

Cryptick - A Digital Asset Portfolio Management Platform

— Final Report —

Charith Amarasinghe, Simon Spurrier, Julian Vossen, Dimitris Nikolaou,
James Griffiths, Konstantin Hemker.

{ca508, ss19616, jv1914, dn1217, jrg17, knh116}@doc.ic.ac.uk

Supervisor: Prof. William Knottenbelt
Course: CO530, Imperial College London

16th May, 2018

Contents

1	Introduction	1
1.1	Background	1
1.2	Motivation	1
2	Specification	2
2.1	Description	2
2.2	Use Case	2
2.3	Product Requirements	2
2.4	Risks	3
2.5	Product Deliverables	3
3	Design	3
3.1	Architecture Overview	3
3.2	Strategy Execution (Cryptick)	4
3.3	Front-end (Cryptock)	6
3.4	Exchange Interaction (xWrap)	8
4	Methodology	9
4.1	Development strategy	9
4.1.1	Project Management	9
4.1.2	Software Tools	11
4.1.3	Development Workflow	11
4.2	Subproblems	14
4.2.1	Cryptick	14
4.2.2	Front-end	17
4.2.3	xWrap Methods	18
4.2.4	Deployment	20
4.3	Testing	20
4.3.1	Testing Overview	20
4.3.2	Approach	21
4.3.3	Testing Modules	21
4.3.4	Testing Risks & Notable Omissions	23
4.3.5	Current Results	23
4.4	Challenges	24

4.4.1	Architecture	24
4.4.2	Wallet Management	24
4.4.3	Poorly Documented Exchange APIs	24
4.4.4	Testing Trading Strategies	24
5	Group Work	25
5.1	Project Milestones	25
5.2	Group Organisation	26
5.3	Meeting Log	26
5.4	Work Log	26
6	Final Product	26
6.1	Project Summary	26
6.2	Front-end Overview	26
6.3	Strategy Overview	28
6.4	Exchange Wrapper Overview	32
6.5	Next steps for Cryptick	33
	Appendices	40
A	Installation Instructions	40
A.1	System Requirements	40
A.2	Installing Dependencies	40
B	xWrap API Documentation	41
C	Strategy API	43
D	Test coverage	45
D.1	Strategy coverage	45
D.2	Front-end coverage	47
D.3	xWrap coverage	48
E	Meeting Log	48
E.1	Commit Log	50
E.1.1	Strategy	50
E.1.2	Front-end	50
E.1.3	xWrap	51
E.2	Facts and Figures	51

1 Introduction

1.1 Background

The Digital Asset Boom¹ Since Bitcoin was introduced in 2008, with a white-paper describing the world's first decentralised currency[12], its market cap has grown to over \$100 billion and spawned an asset class worth almost \$1 trillion by the end of 2017[3].

Today, there are thousands of digital assets, attracting media attention across the world as they challenge traditional ideas of money and exchange. However, the market for digital assets remains highly volatile and relatively untouched by government regulation.

Institutional and Retail Adoption With the unprecedented increase in value of the digital asset market during 2017, retail investment grew at an astonishing rate. Institutions, keen to meet this demand and gain exposure to a volatile and lucrative asset class in an era of dormancy in traditional markets, are starting to take notice.

1.2 Motivation

A Challenging Market The market for digital assets remains highly fragmented - a large number of online only exchanges, operating in various jurisdictions, each offering a different subset of trading pairs. Very rarely do two exchanges list the same price for a given trading pair because of differences in fee structure and trading restrictions.

In most cases, despite the billions of dollars of daily trading volume, these markets are not subject to traditional financial regulation. As a result exchanges are run with varying levels of competence and are often subject to price manipulation.

Given these conditions, a need for a platform providing controlled and uniform access to these markets has arisen. This makes the exchange of digital assets more straightforward and less risky, optimising for price and speed.

Investment Industry Investment funds not only need to trade effectively but must maximise returns. This may be on a long term basis, where funds should be allocated efficiently to minimise unsystematic risk and gain exposure to outperforming assets, or a short term basis, using optimal high-frequency trading algorithms to capture immediate profits.

Institutions require a reliable tool for portfolio allocation and trading strategy testing in the market for digital assets.

Miners and Infrastructure Cryptocurrency miners drive the ecosystem and generate and hold large amounts of crypto-currency. To stabilise their portfolio and grow their digital assets they need automated portfolio balancing and a robust trading system that can deal with large volumes of trades without impacting the market (trade slicing).

Other infrastructure providers, such as cryptocurrency payment services, need to manage their digital asset holdings in a similar way, and be able to quickly find liquidity to serve customer's needs whilst minimising risk.

The Consumer For digital assets to experience true mass adoption there must be a consumer level trading application for managing and investment a cryptocurrency portfolio, analogous to online banking services today. Consumers must be able to optimise and control their own portfolio through a user-friendly interface.

¹In this section, and throughout this final report, some paraphrased and verbatim content from the first[21] and second[22] reports is included without further citation.

2 Specification

2.1 Description

The *Cryptick* platform is a portfolio management tool for cryptocurrencies. It incorporates an algorithm to auto-diversify a portfolio in real time and respond to market events and market microstructure. The platform provides a unified interface to different exchange to overcome the fragmented market. Users are able to monitor the performance of their portfolio through a web dashboard and tweak parameters to reflect their desired portfolio composition. More advanced users are also be able to write their own trading strategy and test its functionality using a backtest environment [21].

2.2 Use Case

There are two major use cases for the *Cryptick* platform:

Cryptocurrency Portfolio Diversification The first type of user is one that typically has a constant stream of income in a cryptocurrency (e.g. a mining company) and wants to diversify their holdings into a variety of other currencies with minimal effort to reduce cluster risk. *Cryptick* leverages a capital allocation theory named *Markowitz Portfolio Theory* that has been heavily used in the financial markets over the past half century. The theory argues that investors can optimise the risk-return ratio of their portfolios through diversifying across different assets (i.e. cryptocurrencies), benefiting from the imperfect correlation between those assets. Based on this theory, the user provides *Cryptick* with their current portfolio and the platform suggests multiple optimal portfolio mixes which which the user can select.

Cryptocurrency Trading Strategy Backtesting The second type of user is typically a more sophisticated cryptocurrency investor who wants to have an environment to backtest and execute their trading strategies. This customer will purchase the software package as a virtual appliance and install it on a host with local network access. An administrator will then configure the software with API keys for cryptocurrency exchanges, wallet addresses and the currency types allowed for trade. The customer (or their authorised employees) will then be able to view their portfolio balances, choose a target portfolio from choices presented by the software, and start a trading process that would trade towards the selected target.

2.3 Product Requirements

All of the minimum requirements set at the onset of this project were met. Additionally, the majority of stretch goals were also met. What follows is a list of the requirements met, the last two bullets of which state the stretch goals that were met [21].

- A strategy for portfolio diversification, grounded in *Markowitz Portfolio Theory*, parametrised by a client-specified risk level, and tested on historical data.
- A trading system which interfaces with live exchanges to receive market data and issues appropriate orders determined by the strategy to an exchange mock which allows paper-trading.
- An interactive browser-based dashboard for visualisation of the current state of the portfolio and strategy, and for adjusting parameters of the strategy through UI controls.
- Support for data from cryptocurrency exchanges Bitfinex, GDAX and Poloniex.
- Support cryptocurrencies Bitcoin, Bitcoin Cash, Litecoin, Ethereum, ZCash and Monero.
- Implement a trading architecture which supports other strategy types - for example, arbitrage strategies through the unified exchange interface.
- In-application strategy backtesting facilities.
- Central management and logical partitioning of exchange wallets.

2.4 Risks

A high level view of the risks faced in this project are listed below:

- Markowitz Portfolio Theory was developed based on traditional financial markets which are significantly less volatile than cryptocurrencies. It is possible that digital asset market behaviour does not conform to traditional financial modelling posing a moderate risk to the project - deeper research is required [7].
- The trading and simulation features depend on external data. Hence, there is a moderate risk of external data sources becoming unavailable.

2.5 Product Deliverables

- An application that is supposed to be long-running in a headless environment, which fetches real-time market data, evaluates this data using a trading strategy, and sends orders to virtual exchanges accordingly.
- A browser-based interface for the above described application which provides inputs for user defined strategy parameters, a visualisation of the portfolio's state, and trade execution monitoring [21].

3 Design

3.1 Architecture Overview

The broad function of the *Cryptick* platform makes system architecture critical in building an extensible and maintainable product. The Separation of Concerns principle was applied to determine the three primary modules of the platform: exchange interaction (xWrap), strategy execution (Cryptick) and front-end (Cryptock).

For overall application structure, a micro-service architecture was selected, as the exchange interaction and strategy execution components best benefited from two concurrency patterns (coroutines and threading) that would not integrate together in a single monolithic structure. Though the front-end could be hosted by one of the other components, it was decided to make it a dedicated micro-service in order to simplify development. Further reasoning behind this choice is discussed in Section 4.4. An initial high-level diagram for the system architecture is shown in Figure 1.

Strategy Execution (Cryptick) Cryptick is a framework for writing and testing Python-based cryptocurrency trading strategies using a simple, easy-to-use API. Individual strategies run inside dedicated processes and have their lifecycle managed by a main Cryptick process. Strategies are able to create and update UI components visible from the front-end through an API layer made available by Cryptick (Table 6 - Appendix B), as well as call the xWrap API to connect the strategy execution to the available cryptocurrency exchanges (Table 5 - Appendix B). The user of the platform can choose either to use provided strategies (e.g. the efficient frontier strategy) or write their own strategy using the APIs.

Front-end User Interface (Cryptock) Cryptock is a React-based web interface that displays the status and operation of running Cryptick strategies and enables wallet-management. It connects to both back-end components (Cryptick/xWrap) to post strategy information, wallet balances and order streams. Information is available to the user at different levels:

- Strategy-level: information per run
- User-level: aggregated information about all strategies and wallets
- Market-level: general cryptocurrency market information

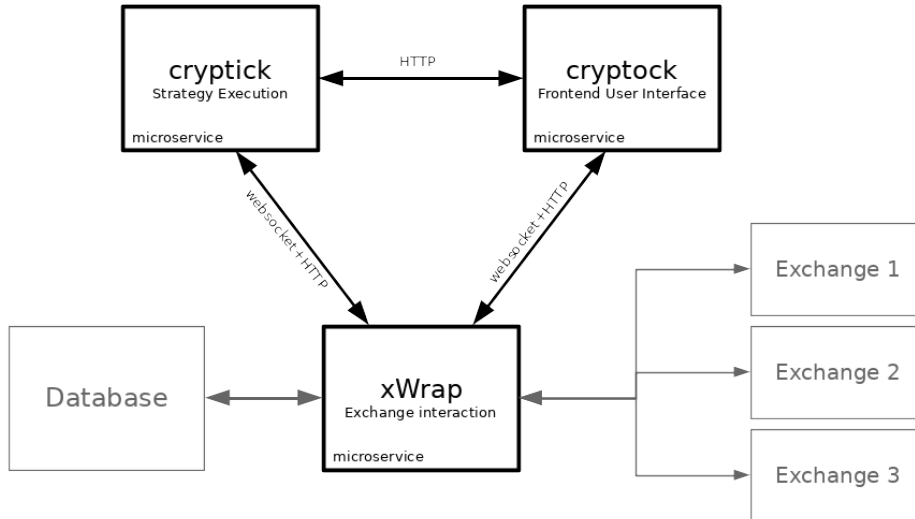


Figure 1: High Level Design Diagram

Exchange Interaction (xWrap) xWrap builds an abstraction layer to multiple cryptocurrency exchanges and manages the lifecycle of buy/sell orders and the collection of pricing data. It additionally provides a simulated trading mode (termed ‘paper’ trading) atop supported exchanges in order to test strategies in a sandbox environment without risking real money.

3.2 Strategy Execution (Cryptick)

Requirements The following requirements for the strategy execution engine are derived from the project requirements:

- Support execution of different trading strategies
- Communicate with the exchange interface (xWrap) to receive exchange data and manage orders
- Simulate the execution of trading strategies using historical data (backtesting)
- Expose strategy management API to the front-end

In addition to the above requirements, the ease of operation and usability without the front-end for strategy developers was prioritised.

Design Overview Figure 2 provides a high-level overview of the strategy runner architecture. A server process provides an HTTP API for the front-end and creates processes for executing strategies. Each of these instantiations has a central communication node - the message hub - as well as communication interfaces and the strategy logic itself.

Internal Dataflow The message hub serves as distributor for all data in the trading system and is handled by each strategy block. The exchange wrapper can connect to the message hub through an adapter and send or receive messages through a fixed message format. The chosen format of messages exchanged over the hub is similar to JSON-RPC 2.0, which specifies two message formats.

- Notification/Call: specifies ‘method’ and ‘response_method’ fields.

```
{
  "mhubrpc": "0.0.1",
```

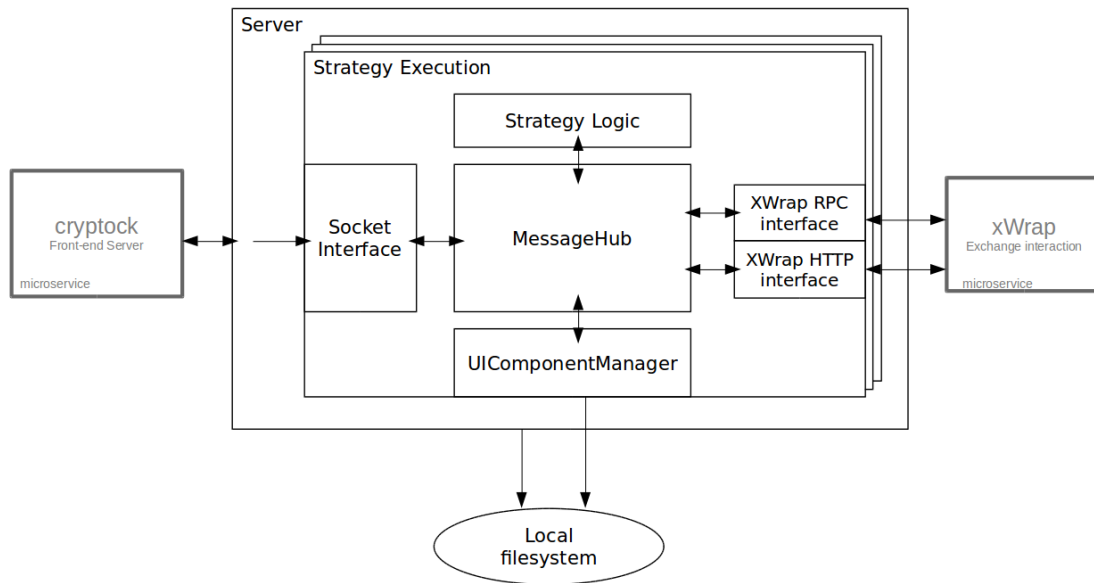


Figure 2: Strategy runner design

```

"method": "exchange.order",
"params": ["bitfinex", "eth_usd", 10000],
"response_method": "1231709"
}
  
```

- Result/Error: specifies 'response_method' from initial call in 'method' field. Example:

```

{
  "mhubrpc": "0.0.1",
  "method": "1231709",
  "result": "success"
}
  
```

As shown in the above example of a trade order being sent, hierarchical method names of the format 'provider.method' were used. The *provider* of a message is usually the name of a data provider (e.g. 'twitter', 'exchange'). The *method* would be the action executed on the *provider*, such as sending off an order.

Using a central message hub enables auditing the information flow. For this purpose, the strategy runner provides a web dashboard, which shows all messages that went through the message hub for each strategy run. The dashboard is a small single-page application based on AngularJS, which is hosted by the same server which provides the HTTP API as described later in this section. A screenshot of the dashboard is shown in figure 14 in the results section.

Strategy Logic All trading logic for a strategy is encapsulated in a single class. On instantiation, the strategy class registers callback functions for messages of interest with the message hub. Within these callbacks, the strategy can react to messages such as exchange events, user actions or periodic self-scheduled triggers. Such reactions can range from simple internal state updates to placing orders by sending appropriate messages to the message hub.

This strict encapsulation of the trading logic makes it easy to implement new strategies and support the execution of arbitrary trading strategies. Moreover, encapsulating the trading logic is also required for backtesting, where the exact same trading logic needs to be simulated using historical exchange data.

UI Components UI components establish a bidirectional communication between a user and a strategy run. These components consist of a format specification and a data array and can be rendered by the front-end according to a defined set of supported component types. A strategy can create and update components by posting messages to the message hub. The UI component manager has registered callbacks for these messages and writes the component data to the local filesystem. For each component, a *JSON* file holds the format specification, and a *CSV* file stores the data itself. We chose simple files over database management systems according to the design priority of low operational complexity to enable strategy developers to run Cryptick easily. When users log to components, they can directly inspect the data in *CSV* files. Advantages of DBMS diminish in this setting where only a single writer instance (`UIComponentManager`) writes to the data for each run, the total data volume is low and the data structure is simple.

For the communication from the user to the strategy, the front-end can post data to individual components. These posts are then converted into valid message hub messages and forwarded via the message hub socket interface.

CLI As users should be able to develop new trading strategies without running the front-end server, the strategy runner comes with a command-line interface (CLI) itself. Using this CLI, the user can run single strategy instantiations by specifying the mode and initial conditions. For example, the following command will start a backtest run of a strategy called `HodlStrategy`, for a specified time, with a initial balance of 10000 USD:

```
python -m cryptick.run HodlStrategy -m backtest --gdax-usd 10000 \  
-s '01-01-2018' -e '16-05-2018'
```

Using the CLI, a strategy will run synchronously in the console.

API Server Given the requirement of exposing the strategy execution management to the front-end, the strategy runner provides an HTTP API server to manage strategy runs. With this API, the front-end can request available strategies, instantiate and terminate runs, and communicate with the running strategies. The full API documentation can be found in the appendix.

As opposed to the CLI, the API server will start strategies asynchronously, by creating separate processes for each strategy run.

Backtesting For backtesting, a run process is instantiated with an exchange simulator instead of the `XWrapInterface`. Hence, the exchange simulator provides callbacks for all the messages which would normally be send to `xWrap`. The simulator requests historical exchange data from `xWrap` to simulate order matching and determine the corresponding prices.

3.3 Front-end (Cryptock)

Design Overview The main design of the front-end is shown in Figure 3 and shows both internal dataflows through the front-end as well as the building blocks required for communication with the strategy and the exchange wrapper. The control and data flow of the front-end is discussed in the following section and a more detailed discussion about implementation considerations is included in section 4.

Reducer All state management happens through the reducer, which keeps track of the state of the application and specifies how the application's state changes in response to actions. This is just a composition of pure functions from state-to-state. The initial state of the application is stored in the reducer as an immutable object whose members are aligned with the format expected by the `xWrap` and front-end API.

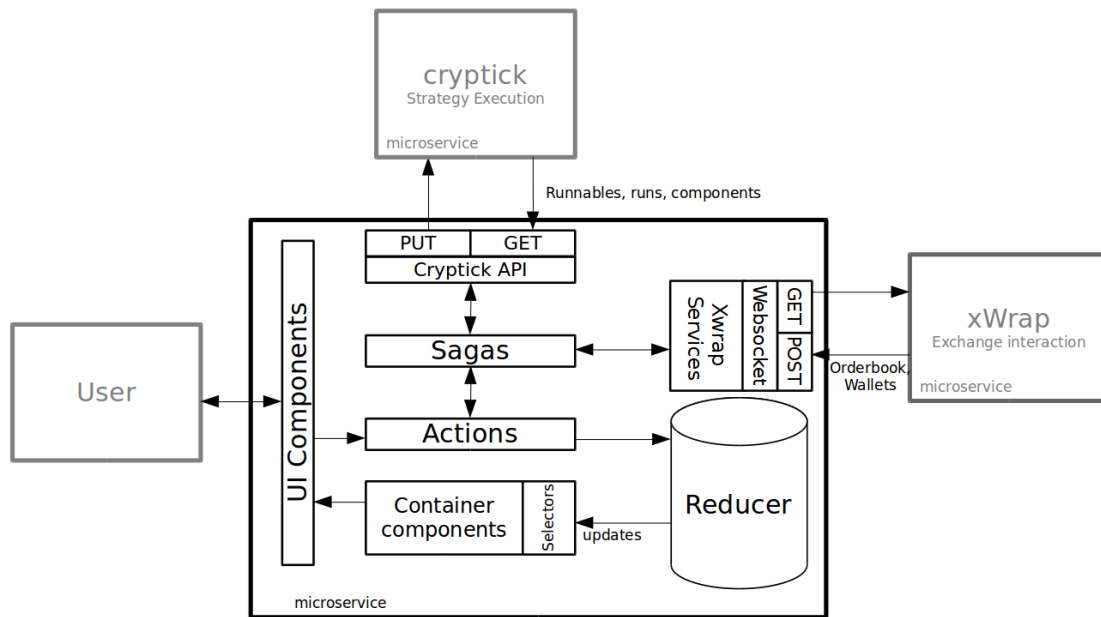


Figure 3: Front-end design

Actions Fundamentally, all communication throughout the application occurs through actions. State updates may only occur in response to actions, whose types are matched on to produce the desired effect. Actions can be triggered by any part of the application upstream of the reducer, such as upon loading a UI component, on receipt of websocket communication, or by user interaction (e.g. *onClick*, *onHover*, etc.). Actions consist of types and payloads of information, used for updating the application's state, or intercepted by middleware (in our case, *sagas*) to perform intermediate transformations or effects (most notably, *AJAX* requests). An example of an action would be adjusting the portfolio weights on the efficient frontier strategy using the sliders component. As soon as this action is dispatched, the reducer, which is listening for all actions, matches on this action type and updates the state of the application corresponding to its payload. Actions within the **Redux** architecture are the only source of information for the reducer, leading to consistent state management with a single source of truth.

Container Components and Selectors Container components manage the data used by the UI. Using container components separated from the rendering (UI) components separates the process of data-fetching and rendering for each component. Any state updates broadcast by the reducer are heard by container components, which map the state of the application (and in our case, router state) to the required properties of the UI component. The main upside of separating rendering and container components is that components can be re-used more easily and the rendering component is not concerned with state management. Containers themselves use selectors to efficiently compute derived data from application state. This efficiency is achieved through memoisation and functional composition, allowing the functions to be called on every state update, but only being fully evaluated if the inputs have changed. The `reselect` library was chosen for selectors.

UI Components The container components map the state of the application to the properties of the rendering components that were built using **React** in conjunction with Google's **Material-UI**. All chart representations were rendered using `react-chartjs-2`. Rendering components also provide an interface to the actions available to the user (see above).

Sagas *Sagas* are a relatively old concept from the database world to manage state-updating transactions, but novel in the front-end development context. For our application, *sagas* act as 'database' middleware, sitting just in front of the reducer. These listen for particular action types which are used

to communicate requests for API-calls and fork threads to perform these calls asynchronously, or sit waiting for socket communication, before dispatching actions denoting API success or failure to the reducer. This isolates all network side-effects to this portion of the app. `Redux-saga` is employed as an implementation of the *saga* concept in our application.

API Interaction API interaction is done with the help of two front-end servers. Each front-end server serves the application itself and associated assets, and talks to the back-end services. One front-end server serves assets for the live-trading version of the application, and communicates with the live-trading `xWrap` service. The other does the same but for paper and backtest trading. Both of these server instances speak to the same instance of the Cryptick strategy runner micro-service, as that instance manages all trading modes of strategies. These front-end server instances act as amalgamation of proxy servers and static-asset servers, built in a fairly simple manner with `express`.

3.4 Exchange Interaction (`xWrap`)

The project specification calls for a trading system which is capable of fetching live data from cryptocurrency exchanges and virtually trading according to real-time market conditions in paper trading mode. Exchange API formats can differ significantly and offer different levels of functionality to users, and a unified abstraction would make extending the system in the future much easier.

The `xWrap` service acts as a local cryptocurrency exchange, transparently handing orders to the GDAX, BitFinex and Poloniex APIs while providing high level management and error recovery capability to client applications. It is built atop a single high performance asynchronous event loop that services network connections, handles events and carries out scheduled tasks. What follows is a discussion of key design concepts; full documentation is bundled with this report and made available in the appendix.

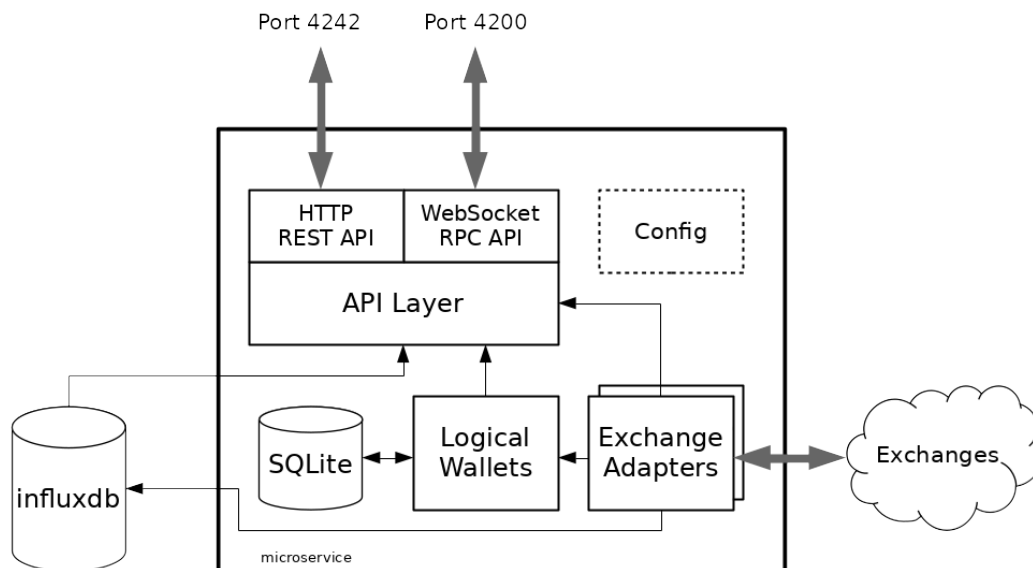


Figure 4: Simplified Diagram of `xWrap`

Design The component diagram for the `xWrap` service is shown in figure 4. The service is packaged as a Python module and requires CPython version 3.6 or later on MacOS or Linux. A brief description of the various components follows.

API Layer The API layer is a set of functions which expose the core functionality of `xWrap` to client services. Consult the `xWrap` documentation in the appendix for a full list of supported API methods. These functions are implemented in the `service.ExchangeWrapper` class.

HTTP REST API The HTTP API module translates incoming HTTP requests into API calls on the API layer. The HTTP server is provided by the `sanic` library for Python and the API stubs are specified in the `http_server` module.

WebSocket RPC API The RPC API module translates incoming RPC requests into API calls on the API layer. This is implemented by the `rpc.RpcInterface` and `wsock_server.WebsocketServer` classes.

Config The Config module provides a global configuration to all other modules. Variables can be set either through a JSON configuration file, or in certain cases, overridden by OS environment variables. Example configuration variables include ports for serving APIs, security credentials for exchange API access and tradeable currencies. The Config module allows for flexible configuration of the xWrap service when deployed.

Exchange Adapters The exchange adapters form the core of the exchange interaction tasks. These are implemented as Python modules in the `exchange` package which abstracts the Web APIs for different providers into a unified set of Python methods. The modules also spawn and maintain recurring tasks that poll exchanges for certain types of data. For exchanges delivering Level 2 market data, exchange adapters additionally subscribe to push APIs to maintain real-time exchange order books.

Logical Wallets Logical wallets allow the logical partitioning of exchange wallets into wallet sets. This allows users to sandbox portions of funds held on exchanges to a single strategy and ensure that strategies are limited to seeing and using only those allocated funds. This functionality is implemented in the `models.VirtualWallet` and `logical_wallet.LogicalWalletManager` classes.

4 Methodology

4.1 Development strategy

4.1.1 Project Management

To ensure the timely completion of the project, Agile software development was used. Agile specifies an iterative and flexible approach to the development of lean software that requires the regular input of members of all teams.

In particular, two key Agile frameworks called Scrum and Kanban were employed. Scrum focuses on breaking down the development work into blocks and completing them in an iterative fashion within a set time limit. Each iteration is called a Sprint and the time of each Sprint is set by the Scrummaster (in our case the team leader)[16]. We chose small iterations of two weeks as this size of time frame allowed us to be more adaptive. Scrum also entails regular quick meetings to track progress. Therefore, we met on average twice a week for a total of four times per sprint. The Kanban framework places its focus on the visualisation of the blocks that are created on every sprint and this was very valuable in tracking progress within each sprint.

Agile was chosen mainly due to its adaptive and iterative nature. Moreover, some team members previously worked on projects using these techniques. The implementation of Agile was made on Asana, a software that allows us to easily use Scrum and Kanban. Figure 5 is a snippet of the *Cryptick* Kanban board on the 22/02/2018.

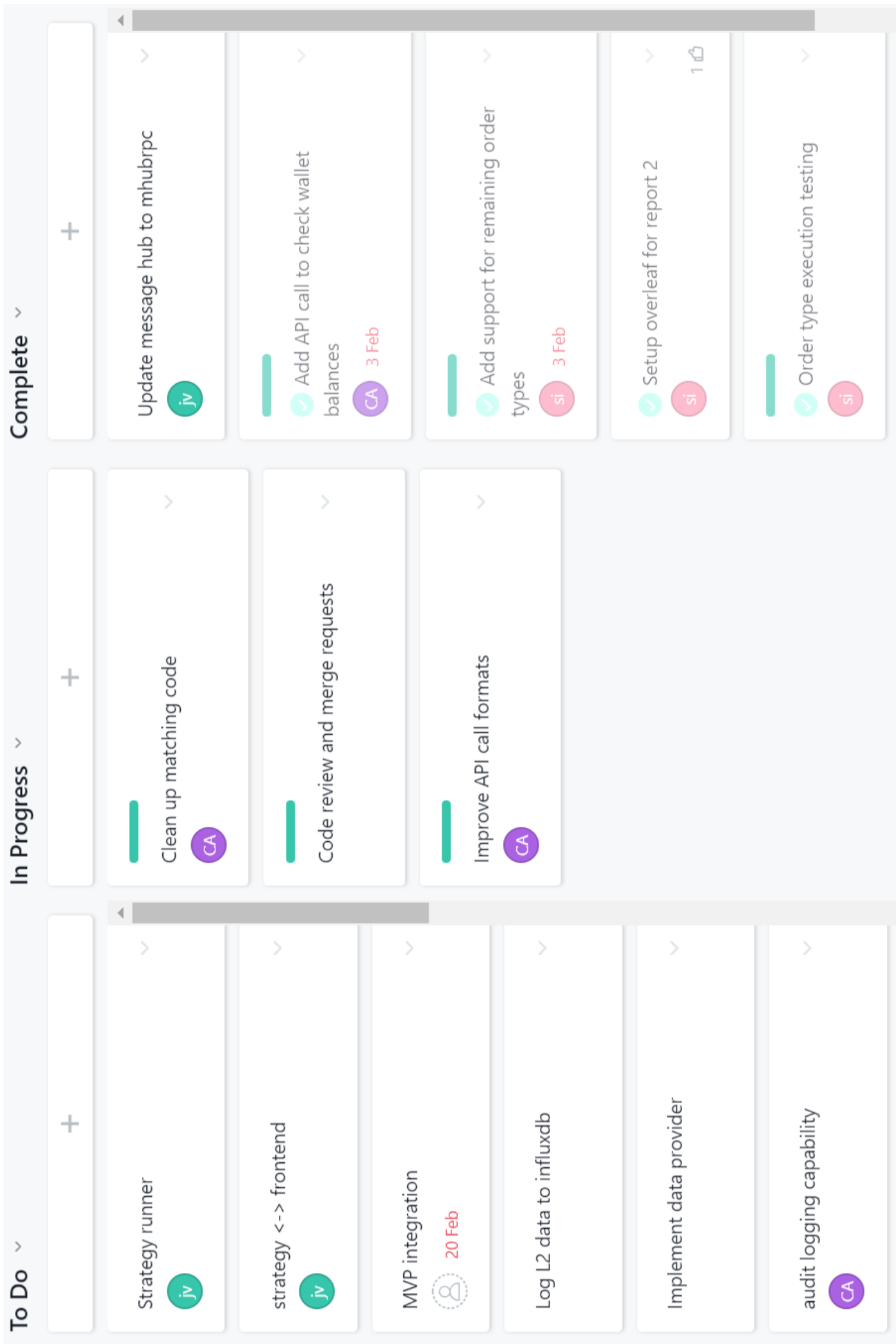


Figure 5: Kanban Board 22/02/2018.

4.1.2 Software Tools

Programming Language The choice of programming language, unless constrained by application specific requirements, is largely determined by the skill-base of the organisation². For this project, the ability to easily interface with third-party web services and perform fixed-precision computation on large datasets is critical. Python was chosen as the main back-end development language due to the team’s past experience and for its strong library support in the above areas.

Computer Resources As the end-product is a web application, the only external resources required are servers for hosting the application and any supporting services. No special compute or storage requirements were needed and the Department of Computing’s Apache CloudStack virtualisation platform was used to create VMs. The Cryptick platform is hosted on 3 VMs, one for production, one for development and one for the time-series database. A fourth VM provides a GitLab CI Runner for Continuous Integration.

Software Packages The final product (section 6) depends on a number of third-party libraries and services. Associated license requirements can pose significant limitations on commercial use. Open-source dependencies should thus be licensed under a BSD-style or similarly permissive license that permits free commercial use without any restrictions on the end-product. Web services to be used, such as cryptocurrency exchanges, will be priced on a per-transaction commission or subject to volume-pricing agreements. These costs will be unavoidable for any participant in the ecosystem.

Version Control and Code Review Git has been chosen as the version control system for this project. This is largely due to the wide familiarity with Git within the team. Alternative tools such as Mercurial and SVN pose no specific advantage within the scope of this project. GitLab and Imperial’s GitHub Enterprise were evaluated for hosting a co-ordinating repository, code review functionality, a bug tracker and support for continuous integration. GitLab was selected due to simpler CI integration with GitLab Runner.

4.1.3 Development Workflow

Workflow A structured workflow forms an integral part of managing day-to-day collaborative software development. It primarily guides the effective use of version control software, issue trackers and code review processes to ensure the creation of high quality, readily deployable code. Given the size of the team, a branch-review-merge strategy is proposed² as it maps smoothly with the built-in code-review functionality on GitLab[20]. In this model, a new feature is developed in its own branch by one or more developers. When a feature is completed, the primary developer will raise a merge request to the upstream branch and resolve any merge conflicts that have developed. The subsequent diff will be brought up for approval by the maintainer of the upstream branch. A merge request that passes this review process will be merged upstream. Figure 6 shows an example review of a merge request using GitLab.

Continuous Delivery and Continuous Integration CI/CD practices enforce rapid releases of code to production environments and incremental changes to software. Such a system is adopted in-line with modern best practice in order to achieve highly visible iterative progress. Figure 7 shows an extract of the GitLab CI system for the xWrap process. Branches failing tests or missing coverage need to be fixed before they can be merged upstream.

Development Practices Various project-specific practices such as Git workflow, code style, commit strategy, documentation requirements and test practices are documented and maintained within a central docs repository on GitLab.

²Certain assertions in this section are based on the teams experiences or observations from industry.

The screenshot displays a GitLab merge request for the project 'crypto-portfolio-mgmt / cryptick'. The interface includes a top navigation bar with icons for home, search, and notifications. Below the navigation bar, there are tabs for 'This project', 'Search', and '6/9 discussions resolved'. The main content area is divided into two sections: a code diff and a discussion.

Code Diff:

```
19 + PID_FILENAME = '.pid'
20 + PORT_FILENAME = '.port'
21 +
22 + app = Flask(__name__)
23 +
24 + def new_run_id(name):
25 +     """Generates a new unique id for a run by appending a number to its name"""
26 +     for i in range(10000000):
27 +         run_id = '{}_{}'.format(name.lower(), i)
28 +         if not os.path.exists(os.path.join(STORE_DIR, run_id)):
```

ca508 @ca508 commented a month ago
This can cause a race condition?

Something like:

```
with lock:
    p = os.path.join(STORE_DIR, run_id)
    if not os.path.exists(p):
        os.makedirs(p)
    return run_id
```

?

Julian Vossen @jv1914 commented a month ago
Yeah, well spotted. Will fix.

Julian Vossen @jv1914 changed this line in version 5 of the diff a month ago

Figure 6: GitLab Merge Request

The screenshot shows the GitLab CI Pipelines interface for the project 'crypto-portfolio-mgmt / exchange_wrapper'. The top navigation bar includes 'Project', 'Repository', 'Issues', 'Merge Requests', 'Pipelines', 'Wiki', 'Snippets', 'Members', and 'Settings'. The 'Pipelines' tab is active, showing a list of pipeline runs. The left sidebar contains navigation options: 'All', 'Pending', 'Running', 'Finished', 'Branches', 'Tags', 'Pipelines', 'Jobs', 'Schedules', 'Environments', and 'Charts'. A 'Run Pipeline' button and a 'CI Lint' button are visible. The main content area displays a table of pipeline runs with columns for Status, Pipeline, Commit, Stages, and a refresh icon.

Status	Pipeline	Commit	Stages	
failed	#49878 by latest	feat_live_ord... -> 4da0b51d very theoretically poloniex should work	failed	01:00:04 about 5 hours ago
passed	#49862 by latest	v0.5 -> e7831655 Dirty fixes	passed	00:02:50 about 12 hours ago
passed	#49833 by latest	v0.5 -> c6c47daf Fix nasty notifications - bad blinker, b...	passed	00:02:52 about 13 hours ago
failed	#49825 by latest	feat_live_ord... -> 95f13ae3 bitfinex needs debugging	failed	01:00:05 about 13 hours ago
failed	#49810 by latest	feat_live_ord... -> c0f7f4d7 bitfinex update formatting and subsc...	failed	01:00:05 about 14 hours ago
failed	#49789 by latest	feat_live_ord... -> ccbe0d9b start bitfinex livebook version	failed	01:00:04 about 17 hours ago

Figure 7: GitLab CI Pipelines

Internal Communications Team communications were largely carried out through the Slack platform. In addition to providing convenient web and mobile applications, Slack allowed the creation of ‘Channels’ for each workstream. This enabled interested parties to passively monitor design discussions on parallel streams as well as to keep a log of all discussions. The support for chat bots also enabled integration with third-party services such as GitLab and Sentry.IO.

Metrics and Error Reporting As large parts of the Cryptick platform are long running processes, bugs can cause errors or unexpected behaviour at any time, and developers might not always notice them. As such, error reporting and proactive monitoring are key to identifying issues and fixing them. Two major strategies were adopted to address this requirement: The third-party Sentry.IO service[5] is used to log and alert on runtime exceptions and the Prometheus monitoring system[14] is used to instrument key sections of code to monitor for critical variables such as response times, error rates and message counts[15]. Figure 8 shows the dashboard of the Sentry.IO web application and Figure 9 shows graphs of some of the metrics monitored on the xWrap service.

Documentation APIs for the *Cryptick* platform are documented using Python Doc-Strings and text files in *Markdown* format. The Python `sphinx` library is used to generate HTML and PDF documentation from these sources. The reference documentation attached in the appendix to this report was generated in this manner.

4.2 Subproblems

As mentioned in section 3, the main workstreams of the project were broken down according to its subproblems - the strategy, the exchange wrapper and the front-end. On top of these general subproblems, the main overarching subproblems are discussed in the following, namely *integration* of the three workstreams and *deployment* of the Cryptick platform.

4.2.1 Cryptick

Portfolio Diversification Strategy The Markowitz Portfolio Theory was used as the portfolio diversification method. This theory is very commonly applied to traditional financial portfolios (e.g. bonds, stocks) and as such it was trusted by the *Cryptick* team.

Input The strategy receives two inputs. The first one is the Open-High-Low-Close data of the supported cryptocurrency on a 5 minute interval over the past 180 days. The strategy does not need to communicate with any exchanges as the xWrap component of *Cryptick* collects and adjusts the data. Therefore, the strategy communicates with xWrap and collects the data needed. The second input is from the users of the platform. When diversifying their portfolio the user is asked to specify a risk/return trade-off.

Data Processing The strategy then processes the data and generates 10,000 portfolios with random weights over the cryptocurrencies selected. For these 10,000 portfolios an expected return and a variance is calculated based on the historic prices. The return of a portfolio is calculated as the weighted average of the historic returns. The portfolio variance is calculated by taking into account portfolio weights and the correlation coefficient between the portfolio assets [10]. The formula for portfolio variance can be found in the equation below.

$$Variance = \sum_{i=1}^N \sum_{j=1}^N (w_j * \rho_{ij}) * (w_i)$$

(1)

w_i :weight of asset i
 N :number of assets in portfolio
 ρ_{ij} :correlation coef. between asset i and j

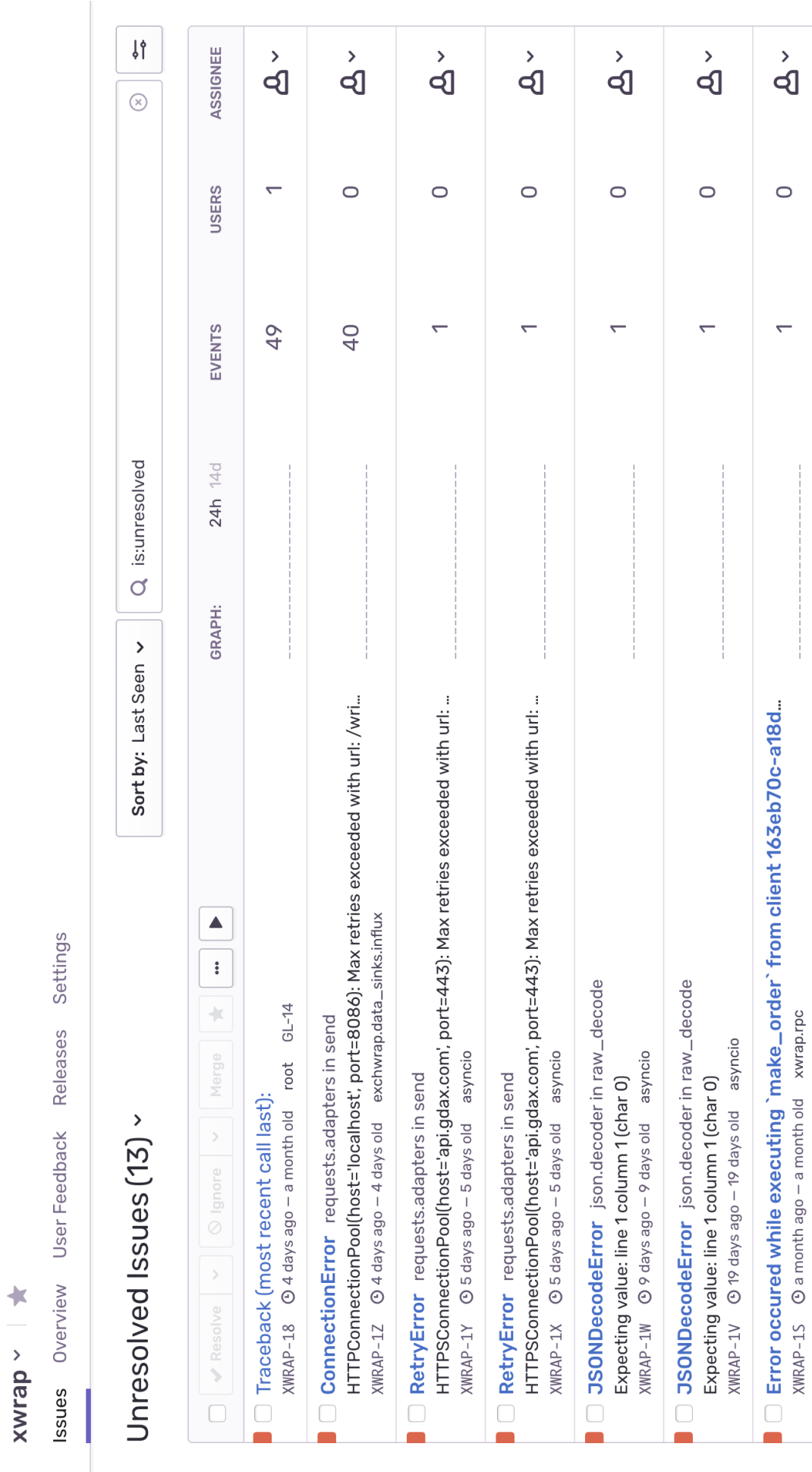


Figure 8: Sentry.IO Dashboard



Figure 9: Prometheus Stats for xWrap Rendered in Grafana

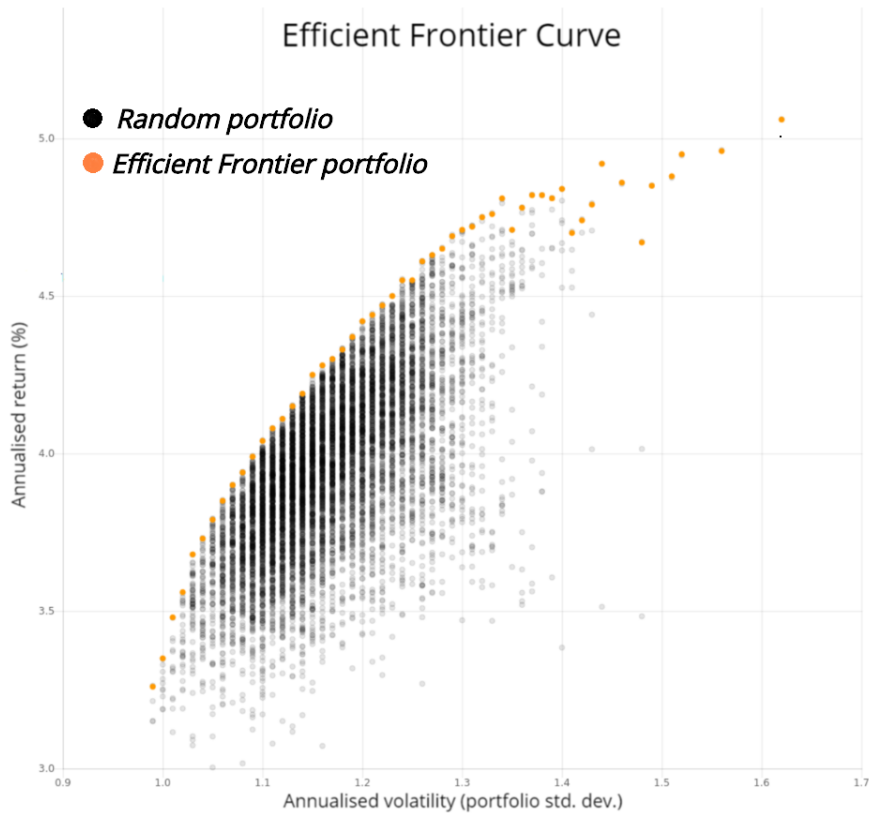


Figure 10: The Efficient Frontier Curve.

For each random portfolio, a Sharpe ratio is calculated which reflects the return earned in *excess* of the risk free rate. As a risk free rate, a semi-annual return of 3% was used as that appears to be the rate of US 10Y bonds [1]. It follows that the portfolio with the highest Sharpe ratio should be the one selected assuming that the investor is willing to take any level of risks. The formula for the Sharpe ratio is given below [11].

$$\text{Sharpe ratio} = \frac{(\bar{r}_p - r_f)}{\sigma_p} \quad (2)$$

r_f : risk free asset
 \bar{r}_p : mean portfolio return
 σ_p : portfolio standard deviation

Output Out of the 10,000 portfolios, the ones with the highest return for every level of risk are determined. These portfolios are said to create the efficient frontier as it would be inefficient for investors to choose any portfolios other than these. An illustration of the random portfolios and the *Efficient Frontier* can be viewed in Figure 10. The strategy writes all these portfolios that make up the *Efficient Frontier* together with their expected return, variance and Sharpe ratio into a UI component which is then displayed on the front-end. It also adds the covariance matrix between all selected cryptocurrencies together with all single-asset portfolios for recomputing metrics for user-specified portfolios.

4.2.2 Front-end

The methods used to meet the requirements of the front-end application are described below. While these are to an extent touched upon by the content of 3.3, we here go into more detail as to the methods used to overcome both intellectual and technical problems, and alternatives that were considered.

Virtualising the UI One of the significant challenges faced by the front-end has been maintaining responsiveness in the face of large and unpredictable volumes of data from back-end sources, both in stream- and batch-form. These two forms of data-source have different requirements for maintaining responsiveness: streams require being able to handle unpredictable frequency of messages effectively, while batch data requires being able to process large volumes of data and update the UI without resulting in noticeable slowdown. Both of these requirements are satisfied through a combination of virtualising the UI and batching updates to the actual DOM based on a 'virtual' DOM. The displaying of large quantities of batch data is mitigated through the use of virtual tables, which render only as much data as is visible, calculated according to scroll position.

Unified Theming While CSS lays the foundation for web-UI styling, operating on a team-project requiring consistent theming while also having granular control over how the theme is expressed in particular components exposes its inadequacy as a way of managing styling in itself. To remedy this, we used the high-level **Material** UI library for **React** to enable automatic theme- and styling-injection, as well as access to a consistent set of pre-styled components. This provides a basic set of building-block components all determined by a single theme configuration. If properties of the theme are changed, these changes propagate down through the entire application. Moreover, when needed, theme properties can be overridden by particular components. This enables a uniformly styled UI, where components designed by team members automatically adopt the theme of the application.

Unidirectional data-flow & Immutability To facilitate an easy division-of-labour between members of the front-end team and reduce mental overhead, a strict unidirectional data-flow and pub-sub architecture was adopted for data management. This is implemented through the **Redux** library. All communication is done through a single channel (via actions), and actions may only be dispatched from outside of the data-store. While this leads to greater initial overhead in setting the project up, it has enabled us to easily reason about the project as time has gone on. Utilising **seamless-immutable** to enforce immutability throughout the project has also ensured the absence of bugs caused by state-mutation due to side-effecting functions in other parts of the project.

Development server One of the challenges of developing the front-end in tandem with the back-end has been managing data-dependencies for the development process. Thus rather than initially developing against the changing back-end, we have primarily developed against sample data in our production server, and transitioned to live servers once the project was bootstrapped.

4.2.3 xWrap Methods

Some of the key design decisions were already outlined in section 3.4. On top of these design choices, the following methods were used in the implementation of the exchange wrapper.

Paper Trading As *xWrap* must support both paper and live trading on an exchange, it was desirable to reduce code duplication between these two modes of operation. Lightweight subclasses were considered but this would require additional work from a developer adding support for a new exchange, thus the two modes of operation were implemented via dynamic inheritance at start-up, whereby the exchange adapter selection can be overwritten with simulation routines when an exchange is loaded in 'paper' trading mode. Trades are matched against real-time order books maintained via subscriptions to level 2 trading data, and as such have a high degree of accuracy.

Easing Client Development Developing for an API only web service is sometimes difficult during the early stages of a project due to the absence of a user friendly interface to test API calls. From the API developers perspective, it is also difficult to spot the nuances of API usage and address any design issues pro-actively. To overcome this obstacle, *xWrap* provides its own web-based dashboard to allow users to view data and perform most core operations. This dashboard as shown in section 6.4 uses the same APIs as client micro-services and serves as a reference example for API usage.

Event Loop The *xWrap* service is largely I/O driven, acting as both a client and server, and carrying out tasks when certain events are triggered (a new level opens on a cryptocurrency order book, an order is filled by an exchange, etc.). An Event Loop design pattern lends itself well to this type of application, allowing programmers to segment behaviour into tasks triggered by events (incoming/outgoing messages, timer ticks, etc.). Python provides a number of libraries to implement such a loop ranging from the DIY solutions around the `select()` system call to fully fledged frameworks such as **Tornado**. For the design of *xWrap*, three main approaches were considered for implementing an event loop:

Worker Threads Threads are a popular concurrency primitive. Any parallelism advantages however are nullified thanks to the CPython Global Interpreter Lock (GIL) which prevents parallel execution of Python bytecode. Moreover, threaded applications can be notoriously difficult to debug when data races occur, and are in this respect inferior to the following solutions.

Custom Asynchronous Loop The `select()` system call and newer alternatives such as *epoll* and *libuv* provide a versatile mechanism to implement non-blocking I/O. This combined with a simple task manager can be used to implement a basic event loop in Python. The downside is that any I/O participant must declare and manage callbacks whenever it waits on I/O and the management of these callbacks can become tedious and difficult to debug.

AsyncIO The `asyncio` library was added in Python 3.4 to provide a more programmer friendly approach to asynchronous programming in Python. The library adds a new programming paradigm to the language and thus presents a steep learning curve to even experienced Python programmers. However, it presents the most elegant solution to the problem of an event loop, as Python functions can be transformed into asynchronous tasks through addition of a single keyword. The downside however is that co-routines (as created by `asyncio`) do not support pre-emption and thus tasks must voluntarily yield execution for other tasks to run. Any blocking operations inside a co-routine will block the entire process. The lack of pre-emption however has an advantage in simplifying the process of mutual exclusion, as context switches are solely under the control of the programmer.

After prototyping each implementation option, the `asyncio` option was selected as it presented the most modern and concise solution to designing an event loop. Some tasks such as rate limited HTTP operations are simpler to express with blocking I/O, and the Python built-in `concurrent.futures` library was used to synchronise threads and co-routines. The third-party `janus` library further provides a queue implementation that provides both synchronous and async friendly access.

Storing Time-Series Data Strategies require historic market data in order to compute metrics such as asset risk. These volumes of data can span multiple years and cannot be queried from exchanges on-demand. Thus the *xWrap* service must collect and store them such that they can be later queried by strategies. This data is indexed by time and can potentially reach volumes of a few hundred datapoints per minute.

A number of specialised Time-Series Databases (TSDB) and relational databases were considered for storing this data and PostgreSQL and InfluxDB were shortlisted for consideration. InfluxDB was selected for use mainly due to its simplicity and ease-of-use compared to PostgreSQL which required additional tuning for the expected volumes of data.

Simulated Trading It is impossible to model true market impact - the reaction the market would have to your trading activity. However, assuming any market impact is negligible - that an individual's trading activity is insignificant compared to traded volume over the period of their order - a paper or back-testing trading environment has value for testing trading strategies, particularly those focussing mostly on high frequency, short term trades.

At a level as granular as a level 2 order book, it is necessary to maintain control over what live liquidity "should be" available to prevent trading against the same liquidity twice. A record of

past virtual trades was implemented that matches off with incoming orders as they update the locally maintained book.

Matching Logic The virtual exchange implements a matching engine according to a typical central limit order book[17]. Orders are matched according to price time priority - virtual orders they are always traded at the best available price. Any order not immediately executable sits "on the book" until it matches against an incoming order. We model this by maintaining a record of open order which may match against new updates to the order book.

Four order types are available: limit order, market order, fill or kill (FOK) and immediate or cancel (IOC). These are the most commonly used order types on live exchanges. A limit order will execute the order up to a specified price limit and will remain "on the book" until it is matched or cancelled. A market order executes the full size of the order at the best currently available price. An IOC order trades all available size up to a specified limit and cancels the residual. A FOK order will trade the entire size of the order up to the limit or cancel the order.

This matching logic is typical of major trading venues across asset classes.

Maintaining Real-Time Order Books Maintaining live local order books allows high frequency trading strategies to be thoroughly tested and to competitively trade live. It also provides better pricing information for strategy analysis and ensures that trade execution achieves an optimal price in live trading and a realistic price during testing.

A live order book type was created with a universal message format for book updates and trades, incorporating double trade protection. At least 100 price levels are maintained for every traded currency on every supported exchange, with size aggregated at each level. A snapshot provides the order book baseline and update messages are pushed from the exchange over a websocket interface as soon as the exchange's order book changes.

4.2.4 Deployment

Motivations for Docker Deploying micro-services, dependencies and configuration can introduce significant systems administration overhead. Docker containerisation was adopted to minimise this overhead and speed up deployment. Each micro-service component is packaged with its own *Dockerfile* which defines how to build a container for that service. These individual containers are then orchestrated with the `docker-compose` tool in order to bring up the entire Cryptick platform with a single command-line invocation. This process makes deployment far more convenient than with manual installation and management of individual components.

4.3 Testing

4.3.1 Testing Overview

As mentioned in section 3, the project was broadly split into three main components - the strategy, an exchange wrapper and a front-end web interface. For testing, the three components are unit tested individually and integration tested against the predefined API contracts, resulting in a high test coverage and confidence in each component [2]. The final end-to-end testing will then attempt to capture any contractual disagreements between the three micro-services by running in paper trading mode. This simulates a real trading environment and captures the full range of functionality and behaviour by trading virtual assets according to real exchange data. The three test types used for the *Cryptick* platform are summarised below:

Unit Tests These are white-box tests that are performed on individual functions and methods in isolation. The completeness of the unit tests was measured by looking at total code coverage (see section 4.3.5). Unit tests were kept relatively simple and rely on mock functions to mimic the behaviour of external dependencies.

Integration Tests These tests check the links between unit tested objects and/or external services (databases, web APIs) to ensure that the individual components of a micro-service work together correctly. They are most commonly grey-box tests and have a lower reliance on mocks.

System Testing These black-box tests verify the functionality of the entire application. As the prior stages would verify the correctness of individual blocks to a significant extent, end-to-end tests seek to verify that the application functions at a high level. It is achieved by running the application against typical use cases and evaluating the resulting behaviour against expected outcomes.

4.3.2 Approach

- **Designing test cases:** Design of test cases depended heavily on the component of the project that was being tested. At a high level, the design of every test was composed of the input data, the order of execution of the test cases and the output of the test.
- **Preparing test data:** Test data was prepared in order to ensure high coverage with the use of credible, reliable and most of all representative data. Additionally, keeping the size of the data to a minimum level was also an objective.
- **Running tests with mock data:** The tests were first run with no data to ensure correct test behaviour. That was followed by testing with invalid data to ensure that correct errors would be raised. The last step was running the test with a valid dataset and ensuring no errors were raised.
- **Interpreting results:** To interpret the results, the design of the test had already specified an expected outcome with which the test result would be compared. An error was raised if there was a mismatch between the two.

4.3.3 Testing Modules

Exchange Wrapper (xWrap)

Testing Suite *xWrap* tests were written using the Python standard library `unittest` module which provided a powerful testing framework as well as a mocking facility (via `unittest.mock` in Python3). The `nose2` package was used as a test-runner to run the test-suite and generate branch and line coverage with `coverage.py`. A GitLab CI [6] script runs unit and integration tests on every commit and alerts users via Slack and Email of any failures.

Unit & Integration Testing Unit tests were written with in-depth knowledge of the underlying code (white box) and individual tests were designed to cover method-level scope. These tests were designed primarily to test correct business logic as well as to ensure robust error-handling on unexpected input/output. In order to test methods independently of their dependencies, Mock objects which synthesised return values were used to replace them. In Python, mocking was easily achieved by way of monkey patching code within test fixtures.

Integration tests take a grey-box or black-box approach to testing and rely on minimal knowledge of internal structure. The aim is to test combinations of unit-tested components for correct functionality. Integration tests for *xWrap* range from testing the integration of independent test units to the testing of *xWrap* as a stand-alone micro-service.

Challenges *xWrap* interacts with a number of external web services. In testing, this posed a significant challenge as live services could not be relied upon in tests. Several options were evaluated, but ultimately mocking was used to fake web requests and respond with pre-recorded API responses.

The use of `asyncio` for cooperative multitasking with co-routines added additional complexity to the testing process as `unittest` does not natively support testing co-routines. Following a common

workaround for the `unittest` framework, a lightweight async wrapper was implemented to simplify the test implementation [8]. For integration tests, a server thread was spawned with its own event loop for code under test while the test fixture ran in the main thread. Interactions between the two were performed over local network connections and threadsafe queues.

Current Status & Improvements The exchange wrapper currently achieves 90% code coverage and 85% branch coverage through unit and integration tests. Most of the untested lines are in error handling code for system level functions such as thread and socket handling. These facilities will need to be mocked in order to achieve coverage or extensively code-reviewed as the underlying events that trigger them are difficult to simulate inside test fixtures.

Strategy (Cryptick)

Testing Suite As the trading strategy framework, *cryptick*, builds on an identical technology stack as the exchange wrapper, it was decided to adapt the same testing suite. A similar GitLab Continuous Integration (CI) chain was also configured using Python's `unittest`, `nose2` and `coverage.py` modules.

Unit testing White-box unit tests are used on a single class or function scope to ensure each individual part's correctness. This is particularly crucial for the logic of the trading strategy itself. Without appropriate unit tests, potentially severe errors in the trading logic are hard to identify. For strategy testing, outputs generated from historical data were compared against manually computed values to test the correctness of the implementation.

Challenges Large changes in code structure can occur regularly in early stages of the Agile development process. As a result, a very fine-grained unit test suite could become obsolete in subsequent iterations. Therefore, an adaptive approach to unit testing was selected such that tests increase in granularity as the codebase matures.

Current Status & Improvements The strategy component currently achieves 65% code coverage through unit and integration tests. Some lines of code are currently untested as they have been added to adhere to functionality that will be implemented in later stages of production. It is projected that once this functionality is added, the strategy component will achieve 80% code coverage.

Front-end (Cryptock)

Testing Suite The primary testing technology used on the front-end is Facebook's `Jest`. This was chosen for its powerful mocking capabilities and simple integration with React (the front-end UI library). `Enzyme`, a library for shallow rendering, is used to aid UI-testing.

Unit Testing The most crucial aspects of the application (state management and inter-component communication) are given the most focus in unit-testing, though all units are thoroughly tested through white- and black-box tests (for those components using external libraries which were not mocked or stubbed out). API units were tested with the help of a mock adapter for our Ajax library (`axios-mock-adapter`).

GUI Testing GUI testing was performed iteratively with each new feature addition. As the front-end was designed in accordance with Google's Material-UI guidelines, it can be considered a composition of tried-and-tested components, and thus we needed only test this composition of GUI elements, rather than needlessly starting from the ground up.

Challenges The main challenge for testing the frequently-changing front-end has been the same as that mentioned in 4.3.3. Namely, the challenge has been striking a balance between the test suite being sufficiently coarse to describe application functionality across rapid iterations and sufficiently fine so as not to pass trivially. Accordingly, smoke-tests were employed during the early stages of development until a more concrete structure was settled on, at which point more comprehensive tests were written.

End to End Testing End-to-end testing is currently performed by running the system in 'paper' mode. Due to the different languages and environments (docker containers) that the individual micro-services run in, coverage cannot be measured in this scenario. It is proposed to implement a custom automated end-to-end test suite once interface APIs stabilise. However, as mentioned in Section 4.3.1, the high coverage of individual services minimises the importance of these tests.

4.3.4 Testing Risks & Notable Omissions

The testing methods used so far do not cover comprehensive end-to-end functionality, but are unit-specific to each of the three work streams. This is due to the development workflow which is optimised for time. End-to-end functionality will be black-box tested at a later stage with a fully integrated product.

Some code was 'smoke-tested' to ensure stability and basic functionality at this stage of development. This will be further verified in future testing to ensure code correctness.

4.3.5 Current Results

A summary of code coverage for the project is seen below. Full output from the coverage tools are included in Appendix D.

Table 1: Project Coverage Summary

Work Unit	Line Count	Line Coverage	Branch Count	Branch Coverage
Exchange wrapper (xWrap)	2405	82%	690	85%
Strategy (Cryptick)	1401	70%		
Front-end (Cryptock)	(66%)	67%		50%

4.4 Challenges

Many challenges were encountered and overcome during the development process of the *Cryptick* platform. A selection of these challenges and the means by which they were addressed are discussed below.

4.4.1 Architecture

The *Cryptick* platform is composed of multiple components which have widely differing functionality (e.g. strategy development, exchange management, user interface, etc...). Developing these functionalities in parallel inside a monolithic architecture leads to multiple organisational bottlenecks and requires a high degree of coordination between work streams. In addition this approach complicates testing as code units are often tightly coupled with each other and require heavy use of mocking for testing.

This challenge was overcome by organising the project in a micro-service architecture. This design approach allowed the full functionality of the project to be divided into three smaller standalone sub problems that could be independently solved. Pre-defining application programming interfaces (APIs) between these units during early development and regular integration testing mitigated the risk of incompatibility between sub-units at final integration.

4.4.2 Wallet Management

Cryptocurrency can be stored on hardware cold wallets, hot/cold software wallets or on cryptocurrency exchanges. The *Cryptick* platform currently supports exchange wallets only, and a challenge arises in partitioning funds held on these wallets between strategies that may be executing in parallel. The supported exchanges do not provide any wallet management capability within a single customer account. In addition, transferring funds between cryptocurrency wallets can incur significant transfer fees and introduce processing delays.

This issue was addressed by implementing a logical wallet feature in the *xWrap* service. A strategy can create a set of logical wallets and allocate funds to them from any unallocated exchange wallet balances. *xWrap* ensures that any trades generated by the strategy are accepted only if the logical wallets hold sufficient balance. Trades leading to movements or holds of funds on exchange wallets are reflected on logical wallets by the *xWrap* service ensuring consistency with exchange behaviour. On termination, the strategy can deactivate a logical wallet set and allow users to reallocate those funds to another strategy run.

4.4.3 Poorly Documented Exchange APIs

Certain cryptocurrency exchanges were found to have inaccurate API documentation and, in some cases, non functional APIs. This posed a risk to the implementation of key features of the *Cryptick* platform. The unpredictable availability of the Bitfinex API and the undocumented nature of the Poloniex websocket API are examples of this challenge.

In all cases, these challenges were overcome by workarounds at the cost of reduced capability or accuracy. Though non-ideal, the effects of these mitigations are systematic to any cryptocurrency trading software that utilises the affected exchanges and cannot be resolved until the underlying bugs are fixed.

4.4.4 Testing Trading Strategies

Unit-testing strategies posed a challenge as, in most cases, extensive mocking was required for the supply of user input (as provided by the front-end) and trade execution (as provided by *xWrap*). Mocking extensively in this way was not feasible within the time constraints of the project.

This challenge was overcome through a combination of two approaches. Firstly, the backtest feature allowed for the verification of the correctness of a strategy across large timescales. Secondly, integration tests were built to incorporate a running instance of the *xWrap* service and fake user input from the front-end. The latter tests covered the minutiae of trade execution and response, while the former covered algorithmic correctness. This two-pronged approach allowed for a high test confidence, but also complicated the test coverage reports.

5 Group Work

5.1 Project Milestones

At the start of the project, an initial milestone table was set to track internal deadlines. The status at the conclusion of the project is shown in Table 2.

Table 2: Project Milestones

Milestone	Due Date	Completed
Finalise high-level architecture	31/01/2018	30/01/2018
Specify interfaces between components	09/02/2018	06/02/2018
Exchange: Receive exchange data streams		
Front-end: Basic dashboard		01/02/2018
Exchange: Make historical OHLC data available	16/02/2018	16/02/2018
Strategy: Send visualisations to front-end		
Strategy: Efficient Frontier strategy making orders	23/02/2018	28/02/2018
Exchange: Simulate orders on a virtual exchange	23/02/2018	28/02/2018
Front-end: Display strategy visualisations		
Exchange: Send logs to front-end		
Front-end: Display exchange logs	02/03/2018	03/03/2018
Strategy: Backtest Efficient Frontier		
Second Report	05/03/2018	04/03/2018
Final integration		
Product meets minimum requirements	10/03/2018	15/03/2018
Front-end: render live data from server		
Exchange: message hub communications		
Strategy: general backtesting environment	30/03/2018	14/04/2018
Exchange: live trading	14/04/2018	Not complete
Front-end: front-end finalised	28/04/2018	15/05/2018
Final product	30/04/2018	15/05/2018
Finalise documentation	30/04/2018	11/05/2018
Final report	16/05/2018	15/05/2018

After the high-level architecture was finalised and the interface between components specified, the focus moved to implementation. OHLC (open-high-low-close) data was made available to the trading strategy, a key dependency for the strategy work stream. This was followed by basic integration of the platform, whereby the efficient frontier portfolio strategy was able to place paper trading orders on a virtual exchange. This integration milestone lagged behind its due date but, due to an extensible and robust code base, the accelerated forward timeline meant that the critical schedule remained mostly unchanged [22].

Completing the final integration took longer than planned. The minimum requirements mentioned in the specification section were all met by mid-March, relatively close to deadline that was set from the initial planning. The majority of the remaining stretch goals were also met, however, that only took place with a two week delay relative to initial deadline. The only goal that was not met by the

time of the report submission was the live trading feature. However, it is planned to implement this functionality by the date of the software demonstration.

5.2 Group Organisation

Workstreams As shown by table 5.2, this project consists of three major workstreams, each with two team members responsible for primary implementation and scheduling. This approach allows for pair programming and development collaboration between workstreams in the Scrum Sprints. Sub-team members have been allocated based on previous experience and personal preference. Regular code reviews with the full team were used in order to coordinate between workstreams in addition to the Scrum and Kanban practices.

Table 3: Parallel work streams

Workstream	Team Members
Trading System	Charith Amarasinghe, Simon Spurrier
Front-end & UI	Konstantin Hemker, James Griffiths
Portfolio strategy	Julian Vossen, Dimitris Nikolaou

5.3 Meeting Log

Through the course of the project, a number of well-documented meetings were held which are summarised in Appendix D. After each meeting, notes were circulated via email to each member and To Dos were assigned on Asana. As the group was relatively close-knit outside of university, there were numerous undocumented discussions and meetings in less formal settings.

5.4 Work Log

On top of the group meetings, the most representative proxy of the time expenditure of each workstream spent on each topic is the commit history. Note that on top of the git history, there were some code review sessions and pair programming sessions that contributed to the total time expenditure in table 5.4. A commit history over time with the relevant milestones is available in Appendix D.

6 Final Product

6.1 Project Summary

The *Cryptick* platform implements a fully featured cryptocurrency trading suite, incorporating a default portfolio diversification strategy, support for custom strategies, a web-based front-end, and a unified exchange interface (xWrap).

Adherence to spec All aspects of the core specification were met. This in addition to key stretch goals which complete the back-testable strategy architecture.

6.2 Front-end Overview

Run configuration The final front-end has a tab-based routing system connected by a navigation bar at the top of each page (see Figure 11). The landing page is the run configuration page (Fig. 11) and allows the user to view its recent runs, add wallets and start a new strategy run. Using the drop-down menus and sliders, the user can allocate its wallet balances from the different exchanges to that run. Strategies can be started in three modes, as discussed in the previous sections - paper trading, backtesting and live trading.

Table 4: Work Log by Topic

Workstream/Topics	Time expenditure (in man hrs)
General	
Idea generation & spec	37
High level design	14
Project admin	23
Report writing	163
Group Meetings	280
Subtotal	517
Strategy	
Vis. to front-end	30
EF making orders	42
EF backtest	10
Final API spec	30
General backtesting env	50
MessageHub integration	20
Full integration	40
Tests	25
Subtotal	247
Front-end	
Basic dashboard	38
Strategy visualisations	31
Render Orderbook	5
Render live data	20
Strategy API calls	15
Wallet management	28
Full integration	50
Tests	32
Subtotal	302
xWrap	
Overall architecture	19
Recv. data streams	35
OHLC data available	27
Virtual exchange	51
Logical wallets	32
Dashboard	16
Strategy API	29
Live book	36
Full integration	38
Tests	15
Subtotal	298
Total	1364

Strategy Tracker Once a run has started, a new strategy overview tab appears displaying details about that run. Figure 12 shows the strategy overview page of a running efficient frontier strategy. The blue lines on the graph show the weight history of the portfolio, i.e. the "path" it took to reach the efficient frontier. The efficient frontier is made up of the red points marked in the graph, where each point can be selected to start the diversification process towards those portfolio weights. To provide full customizability of the portfolio, the slider components in Figure 12 allow for the full customisation of the target portfolio. Moreover, actions can be dispatched that pause or update the strategy based on the current selection. Any update that is selected must be confirmed in an extra pop-up window

before trading starts based on the selection. The strategy page also has an overview of the orderbook which displays trades that were executed based on that strategy (Fig. 18).

Overview page The overview page carries any information relevant to the user across all strategies. These include things such as trades executed, wallet distribution across strategies and performance metrics of their strategies. At the time of writing, this page only displayed the trading data (orderbook), but it is expected to be completed by the date of the software demonstration.

Further features To further improve the front-end's functionality one could implement an in-browser editor e.g. through `codemirror` [13] which allows the user to utilise the API supplied by the front-end to implement custom trading strategies. The functionality on the back-end already exists as a command-line tool, where it is possible write a `strategy.py` file that leads to the execution of this strategy through the *MessageHub* and the *exchange wrapper*. Therefore, providing a clean interface of this functionality to the user would be of great value.

6.3 Strategy Overview

The strategy component supports the execution of different trading strategies, by communicating with `xWrap` to obtain data and execute trades. The strategy can simulate the execution of trading strategies using historical data in a backtest environment. Strategies may either be pre-defined (e.g. efficient frontier diversification) or user-defined.

Event-Driven Backtesting *Cryptick* implements event-driven backtesting which grants significant advantages over conventional approaches. This means high frequency trading strategies, triggered by particular events, are truly backtestable, assuming no market impact. This is in contrast to popular open source alternatives to *Cryptick*, where trading logic is periodically executed in a loop, independent of any events or market conditions. *Cryptick* also support periodic execution by self-scheduling periodic trigger events. As all communication during trading and trading simulations is routed through a central message hub, the system is easily auditable. A screenshot of the message hub dashboard displaying some messages is shown in Figure 14.

Figure 11: Configuring a run on the front-end

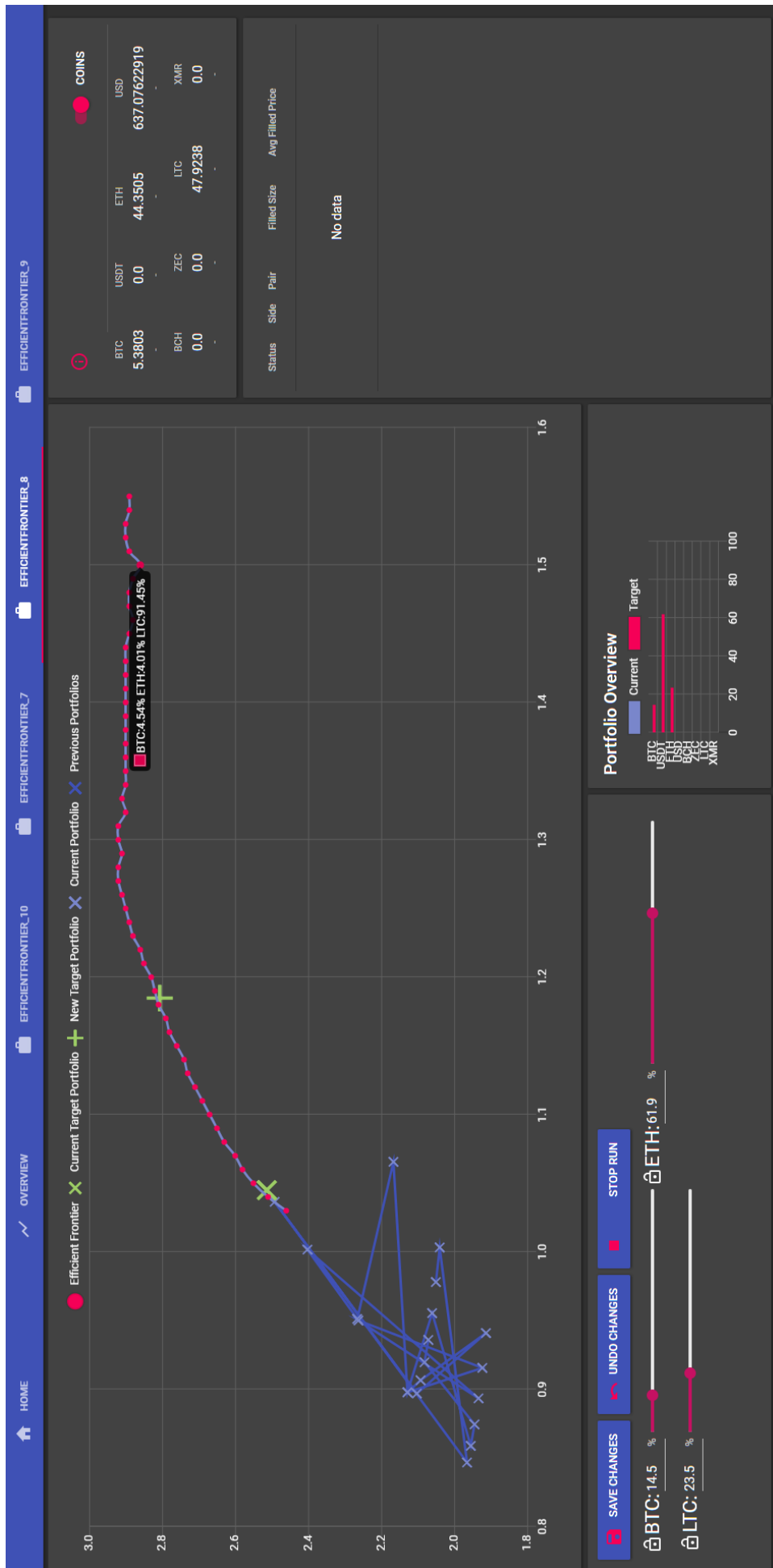


Figure 12: Strategy Overview Page

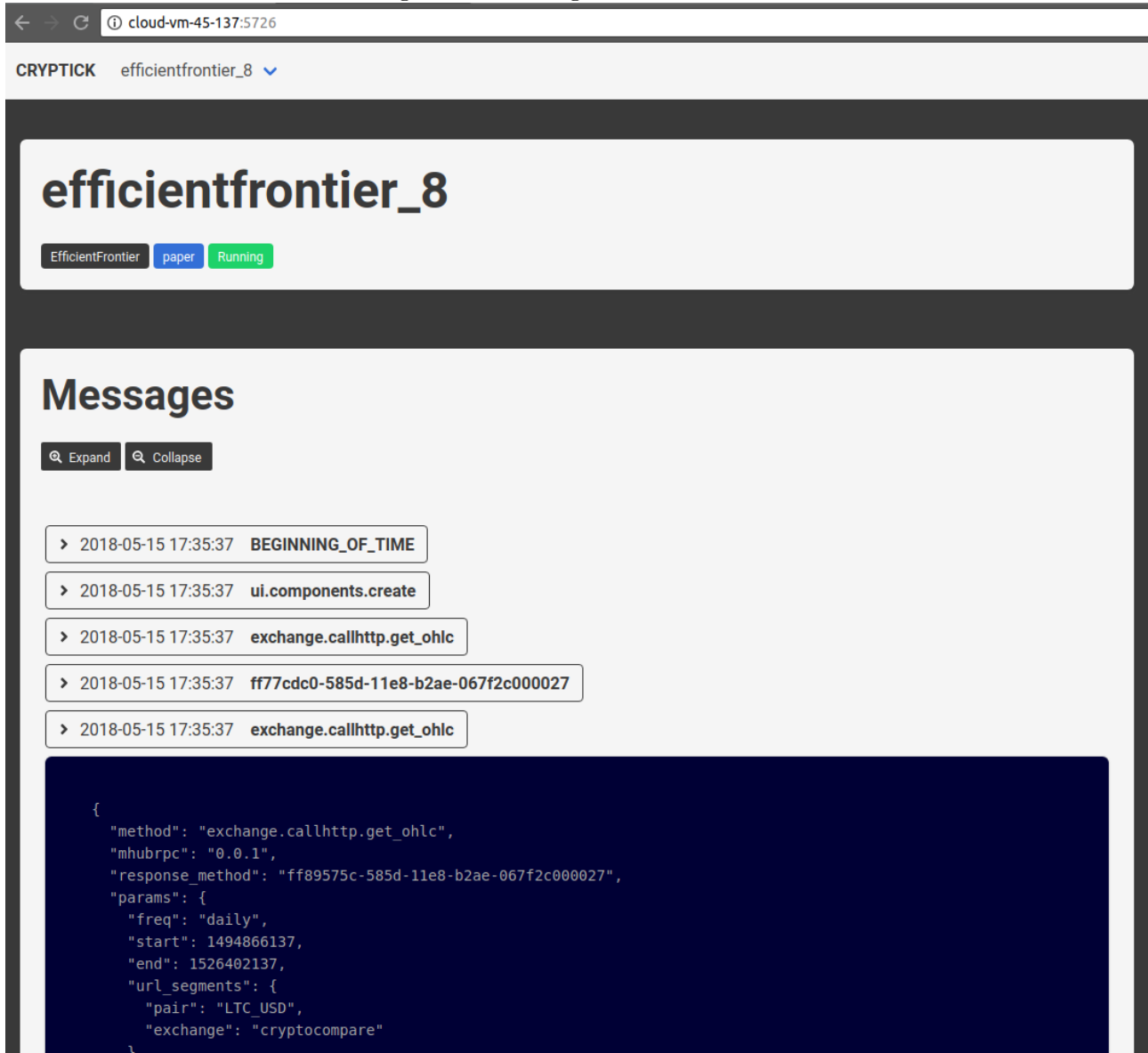
HOME
OVERVIEW
EFFICIENTFRONTIER_10
EFFICIENTFRONTIER_7

Drag a column header here to group by that column

Status	Side	Pair	Order Size	Limit	Filled Size	Avg Filled Price	Exchange	Run	At ↓
Filter...	Filt...	Filter...	Filter...	Filter...	Filter...	Filter...	Filter...	Filter...	Filter...
open	sell	BTC_USD	0.5882	7649.57	0.5882	8499.53000000	gdax	ef01	2018-05-15T18:37:32...
done	sell	BTC_USD	0.5882	7649.57	0.5882	8499.53000000	gdax	ef01	2018-05-15T18:37:32...
open	sell	BTC_USD	0.5882	7649.57	0.5882	8499.53000000	gdax	ef01	2018-05-15T18:37:32...
done	sell	BTC_USD	0.5882	7649.57	0.5882	8499.53000000	gdax	ef01	2018-05-15T18:37:32...
open	buy	ETH_USD	5.6153	783.47	5.6153	712.26000000	gdax	ef01	2018-05-15T18:37:32...
done	buy	ETH_USD	5.6153	783.47	5.6153	712.26000000	gdax	ef01	2018-05-15T18:37:32...

Figure 13: Front-end Overview Page

Figure 14: Message Hub Dashboard



User-Defined Strategies *Cryptick* supports development of custom strategies by the user which may utilize the back-testable architecture and interface with other services of the platform.

Efficient Frontier The strategy module includes a default diversification strategy based on Markowitz portfolio theory. The existing portfolio is automatically diversified towards a risk-return optimal point, based on user preferences.

6.4 Exchange Wrapper Overview

xWrap provides a uniform interface to GDAX, Bitfinex, and Poloniex for virtual trading in all supported currency pairs. It implements a live order book and corresponding double trade protection for exchanges that reliably support a live level 2 websocket data feed. An independent dashboard provides additional order monitoring and execution capabilities.

Data gathered from exchanges is available to other platform components and stored for future analysis and use in back-testing.

Virtual Exchange *xWrap* implements a virtual exchange with central limit order book matching logic, that matches trades against live market data from supported exchanges. It supports FOK, IOC,

limit and market orders in all trade-able currencies. This includes management of virtual wallets for each exchange and currency.

Dashboard An interactive dashboard allows the user to manually manage orders and monitor trading activity. It is also useful for audit purposes and platform debugging. Additionally, the user may view and manage virtual wallets.

Live Order Books It was not possible to provide a live order book for Poloniex or Bitfinex due to poor quality or non-existent documentation. For example, Poloniex do not officially support or document their push API at all. In these cases, a polled order book is used and immediate double trading is prevented with a trading time delay.

Live trading Comprehensive live trading on-exchange was not implemented due to time and resource constraints. However, it is demonstrable as a proof of concept.

Data management The exchange wrapper stores exchange data and live book update data that it uses in normal operation. This data is accessible to other platform components for analysis and testing.

6.5 Next steps for Cryptick

Assuming future implementation of live trading, whilst reliably operational, the project was constrained by time and resource, so the platform cannot be guaranteed for operation in a live business environment in its current state. However, with further development, the platform could be developed into a stable, viable product for financial applications.

Several options exist for the future of Cryptick:

- **Open source** - Relinquish ownership and allow the open source community to continue development.
- **Private use** - Use the platform for trading personal funds or as a basis for a trading business.
- **Develop as product** - Develop the product for use as a supported financial application for institutional or retail users.
- **Sell codebase** - Sell the codebase as-is or with some further development.

Exchanges

Exchange	Status	Mode	Pairs
gdax	✓ ready	paper	BTC_USD LTC_USD ETH_USD BCH_BTC ETH_BTC BCH_USD
poloniex	✓ ready	paper	XMR_USDT ZEC_ETH ETH_USDT BTC_USDT ZEC_USDT LTC_USDT ETH_BTC BCH_USDT BCH_ETH XMR_BTC LTC_XMR ZEC_XMR ZEC_BTC BCH_BTC LTC_BTC
cryptocompare	✓ ready	live	BCH_USD LTC_BTC LTC_USD XMR_USD ETH_USD XMR_BTC ZEC_USD BCH_BTC ETH_BTC ZEC_BTC BTC_USD
bitfinex	✓ ready	paper	LTC_BTC LTC_USD BCH_BTC ETH_BTC BTC_USD XMR_USD BCH_ETH XMR_BTC BCH_USD ZEC_BTC ZEC_USD ETH_USD

Wallets

Wallet Set: 1 (paperex)
Paper Exchange Simulated

Id	Exchange	Currency	Balance	Available	Hold	Balance USD	Available USD	Hold USD
1	bitfinex	ETH	5402.0	5402.0	0.0	US\$3,725,381.26	US\$3,725,381.26	US\$0.00
2	bitfinex	BTC	35521.0	35521.0	0.0	US\$292,382,586.46	US\$292,382,586.46	US\$0.00
3	bitfinex	USD \$	0.0	0.0	0.0	US\$0.00	US\$0.00	US\$0.00
4	bitfinex	LTC	161176.0	161176.0	0.0	US\$22,029,535.68	US\$22,029,535.68	US\$0.00

Figure 15: xWrap Exchange Status

xWrap Dashboard
Logs Exchanges Wallets Orders
OK

Orders

+ New Order

Open Orders

Date	Time	COID	Exchange	Run ID	Type	Side	Status	Pair	Size	Limit	Progress
- No Orders -											

Recent Orders

Date	Time	COID	Exchange	Run ID	Type	Side	Status	Pair	Size	Limit	Progress
16/05	07:48	0c3760	gdax	ef01	limit	buy	done	ETH_USD	5.6285	781.72	<div style="width: 100%;"><div style="width: 100%;"></div></div> Show Details
16/05	07:48	8ec66a	gdax	ef01	limit	sell	done	BTC_USD	0.5874	7659.75	<div style="width: 100%;"><div style="width: 100%;"></div></div> Show Details
16/05	07:47	75c0fc	gdax	ef01	limit	buy	done	ETH_USD	5.6285	781.72	<div style="width: 100%;"><div style="width: 100%;"></div></div> Show Details
16/05	07:47	fb807	gdax	ef01	limit	sell	done	BTC_USD	0.5874	7659.75	<div style="width: 100%;"><div style="width: 100%;"></div></div> Show Details
16/05	07:46	f29ba0	gdax	ef01	limit	buy	done	ETH_USD	5.6285	781.72	<div style="width: 100%;"><div style="width: 100%;"></div></div> Show Details
16/05	07:46	ad0f8c	gdax	ef01	limit	sell	done	BTC_USD	0.5874	7659.75	<div style="width: 100%;"><div style="width: 100%;"></div></div> Show Details
16/05	07:45	7fb458	gdax	ef01	limit	buy	done	ETH_USD	5.6285	781.72	<div style="width: 100%;"><div style="width: 100%;"></div></div> Show Details
16/05	07:44	139f10	gdax	ef01	limit	sell	done	BTC_USD	0.5874	7659.75	<div style="width: 100%;"><div style="width: 100%;"></div></div> Show Details
16/05	07:44	42b3ba	gdax	ef01	limit	buy	done	ETH_USD	5.6285	781.72	<div style="width: 100%;"><div style="width: 100%;"></div></div> Show Details
16/05	07:43	03e28e	gdax	ef01	limit	sell	done	BTC_USD	0.5874	7659.75	<div style="width: 100%;"><div style="width: 100%;"></div></div> Show Details
16/05	07:43	d64747	gdax	ef01	limit	buy	done	ETH_USD	5.6285	781.72	<div style="width: 100%;"><div style="width: 100%;"></div></div> Show Details
16/05	07:42	686cd8	gdax	ef01	limit	sell	done	BTC_USD	0.5874	7659.75	<div style="width: 100%;"><div style="width: 100%;"></div></div> Show Details
16/05	07:41	07b4c7	gdax	ef01	limit	buy	done	ETH_USD	5.6285	781.72	<div style="width: 100%;"><div style="width: 100%;"></div></div> Show Details
16/05	07:41	232c5e	gdax	ef01	limit	sell	done	BTC_USD	0.5874	7659.75	<div style="width: 100%;"><div style="width: 100%;"></div></div> Show Details

Figure 16: xWrap Order List

Wallets

[+ New Wallet Set](#)

Wallet Set: 1 (paperex)
Paper Exchange Simulated

bitfinex
 gdax
 poloniex
 aggregate

	Id	Exchange	Currency	Balance	Available	Hold	Balance USD	Available USD	Hold USD
1	bitfinex	ETH	5402.0	5402.0	0.0	US\$3,725,327.24	US\$3,725,327.24	US\$0.00	
2	bitfinex	BTC	35521.0	35521.0	0.0	US\$292,310,123.62	US\$292,310,123.62	US\$0.00	
3	bitfinex	USD \$	0.0	0.0	0.0	US\$0.00	US\$0.00	US\$0.00	
4	bitfinex	LTC	161176.0	161176.0	0.0	US\$22,013,418.08	US\$22,013,418.08	US\$0.00	
5	bitfinex	XMR	0.0	0.0	0.0	US\$0.00	US\$0.00	US\$0.00	
6	bitfinex	ZEC	0.0	0.0	0.0	US\$0.00	US\$0.00	US\$0.00	
7	bitfinex	BCH	0.0	0.0	0.0	US\$0.00	US\$0.00	US\$0.00	

Make Transaction Movement:

0

A positive or negative number.

Figure 17: xWrap Wallet List

xWrap Dashboard Logs Exchanges Wallets Orders

● OK

Logs

xWrap Version: 0.5.5

Currencies: ["ZEC", "BTC", "ETH", "USD", "XMR", "LTC", "USDT", "BCH"]

Exchanges: ["gdax", "poloniex", "bitfinex"]

```

16/05/18 08:58:37.399 user.orders.1[3411] INFO Order 103717ee-0f68-4a5c-84b3-48fc793a867c is done
16/05/18 08:58:37.332 user.orders.1[3411] DEBUG Traded 10.00000000 @ 8508.31031813702
16/05/18 08:58:35.199 user.orders.1[3411] DEBUG Got order 103717ee-0f68-4a5c-84b3-48fc793a867c
16/05/18 08:58:31.999 user.orders.1[3411] INFO Order 103717ee-0f68-4a5c-84b3-48fc793a867c is done
16/05/18 08:58:31.974 user.orders.1[3411] DEBUG Traded 10.00000000 @ 8508.31031813702
16/05/18 08:58:29.638 user.orders.1[3411] WARNING Order 2144 (103717ee-0f68-4a5c-84b3-48fc793a867c) invalid exchtime
16/05/18 08:57:37.776 user.orders.2[3411] INFO Order fdda0d62-ab25-4670-b883-f1967f1440db is done
16/05/18 08:57:37.746 user.orders.2[3411] DEBUG Traded 15.00000000 @ 8506.567503886333333333333333
16/05/18 08:57:35.757 user.orders.2[3411] DEBUG Got order fdda0d62-ab25-4670-b883-f1967f1440db
16/05/18 08:48:57.706 user.orders.ef01[3411] INFO Order 0c376086-cb35-480a-b9d0-6f82380c4bd7 is done
16/05/18 08:48:57.668 user.orders.ef01[3411] DEBUG Traded 5.6285 @ 711.04000000
16/05/18 08:48:56.275 user.orders.ef01[3411] DEBUG Got order 0c376086-cb35-480a-b9d0-6f82380c4bd7
16/05/18 08:48:49.430 user.orders.ef01[3411] INFO Order 8ec66ab9-bf6e-46ed-a61c-c88a26815606 is done
16/05/18 08:48:49.399 user.orders.ef01[3411] DEBUG Traded 0.5874 @ 8510.84000000
16/05/18 08:48:47.388 user.orders.ef01[3411] DEBUG Got order 8ec66ab9-bf6e-46ed-a61c-c88a26815606

```

Figure 18: xWrap Logs

References

- [1] Bloomberg. U.s.-10y yield, May 2018. <https://www.bloomberg.com/quote/USGG10YR:IND>.
- [2] Toby Clemson. Testing strategies in a microservice architecture, November 2014. <https://martinfowler.com/articles/microservice-testing/>.
- [3] CoinMarketCap. Global charts - total market capitalisation & restriction map, January 2018. <https://coinmarketcap.com/charts/>.
- [4] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irving, 2000.
- [5] Inc. Functional Software. Sentry.io website, May 2018. <https://sentry.io/welcome/>.
- [6] GitLab. Gitlab ci documentation, May 2018. <https://docs.gitlab.com/ee/ci/>.
- [7] Rodrigo Gomez-Grassi. Markowitz portfolio optimization for cryptocurrencies in catalyst, September 2017. <https://blog.enigma.co/markowitz-portfolio-optimization-for-cryptocurrencies-in-catalyst-b23c38652556>.
- [8] Manuel Grinberg. Unit testing asyncio code, February 2017. <https://blog.miguelgrinberg.com/post/unit-testing-asyncio-code>.
- [9] JSON-RPC Working Group. Json-rpc 2.0 specification, January 2013. <http://www.jsonrpc.org/specification>.
- [10] Investopedia. Portfolio variance, January 2013. <https://www.investopedia.com/terms/p/portfolio-variance.asp>.
- [11] Investopedia. Sharpe ratio, January 2014. <https://www.investopedia.com/terms/s/sharperatio.asp>.
- [12] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Unpublished, October 2008.
- [13] CodeMirror Project. Codemirror project, May 2018. <https://github.com/codemirror/CodeMirror>.
- [14] Bjrn Rabenstein. Prometheus: Monitoring at soundcloud, January 2015. <https://developers.soundcloud.com/blog/prometheus-monitoring-at-soundcloud>.
- [15] Rancher. Red method for prometheus monitoring - 3 key metrics for monitoring, July 2017. <https://rancher.com/red-method-for-prometheus-3-key-metrics-for-monitoring/>.
- [16] Ken Schwaber. *Agile Project Management with Scrum (Microsoft Professional)*. 2004.
- [17] Gary Shorter. The central limit order book (clob) option for linking u.s. stock markets, 2000.
- [18] S. Spurrier and C. Amarasinghe. xwrap documentation 0.4.3. https://www.doc.ic.ac.uk/project/2017/530/g1753006/xwrap_docs/v0.4/, April 2018.
- [19] Talin. Pep 3102 – keyword-only arguments, April 2006. <https://www.python.org/dev/peps/pep-3102/>.
- [20] Emily von Hoffmann and GitLab. Demo - mastering code review with gitlab, March 2017. <https://about.gitlab.com/2017/03/17/demo-mastering-code-review-with-gitlab/>.
- [21] Julian Vossen et al. Digital asset portfolio management platform - report one. Unpublished, January 2018.
- [22] Julian Vossen et al. Digital asset portfolio management platform - report two. Unpublished, January 2018.

Glossary

- backtesting** Backtesting is the process of mocking a strategy run in paper trading mode over past time, usually in order to test its performance before live trading. 26
- CLI** Command-line interface. 6
- DBMS** Database Management System. 6
- efficient frontier** The efficient frontier is the most efficient portfolio mix of any combination of assets that optimises for risk (std. dev.) and return. 17
- FOK** Fill or kill order. The order is immediately executed in its full size at its limit or is cancelled. 20
- high-frequency trading** A style of trading whereby a large number of orders are placed in a short period of time that may respond to market conditions at very fast speeds. 1
- IOC** Immediate or cancel order. The order is executed against all liquidity priced better than its limit and any residual is cancelled. 20
- limit order** An order executed against any liquidity better than the limit. Any amount not traded will remain on the order book at the limit price. 20
- live trading** Trading real money on real exchanges. 20
- logical wallet** Logical partition of funds within an exchange wallet. 9
- market cap** The number of shares of a stock multiplied by the quantity issued. Taking inspiration from this, the market capitalisation of a cryptocurrency is the value of one unit multiplied by the number of units in circulation. 1
- market order** An order executed in its full size at the best available price at time of execution. 20
- order book** A record of buy and sell orders with price and size, ordered by price and time of order submission. 9
- paper trading** Paper trading is a trading feature that attempts to reproduce all features of a live trading market so that a user may practice or test trade behaviour without financial risk. 20
- portfolio** A portfolio is a grouping of financial assets, usually comprising of many different trade-able assets on financial markets, such as a stocks, bonds, and cryptocurrencies. 1
- run** A strategy run is an instantiation of a runnable. Note that there can be multiple runs of a single runnable. 44
- runnable** Runnable strategies are the set of all strategies available to the user that he can start. 44
- trading pair** Commonly referred to as currency pair is the quotation and pricing structure of the currencies traded in the foreign exchange market. An example of a trading pair would be USD/EUR or BTC/ETH in this context. 1
- UI** User interface. 2
- virtual wallet** A simulated exchange wallet. 33

Appendices

A Installation Instructions

A.1 System Requirements

It is recommended that the host environment meets the following requirements:

- A dual-core CPU running at 2GHz or higher
- Minimum 4GB of RAM (2GB if InfluxDB hosted seperately)
- Ubuntu 16.04 or later (Ubuntu 16.04 LTS recommended)
- CPython 3.6 or later
- Docker CE 1.8 or later
- Docker Compose 1.21.2 or later
- InfluxDB (OSS) 1.5.0 or later
- NodeJS 9.0 or later

A.2 Installing Dependencies

Each host machine should have the required dependencies installed. The easiest way to accomplish this is by using the scripts in the `devops` package bundled with this project.

InfluxDB and Cryptick can be co-located on the same VM, but due to InfluxDB's greedy memory allocation, it is best if it is hosted on a dedicated VM when used in production.

Deployment

1. To run *Cryptick*, *InfluxDB* and *xWrap*, the easiest solution is to use the bundled `docker-compose.yml` file.

```
docker-compose build
```

2. To bring up the services:

```
docker-compose up
```

3. This should bring up the services on the ports 4200 (xWrap RPC), 4242 (xWrap HTTP API), 5726 (Cryptick API) and 8086 (InfluxDB). These can be tested by:

```
curl localhost:5726/runnables
curl localhost:4242/xwrap/v1/status
```

4. To run the cryptick front-end, `npm` must be installed and a reverse proxy must should be used:

```
cd cryptick
npm install
npm build-paper
npm serve-paper
```

5. The front-end server will now be up on port 8200. Proxy this to port 80 (or for testing, use reverse SSH tunneling).

```
ssh -L 8200:localhost:8200 remote_server_address
```

Table 5: xWrap API Methods

API call	RPC	HTTP	Description	R/W
echo	Y	N	Echoes parameters	W
make_order	Y	Y	Creates a Order	W
cancel_order	Y	Y	Cancels an Order	W
get_exchanges	Y	Y	Gets exchanges and statuses	R
get_pricing	Y	Y	Get average pricing for all active pairs	R
get_wallet_set	Y	Y	Get state of a wallet set	R
create_wallet_set	Y	Y	Allocate a new wallet set	W

B xWrap API Documentation

Exchange Interaction API The Exchange Interaction API is supplied by the xWrap microservice and is consumed by both strategy instances and the Front-end UI. To support both these use-cases, a RPC based and HTTP based API is provided. The main xWrap API methods are outlined in Table 5, detailed documentation can be found in the xWrap Reference Documentation[18].

RPC API RPC API allows connected clients to execute methods on the xWrap service as if they were local function calls. This functionality requires a message format that encodes a function name, arguments list and return value into a network packet, and stub functions that marshal and unmarshal Python function calls into these messages.

Multiple RPC implementations were evaluated including gRPC, XML-RPC and JSON-RPC. The JSON-RPC format transported over WebSockets was eventually selected due to its simplicity and wide platform support. The *tinyrpc* library for Python is used for standards compliant message serialisation and de-serialisation.

JSON-RPC allows the transport of method arguments either as a list (positional arguments in Python) or a dictionary (keyword arguments in Python), but not both. It was decided to utilise keyword arguments across all RPC APIs in order to reduce the risk of mistakenly swapping arguments in a function call. An example Python client API call is shown below:

```
wallet = xwrapapi.get_wallet(exchange="gdax", base="BTC", wallet_set_id=2)
```

When executed, the stub function for `get_wallet` will encode the specified parameters into a JSON-RPC message and send it to the server via the RPC websocket. The serialised wire-format of the above call is as follows:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "get_wallet"
  "args": {"exchange": "gdax", "base": "BTC"}
}
```

The server decodes this message and maps it to an internal Python function call. This server-side method is declared with the signature `get_wallet(self, *, exchange, base)`. This definition uses Python keyword only argument syntax and will result in the remote call failing if there is an argument mismatch[19]. An example server response is shown below.

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {"balance": "100.0", "id": 9}
}
```

After de-serialisation on the client-side, the `wallet` variable will now be set to the python *dict* `{"balance": "100.0", "id": 9}`. The JSON-RPC standards document can be consulted for a more detailed description of the protocol[9].

HTTP API The HTTP API provides easy access to most data held by the xWrap service such as wallet balances and open orders. Each exposed API method maps to a URI on a xWrap HTTP server and request parameters can be passed as part of the URI structure. For example, the `get_wallet` API could be invoked via the HTTP API as a simple HTTP GET call as follows:

```
GET /xwrap/v1/wallets/gdax/BTC HTTP/1.1
Host: cloud-vm-45-137.doc.ic.ac.uk
Connection: keep-alive
```

With the matching response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Connection: Closed

{"balance": "100.0", "id": 9}
```

API calls that fetch data from the server are implemented by HTTP `GET` methods while calls that change or add data to the server are implemented with HTTP `POST`. APIs were designed with REST (Representational State Transfer) concepts in mind, but adherence to these concepts was not strict[4].

The API version in the URL string allows backward incompatible changes to API formats to be deployed without breaking support for existing clients.

Notifications The xWrap service generates notification messages whenever it becomes aware of an update from one or more upstream services such as cryptocurrency exchanges or data providers. JSON-RPC provides a convenient mechanism to deliver these messages to clients (JSON-RPC notifications), however the HTTP 1.1 protocol does not provide a mechanism to push data from the server to clients in this way. The *SocketIO* library is thus used to serve notifications to HTTP clients by way of WebSocket connections initiated through the HTTP server.

Alternatives The choice of building JSON-RPC over Websocket and HTTP APIs as discussed above was mainly due to their simplicity and cross-platform capability in a micro-service environment. Several other RPC formats and transports were considered such as:

Message Queue (Transport) Message Queues are a common method of inter-process communication in a microservice environment. Options range from brokered queues such as RabbitMQ to lightweight decentralised alternatives such as NSQ or ZMQ. However, as the microservices described above are not to be used in a clustered environment, message queues would add unnecessary overhead for little benefit.

TCP Network Sockets (Transport) Plain TCP sockets could be used for transporting API call payloads, however this introduces new requirements such as message framing (if streaming API calls) and the implementation of an additional compatibility layer for web browser clients. The WebSocket and HTTP protocols do not have these disadvantages.

MsgPack or Protobuf RPC (Format) MsgPack and Protobuf are alternative means to serialise RPC calls over the wire. Both would result in much shorter message lengths and, in the latter case, means of defining a strict schema. The disadvantage of both however is the lack of human readability. As all the above microservices operate on a single machine, the additional data overhead has negligible impact.

gRPC (Transport+Format) gRPC is a powerful framework for developing cross-platform, cross-language RPC services using Protobuf and HTTP/2. However, its design requires the automated generation of stub code at both client and server and adds additional dependencies. As the APIs being built are mainly for internal use in a non-cluster environment, it was thought that employing gRPC would add too much development overhead.

C Strategy API

The strategy API is implemented via HTTP GET and PUT calls.

Table 6: Cryptick API Routes

API call	Description
GET/runnables	Gets list of runnable strategies
GET/runs	Get list of all runs and their status
GET/runs/ <i>run-id</i>	Get details on a run
GET/runs/ <i>run-id</i> /components	Get list of all UI components created by a run with given ID
GET/runs/ <i>run-id</i> /messages	Returns all messages that went through the message hub during a run
PUT/runnables/ <i>runnable-name</i> /start	Runs a strategy with given name and returns run-ID
PUT/run/ <i>run-id</i> /stop	Stops the run with given ID
PUT/runs/ <i>run-id</i> /components/ <i>component-id</i>	Send json-encoded data to specific component

HTTP GET API calls GET methods allow the front-end to query data on running and runnable strategies. GET methods are called by front-end components when rendering to query relevant data. A full list of GET methods can be found in Table 1 as well as the full documentation (**citation needed**). An example response from the back-end for querying the actively running strategies would be the following:

```
GET /runs HTTP/1.1
Host: localhost
Connection: close

HTTP/1.1 200 OK
Content-Type: application/json
Connection: Closed

{
  "demostrategy_1": {
    "mode": "backtesting",
    "run_id": "demostrategy_1",
    "running": false,
    "strategy": "DemoStrategy"
  },
  "echostrategy_0": {
    "mode": "paper",
    "run_id": "echostrategy_0",
    "running": false,
    "strategy": "EchoStrategy"
  },
  "efficientfrontier_0": {
```

```
    "mode": "live",
    "run_id": "efficientfrontier_0",
    "running": false,
    "strategy": "EfficientFrontier"
  }
}
```

Note that by differentiating between runnables and running strategies, the API allows for creating multiple runs (instantiations) of one type of runnable. For example, if a user wanted to run multiple efficient frontier strategies at the same time, this would be possible by creating multiple runs of a runnable strategy type "EfficientFrontier".

HTTP PUT API calls PUT methods are usually used to start and stop runnables/running strategies or to adjust the portfolio weights of a running strategy. The full list of PUT methods can be found in Table 1 and a sample request to start a backtest run can be found below:

PUT /runnables/HodlStrategy/start HTTP/1.1

Host: localhost

Content-Type: application/json

Connection: close

```
{
  "mode": "backtest",
  "start": 1523003864,
  "end": 1523823864,
  "balances": [
    {
      "exchange": "gdax",
      "base": "USD",
      "balance": 1000
    },
    {
      "exchange": "gdax",
      "base": "BTC",
      "balance": 10
    }
  ]
}
```

D Test coverage

D.1 Strategy coverage

<i>Module ↓</i>	<i>statements</i>	<i>missing</i>	<i>excluded</i>	<i>coverage</i>
cryptick/__init__.py	0	0	0	100%
cryptick/api/__init__.py	2	0	0	100%
cryptick/api/filewriter.py	87	36	0	59%
cryptick/api/socket.py	34	20	0	41%
cryptick/exchange/__init__.py	2	0	0	100%
cryptick/exchange/xsim/__init__.py	1	0	0	100%
cryptick/exchange/xsim/xsim.py	150	66	0	56%
cryptick/exchange/xsim/xwrapprovider.py	77	67	0	13%
cryptick/exchange/xwrap/__init__.py	2	0	0	100%
cryptick/exchange/xwrap/api.py	118	47	0	60%
cryptick/exchange/xwrap/interface.py	92	58	0	37%
cryptick/messagehub.py	102	9	0	91%
cryptick/run.py	60	24	0	60%
cryptick/server.py	222	106	0	52%
cryptick/strategy/__init__.py	6	0	0	100%
cryptick/strategy/component_demo.py	28	8	0	71%
cryptick/strategy/demo.py	30	2	0	93%
cryptick/strategy/echo.py	10	0	0	100%
cryptick/strategy/efficient_frontier/__init__.py	1	0	0	100%
cryptick/strategy/efficient_frontier/ef_strategy.py	159	114	0	28%
cryptick/strategy/efficient_frontier/ef_util.py	159	10	0	94%
cryptick/strategy/hodl.py	31	15	0	52%
cryptick/strategy/strategy.py	2	0	0	100%
cryptick/utils/__init__.py	1	0	0	100%
cryptick/utils/sync.py	25	1	0	96%
Total	1401	583	0	58%

Figure 19: Strategy - Unit Test Coverage Report

Coverage report: 70%				
<i>Module ↓</i>	<i>statements</i>	<i>missing</i>	<i>excluded</i>	<i>coverage</i>
cryptick/_init_.py	0	0	0	100%
cryptick/api/_init_.py	2	0	0	100%
cryptick/api/filewriter.py	87	18	0	79%
cryptick/api/socket.py	34	20	0	41%
cryptick/exchange/_init_.py	2	0	0	100%
cryptick/exchange/xsim/_init_.py	1	0	0	100%
cryptick/exchange/xsim/xsim.py	150	12	0	92%
cryptick/exchange/xsim/xwrapprovider.py	77	15	0	81%
cryptick/exchange/xwrap/_init_.py	2	0	0	100%
cryptick/exchange/xwrap/api.py	118	13	0	89%
cryptick/exchange/xwrap/interface.py	92	10	0	89%
cryptick/messagehub.py	102	19	0	81%
cryptick/run.py	60	24	0	60%
cryptick/server.py	222	222	0	0%
cryptick/strategy/_init_.py	6	0	0	100%
cryptick/strategy/component_demo.py	28	18	0	36%
cryptick/strategy/demo.py	30	15	0	50%
cryptick/strategy/echo.py	10	6	0	40%
cryptick/strategy/efficient_frontier/_init_.py	1	0	0	100%
cryptick/strategy/efficient_frontier/ef_strategy.py	159	15	0	91%
cryptick/strategy/efficient_frontier/ef_util.py	159	10	0	94%
cryptick/strategy/hodl.py	31	6	0	81%
cryptick/strategy/strategy.py	2	0	0	100%
cryptick/utils/_init_.py	1	0	0	100%
cryptick/utils/sync.py	25	1	0	96%
Total	1401	424	0	70%

coverage.py v4.5.1, created at 2018-05-15 19:30

Figure 20: Strategy - Integration Test Coverage Report

D.2 Front-end coverage

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
All files	66.7	50	57.58	66.75	
src	100	50	100	100	
App.js	100	50	100	100	21
src/actions	83.17	100	52.78	94.59	
index.js	80.95	100	42.86	94.92	55,59,63
utils.js	94.12	100	87.5	93.33	49
src/components/AppBar	95.83	75	100	95.24	
index.js	95.83	75	100	95.24	66
src/components/Control	100	100	100	100	
Control.js	100	100	100	100	
src/components/Delta	84.62	100	60	83.33	
Delta.js	84.62	100	60	83.33	21,27
src/components/Graph	66.67	37.5	57.14	65	
Graph.js	66.67	37.5	57.14	65	... 123,159,160
src/components/Information	54.55	0	60	50	
index.js	54.55	0	60	50	40,44,45,46,47
src/components/Modal	100	86.36	100	100	
ConfirmationModal.js	100	86.36	100	100	61,66,89
src/components/OrderBook	100	100	100	100	
OrderBookFull.js	100	100	100	100	
OrderBookSimple.js	100	100	100	100	
src/components/RunConfiguration	56.52	40.91	33.33	58.54	
DatePickers.js	100	100	100	100	
RunAllocationForm.js	50	0	16.67	62.5	37,38,53
RunConfigurationForm.js	48.28	50	21.43	48.15	... 118,150,226
sanitizeForAPI.js	100	50	100	100	42,43
src/components/RunTable	50	14.29	20	50	
RunTableSimple.js	50	14.29	20	50	... 44,46,48,53
src/components/Sliders	100	70	100	100	
Sliders.js	100	100	100	100	
ThemedSlider.js	100	50	100	100	16,22,27
src/components/TextLog	100	50	100	100	
index.js	100	50	100	100	22
src/components/WalletDisplay	48.84	33.33	29.41	47.5	
BalanceItem.js	68.75	50	50	66.67	55,72,73,74,78
Simple.js	37.04	13.33	18.18	36	... 126,144,154
src/components/utils	5.71	0	0	5.88	
AutoTableResizer.js	5.71	0	0	5.88	... 79,81,90,92
src/containers	57.69	28.81	52.38	55.08	
AppBarContainer.js	100	100	100	100	
ControlContainer.js	61.54	50	40	58.33	28,30,38,39,40
DeltaContainer.js	66.67	40	100	66.67	23,24,26
GraphContainer.js	66.67	40	66.67	63.64	26,27,29,40
InformationContainer.js	63.64	30	66.67	60	15,16,22,31
ModalContainer.js	40	11.76	50	39.29	... 81,82,83,85
OrderBookFullContainer.js	44.44	100	28.57	37.5	16,17,18,19,20
OrderBookSimpleContainer.js	75	50	100	75	10
SlidersContainer.js	72.73	50	50	70	21,27,28
WalletDisplaySimpleContainer.js	72.73	40	66.67	70	24,26,34
WalletDisplayTableDetail.js	40	22.22	33.33	38.46	... 23,27,28,29
src/pages	20.34	0	0	23.53	
EfficientFrontierPage.js	15	0	0	16.67	... 139,141,150
GenericRunPage.js	15.79	0	0	17.65	... 111,120,121
OverviewPage.js	50	100	0	50	7
Root.js	27.78	0	0	35.71	... 91,92,96,97
src/reducers	69.66	70	64.1	70.24	
index.js	67.5	69.83	61.76	68.42	... 469,484,503
mkLocalComponents.js	88.89	75	80	87.5	24,25
src/sagas	38.98	0	23.08	38.98	
index.js	38.98	0	23.08	38.98	... 157,158,160
src/selectors	92.77	80	88.64	92.54	
index.js	92.77	80	88.64	92.54	21,62,73,74,162
src/services	95.56	100	91.3	100	
index.js	95.56	100	91.3	100	
src/store	83.33	100	50	100	
configureStore.js	83.33	100	50	100	
src/test_utils	100	100	100	100	
state.js	100	100	100	100	
src/utils	100	100	100	100	

Figure 21: Front-end (Cryptock) - Test Coverage Report

D.3 xWrap coverage

Coverage report: 82%


Module ↓	statements	missing	excluded	branches	partial	coverage
exchange_wrapper/_init_.py	1	0	0	0	0	100%
exchange_wrapper/client.py	104	29	0	18	1	70%
exchange_wrapper/config.py	47	5	0	22	2	87%
exchange_wrapper/connection.py	98	13	0	20	7	83%
exchange_wrapper/data_sinks/_init_.py	0	0	0	0	0	100%
exchange_wrapper/data_sinks/influx.py	76	13	0	18	2	76%
exchange_wrapper/data_sinks/sink.py	22	1	0	0	0	95%
exchange_wrapper/db.py	5	0	0	0	0	100%
exchange_wrapper/exchange/_init_.py	0	0	0	0	0	100%
exchange_wrapper/exchange/base.py	200	28	0	56	6	84%
exchange_wrapper/exchange/bitfinex.py	57	0	0	12	0	100%
exchange_wrapper/exchange/compare.py	84	9	0	20	3	88%
exchange_wrapper/exchange/gdax.py	132	8	0	42	1	95%
exchange_wrapper/exchange/poloniex.py	62	1	0	14	1	97%
exchange_wrapper/exchange/virtual.py	146	21	0	60	13	80%
exchange_wrapper/http_server.py	238	72	0	46	17	66%
exchange_wrapper/live_book/_init_.py	0	0	0	0	0	100%
exchange_wrapper/live_book/live_book.py	57	23	0	36	2	52%
exchange_wrapper/logical_wallet.py	74	9	0	36	4	85%
exchange_wrapper/matching.py	37	1	0	22	1	97%
exchange_wrapper/models.py	166	8	0	42	4	94%
exchange_wrapper/pair.py	22	1	0	2	0	96%
exchange_wrapper/rpc.py	92	3	7	12	0	97%
exchange_wrapper/service.py	414	93	55	144	30	73%
exchange_wrapper/util/_init_.py	30	2	0	14	0	95%
exchange_wrapper/util/token_bucket.py	19	0	0	2	0	100%
exchange_wrapper/wallet.py	27	2	0	0	0	93%
exchange_wrapper/wsock_client.py	50	7	0	12	2	82%
exchange_wrapper/wsock_server.py	109	9	0	36	5	88%
exchange_wrapper/wsocklog/_init_.py	0	0	0	0	0	100%
exchange_wrapper/wsocklog/formatter.py	14	4	0	2	0	62%
exchange_wrapper/wsocklog/handler.py	22	6	0	2	0	67%
Total	2405	368	62	690	101	82%


coverage.py v4.5.1, created at 2018-05-15 17:15

Figure 22: xWrap - Test Coverage Report

E Meeting Log

Table 7: Meeting log

	Date	Topic	Main issues	Ext. present
30 min	20/04/2018	Next Steps for Cryptick	Final Integration, post-project use	W. Knottenbelt, S. Werner
2h	11/04/2018	Integration	Report 3 to dos, final deadline to dos	None
2h	30/03/2018	Integration	API issues, L3 orderbook, wallet management	None

Continuation of Table 7				
	Date	Topic	Main issues	Ext. present
3h	24/03/2018	Workstreams	Currency conversion issues, L3 orderbook, Strategy back-testing	None
2h	21/03/2018	Integration	Wallet Management, detailed API discussion, etc.	None
30 min	14/03/2018	Progress update	Individual workstreams on track, final implementation	W. Knottenbelt
2h	28/02/2018	Report 2	Finalising report 2	None
4h	21/02/2018	Integration, Testing, Backtesting	Integrating xWrap with Strategy, Testing strategies, Back-tester	None
4h	18/02/2018	Interaction front-end with strategy	xWrap update, MessageHub status, Web Socket clients	None
3h	13/02/2018	Workstream communication	MessageHub implementation	None
3h	09/02/2018	Workstream communication	Data exchange strategy to front-end, paper trading GDAX, web socket clients	None
4h	05/02/2018	Detailed functionality per workstream	MessageHub format, min API spec, strategy backtesting, front-end data requirements	None
30 min	25/01/2018	Project Outline	Expand on main project objectives	W. Knottenbelt
2h	21/01/2018	Report	Report 1, further architecture decisions	None
3h	17/01/2018	Further implementation	Further implementation, front-end wireframes, xWrap optimisation	None
3h	14/01/2018	Further implementation	Efficient frontier discussion, data wish lists, communication of workstreams (message-Hub)	None
4h	11/01/2018	Starting implementation	Implementing the efficient frontier, High-level architecture, Report One To Dos	None
2h	09/01/2018	Term outline	Main project schedule and milestones, election of responsibilities	None
2h	18/12/2017	Internal Project Kick-off	Problem breakdown, overview of trading processes, value chain of the product	None

E.1 Commit Log

E.1.1 Strategy

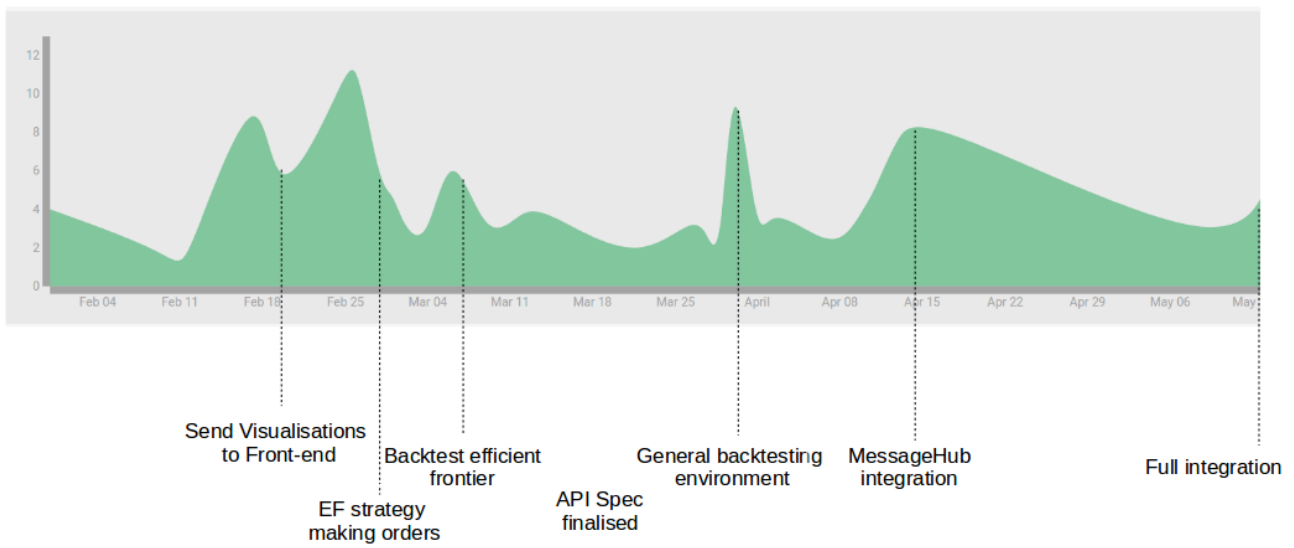


Figure 23: Strategy - Commit History January 17th 2018 - May 15th 2018

E.1.2 Front-end

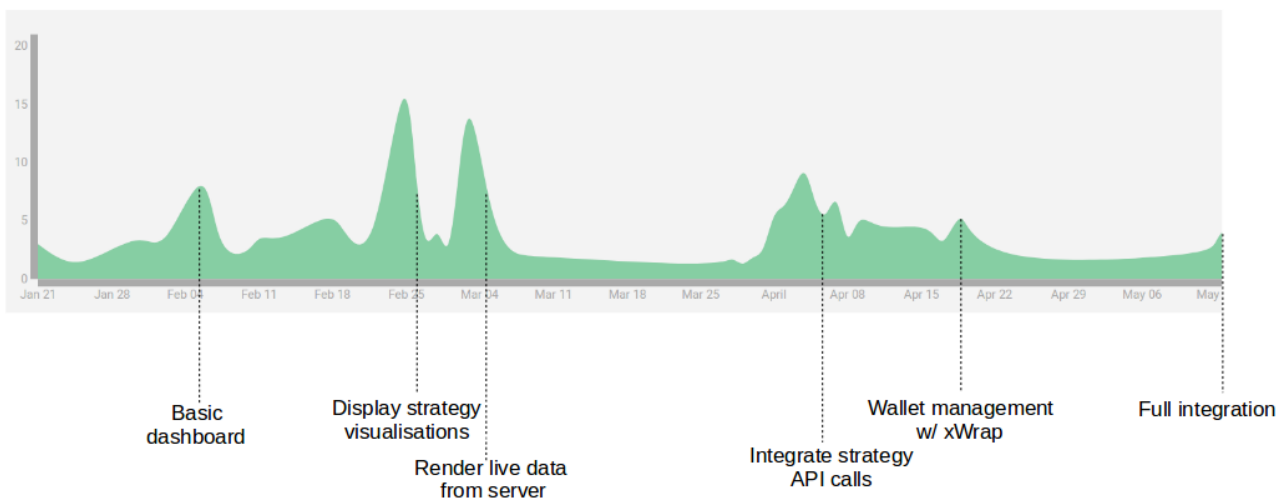


Figure 24: Front-end - Commit History January 17th 2018 - May 15th 2018

E.1.3 xWrap

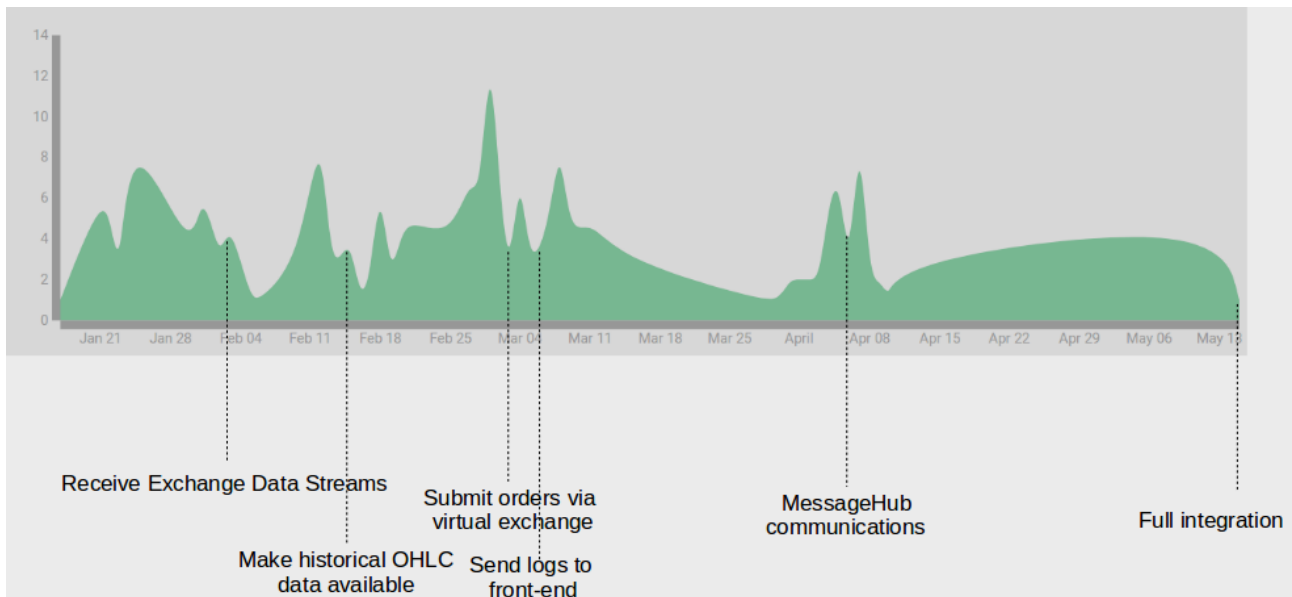


Figure 25: xWrap - Commit History January 17th 2018 - May 15th 2018

E.2 Facts and Figures

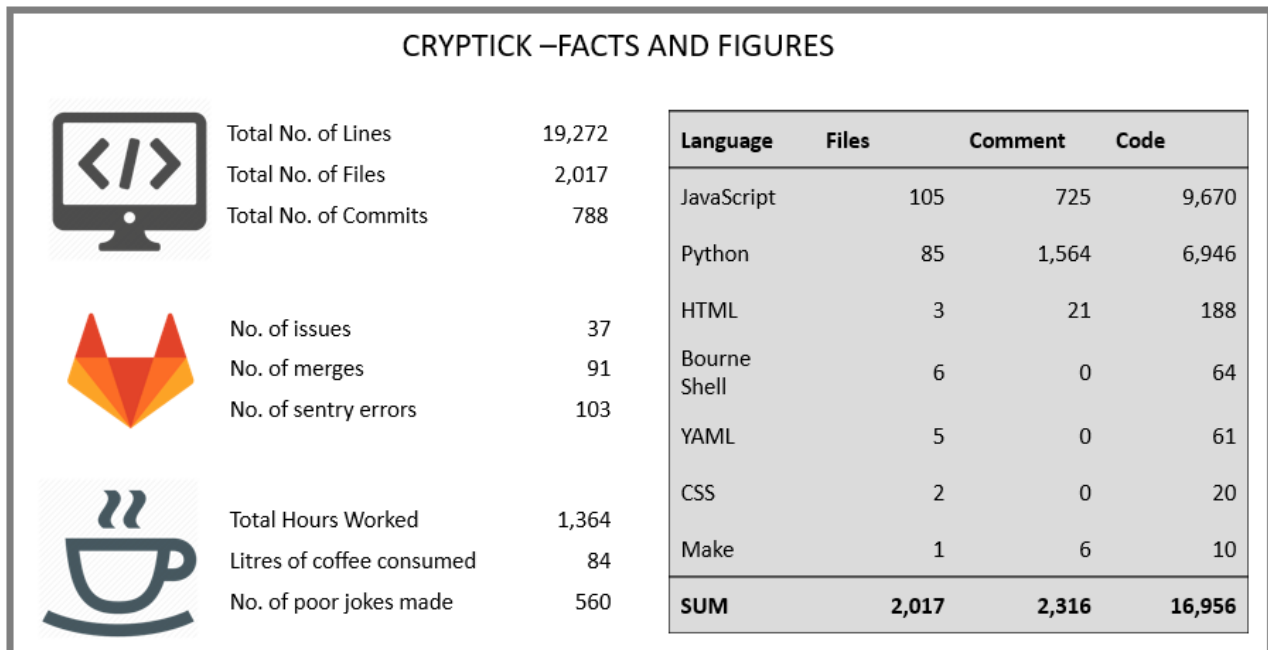


Figure 26: Cryptick - Facts and Figures

xWrap Documentation

Release 0.6.0

May 16, 2018

CONTENTS

1	Contents	3
1.1	Getting Started	3
1.2	Strategy API	5
1.3	Frontend API	6
1.4	Exposed Functions	10
1.5	Notifications	17
1.6	exchange_wrapper package	20
	HTTP Routing Table	41
	Python Module Index	43
	Index	45

The *xWrap* module provides a web-service that abstracts different trading and wallet management APIs from cryptocurrency exchanges to a common RPC friendly API. The wrapper seeks to bridge the different capabilities provided by upstream exchanges such that client code can largely be exchange agnostic.

xWrap also provides basic paper trading functionality against supported cryptocurrency exchanges.

CONTENTS

1.1 Getting Started

To run xWrap in development make sure a valid virtual environment is activated with atleast Python 3.6.

```
# build the dashboard
$ cd dashboard/
$ npm install
$ npm run build
```

```
$ virtualenv --python=/usr/bin/python3 env
$ . env/bin/activate
$ pip install -r requirements.txt
$ export INFLUXDB_DSN="influxdb://influx.cryptick.fun/cryptick"
$ export SQLITE_DB="xwrap.db"
$ python -m exchange_wrapper.service init_db $SQLITE_DB
$ python -m exchange_wrapper.service run --config /path/to/config.json
```

For production environments, use the docker image.

1.1.1 Configuration

The xWrap service can be configured via environment variables and a configuration file.

Configuration JSON File

Use the `settings-sample.json` file in the root directory as a starting point. The configuration file sets the available currencies and exchange classes to load.

```
{
  "currencies": [
    "ZEC",
    "BTC",
    "ETH",
    "USD",
    "XMR",
    "LTC",
    "USDT",
    "BCH"
  ],

```

(continues on next page)

(continued from previous page)

```
"exchanges": {
  "gdax": {
    "mode": "paper",
    "class": "exchange_wrapper.exchange.gdax.GDAXExchange"
  },
  "poloniex": {
    "mode": "paper",
    "class": "exchange_wrapper.exchange.poloniex.PoloniexExchange"
  },
  "bitfinex": {
    "mode": "paper",
    "class": "exchange_wrapper.exchange.bitfinex.BitfinexExchange"
  }
}
```

Required Environment Variables

SQLite

SQLite is used to store order state information and wallet transactions. the `SQLITE_DB` environment variable must be set to a path for the database file.

To setup the initial database:

```
$ export SQLITE_DB="/path/to/xwrap.db"
$ python -m exchange_wrapper.service init_db $SQLITE_DB
Create Table for WalletSet
Create Table for Order
Create Table for WalletLine
Create Table for VirtualWallet
Create Table for Trade
```

Influx DB (Optional)

InfluxDB is used for logging various time-series datapoints such as historical tick data and logs of wallet balances. To enable InfluxDB support, a DSN must be supplied via the `INFLUXDB_DSN` environment variable.

```
export INFLUXDB_DSN="influxdb://influx.local:8086/cryptick"
```

If InfluxDB is not configured the following features will be unavailable:

- `get_ohlcv` API calls on exchange
- `get_wallet_history` API calls

API Endpoint Listen IPs and Ports

xWrap currently exposes a RPC interface and a REST/SocketIO interface. The listen IP and port can be configured via environment variables. By default the services will listen for global traffic (binding to `0.0.0.0`) and the RPC service will be on port 4200 and the REST/SocketIO service on port 4242.

```
export RPC_HOST="172.16.0.3"
export RPC_PORT="2000"
export WEB_HOST="172.16.2.3"
export WEB_PORT="2001"
```

1.1.2 Running Tests

Use nose2 to run tests.

```
. env/bin/activate
nose2 --with-coverage
```

1.1.3 Building a Docker Image

```
$ docker build -t xwrap .
```

1.2 Strategy API

The strategy connects to xWrap via a Websocket Stream that supports the JSON-RPC protocol.

Messages on the stream interface should be assumed to be asynchronous. I.e. responses may be out of order. See the [spec](#).

JSON-RPC supports two main types of message:

- A remote procedure call where the message will elicit a response
- A notification where the notification will not be responded to

Methods accessible via this RPC are described [here](#)

1.2.1 Request Format

Each message expecting a response should have a unique id generated by the client. This id will be used to match requests to responses and must be unique to the connection.

```
{
  "jsonrpc": "2.0",
  "method": "make_order",
  "params": {...},
  "id": 240
}
```

The params object contains the call specific parameters. If id is omitted, the message is considered to be a notification and will not elicit a response.

1.2.2 Response Format

On success:

```
{
  "jsonrpc": "2.0",
  "result": "accepted",
  "id": 240
}
```

On failure:

```
{
  "jsonrpc": "2.0",
  "error": {
    "code": -32000,
    "message": "Insufficient balance in wallet",
  },
  "id": 240
}
```

1.3 Frontend API

The frontend can utilise a read-only API to observe the status of xWrap.

1.3.1 REST Endpoints

The following REST endpoints are served by a `sanic` web server.

Provides a HTTP API to xWrap

`exchange_wrapper.http_server.create_new_order(request)`

POST /xwrap/v1/orders/new

Maps to `make_order()`

`exchange_wrapper.http_server.deactivate_wallet_set(request)`

POST /xwrap/v1/wallets/new

Maps to `deactivate_wallet_set()`

`exchange_wrapper.http_server.get_allocated_wallet_sets(request)`

GET /xwrap/v1/wallets/allocated

Maps to `get_allocated_wallet_set()`

`exchange_wrapper.http_server.get_book(request, exchange, pair)`

GET /xwrap/v1/(str: exchange)/book/

str: pair Maps to `get_book()`

`exchange_wrapper.http_server.get_exchange_pricing(request, exchange)`

GET /xwrap/v1/(str: *exchange*)/pricing

The best bid offer from the exchange (*exchange*).

Example request:

```
GET /xwrap/v1/gdax/pricing HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
  "BCH_BTC": {
    "bid": 0.1145,
    "ask": 0.1204
  },
  "BCH_USD": {
    "bid": 1259.33,
    "ask": 1273.73
  }
}
```

Status Codes

- 200 OK – no error
- 404 Not Found – there's no such exchange

`exchange_wrapper.http_server.get_ohlc(request, exchange, pair)`

GET /xwrap/v1/ohlc/(str: *exchange*)/

str: *pair* OHLC data for the specified exchange and pair.

Returned datapoints have the following format: [*time, low, high, open, close, volume, received_at*]

Example request:

```
GET /xwrap/v1/ohlc/gdax/BTC_USD HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
  "exchange": "gdax",
  "pair": "BTC_USD",
  "generated_at": 1520551670
  "points": [
    [1520525700, 9879.59, 9879.59, 9857.2, 9857.21, 19.04536155, 1520551661]
  ]
}
```


Query Parameters

- **start** – an integer unix timestamp in seconds
- **end** – an integer unix timestamp in seconds
- **received_before** – an integer unix timestamp to filter by *received_at*
- **freq** – frequency of datapoints - valid are *highest* and *daily*

Status Codes

- **200 OK** – no error
- **400 Bad Request** – *start* or *end* set to a non-integer value
- **503 Service Unavailable** – underlying backend database not reachable/configured

`exchange_wrapper.http_server.get_total_wallet_sets(request)`

GET /xwrap/v1/wallets/total

Maps to `get_total_wallet_set()`

`exchange_wrapper.http_server.get_unallocated_exchange(request)`

GET /xwrap/v1/wallets/unallocated_exchange

Maps to `:py:meth:~exchange_wrapper.service.ExchangeWrapper.get_unallocated_per_exchange`

`exchange_wrapper.http_server.get_unallocated_wallet_sets(request)`

GET /xwrap/v1/wallets/unallocated

Maps to `get_unallocated_wallet_set()`

`exchange_wrapper.http_server.get_wallet_set(request, set_id)`

GET /xwrap/v1/wallets/(int: set_id)

Maps to `get_wallet_set()`

`exchange_wrapper.http_server.get_wallet_set_history(request, set_id)`

GET /xwrap/v1/wallets/history/(int: set_id)

Get history of wallet set state.

Example request:

```
GET /xwrap/v1/wallets/history/4 HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
  "timestamp":1523146996.0901162624,
  "run_id":"test02",
  "id":4,
```

(continues on next page)

(continued from previous page)

```

"current":{
  "timestamp":1523146996.0630502701,
  "description":"Logical Wallet for Run: test02",
  "id":4,
  "run_id":"test02",
  "wallets":[
    {
      "currency":"BTC",
      "exchange":"gdax",
      "id":30,
      "balance":"883",
      "available":"883",
      "hold":"0",
      "present_value":{
        "available_usd":"6167023",
        "balance_usd":"6167023",
        "hold_usd":"0"
      }
    },
    ...
  ],
  "type":"logical"
},
"data":[
  [
    1523146633,
    {
      "currency":"BTC",
      "id":"30",
      "balance":882.93316402,
      "available":882.93316402,
      "hold":0,
      "present_value":{
        "balance_usd":6125560.7293481147,
        "available_usd":6125560.7293481147,
        "hold_usd":0
      }
    },
    ...
  ]
]
}

```

Status Codes

- **200 OK** – no error
- **404 Not Found** – Cannot find wallet set with *id*
- **503 Service Unavailable** – underlying backend database not reachable/configured

exchange_wrapper.http_server.**get_wallet_sets** (*request*)

GET /xwrap/v1/wallets/sets

Maps to *get_all_wallet_sets()*

exchange_wrapper.http_server.**list_exchanges** (*request*)

GET /xwrap/v1/exchanges

Maps to `get_exchanges()`

`exchange_wrapper.http_server.list_open_orders(request)`

GET /xwrap/v1/orders/open

Maps to `list_open_orders()`

`exchange_wrapper.http_server.list_recent_orders(request)`

GET /xwrap/v1/orders/recent

Maps to `list_recent_orders()`

`exchange_wrapper.http_server.main(request)`

GET /xwrap/v1/status

Maps to `get_status()`

`exchange_wrapper.http_server.make_logical_transaction(request)`

POST /xwrap/v1/wallets/virtual/transaction

Maps to `add_virtual_wallet_tx()`

`exchange_wrapper.http_server.make_transaction(request)`

POST /xwrap/v1/wallets/new

Maps to `create_wallet_set()`

`exchange_wrapper.http_server.metrics(request)`

Metrics endpoint for prometheus.

`exchange_wrapper.http_server.notify(sender, *, notification, payload)`

Callback for notifications to be sent to socketio clients.

`exchange_wrapper.http_server.uri_map(request)`

GET /uri_map

Returns a list of URIs and the docstring of the handler.

1.3.2 SocketIO Channels

Notifications described [here](#) are broadcast on SocketIO channels. The channel name will be the notification name, eg: `order` or `trade`.

1.4 Exposed Functions

xWrap exposes a number of functions to client code via a JSON-RPC or REST interface. Both API mechanisms expose the same underlying python functions as described here. These function calls only accept keyword arguments and will use `Keyword-Only Arguments` as specified in PEP 3102.

`connection_id` is a special keyword argument inserted by xWrap to all method calls. It is used in situations where the function call can result in later asynchronous notifications that need to be sent to originating connection.

1.4.1 Common Conventions

- A `pair` is represented in the format `BASE_QUOTE` where `BASE` and `QUOTE` will be valid currency codes.
- An `exchange` value must correspond to the name of a loaded exchange.
- If an API call that specifies an `exchange` parameter is made before that exchange is in the `ready` state, the API call will raise an error.

1.4.2 Utility Functions

Get Status (`get_status`)

`ExchangeWrapper.get_status(**kwargs)`
Get the status of xWrap and configuration information.

Returns: A `dict` with status information.

```
{
  "status": "online",
  "version": "0.4.1",
  "config": {
    "currencies": ["BTC", "LTC", "USD"],
    "exchanges": ["gdax"]
  }
}
```

Echo (`echo`)

Will echo any parameters that this method is called with to the sender as a response. Use to test RPC interfaces.

Eg: calling `echo(foo="bar", baz=9)` will return a `dict` `{"foo": "bar", "baz": 9}`.

`ExchangeWrapper.echo(*, connection_id, **kwargs)`
Echoes sent keyword arguments back to the sender.

Echo Notify (`echo_notify`)

Will echo any parameters that this method is called with to the sender as a notification. Use to test notifications.

Eg: calling `echo_notify(foo="bar", baz=9)` will return a `dict` `{"status": "ok", "receivers": 2}` if there were 2 receivers listening for notifications. Each receiver would receive a `echo_ret` notification with the payload `{"foo": "bar", "baz": 9}`.

`ExchangeWrapper.echo_notify(*, connection_id, **kwargs)`
Creates an `echo_ret` notification with the sent keywords.

Returns: A dictionary with `"status": "ok"` and the number of notification `receivers` that would receive this notification.

1.4.3 Exchange Functions

Get Exchanges (get_exchanges)

ExchangeWrapper.**get_exchanges** (*, mode=None, **kwargs)

Return a list of dictionaries describing the registered exchanges on this server.

Args:

param mode Filters the exchanges by their mode. Valid modes are `paper` or `live`.

Returns: A list of dict's corresponding to exchanges supported by the system.

- `exchange`: The exchange id.
- `class`: The python class handling the exchange.
- `status`: The status of the exchange.
- `pairs`: A list of pairs supported by the exchange.
- `currencies`: List of currencies supported by the exchange.
- `mode`: The mode the exchange is loaded in.

```
[
  {
    "exchange": "gdax",
    "class": "xwrap.exchange.gdax.GDAXExchange",
    "status": "starting",
    "pairs": ["BTC_USD", "LTC_USD"],
    "currencies": ["BTC", "LTC", "USD"],
    "mode": "paper"
  },
  ...
]
```

A few notes:

- See `get_status()` in the Exchange class for valid state values.
- If the exchange status is not ready then some data might be out of date, and api calls can fail.
- type can be `paper` or `live`. A paper exchange will simulate trades against real-time orderbooks.

1.4.4 Wallet Functions

Get Wallet Set (get_wallet_set)

ExchangeWrapper.**get_wallet_set** (set_id=None, aggregate_only=False, **kwargs)

Return the wallet set with the specific ID.

Get All Wallet Sets (get_all_wallet_sets)

ExchangeWrapper.**get_all_wallet_sets** (aggregate_only=False, **kwargs)

Get all the active wallet sets on the exchange.

Returns: A dict of WalletSet objects keyed by id.

Example:

```
{
  "1": [WalletSet Object],
  "2": [WalletSet Object],
  "3": [WalletSet Object],
  "total": [WalletSet Object],
  "unallocated": [WalletSet Object],
  "allocated": [WalletSet Object],
}
```

The total, unallocated and allocated wallet sets are described in the corresponding sibling function documentation.

A WalletSet object is represented as:

```
{
  "timestamp": 153998293,
  "description": "This is a set of logical wallets",
  "id": 1,
  "wallets": [
    [Wallet Object],
    [Wallet Object],
    [Wallet Object]
  ],
  "type": "logical"
}
```

A Wallet object is represented as:

```
{
  "currency": "IOTA",
  "exchange": "bitfinex",
  "id": 24,
  "balance": "1000000.00",
  "available": "1000000.00",
  "hold": "0.00",
  "present_value": {
    "balance_usd": "10.00",
    "available_usd": "10.00",
    "hold_usd": "0.00"
  }
}
```

See `_wallet_to_dict()` for field descriptions.

For more information on field values see documentation for *WalletSet* and *VirtualWallet*.

Get Allocated Wallets (`get_allocated_wallet_set`)

`ExchangeWrapper.get_allocated_wallet_set(**kwargs)`

Get a aggregate of all active wallet sets aggregated by currency.

Returns: A WalletSet object as described in `get_all_wallet_sets()`.

Get Unallocated Wallets (`get_unallocated_wallet_set`)

`ExchangeWrapper.get_unallocated_wallet_set(**kwargs)`

Get a wallet set of unallocated exchange funds aggregated by currency.

Get All Wallets (`get_total_wallet_set`)

`ExchangeWrapper.get_total_wallet_set(**kwargs)`

Get the total funds (allocated+unallocated). This will equal the aggregate of exchange wallet balances.

Create Wallet Set (`create_wallet_set`)

`ExchangeWrapper.create_wallet_set(*, run_id, wallets, **kwargs)`

Create a new wallet set.

Args:

param run_id The `strategy_run_id` to associate this set with.

param wallets A list of Wallet objects.

Each Wallet object in the `wallets` list must contain the following fields: `exchange`, `currency` and `allocation`.

Example:

```
{
  "run_id": "my_great_run_01",
  "wallets": [
    {
      "exchange": "gdax",
      "currency": "USD",
      "allocation": "500.0"
    },
    {
      "exchange": "gdax",
      "currency": "LTC",
      "allocation": "5.0"
    },
    {
      "exchange": "gdax",
      "currency": "ETH",
      "allocation": "10.0"
    },
  ]
}
```

Returns: A dict with the id of the new wallet set.

```
{ "id": 5 }
```

If the wallet cannot be allocated, then an error response will be returned:

```
{"error": "error message"}
```

Possible errors are:

- `insufficient_funds`: Insufficient unallocated funds.
- `bad_wallet_config`: One or more bad wallet/exchange pairs

1.4.5 Order Functions

Make Order (`make_order`)

`ExchangeWrapper.make_order` (*, *connection_id*, *run_id*, *exchange*, *client_oid*, *type*, *pair*, *size*, *side*, *wallet_set_id=None*, *limit=0*, *cancel_after=0*, ***kwargs*)

Make an order on the specified exchange.

Args:

param exchange Identifier for the exchange.

param connection_id Identifier for connection sending this request. Use to deliver Trade and Order notifications.

param client_oid Unique identifier for this order from the client.

param run_id identifier for a strategy instance.

param wallet_set_id Wallet Set to run this transaction on.

param side *buy* or *sell*

param type A valid order type string.

param pair A pair name supported by this exchange.

param size Order size.

param limit Order limit value for limit orders.

param cancel_after Seconds after which this order should cancel.

param *kwargs*** Other keyword arguments which are ignored.

Returns: A tuple (*status*, *message*) is returned. If *status* returns as *invalid* or *rejected* then *message* will return a descriptive reason.

Several different order types are supported:

- `limit`: Standard limit order that is good till cancelled
- `limit-immediate-cancel`: Cancels the remaining size on an order instead of putting on the book
- `limit-fill-kill`: Rejects the order if the entire size cannot be matched
- `market`: Executed at the best price available

Of the above, only the plain `limit` order will be left on the order book.

List Open Orders(`list_open_orders`)

`ExchangeWrapper.list_open_orders` (***kwargs*)

Get a list of open orders.

Returns: A list of Order objects, eg:

```
[
  {
    "id":30,
    "created_at":"2018-04-07T02:29:58.933222",
    "wallet_set_id":2,
    "exchange":"gdax",
    "client_order_id":"8eed73b3-309c-4153-b6e5-0735b484aad3",
    "run_id":"asda",
    "type":"limit",
```

(continues on next page)

(continued from previous page)

```

        "pair": "LTC_USD",
        "size": "1",
        "side": "buy",
        "limit": "200",
        "cancel_after": 0,
        "status": "done",
        "done_reason": "filled",
        "done_at": "2018-04-07T02:29:59.119495",
        "exchange_at": "2018-04-07T02:29:59.070921",
        "last_updated_at": "2018-04-07T02:29:32.387727",
        "filled_size": "1",
        "avg_price": "117.55"
    },
    ...
]

```

The field descriptions are described in *Order*.

List Recent Orders(*list_recent_orders*)

`ExchangeWrapper.list_recent_orders` (*, *include_open=False*)

Get a list of open orders.

Returns: A list of Order objects. See *list_open_orders* for example.

1.4.6 Pricing Functions

Convert to USD (*convert_to_usd*)

`ExchangeWrapper.convert_to_usd` (*, *currency*, *value*, *side='bid'*)

Convert the specified currency value into USD.

Args:

param currency A currency code to convert into USD.

param value The value to convert.

Returns: The converted currency value. If the currency cannot be converted `None` will be returned.

Get Book (*get_book*)

`ExchangeWrapper.get_book` (*, *exchange*, *pair*, ***kwargs*)

Get the recent order book snap.

Args:

param exchange A valid exchange.

param pair A exchange on the pair.

Returns: A `dict` with `asks` and `bids` keys that reference lists of [`level`, `size`].

Example:

```
{
  "bids": [
    [290, 0.01],
    [291, 0.01],
    ...
  ],
  "asks": [
    [300, 0.01],
    [301, 0.01],
    ...
  ]
}
```

Get Pricing (`get_pricing`)

`ExchangeWrapper.get_pricing(*, exchange, **kwargs)`

Returns pricing from an exchange.

See exchange documentation for how this is computed.

Args:

param exchange The exchange name.

Returns: A dict with pair ids as keys and {"bids": value, "asks": value} as values:

```
{"ETH_USD": {"bids": 105, "asks": 102}, ...}
```

1.4.7 Virtual Wallet Functions

Add Virtual Wallet Transaction (`add_virtual_wallet_tx`)

`ExchangeWrapper.add_virtual_wallet_tx(*, exchange, base, movement, wallet_id=None, **kwargs)`

Add a transaction to a paper exchange virtual wallet.

Args:

param exchange Identifier for the exchange.

param base Currency for the selected wallet.

param movement The credit (positive) or debit (negative) amount.

param wallet_id Virtual wallet Id. Optional.

param **kwargs Other keyword arguments which are ignored.

Returns: Returns the `id` for the wallet trasaction. This id can be used with the `modify_vwallet_line()` call.

1.5 Notifications

`xWrap` generates notifications to certain events such as executed trades or completed orders that client code may wish to subscribe to.

In the sections titles below, the notification name is shown in monospace.

1.5.1 Order Status

Order (`order`)

Order notifications are issued on the creation or update of an order placed by a `make_order` command. They contain the following fields which are generated by the `to_dict()`.

`Order.to_dict()`

Encode an order as a `dict`.

Keys:

- `id`: xWrap Order ID
- `created_at`: Timestamp for order creation (ISO8601 format)
- `wallet_set_id`: Wallet Set ID (Optional)
- `exchange`: Exchange name
- `client_order_id`: Client Order ID
- `run_id`: Run ID associated with this order
- `type`: Order type (Options: *limit*, *immediate-or-cancel limit-fill-kill*, *market*)
- `pair`: A pair in `BASE_QUOTE` format (BTC_USD)
- `size`: Order size
- `side`: Order size (buy or sell)
- `limit`: Limit price (if market order, then 0)
- `cancel_after`: Seconds after which orders automatically cancel
- `status`: Order status
- `done_reason`: Descriptive reason for status
- `done_at`: Timestamp at which order closed (ISO8601 format)
- `exchange_at`: Timestamp at which order arrived at exchange
- `last_updated_at`: Timestamp for last order update
- `filled_size`: Size of order which has been filled
- `avg_price`: Average price for fill

Trade (`trade`)

Trade notifications are issued at every fill from an exchange.

`Trade.to_dict()`

Encode trade as a `dict`.

Keys:

- `client_order_id`: Client Order ID
- `run_id`: Run ID for the order associated with this trade
- `type`: Type of Order associated with this Trade
- `side`: Side for associated Order

- pair: Pair for associated Order
- exec_amount: Size of this trade
- exec_price: Amount this trade executed at
- fee: Exchange fee for this order
- maker: True if this order was a liquidity maker
- ext_id: Exchange ID for this trade (if any)

1.5.2 General

Log Messages (logmessages)

Log Messages are issued from the python logging library. Any messages from the `xwrap` logger greater or equal to WARNING or any messages from the `user` logger are emitted by default.

Example:

```
{
  "created":1523120885.1184688,
  "filename":"virtual.py",
  "funcName":"trade_executor",
  "levelname":"INFO",
  "levelno":20,
  "lineno":188,
  "module":"virtual",
  "msecs":118.4687614440918,
  "name":"user.orders.test01",
  "pathname":"./exchange_wrapper/exchange/virtual.py",
  "process":3067,
  "processName":"MainProcess",
  "relativeCreated":15570915.82942009,
  "thread":139932965160256,
  "threadName":"MainThread",
  "message":"Order 1faaae49-64bf-4dc5-847b-b4a92fd791fc is done"
}
```

Exchange Status (exchstatus)

Exchange notifications are issued when an Exchange changes state.

Example:

```
{
  "exchange": "gdax",
  "from": "starting",
  "to": "ready"
}
```

1.6 exchange_wrapper package

1.6.1 Subpackages

exchange_wrapper.util package

Submodules

exchange_wrapper.util.token_bucket module

Token Bucket —

Implement a token bucket algorithm to rate limit connections. This implementation will be local to a thread and will provide a method that can block until a token is available.

Does not support bursts.

```
class exchange_wrapper.util.token_bucket.TokenBucket (tokens_per_second)  
    Bases: object
```

```
    request_token ()
```

Decrements token and returns if tokens are available. Otherwise blocks until a token can be awarded.

Module contents

```
exchange_wrapper.util.dec2str (decimal_value, dp=8)
```

```
exchange_wrapper.util.dt_to_msts (dt)
```

```
exchange_wrapper.util.dt_to_str (dt)
```

```
exchange_wrapper.util.dt_to_ts (dt)
```

```
exchange_wrapper.util.fnmatch_dicts (param, ref)
```

1.6.2 Submodules

1.6.3 exchange_wrapper.client module

```
class exchange_wrapper.client.Client (name='generic')  
    Bases: object
```

```
    add_virtual_wallet_tx (exchange, base, amount)
```

```
    cancel_order (exchange, client_order_id)
```

```
    get_wallets ()
```

```
    limit_buy (exchange, pair, size, limit, oid=None, type='limit')
```

```
    limit_sell (exchange, pair, size, limit, oid=None, type='limit')
```

```
    parse_recv (json_str)
```

```
exchange_wrapper.client.collate_responses (resp, ws)
```

```
exchange_wrapper.client.execute_via_rpc (ws_url, client, rpc_msg, timeout=0.5, filter_notifications=None)
```

`exchange_wrapper.client.websocket_query` (*url, message, count=1*)

`exchange_wrapper.client.websocket_query_timeout` (*url, message, timeout*)

1.6.4 exchange_wrapper.config module

class `exchange_wrapper.config.Config` (*envvar_config_key='env'*)

Bases: `object`

get (*key, default=None, cast=None*)

getenv (*key, default=None, cast=None*)

load (*config_file_path*)

`exchange_wrapper.config.cast_bool` (*value*)

`exchange_wrapper.config.recursive_update` (*old_dict, new_dict*)

1.6.5 exchange_wrapper.connection module

Models a HTTP connection to an exchange. Accepts requests in a queue and pushes data to the exchange based on a rate-limit.

class `exchange_wrapper.connection.Connection` (*name, config, server_address="", max_queue=0, rate_limit=None*)

Bases: `object`

`config` is a dict containing secrets for HMAC auth

init ()

Init the session and test API?

run ()

Start the worker thread

send_request (*method, endpoint, priority=99, **kwargs*)

Enqueue a HTTP request for dispatch to the server.

Args: `method` (`str`): A valid HTTP method. `endpoint` (`str`): The endpoint path on the server. ****kwargs:** a list of keyword arguments to pass the underlying

`requests.Request` constructor.

Returns: Returns a (`concurrent.futures.Future`) object for the request if sufficient space was available in the queue, otherwise returns `None`. The Future's value will be set to `requests.Response` object on completion of the request.

send_request_async (**args, loop=None, **kwargs*)

stop ()

exception `exchange_wrapper.connection.ConnectionError` (*e*)

Bases: `Exception`

`exchange_wrapper.connection.connection_worker` (*connection*)

Thread to process requests on a connection

1.6.6 exchange_wrapper.db module

```
class exchange_wrapper.db.BaseModel (*args, **kwargs)
    Bases: peewee.Model

    DoesNotExist
        alias of BaseModelDoesNotExist

    id = <peewee.AutoField object>
```

1.6.7 exchange_wrapper.http_server module

Provides a HTTP API to xWrap

```
class exchange_wrapper.http_server.HttpServer (xwrap, host, port)
    Bases: object

    notify_connect (sid)

    start ()
```

```
exchange_wrapper.http_server.create_new_order (request)
```

```
POST /xwrap/v1/orders/new
    Maps to make_order ()
```

```
exchange_wrapper.http_server.deactivate_wallet_set (request)
```

```
POST /xwrap/v1/wallets/new
    Maps to deactivate_wallet_set ()
```

```
exchange_wrapper.http_server.get_allocated_wallet_sets (request)
```

```
GET /xwrap/v1/wallets/allocated
    Maps to get_allocated_wallet_set ()
```

```
exchange_wrapper.http_server.get_book (request, exchange, pair)
```

```
GET /xwrap/v1/(str: exchange)/book/
    str: pair Maps to get_book ()
```

```
exchange_wrapper.http_server.get_exchange_pricing (request, exchange)
```

```
GET /xwrap/v1/(str: exchange)/pricing
    The best bid offer from the exchange (exchange).
```

Example request:

```
GET /xwrap/v1/gdax/pricing HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```

HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
  "BCH_BTC": {
    "bid": 0.1145,
    "ask": 0.1204
  },
  "BCH_USD": {
    "bid": 1259.33,
    "ask": 1273.73
  }
}

```

Status Codes

- 200 OK – no error
- 404 Not Found – there's no such exchange

`exchange_wrapper.http_server.get_exchange_unallocated(request)`

`exchange_wrapper.http_server.get_ohlcv(request, exchange, pair)`

GET `/xwrap/v1/ohlcv/(str: exchange) /`
str: `pair` OHLC data for the specified exchange and pair.

Returned datapoints have the following format: `[time, low, high, open, close, volume, received_at]`

Example request:

```

GET /xwrap/v1/ohlcv/gdax/BTC_USD HTTP/1.1
Host: example.com
Accept: application/json, text/javascript

```

Example response:

```

HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
  "exchange": "gdax",
  "pair": "BTC_USD",
  "generated_at": 1520551670
  "points": [
    [1520525700, 9879.59, 9879.59, 9857.2, 9857.21, 19.04536155, 1520551661]
  ]
}

```

Query Parameters

- **start** – an integer unix timestamp in seconds
- **end** – an integer unix timestamp in seconds
- **received_before** – an integer unix timestamp to filter by `received_at`

- **freq** – frequency of datapoints - valid are *highest* and *daily*

Status Codes

- **200 OK** – no error
- **400 Bad Request** – *start* or *end* set to a non-integer value
- **503 Service Unavailable** – underlying backend database not reachable/configured

exchange_wrapper.http_server.**get_total_wallet_sets** (*request*)

GET /xwrap/v1/wallets/total

Maps to `get_total_wallet_set()`

exchange_wrapper.http_server.**get_unallocated_exchange** (*request*)

GET /xwrap/v1/wallets/unallocated_exchange

Maps to :py:meth:`~exchange_wrapper.service.ExchangeWrapper.get_unallocated_per_exchange`

exchange_wrapper.http_server.**get_unallocated_wallet_sets** (*request*)

GET /xwrap/v1/wallets/unallocated

Maps to `get_unallocated_wallet_set()`

exchange_wrapper.http_server.**get_wallet_set** (*request, set_id*)

GET /xwrap/v1/wallets/(int: set_id)

Maps to `get_wallet_set()`

exchange_wrapper.http_server.**get_wallet_set_history** (*request, set_id*)

GET /xwrap/v1/wallets/history/(int: set_id)

Get history of wallet set state.

Example request:

```
GET /xwrap/v1/wallets/history/4 HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
  "timestamp":1523146996.0901162624,
  "run_id":"test02",
  "id":4,
  "current":{
    "timestamp":1523146996.0630502701,
    "description":"Logical Wallet for Run: test02",
    "id":4,
    "run_id":"test02",
    "wallets":[
```

(continues on next page)

(continued from previous page)

```

    {
      "currency": "BTC",
      "exchange": "gdax",
      "id": 30,
      "balance": "883",
      "available": "883",
      "hold": "0",
      "present_value": {
        "available_usd": "6167023",
        "balance_usd": "6167023",
        "hold_usd": "0"
      }
    },
    ...
  ],
  "type": "logical"
},
"data": [
  [
    1523146633,
    {
      "currency": "BTC",
      "id": "30",
      "balance": 882.93316402,
      "available": 882.93316402,
      "hold": 0,
      "present_value": {
        "balance_usd": 6125560.7293481147,
        "available_usd": 6125560.7293481147,
        "hold_usd": 0
      }
    }
  ],
  ...
]
]
}

```

Status Codes

- **200 OK** – no error
- **404 Not Found** – Cannot find wallet set with *id*
- **503 Service Unavailable** – underlying backend database not reachable/configured

`exchange_wrapper.http_server.get_wallet_sets` (*request*)

GET /xwrap/v1/wallets/sets

Maps to `get_all_wallet_sets()`

`exchange_wrapper.http_server.list_exchanges` (*request*)

GET /xwrap/v1/exchanges

Maps to `get_exchanges()`

`exchange_wrapper.http_server.list_open_orders` (*request*)

GET /xwrap/v1/orders/open

Maps to `list_open_orders()`

`exchange_wrapper.http_server.list_recent_orders(request)`

GET /xwrap/v1/orders/recent

Maps to `list_recent_orders()`

`exchange_wrapper.http_server.log_404(request, exception)`

`exchange_wrapper.http_server.main(request)`

GET /xwrap/v1/status

Maps to `get_status()`

`exchange_wrapper.http_server.make_logical_transaction(request)`

POST /xwrap/v1/wallets/virtual/transaction

Maps to `add_virtual_wallet_tx()`

`exchange_wrapper.http_server.make_transaction(request)`

POST /xwrap/v1/wallets/new

Maps to `create_wallet_set()`

`exchange_wrapper.http_server.metrics(request)`

Metrics endpoint for prometheus.

`exchange_wrapper.http_server.notify(sender, *, notification, payload)`

Callback for notifications to be sent to socketio clients.

`exchange_wrapper.http_server.on_connect(sid, environ)`

`exchange_wrapper.http_server.uri_map(request)`

GET /uri_map

Returns a list of URIs and the docstring of the handler.

1.6.8 exchange_wrapper.logical_wallet module

class `exchange_wrapper.logical_wallet.LogicalWalletManager(xwrap)`

Bases: `object`

`get_allocated()`

`get_exchange_total()`

`get_unallocated()`

`get_unallocated_per_exchange()`

`get_wallet_set(id)`

`get_wallet_sets(type)`

1.6.9 exchange_wrapper.matching module

`exchange_wrapper.matching.match_limit_order` (*book, side, size, limit*)
Execute limit order in paper trading environment.

`exchange_wrapper.matching.match_market_order` (*book, side, size*)
Execute market order in paper trading environment.

1.6.10 exchange_wrapper.models module

class `exchange_wrapper.models.Order` (*args, **kwargs)

Bases: `exchange_wrapper.db.BaseModel`

Represent a cryptocurrency order

DoesNotExist

alias of `OrderDoesNotExist`

`avg_price` = <peewee.DecimalField object>

`cancel_after` = <peewee.IntegerField object>

`client_order_id` = <peewee.CharField object>

`created_at` = <peewee.DateTimeField object>

`done_at` = <peewee.DateTimeField object>

`done_reason` = <peewee.CharField object>

`exchange` = <peewee.CharField object>

`exchange_at` = <peewee.DateTimeField object>

`filled_size` = <peewee.DecimalField object>

`get_quote_amounts` ()

`get_wallets` (*wallet_set=None*)

Returns the quote and base wallet from the wallet set.

Args:

param `wallet_set` A wallet set. If *None* given (default), will use `Order.wallet_set`.

Returns: A tuple, (*base_wallet, quote_wallet*) which are `VirtualWallet` objects for the quote and base currency respectively.

`id` = <peewee.AutoField object>

`last_updated_at` = <peewee.DateTimeField object>

`limit` = <peewee.DecimalField object>

`logger`

`orders`

`pair` = <peewee.CharField object>

`remaining_size`

`settled` = <peewee.BooleanField object>

`side = <peewee.CharField object>`
`size = <peewee.DecimalField object>`
`status = <peewee.CharField object>`
`strategy_run_id = <peewee.CharField object>`

`to_dict()`

Encode an order as a `dict`.

Keys:

- `id`: xWrap Order ID
- `created_at`: Timestamp for order creation (ISO8601 format)
- `wallet_set_id`: Wallet Set ID (Optional)
- `exchange`: Exchange name
- `client_order_id`: Client Order ID
- `run_id`: Run ID associated with this order
- `type`: Order type (Options: *limit, immediate-or-cancel limit-fill-kill, market*)
- `pair`: A pair in `BASE_QUOTE` format (BTC_USD)
- `size`: Order size
- `side`: Order size (buy or sell)
- `limit`: Limit price (if market order, then 0)
- `cancel_after`: Seconds after which orders automatically cancel
- `status`: Order status
- `done_reason`: Descriptive reason for status
- `done_at`: Timestamp at which order closed (ISO8601 format)
- `exchange_at`: Timestamp at which order arrived at exchange
- `last_updated_at`: Timestamp for last order update
- `filled_size`: Size of order which has been filled
- `avg_price`: Average price for fill

`type = <peewee.CharField object>`

`unfilled_size`

`update_wallets (wallet_set=None)`

Update the `wallet_set` associated with an order.

Args:

param wallet_set The wallet set to update, if this is set to None (default) then the `wallet_set` from the `Order` object will be used.

`wallet_lines`

`wallet_set = <ForeignKeyField: "order"."wallet_set">`

`wallet_set_id = <ForeignKeyField: "order"."wallet_set">`

```

class exchange_wrapper.models.Trade(*args, **kwargs)
    Bases: exchange_wrapper.db.BaseModel

    Represents a single trade that makes up an order

    DoesNotExist
        alias of TradeDoesNotExist

    exec_amount = <peewee.DecimalField object>
    exec_price = <peewee.DecimalField object>
    ext_id = <peewee.CharField object>
    fee = <peewee.DecimalField object>
    id = <peewee.AutoField object>
    maker = <peewee.BooleanField object>
    order = <ForeignKeyField: "trade"."order">
    order_id = <ForeignKeyField: "trade"."order">
    order_type = <peewee.CharField object>
    to_dict()
        Encode trade as a dict.

    Keys:
        • client_order_id: Client Order ID
        • run_id: Run ID for the order associated with this trade
        • type: Type of Order associated with this Trade
        • side: Side for associated Order
        • pair: Pair for associated Order
        • exec_amount: Size of this trade
        • exec_price: Amount this trade executed at
        • fee: Exchange fee for this order
        • maker: True if this order was a liquidity maker
        • ext_id: Exchange ID for this trade (if any)

class exchange_wrapper.models.VirtualWallet(*args, **kwargs)
    Bases: exchange_wrapper.db.BaseModel

    Funds put on hold will not be available to the user. So if a debit is held then that affects the ‘available’ balance but not the ‘ledger’ balanceself. For a credit on hold, it does not affect the ‘available’ balance but does affect the ‘ledger’ balance.

    DoesNotExist
        alias of VirtualWalletDoesNotExist

    address = <peewee.CharField object>
    allocation = <peewee.DecimalField object>
    available
    balance

```

```
base = <peewee.CharField object>
exchange = <peewee.CharField object>
get_held_for (order)
hold
id = <peewee.AutoField object>
make_transaction (movement, order=None, held=False)
wallet_lines
wallet_set = <ForeignKeyField: "virtualwallet"."wallet_set">
wallet_set_id = <ForeignKeyField: "virtualwallet"."wallet_set">
```

```
class exchange_wrapper.models.WalletLine (*args, **kwargs)
    Bases: exchange_wrapper.db.BaseModel
```

Represent a transaction in a walletself.

Variables

- *date* – Date column for when this transaction was created
- *wallet_id* – Id for this wallet
- *movement* – Transaction amount
- *on_hold* – If this transaction is being held (waiting for settlement)
- *order* – Order that this transaction relates to (if any)

DoesNotExist

alias of WalletLineDoesNotExist

```
date = <peewee.DateTimeField object>
id = <peewee.AutoField object>
movement = <peewee.DecimalField object>
on_hold = <peewee.BooleanField object>
order = <ForeignKeyField: "walletline"."order">
order_id = <ForeignKeyField: "walletline"."order">
wallet = <ForeignKeyField: "walletline"."wallet">
wallet_id = <ForeignKeyField: "walletline"."wallet">
```

```
class exchange_wrapper.models.WalletSet (*args, **kwargs)
    Bases: exchange_wrapper.db.BaseModel
```

A set of logical wallet used by a strategy.

DoesNotExist

alias of WalletSetDoesNotExist

```
active = <peewee.BooleanField object>
aggregate
    Returns a dict of wallets aggregated by currency.
created = <peewee.DateTimeField object>
description = <peewee.CharField object>
```

get_wallet (*exchange, currency*)

Return the wallet for an *exchange* and *currency* from this set.

Args:

param exchange An exchange name.

param currency A currency code.

Returns: A `Wallet` if a matching wallet exists, otherwise `None`.

`id = <peewee.AutoField object>`

`orders`

`run_id = <peewee.CharField object>`

`type = <peewee.CharField object>`

`wallets`

1.6.11 exchange_wrapper.pair module

class `exchange_wrapper.pair.Pair` (*x_id, base, quote, price_prec, min_order, max_order*)

Bases: `object`

Models a cryptocurrency pair available on an exchange

Creates a `Pair` object. A pair string is defined as `{base}_{order}`.

For a sell: `base` is the currency you have, `quote` what you want For a buy: `base` is the currency you want, `quote` what you have

For uniformity, this class maintains all currencies codes and pair codes in uppercase.

Args: `x_id`: An identifier for this pair from the exchange `base`: base currency for this pair `quote`: quote currency for this pair `price_prec`: price precision for the quote in decimal places `min_order`: minimum size for any order `max_order`: maximum size for any order

check_price (*price*)

Check if the given price does not exceed the precision provided by the exchange. To be used to check that any limit prices sent to the exchange are valid.

Args: `price`: any numeric type or convertible string.

Returns: boolean: True if the price is within the precision requirements.

check_size (*order_size*)

Check if a given `order_size` is within the min and max order sizes specified.

Internally converts `order_size` to a fixed precision number. Should check that the `order_size` is a multiple of the `min_order`.

Args: `order_size`: any numeric type or convertible string.

Returns: boolean: True if the order size is valid.

name ()

Returns a currency pair in the format `{base}_{quote}`. All characters will be uppercase.

1.6.12 exchange_wrapper.rpc module

exception exchange_wrapper.rpc.**ApiUsageError**

Bases: `Exception`

data = `None`

jsonrpc_error_code = `-32001`

exception exchange_wrapper.rpc.**MethodNotFoundError**

Bases: `Exception`

data = `None`

jsonrpc_error_code = `-32601`

exception exchange_wrapper.rpc.**ProcessingError**

Bases: `Exception`

data = `None`

jsonrpc_error_code = `-32002`

class exchange_wrapper.rpc.**RpcInterface** (*app_class, config*)

Bases: `object`

Creates a server object that will run an asyncio loop and host the one or more service classes.

audit_loop (*scheduled_at*)

Audit the event loop and log the latency (if any).

Args:

param scheduled_at The time at which this function was scheduled

on the loop via *call_later*

get_connection ()

handle_message (*connection_id, message_str*)

Handle messages from a connection and hand them over an installed service. Pass back any response over the originating connection.

Params:

connection_id: Id which can be used to address the connection this message arrived on.

message_str: The received message in string form.

on_connect_hook (*callback*)

on_disconnect_hook (*callback*)

run_forever ()

send_to (*conn_id, method, args, kwargs, one_way=False*)

send_to_all (*method, args, kwargs*)

shutdown ()

Shuts down the event loop.

1.6.13 exchange_wrapper.service module

class exchange_wrapper.service.ExchangeWrapper (*wrapper*)

Bases: object

Implements a service that gets data from cryptocurrency exchanges and provides RPC and PubSub interfaces via an abstracted API format to interfact with the upstream exchanges.

add_virtual_wallet_tx (*, *exchange, base, movement, wallet_id=None, **kwargs*)

Add a transaction to a paper exchange virtual wallet.

Args:

param exchange Identifier for the exchange.

param base Currency for the selected wallet.

param movement The credit (positive) or debit (negative) amount.

param wallet_id Virtual wallet Id. Optional.

param **kwargs Other keyword arguments which are ignored.

Returns: Returns the *id* for the wallet trasaction. This *id* can be used with the `modify_vwallet_line()` call.

allocate_to_wallet_set (*, *wallet_set_id, exchange, base, movement*)

cancel_order (*, *exchange, client_order_id, **kwargs*)

Request a cancellation of the specified order.

convert_to_usd (*, *currency, value, side='bid'*)

Convert the specified currency value into USD.

Args:

param currency A currency code to convert into USD.

param value The value to convert.

Returns: The converted currency value. If the currency cannot be converted `None` will be returned.

create_wallet_set (*, *run_id, wallets, **kwargs*)

Create a new wallet set.

Args:

param run_id The *strategy_run_id* to associate this set with.

param wallets A `list` of `Wallet` objects.

Each `Wallet` object in the *wallets* list must contain the following fields: `exchange`, `currency` and `allocation`.

Example:

```
{
  "run_id": "my_great_run_01",
  "wallets": [
    {
      "exchange": "gdax",
      "currency": "USD",
      "allocation": "500.0"
    },
    {
```

(continues on next page)

(continued from previous page)

```

        "exchange": "gdax",
        "currency": "LTC",
        "allocation": "5.0"
    },
    {
        "exchange": "gdax",
        "currency": "ETH",
        "allocation": "10.0"
    },
]
}

```

Returns: A `dict` with the `id` of the new wallet set.

```
{ "id": 5 }
```

If the wallet cannot be allocated, then an error response will be returned:

```
{"error": "error message"}
```

Possible errors are:

- `insufficient funds`: Insufficient unallocated funds.
- `bad wallet config`: One or more bad wallet/exchange pairs

deactivate_wallet_set (*, *wallet_set_id*)

Deactivate a wallet set.

echo (*, *connection_id*, ***kwargs*)

Echos sent keyword arguments back to the sender.

echo_notify (*, *connection_id*, ***kwargs*)

Creates an *echo_ret* notification with the sent keywords.

Returns: A dictionary with “*status*”: “*ok*” and the number of notification *receivers* that would receive this notification.

get_all_wallet_sets (*aggregate_only=False*, ***kwargs*)

Get all the active wallet sets on the exchange.

Returns: A `dict` of `WalletSet` objects keyed by `id`.

Example:

```

{
    "1": [WalletSet Object],
    "2": [WalletSet Object],
    "3": [WalletSet Object],
    "total": [WalletSet Object],
    "unallocated": [WalletSet Object],
    "allocated": [WalletSet Object],
}

```

The total, unallocated and allocated wallet sets are described in the corresponding sibling function documentation.

A `WalletSet` object is represented as:

```
{
  "timestamp": 153998293,
  "description": "This is a set of logical wallets",
  "id": 1,
  "wallets": [
    [Wallet Object],
    [Wallet Object],
    [Wallet Object]
  ],
  "type": "logical"
}
```

A Wallet object is represented as:

```
{
  "currency": "IOTA",
  "exchange": "bitfinex",
  "id": 24,
  "balance": "1000000.00",
  "available": "1000000.00",
  "hold": "0.00",
  "present_value": {
    "balance_usd": "10.00",
    "available_usd": "10.00",
    "hold_usd": "0.00"
  }
}
```

See `_wallet_to_dict()` for field descriptions.

get_allocated_wallet_set (**kwargs)

Get a aggregate of all active wallet sets aggregated by currency.

Returns: A WalletSet object as described in `get_all_wallet_sets()`.

get_book (*, exchange, pair, **kwargs)

Get the recent order book snap.

Args:

param exchange A valid exchange.

param pair A exchange on the pair.

Returns: A dict with asks and bids keys that reference lists of [level, size].

Example:

```
{
  "bids": [
    [290, 0.01],
    [291, 0.01],
    ...
  ],
  "asks": [
    [300, 0.01],
    [301, 0.01],
    ...
  ]
}
```

get_exchanges (*, mode=None, **kwargs)

Return a list of dictionaries describing the registered exchanges on this server.

Args:

param mode Filters the exchanges by their mode. Valid modes are `paper` or `live`.

Returns: A list of dict's corresponding to exchanges supported by the system.

- exchange: The exchange id.
- class: The python class handling the exchange.
- status: The status of the exchange.
- pairs: A list of pairs supported by the exchange.
- currencies: List of currencies supported by the exchange.
- mode: The mode the exchange is loaded in.

```
[
  {
    "exchange": "gdax",
    "class": "xwrap.exchange.gdax.GDAXExchange",
    "status": "starting",
    "pairs": ["BTC_USD", "LTC_USD"],
    "currencies": ["BTC", "LTC", "USD"],
    "mode": "paper"
  },
  ...
]
```

get_pricing (*, exchange, **kwargs)

Returns pricing from an exchange.

See exchange documentation for how this is computed.

Args:

param exchange The exchange name.

Returns: A dict with pair ids as keys and {"bids": value, "asks": value} as values:

```
{"ETH_USD": {"bids": 105, "asks": 102}, ...}
```

get_status (**kwargs)

Get the status of xWrap and configuration information.

Returns: A dict with status information.

```
{
  "status": "online",
  "version": "0.4.1",
  "config": {
    "currencies": ["BTC", "LTC", "USD"],
    "exchanges": ["gdax"]
  }
}
```

get_total_wallet_set (**kwargs)

Get the total funds (allocated+unallocated). This will equal the aggregate of exchange wallet balances.

get_unallocated_per_exchange (**kwargs)

Get the unallocated funds deaggregated by exchange.

Returns: The return of this function is a `dict` of `dict` objects:

Example:

```
{
  "gdax": {
    "BTC": WalletObject,
    "LTC": WalletObject,
    "USD": WalletObject
  }
}
```

get_unallocated_wallet_set (**kwargs)

Get a wallet set of unallocated exchange funds aggregated by currency.

get_wallet_set (set_id=None, aggregate_only=False, **kwargs)

Return the wallet set with the specific ID.

handle_request (method, args, kwargs, connection_id)

Entry point for RPC calls.

init (config)

list_open_orders (**kwargs)

Get a list of open orders.

Returns: A list of Order objects, eg:

```
[
  {
    "id":30,
    "created_at":"2018-04-07T02:29:58.933222",
    "wallet_set_id":2,
    "exchange":"gdax",
    "client_order_id":"8eed73b3-309c-4153-b6e5-0735b484aad3",
    "run_id":"asda",
    "type":"limit",
    "pair":"LTC_USD",
    "size":"1",
    "side":"buy",
    "limit":"200",
    "cancel_after":0,
    "status":"done",
    "done_reason":"filled",
    "done_at":"2018-04-07T02:29:59.119495",
    "exchange_at":"2018-04-07T02:29:59.070921",
    "last_updated_at":"2018-04-07T02:29:32.387727",
    "filled_size":"1",
    "avg_price":"117.55"
  },
  ...
]
```

The field descriptions are described in *Order*.

list_recent_orders (*, include_open=False)

Get a list of open orders.

Returns: A list of Order objects. See *list_open_orders* for example.

log_wallet_set (*sender*, *, *wallet_set*, *key*='wallet_set_changes')

Log wallet balances

make_order (*, *connection_id*, *run_id*, *exchange*, *client_oid*, *type*, *pair*, *size*, *side*, *wallet_set_id*=None, *limit*=0, *cancel_after*=0, ***kwargs*)

Make an order on the specified exchange.

Args:

param exchange Identifier for the exchange.

param connection_id Identifier for connection sending this request. Use to deliver Trade and Order notifications.

param client_oid Unique identifier for this order from the client.

param run_id identifier for a strategy instance.

param wallet_set_id Wallet Set to run this transaction on.

param side *buy* or *sell*

param type A valid order type string.

param pair A pair name supported by this exchange.

param size Order size.

param limit Order limit value for limit orders.

param cancel_after Seconds after which this order should cancel.

param **kwargs Other keyword arguments which are ignored.

Returns: A tuple (*status*, *message*) is returned. If *status* returns as *invalid* or *rejected* then *message* will return a descriptive reason.

notify_wallet_set_update (*sender*, *, *wallet_set*)

Create a notification signal with the updated wallet set as JSON.

register_exchange (*klass*, *name*, *config*, *mode*, *currencies*)

shutdown ()

subscribe_notifications (*conn*, *connection_id*)

unsubscribe_notifications (*conn*, *connection_id*)

`exchange_wrapper.service.tag_type_exception` (*tag*, *typ*, *value*)

1.6.14 exchange_wrapper.wallet module

class `exchange_wrapper.wallet.Wallet` (*exchange*, *base*, *available*, *balance*, *on_hold*, *allocation*=0)

Bases: `object`

Dumb object to hold wallet balances

allocation

available

balance

hold

1.6.15 exchange_wrapper.wsock_server module

A websocket server that can track multiple client connections.

class exchange_wrapper.wsock_server.**WebSocketServer** (*message_handler*)

Bases: `object`

add_on_connect_hook (*callback*)

add_on_disconnect_hook (*callback*)

handler (*websocket, path*)

make_server (*host, port, **kwargs*)

receive_handler (*id, websocket, path*)

run_server (**args, **kwargs*)

send_handler (*id, websocket, path*)

send_to (*id, message*)

send_to_all (*message*)

shutdown ()

exchange_wrapper.wsock_server.**address_to_str** (*address_tup*)

1.6.16 Module contents

HTTP ROUTING TABLE

/uri_map

GET /uri_map, 10

/xwrap

GET /xwrap/v1/(str:exchange)/book/(str:pair),
6

GET /xwrap/v1/(str:exchange)/pricing, 6

GET /xwrap/v1/exchanges, 9

GET /xwrap/v1/ohlcv/(str:exchange)/(str:pair),
7

GET /xwrap/v1/orders/open, 10

GET /xwrap/v1/orders/recent, 10

GET /xwrap/v1/status, 10

GET /xwrap/v1/wallets/(int:set_id), 8

GET /xwrap/v1/wallets/allocated, 6

GET /xwrap/v1/wallets/history/(int:set_id),
8

GET /xwrap/v1/wallets/sets, 9

GET /xwrap/v1/wallets/total, 8

GET /xwrap/v1/wallets/unallocated, 8

GET /xwrap/v1/wallets/unallocated_exchange,
8

POST /xwrap/v1/orders/new, 6

POST /xwrap/v1/wallets/new, 10

POST /xwrap/v1/wallets/virtual/transaction,
10

PYTHON MODULE INDEX

e

- exchange_wrapper, 39
- exchange_wrapper.client, 20
- exchange_wrapper.config, 21
- exchange_wrapper.connection, 21
- exchange_wrapper.db, 22
- exchange_wrapper.http_server, 22
- exchange_wrapper.logical_wallet, 26
- exchange_wrapper.matching, 27
- exchange_wrapper.models, 27
- exchange_wrapper.pair, 31
- exchange_wrapper.rpc, 32
- exchange_wrapper.service, 33
- exchange_wrapper.util, 20
- exchange_wrapper.util.token_bucket, 20
- exchange_wrapper.wallet, 38
- exchange_wrapper.wsock_server, 39

A

active (exchange_wrapper.models.WalletSet attribute), 30
 add_on_connect_hook() (exchange_wrapper.wsock_server.WebsocketServer method), 39
 add_on_disconnect_hook() (exchange_wrapper.wsock_server.WebsocketServer method), 39
 add_virtual_wallet_tx() (exchange_wrapper.client.Client method), 20
 add_virtual_wallet_tx() (exchange_wrapper.service.ExchangeWrapper method), 33
 address (exchange_wrapper.models.VirtualWallet attribute), 29
 address_to_str() (in module exchange_wrapper.wsock_server), 39
 aggregate (exchange_wrapper.models.WalletSet attribute), 30
 allocate_to_wallet_set() (exchange_wrapper.service.ExchangeWrapper method), 33
 allocation (exchange_wrapper.models.VirtualWallet attribute), 29
 allocation (exchange_wrapper.wallet.Wallet attribute), 38
 ApiUsageError, 32
 audit_loop() (exchange_wrapper.rpc.RpcInterface method), 32
 available (exchange_wrapper.models.VirtualWallet attribute), 29
 available (exchange_wrapper.wallet.Wallet attribute), 38
 avg_price (exchange_wrapper.models.Order attribute), 27

B

balance (exchange_wrapper.models.VirtualWallet attribute), 29
 balance (exchange_wrapper.wallet.Wallet attribute), 38
 base (exchange_wrapper.models.VirtualWallet attribute), 29
 BaseModel (class in exchange_wrapper.db), 22

C

cancel_after (exchange_wrapper.models.Order attribute), 27
 cancel_order() (exchange_wrapper.client.Client method), 20
 cancel_order() (exchange_wrapper.service.ExchangeWrapper method), 33
 cast_bool() (in module exchange_wrapper.config), 21
 check_price() (exchange_wrapper.pair.Pair method), 31
 check_size() (exchange_wrapper.pair.Pair method), 31
 Client (class in exchange_wrapper.client), 20
 client_order_id (exchange_wrapper.models.Order attribute), 27
 collate_responses() (in module exchange_wrapper.client), 20
 Config (class in exchange_wrapper.config), 21
 Connection (class in exchange_wrapper.connection), 21
 connection_worker() (in module exchange_wrapper.connection), 21
 ConnectionError, 21
 convert_to_usd() (exchange_wrapper.service.ExchangeWrapper method), 33
 create_new_order() (in module exchange_wrapper.http_server), 22
 create_wallet_set() (exchange_wrapper.service.ExchangeWrapper method), 33
 created (exchange_wrapper.models.WalletSet attribute), 30
 created_at (exchange_wrapper.models.Order attribute), 27

D

data (exchange_wrapper.rpc.ApiUsageError attribute), 32
 data (exchange_wrapper.rpc.MethodNotFoundError attribute), 32
 data (exchange_wrapper.rpc.ProcessingError attribute), 32
 date (exchange_wrapper.models.WalletLine attribute), 30
 deactivate_wallet_set() (exchange_wrapper.service.ExchangeWrapper method), 34

- deactivate_wallet_set() (in module exchange_wrapper.http_server), 22
 - dec2str() (in module exchange_wrapper.util), 20
 - description (exchange_wrapper.models.WalletSet attribute), 30
 - DoesNotExist (exchange_wrapper.db.BaseModel attribute), 22
 - DoesNotExist (exchange_wrapper.models.Order attribute), 27
 - DoesNotExist (exchange_wrapper.models.Trade attribute), 29
 - DoesNotExist (exchange_wrapper.models.VirtualWallet attribute), 29
 - DoesNotExist (exchange_wrapper.models.WalletLine attribute), 30
 - DoesNotExist (exchange_wrapper.models.WalletSet attribute), 30
 - done_at (exchange_wrapper.models.Order attribute), 27
 - done_reason (exchange_wrapper.models.Order attribute), 27
 - dt_to_msts() (in module exchange_wrapper.util), 20
 - dt_to_str() (in module exchange_wrapper.util), 20
 - dt_to_ts() (in module exchange_wrapper.util), 20
- ## E
- echo() (exchange_wrapper.service.ExchangeWrapper method), 34
 - echo_notify() (exchange_wrapper.service.ExchangeWrapper method), 34
 - exchange (exchange_wrapper.models.Order attribute), 27
 - exchange (exchange_wrapper.models.VirtualWallet attribute), 30
 - exchange_at (exchange_wrapper.models.Order attribute), 27
 - exchange_wrapper (module), 39
 - exchange_wrapper.client (module), 20
 - exchange_wrapper.config (module), 21
 - exchange_wrapper.connection (module), 21
 - exchange_wrapper.db (module), 22
 - exchange_wrapper.http_server (module), 22
 - exchange_wrapper.logical_wallet (module), 26
 - exchange_wrapper.matching (module), 27
 - exchange_wrapper.models (module), 27
 - exchange_wrapper.pair (module), 31
 - exchange_wrapper.rpc (module), 32
 - exchange_wrapper.service (module), 33
 - exchange_wrapper.util (module), 20
 - exchange_wrapper.util.token_bucket (module), 20
 - exchange_wrapper.wallet (module), 38
 - exchange_wrapper.wsock_server (module), 39
 - ExchangeWrapper (class in exchange_wrapper.service), 33
 - exec_amount (exchange_wrapper.models.Trade attribute), 29
 - exec_price (exchange_wrapper.models.Trade attribute), 29
 - execute_via_rpc() (in module exchange_wrapper.client), 20
 - ext_id (exchange_wrapper.models.Trade attribute), 29
- ## F
- fee (exchange_wrapper.models.Trade attribute), 29
 - filled_size (exchange_wrapper.models.Order attribute), 27
 - fnmatch_dicts() (in module exchange_wrapper.util), 20
- ## G
- get() (exchange_wrapper.config.Config method), 21
 - get_all_wallet_sets() (exchange_wrapper.service.ExchangeWrapper method), 34
 - get_allocated() (exchange_wrapper.logical_wallet.LogicalWalletManager method), 26
 - get_allocated_wallet_set() (exchange_wrapper.service.ExchangeWrapper method), 35
 - get_allocated_wallet_sets() (in module exchange_wrapper.http_server), 22
 - get_book() (exchange_wrapper.service.ExchangeWrapper method), 35
 - get_book() (in module exchange_wrapper.http_server), 22
 - get_connection() (exchange_wrapper.rpc.RpcInterface method), 32
 - get_exchange_pricing() (in module exchange_wrapper.http_server), 22
 - get_exchange_total() (exchange_wrapper.logical_wallet.LogicalWalletManager method), 26
 - get_exchange_unallocated() (in module exchange_wrapper.http_server), 23
 - get_exchanges() (exchange_wrapper.service.ExchangeWrapper method), 35
 - get_held_for() (exchange_wrapper.models.VirtualWallet method), 30
 - get_ohlc() (in module exchange_wrapper.http_server), 23
 - get_pricing() (exchange_wrapper.service.ExchangeWrapper method), 36
 - get_quote_amounts() (exchange_wrapper.models.Order method), 27
 - get_status() (exchange_wrapper.service.ExchangeWrapper method), 36
 - get_total_wallet_set() (exchange_wrapper.service.ExchangeWrapper method), 36
 - get_total_wallet_sets() (in module exchange_wrapper.http_server), 24

get_unallocated() (exchange_wrapper.logical_wallet.LogicalWalletManager method), 26
 get_unallocated_exchange() (in module exchange_wrapper.http_server), 24
 get_unallocated_per_exchange() (exchange_wrapper.logical_wallet.LogicalWalletManager method), 26
 get_unallocated_per_exchange() (exchange_wrapper.service.ExchangeWrapper method), 36
 get_unallocated_wallet_set() (exchange_wrapper.service.ExchangeWrapper method), 37
 get_unallocated_wallet_sets() (in module exchange_wrapper.http_server), 24
 get_wallet() (exchange_wrapper.models.WalletSet method), 30
 get_wallet_set() (exchange_wrapper.logical_wallet.LogicalWalletManager method), 26
 get_wallet_set() (exchange_wrapper.service.ExchangeWrapper method), 37
 get_wallet_set() (in module exchange_wrapper.http_server), 24
 get_wallet_set_history() (in module exchange_wrapper.http_server), 24
 get_wallet_sets() (exchange_wrapper.logical_wallet.LogicalWalletManager method), 26
 get_wallet_sets() (in module exchange_wrapper.http_server), 25
 get_wallets() (exchange_wrapper.client.Client method), 20
 get_wallets() (exchange_wrapper.models.Order method), 27
 getenv() (exchange_wrapper.config.Config method), 21

H

handle_message() (exchange_wrapper.rpc.RpcInterface method), 32
 handle_request() (exchange_wrapper.service.ExchangeWrapper method), 37
 handler() (exchange_wrapper.wsock_server.WebsocketServer method), 39
 hold (exchange_wrapper.models.VirtualWallet attribute), 30
 hold (exchange_wrapper.wallet.Wallet attribute), 38
 HttpServer (class in exchange_wrapper.http_server), 22

I

id (exchange_wrapper.db.BaseModel attribute), 22
 id (exchange_wrapper.models.Order attribute), 27
 id (exchange_wrapper.models.Trade attribute), 29
 id (exchange_wrapper.models.VirtualWallet attribute), 30
 id (exchange_wrapper.models.WalletLine attribute), 30
 id (exchange_wrapper.models.WalletSet attribute), 31

init() (exchange_wrapper.service.ExchangeWrapper method), 37
 jsonrpc_error_code (exchange_wrapper.rpc.ApiUsageError attribute), 32
 jsonrpc_error_code (exchange_wrapper.rpc.MethodNotFoundError attribute), 32
 jsonrpc_error_code (exchange_wrapper.rpc.ProcessingError attribute), 32

L

last_updated_at (exchange_wrapper.models.Order attribute), 27
 limit (exchange_wrapper.models.Order attribute), 27
 limit_buy() (exchange_wrapper.client.Client method), 20
 limit_sell() (exchange_wrapper.client.Client method), 20
 list_exchanges() (in module exchange_wrapper.http_server), 25
 list_open_orders() (exchange_wrapper.service.ExchangeWrapper method), 37
 list_open_orders() (in module exchange_wrapper.http_server), 25
 list_recent_orders() (exchange_wrapper.service.ExchangeWrapper method), 37
 list_recent_orders() (in module exchange_wrapper.http_server), 26
 load() (exchange_wrapper.config.Config method), 21
 log_404() (in module exchange_wrapper.http_server), 26
 log_wallet_set() (exchange_wrapper.service.ExchangeWrapper method), 38
 logger (exchange_wrapper.models.Order attribute), 27
 LogicalWalletManager (class in exchange_wrapper.logical_wallet), 26

M

main() (in module exchange_wrapper.http_server), 26
 make_logical_transaction() (in module exchange_wrapper.http_server), 26
 make_order() (exchange_wrapper.service.ExchangeWrapper method), 38
 make_server() (exchange_wrapper.wsock_server.WebsocketServer method), 39
 make_transaction() (exchange_wrapper.models.VirtualWallet method), 30
 make_transaction() (in module exchange_wrapper.http_server), 26

- maker (exchange_wrapper.models.Trade attribute), 29
 - match_limit_order() (in module exchange_wrapper.matching), 27
 - match_market_order() (in module exchange_wrapper.matching), 27
 - MethodNotFoundError, 32
 - metrics() (in module exchange_wrapper.http_server), 26
 - movement (exchange_wrapper.models.WalletLine attribute), 30
- ## N
- name() (exchange_wrapper.pair.Pair method), 31
 - notify() (in module exchange_wrapper.http_server), 26
 - notify_connect() (exchange_wrapper.http_server.HttpServer method), 22
 - notify_wallet_set_update() (exchange_wrapper.service.ExchangeWrapper method), 38
- ## O
- on_connect() (in module exchange_wrapper.http_server), 26
 - on_connect_hook() (exchange_wrapper.rpc.RpcInterface method), 32
 - on_disconnect_hook() (exchange_wrapper.rpc.RpcInterface method), 32
 - on_hold (exchange_wrapper.models.WalletLine attribute), 30
 - Order (class in exchange_wrapper.models), 27
 - order (exchange_wrapper.models.Trade attribute), 29
 - order (exchange_wrapper.models.WalletLine attribute), 30
 - order_id (exchange_wrapper.models.Trade attribute), 29
 - order_id (exchange_wrapper.models.WalletLine attribute), 30
 - order_type (exchange_wrapper.models.Trade attribute), 29
 - orders (exchange_wrapper.models.Order attribute), 27
 - orders (exchange_wrapper.models.WalletSet attribute), 31
- ## P
- Pair (class in exchange_wrapper.pair), 31
 - pair (exchange_wrapper.models.Order attribute), 27
 - parse_recv() (exchange_wrapper.client.Client method), 20
 - ProcessingError, 32
- ## R
- receive_handler() (exchange_wrapper.wsock_server.WebsocketServer method), 39
 - recursive_update() (in module exchange_wrapper.config), 21
 - register_exchange() (exchange_wrapper.service.ExchangeWrapper method), 38
 - remaining_size (exchange_wrapper.models.Order attribute), 27
 - request_token() (exchange_wrapper.util.token_bucket.TokenBucket method), 20
 - RpcInterface (class in exchange_wrapper.rpc), 32
 - run() (exchange_wrapper.connection.Connection method), 21
 - run_forever() (exchange_wrapper.rpc.RpcInterface method), 32
 - run_id (exchange_wrapper.models.WalletSet attribute), 31
 - run_server() (exchange_wrapper.wsock_server.WebsocketServer method), 39
- ## S
- send_handler() (exchange_wrapper.wsock_server.WebsocketServer method), 39
 - send_request() (exchange_wrapper.connection.Connection method), 21
 - send_request_async() (exchange_wrapper.connection.Connection method), 21
 - send_to() (exchange_wrapper.rpc.RpcInterface method), 32
 - send_to() (exchange_wrapper.wsock_server.WebsocketServer method), 39
 - send_to_all() (exchange_wrapper.rpc.RpcInterface method), 32
 - send_to_all() (exchange_wrapper.wsock_server.WebsocketServer method), 39
 - settled (exchange_wrapper.models.Order attribute), 27
 - shutdown() (exchange_wrapper.rpc.RpcInterface method), 32
 - shutdown() (exchange_wrapper.service.ExchangeWrapper method), 38
 - shutdown() (exchange_wrapper.wsock_server.WebsocketServer method), 39
 - side (exchange_wrapper.models.Order attribute), 27
 - size (exchange_wrapper.models.Order attribute), 28
 - start() (exchange_wrapper.http_server.HttpServer method), 22
 - status (exchange_wrapper.models.Order attribute), 28
 - stop() (exchange_wrapper.connection.Connection method), 21
 - strategy_run_id (exchange_wrapper.models.Order attribute), 28
 - subscribe_notifications() (exchange_wrapper.service.ExchangeWrapper method), 38

T

tag_type_exception() (in module exchange_wrapper.service), 38
 to_dict() (exchange_wrapper.models.Order method), 28
 to_dict() (exchange_wrapper.models.Trade method), 29
 TokenBucket (class in exchange_wrapper.util.token_bucket), 20
 Trade (class in exchange_wrapper.models), 28
 type (exchange_wrapper.models.Order attribute), 28
 type (exchange_wrapper.models.WalletSet attribute), 31

U

unfilled_size (exchange_wrapper.models.Order attribute), 28
 unsubscribe_notifications() (exchange_wrapper.service.ExchangeWrapper method), 38
 update_wallets() (exchange_wrapper.models.Order method), 28
 uri_map() (in module exchange_wrapper.http_server), 26

V

VirtualWallet (class in exchange_wrapper.models), 29

W

Wallet (class in exchange_wrapper.wallet), 38
 wallet (exchange_wrapper.models.WalletLine attribute), 30
 wallet_id (exchange_wrapper.models.WalletLine attribute), 30
 wallet_lines (exchange_wrapper.models.Order attribute), 28
 wallet_lines (exchange_wrapper.models.VirtualWallet attribute), 30
 wallet_set (exchange_wrapper.models.Order attribute), 28
 wallet_set (exchange_wrapper.models.VirtualWallet attribute), 30
 wallet_set_id (exchange_wrapper.models.Order attribute), 28
 wallet_set_id (exchange_wrapper.models.VirtualWallet attribute), 30
 WalletLine (class in exchange_wrapper.models), 30
 wallets (exchange_wrapper.models.WalletSet attribute), 31
 WalletSet (class in exchange_wrapper.models), 30
 websocket_query() (in module exchange_wrapper.client), 20
 websocket_query_timeout() (in module exchange_wrapper.client), 21
 WebSocketServer (class in exchange_wrapper.wsock_server), 39