a row may be executed more than once. A row is always referred to by the same row number / label no matter how many times execution has passed through it- and it will always hold the result of the *most recent* execution.

Each row has properties that are updated each time execution passes through it, which can be referred to by other rows using the `$r.<property>` syntax (see the section on Expressions and Special Characters in Parameters).

AND Patterns

You might notice that race blocks offer a kind of OR pattern, but there is no equivalent AND. This is because I couldn't think of a way to determine AND pattern time thresholds in a universal manner. I suggest the pattern shown in Figure 16.

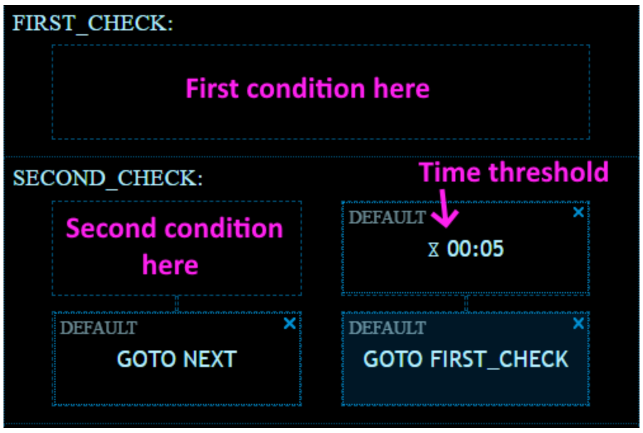


Figure 16: Demonstration of AND pattern

Modules

To see a list of modules, switch to *Modules* in the *Bots/Modules* tab. Dragging and dropping a module into an empty cell will instantiate it there, and its parameters will be shown in the *Inspector* pane. Refer to documentation on a specific module to understand what it does and how the parameters change its behaviour.

Each module (including the Start and End modules) comes with a set of common properties that can be accessed by toggling the *Advanced Properties Toggle* in the Inspector pane (see Figure 16). Here you can set the tracker that is the subject of the module, the row to which execution will jump if an error occurs, and statements to be run before and after the module's execution (see the section on Statements). The statements to run after the module has completed can refer to that row's results, making it useful for post-processing data. A common use case of these statement blocks would be to convert token amounts between the large integers that blockchains accept and the smaller decimal numbers that users are used to.

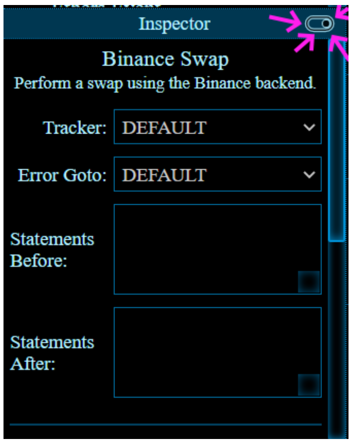

Inspector
Binance Swap
Perform a swap using the Binance backend.
Tracker: DEFAULT
Error Goto: DEFAULT
Statements Before:
Statements After:

Figure 17: Showing advanced properties of a module

Note that some modules require a certain type of tracker as subject, while some others have no such requirement BUT will result in an error or- worse- unexpected behaviour if you refer to the subject tracker (e.g. using `$t.tokenAddress` in a parameter) and it doesn't *have* that property at all or it doesn't mean the same thing.

Press the small x on the top right of a module to delete it from the cell it occupies.

When setting parameters for a module, right-clicking on a text field will bring up a bigger text editor.

Expressions and Special Characters in Parameters

A lot of parameters can take an expression as their value. In an expression, there are a range of special characters that get substituted for real values at runtime.

Special Character	Details
$p or $P	The price on activation of the module, given as a decimal number.
$t.<property> **or** **$T.<property>**	The <property> of the tracker that is the subject of the module. These can vary between tracker types. For Binance trackers these are • name • ticker • type • tokenSymbol • comparatorSymbol • comparatorIsFiat • isActive For ethers trackers: • name • tokenAddress • tokenDecimals • tokenSymbol • comparatorAddress • comparatorDecimals • comparatorSymbol • pairAddress • pairDecimals • comparatorIsToken1 • comparatorIsFiat • isActive
$r.<row >.<property> **or** **$R.<row>.<property>**	The <property> of <row> from its most recent execution. A <row> is given by its row number (seen to the left of the *Bot Logic* pane) or its label. Properties include • entryPrice • entryPriceFiat • exitPrice • exitPriceFiat • error (if any) • traceOutput (a summary) • result The result is the result of the most recent execution of the last module in that row. In Race Blocks it is ultimately the result of the module that ran in the second internal row; however, the second internal row can use this syntax to refer to the first internal row. If the result is an object that contains key-value pairs, these can be accessed using a further dot notation. Many calls to APIs return such objects (for example, $r.binanceBalanceRow.result.**balance**).
$d{<format>}	The current date, expressed according to the given <format>. Please

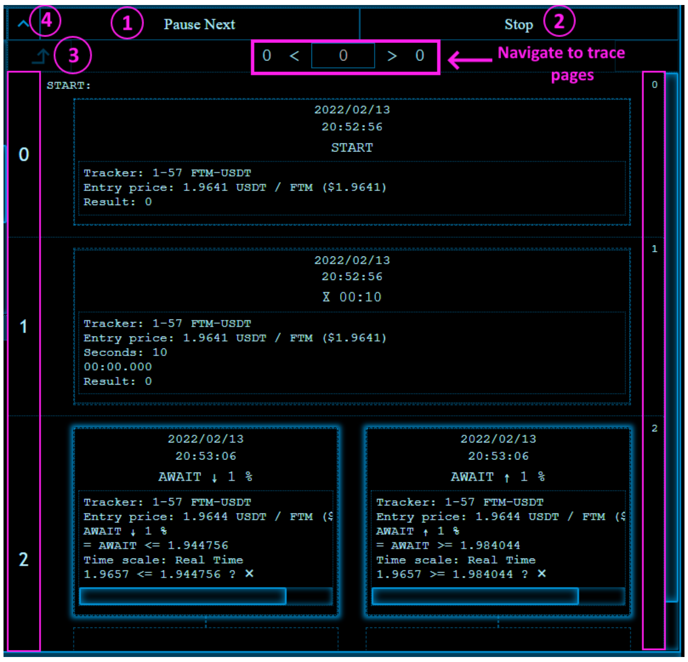
or **$D{<format>}**	refer to https://arshaw.com/xdate/#Formatting for a list of special characters that can be used here. For example, `$d{MMM d, yyyy}` would be substituted with something like `Feb 13, 2022.`
$v.<name> **or** **$V.<name>**	A local bot variable. These must be assigned to in a statement somewhere before being referred to. They can also be updated in a statement. Each time a bot runs it forgets its old local variables and local variables cannot be referred to from within other bots. Refer to the section on Statements for more.
$g.<name> **or** **$G.<name>**	A global variable. These must be listed in the global variables accessed by pressing the corresponding button in Botchi's toolbar. Changing these variables in Statement modules changes it for all bots that refer to it. In other words, they are shared across the app. Refer to the section on Statements for more.

Running a Bot

If there are no empty cells or invalid parameter values, the *Run* button at the top of the *Bot Logic* pane will enable, allowing you to run your bot. It is assumed that by the time you press that button, you have checked that the bot is as you want it to be. Please always keep in mind that bots are capable of powerful actions and like most things in crypto there is potential for irreversible, massive and *swift* damage to your portfolio.

Traces

When a bot is started, it will show the live trace (see Figure 18), divided into pages of 50 rows each. If the bot is set to log to disk, then when a trace page is filled up it is immediately written out to disk. Pressing **(1)** will pause the bot after the current module is completed. Pressing **(2)** will halt the bot and deactivate all active modules. Note that a module may continue to run for a time after it is deactivated- it depends on what the module is doing at that moment (for example, if it is waiting for a reply from a call to an API, it might continue to wait for that reply). For bots that have been spawned by other bots, pressing **(3)** travels back up a level to the parent bot. Pressing **(4)** will toggle the interface between the bot's logic and the bot's current (or most recent) trace.



```
START:

2022/02/13
20:52:56
START
Tracker: 1-57 FTM-USDT
Entry price: 1.9641 USDT / FTM ($1.9641)
Result: 0

2022/02/13
20:52:56
X 00:10
Tracker: 1-57 FTM-USDT
Entry price: 1.9641 USDT / FTM ($1.9641)
Seconds: 10
00:00.000
Result: 0

2022/02/13
20:53:06
AWAIT ↓ 1 %
Tracker: 1-57 FTM-USDT
Entry price: 1.9644 USDT / FTM ($
AWAIT ↓ 1 %
= AWAIT <= 1.944756
Time scale: Real Time
1.9657 <= 1.944756 ? ✕

2022/02/13
20:53:06
AWAIT ↑ 1 %
Tracker: 1-57 FTM-USDT
Entry price: 1.9644 USDT / FTM ($
AWAIT ↑ 1 %
= AWAIT >= 1.984044
Time scale: Real Time
1.9657 >= 1.984044 ? ✕
```

Figure 18: A running bot's trace

The row numbers on the left correspond to that row's number in the bot's logic. The row numbers on the right increment for each row added to trace as the bot runs.

All module traces contain at least the date and time activated, its name, the module's subject tracker and the price of the subject tracker's token at the time of activation. The rest varies depending on what the module actually does, but should give a good indication of a) what active modules are *doing*, and b) what completed modules have *done*.

Note that right-clicking on a module trace will present its output in a bigger textbox.

Traces are always stored in plain text. If a trace somehow ended up in the hands of a bad actor, there’s not much they can do with it- the trace doesn’t contain sensitive information. However, if a bad actor found a way to edit these trace files when a bot is running, and then they (or the user) navigate to a trace page that is not cached in RAM (and therefore needs a disk lookup), any scripting embedded in that page will get executed. The assumption is, if a bad actor had as much control over your system to do that while a bot is running, Botchi is already considered compromised. The alternative is to encrypt all trace files the same way auth objects are- this is a design decision I’m open to changing.

Built-In Modules

Parameters marked with [EXPRESSION] get parsed as expressions (see the section on Expressions and Special Characters in Parameters).

Binance Balance

Summary:

Retrieves the balance of the user with the given auth.

Parameters:

- **Token**: [EXPRESSION] Symbol of the token to get the balance of in the user's wallet
- **Auth**: The auth object identifying and granting access to the user's Binance account

Result:

An object with one item: "balance". Note that this is different to the result of the Ethers Balance module which returns a decimal number, not an object.

Binance Query

Summary:

Returns the result of a query to the Binance API. You can find the documentation for the Binance API here.

Parameters:

- **Query**: [EXPRESSION] The query part of what will be the URL. For example, according to the Binance documentation /sapi/v1/capital/withdraw/history has no mandatory parameters (apart from timestamp, which is taken care of), so that is a valid query by itself. If you wanted to supply the coin parameter to, say, only retrieve information about your ETH withdrawals, that would be /sapi/v1/capital/withdraw/history?coin=ETH
- **Auth**: The auth object identifying and granting access to the user's Binance account

Result:

Dependent on the query. Usually an object or an array of objects.

Binance Swap

Summary:

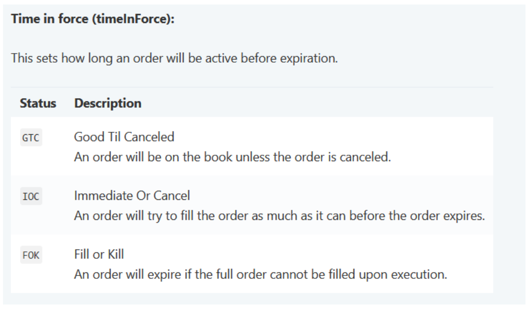
Performs a swap using the Binance backend

Parameters:

- **Order**: Either Market or Limit.
- **Type**: Either Buy or Sell
- **Token**: [EXPRESSION] The symbol of the token to buy or sell
- **Comparator**: [EXPRESSION] The symbol of the token to buy *with* or sell *for*.
- **Specifying Exact**: For market orders only. The token that the parameter **Quantity** is specifying. For example, specifying the exact amount of *token* for a market sell will tell Binance to sell that exact amount of *token* for as many

comparator as possible; specifying *comparator* for a market sell will tell Binance to sell however many *token* necessary to result in that exact amount of *comparator*.

- **Quantity**: [EXPRESSION] For market orders, this is the amount of *token* to buy or sell, or the amount of *comparator* to buy with / sell for. For limit orders, this is simply the amount of token to buy or sell. This is can be given as a percentage (append %) which will resolve to that percentage of the user's balance of the specified token.
- **Price**: [EXPRESSION] For limit orders only. The minimum price that a sell will execute, or the maximum price that a buy will execute.
- **Time In Force**: For limit orders only. From the Binance API docs:



Time in force (timeInForce):

This sets how long an order will be active before expiration.

Status	Description
`GTC`	Good Til Canceled An order will be on the book unless the order is canceled.
`IOC`	Immediate Or Cancel An order will try to fill the order as much as it can before the order expires.
`FOK`	Fill or Kill An order will expire if the full order cannot be filled upon execution.

Figure 19: Time in force options

- **Auth**: The auth object identifying and granting access to the user's Binance account

Result:

Botchi requests the RESULT response type from Binance and adds some information of its own to make life a bit easier. You can see what is returned from a RESULT request here, to which Botchi adds

- averagePrice
- tokenAmountIn (0 for buys)
- tokenAmountOut (0 for sells)
- comparatorAmountIn (0 for sells)
- comparatorAmountOut (0 for buys)

Binance Wait

Summary:

Waits for either a Binance withdraw or Binance deposit to process. After making a deposit to your Binance account from a blockchain wallet, even after the transaction goes through on the blockchain it is not yet ready to actually spend on your Binance account, so this is useful if you want to automate a process involving that.

Parameters:

- **Type**: Deposit or Withdraw
- **Tx**: [EXPRESSION] The transaction hash for the transaction. Note that this is the **txId** field of the response to a Binance withdraw request. For deposits this is just the hash of the transfer transaction.

- **Timeout Secs**: Maximum time to wait for the transaction to be processed. Note that this doesn't affect the actual transaction- it just gives up waiting for it and errors after this amount of time has elapsed.
- **Auth**: The auth object identifying and granting access to the user's Binance account

Result:

An object with information about the deposit/withdrawal and its status (see here).

Binance Withdraw

Summary:

Withdraws a token to a blockchain wallet. Note that Botchi does not support withdrawals to or deposits from a bank account to Binance. Also note that there is no equivalent Binance Deposit module because you would do that using the Ethers Transfer module.

Parameters:

- **Network**: Identifies the blockchain to withdraw to. If blank, this will be the default chainf or the given **Token**. The full description of all coins for a given user, including their supported and default networks, you can use the Binance query /sapi/v1/capital/config/getall. It is long. One option is to use the Write File module to write the result of this call to a file which you can then open in an application more suitable for reading JSON objects (for example, most browsers are probably better for it than Botchi's trace output box or even the enlarged textbox you can see by right-clicking on the trace).
- **Token**: [EXPRESSION] The symbol identifying the token to be withdrawn.
- **Quantity**: [EXPRESSION] The quantity of token to be withdrawn.
- **To Address**: [EXPRESSION] The blockchain wallet to withdraw the tokens to. Please double-check this, along with the **Network**, to ensure you are sending your tokens where you want them and where they will not be lost for eternity.
- **Await Confirm**: Checking this will negate the necessity for adding a Binance Wait module between this module and making use of the tokens from your blockchain wallet (assuming the withdrawal succeeds).
- **Confirm Timeout Secs**: Only if **Await Confirm** is true. Similar to the **Timeout Secs** of a Binance Wait. Note that this doesn't affect the actual transaction- it just gives up waiting for it and errors after this amount of time has elapsed.
- **Auth**: The auth object identifying and granting access to the user's Binance account

Result:

An object with information about the withdrawal and its status (see here).

Ethers Balance

Summary:

Retrieves the amount of a token in a given wallet. This is a read operation and as such does not need any auth object.

Parameters:

- **API**: The API to use for the request.
- **Token Address**: The contract address of the token in question.
- **Wallet Address**: The wallet address to check the balance of.

Result:

The balance, in decimals. Note that this is different to the result of the Binance Balance module, which returns an object.

Ethers Call

Summary:

Calls a function on a blockchain contract. Note that expression resolution is done by substitution of special characters only- no math is carried out. If you set the value of Arg 0 to `$p + 10`, the function will be called with argument 0 as something like `2.19 + 10`, NOT `3.19`. You can use a Statements module or the **Statements Before** advanced property of *this* module to carry out mathematics and refer to that result in the parameters.

Parameters:

- **API**: The API to use for the request.
- **Contract**: [EXPRESSION] The address of the contract.
- **Num Attempts**: If the transaction fails, Botchi will try again until it runs out of attempts.
- **ABI Fragment**: This is either a human readable or JSON ABI fragment detailing the function to be called. An example of a human readable fragment is
  ```
  function betBull(uint256 epoch) external payable
  ```
  while the corresponding JSON ABI fragment is
  ```
  {"inputs":[{"internalType":"uint256","name":"epoch","type
  ":"uint256"}],"name":"betBull","outputs":[],"stateMutabil
  ity":"payable","type":"function"}
  ```
  Both can be found using the normal blockchain explorer websites from the contract tab of verified contracts (see Figure 19). The former can be copied straight from the contract code, while the latter can be found at the bottom of the page under *Contract ABI* (it can be difficult to parse where one fragment stops and another starts- you only want the fragment pertaining to the function you are calling).

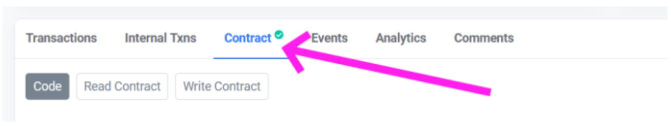


Figure 20: The contract tab of a contract's page on ftmscan.com

- **Arg 0… Arg N**: [EXPRESSION] Arguments to be supplied to the contract. The ABI fragment determines how many of these there are (to a maximum of 10).

- **Auth**: The auth object identifying and granting access to the blockchain wallet. A lot of function calls are read operations which don't need a wallet to make a transaction and therefore don't need an auth. If an auth is specified, a few more parameters are relevant.
- **Custom Gas Price**: Only if an auth is provided. This is the amount of gas to pay. Leave blank to pay the recommended amount of gas. This can be given as a percentage, which is taken to be a percentage of the recommended gas price.
- **Max Gas Price**: Only if an auth is provided. The maximum price for gas you are willing to pay. If the custom gas price is given as a percentage, and what that resolves to is greater than the maximum gas price given but the maximum price is greater than the *recommended* gas price, then the transaction continues using the maximum gas price.
- **Value Field**: [EXPRESSION] Some function calls make use of the value field of a transaction. You can specify that field here.

Result:

Depends on the function call.

Ethers Event

Summary: Blocks until an event is fired in a contract.

Parameters:

- **API**: The API to use for the request.
- **Contract**: [EXPRESSION] The address of the contract.
- **ABI Fragment**: This is either a human readable or JSON ABI fragment detailing the function to be called. Please see the same parameter described under the Ethers Call module.

Result:

Depends what information the contract gives along with notification that the event has fired.

Ethers Liquidity

Summary: Deposits the *token* and *comparator* of the subject tracker into the tracker's exchange to create LP tokens, or trades back LP tokens to the subject tracker's exchange for individual *token* and *comparator*.

Parameters:

- **Type**: Add or Remove
- **Token Quantity**: [Expression] Only valid when adding liquidity. This is the amount of *token*. The amount of comparator will be equal tot his amount in value. This can be given as a percentage (append %) which will resolve to that percentage of the user's *token* balance.
- **Liquidity Amount**: [Expression] Only valid when removing liquidity. This is the amount of LP tokens to remove and split back into *token* and *comparator*. This can be given as a percentage (append %) which will resolve to that percentage of the user's LP balance.
- **Min Eth Reserved**: Only valid when adding liquidity. Although "Eth" is used here, this is the minimum *native token* you want to leave yourself and is only relevant when *token* or *comparator* are that native token. For example, say your subject tracker is {*token*: HERTZ, *comparator*: FTM, *exchange*: SpookySwap (which is on Fantom blockchain)}. If you have enough HERTZ to pair with *all* of your FTM for LP and write 100% for **Token Quantity**, Botchi will do exactly that and leave you without gas unless you also set *this* parameter to, say, 2.
- **Slippage**: Slippage is the tolerance, given as a percentage, for price change you're willing to accept between the time you submit the transaction to the time it is filled.
- **Custom Gas Price**: Please see the same parameter under the Ethers Call module.
- **Max Gas Price**: Please see the same parameter under the Ethers Call module.
- **Timeout Secs**: Maximum time to wait for the transaction to be processed. Calls to add or remove liquidity have a deadline argument, and that is where this is used. A timeout means the transaction has failed.
- **Num Attempts**: If the transaction fails, Botchi will try again until it runs out of attempts.
- **Auth**: The auth object identifying and granting access to the blockchain wallet.

- **Result**:

  An object with the following fields:
  - transactionHash
  - lpBanaceBefore
  - lpBalanceAfter
  - tokenBalanceBefore
  - tokenBalanceAfter
  - comparatorBalanceBefore
  - comparatorBalanceAfter

Ethers Swap

Summary:

Performs a swap using the Ethers backend.

Parameters:

- **Type**: Buy or Sell
- **Specifying**: You can either specify exact *token* or exact *comparator*. This is the token that the **Quantity** parameter is specifying. For example, specifying the exact quantity of *token* will tell Botchi to sell that exact amount of *token* for as many *comparator* as possible; specifying *comparator* tell Botchi to sell however many *token* necessary to result in that exact amount of *comparator*.
- **Quantity**: [EXPRESSION] This is the amount of *token* to buy or sell, or the amount of *comparator* to buy with / sell for. This is can be given as a percentage (append %) which will resolve to that percentage of the user's balance of the specified token.
- **Slippage**: Slippage is the tolerance, given as a percentage, for price change you're willing to accept between the time you submit the transaction to the time it is filled.
- **Custom Gas Price**: Please see the same parameter under the Ethers Call module.
- **Max Gas Price**: Please see the same parameter under the Ethers Call module.
- **Timeout Secs**: Maximum time to wait for the transaction to be processed. Calls to swap tokens have a deadline argument, and that is where this is used. A timeout means the transaction has failed.
- **Num Attempts**: If the transaction fails, Botchi will try again until it runs out of attempts.
- **Auth**: The auth object identifying and granting access to the blockchain wallet.

Result:

An object with the following properties:

- transactionHash
- tokenBalanceBefore
- tokenBalanceAfter
- comparatorBalanceBefore
- comparatorBalanceAfter
- tokenAmountIn (0 for type Buy)
- tokenAmountOut (0 for type Sell)
- comparatorAmountIn (0 for type Sell)
- comparatorAmountOut (0 for type Buy)
- averagePrice

Ethers Transfer

Summary:

Transfers tokens to a wallet.

Parameters:

- **API**: The API to use for the request.
- **Token Address:** [EXPRESSION] The address of the token to be swapped.
- **To Address:** [EXPRESSION] The wallet address that the tokens will be sent to.
- **Quantity**: [EXPRESSION] The number of tokens to transfer. This is can be given as a percentage (append %) which will resolve to that percentage of the sender wallet's balance.
- **Custom Gas Price**: This is the amount of gas to pay. Leave blank to pay the recommended amount of gas. This can be given as a percentage, which is taken to be a percentage of the recommended gas price.
- **Max Gas Price**: The maximum price for gas you are willing to pay. If the custom gas price is given as a percentage, and what that resolves to is greater than the maximum gas price given but the maximum price is greater than the *recommended* gas price, then the transaction continues using the maximum gas price.
- **Num Attempts**: If the transaction fails, Botchi will try again until it runs out of attempts.
- **Auth**: The auth object identifying and granting access to the blockchain wallet.

Result:

An object with the following properties:

- transactionHash
- senderBalanceBefore
- senderBalanceAfter
- receiverBalanceBefore
- receiverBalanceAfter
- tokenAmountIn

Child Process

Summary:

Runs a command in a child process.

Parameters:

- **Hide window**: Determines whether the shell window should be hidden. If the window is not hidden, the result of stdOut and stdError are not captured in the results of this module- that's all shown in the shell window.
- **Is Blocking:** Determines whether Botchi should wait until the process exits before moving on to the next module.

Result:

If **Hide Window** is checked, the result is an object containing the following properties:

- stdOut
- stdError

Email

Summary:

Sends an email using SMTP.

Parameters:

- **Is Blocking**: Whether Botchi should wait until confirmation or error. Note that with this unchecked, you won't be notified of what the error *was* if there's an error- execution moves on as soon as the request to send an email is sent.
- **To**: [EXPRESSION] The email address to send the email to.
- **Subject**: [EXPRESSION] The subject line of the email.
- **Text**: [EXPRESSION] The body of the email (reminder that you can right click on text-box parameters to see a bigger text box.
- **Auth**: The auth (i.e. username and password) of whatever email account you're using, and the SMTP host through which you're sending the emails. I have been using my Gmail with this module- for this, the host is `smtp.gmail.com` and you will need to enable less secure apps here.

Result:

"Success!" on success if blocking.

Go To

Summary:

Jump to a row. Note that if this is in the first internal row of a race block and given the value NEXT, execution will move to the second internal row of the same race block.

Parameters:

- **Goto**: The row to jump to. It is recommended to use row labels with this module in particular.

Result:

Goto modules reflect the result of the previously completed module. The most practical benefit of this is that if a Goto is in the second row of a race block, other rows referring to the race block's results will see the results of the *first* internal row.

Random Number

Summary:

Generates a random number of uniform probability within a range. Note that this is not cryptographically secure- it simply uses `Math.random()`.

Parameters:

- **Is Integer**: Whether the result should be a whole number.
- **Min:** Minimum of the range. This inclusive. Assumed to be an integer if **Is Integer** is checked.
- **Max**: Maximum of the range. This is exclusive. Assumed to be an integer if **Is Integer** is checked.

Result:

The random number.

Spawn Bot

Summary:

Spawns a bot based on an existing bot. Note that this takes a copy of the bot as it exists each time execution reaches this module, but after that point the two bots are independent (you can edit the template bot and not affect the spawned copy).
A link to the spawned bot will be added to the trace, but a clearer view of the spawn tree can be seen by right-clicking the root bot's name in the workspace panel and selecting *Show Tree*. This view updates in real time and the bots therein can be accessed by clicking on them. See the section on Bots and Figure 12 in particular for more information.
The workspace (if any) of the bot wherein this module resides will be given preference in the search for a bot matching the **Bot Group** and **Bot Name** parameters.

Parameters:

- **Bot Group**: The name of the bot group to copy.
- **Bot Name**: The name of the bot to copy.
- **Start From Row**: Which row to start the spawn's execution at. A useful pattern might be to initialise variables for a bot in the Start module, and then skip that first row if you're using the **Initial Statements** to run the bot with custom variables.
- **Initial Statements**: Functions the same way as the **Statements Before/After** advanced properties and the **Statements** parameter of the Statements module, except the variables resolved in this field are attached to the new bot. Accordingly, the only difference is that in *this* field only local variables (`$v.<variable>`) are allowed.
- **Is Blocking**: Whether this module should block until the spawned bot completes. If this module is blocking, the spawned bot is halted when this module is

deactivated (for example, because it is in the first internal row of a race block and the other module in the first internal row has completed.

Result:

An object containing the local variables of the bot. If this module is non-blocking, this will just be the result of the Initial Statements being resolved, otherwise it will contain all local variables and the values that they were when the bot completes.

Statements

Summary:

Evaluates a set of statements, one per line, which assign the result of an expression to either a local or global variable. Local variables are of the form `$v.<name>` and global variables are of the form `$g.<name>`. Global variables must be listed in the global variables accessed by pressing the corresponding button in Botchi's toolbar. For local variables, you only need to make sure you have assigned something to them before referring to them. Statements are evaluated line by line so statement blocks like

```
$v.hundred = 10 * 10
$v.million = $v.hundred * $v.hundred * $v.hundred
```

are fine because the local variable `hundred` is assigned a value before it is referred to.

There are two types of expressions, dictated by the assignment operator used. Statements of the form X = Y treat the expression on the right (Y) as a mathematical expression: substitutions are made (see the section on Expressions and Special Characters in Parameters for more on that) and then evaluated mathematically.

Available operators are

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Power: ^

There are also comparison operators:

- Equality: ==
- Less than: <
- Less than or equal: <=
- Greater than: >
- Greater than or equal: >=

which resolve to 1 if true, or 0 otherwise.

In statements of the form X := Y, substitution occurs just as in mathematical expressions, but then the result is assigned to X as a string. For example

```
$v.hundredString := 10 * 10
```

will fill `hundredString` with the string "10 * 10".

The right side of an expression can be any string of characters that don't happen to also conform to the rules of mathematical expressions by enclosing them within double-quotes. For example

```
      $v.contractOfInterest := "0xef27da0…"
```

assigns `$v.contractOfInterest` the string of characters 0xef27da0…. Which can be used in parameters (for example, the **Contract** parameter of the Ethers Call module).

String values (apart from the empty string "") are coalesced to 1 in mathematical expressions.

Statement modules can fill both spaces in the first internal row of a race block. Which column in the second internal row is dictated by the expression of the last statement line: if it resolves to 0, the left column will activate. This may result in a few statement modules used in such way being of the form

```
      $v.dummy = $v.contractCallResult == "true"
```

to branch depending on the results of a comparison operator.

Parameters:

- **Statements**: A set of statements, one per line, of the form X = Y or X := Y, where X is of the form `$v.<name>` or `$g.<name>` and Y is an expression.

Result:

Whatever the expression of the last statement line resolved to.

Throw Error

Summary:

Halts the bot with an error message. Each module has an advanced property that dictates which row execution should jump to if an error is encountered. Of course you can just jump to the END row on error, and this is the default. The difference between doing that and throwing an error is: if a blocking Spawn Bot module spawns a bot that throws an error, that error propagates to the Spawn Bot module itself.

Parameters:

- **Error Prefix**: A message to add to the beginning of the error. If the previous module errored (i.e. an error occurred in a module and that module's **Error Goto** sent it to a row containing this module) then the error message from that module is appended to the error prefix. If there has been no previous row or there was no error executing it, the error prefix is the entire error message.

Result:

This module does not technically *have* a result- it never gets to finish because it throws an error which immediately halts the bot.

Timer

Summary:

Waits for either a certain period of time, or for a certain date and time, before completing.

Parameters:

- **Type**: Whether the timer is waiting for a certain *Date*, or for a certain *Time*.
- **Date**: Only valid when awaiting a datetime. The required format is one of
  - a time given in 24-hour format with an optional colon (e.g. 01:30 or 0130), which will wait for the clock to next strike that time. This is useful for daily tasks (such as in an auto-compounder).
  - in ISO8601 format with seconds optional (e.g. 2021-10-13T12:13).

  These default to local time.
- **Timer**: Only valid when waiting for a period of time. The required format is HH:MM:SS.MS filled from right to left with MS optional (e.g. 13:01 is 13 mins and 1 sec and 0.500 is 500 milliseconds.

Result:

0

User Input

Summary:

Waits for user input before proceeding. The inputs will be shown in the trace of the bot. User Input modules can fill both spaces in the first internal row of a race block. Which column in the second internal row is dictated by which button is pressed (okay or cancel).

Parameters:

- **Message**: This is the message that will be presented to the user explaining the purpose of the input.
- **Input Variables**: These specify the input fields. The format required is the same as the **Statements Before/After** advanced properties and the **Statements** parameter of the Statements module, except that here only local variables (`$v.<variable>`) are allowed. The expression on the right-hand side of the equal sign is mandatory, and is used to calculate the initial value of the input field. Whatever the user inputs into the field will be assigned to the local variable.
- **Okay Button Text**: The text displayed for the okay button.
- **Cancel Button Text**: The text displayed for the cancel button. If this field is empty, there will be no cancel button displayed to the user.

Result:

An object containing the input variables mapped to their resultant values.

Write File

Summary:

Writes text to a file.

Parameters:

- **File Path**: The path to the file to be written to. You can use the button below this field to open a file selector dialogue.
- **Is Append**: Whether the file will be overwritten or appended to.
- **Text**: [EXPRESSION] The text to write to file (reminder that you can right click on text-box parameters to see a bigger text box.
- **Is Blocking**: Whether Botchi should wait until confirmation or error. Note that with this unchecked, you won't be notified of what the error *was* if there's an error- execution will have moved on.

Result:

0

Await Fall

Summary:

Blocks until the price falls below (or equal to) the trigger.

Parameters:

- **Fall Trigger**: [EXPRESSION] Specifies the price trigger. This is can be given as a percentage (append %) which will resolve to that percentage of the price upon entering this module.

Result:

An object containing a "price" property which is the price that tripped the trigger. For trackers with low liquidity where one swap can swing the price quite dramatically, this will be noticeably different to the trigger price itself).

Await Fall Then Rise

Summary:

Based on the concept of a trail buy, this module waits until the price rises a certain amount after falling.

Parameters:

- **Minimum Fall**: [EXPRESSION] The minimum fall before the module starts checking for a satisfactory rise. This is can be given as a percentage (append %) which will resolve to that percentage of the price upon entering this module.
- **Then Rise**: [EXPRESSION] The condition for a satisfactory rise. This is always a percentage.
    - If **Rise Percent Of** is *Delta (change)*, the module will wait until the price rises a given percentage of the fall. For example, say the module is activated at a price of $1.00. The price then falls and hits the **Minimum Fall** of 10% (so the price is now $0.90). Say the **Then Rise** is 20%. If the price rises by $0.02 (20% of $0.10) from here the module would complete. If the price continued to fall to $0.80, the module would then be looking for a rise of $0.04 (20% of $0.20) before completing.

      A third check is always carried out after the **Then Rise** condition has been satisfied: if the price has jumped up too high- back above the price upon entering this module- then the module resets to wait for the **Minimum Fall** trigger again (which is not re-derived; it will be the same price as it was the first time). In our example, if a big transaction pushed the price from $0.80 to $1.10, even though the **Then Rise** condition is definitely satisfied the module will reset and wait for the price to fall to at least $0.90 before waiting again for the **Then Rise** condition.
    - If **Rise Percent Of** is *Price*, the module will wait until the price rises a certain percentage of the lowest price it has reached. For example, say the module is activated at a price of $1.00. The price then falls and hits the **Minimum Fall** of 10% (so the price is now $0.90). Say the **Then Rise** is 5%. If the price rises by $0.045 (5% of $0.90) from here the module would complete. If the price continued to fall to $0.80, the module would then be looking for a rise of $0.04 (5% of $0.80) before completing.

Just like in the case of the **Rise Percent Of** being *Delta (change)*, a third check is carried out. The difference between that and when the **Rise Percent Of** is *Price* is that here, a reset occurs if the price has jumped above the **Minimum Fall**.

- **Rise Percent Of**: Specifies whether the rise condition derives from the magnitude of the fall (the delta, or change, in price) or the lowest price reached.

Result:

An object containing a "price" property which is the price that tripped the module's completion.

Await Next Candle

Summary:

Waits for the current candle to complete.

Parameters:

- **Candle**: The time scale that determines which candle to await the finish of.

Result:

0

Await Rise

Summary:

Blocks until the price rises above (or equal to) the trigger.

Parameters:

- **Rise Trigger**: [EXPRESSION] Specifies the price trigger. This is can be given as a percentage (append %) which will resolve to that percentage of the price upon entering this module.

Result:

An object containing a "price" property which is the price that tripped the trigger. For trackers with low liquidity where one swap can swing the price quite dramatically, this will be noticeably different to the trigger price itself).

Await Rise Then Fall

Summary:

Based on the concept of a trail sell, this module waits until the price falls a certain amount after rising.

Parameters:

- **Minimum Rise**: [EXPRESSION] The minimum rise before the module starts checking for a satisfactory fall. This is can be given as a percentage (append %) which will resolve to that percentage of the price upon entering this module.
- **Then Rise**: [EXPRESSION] The condition for a satisfactory fall. This is always a percentage.
    - If **fall Percent Of** is *Delta (change)*, the module will wait until the price falls a given percentage of the rise. For example, say the module is activated at a price of $1.00. The price then rises and hits the **Minimum Rise** of 10% (so the price is now $1.10). Say the **Then Fall** is 20%. If the price falls by $0.02 (20% of $0.10) from here the module would complete. If the price continued to rise to $1.20, the module would then be looking for a fall of $0.04 (20% of $0.20) before completing.

      A third check is always carried out after the **Then Fall** condition has been satisfied: if the price has jumped up too low- back below the price upon entering this module- then the module resets to wait for the **Minimum Rise** trigger again (which is not re-derived; it will be the same price as it was the first time). In our example, if a big transaction pushed the price from $1.20 to $0.90, even though the **Then Fall** condition is definitely satisfied the module will reset and wait for the price to rise to at least $1.10 before waiting again for the **Then Fall** condition.
    - If **Fall Percent Of** is *Price*, the module will wait until the price falls a certain percentage of the highest price it has reached. For example, say the module is activated at a price of $1.00. The price then rises and hits the **Minimum Rise** of 10% (so the price is now $1.10). Say the **Then Fall** is 5%. If the price falls by $0.055 (5% of $1.10) from here the module would complete. If the price continued to rise to $1.20, the module would then be looking for a fall of $0.06 (5% of $1.20) before completing.

      Just like in the case of the **Fall Percent Of** being *Delta (change)*, a third check is carried out. The difference between that and when the **Fall Percent Of** is *Price* is that here, a reset occurs if the price has jumped below the **Minimum Rise**.
- **Fall Percent Of**: Specifies whether the fall condition derives from the magnitude of the rise (the delta, or change, in price) or the lowest price reached.

Result:

An object containing a "price" property which is the price that tripped the module's completion.

Technical Analysis

Summary:

This module contains a set of technical indicators (Bollinger Bands, MFI and RSI). The indicators that are enabled must all be satisfied at the same time for the module to complete. For logical patterns requiring *either* indicator A *or* indicator B to be satisfied, I suggest using race blocks.

Parameters:

- **Time Scale**: Most technical indicators make use of candle data in some way, and so a time scale is required. This parameter is set for all indicators. Note that Bollinger Bands, which are overlayed onto the chart, will only be visible when the corresponding time scale is selected on the chart.
- **Snapshot Mode**: For occasions where you might just want to get a reading of the current indicators, check this box. The module will calculate the indicator values and immediately complete.

- **MFI**: Whether to include the Money Flow Index indicator.
- **Frame Length**: How many of the most recent candle to include in the MFI calculation for the current candle.
- **Condition**: How the calculated MFI is compared to the **Target** to determine whether the indicator is satisfied. For example, lessThan would require the MFI be less than the **Target**.
- **Target**: The target MFI.

- **RSI**: Whether to include the Relative Strength Index indicator.
- **Frame Length**: How many of the most recent candle to include in the RSI calculation for the current candle.
- **Condition**: How the calculated RSI is compared to the **Target** to determine whether the indicator is satisfied. For example, lessThan would require the RSI be less than the **Target**.
- **Target**: The target RSI.

- **Bollinger**: Whether to include the Bollinger Bands indicator.
- **Frame Length**: How many of the most recent candle to include in the calculation for the current candle's Bollinger Band values.
- **Std Devs**: How many standard deviations away from the moving average to place the bands.
- **Trigger:** Determines when the indicator is satisfied. *Close High* is satisfied when the candle for the relevant time scale closes above the top band, while Pass High is satisfied as soon as the price moves above that top band. A similar distinction is made between the *Close Low* and *Pass Low* options.

Result:

An object containing the results of the included indicators at module completion. This will be some or all of:

- mfi
- rsi
- bollingerLow
- bollingerMidline
- bollingerHigh

Custom Modules (Plugins)

Custom modules are modules that can be added to Botchi to extend it in an easily shareable way (rather than forking just to add a desired built-in module). They can be written the same way as built-in modules, and have the same access to the backends. This includes calling backend functions to initiate swaps, transfers, etc. so please only use modules that you either wrote yourself or that have been vetted in some satisfactory way. Again: **this is a potential attack vector** for malicious actors to take advantage of, but only if you accept modules from those malicious actors.

Creating a Plugin

If there's demand, I can totally write up something explaining how to code a module but I think an examination of existing modules should suffice. I suggest starting at some of the simpler ones first (like Start or End) to get an idea of the basics.

Adding a Plugin

A plugin is simply a JavaScript file, which can be placed into the plugins folder (see the section on User Data Files). At the next startup, Botchi will prompt you to review and accept the new plugins. If the plugins are ever edited, Botchi will again prompt for your acceptance of these files at startup. After plugins have been accepted, they can be found together with the built-in modules, differentiated by their text colour.

Appendix A: Common Contract Addresses

This is only a select few of the addresses I have happened to have in front of me, for the chains I've happened to test. I've tried to include a USD stablecoin for each chain because having a tracker for the chain's native token vs fiat enables the value of all trackers that consist of a token paired with that native token to be displayed in USD.

Avalanche-C:

- WAVAX: 0xb31f66aa3c1e785363f0875a1b74e27b85fd66c7
- USDT.e: 0xc7198437980c041c805a1edcba50c1ce5db95118
- Pangolin
    - Factory: 0xefa94DE7a4656D787667C749f7E1223D71E9FD88
    - Router: 0xE54Ca86531e17Ef3616d22Ca28b0D458b6C89106
- Joe
    - Factory: 0x9Ad6C38BE94206cA50bb0d90783181662f0Cfa10
    - Router: 0x60aE616a2155Ee3d9A68541Ba4544862310933d4

Binance:

- WBNB: 0xbb4CdB9CBd36B01bD1cBaEBF2De08d9173bc095c
- USDC: 0x8AC76a51cc950d9822D68b83fE1Ad97B32Cd580d
- Pancake Swap V2:
    - Factory: 0xca143ce32fe78f1f7019d7d551a6402fc5350c73
    - Router: 0xbb4CdB9CBd36B01bD1cBaEBF2De08d9173bc095c

Ethereum:

- WETH: 0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2
- USDT: 0xdAC17F958D2ee523a2206206994597C13D831ec7
- USDC: 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48
- Uniswap V2:
    - Factory: 0x5C69bEe701ef814a2B6a3EDD4B1652CB9cc5aA6f
    - Router: 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D

Fantom:

- WFTM: 0x21be370D5312f44cB42ce377BC9b8a0cEF1A4C83
- USDC: 0x04068DA6C83AFCFA0e13ba15A6696662335D5B75
- Spooky:
    - Factory: 0x152eE697f2E276fA89E96742e9bB9aB1F2E61bE3
    - Router: 0xf491e7b69e4244ad4002bc14e878a34207e38c29
- Spirit:
    - Factory: 0xef45d134b73241eda7703fa787148d9c9f4950b0
    - Router: 0x16327e3fbdaca3bcf7e38f5af2599d2ddc33ae52

Appendix B: Running Botchi 24/7 in the Cloud

It may be impractical for you to run Botchi 24/7 on a local machine. Renting virtual machines in the cloud is an effective solution to this problem. Botchi does require a graphical desktop, but these can usually be installed on top of basic Linux virtual machines.

I have managed to get Botchi running on an Ubuntu instance in Microsoft Azure cloud. This was going to be a more in-depth tutorial but I've been locked out of my account until I provide a phone number, which I refuse to do. What I can offer right now is the basic steps I noted as I was setting it up:

1. Set up an ubuntu VM (Note: Botchi is not very resource heavy)
2. Add static public IP address
3. open SSH port on VM
4. On local machine: ssh -vvv -T <username>@<ip>
5. On local machine: ssh -i "<path to ssh private key>" <username>@<ip>
6. On VM: sudo apt-get install ubuntu-desktop
7. On VM:
    a. sudo apt-get remove xrdp vnc4server tightvncserver
    b. sudo apt-get install tightvncserver
    c. sudo apt-get install xrdp
    d. Create a file /etc/X11/Xwrapper.config containing allowed_users = anybody
8. Open RDP port on VM (recommend disabling this rule after each session)
9. Connect via RDP on azure portal

I am confident this is not the only cloud provider through which this can be done (another to consider is Google's Compute Engine platform), and I am sure there are better ways than RDP to connect. If someone can improve this or write their own tutorial, I'd be happy to link to them. I am far from an expert on cloud providers and managing virtual machines.