

Crypto Advisor: A Web Application for Spotting Cross-Exchange Cryptocurrency Arbitrage Opportunities

Robert-Christian Oanță^a and Adriana Mihaela Coroiu^b

Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Mihail Kogălniceanu, Cluj-Napoca, Romania

Keywords: Cryptocurrency, Cross-Exchange Arbitrage, Centralized Exchanges, Trading Automation, Arbitrage Opportunities Spotting, Web Application, Cloud Computing.

Abstract: The subject of this paper revolves around cryptocurrencies and trading automation, more precisely, on a low-risk trading strategy called cross-exchange arbitrage. An approach that capitalizes on market inefficiencies by frequently buying and selling on two distinct exchanges in order to accumulate minor profits. Our solution is a web application for identifying opportunities produced by this strategy and delivering them in a user-friendly manner, as well as in a structured format for developers. The main objective was to develop a production-ready tool that is useful for professional traders, utilizing real data in real-world circumstances.

1 INTRODUCTION

Cryptocurrencies are digital assets or currencies based on blockchain technology, which enables payment and transaction verification in the absence of a centralized custodian. Bitcoin, the most well-known cryptocurrency, was first described in a paper by Satoshi Nakamoto (Nakamoto, 2008) in 2008, and it was launched in 2009. The cryptocurrency market has changed significantly since then. On more than 100 exchanges around the world, more than 50 million active investors trade bitcoin and other cryptocurrencies. We have long been fascinated by trading automation, not only on the cryptocurrency market but also on the stock market. But as we all know, trading does not guarantee profits, particularly when trading without a strategy, therefore we tried to design a system that minimizes risk and focuses on nearly market-neutral strategies that can guarantee long-term earnings. This paper aims to provide comprehensive research on a specific low-risk cryptocurrency trading strategy called cross-exchange arbitrage.


2 THEORETICAL BACKGROUND


Blockchain Technology. Blockchain, as explained by Fang in (Fang et al., 2022), is a digital ledger of

economic transactions that can be used to record anything with inherent worth, not simply financial transactions. In its most basic form, a Blockchain is a collection of immutable data entries with timestamps that are maintained by a distributed network of machines. A transaction is, for example, a file that says, "A pays X Bitcoins to B" and is signed using A's private key. This is the most fundamental form of public-key cryptography, as well as the foundation for cryptocurrencies. On these networks, cryptocurrencies are the tokens used to send value and pay for transactions. They can be regarded as Blockchain tools, but in certain situations, they can also act as resources or utilities.

Cryptocurrency Exchanges. A cryptocurrency exchange or Digital Currency Exchange (DCE) is a business that allows customers to trade cryptocurrencies. The practical part of this thesis revolves a lot around exchanges and the variation in prices of assets on these exchanges, so it is important to understand what type of markets exist and why there might be a price discrepancy for similar assets. To expand on the above statement, there are two primary sorts of exchanges:

- **Centralized exchanges (CEX):** as the term implies, CEXs are crypto exchanges founded by centralized entities that control the exchange's ownership. The pricing of assets on centralized exchanges is determined by the most recently matched bid-ask order on the exchange order book.

^a  <https://orcid.org/0000-0002-5823-1827>

^b  <https://orcid.org/0000-0001-5275-3432>

- Decentralized exchanges (DEX): in contrast to CEXs, DEXs rely on arbitrage traders to maintain prices in check. Instead of an orderbook in which buyers and sellers are matched to trade cryptocurrency assets, these exchanges utilize liquidity pools.

To be noted, the practical application of this paper is related mostly to CEXs.

Arbitrage Trading Strategies. Defining cryptocurrency trading in a simple form - is the act of buying and selling cryptocurrencies with the purpose of profit. A trading strategy is an algorithm of predefined rules and events that trigger buying and selling crypto assets on one or multiple markets (as mentioned in (Jothi and Oswalt Manoj, 2022)). There are numerous trading strategies available, but the focus of this thesis is on arbitrage trading. Furthermore, arbitrage trading can be divided into multiple sub-strategies as described in (Shynkevich, 2021), such as cross-exchange arbitrage, spatial arbitrage, triangular arbitrage, decentralized arbitrage, and statistical arbitrage. As mentioned previously, our focus is on the cross-exchange arbitrage strategy.

3 CROSS-EXCHANGE ARBITRAGE METHOD

Cross-exchange arbitrage is a low-risk strategy that capitalizes on market inefficiencies by frequently buying and selling on two distinct exchanges in order to accumulate minor profits.

3.1 Why Is this Strategy Low-Risk?

As described in (Lacity, 2020), traders that recognize arbitrage opportunities and capitalize on them do so with the expectation of making a fixed profit, rather than assessing market sentiments or relying on other predictive pricing strategies. Also, depending on the resources available to traders, it is possible to enter and exit an arbitrage trade in seconds or minutes. Although this sounds good in theory, numerous factors must be considered, such as fees, timing, availability, and profitability.

On the majority of exchanges, we can distinguish the following sorts of usual fees:

- Withdraw fees: These are the costs incurred when withdrawing a crypto asset from an exchange's digital wallet. Typically, these fees are flat (set amount) and determined by the coin you desire to withdraw and the blockchain on which the withdrawal activity is performed.

- Trading fees: These are the charges associated with purchasing and selling crypto on an exchange. They are often based on percentages, ranging from 0.05 percent to 0.5 percent, and are primarily divided into two categories: maker fees and taker fees, according to (Lansky, 2018).

3.2 Timing and Availability

Time is of the essence in arbitrage trading. The pricing discrepancy between two exchanges tends to shrink as more traders take advantage of a given arbitrage opportunity, this is due to the law of supply and demand. Some of the factors that could affect the time it takes to execute a successful arbitrage trade are listed below (as presented in (Fang et al., 2022)):

- Transaction speed: Because you must perform cross-exchange transactions, the time it takes to validate these transitions on the blockchain may have an impact on the effectiveness of your arbitrage trading strategy.
- AML checks: Although arbitrage trading is not illegal in any kind of way, when substantial sums of money are moved by a trader, exchanges frequently conduct anti-money laundering (AML) checks. Such examinations might take weeks in some circumstances.
- Offline wallets: Crypto exchanges are prone to outages. For one reason or another, crypto exchanges may restrict the withdrawal and deposit of specific digital assets.

3.3 Algorithm Steps

This section presents the steps of executing a cross-exchange arbitrage trade. An arbitrage opportunity is spotted between exchange A and exchange B for the coin BTC, pair BTC/USDT. First of all, assuming we have a certain amount of USDT in our account on exchange A.

1. Trade BTC/USDT pair on exchange A, effectively buying on the ask price for how much USDT amount we have. Therefore, we have successfully converted our amount of USDT into BTC. By doing this operation, we have incurred a trading fee.
2. Withdraw the amount of BTC we have to exchange B, making sure that the withdraw chain from exchange A is the same with deposit chain from exchange B. By doing this operation, we have incurred a withdraw fee.
3. Once the BTC is deposited on exchange B, we trade USDT/BTC, effectively selling on the bid

price for how much BTC amount we have. Therefore, we have successfully converted our amount of BTC into USDT. By doing this operation, we have incurred another trading fee.

4. Now we are left with more USDT than we first started, effectively making a profit. This has been a successful arbitrage cycle.
5. Optionally, we can repeat the cycle by withdrawing our USDT back to exchange A. Note that this operation is taxed by another withdraw fee.

E. Profitability The formula based on quantity, rate, trade, and fee can be used to estimate the profitability of an arbitrage opportunity (the result is percentage-based on the initial quantity).

4 TECHNOLOGIES

In this section, we analyze the advantages and limitations of our application’s underlying technologies. We will cover web application architecture, client-side and server-side frameworks, and cloud deployment.

Client-Server Architecture. A client-server architecture divides an application into client and server components. This application type is executed across a computer network connecting the client and the server. The server provides the essential functionality of such a design: any number of clients can connect to the server and request that it accomplish a task. The server accepts these requests, completes the requested task, and returns any necessary results to the client, according to (Fraternali, 1999).

Single-Page Applications. There are numerous sorts of web applications, including static web applications, dynamic web applications, e-commerce applications, portal web applications, progressive web applications, etc. Our application is classified as a Single-Page Application (SPA). Following (Hacker and Hatemi-J, 2012), single-page web apps allow for dynamic interaction by modifying the current page’s content rather than loading entirely new pages from the server whenever a user action is done. AJAX, a condensed form of Asynchronous JavaScript and XML, is the foundation for enabling page communications and, thus, for making SPAs a reality. Single-page applications are similar to traditional desktop applications in that they do not disrupt the user experience.

FastAPI. FastAPI is a modern, fast (high-performance), server-side web framework for building APIs with Python 3.6+ based on standard Python type hints. Despite Python’s reputation as

being a slower programming language because it is interpreted rather than compiled, FastAPI provides the same level of performance as typically quicker languages. As depicted in Figure 1, we can see a performance benchmark in comparison with other frameworks written in Go, JavaScript and Python.

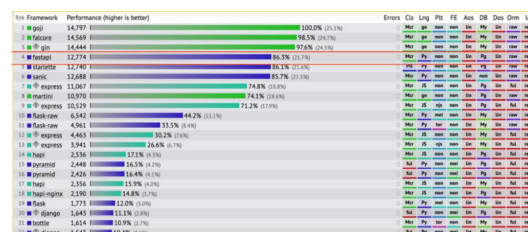


Figure 1: Performance Benchmark.

It may not be clear how FastAPI is able to achieve this staggering performance only by looking under the hood we can see that this framework makes use of the async package in Python. FastAPI is capable of integrating so well with Asyncio because it is built atop Starlette, a lightweight ASGI framework/tool that is perfect for developing async web services. As a spiritual successor to WSGI, ASGI is a standard interface. It allows for interoperability throughout the whole Python async web stack, including servers, applications, middleware, and individual components. Concurrency is applied at the request/response level; thus your application will act in a non-blocking manner throughout the stack. As a result, the performance gains are significant. There are multiple other capabilities in FastAPI’s toolbox, including the use of pydantic schemas, powerful dependency injection, and Jinja2 integration, as detailed in (Kornienko et al., 2021).

Vue. Vue is a client-side JavaScript framework for building user interfaces and single-page applications (SPAs). It is built on top of conventional HTML, CSS, and JavaScript, and it is based on a declarative, component-based programming model that helps you create user interfaces quickly, no matter how basic or complex they are. It mainly follows the architectural pattern of model-view-viewmodel. Inspired by Vue’s team comparison (Hanchett and Listwon, 2018), in the following paragraphs we will compare it with two other popular client-side frameworks React and Angular. React and Vue share some similarities such as utilizing a virtual DOM and providing reactive and composable view components. All React components use JSX, a declarative XML-like syntax that operates withing JavaScript. Using JSX to render functions has the benefit of harnessing the capability of a full programming language (JavaScript) to build your view. Despite supporting JSX, Vue’s default experi-

ence is templates, a simpler alternative. The benefit of templates is that they are more intuitive to understand for developers that have worked with HTML, and that existing applications can be progressively migrated to make use of Vue's reactivity features. Some of Vue's syntax will resemble that of AngularJS (e.g. v-if vs ng-if). This is due to the fact that AngularJS got a number of things right, and these were an early source of inspiration for Vue. Both in terms of API and design, Vue is considerably easier to use than AngularJS. In most cases, learning enough to construct non-trivial applications takes less than a day, but this is not the case with AngularJS. Moreover, Vue is a more flexible, modular solution than AngularJS, which has firm convictions about how your applications should be constructed. While this makes Vue more adaptable to a wider range of projects, we also acknowledge that there are instances when it's beneficial to have some decisions made for you so you can get straight to working.

4.1 Cloud Deployment

Cloud Deployment is the process of deploying an application using one or more cloud-based hosting models, such as software as a service (SaaS), platform as a service (PaaS), and/or infrastructure as a service (IaaS). This covers designing, planning, executing, and running cloud workloads.

4.1.1 Heroku Server-Side Deployment

Heroku is a platform as a service (PaaS) that enables developers to build, run, and operate applications entirely in the cloud. PaaS architectures typically include operating systems, programming-language execution environments, libraries, databases, web servers, and connectivity to some platforms. The Heroku cloud service platform is based on a managed container (called dynos within the Heroku paradigm) system. It has integrated data services and a powerful ecosystem for deploying and running modern applications. Heroku is, in our opinion, a superior alternative for developers new to cloud deployment than its IaaS competitors, Azure or AWS, due to the fact that in a PaaS service, you do not have to worry about managing containers, load balancers, or databases; all are given as a managed service. Its pricing (free 500 hours of dyno usage each month), python build-packs, and various add-ons, like free databases, SSL certificates, custom domain names, smtp mail server, among others, are a few of its attractions.

4.1.2 Netlify Client-Side Deployment

Similarly to Heroku, Netlify is also a PaaS, the main distinction is that Netlify mainly focuses on hosting static websites with serverless backends, whereas Heroku is built to host dynamic, server-side rendered websites and apps. The way Netlify works is quite innovative as it aims to decouple the client-side from the server-side part of an app. By the use of the Jamstack, Netlify delivers a precompiled and optimized static frontend which is deployed globally via the Edge CDN. The server-side consists of multiple serverless functions distributed as microservices, hence lowering load. In addition, these functions are deployed to AWS Lambda and turned into API endpoints. However, in our particular case, a serverless backend was unnecessary, as all of the server-side code was deployed to the aforementioned Heroku. Some of its perks include making the deployment process fast and intuitive, pricing (300 minutes of free build time per month), and add-ons such as CD integration with GitHub, free TLS encryption, a custom domain name, and SPA integration.

5 SIMILAR SOLUTIONS

As the sector for arbitrage trading automation is not highly developed, there are not as many similar popular solutions in this field. Although arbitrage trading is quite popular, it is often conducted by individuals using proprietary software or open-source bots, which require specialized programming and cryptocurrency skills to operate. To clarify, crypto trading automation is developed, however, crypto arbitrage trading automation is not. There are numerous powerful bots that trade cryptocurrencies on your behalf using typically straightforward tactics. Some of these general bots are included on popular platforms such as Pionex, Kucoin, and Coinrule, among others. Except for Pionex, which offers some sort of arbitrage bot, the rest of the mentioned platforms do not really offer any solutions. Moreover, Pionex's trading bot utilizes spot-futures arbitrage, which is an entirely different story from cross-exchange arbitrage. Although, in the course of our extensive research, we have uncovered a number of relatively obscure and somewhat popular platforms that are comparable to our concept. Table 1 presents an unbiased comparison between these alternatives. Note that Crypto Advisor is our solution.

Legend for the table content: a - Automated System: the ability to automatically trade cryptocurrency on your behalf. b - Spotter: continuous finder of arbitrage opportunities without user input. c - Spotting

Table 1: Solution APPs comparison.

Characteristics	Crypto Advisor	ArbiTool	ArbiSmart	Cryptohopper	Coygo	Hummingbot
Web Platform	Yes	Yes	Yes	Yes	Yes	No
Main Function	Spotter	Spotter	Automated Platform	Automated Platform	Spotter and Automated Platform	Terminal-like Bot and ecosystem
Automated System	No	No	Yes	Yes	Yes	Yes
Programmable Bot	No	No	No	No	No	Yes
Spotterb	Yes	Yes	Yes	Yes	Yes	Yes, but limited
Spotting tablec	Yes	Yes	No	No	Yes	No
History tabled	Yes	No	No	No	No	No
Wallet Statuse	No	Yes	N/A	N/A	? o	Yes
Tx Timef	No	Yes	N/A	N/A	?	No
Complex Filtersg	Yes	Yes	N/A	N/A	Yes	N/A
Email Alerts	Yes	Yes for Premium	N/A	N/A	?	N/A
Market Depth	Yes	Yes	No	No	Yes	Yes
Orderbook Table	No	Yes	No	No	?	?
Public API	Yes	No	No	Yes	Swapped for trading terminal	N/A
Opportunities Endpointh	Yes	N/A	N/A	Something similar	N/A	N/A
History Endpointi	Yes	N/A	N/A	Yes	N/A	N/A
Opportunities WebSocketj	Yes	No	No	No	N/A	N/A
Subscription	Free	Free and premium	Fee-based	Free and premium	Free and premium	Open-source and Free
Nr of supported exchanges	10	35	20	9	11	33
DEX supportk	No	No	No	No	No	Yes
Nr of supported currencies	1000	?.probably >1000	?	75	?	?.probably >1000

table: web or in-app/terminal presentation of current arbitrage opportunities. d - History table: web or in-app/terminal presentation of past arbitrage opportunities. e - Wallet Status: provides the wallet status (offline, online) for multiple currencies and exchanges. f - Tx time: estimated transaction time between two exchanges. g - Complex filters: complex filtering capabilities for the spotting table. h - Opportunities Endpoint: API endpoint for receiving JSON information about current arbitrage opportunities. i - History Endpoint: API endpoint for past arbitrage opportunities or useful backtesting info. j - Opportunities WebSocket: 24/7 streaming WebSocket for current arbitrage opportunities. k - DEX support: supports the usage of decentralized exchanges. l - Exchange Connectors: how does the platform connect to various exchanges (automatic, no need for connection, private API keys). m - Spotting/Trading latency: the delay in spotting/trading an arbitrage opportunity. n - N/A, not applicable. o - ?, unknown, no data.

an extension of the API by continuously delivering arbitrage opportunities to connected clients. In order to present the API more effectively, we have also designed a web application that is built upon it. The client-side consumes the data supplied by our API and displays it to the client in an easy-to-use format. Additionally, we have also implemented user authentication, email notifications, and OpenAPI standards documentation.

6.2 Pre-Implementation Analysis

After analyzing the application’s requirements, we have determined that its development lifecycle consists of four major stages: Arbitrage algorithm implementation and data collection, API Implementation, Front-End/UI Implementation, and Cloud Deployment. The development was conducted in the presented order. To further highlight the features and constraints of our application, we have prepared the following use cases diagram, depicted in Figure 2.

6 PRACTICAL APPLICATION

6.1 Project Description

As presented in the previous section, our solution is Crypto Advisor. A web application for spotting cross-exchange cryptocurrency arbitrage opportunities. This application’s primary objective was to provide a robust public API solution that works as a data collector, locating the values of cryptocurrencies on many exchanges and spotting arbitrage opportunities between them. Moreover, a public streaming WebSocket was developed, which essentially functions as

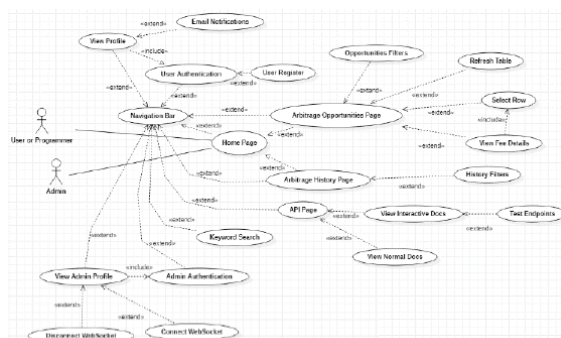


Figure 2: Use cases diagram.

6.3 Domain Models

We have also constructed a diagram depicting the domain models of our application. SQLAlchemy ORM creates the tables in our database based on these models (as we can see in Figure 3).

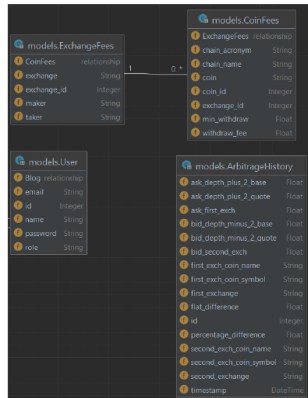


Figure 3: Domain Models Diagram.

6.4 Architecture

6.4.1 Client-Side Architecture

As Vue.js is a client-side component-based framework employing the model-view-viewmodel pattern, reusability is strongly encouraged, which is why components are embedded in other components or views in the diagram below: Figure 4.

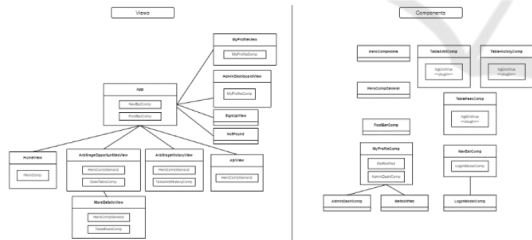


Figure 4: Client-Side Architecture Diagram.

Views have the same capabilities as components, but they are mostly utilized for routing; hence, you could say that a distinct page is a view.

6.4.2 Server-Side Architecture

To illustrate how the layers communicate with each other, we have constructed a diagram of the entire server-side application: Figure 5.

Note that we omitted fields and methods because it would make the diagram much more difficult to comprehend. Usually, python applications are difficult to portray in a UML class diagram since python is not as restrictive in OOP and classes are not mandatory.

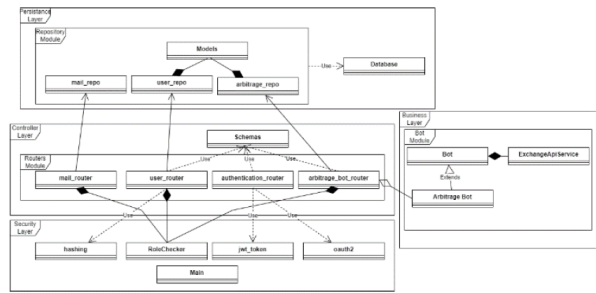


Figure 5: Server-side Architecture Diagram.

Therefore, some of the so-called classes in figure 6 are essentially simple files with static attributes and methods that behave similarly to a singleton class

6.4.3 Deployment Architecture

In order to better understand the security procedures and layers of our application, we have created a deployment diagram (we can see in Figure 6) depicting how our services are distributed over different servers.

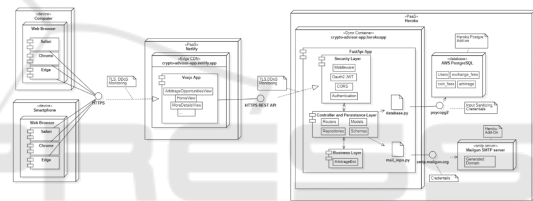


Figure 6: Deployment Architecture Diagram.

Essentially, any device with a browser can access our system via the URL: <https://crypto-advisor-app.netlify.app/home>

The front-end service then interacts with the Heroku-deployed REST API. In addition, we have utilized Heroku's add-ons such as an AWS PostgreSQL database and Mailgun SMTP server.

6.5 Business Layer

The Business layer of this application encapsulates the module Bot-Folder where all the logic behind arbitrage spotting is implemented. As shown in Figure 5, the module has high cohesiveness between its components and is only loosely coupled to a Rest Controller responsible for arbitrage endpoints. This module can be easily isolated from the rest of the application because it operates independently. The abstract class Bot provides a generalization of the Arbitrage-Bot class. This class mostly adheres to a Strategy-like Pattern and has concrete methods for data collecting and parsing that can be used in any kind of Bot (Do This,).

6.5.1 Data Collecting and Parsing

The primary purpose of the `ExchangeApiService` class is to connect to external APIs and scrape websites. Multiple external services, such as the CoinMarketCap API, the CCXT library (onl,), and the CoinMarketFees website (Atzberger et al., 2022), are present in the class and are actively utilized. The CCXT library is responsible for collecting data such as coin prices and order book details, users of this library will have direct access to the source (the API endpoints of various exchanges), resulting in the most accurate data available. The CoinMarketCap API is mostly used for retrieving the topmost 1000 popular currencies because these coins often have the highest trading activity and are more likely to provide arbitrage opportunities.

As the ccxt library is not very reliable on fees, we have used beautiful soup (a python library for web scraping (Richardson, 2007)) to collect as much data as possible from the CoinMarketFees website, where fees are typically accurate.

6.5.2 Data Manipulation and Arbitrage Spotting

`ArbitrageBot` is an implementation of the generic `Bot` class; its primary function is to manipulate received data and identify arbitrage opportunities. The `gather-data` method relies on the superclass's methods in order to retrieve data from the `ExchangeApiService`. After that, a comparison is made between all exchanges, and then all pairs of the used exchanges, under a predetermined threshold of percentage difference, and a JSON-style dictionary is constructed containing all opportunities. Information about market liquidity and fees is obtained through other methods.

6.6 Controller Layer

This layer's purpose is to expose the Rest endpoints. Encapsulating the routers module, which contains the API's controllers for its many functionalities (user router, authentication router, mail router and arbitrage router). This module primarily defines the URLs, parameters, dependencies, and response schemas of our endpoints. In addition, by utilizing dependency injection, unauthorized access to the resources is prevented. The database connection is also provided using injections by referencing a function from our connection file which yields a `Session` object that can be later used for queries (Copeland, 2008). In order to maintain a clean separation between our layers, endpoint functionality is delegated to the repository module of the persistent layer.

Endpoints. One of the top objectives in creating our endpoints was to ensure full synchronicity between them so that time-intensive endpoints, such as the endpoint for identifying opportunities, would not cause the application to underperform. Thus, endpoints will function optimally when several simultaneous calls are made without any time delays. The arbitrage router contains endpoints primarily associated with our business layer, such as endpoints for identifying opportunities, storing scrapped fees in our database, retrieving past arbitrage opportunities from our database, obtaining trading fees for specific coins and exchanges, a `WebSocket` for continuous streaming our opportunities, and endpoints for internal connection and disconnection from the `WebSocket` (Pterneas, 2013). The authentication router is mostly associated with the security layer, where all functionality is outsourced, and its primary job is to expose a login endpoint. Similarly, to the authentication router, the Mail router has the primary purpose of exposing an endpoint related to the email notifications for our users. Lastly, the User router is related to CRUD operations on our users, namely creating (registering), reading, updating, and deleting user information.

WebSocket. Due to challenges in gathering data in the client-side portion of this application, as time-intensive endpoints would be terminated on refreshing, losing crucial information, our solution is a `WebSocket` that continuously broadcasts the most recently computed list of opportunities. Thus, when required, the client-side can obtain all data almost instantly. Note that this `WebSocket`, as well as a few arbitrage endpoints, are publicly accessible via our API solution. Several setbacks have occurred throughout the development of this websocket. Typically, websockets should be a separate service that operates independently from the rest of the application. However, this is not possible in our case, thus our websocket is essentially incorporated as an endpoint, which has led to numerous difficulties in maintaining synchronicity. For the rest of the endpoints, synchronicity is assured by design, as the `FastApi` framework inserts the endpoint functions in a self-managing thread or event pool, but this is not the case with websockets. We had to manually obtain the running event loop and inject the websocket's functionality, which proved to be quite challenging, but we were ultimately successful. However, additional optimizations are required as the number of external clients connecting to our websocket affects its performance.

6.7 Persistence Layer

The persistence layer consists primarily of the ORM models (Figure 4) that are used to generate database tables, as well as the repository module. As indicated before, the repository module is responsible for accessing the database tables utilizing the injected Session object and defining the endpoint functionality. Essentially, each Rest Controller has its own associated repository. Typically, functions are divided into computational functions and database manipulation functions. In addition, common HTTP exceptions such as 404 not found error, 500 internal server error and 409 conflict error are raised here (Zhu et al., 2015).

6.8 Security Layer

As this application is distributed on the web, we have invested a great deal of work ensuring that it has a solid security foundation. This module consists of functionalities for user authentication through RBAC (Sandhu, 1998) and oauth2's JWT (Bucher and Christensen, 2019; MAN,), as well as middleware (Middleware, 2006) and CORS (Abdelhamied, 2016). Firstly, in order to assure that our user's credentials are safe we have used OAuth2PasswordRequestForm and OAuth2PasswordBearer from FastAPI's library (Bansal and Ouda, 2022), which is mainly represented through a form-data in the endpoint request body. Furthermore, user's credentials are stored in the database in an encrypted format using the bcrypt hashing algorithm which ensures that brute force and rainbow table attacks are not feasible (Sriramya and Karthika, 2015). Our login endpoint from the authentication router verifies through bcrypt that the supplied credentials match those stored in the database. After that, it returns the JWT token with the expiration date, username, email and the role of our user embedded in itself. This token can be later decrypted using the Secret Key. All of our critical endpoints necessitate an Authorization header with a valid unexpired token in order to function. Furthermore, role-based access to resources is ensured by checking the provided token and matching it to the current user token. Even though a common user has a valid token, RBAC prevents him from accessing admin-specific endpoints. Although middleware is generally a separate server that operates as a proxy, we only have some sort of middleware that intercepts API calls and runs a specific validation code segment. Our middleware function validates client IP addresses based on a whitelist of allowed addresses, returning a 400 Bad Request Error otherwise. As our client-

side and server-side are placed on separate servers, we have assured via CORS that only calls from our specific client-side resource, namely crypto-advisor-app.netlify.app, are permitted. This is achieved by specifying an origin header, which CORS can afterward validate. The psycopg2 adapter for database connection is provided by the SQLAlchemy package, and besides transporting credentials or other sensitive information such as host or port number, it ensures that the queries are properly sanitized and do not contain any kind of hidden compromising commands that might affect our database (Nüst and Ostermann, 2020). Both client-side and server-side resources are accessible through https with TLS encryption [31], with certificates issued by the PaaS provider. The providers also offer active DDoS [32] monitoring, which is another popular form of cyberattack.

7 CONCLUSION AND FUTURE DEVELOPMENTS

The objective of this thesis was to assess the efficacy of the cross-exchange arbitrage strategy and build a web application with an underlying API that facilitates this approach. We can proudly say this was accomplished. We have developed a robust API that delivers precise data with low latency. Our API solution is highly specialized in cross-exchange arbitrage for centralized exchanges, making it rather distinctive in terms of its capabilities. Obviously, there are other APIs that provide similar data, but we were unable to discover one that incorporates every aspect required to perform an arbitrage trade, as our API does. We hope to eventually expand our solution in the following ways: Firstly, we would like to incorporate additional exchanges, especially decentralized ones, and also more coins. We would also like to make our system more efficient by utilizing graph networks and cost-based pathfinding algorithms, such as Dijkstra's algorithm. As stated in the first section, we have only considered one of the several arbitrage approaches; therefore, we would like to develop solutions for additional strategies in the future.

ACKNOWLEDGEMENTS

The publication of this paper was supported by the 2022 Development Fund of the Babeş-Bolyai University (UBB).

The extended version of this paper and large explanations were part of the bachelor thesis of Robert

Oanta, study program Mathematics and Computer Science, UBB.

REFERENCES

- Abdelhamied, M. A. H. (2016). Client-side security using cors.
- Atzberger, D., Scordialo, N., Cech, T., Scheibel, W., Trapp, M., and Döllner, J. (2022). Codecv: Mining expertise of github users from coding activities. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 143–147. IEEE.
- Bansal, P. and Ouda, A. (2022). Study on integration of fastapi and machine learning for continuous authentication of behavioral biometrics. In *2022 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–6. IEEE.
- Bucher, P. P. and Christensen, C. J. (2019). Oauth 2.
- Copeland, R. (2008). *Essential sqlalchemy*. ” O’Reilly Media, Inc.”.
- Do This, W. W. Refactoring, design patterns and extreme programming.
- Fang, F., Ventre, C., Basios, M., Kanthan, L., Martinez-Rego, D., Wu, F., and Li, L. (2022). Cryptocurrency trading: a comprehensive survey. *Financial Innovation*, 8(1):1–59.
- Fraternali, P. (1999). Tools and approaches for developing data-intensive web applications: a survey. *ACM Computing Surveys (CSUR)*, 31(3):227–263.
- Hacker, S. and Hatemi-J, A. (2012). A bootstrap test for causality with endogenous lag length choice: theory and application in finance. *Journal of Economic Studies*, 39(2):144–160.
- Hanchett, E. and Listwon, B. (2018). *Vue.js in Action*. Simon and Schuster.
- Jothi, K. and Oswalt Manoj, S. (2022). A comprehensive survey on blockchain and cryptocurrency technologies: Approaches, challenges, and opportunities. *Blockchain, Artificial Intelligence, and the Internet of Things: Possibilities and Opportunities*, pages 1–22.
- Kornienko, D., Mishina, S., and Melnikov, M. (2021). The single page application architecture when developing secure web services. In *Journal of Physics: Conference Series*, volume 2091, page 012065. IOP Publishing.
- Lacity, M. (2020). Crypto and blockchain fundamentals. *Ark. L. Rev.*, 73:363.
- Lansky, J. (2018). Possible state approaches to cryptocurrencies. *Journal of Systems integration*, 9(1):19.
- MAN, B. J. User interface for ddos mitigation configuration.
- Middleware, R. H. (2006). Hibernate: Relational persistence for java and .net. *Red Hat Middleware*.
- Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, page 21260.
- Nüst, D. and Ostermann, F. (2020). Reproducibility review of: Window operators for processing spatio-temporal data streams on unmanned vehicles. *Open Science Framework*, 10.
- Pterneas, V. (2013). *Getting Started with HTML5 Web-Socket Programming*. Packt Publishing.
- Richardson, L. (2007). Beautiful soup documentation.
- Sandhu, R. S. (1998). Role-based access control. In *Advances in computers*, volume 46, pages 237–286. Elsevier.
- Shynkevich, A. (2021). Bitcoin arbitrage. *Finance Research Letters*, 40:101698.
- Sriramya, P. and Karthika, R. (2015). Providing password security by salted password hashing using bcrypt algorithm. *ARPJ journal of engineering and applied sciences*, 10(13):5551–5556.
- Zhu, J., He, P., Fu, Q., Zhang, H., Lyu, M. R., and Zhang, D. (2015). Learning to log: Helping developers make informed logging decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 415–425. IEEE.