Distributed Systems Project Report

Alex O'Donnell	Cian Kelly	Danny Murray	Eoin Mc Mahon
20456522	20429616	19331403	20387436

Synopsis:

Application Domain

Our project is a cryptocurrency management platform designed for users interested in managing various crypto-related activities in a unified environment.

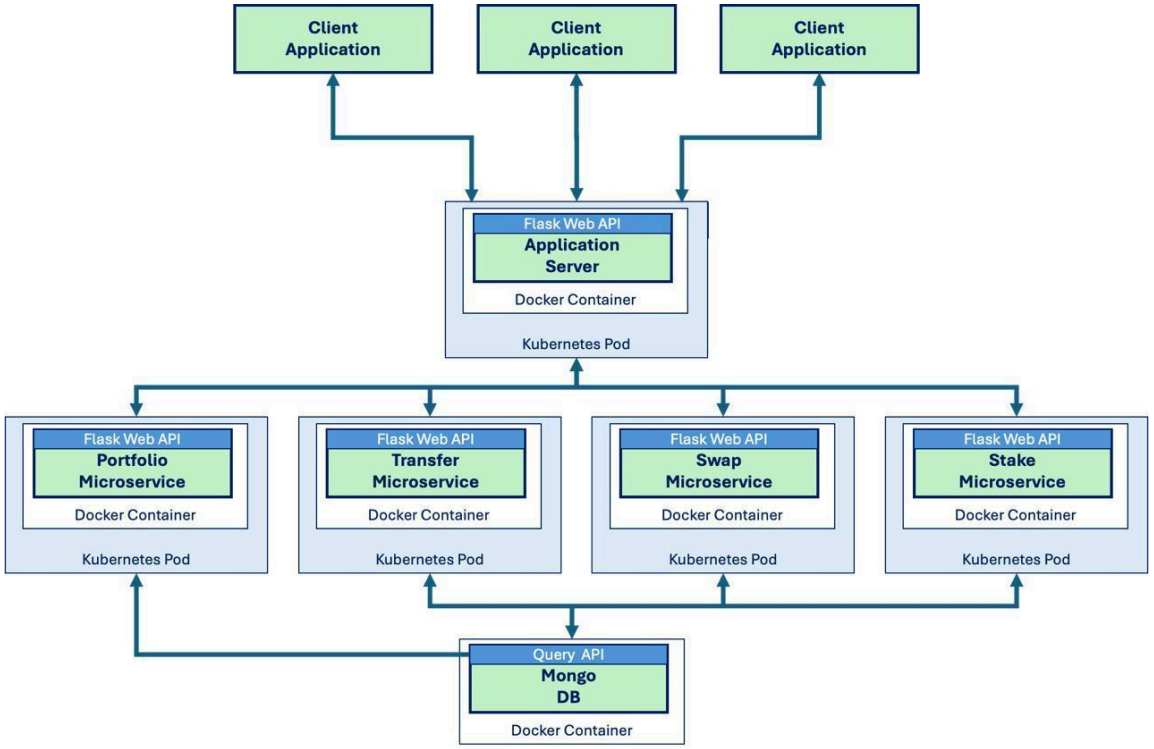
System Functionality

The platform provides a comprehensive suite of services tailored for cryptocurrency enthusiasts and investors. Here's what users can expect:

1. **Login and Sign up**
    a. Login: Users who have already registered an account have prompted a secure authentication mechanism to ensure that access to user accounts is protected.
    b. Sign Up: New users are able to sign up for an account, create a wallet address and password, and load an initial Solana balance.
2. **Portfolio Viewing:** Users can easily view the details of their cryptocurrency holdings, helping them manage their assets effectively.
3. **Stake Solana:** This feature allows users to stake their Solana tokens, enabling them to earn rewards on their holdings.
4. **Transfer:** Users can transfer their cryptocurrencies to other users securely within the platform.
5. **Token Swap on Exchange with Live Pricing:** Offers the ability to swap different cryptocurrencies directly on the exchange, supported by real-time pricing to ensure the best possible rates.
6. **View Swap History and Swap Details:** The swap microservice not only handles the exchange transactions but also allows users to view their past swaps and retrieve detailed information about each swap using a specific reference number.

Microservices Architecture

Each key feature of the platform — Portfolio, Swap, Stake, and Transfer — is implemented as a distinct microservice. This architectural choice enhances the scalability and resilience of the system, allowing each component to be developed, maintained, and scaled independently.



Technology Stack

The following list outlines the main technologies utilized in our distributed systems project, detailing their specific functions and the reasons for their selection.

Python

Used for: General programming language for building the application's logic and processing data
Reason: Python's simplicity and powerful libraries make it ideal for rapid development and handling various data types

Flask

Used for: A micro web framework to build and serve restful web APIs
Reason: Flask provides the flexibility needed for our microservices architecture due to its lightweight and modular design.

Gunicorn

Used for: A Python WSGI HTTP Server for UNIX, serving Flask applications across multiple workers.
Reason: Gunicorn was chosen for its ability to handle multiple requests simultaneously, improving the efficiency and scalability of our web services.

Flasgger (Swagger UI via Flask)

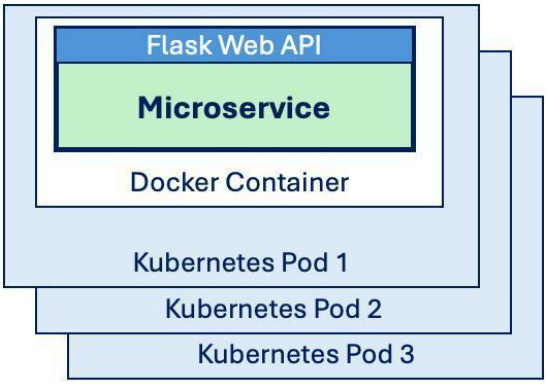
Used for: API documentation and testing
Reason: Flasgger was implemented to automatically generate and visualize API documentation, simplifying both development and API endpoint testing.

Docker

Used for: Containerizing the application and its environments
Reason: Docker simplifies deployment and provides consistency across development, testing, and production environments, reducing compatibility issues

Kubernetes

Used for: Container orchestration and management, to implement fault tolerance and scalability
Reason: Kubernetes supports our system's scalability and load balancing needs, ensuring efficient management of containerized services. Kubernetes pods ability to maintain themselves and restart if killed allows for the implementation of fault tolerance.



MongoDB

Used for: Storing and managing databases in a format that supports our service's structure needs
Reason: MongoDB is a NoSQL database ideal for handling large volumes of distributed data. Its flexibility with schema-less data and performance efficiency under high load conditions make it suitable for our application's data management requirements.

System Overview

There are three main parts to our system. The client/server, our microservices and our database. Each of the containers in the diagram below are orchestrated by Kubernetes. The following image is a diagram of how our system interacts with each other:

Actors connect to our server, which acts like a broker to handle requests to our microservices as seen in the diagram above. Each microservice uses Flask to create a REST API framework. As Flask comes with a built-in development server which is not suitable for production use, after the development of our microservices we created a dedicated WSGI (Web Server Gateway Interface) server using Gunicorn. This allows us to handle multiple concurrent connections efficiently, making them suitable for serving web applications with high traffic loads, and improving the scalability of our system.

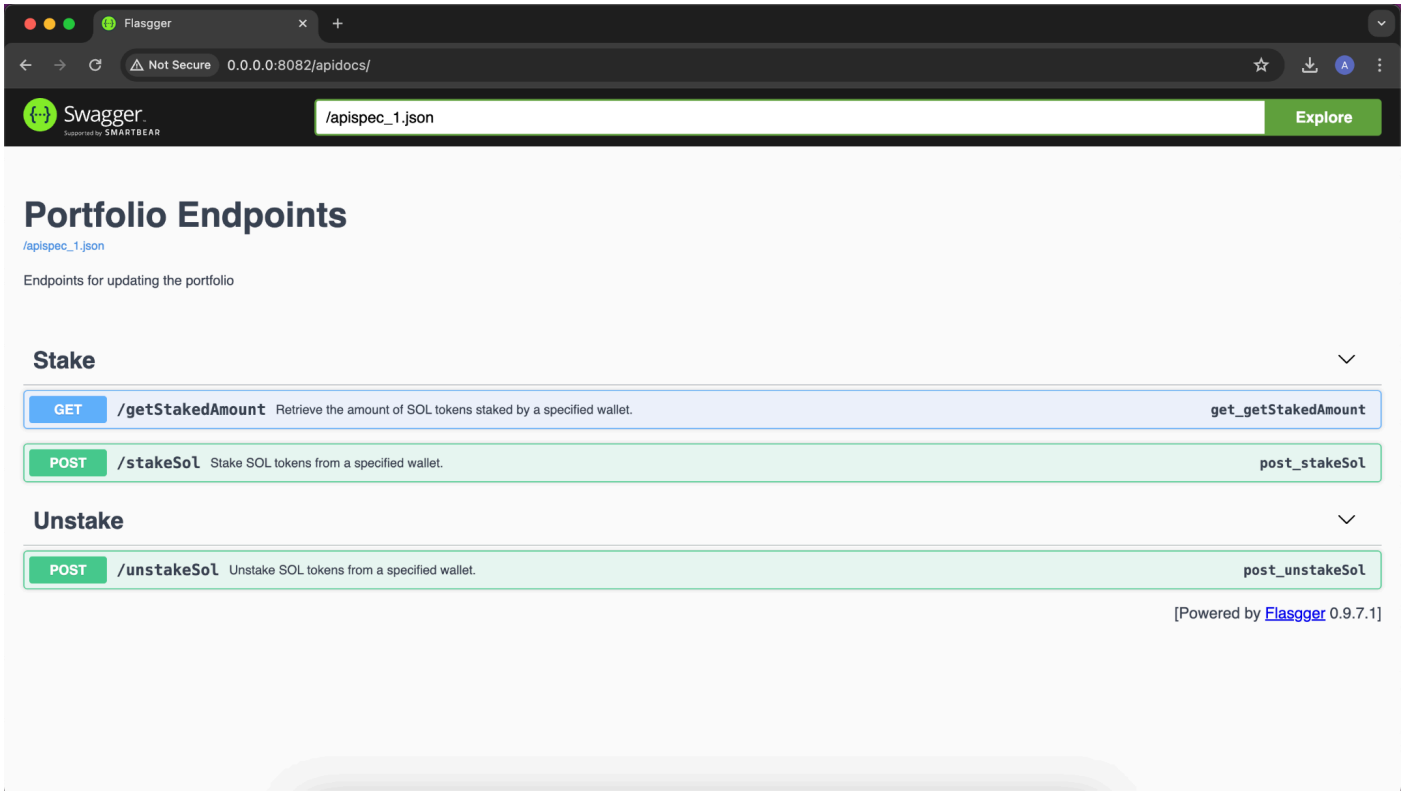
The first thing an actor sees when connecting to our server is the option to login or create an account. If they log in, their wallet address and password are checked against the database to see if it exists. If they decide to create a new wallet, the wallet address, password and solana balance are added to our database. We decided to use a MongoDB database as it is a No-SQL database, and is designed to handle large volumes of distributed data. Additionally,

MongoDB has horizontal scalability, meaning that it can easily distribute data across multiple nodes in a cluster. This allows us to scale our database horizontally by adding machines to the cluster as our data and workload grow, without requiring significant changes to our application, making it more easily scalable.

Leveraging Docker containers and Kubernetes for scalability and fault tolerance, Flask serves as the primary interface for microservices interaction. Each microservice operates on a dedicated port, enabling actors to connect and submit HTTP POST or GET requests to specific endpoints hosted on the corresponding microservice port. Based on the actor's request, the microservices interact with the database to either update user information or retrieve requested data in response to received requests. As seen in the diagram above, the only microservice that doesn't allow writes to the database is portfolio, as it only allows users to view the current state of their portfolio.

Flasgger, a Flask extension emulating Swagger, was also employed as it automates the generation and visualization of API documentation. It automatically documents endpoints and parameters based on function annotations and route definitions.

To increase scalability and fault tolerance, we tailored the configuration of our Kubernetes YAML files to our needs. We increased the replica count such that if one pod fails, another can take over. We also enabled liveness and readiness probes. This helps Kubernetes know when to restart a pod if it becomes unresponsive and when it's ready to start serving traffic. We also implemented a rolling update strategy, as it minimizes downtime and ensures there's always a certain number of pods running. During the testing of our system, we employed a manual pod deletion test to observe if Kubernetes automatically creates a new one to maintain the desired number of replicas. As expected, this worked perfectly. We also used scaling tests to manually scale our deployment up and ensure that Kubernetes can properly manage the number of pods if we were to ever receive a surge in traffic. Again this worked perfectly as seen in the video in our submission. This use of Kubernetes vastly increased our fault tolerance, ensuring that the system could still function as expected if there were to be a problem with any of our pods. It also increased scalability, as it ensures that there are the necessary number of pods running if a surge were to occur.



Example of API Documentation generated via Swagger