

Software Design Specification

# ODIN

A Mock Trading Platform with Internal OTC (Over-The-Counter) Market

# CONTENTS

1.	ACRONYMS AND GLOSSARY .....	3
2.	INTRODUCTION .....	4
	PURPOSE .....	4
	SCOPE .....	4
3.	SYSTEM OVERVIEW .....	5
	DESIGN CONSIDERATIONS .....	6
	Microservice Architecture .....	6
	Monolithic Architecture Pattern .....	7
4.	SYSTEM ARCHITECTURE .....	10
	OVERVIEW .....	10
	ARCHITECTURAL STRATEGY .....	11
	DEPENDENCIES AND CONSTRAINS .....	11
5.	SUBDOMAINS .....	12
	SECURITY MANAGEMENT .....	13
	Features .....	13
	User Interfaces .....	13
	ACCOUNT MANAGEMENT .....	14
	Features .....	14
	User Interfaces .....	15
	CATALOG MANAGEMENT .....	16

Features .....	16
ORDER MANAGEMENT (BROKERAGE) .....	17
Features .....	17
User Interface.....	17
INTERNAL OTC MARKET .....	19
Features .....	19
RESEARCH.....	20
Features .....	20
User Interface.....	20
6. FUTURE SYSTEM ARCHITECTURE .....	21
7. REFERENCES .....	22

## 1. ACRONYMS AND GLOSSARY

**2FA – 2 Factor Authentication:** It is a method of confirming users' claimed identities by using a combination of two different factors: i) something they know; ii) something they have; or iii) something they are.

**CRUD – Create, Read, Update, Delete:** In computer programming, it refers to the four basic functions of persistent storage.

**FUTF – Future Feature:** A feature that is not part of the scope of this project but may be implemented in the future.

**IPFS – InterPlanetary File System:** A protocol and network designed to create a content-addressable, peer-to-peer method of storing and sharing hypermedia in a distributed file system.

**KYC – Know Your Customer:** The process of a business verifying the identity of its clients and assessing the potential risks of illegal intentions for the business relationship.

**MVP – Minimum Viable Product:** A set of features sufficient to deploy the product.

**NYSE – New York Stock Exchange:** An American stock exchange.

**NASDAQ - National Association of Securities Dealers Automated Quotations:** An American stock exchange.

**OTC – Over-The-Counter:** A security traded in some context other than on a formal exchange such as NYSE or NASDAQ.

## 2. INTRODUCTION

### PURPOSE

This document is designed to be a reference to anyone interested in the overall architecture of the Odin application project. This document describes the high-level scope and features of Odin's modules, components, services, and databases. This design document includes, but is not limited to, the following information for the Odin application: system overview, design considerations, overall system architecture and application features.

The application features are classified in two groups: (i) Minimum Viable Product, which represents the features sufficient to deploy the product; and (ii) Future Features, which represents futures that may be added to the platform in the future.

The acronym "MVP" will be used to identify the Minimum Viable Product features, and the acronym "FUTF" will be used to identify Future Features.

### SCOPE

The objective of this project is to develop Odin, a "mock" trading platform with an internal OTC (Over-The-Counter) market. It must be developed in a four-week timeframe as a project for Byte Academy's Intensive Program in Python, Fintech, and Blockchain (New York / NY, 2018).

Odin is described here as a "mock" trading platform because it will not allow trading with real money, but only simulate the capabilities of a real trading platform by using market data available through external data providers. Deposits and withdrawals from the trading platform will also be simulated.

This document lists both MVP and Future Features, however, only MVP Features are part of the scope of this project.

### 3. SYSTEM OVERVIEW

Odin is a trading platform with an internal OTC exchange and is composed of six subdomains (Figure 1), which are listed and briefly described below. Each subdomain may be composed of one or more component(s). A more detailed description of each subdomain, its components, features and related user interfaces are provided later in this document.

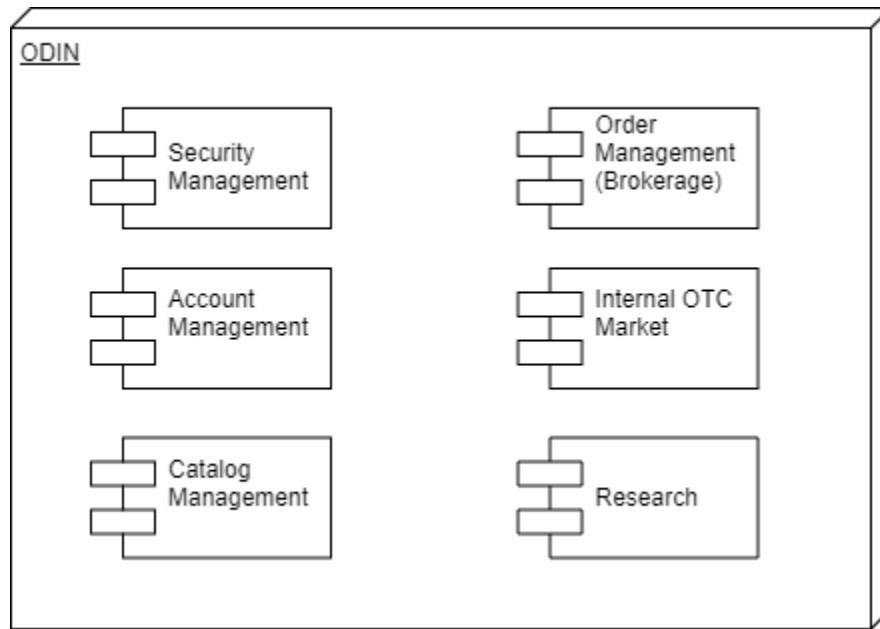


Figure 1. Odin Subdomains

- **Security Management:** Allow account's holders to sign in the platform using a unique login and password;
- **Account Management:** Allow clients to register for an account, view the balance and transactions history;
- **Catalog Management:** Allow administrators to load and manage data related to exchanges, instruments (assets) and markets from external data providers.
- **Internal OTC Market:** an "internal exchange", which allows clients to buy or sell instruments between the account's holders only, without accessing Formal Exchanges (for example, NYSE and NASDAQ). The reason for creating this internal OTC Market is to be able to explore and learn about implementing an Order Matching Engine, and in the future, a Clearing House using Blockchain Technology (FF).
- **Order Management (Brokerage):** Allow clients to buy and sell instruments listed in the Formal Exchanges (NYSE, Nasdaq, for example) or the internal OTC market.
- **Research:** allows clients to access market information generated by research tools. Initially, this module will consume information generated by a Market Sentiment tool that will be created by Julio Cernadas, another student in the Byte Academy Intensive Program. The details of this tool are not part of this document.

## DESIGN CONSIDERATIONS

A software application can be structured using different approaches. Two different Architecture Patterns were considered for this project: Microservice Architecture and Monolithic Architecture. Before we discuss the chosen architecture for Odin, let's look at the overall description, benefits, and drawbacks of each architecture pattern.

### Microservice Architecture

According to MULESOFT (1), *"Microservice capabilities are expressed formally with business-oriented APIs. They encapsulate a core business capability, and as such, are valuable assets to the business. The implementation of the service, which may involve integrations with systems of record, is completely hidden as the interface is defined purely in business terms. The positioning of services as valuable assets to the business implicitly promotes them as adaptable for use in multiple contexts. The same service can be reused in more than one business process or over different business channels or digital touchpoints, depending on need. Dependencies between services and their consumer are minimized by applying the principle of loose coupling. By standardizing on contracts expressed through business-oriented APIs, consumers are not impacted by changes in the implementation of the service. This allows service owners to change the implementation and modify the systems of record or service compositions which may lie behind the interface and replace them without any downstream impact"*.

This solution offers the following benefits and drawbacks, as described by RICHARDSON (2):

#### "Benefits:

- *Enables the continuous delivery and deployment of large, complex applications.*
- *Better testability - services are smaller and faster to test*
- *Better deployability - services can be deployed independently*
- *It enables you to organize the development effort around multiple, auto teams. It enables you to organize the development effort around multiple teams. Each team owns and is responsible for one or more single service. Each team can develop, deploy and scale their services independently of all the other teams*
- *Each microservice is relatively small*
- *Easier for a developer to understand*
- *The IDE is faster making developers more productive*
- *The application starts faster, which makes developers more productive, and speeds up deployments*

- *Improved fault isolation. For example, if there is a memory leak in one service then only that service will be affected. The other services will continue to handle requests. In comparison, one misbehaving component of a monolithic architecture can bring down the entire system*
- *Eliminates any long-term commitment to a technology stack. When developing a new service, you can pick a new technology stack. Similarly, when making major changes to an existing service you can rewrite it using a new technology stack.*

#### Drawbacks:

- *Developers must deal with the additional complexity of creating a distributed system*
- *Developer tools/IDEs are oriented on building monolithic applications and don't provide explicit support for developing distributed applications*
- *Testing is more difficult*
- *Developers must implement the inter-service communication mechanism*
- *Implementing use cases that span multiple services without using distributed transactions is difficult*
- *Implementing use cases that span multiple services requires careful coordination between the teams*
- *Deployment complexity. In production, there is also the operational complexity of deploying and managing a system comprised of many different service types*
- *Increased memory consumption. The microservice architecture replaces N monolithic application instances with NxM services instances. If each service runs in its own JVM (or equivalent), which is usually necessary to isolate the instances, then there is the overhead of M times as many JVM runtimes. Moreover, if each service runs on its own VM (e.g. EC2 instance), as is the case at Netflix, the overhead is even higher".*

### Monolithic Architecture Pattern

By contrast, MULESOFT (1) says that: "A monolithic application is built as a single unit. Enterprise Applications are built in three parts: a database (consisting of many tables usually in a relational database management system), a client-side user interface (consisting of HTML pages and/or JavaScript running in a browser), and a server-side application. This server-side application will handle HTTP requests, execute some domain-specific logic, retrieve and update data from the database, and populate the HTML views to be sent to the browser. It is a monolith – a single logical executable. To make any alterations to the system, a developer must build and deploy an updated version of the server-side application."

This solution offers the following benefits and drawbacks, as described by RICHARDSON (3):

#### "Benefits:



- *Simple to develop - the goal of current development tools and IDEs is to support the development of monolithic applications*
- *Simple to deploy - you simply need to deploy the directory hierarchy on the appropriate runtime*
- *Simple to scale - you can scale the application by running multiple copies of the application behind a load balancer*

Drawbacks:

- *The large monolithic code base intimidates developers, especially ones who are new to the team. The application can be difficult to understand and modify. As a result, development typically slows down. Also, because there are not hard module boundaries, modularity breaks down over time. Moreover, because it can be difficult to understand how to correctly implement a change in the quality of the code declines over time. It's a downward spiral.*
- *Overloaded IDE - the larger the code-base the slower the IDE and the less productive developers are.*
- *Overloaded web container - the larger the application the longer it takes to start up. This had had a huge impact on developer productivity because of time wasted waiting for the container to start. It also impacts deployment too.*
- *Continuous deployment is difficult - a large monolithic application is also an obstacle to frequent deployments. To update one component, you must redeploy the entire application. This will interrupt background tasks (e.g. Quartz jobs in a Java application), regardless of whether they are impacted by the change, and possibly cause problems. There is also the chance that components that haven't been updated will fail to start correctly. As a result, the risk associated with redeployment increases, which discourages frequent updates. This is especially a problem for user interface developers since they usually need to iterative rapidly and redeploy frequently.*
- *Scaling the application can be difficult - a monolithic architecture is that it can only scale in one dimension. On the one hand, it can scale with an increasing transaction volume by running more copies of the application. Some clouds can even adjust the number of instances dynamically based on load. But on the other hand, this architecture can't scale with an increasing data volume. Each copy of application instance will access all the data, which makes caching less effective and increases memory consumption and I/O traffic. Also, different application components have different resource requirements - one might be CPU intensive while another might memory intensive. With a monolithic architecture we cannot scale each component independently*
- *The obstacle to scaling development - A monolithic application is also an obstacle to scaling development. Once the application gets to a certain size it's useful to divide up the engineering organization into teams that focus on specific functional areas. For example, we might want to have the UI team, accounting team, inventory team, etc. The trouble with a monolithic application is that it prevents the teams from working independently. The teams*

*must coordinate their development efforts and redeployments. It is much more difficult for a team to make a change and update production.*

- *Requires a long-term commitment to a technology stack - a monolithic architecture forces you to be married to the technology stack (and in some cases, to a particular version of that technology) you chose at the start of development. With a monolithic application, can be difficult to incrementally adopt a newer technology. For example, let's imagine that you chose the JVM. You have some language choices since as well as Java you can use other JVM languages that inter-operate nicely with Java such as Groovy and Scala. But components written in non-JVM languages do not have a place within your monolithic architecture. Also, if your application uses a platform framework that subsequently becomes obsolete then it can be challenging to incrementally migrate the application to a newer and better framework. It's possible that to adopt a newer platform framework you have to rewrite the entire application, which is a risky undertaking."*

# 4. SYSTEM ARCHITECTURE

## OVERVIEW

The overall system architecture is shown in Figure 2.

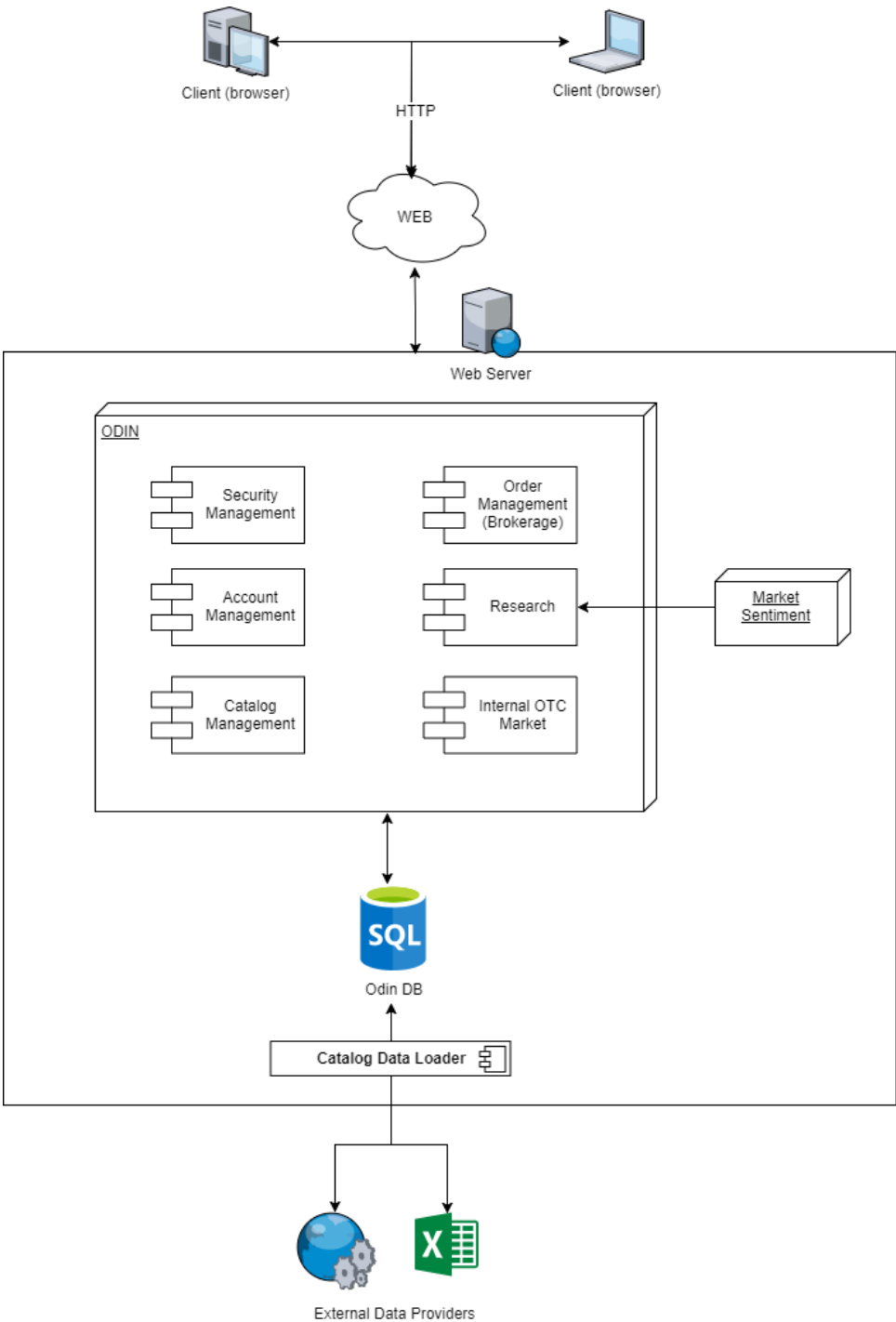


Figure 2. Odin System Architecture

## ARCHITECTURAL STRATEGY

The core of Odin Platform will be developed using a Monolithic Architecture pattern with a centralized database, which is simpler to develop and deploy. The main reason for choosing this architecture is the timeframe constraint related to the project: four weeks. However, a few sub-components related to loading data from external data providers and saving this data in the application database will be developed as independent services. But at this stage, these services will share the same database with the core application.

Although a Microservice Architecture facilitates scaling, loose coupling between components and independency between teams and development languages, one of its drawbacks is the additional complexity of creating a distributed system and implementing an inter-service communication protocol. This drawback directly reflects in the time needed for development, which would certainly result in not meeting the timeframe constraint associated with this project.

## DEPENDENCIES AND CONSTRAINS

The application will be developed as a client/server Web Application, available to its users through a web browser. All application components and the database will be installed in a single Web Server.

Users must use a browser installed in a desktop/laptop computer to access the application. The application won't adjust to browsers installed on mobile devices. Specific mobile client applications (IOS, Android, etc.) won't be developed.

This project will be developed using Python3 and Postgres database. Additional languages such as JavaScript and Solidity may be used in future features if time is enough to learn such languages and apply this knowledge into the project. A detailed list of python packages will be provided at the end of the implementation, as an update to this document.

## 5. SUBDOMAINS

This section describes each subdomain of Odin platform, identifying its components, a list of features and mockups of main user interfaces related to each subdomain. As described in the section PURPOSE, Odin is a “mock trading platform”, meaning that no real transaction (deposit, withdrawal) or order will be carried out in the real world, nor real money will be used in the platform. Also, remember that the acronym “MVP” will be used to identify the Minimum Viable Product features, and the acronym “FUTF” will be used to identify Future Features.

Figure 3 shows Odin’s subdomains and the components of each subdomain. While current components may be composed of MVP and FUTF Features, Future Components are only composed of Future Features.

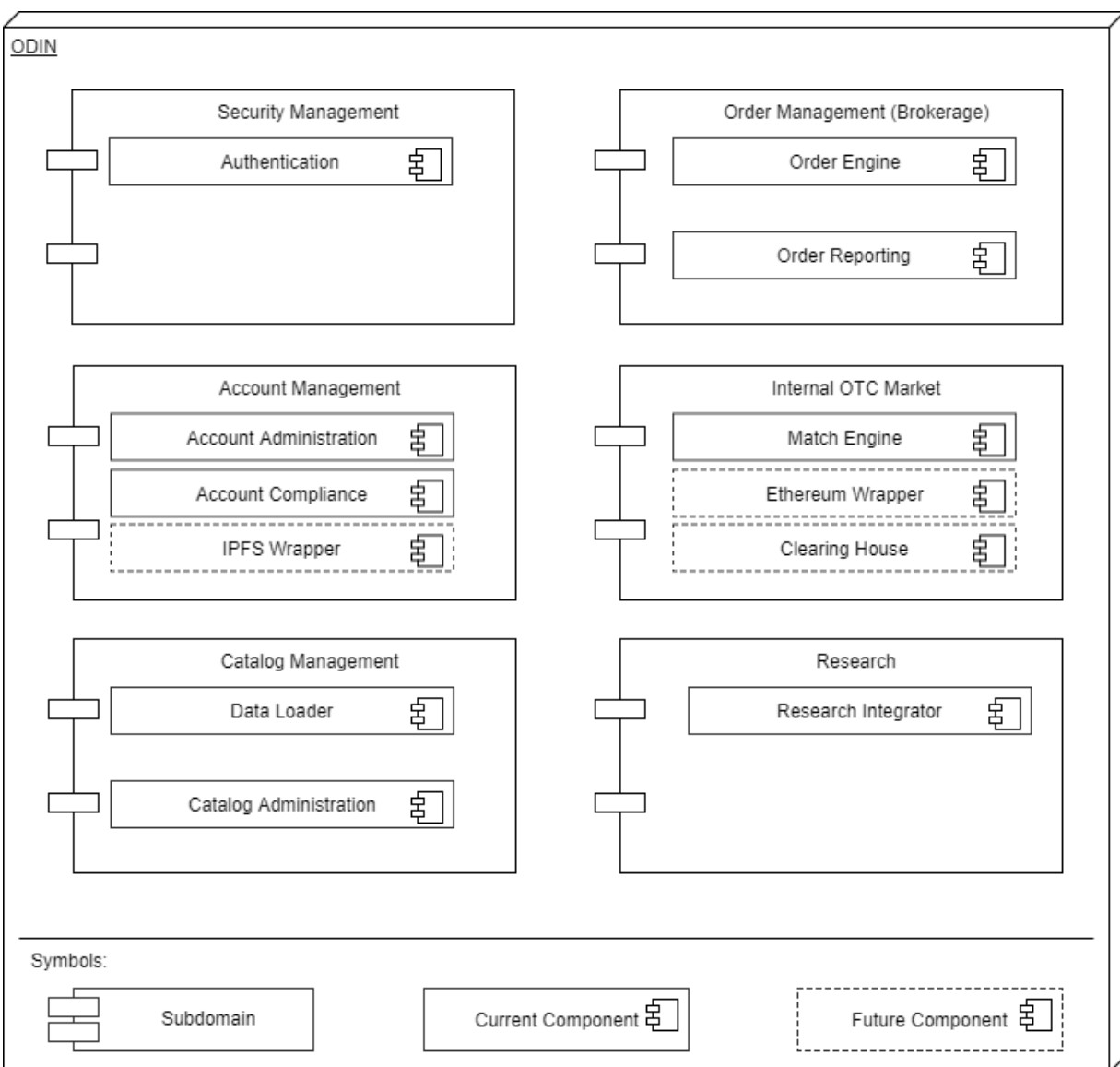


Figure 3. Odin Subdomains' Components

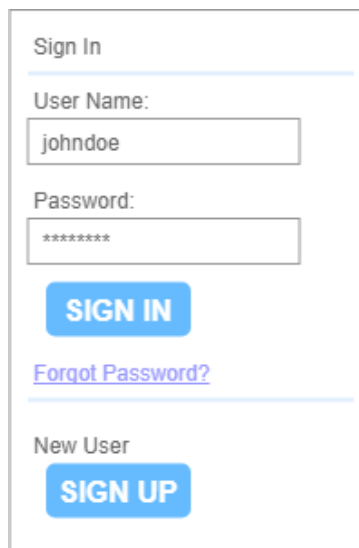
## SECURITY MANAGEMENT

This subdomain is responsible to ensure that only registered users connect to the platform. It is implemented in a single component name “Authentication”.

### Features

- Allow a user to sign up to the platform, by registering a combination of username and password (MVP);
- Allow a user to sign in to the platform using the username and password registered (MVP);
- Allow a user to update its password, assuming it knows its current username and password (MVP);
- Allow a user to request a password reset, in case it forgets its password (FUTF);
- Allow a user to enable/disable 2FA (Two Factor Authentication) (FUTF);
- Register user activity, by recording the timestamp and IP address in every sign in (FUTF).

### User Interfaces



The image shows a user interface for signing in and signing up. It is divided into two sections by a horizontal line. The top section is titled "Sign In" and contains a "User Name:" label with a text input field containing "johndoe", a "Password:" label with a password input field containing "\*\*\*\*\*", a blue "SIGN IN" button, and a blue link "Forgot Password?". The bottom section is titled "New User" and contains a blue "SIGN UP" button.

Figure 4. User Sign In and Sign Up

## ACCOUNT MANAGEMENT

This subdomain is responsible for providing clients a way to manage their accounts, by updating their data information, uploading documents, viewing account balances, depositing and withdrawing funds, and view transactions history (orders, deposits, withdrawals).

At this stage, this subdomain is composed of two components:

- Account Administration: responsible for account creations, updates, deposit and withdrawals of funds, as well as providing reporting services as account balance, and deposit and withdrawal history.
- Account Compliance: responsible for ensuring that the account meets KYC (Know Your Customer) requirements;

However, one of the Future Features (FUTF) listed below relies on integration with IPFS protocol. This feature will be implemented as a third component, an independent service named “IPFS Wrapper”, that will be deployed to serve as an abstraction layer to the IPFS protocol.

### Features

- Create a client account and associate it with a user (MVP);
- Update client account data (MVP);
- Upload client documents as part of the KYC process (MVP);
- Allow the client to deposit and withdrawal funds to/of its account (simulation) (MVP);
- Allow the client to view a history of deposits and withdrawals (MVP);
- Allow the client to check its account balance (MVP);
- Allow the client to capture a live picture from its computer and upload it to its account as part of the KYC process (FUTF);
- Validate the user identify by comparing the document picture with a live picture uploaded (FUTF);
- Allow the client to generate a wallet number to trade cryptocurrencies (FUTF).
- Allow the client to generate multiple wallet numbers to trade cryptocurrencies (FUTF).
- Save client documents in the IPFS network (FUTF).

## User Interfaces

### Account

---

-Attachment 1

-Attachment 2

-Attachment 3

Create / Update

Figure 5. Account Creation / Update

Account	Balance: 000.00						
<hr/>							
Deposits							
<table><tr><th>Date</th><th>Amount</th></tr><tr><td>00/00/0000 00:00:00</td><td>000,000.00</td></tr><tr><td>00/00/0000 00:00:00</td><td>000,000.00</td></tr></table>		Date	Amount	00/00/0000 00:00:00	000,000.00	00/00/0000 00:00:00	000,000.00
Date	Amount						
00/00/0000 00:00:00	000,000.00						
00/00/0000 00:00:00	000,000.00						
Withdrawals							
<table><tr><th>Date</th><th>Amount</th></tr><tr><td>00/00/0000 00:00:00</td><td>000,000.00</td></tr><tr><td>00/00/0000 00:00:00</td><td>000,000.00</td></tr></table>		Date	Amount	00/00/0000 00:00:00	000,000.00	00/00/0000 00:00:00	000,000.00
Date	Amount						
00/00/0000 00:00:00	000,000.00						
00/00/0000 00:00:00	000,000.00						

Figure 6. Account Transactions History



## CATALOG MANAGEMENT

This subdomain is responsible to allow system administrators register external data providers and define which exchanges, instruments, and markets will be listed for in the trading platform. It will be divided into two components:

- **Data Loader:** an independent service that will load data from external data providers, parse it and save it to the database. This component has no user interface;
- **Catalog Administration:** a module, part of Odin's core, that allows administrators to manage the data to be listed in the trading platform. It will rely on the data saved to the database by the Data Loader component.

The use of the Data Loader as an independent service allows data loading to be carried out by another application process. At this stage, it will run on the same server as the core application. However, in the future, it may be deployed to another server, releasing resources from the server to the core application.

### Features

- Load data from registered external data providers and save it to the database (MVP);
- Allow administrator to moderate which exchanges, instruments, and markets will be made available in the trading platform (MVP);
- Allow administrator to perform CRUD operations related to exchanges, instruments, and markets (FUTF).

Listings	
Exchanges	
Symbol	Allowed
NYSE	YES
NASDAQ	NO
Instruments	
Symbol	Allowed
AAPL	YES
BTC	NO
Markets	
Symbol	Allowed
AAPL-USD	YES
BTC-USD	NO

Figure 7. Administrator's Moderation of Listed Exchanges, Instruments, and Markets

## ORDER MANAGEMENT (BROKERAGE)

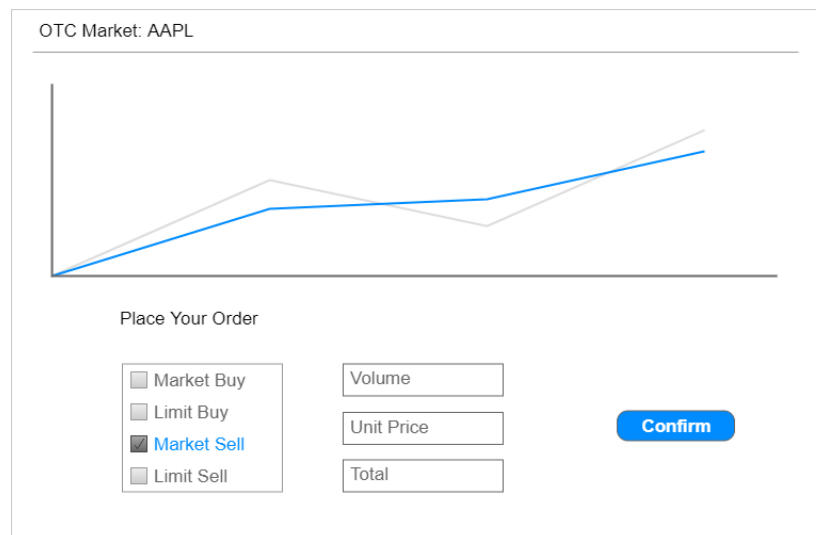
This subdomain is responsible for allowing clients to send orders to buy or sell instruments in a specific market, listed on the trading platform. It is also responsible for canceling unfilled open orders, as well as showing information related to the order book or ticker/quote of a specific instrument. At this stage, it will be divided into two components:

- Order Engine: Responsible to send buy, sell or cancel orders to an exchange;
- Order Reporting: Responsible to present live and historical data related to ticks/quotes of an instrument, as well as the account's open and closed orders.

### Features

- Allow a client to send an order to Market Buy or Market Sell in an external exchange (MVP);
- Allow a client to send an order to Market/Limit Buy or Market/Limit Sell in the internal OTC Market (MVP);
- Allow a client to send an order to cancel an open Market/Limit Buy or Market/Limit Sell order (MVP);
- Allow a client to view the order book of an instrument in the internal OTC Market (MVP);
- Allow a client to view the last tick/quote of an instrument in a given market/exchange (MVP);
- Allow a client to view tick history of an instrument in a given market/exchange (MVP);
- Allow a client to view its open / closed orders; (MVP)
- Allow the client to send a conditional order to Market/Limit Buy or Market/Limit Sell in the OTC Market (FUTF);
- Allow the client to view the order book of an instrument in external markets/exchanges (FUTF);

### User Interface



OTC Market: AAPL

Place Your Order

☐ Market Buy  
☐ Limit Buy  
☒ Market Sell  
☐ Limit Sell

Volume  
 Unit Price  
 Total

Confirm

Figure 8. Buy or Sell Orders

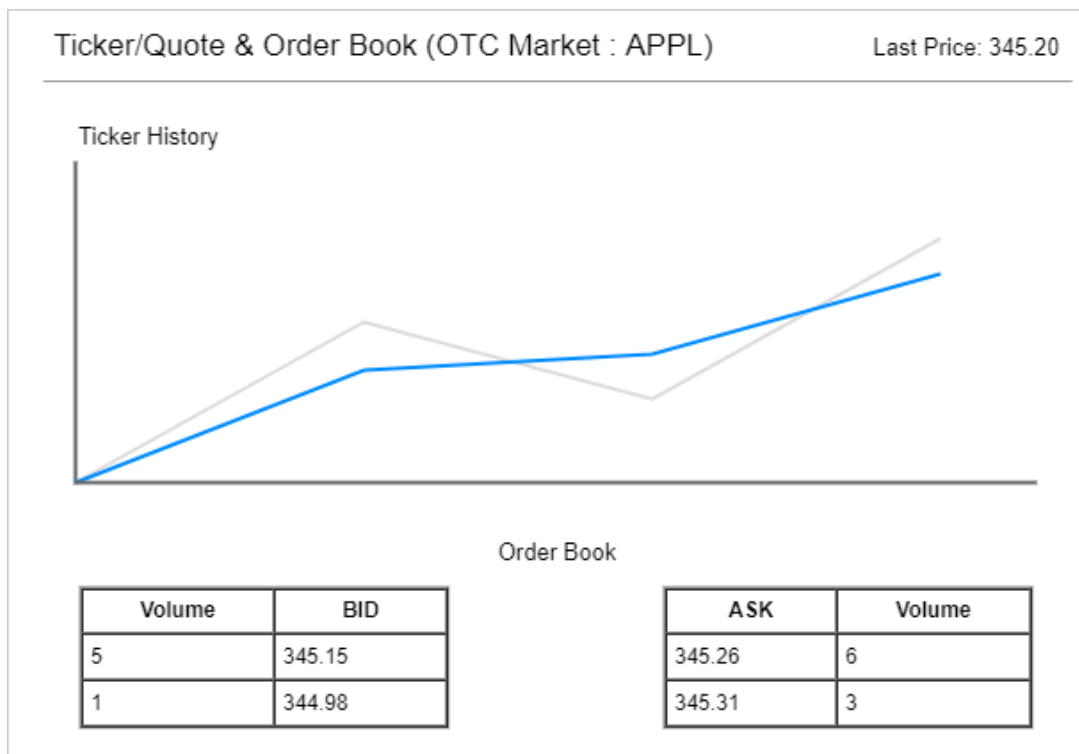


Figure 9. Instrument Last Tick/Quote, Tick History and Order Book

<b>Orders</b>			
<hr/>			
<b>Open</b>			
<hr/>			
Market   Symbol		Volume   Price	
NASDAQ   AAPL		02   352.18	<a href="#">Cancel</a>
NASDAQ   MSFT		05   150.03	<a href="#">Cancel</a>
<hr/>			
<b>Closed</b>			
<hr/>			
Market   Symbol		Volume   Price	
NASDAQ   AAPL		03   354.18	
NASDAQ   MSFT		06   154.18	

Figure 10. Open / Closed Orders and Cancel Open Orders

## INTERNAL OTC MARKET

This subdomain is responsible for implementing an internal exchange mechanism to offer clients an internal OTC market. It does not have a user interface, because the ORDER MANAGEMENT (BROKERAGE) subdomain is responsible for it.

At this stage, this subdomain is composed of a single component named “Match Engine”. However, one of the Future Features (FUTF) listed below relies on integration with Ethereum protocol. This feature will be implemented as two distinct components:

- Ethereum Wrapper: an independent service that will be deployed to serve as an abstraction layer to the Ethereum protocol;
- Clearing House: one or a set of smart contracts, responsible for implementing the functionality of a Clearing House to the OTC Market.

### Features

- Receive and add to the order book, a client’s order Market/Limit Buy or Market/Limit Sell in the OTC Market (MVP);
- Receive and remove from the order book, a client’s order to cancel an open Market/Limit Buy or Market/Limit Sell order (MVP);
- Provide information related to the order book (MVP);
- Match Market/Limit Buy and Market/Limit Sell Orders (MVP);
- Receive and add to the order book, a client’s conditional order to Market/Limit Buy or Market/Limit Sell in the OTC Market (FUTF);
- Perform the clearing process of the OTC Market transactions on the Ethereum protocol (FUTF).

## RESEARCH

This subdomain is responsible to present to the user the result of research analysis conducted by external services/applications. This subdomain is composed of a single component, named “Research Integrator”, that will integrate with a Market Sentiment Analysis tool to be developed by Julio Cernadas, another student at Byte Academy Intensive Program.

### Features

- Present to the client the result of the Market Sentiment Analysis for a specific instrument (MVP).

### User Interface

A final interface for this module will depend on the capabilities available on the external tool. A possible user interface is shown below:

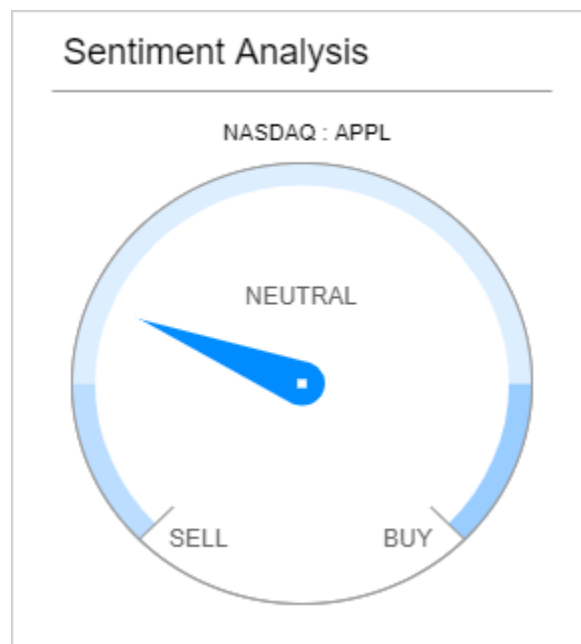


Figure 11. Sentiment Analysis

## 6. FUTURE SYSTEM ARCHITECTURE

The future system architecture, considering MVP and Future Features, is shown in Figure 12Figure 2.

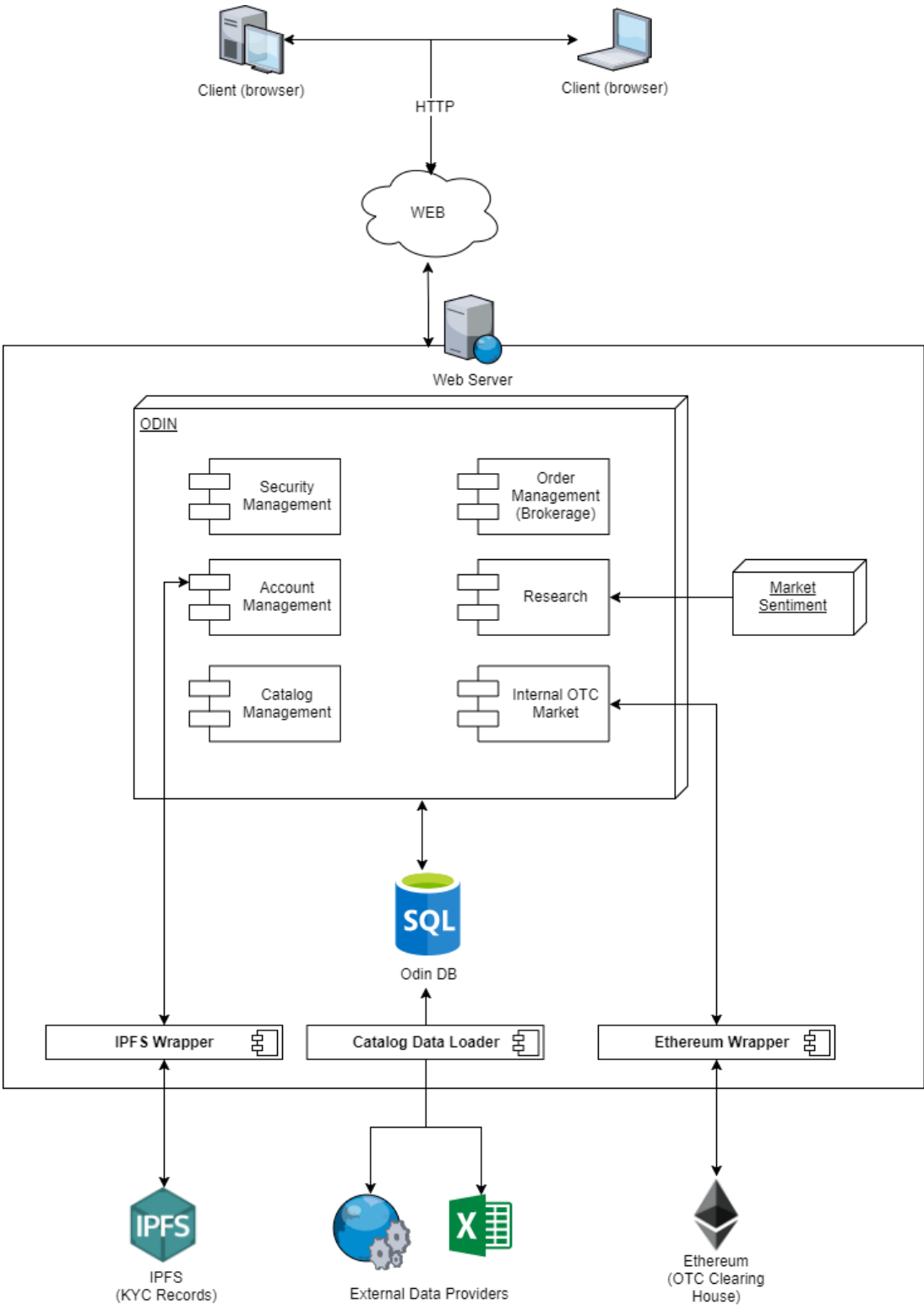


Figure 12. System Architecture Including Future Features (that are not part of this stage of the project)

## 7. REFERENCES

1. MULESOFT. Microservices vs Monolithic Architecture. <https://www.mulesoft.com/resources/api/microservices-vs-monolithic>. Accessed on: 07/22/2018.
2. RICHARDSON, CHRIS. Pattern: Microservice Architecture. <http://microservices.io/patterns/microservices.html>. Accessed on: 07/22/2018.
3. RICHARDSON, CHRIS. Pattern: Monolithic Architecture. <http://microservices.io/patterns/monolithic.html>. Accessed on: 07/22/2018.