DAW-GPT GLOBAL OVERVIEW

INDEX

ARCHITECTURE

The architecture of the DAW GPT web app consists of three main components: the frontend developed in Next.js, the backend built in Python Flask, and the MongoDB database.

The frontend, based on Next.js, handles the user interface and data presentation.

The backend, implemented with Flask, is responsible for processing requests from the frontend. Through Flask, routes are defined that correspond to different functionalities of the application, enabling the execution of necessary operations. For example, it can access the database to retrieve or modify data in response to requests.

The MongoDB database is used as the data management system for the application. Through the pymongo library, the backend can interact with the database for data access, reading, and writing.

The interaction between the frontend and backend occurs through HTTP requests, specifically GET and POST. The frontend sends GET requests to obtain specific data from the database, while POST requests are used to send data to the backend for operations such as creation or updating of elements.

MONGO DB COLLECTION AND DOCUMENT

The MongoDB database used in the DAW GPT web app contains two main collections: "USERS" and "PROJECTS". Below is the description of the structure of both collections.

Collection "USERS":

The "USERS" collection stores user information. Each document represents a user and has the following structure:

- **_id**: an automatically generated unique identifier for each document
- **username**: the username of the user.
- **password**: the password associated with the user's account.
- **Available_Balance**: the available balance in the user's account, represented as a decimal number (balance available for Withdraw/Reinvest).
- **public_key**: the public key associated with the user.
- **transactions**: an array of transactions made by the user, each represented as an object containing the following parameters:
  - **Date:** the date of the transaction.
  - **Amount:** the amount of the transaction, represented as a decimal number.
  - **TransactionHash:** the unique hash of the transaction.
  - **Status:** the status of the transaction.

- **AffiliateCode**: the user's affiliate code.
- **Current_balance**: the current balance of the user, represented as a decimal number (balance that generates income).
- **Project:** The Name or the Id of the project where the user is subscribed.

Collection "PROJECTS":

The "PROJECTS" collection contains the performance levels of the project that the user is subscribed to. Each document represents a project and has the following structure:

- **_id**: an automatically generated unique identifier for each document.
- **RendimentoMensile_Investito**: the monthly return on investment, represented as a decimal number.
- **RendimentoMensile_Livello1**: the monthly return at level 1 of the project, represented as a decimal number.
- **RendimentoMensile_Livello2**: the monthly return at level 2 of the project, represented as a decimal number.
- **RendimentoMensile_Livello3**: the monthly return at level 3 of the project, represented as a decimal number.
- **RendimentoMensile_Livello4**: the monthly return at level 4 of the project, represented as a decimal number.
- **ProjectName**: the name of the project.
- **AffiliateList**: an array containing the affiliate codes associated with the project.

N.B. 1.0
Currently, the DAWGPT code does not use the Project collection because it is set to work on a single test project. It uses two collections that divide the Project collection into dividends and AffiliateList. Therefore, a small update to the Python code needs to be made.

N.B 2.0
The objects in the Projects collection need to include the parameters **"Wallet_Public_key"** and **"Wallet_Private_key"** of the wallet that will serve as the "liquidity pool" in order to automate user withdrawals.

PYTHON CODE

Here is a description of the libraries used:

- **Flask**: Flask is a lightweight framework for creating web applications in Python. It is used to handle HTTP request routing, define endpoints, and generate corresponding HTTP responses.
- **render_template**: It is a Flask function that allows dynamically generating HTML pages using predefined templates. In this code, it may be used to generate the HTML pages to be displayed in the web application.
- **request**: Flask module for handling incoming HTTP requests. It is used to extract data sent via HTTP request.
- **redirect**: Flask module for request redirection. It is used to redirect the user to a specific page after a completed operation.
- **url_for**: Flask module for URL generation. It is used to dynamically generate URLs within the application.

- **jsonify**: Flask module for serializing data into JSON format. It is used to return JSON responses from various endpoints.
- **flask_login**: Flask-Login is a library that simplifies user authentication in Flask. It provides functionality for managing user session, login, logout, and access control to restricted pages.
- **pymongo**: PyMongo is a MongoDB driver for Python. It is used to connect to the MongoDB database and interact with it.
- **flask_cors**: Flask-CORS is a library that handles cross-origin requests (CORS) in Flask. It is used to allow requests from origins other than the application's host.
- **bson**: PyMongo module for handling MongoDB-specific data types (e.g., ObjectId). It is used to serialize and deserialize data between Python and MongoDB.
- **AffiliatesTools**: It is a custom module (defined in another code) containing functions for managing affiliates.
- After the initial part of the code that imports the necessary libraries, the app object is initialized as a Flask application. A secret key for the session is set (**app.secret_key**), and CORS request handling is enabled using **CORS (app)**.
- Subsequently, the code sets up the connection to the MongoDB database using PyMongo. A client object is created for database connection, and objects are initialized to access the collections in the MongoDB database.
- After the database configuration, the code defines a User class that represents a user of the application. The class inherits from Flask-Login's UserMixin class and is used to handle user authentication.
- The code continues by defining various endpoint functions for the web application. Each endpoint corresponds to a specific URL that can be requested by the client. Below are descriptions of some of the endpoints present in the code:
- **/register**: This endpoint handles user registration. It accepts HTTP GET and POST requests. In the case of a POST request, data is extracted from the request object, and user registration is performed in the database.
- **/login**: This endpoint handles user login. It accepts HTTP GET and POST requests. In the case of a POST request, data is extracted from the request object, and credential verification is performed to allow access.
- **/dashboard**: This endpoint returns dashboard information for an authenticated user. It returns user data such as current balance, available balance, public key, and transaction list.
- **/update_public_key**: This endpoint handles updating the user's public key. It accepts HTTP POST requests and updates the public key in the database.
- **/Reinvest_balance**: This endpoint handles reinvestment of available balance by the user. It accepts HTTP POST requests and updates the available balance and current balance of the user in the database.
- **/WithdrawCrypto**: This endpoint handles the withdrawal of a specific amount of cryptocurrency by the user. It accepts HTTP POST requests and performs a withdrawal operation using the specified recipient address and private key.
- **/Affiliates**: This endpoint returns information about the affiliates of the authenticated user. It returns details about the affiliation levels (level1, level2, level3, level4) using a custom function **build_invites_chain**.
- **/Rewards**: This endpoint returns information about rewards for the authenticated user. It returns balances and rewards associated with the affiliation levels using custom functions **BuildNetworkBalances**.
- **/RefreshBalances**: The "RefreshBalances" endpoint is a periodically executed endpoint as a crontab job on the 15th day of each month. Its function is to update the available balances of users in the database based on the generated returns.

HOW THE TRANSACTIONS WORK?

Transactions in DAWGPT enable the transfer of funds between users and the project's wallet. There are primarily two types of transactions:

- **USER-FUND** Transaction: This transaction is initiated when a user clicks on a specific button. Upon clicking the button, Metamask, a digital wallet that supports blockchain transactions, is opened. The transaction is pre-filled with the amount specified by the user and the recipient wallet address, which corresponds to the project's wallet hosted on DAWGPT. Once Metamask is opened, the user is prompted to confirm and send the transaction. This type of transaction allows users to send funds directly and immediately to the project's wallet. Once the transaction is completed and verified, the user's ***current balance*** is increased by the specified value in the transaction.

- **FUND-USER** Transaction: This type of transaction is executed on the server-side. Before executing the transaction, it is verified that the withdrawal amount is less than or equal to the user's available balance. Using the web3 library and the Infura provider, the server can access the blockchain network. By knowing the private key and public key of the fund's wallet, which are stored in the MongoDB database, the server creates a custom transaction towards the recipient user. This process involves creating and executing a transaction through the blockchain network using the private key of the fund's wallet. Once the transaction is executed, the funds are transferred from the project's wallet to the user. The user's ***current balance*** is decreased by the specified value in the transaction once the transaction is verified.

LAST UPDATE TO DO

The last functions to be developed in DAWGPT include:

- **Transaction Verification**: After executing a transaction, it is important to verify if it has been confirmed and included in the blockchain to correctly credit or decrement the user's Available Balance.
- **Auto Reinvestment**: Allows users to automatically reinvest their earnings or generated returns in the project as soon as the Available Balance is updated.
- **License Page and System**: The license page where users can purchase the project's three licenses and implement the "taxation" system derived from the licenses during profit distribution.