UNIVERSITE ABDELMALEK ESSAADI

FACULTE DES SCIENCES ET TECHNIQUES DE TANGER

Rapport

EcoCoin Platform Backend Report

MASTER SIT&BIGDATA

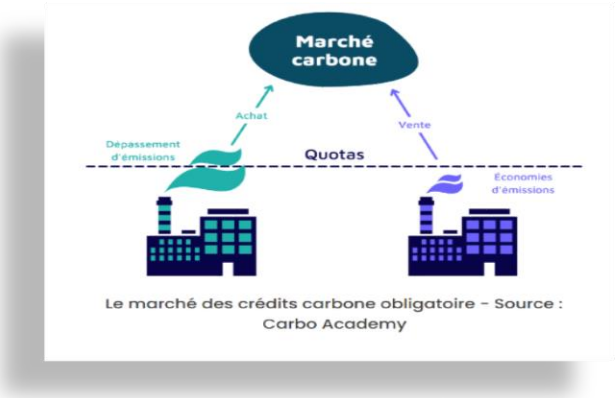
Réalisé par : Doumi Salma & Ababri Chaimae

Année universitaire 2024-2025

Overview Platform

The EcoCoin Platform is a microservice-based architecture designed to facilitate transactions using Carbon MCO2 tokens. It enables users to manage wallets, perform transactions, track market prices, and interact with blockchain-based services. The platform allows users to purchase carbon credits to offset their CO_2 emissions, with each credit corresponding to one ton of CO_2 offset and represented as a digital token on the Ethereum blockchain

The platform allows users to purchase carbon credits to offset their CO_2 emissions. Each carbon credit corresponds to one ton of CO_2 offset and is represented as a digital token on the Ethereum blockchain.



Le marché des crédits carbone obligatoire - Source : Carbo Academy

FastAPI and REST

FastAPI is a modern, high-performance
web framework for building APIs with Python. It enables developers to create applications quickly and securely, offering features such as data validation, dependency injection, and support for asynchronous request handling.

Python Tutorials – Real Python

REST (Representational State Transfer) is an architectural style used for building web services. It provides a set of constraints that a web service should adhere to in order to be considered "RESTful." The key principles of REST include:

- **Identification of resources through URIs**: Each resource is uniquely identified by a Uniform Resource Identifier (URI).
- **Uniform interface for interacting with resources**: A consistent and standardized approach is used to interact with resources, typically through HTTP methods like GET, POST, PUT, and DELETE.
- **Self-descriptive messages**: Each message contains enough information to describe how to process the message, enhancing clarity and understanding.

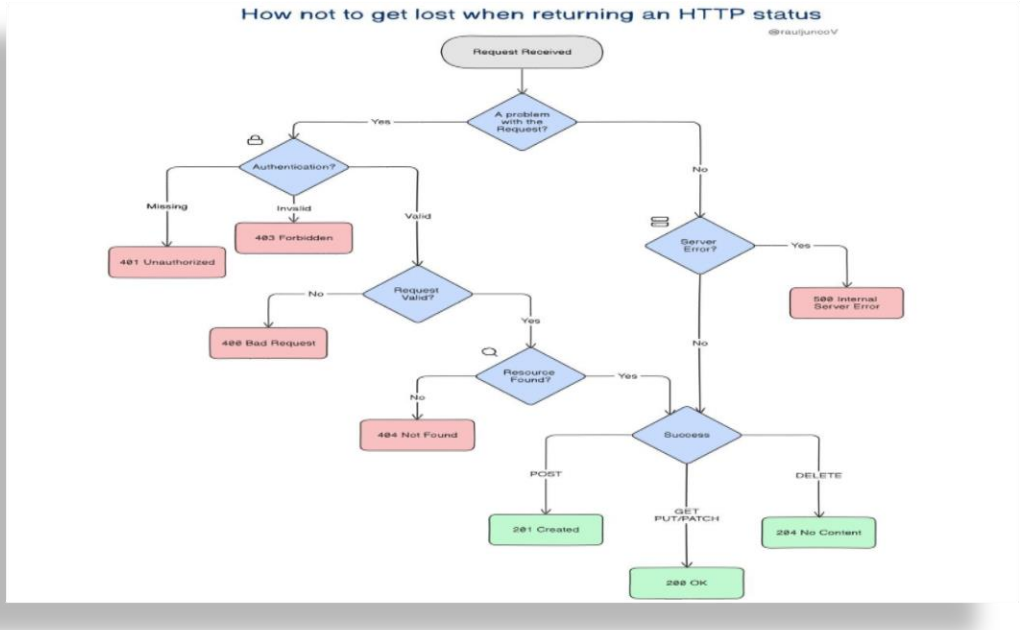
- **Hypermedia as the engine of application state**: Clients interact with resources entirely through hypermedia provided dynamically by application servers.

These principles ensure that web services are scalable, stateless, and can be easily understood and interacted with by clients.

HTTP Status Codes

Selecting the correct HTTP status code when responding to a request is crucial for clear communication with clients. Common status codes include:

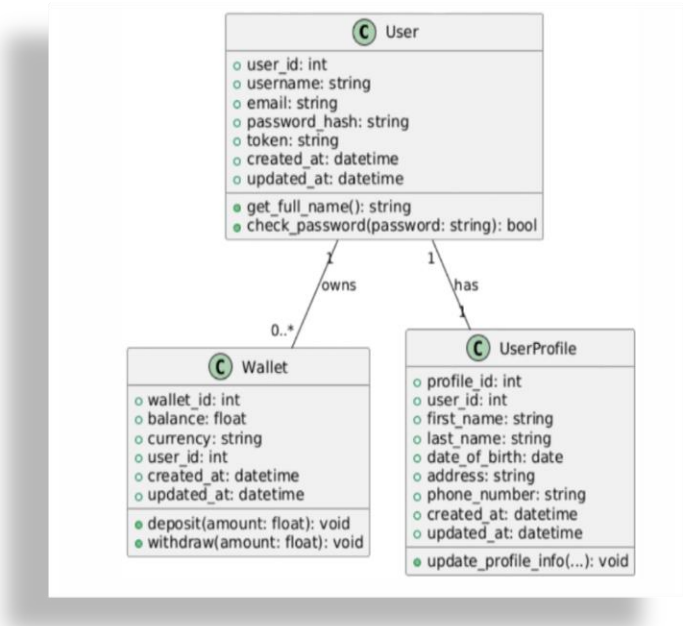
- **200 OK**: The request has succeeded.
- **201 Created**: The request has been fulfilled, resulting in the creation of a new resource.
- **204 No Content**: The server successfully processed the request, but is not returning any content.
- **400 Bad Request**: The server could not understand the request due to invalid syntax.
- **401 Unauthorized**: The client must authenticate itself to get the requested response.
- **403 Forbidden**: The client does not have access rights to the content.
- **404 Not Found**: The server can not find the requested resource.
- **500 Internal Server Error**: The server has encountered a situation it doesn't know how to handle.



This document outlines the backend architecture, microservices, databases, endpoints, and key processes involved.

Data Modeling :

User_diagram



User: Represents a user with attributes and methods, including a `token` for authentication.

Wallet: Represents the user's wallet, with a `currency` attribute typed by the `WalletCurrency` enumeration.

UserProfile: Contains the user's personal information.

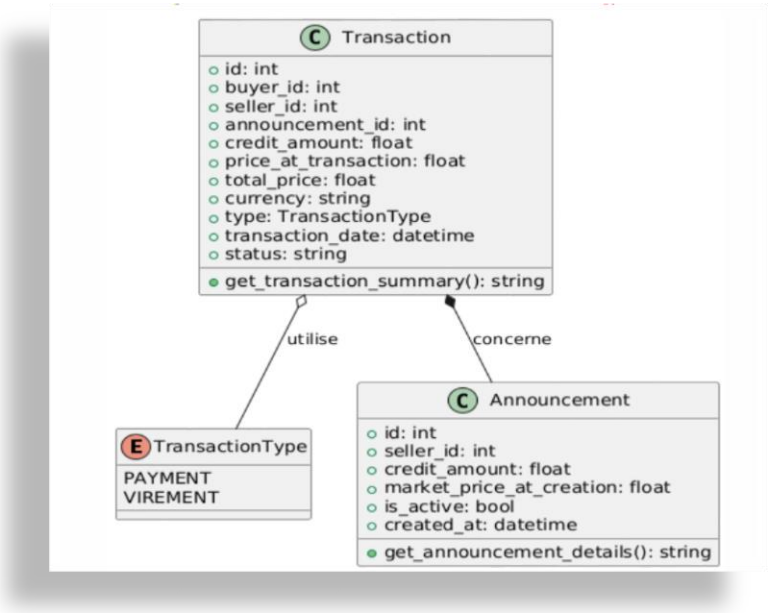
- Relationships:

  A user has a profile (`has`).

  A user can own multiple wallets (`owns`).

Transactions_diagram :

Transaction: Represents a transaction with its attributes, including the type defined by the `TransactionType` enumeration.



Announcement: Represents a carbon credit listing put up for sale by a user.

- Relationships:

An announcement can include multiple transactions (`includes`).

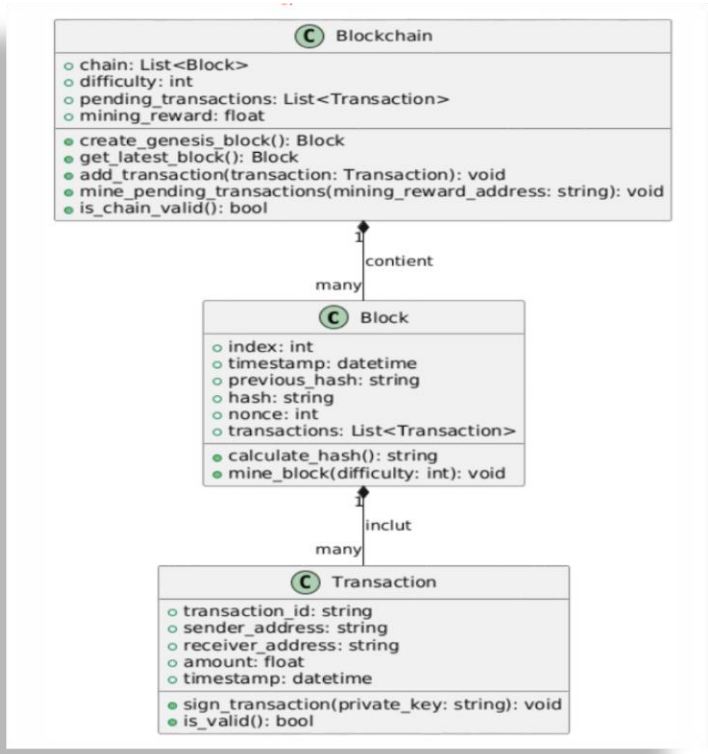
Blockchain_diagram :

CarbonAccount: Manages user registration and wallet linking.

CarbonPayment: Handles carbon credit payments and fiat-to-token conversion.

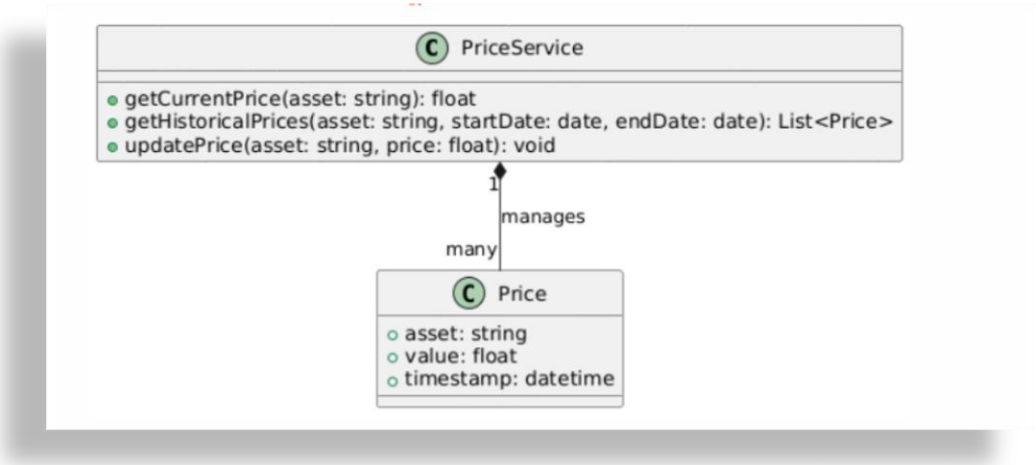
CarbonStaking: Manages staking and unstaking of carbon credits.

CarbonToken: Implements standard ERC-20 token functionalities.

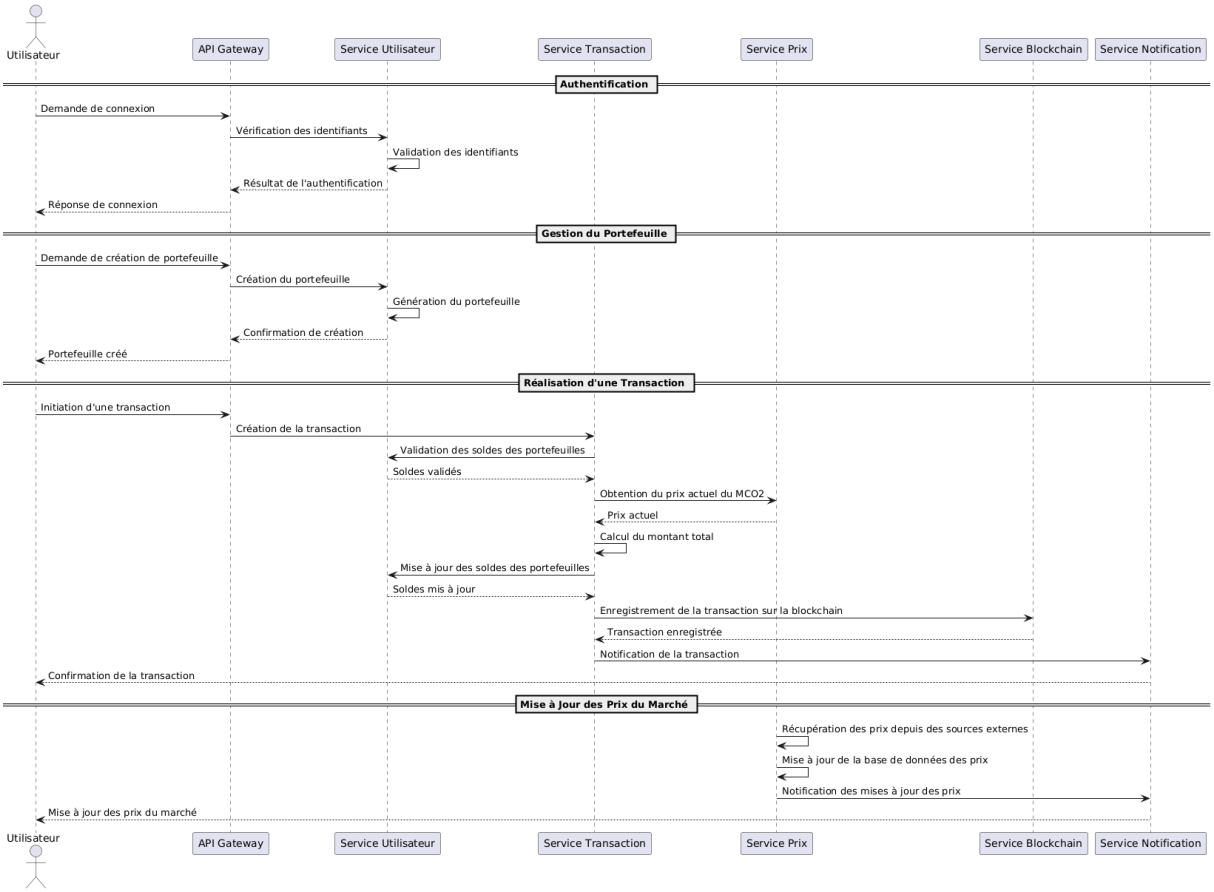


Prices_diagram :

- MarketPrice: Represents the market price of a currency pair at a specific timestamp



Microservices Overview



1. API Gateway

- **Purpose**: Acts as the entry point for all client requests, routing them to the appropriate microservices.
- **Technologies**: FastAPI
- **Responsibilities**:

  - Request routing
  - Authentication and authorization (via JWT/OAuth2)
  - Centralized error handling

2. User Service

- **Purpose**: Manages user accounts and wallets.
- **Database**: user_service_db (PostgreSQL)

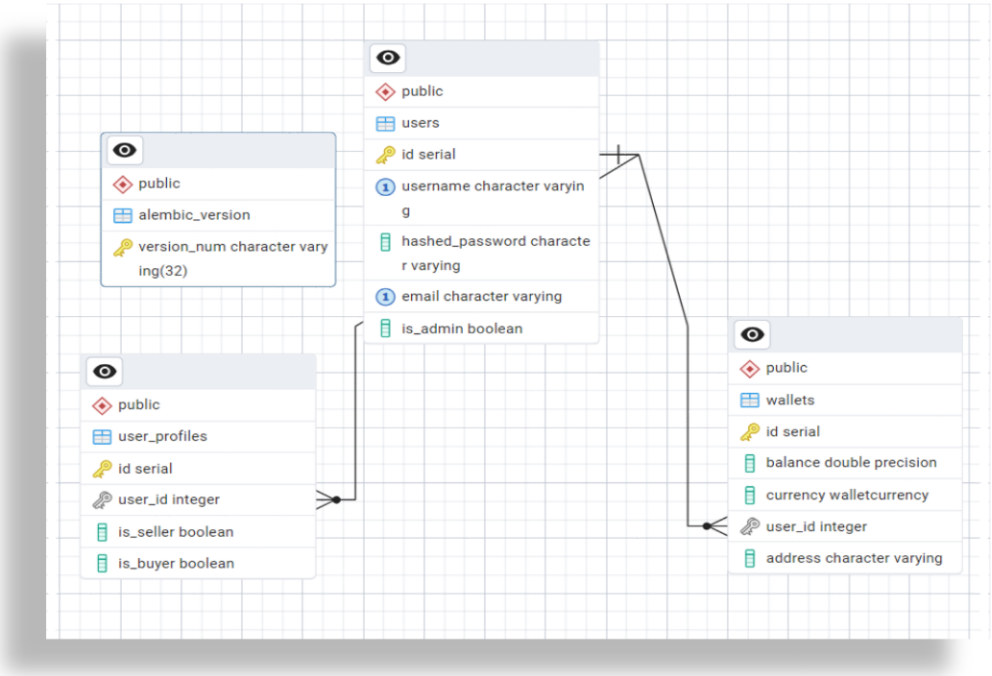
The **User Service** is a microservice dedicated to managing users, profiles, and wallets (USD, ETH, CARBON) for a decentralized system. It provides features such as

authentication, user creation and management, and operations on wallets and user roles.

This service plays a central role in the overall architecture by interacting with other microservices like transaction_service and price_service.

- **Database Diagram**

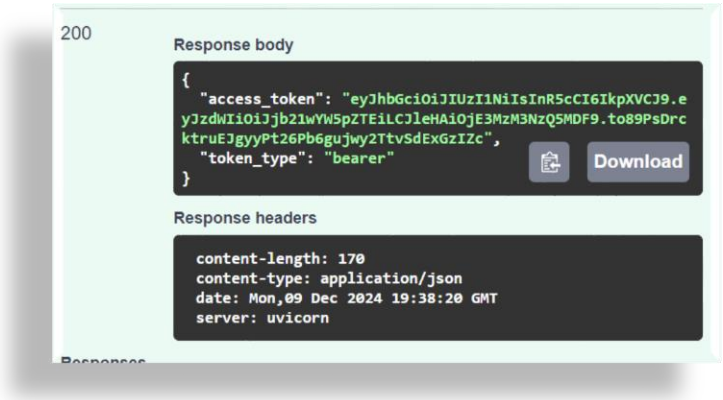
The data model relies on three main tables:



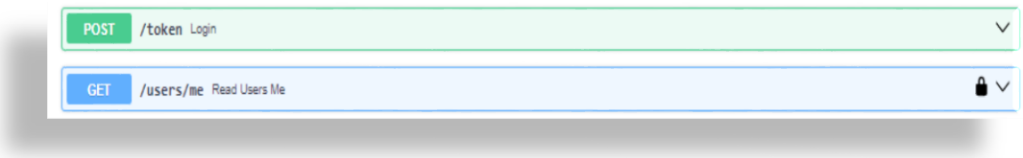
- **Core Features**
  - Authentication

```python
# ================================================
# Authentification & gestion des utilisateurs
# ================================================
@app.post("/token")
def login(form_data: OAuth2PasswordRequestForm = Depends(), db: Session = Depends(get_db)):
    user = authenticate_user(db, form_data.username, form_data.password)
    if not user:
        raise HTTPException(status_code=401, detail="Invalid credentials")
    access_token = create_access_token(data={"sub": user.username})
    return {"access_token": access_token, "token_type": "bearer"}

@app.get("/users/me", response_model=UserResponse)
def read_users_me(token: str = Depends(oauth2_scheme), db: Session = Depends(get_db)):
    """Vérifier et retourner l'utilisateur connecté via son token."""
    payload = verify_access_token(token)
    if not payload:
        raise HTTPException(status_code=401, detail="Invalid token")

    username = payload.get("sub")
    user = get_user_by_username(db, username)
    if not user:
        raise HTTPException(status_code=404, detail="User not found")
    return user
```

- **Endpoints:**
  - POST /token: Generates a JWT token after verifying user credentials.
  - GET /users/me: Returns the authenticated user's information.



- **Security:**
  - Passwords are hashed with bcrypt before being stored.
  - Sessions are secured using signed JWT tokens.



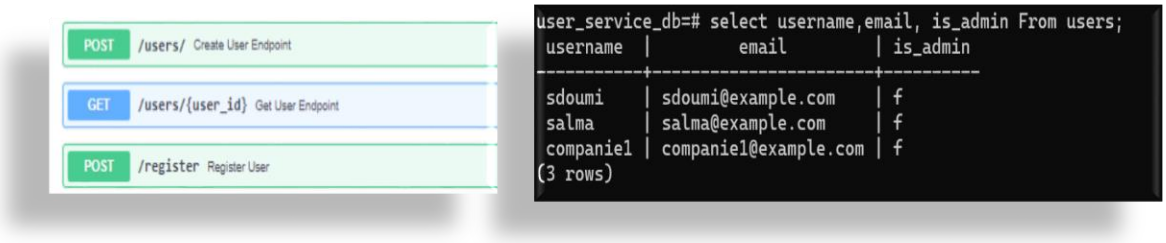
- User Management

```python
def create_user(db: Session, user: UserCreate) -> User:
    """Créer un nouvel utilisateur."""
    if db.query(User).filter(User.username == user.username).first():
        raise HTTPException(status_code=400, detail="Nom d'utilisateur déjà utilisé")
    if db.query(User).filter(User.email == user.email).first():
        raise HTTPException(status_code=400, detail="Email déjà utilisé")

    hashed_password = hash_password(user.password)

    db_user = User(
        username=user.username,
        email=user.email,
        hashed_password=hashed_password,
        is_admin=user.is_admin,
    )
    try:
        db.add(db_user)
        db.commit()
        db.refresh(db_user)
        wallet_types = [WalletCurrency.USD, WalletCurrency.ETH, WalletCurrency.CARBON]
        for currency in WalletCurrency:  # Iterate through enum directly
            print(f"Creating wallet for {currency}")
            wallet_data = WalletCreate(
                user_id=db_user.id,
                balance=0.0,
                currency=currency  # Use the enum directly
            )
            create_wallet_service(db,wallet_data)
        return db_user
```

```python
@app.post("/users/", response_model=UserResponse)
def create_user_endpoint(user: UserCreate, db: Session = Depends(get_db)):
    """Créer un utilisateur et ses wallets par défaut."""
    new_user = create_user(db, user)
    # Création automatique des wallets pour l'utilisateur (USD, ETH, CARBON)
    return new_user
```

- **Endpoints:**

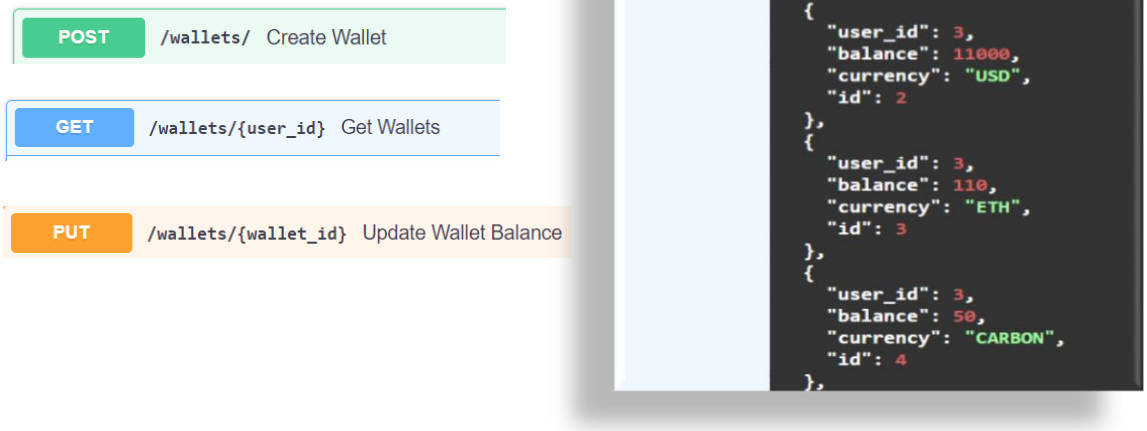
Creates a user with a default profile.



- Wallet Management

```python
# ====================================================
# Gestion des wallets
# ====================================================
@app.post("/wallets/", response_model=WalletResponse)
def create_wallet(wallet: WalletCreate, db: Session = Depends(get_db)):
    return create_wallet_service(db, wallet)

@app.get("/wallets/{user_id}", response_model=list[WalletResponse])
def get_wallets(user_id: int, db: Session = Depends(get_db)):
    wallets = get_wallets_by_user_service(db, user_id)
    return [WalletResponse.from_orm(wallet) for wallet in wallets]

@app.put("/wallets/{wallet_id}", response_model=WalletResponse)
def update_wallet_balance(wallet_id: int, balance: float, db: Session = Depends(get_db)):
    wallet = update_wallet_balance_service(db, wallet_id, balance)
    if not wallet:
        raise HTTPException(status_code=404, detail="Wallet not found")
    return wallet
```

- **Endpoints**



- Profile Management

```python
# ================================================
# Gestion des profils utilisateurs
# ================================================
@app.post("/user_profiles/", response_model=UserProfileResponse)
def create_user_profile(profile: UserProfileCreate, db: Session = Depends(get_db)):
    return create_user_profile_service(db, profile)

@app.get("/user_profiles/{user_id}", response_model=UserProfileResponse)
def get_user_profile(user_id: int, db: Session = Depends(get_db)):
    profile = get_user_profile_service(db, user_id)
    if not profile:
        raise HTTPException(status_code=404, detail="User profile not found")
    return profile

@app.put("/user_profiles/{user_profile_id}", response_model=UserProfileResponse)
def update_user_profile(user_profile_id: int, is_seller: bool, is_buyer: bool, db: Session = Depends(get_db)):
    profile = update_user_profile_service(db, user_profile_id, is_seller, is_buyer)
    if not profile:
        raise HTTPException(status_code=404, detail="User profile not found")
    return profile
```

Libraries Used

- **FastAPI:** Framework to build REST APIs.
- **SQLAlchemy:** ORM to manage interactions with PostgreSQL.
- **bcrypt:** Password hashing.
- **jose:** JWT token generation and verification.
- **Alembic:** Database migration management.

The **User Service** uses **FastAPI** to build RESTful APIs, defining endpoints to manage users, their profiles, and their wallets. Data is validated using Pydantic models, and session management is streamlined with dependency injection. **SQLAlchemy** acts as an ORM to interact with PostgreSQL, allowing data manipulation through Python objects while managing table relationships. User security is ensured by **bcrypt**, which hashes passwords before storage, and **jose**, which handles authentication via secure

JWT tokens containing user information and an expiration date. Finally, **Alembic** is used to manage database migrations, ensuring that the schema remains synchronized with the code models. These tools work together to deliver a robust, secure, and extensible microservice.

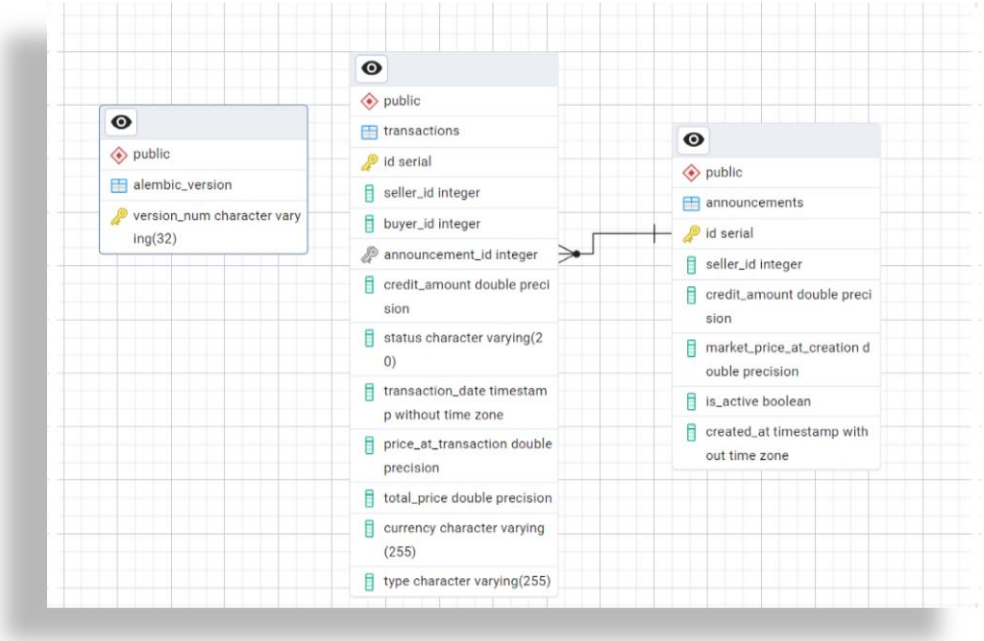
3. Transaction Service

- **Purpose**: Handles transactions, including payments, transfers, and credit purchases.
- **Database**: transaction_service_db (PostgreSQL)

The **Transaction Service** is a microservice responsible for managing announcements and handling transactions, including payments and transfers (virements). It integrates with other services such as the **Price Service** for market prices and the **User Service** for wallet balance validations and updates. This service ensures secure and accurate processing of transactions while maintaining a detailed record for auditing and tracking purposes.

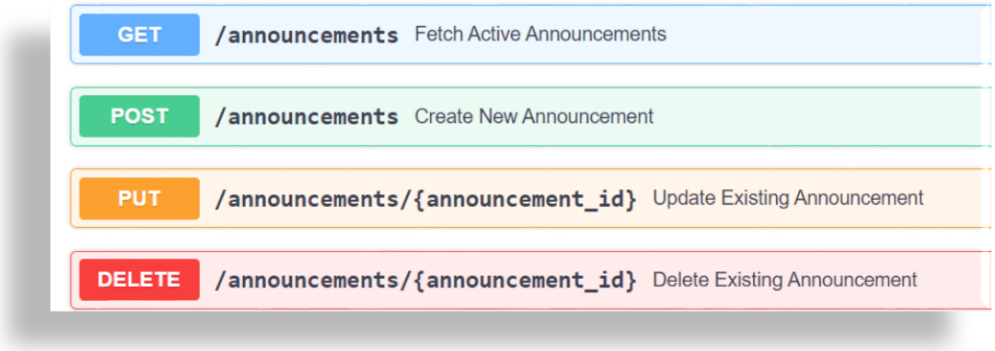
- **Data Model**

– Announcements : Represents a credit listing created by a seller.
– Transactions : Records the movement of credits or funds between users.



- **Functionalities**

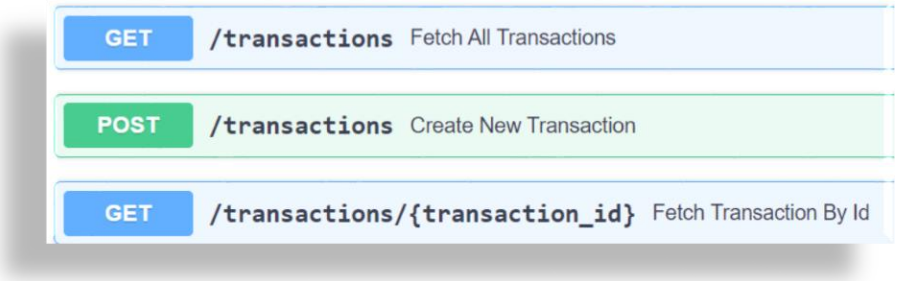
– Announcements Management



– Management transaction:

```python
def create_transaction(db: Session, transaction_data: TransactionCreate):
    """Créer une transaction (payment ou virement)."""
    if transaction_data.type == TransactionType.PAYMENT:
        if transaction_data.currency != "USD":
            raise ValueError("Only USD is allowed for transactions involving MCO2")
        announcement = db.query(Announcement).filter(Announcement.id == transaction_data.announcement_id).first()
        if not announcement or not announcement.is_active:
            raise ValueError("Invalid or inactive announcement")
        seller_balance = get_wallet_balance(announcement.seller_id, "USD")
        buyer_balance = get_wallet_balance(transaction_data.buyer_id, "USD")

        if seller_balance is None:
            raise ValueError(f"Seller with ID {announcement.seller_id} does not have a USD wallet")
        if buyer_balance is None:
            raise ValueError(f"Buyer with ID {transaction_data.buyer_id} does not have a USD wallet")
        try:
            market_price = get_price("MCO2")
        except Exception as e:
            raise ValueError(f"Failed to fetch market price: {e}")
        total_price = transaction_data.credit_amount * market_price
        if buyer_balance < total_price:
            raise ValueError("Insufficient funds in buyer's wallet")
        update_wallet_balance(
            wallet_id=announcement.seller_id,
            new_balance=seller_balance + transaction_data.credit_amount,
            user_id=announcement.seller_id,
            currency="USD"
        )
        update_wallet_balance(
            wallet_id=transaction_data.buyer_id,
            new_balance=buyer_balance - total_price,
            user_id=transaction_data.buyer_id,
            currency="USD"
        )
```

```
  update_wallet_balance(
      wallet_id=transaction_data.buyer_id,
      new_balance=buyer_balance - total_price,
      user_id=transaction_data.buyer_id,
      currency="USD"
  )
  transaction = Transaction(
      seller_id=announcement.seller_id,
      buyer_id=transaction_data.buyer_id,
      announcement_id=transaction_data.announcement_id,
      credit_amount=transaction_data.credit_amount,
      price_at_transaction=market_price,
      total_price=total_price,
      type=transaction_data.type,
  )
  db.add(transaction)
  db.commit()
  db.refresh(transaction)
  return transaction
if transaction_data.type == TransactionType.VIREMENT:
  sender_balance = get_wallet_balance(transaction_data.buyer_id, transaction_data.currency)
  recipient_balance = get_wallet_balance(transaction_data.recipient_id, transaction_data.currency)
  if sender_balance is None:
      raise ValueError(f"Sender with ID {transaction_data.buyer_id} does not have a {transaction_data.currency} wallet")
  if recipient_balance is None:
      raise ValueError(f"Recipient with ID {transaction_data.recipient_id} does not have a {transaction_data.currency} wallet")
  if sender_balance < transaction_data.credit_amount:
      raise ValueError("Insufficient funds in sender's wallet")
  update_wallet_balance(
      wallet_id=transaction_data.buyer_id,
      new_balance=sender_balance - transaction_data.credit_amount,
      user_id=transaction_data.buyer_id,
      currency=transaction_data.currency
  )
  update_wallet_balance(
      wallet_id=transaction_data.recipient_id,
      new_balance=recipient_balance + transaction_data.credit_amount,
      user_id=transaction_data.recipient_id,
      currency=transaction_data.currency
  )

  transaction = Transaction(
      buyer_id=transaction_data.buyer_id,
      seller_id=transaction_data.recipient_id,
      credit_amount=transaction_data.credit_amount,
      currency=transaction_data.currency,
      type=transaction_data.type,
  )
  db.add(transaction)
  db.commit()
  db.refresh(transaction)
```



- **Payment Transactions**:

1. Fetch the announcement and validate that it's active.
2. Validate wallet balances for both buyer and seller.
3. Fetch the current market price via the **Price Service**.
4. Calculate the total price and ensure the buyer has sufficient funds.
5. Update wallet balances for the buyer and seller.

6. Record the transaction in the database.

– **Transfer Transactions (VIREMENT)**:

1. Validate wallets for both sender and recipient.
2. Ensure the sender has sufficient balance.
3. Update wallet balances for both users.
4. Record the transaction in the database.

- **Integration with Other Services**

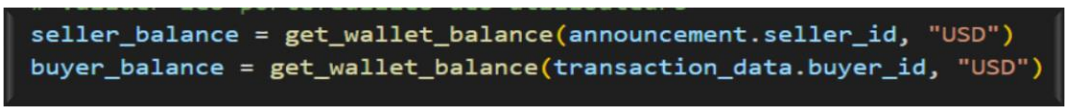
1. Price Service

- **Purpose**: Fetch the current market price for a specific asset (e.g., MCO2).
- **Usage in Payment Transactions**:
  - The get_price function fetches the latest price from the **Price Service**.
  - This price is used to calculate the total transaction cost

```
PRICE_SERVICE_URL = "http://127.0.0.1:8002"  # Adresse du service Price Service

def get_price(currency: str):
    """Récupère le prix actuel de la devise ou de l'actif depuis price_service."""
    if currency != "MCO2":
        raise ValueError("Price service only supports MCO2 transactions")
    url = f"{PRICE_SERVICE_URL}/prices/{currency}"
```

2.User Service

- **Purpose**: Validate and update wallet balances.
- **Usage in Transactions**:
  - Fetch wallet balances for the buyer, seller, or recipient.
  - Update wallet balances after successful transactions.



```
seller_balance = get_wallet_balance(announcement.seller_id, "USD")
buyer_balance = get_wallet_balance(transaction_data.buyer_id, "USD")
```

```
update_wallet_balance(
    wallet_id=transaction_data.buyer_id,
    new_balance=sender_balance - transaction_data.credit_amount,
    user_id=transaction_data.buyer_id,
    currency=transaction_data.currency
)
```

- **Transaction Validation**

  Fields Validation**:**

  - Ensure credit_amount > 0.
  - For PAYMENT transactions, announcement_id is required.
  - For VIREMENT transactions, recipient_id is required.

```
@root_validator
def check_transaction_fields(cls, values):
    credit_amount = values.get('credit_amount')
    if credit_amount <= 0:
        raise ValueError("credit_amount must be greater than 0")
    transaction_type = values.get('type')
    announcement_id = values.get('announcement_id')
    recipient_id = values.get('recipient_id')
```

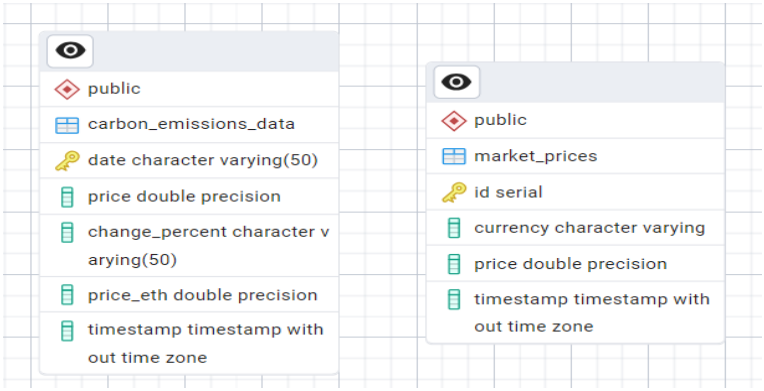
Example Workflow

Creating a Payment Transaction

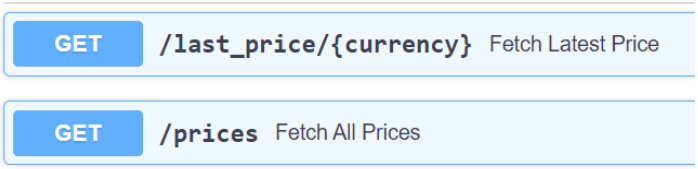
1. **Step 1**: Validate the announcement and its status.
2. **Step 2**: Fetch buyer and seller wallet balances.
3. **Step 3**: Fetch the current market price.
4. **Step 4**: Calculate the total cost and validate buyer's funds.
5. **Step 5**: Update the seller and buyer wallets.
6. **Step 6**: Record the transaction in the database.

4. Price Service

- **Purpose**: Tracks market prices for MCO2 tokens and related currencies.
- **Database**: price_service_db (PostgreSQL)



- **Endpoints**:



Data Scraping and Database Insertion

As part of this project, one of the key steps involves collecting historical data on carbon emissions, associated prices, and their conversion into ETH (Ethereum).

To achieve this, a **web scraping** solution has been implemented. The primary objective is to retrieve information from a web page, analyze it, and store it in a PostgreSQL database for further processing.

This section outlines the detailed steps involved in the implementation.

4.1. Scraping Objective*:*

The goal of the scraping process is to collect historical data on carbon emissions from the site below:

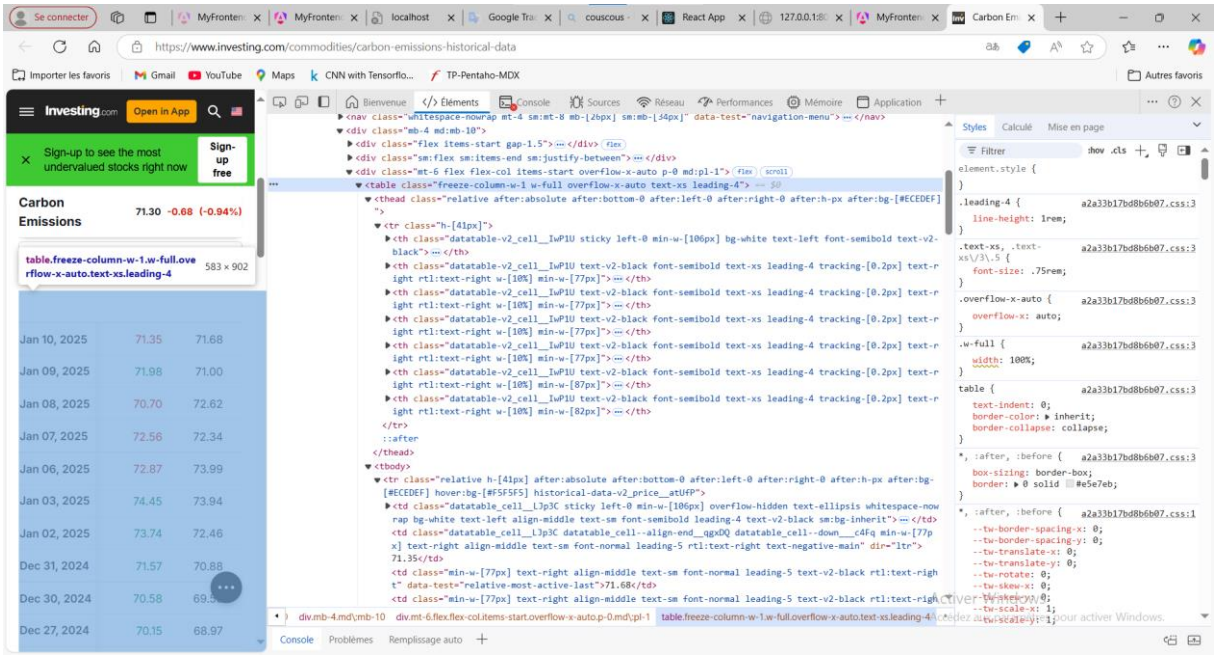
- **URL**: https://www.investing.com/commodities/carbon-emissions-historical-data

1. **Scrape historical carbon price data:**
    - Date.
    - Price in EURO.
    - Percentage change.
2. **Retrieve ETH historical price:**
    - Calculate the ratio between the carbon price in EURO and Ethereum's price in ETH.
3. **Store data in PostgreSQL:**
    - Ensure easy access to carbon price data.

4.2. Extracting Data from the Web Page

The scraping process begins with sending an HTTP request to the target URL using the **requests** library. Once the page is retrieved, it is parsed using **BeautifulSoup**, a Python library used for extracting data from the HTML structure of the page.

But before moving to extracting content we need to understand the form of the page, by inspecting it



Here we can see that the class name of table we need to scrap is ‘freeze-column-w-1’, that is going to simplify our work by specifying the specific content to check.

1. **Retrieving the HTML Page**:
    - A GET request is sent to the target URL.
    - If the request is successful (HTTP status code 200), the HTML page is retrieved for parsing.
2. **Parsing the HTML Content**:
    - Once the page is downloaded, **BeautifulSoup** is used to parse the HTML code and extract the table containing carbon emissions data. This table includes several columns, such as the date, price, and other relevant information.
    - The rows of the table are extracted, ignoring the first row (which contains the table header).

4.3. Retrieving ETH Price

Once the data has been extracted from the page, the next step is to retrieve the ETH price for each date. This is done by querying the **CoinGecko API**.

- Function **get_eth_price_on_date(date)** is defined to perform an API request for each date. The function sends a request to the CoinGecko API to obtain the historical price of ETH on a specific date.

- The ETH price in EUR is retrieved and used to calculate the value of carbon emissions in ETH.

4.4. Parallel Processing of Data

The scraping process can be time-consuming due to the large number of rows to be processed. To improve performance, the processing is done in **parallel** using the **concurrent.futures** library.

- **ThreadPoolExecutor** is used to execute the processing of each row concurrently. Each row of the table is sent to a separate thread to retrieve the data and insert it into the database without blocking the execution of other threads.

4.5. Data Validation and Insertion into the Database

Once the data has been extracted and the ETH price has been retrieved, it needs to be inserted into a PostgreSQL database. Before inserting each row, the code checks if data for a given date already exists to avoid duplicates.

- **PostgreSQL Database**: The database used is PostgreSQL, which contains a table **carbon_emissions_data** for storing the extracted data. This table has the columns bellow:
    - **date**: The date of carbon emissions.
    - **price**: The price of carbon emissions in EUR.
    - **change_percent**: The percentage change in the price.
    - **price_eth**: The price of carbon emissions converted into ETH (calculated by dividing the price by the ETH price).
    - **timestamp**: The timestamp of the insertion.
- **Insertion into the Database**:
    - For each row, the price of carbon emissions is divided by the ETH price to obtain the price in ETH.
    - The data is inserted into the **carbon_emissions_data** table using an SQL query. If an entry for a given date already exists in the database, the insertion is skipped.

4.6. Error Handling and API Rate Limits

The scraping process accounts for several potential issues that might interrupt the execution:

- **API Rate Limits**: If the CoinGecko API rate limit is reached (HTTP status code 429), the retrieval of the ETH price for a given date is skipped, and the function returns None.
- **Execution Errors**: If an error occurs during the scraping or database insertion process, error messages are displayed to help with troubleshooting and debugging.

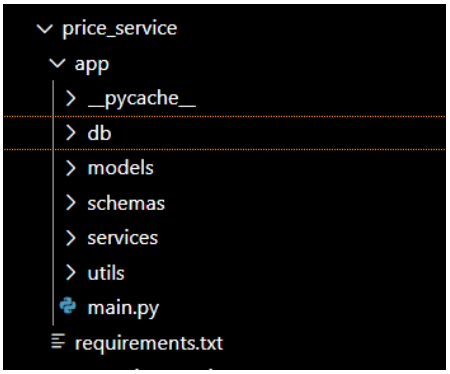
Résultats

1. **Scraped Data**: The data has been correctly extracted and inserted into PostgreSQL.
2. **Carbon/ETH Price Ratio**: Calculated for each date, with handling of missing prices.
3. **Performance**:
    - Multi-threading reduced execution time.
    - Local cache limited repetitive API calls.
4. **Scraped Data**: Data has been correctly extracted and inserted into PostgreSQL.
5. **Carbon/ETH Price Ratio**: A ratio was computed based on Ethereum's price on each date, providing additional insights.
6. **Cross-analysis**: By comparing carbon price trends and ETH prices, we observed an inverse correlation on certain days.

Limitations :

The Investing.com website occasionally closes the carbon market, which results in not being able to retrieve the current carbon price. However, we have implemented a strategy where, if today's data is unavailable, we use the price from the last time the market was open.

Integration in price_service:



In models, we created the models we are going to use; we give it a name market_prices.

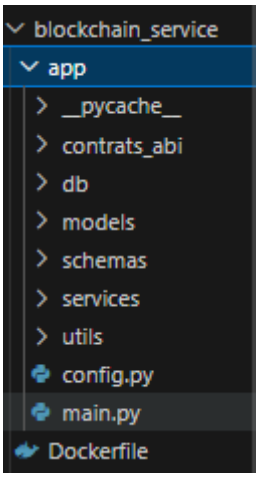
Utils: where we integrated our scraping code.

In schemas, we can find the general schema of our project.

Service: We defined methods that return the last price or all prices, but before it returns the data, it runs the method presented in utils, which scrapes data from the website mentioned.

Main: All endpoints are present, making it simple to use the application just in dependence on those endpoints.

5. Blockchain Service



- **Purpose**: Manages blockchain-based transactions and token-related activities.
- **Database**: None (interacts directly with the blockchain network).

The **Blockchain Service** is a microservice responsible for integrating the application with the Ethereum blockchain. It interacts with **smart contracts** to perform blockchain-based operations such as user registration, wallet linking, carbon credit payments, staking, and ERC-20 token management.

This service ensures secure and transparent transactions by leveraging blockchain's decentralized nature and Web3 technology.

Frameworks and Libraries Used

- **Web3.py**: Interface for interacting with the Ethereum blockchain.
- **FastAPI**: Framework for building RESTful APIs.
- **JWT (via jose)**: Authentication using tokens.
- **CORS Middleware**: Allows cross-origin requests for the API.
- **Smart Contract Integration**: ABI and deployed addresses are loaded dynamically

- **Data Model :**

Smart Contracts Integrated

1. **CarbonAccount**:
    - Manages user registration and wallet linking.
    - Functions:
        - registerUser(address user_address)
        - linkWallet(address user_address, address wallet_address)
        - creditCarbon(address user_address, uint256 amount)

```python
# 1. Fonctionnalités du contrat CarbonAccount
async def register_user(user_address):
    """Appeler la fonction `registerUser` dans le contrat CarbonAccount."""
    try:
        contract_function = carbon_account.functions.registerUser(user_address)
        receipt = await send_transaction(contract_function, USER_ADDRESS, PRIVATE_KEY)
        print(f"Utilisateur {user_address} enregistré avec succès, receipt: {receipt.transactionHash.hex()}")
        return receipt
    except Exception as e:
        print(f"Erreur lors de l'enregistrement de l'utilisateur : {e}")
        raise

async def link_wallet(user_address, wallet_address):
    """Associer un portefeuille à un utilisateur."""
    try:
        contract_function = carbon_account.functions.linkWallet(user_address, wallet_address)
        receipt = await send_transaction(contract_function, USER_ADDRESS, PRIVATE_KEY)
        print(f"Portefeuille {wallet_address} lié à l'utilisateur {user_address} : {receipt.transactionHash.hex()}")
        return receipt
    except Exception as e:
        print(f"Erreur lors de la liaison du portefeuille : {e}")
        raise

async def credit_carbon(user_address, amount):
    """Ajouter des crédits carbone à un utilisateur."""
    try:
        contract_function = carbon_account.functions.creditCarbon(user_address, amount)
        receipt = await send_transaction(contract_function, USER_ADDRESS, PRIVATE_KEY)
        print(f"Ajouté {amount} crédits carbone à {user_address} : {receipt.transactionHash.hex()}")
        return receipt
    except Exception as e:
        print(f"Erreur lors de l'ajout de crédits carbone : {e}")
        raise
```

2. **CarbonPayment**:
    - Handles carbon credit payments and minting.
    - Functions:
        - payForCarbonCredits(address to_address, uint256 amount)
        - convertAndMint(address user_address, uint256 fiat_amount)

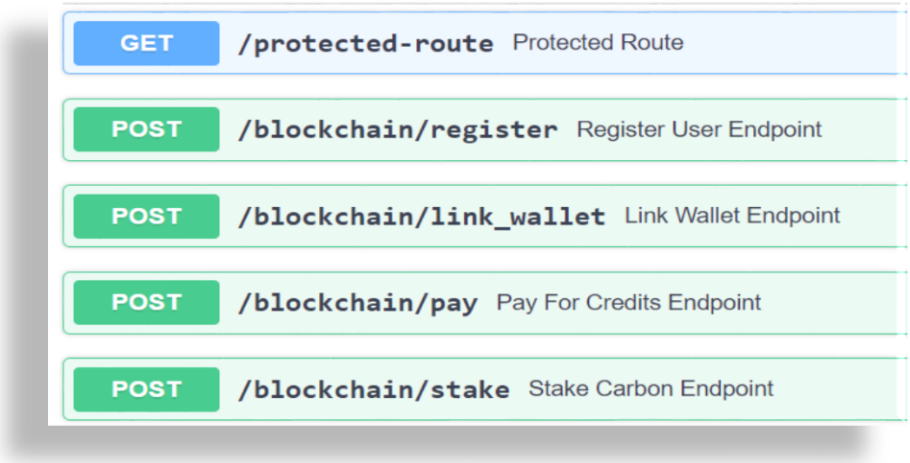
```python
# 2. Fonctionnalités du contrat CarbonPayment
async def pay_for_carbon_credits(to_address, amount):
    """Payer des crédits carbone."""
    try:
        eth_amount = web3.to_wei(amount, 'ether')
        contract_function = carbon_payment.functions.payForCarbonCredits(to_address, amount)
        receipt = await send_transaction(contract_function, USER_ADDRESS, PRIVATE_KEY, eth_amount)
        print(f"Payé {eth_amount} ETH pour {amount} crédits carbone : {receipt.transactionHash.hex()}")
        return receipt
    except Exception as e:
        print(f"Erreur lors du paiement de crédits carbone : {e}")
        raise

async def convert_and_mint(user_address, fiat_amount):
    """Convertir un montant en fiat et créer des crédits carbone."""
    try:
        contract_function = carbon_payment.functions.convertAndMint(user_address, fiat_amount)
        receipt = await send_transaction(contract_function, USER_ADDRESS, PRIVATE_KEY)
        print(f"Converti {fiat_amount} fiat en crédits carbone pour {user_address} : {receipt.transactionHash.hex()}")
        return receipt
    except Exception as e:
        print(f"Erreur lors de la conversion et de la création : {e}")
        raise
```

3. **CarbonStaking**:
    - Manages staking and unstaking of carbon credits.
    - Functions:
        - stake(uint256 amount)
        - unstake(uint256 amount)

```python
# 3. Fonctionnalités du contrat CarbonStaking
async def stake_carbon(amount):
    """Staker des crédits carbone."""
    try:
        contract_function = carbon_staking.functions.stake(amount)
        receipt = await send_transaction(contract_function, USER_ADDRESS, PRIVATE_KEY)
        print(f"Staké {amount} crédits carbone : {receipt.transactionHash.hex()}")
        return receipt
    except Exception as e:
        print(f"Erreur lors du staking : {e}")
        raise

async def unstake_carbon(amount):
    """Retirer des crédits carbone stakés."""
    try:
        contract_function = carbon_staking.functions.unstake(amount)
        receipt = await send_transaction(contract_function, USER_ADDRESS, PRIVATE_KEY)
        print(f"Retiré {amount} crédits carbone stakés : {receipt.transactionHash.hex()}")
        return receipt
    except Exception as e:
        print(f"Erreur lors du unstaking : {e}")
        raise
```

4. **CarbonToken**:
    - ERC-20 contract for token operations.
    - Functions:
        - approve(address spender, uint256 amount)
        - transfer(address to_address, uint256 amount)

```python
# 4. Fonctionnalités du contrat CarbonToken (ERC-20)
async def approve_token(spender_address, amount):
    """Approuver une adresse pour dépenser des tokens."""
    try:
        contract_function = carbon_token.functions.approve(spender_address, amount)
        receipt = await send_transaction(contract_function, USER_ADDRESS, PRIVATE_KEY)
        print(f"Approuvé {spender_address} pour dépenser {amount} tokens : {receipt.transactionHash.hex()}")
        return receipt
    except Exception as e:
        print(f"Erreur lors de l'approbation des tokens : {e}")
        raise

async def transfer_token(to_address, amount):
    """Transférer des tokens ERC-20."""
    try:
        contract_function = carbon_token.functions.transfer(to_address, amount)
        receipt = await send_transaction(contract_function, USER_ADDRESS, PRIVATE_KEY)
        print(f"Transféré {amount} tokens à {to_address} : {receipt.transactionHash.hex()}")
        return receipt
    except Exception as e:
        print(f"Erreur lors du transfert des tokens : {e}")
        raise
```



Relationship Between MetaMask and user_service

1. **Each user has one or more wallets in user_service**:
   - Example: Wallet(id=1, balance=100.0, currency='ETH', user_id=5)
2. **The linking process associates the MetaMask address**

   (e.g., 0xA09e...) with the user's ETH wallet in the user_service database.

MetaMask (Web3 Wallet)

1. **User Interaction**:
   - Users interact with MetaMask to sign blockchain transactions.
   - Each user has an Ethereum address (e.g., 0xA09e8aD094a059C12a6227ef8bE5d00527891304).
2. **Wallets in user_service_db**:
   - Stores wallet balances and types (USD, ETH, CARBON).
   - The wallets table holds off-chain information about wallet balances.
3. **Blockchain**:
   - Smart contracts (**CarbonAccount, CarbonPayment, CarbonStaking, CarbonToken**) handle on-chain operations:
     - User registration.
     - Wallet linking.
     - Carbon credit management.

- Staking and payments.

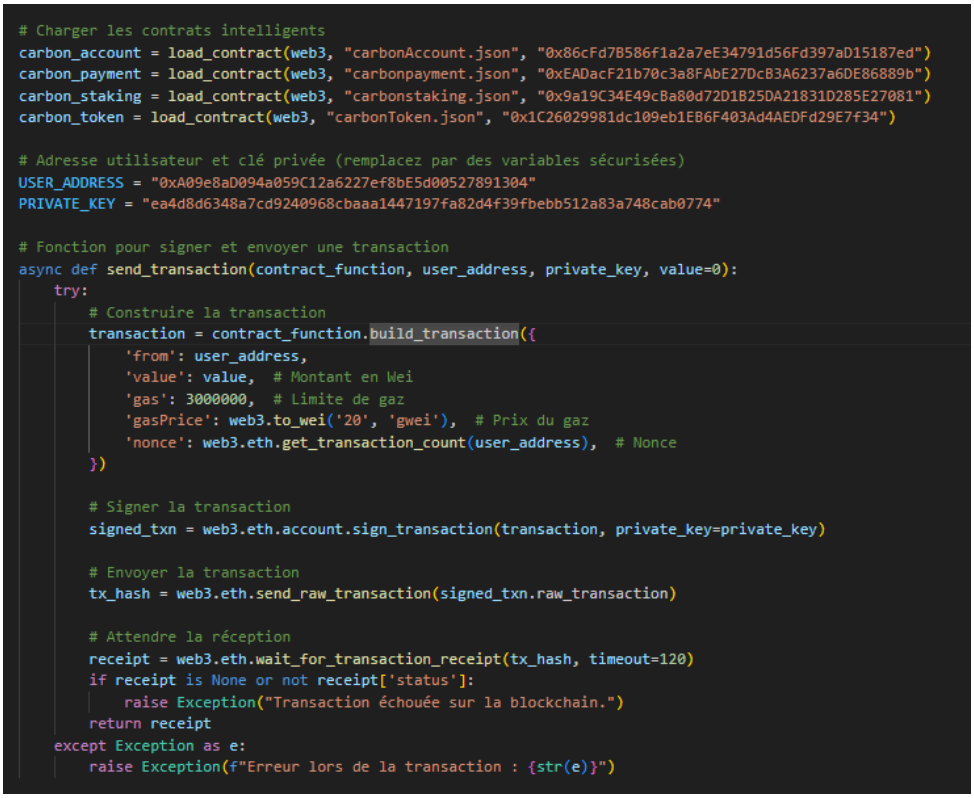
Logic of the Microservice

Step 1: Authentication with JWT (via user_service)

- Users authenticate through **user_service**.
- A **JWT token** is used to validate requests in **blockchain_service**.

Step 2: Interaction Between MetaMask and Blockchain

- Users use their **MetaMask address** to sign transactions.



```python
# Charger les contrats intelligents
carbon_account = load_contract(web3, "carbonAccount.json", "0x86cFd7B586f1a2a7eE34791d56Fd397aD15187ed")
carbon_payment = load_contract(web3, "carbonpayment.json", "0xEADacF21b70c3a8FAbE27DcB3A6237a6DE86889b")
carbon_staking = load_contract(web3, "carbonstaking.json", "0x9a19C34E49cBa80d72D1B25DA21831D285E27081")
carbon_token = load_contract(web3, "carbonToken.json", "0x1C26029981dc109eb1EB6F403Ad4AEDFd29E7f34")

# Adresse utilisateur et clé privée (remplacez par des variables sécurisées)
USER_ADDRESS = "0xA09e8aD094a059C12a6227ef8bE5d00527891304"
PRIVATE_KEY = "ea4d8d6348a7cd9240968cbaaa1447197fa82d4f39fbebb512a83a748cab0774"

# Fonction pour signer et envoyer une transaction
async def send_transaction(contract_function, user_address, private_key, value=0):
    try:
        # Construire la transaction
        transaction = contract_function.build_transaction({
            'from': user_address,
            'value': value,  # Montant en Wei
            'gas': 3000000,  # Limite de gaz
            'gasPrice': web3.to_wei('20', 'gwei'),  # Prix du gaz
            'nonce': web3.eth.get_transaction_count(user_address),  # Nonce
        })

        # Signer la transaction
        signed_txn = web3.eth.account.sign_transaction(transaction, private_key=private_key)

        # Envoyer la transaction
        tx_hash = web3.eth.send_raw_transaction(signed_txn.raw_transaction)

        # Attendre la réception
        receipt = web3.eth.wait_for_transaction_receipt(tx_hash, timeout=120)
        if receipt is None or not receipt['status']:
            raise Exception("Transaction échouée sur la blockchain.")
        return receipt
    except Exception as e:
        raise Exception(f"Erreur lors de la transaction : {str(e)}")
```

- The **blockchain_service** sends transactions to the blockchain via **Web3**.

Step 3: Synchronization Between Blockchain and Wallets (Off-Chain)

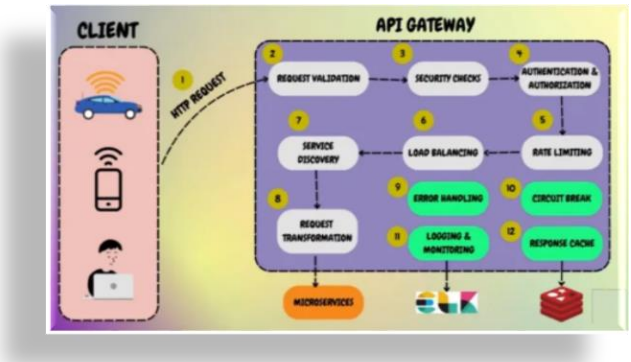
- Blockchain transactions are reflected in the user_service database (user_service_db).
- Example: After an ETH payment via MetaMask, the **CARBON balance** in user_service_db is

LIMITATIONS :

Unfortunately, despite our efforts, we were unable to complete a blockchain transaction during our implementation. While we successfully connected to wallets using MetaMask and established communication with the Ethereum network, the transaction process itself did not succeed. We are unsure of the exact cause at this stage, and it may involve issues such as contract deployment, incorrect transaction configuration, or network-related problems. We are continuing to investigate the issue and will update you once we identify the root cause. Thank you for your understanding.

6. Notification Service

- **Purpose**: Sends notifications to users regarding transaction updates and blockchain events.
- **Database**: None (stateless, uses Kafka for event-driven architecture).
- **Responsibilities**:



- Notify users of transaction and blockchain updates via email or messaging.
- Consume Kafka topics like transaction_events and blockchain_events.

Messages synchrones et asynchrones

- **Synchrones (REST API)** :
  - Validation des utilisateurs (GET /users/{user_id}).
  - Vérification ou mise à jour des portefeuilles (PUT/users/{user_id}/wallets).
- **Asynchrones (Kafka)** :
  - Mise à jour des portefeuilles après une transaction.
  - Notifications sur l'état des transactions.

Steps to Deploy and Run the Platform

1. **Install Required Tools**:
   - Docker and Docker Compose
   - PostgreSQL
   - Python (with venv for virtual environments)
2. **Setup Databases**:
   - Create databases: user_service_db, transaction_service_db, price_service_db.
   - Apply migrations for each microservice:

     Cmd : `alembic upgrade head`

3. **Build and Run Microservices**:
   - Use Docker Compose to build and start all services:

     ```
     docker-compose up --build
     ```

4. **Start Kafka and Zookeeper** (for asynchronous messaging):
   - Start Zookeeper:

     ```
     zookeeper-server-start.sh config/zookeeper.properties
     ```

   - Start Kafka:

     ```
     kafka-server-start.sh config/server.properties
     ```

5. **Verify Services**:
   - Use tools like Postman to test the endpoints of each microservice.
   - Ensure API Gateway routes requests correctly.

Key Processes

1. User Registration

1. User sends a POST /users request to the User Service.
2. User Service validates and stores user data in user_service_db.

2. Wallet Management

1. User retrieves wallet details via GET /users/{id}/wallets.

2. Wallet balance updates through PUT /users/{id}/wallets/{wallet_id}.

3. Transaction Workflow

1. Buyer sends a POST /transactions request to the Transaction Service.
2. Transaction Service validates wallets (via REST API to User Service).
3. Transaction details are stored in transaction_service_db.
4. Kafka produces an event to notify the Blockchain Service if applicable.

4. Market Price Updates

1. Price Service scrapes data from sources like CoinGecko.
2. Prices are stored in price_service_db for historical tracking.

5. Blockchain Interaction

1. Transaction Service sends data to Blockchain Service via Kafka.
2. Blockchain Service processes the transaction on the blockchain network.

6. Notifications

1. Transaction Service or Blockchain Service produces events to Kafka.
2. Notification Service consumes events and sends notifications via email or messaging.

Future Enhancements

- Implement real-time price updates via WebSocket.
- Add user notifications for completed transactions.
- Expand currency support for transactions

Linkage between Backend and FrontEnd :

The important step is to make the application more applicable and usable. For that, we needed to link the frontend and backend.

We started by linking the first server user server and frontend so that the client can authenticate and sign up if they do not have an account yet.

As we are testing the backend with Postman, we have already created endpoints. For example, in the code extract below, we can see endpoints that allow us to create an account or authenticate an existing account.

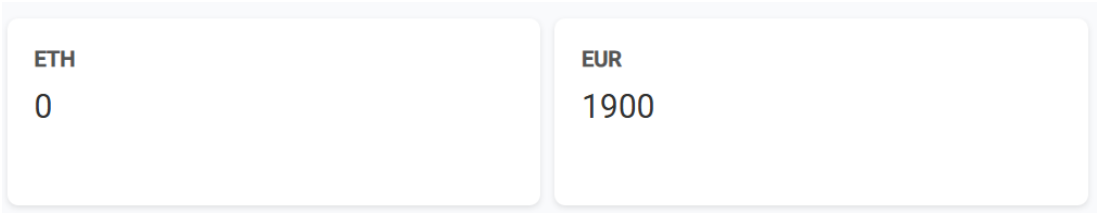
```python
@app.post("/token")
def login(form_data: OAuth2PasswordRequestForm = Depends(), db: Session = Depends(get_db)):
    user = authenticate_user(db, form_data.username, form_data.password)
    if not user:
        raise HTTPException(status_code=401, detail="Invalid credentials")
    access_token = create_access_token(data={"sub": user.username})
    return {"access_token": access_token, "token_type": "bearer","user_id":user.id}

@app.post("/users/", response_model=UserResponse)
def create_user_endpoint(user: UserCreate, db: Session = Depends(get_db)):
    """Créer un utilisateur et ses wallets par défaut."""
    new_user = create_user(db, user)
    # Création automatique des wallets pour l'utilisateur (EURO, ETH, CARBON)
    return new_user
```

In the frontend, we created a service with the command ‘ng generate service api’. In that service, we have been using endpoints to get the data we need.

In the service API, we defined URLs to endpoints of the backend, then created methods to handle the data retrieval.

```typescript
private apiUrl = 'http://localhost:8004/token';
login(username: string, password: string): Observable<{ access_token: string; user_id: number }> {
  const body = new URLSearchParams();
  body.set('username', username);
  body.set('password', password);
  const headers = new HttpHeaders().set('Content-Type', 'application/x-www-form-urlencoded');
  return this.http.post<{ access_token: string; user_id: number }>(this.apiUrl, body.toString(), { headers });
}
 signup(user: { username: string, email: string, password: string }): Observable<any> {
  return this.http.post('http://localhost:8004/users', user);}
```

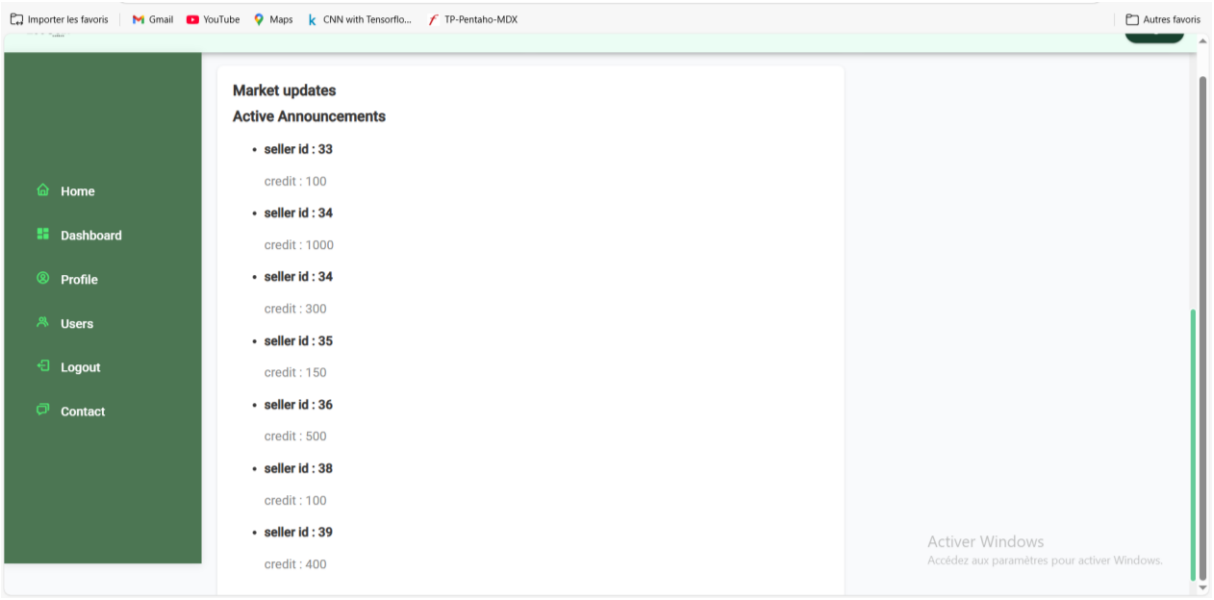
As we linked the user service with frontend, we became able to get wallets balance for the user logged in.



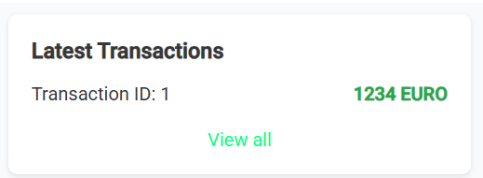
Then with the same way, we linked second service, price service; than linkage was important for our application view, because it appears in the dashboard that appears first after authentication

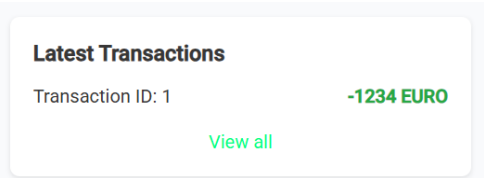
This chart provides prices from 2-12-2024 the date of first price in our database, and it give the permission to filter by ethereum or euro.

After second service we tried to link the third service, which is transactions, we started by getting announcements from table announcement and show them for any user authenticate.



In addition, we succeeds to get transactions effected by the user logged in, with respect that that user pay or get the amount of money.





Remarque: Because of time short, we could not link all the application

Data Analysis :

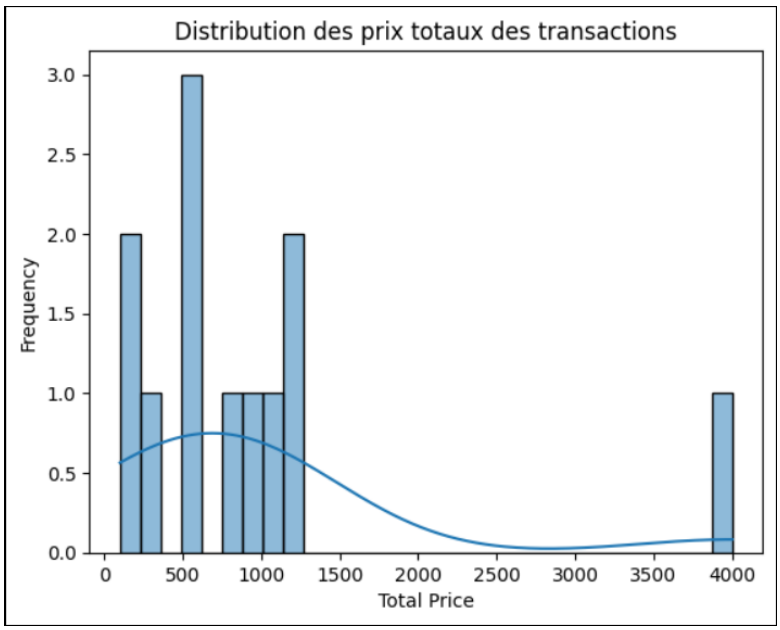
Objectifs de l'Analyse des Données

- Analyze transaction volumes: Calculate the total transaction volumes on the platform from both traditional systems and the blockchain.
- Analyze cryptocurrency movements: Study the evolution of exchanged tokens and their impact on the system.
- Provide interactive financial reports: Use visualization tools to provide real-time insights into financial flows.

Analysis Results

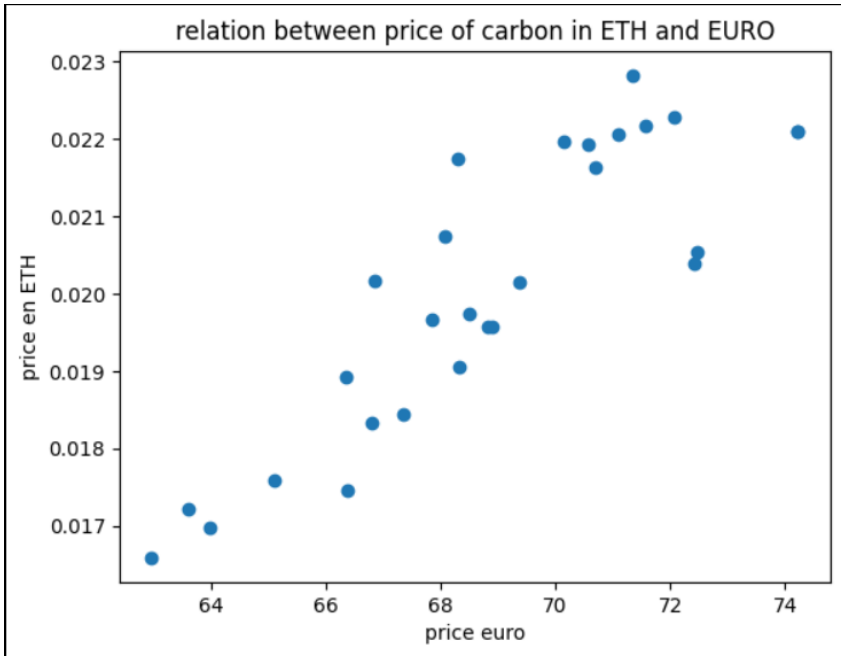
Total Transaction Volumes:

This visualization helps understand that most transactions' total prices are under 1200.



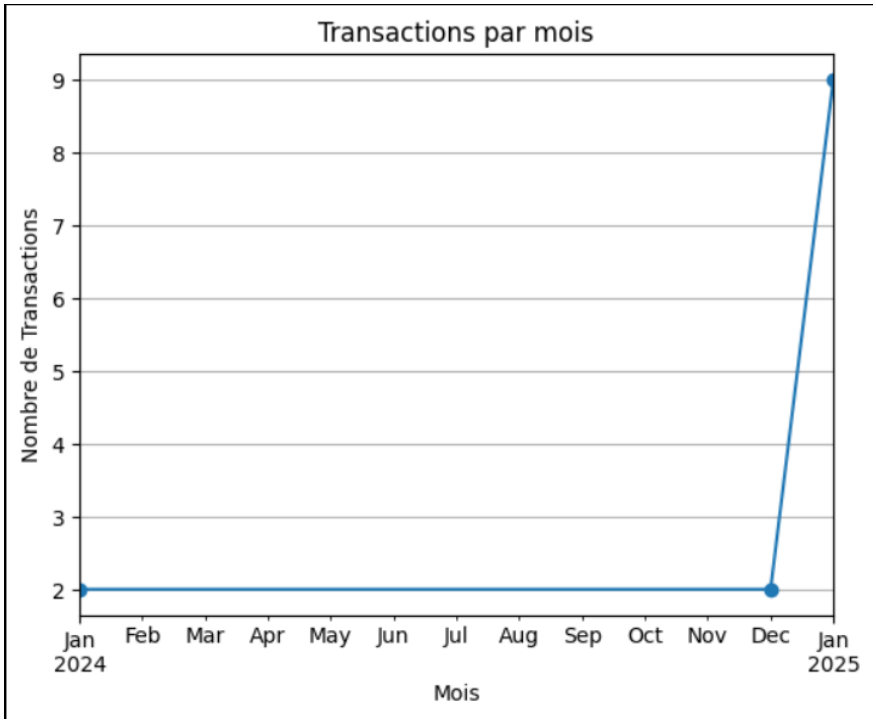
Relation between ETH and EUR Prices

This visualization shows that the price in Ethereum and EUR are correlated, indicating that the ratio between them changes similarly even as the values fluctuate.



Transaction Changes by Month

- From this graph, we observe that transactions in January increased from 2 to 9 compared to the previous month.



Usage of grafana :

Grafana is an open-source platform for monitoring, visualization, and analytics. It allows users to create dynamic dashboards and analyze data from various sources in real time. With its flexible query options and interactive graphs, Grafana is widely used for system monitoring, performance analysis, and business intelligence.

Key features of Grafana include:

- Support for multiple data sources like Prometheus, InfluxDB, MySQL, and more.
- Customizable and interactive dashboards.
- Alerting functionality to notify users of critical changes.
- An intuitive and user-friendly interface.

In this project, Grafana is used to create a dashboard based on data stored in a PostgreSQL database. The dashboard provides visual insights into the data, making it easier to analyze trends and extract valuable information.

Key aspects of using Grafana in this project:

- Data source: PostgreSQL database.
- Purpose: Visualization of data through interactive and customizable dashboards.
- Integration: Grafana is used as a standalone tool and is not directly linked with other parts of the project.

This dashboard presents a detailed analysis of the carbon market system, providing insights into users, transactions, wallet balances, and carbon price trends.

Key Metrics: These metrics provide a quick overview of the system's overall activity and engagement levels.

1. **Total Number of Users:**
    - A total of 19 users are actively registered in the system.
2. **Total Number of Admins:**
    - The system currently does not have any registered admins (0 admins).
3. **Active Announcements:**
    - There are 7 announcements currently active, indicating ongoing activities or market updates.
4. **Transaction Summary:**
    - **Total Transactions:** 13 transactions have been processed.
    - **Total Value:** The combined value of all transactions is 11503.
    - **Average Value:** The average value of a single transaction is 959.

Carbon Prices Analysis:

1. **EUR Carbon Prices:**
    - The graph displays fluctuations in carbon prices (in EUR) over time, ranging between **63.0** and **74.2**.
    - The trend helps to monitor market behavior and evaluate price volatility.
2. **ETH Carbon Prices:**
    - The ETH carbon prices range from **0.0166** to **0.0228**, showing variability over the observed timeframe.
    - This visualization is crucial for understanding the performance of ETH in the carbon market.

Wallet Balance Analysis:

1. **Average Wallet Balance by Currency:**
    - Wallet balances are categorized by currency (CARBON, EURO, ETH) with the following averages:
        - **CARBON:** 375

- **EURO:** 675
- **ETH:** 751

- The chart highlights the distribution of wallet balances, helping identify which currencies are predominantly held by users.

Transaction Analysis:

1. **Effected Transactions:**
   - Transactions are categorized into two types: **Payment** and **Virement**.
     - **Payment:** Includes 7 transactions with a total value of **3935**.
     - **Virement:** Includes 6 transactions with a total value of **7568**.
   - This analysis reveals that "Virement" transactions account for a higher total value compared to "Payment."

1. Top Sellers by Revenue

- The table ranks sellers by their total revenue (€) generated from transactions:
  - **Seller 46**: €4000 (highest revenue)
  - **Seller 47**: €2468
  - **Seller 34**: €1075
  - **Seller 35**: €1030

This visualization highlights the most profitable sellers, offering insights into which sellers are contributing the most to the market's success.

2. Transaction Status Breakdown

- The pie chart shows the distribution of transactions by status:
    - **Completed Transactions**: 8 (62% of total)
    - **Pending Transactions**: 5 (38% of total)

This breakdown emphasizes the proportion of successful transactions while identifying those that are still pending resolution.

3. Transaction Count and Total Value by Currency

- A bar chart compares the number of transactions and their total value across currencies:
    - **EURO**: 4 transactions totaling €7498
    - **CARBON**: 6 transactions totaling €2675
    - **ETH**: 3 transactions totaling €1330

This visualization reveals that the EURO currency dominates in terms of transaction value, whereas CARBON has the highest transaction count.

4. Buyer Activity Analysis

- A dual-axis chart analyzes buyers based on their:
    - **Total Spend (€):** Reflects how much buyers have spent in total.
    - **Number of Transactions:** Indicates how many transactions each buyer has completed.

This chart helps in understanding buyer behavior, identifying high-value buyers, and assessing transaction activity trends.

5. Wallet Distribution by Currency

- The bar chart displays wallet balances and the number of users holding each currency:
    - **EURO Wallets**: 16 users with a total balance of €10800
    - **ETH Wallets**: 16 users with a total balance of €12017
    - **CARBON Wallets**: 16 users with a total balance of €6000

This visualization provides insights into the distribution of wallet balances and highlights ETH as the most held currency in terms of total value.

Conclusion :

1. **Revenue Leaders**: Seller 46 leads in revenue generation, making them a key contributor to the market.
2. **Transaction Status**: While most transactions are completed, 38% are still pending, indicating room for operational improvement.
3. **Currency Trends**: EURO is the dominant currency for transaction value, while CARBON has a higher transaction count.
4. **Buyer Patterns**: Certain buyers demonstrate significant spending and transaction activity, suggesting high engagement.
5. **Wallet Balances**: ETH currency wallets hold the highest total value, indicating a strong preference among users.