



**POLITECHNIKA
RZESZOWSKA**
im. IGNACEGO ŁUKASIEWICZA



**WYDZIAŁ
MATEMATYKI
I FIZYKI STOSOWANEJ**
POLITECHNIKI RZESZOWSKIEJ

Złotek Sebastian

Aplikacja do przetwarzania transakcji finansowych z
użyciem Apache Kafka

Usługi Sieciowe w Biznesie

Rzeszów, 2023

Spis treści

Wstęp	4
1. Wprowadzenie do kluczowych terminów.....	5
1.1. Apache Kafka.....	5
1.2. Zastosowania Apache Kafki	6
1.3. Apache ZooKeeper.....	6
1.4. Broker.....	7
1.5. Topic	7
1.6. Docker	8
2. Konfiguracja Dockera.....	9
3. Pliki tekstowe requirements.....	12
4. Wykorzystanie Apache Kafka w języku Python	12
4.1. Potrzebne biblioteki i moduły	12
4.2. Tworzenie fałszywych informacji w generatorze	14
4.3. Modelowanie danych finansowych.....	15
4.4. Wykrywanie fałszywej transakcji	16
5. Uruchamianie dockera za pomocą wiersza poleceń lub terminala	18
6. Podsumowanie	22
7. Spis rysunków.....	22
8. Użyte materiały.....	23

Wstęp

Niniejszy projekt skupia się na stworzeniu zaawansowanej aplikacji do przetwarzania transakcji finansowych, która wykorzystuje popularne narzędzia takie jak Apache Kafka, Python i Apache ZooKeeper. Aplikacja ma na celu dostarczenie niezawodnego i skalowalnego rozwiązania do przesyłania i przetwarzania danych transakcyjnych w czasie rzeczywistym.

W dzisiejszych czasach, wraz z dynamicznym rozwojem rynków finansowych, przetwarzanie transakcji staje się kluczowym elementem dla instytucji finansowych i przedsiębiorstw. Efektywne zarządzanie i przetwarzanie danych transakcyjnych w czasie rzeczywistym jest niezwykle ważne dla zapewnienia płynności operacji finansowych oraz monitorowania ryzyka.

Celem tego projektu jest zatem połączenie tych trzech narzędzi - Apache Kafka, Pythona i Apache ZooKeeper - w celu stworzenia kompleksowej aplikacji do przetwarzania transakcji finansowych. Dodatkowo, w ramach tego projektu aplikacja zostanie skonfigurowana do działania w środowisku kontenerowym przy użyciu narzędzia Docker. Docker zapewnia izolację i skalowalność, umożliwiając nam łatwe tworzenie, wdrażanie i uruchamianie aplikacji wewnątrz kontenerów. Dzięki temu możliwe będzie łatwe zarządzanie infrastrukturą i wdrożenie aplikacji na różnych środowiskach.

W kontekście komunikacji między aplikacją a Apache Kafka, zostanie utworzona dedykowana sieć (kafka-network) w środowisku Docker. Sieć ta umożliwi komunikację między różnymi kontenerami, co jest istotne w przypadku aplikacji wykorzystującej Kafka do przesyłania danych transakcyjnych. Dzięki temu aplikacja będzie mogła bezproblemowo korzystać z funkcjonalności Kafka wewnątrz środowiska kontenerowego.

Wprowadzenie Dockera i kafka-network do tego projektu dodaje elastyczności, skalowalności i izolacji, co przyczynia się do tworzenia stabilnej i wydajnej aplikacji do przetwarzania transakcji finansowych. Pozwala to również na łatwą replikację i zarządzanie aplikacją w środowiskach produkcyjnych.

1. Wprowadzenie do kluczowych terminów

1.1. Apache Kafka

Apache Kafka to otwarte oprogramowanie, które służy do przesyłania strumieniowego (streaming) danych w czasie rzeczywistym. Jest ono zaprojektowane w celu umożliwienia efektywnego i niezawodnego przesyłania dużej ilości danych między różnymi aplikacjami lub komponentami systemów. Kafka jest często wykorzystywana w kontekście przetwarzania strumieniowego, analizy danych czasu rzeczywistego, logów zdarzeń oraz budowania architektur mikroserwisów.

Główne zalety Apache Kafka to:

1. Skalowalność - może obsługiwać bardzo duże ilości danych i łatwo skalować się w górę, dostosowując się do rosnących potrzeb biznesowych.
2. Wydajność - osiąga wysoką wydajność dzięki swojej architekturze opartej na zapisie na dysku oraz możliwości równoczesnej obsługi wielu producentów i konsumentów.
3. Trwałość danych - przechowuje dane w sposób trwały na dysku, co zapewnia niezawodność i możliwość odtworzenia danych w przypadku awarii.
4. Zachowanie kolejności - gwarantuje zachowanie kolejności przesyłanych danych w ramach poszczególnych partycji, co jest kluczowe w przypadku przetwarzania strumieniowego.
5. Przetwarzanie w czasie rzeczywistym - umożliwia przetwarzanie danych w czasie rzeczywistym, co jest niezwykle ważne w przypadku aplikacji finansowych, które wymagają natychmiastowej reakcji na zmiany rynkowe.

Kafka składa się z kilku głównych komponentów, takich jak producenci (tworzący strumień danych) i konsumenci (odbierający strumień danych), a także tematy (kategorie, do których są przesyłane dane) i partycje (fragmenty tematów, które są replikowane i przetwarzane równolegle).

1.2. Zastosowania Apache Kafki

Monitorowanie aktywności - używana do monitorowania aktywności. Aktywność może dotyczyć witryny internetowej lub czujników i urządzeń fizycznych. Producent może publikować surowe dane pochodzące z różnych źródeł, które później można wykorzystać do identyfikacji trendów i wzorców.

Przesyłanie wiadomości - wykorzystana jako pośrednik wiadomości między usługami. Jeśli wdrażasz architekturę mikroservisów, możesz mieć mikroservis jako producenta i inny jako konsumenta. Na przykład: jeden mikroservis może być odpowiedzialny za tworzenie nowych kont, a inny za wysyłanie e-maili do użytkowników w sprawie utworzenia konta.

Agregacja dzienników - zbieranie dzienników z różnych systemów i przechowywania ich w centralnym systemie w celu dalszego przetwarzania.

ETL - strumieniowanie niemal w czasie rzeczywistym, co pozwala na stworzenie procesu Extract, Transform, Load w zależności od potrzeb.

Baza danych - na podstawie wcześniej wspomnianych aspektów można powiedzieć, że Kafka działa również jako baza danych. Nie jest to typowa baza danych z funkcją zapytywania danych według potrzeb, ale można przechowywać dane w Kafce tak długo, jak chcesz, bez ich konsumowania.

1.3. Apache ZooKeeper

Apache ZooKeeper to otwarte oprogramowanie, które pełni rolę centralnego systemu do zarządzania i monitorowania rozproszonych aplikacji. ZooKeeper zostało stworzone w celu zapewnienia koordynacji, synchronizacji i zarządzania danymi w systemach rozproszonych.

Główne funkcje i cechy Apache ZooKeeper to:

1. Zarządzanie zespołem - ZooKeeper umożliwia zarządzanie zespołem aplikacji rozproszonych poprzez utrzymanie hierarchicznego drzewa węzłów. Każdy węzeł w drzewie może przechowywać pewne dane, zwane zorientowanymi na daną aplikację
2. Synchronizacja procesów - mechanizmy synchronizacji procesów w systemach rozproszonych. Przykładem może być blokada, która pozwala na sekwencyjne wykonywanie operacji przez różne węzły.

3. Monitoring i powiadomienia - dostarcza funkcje monitorowania i powiadamiania, które umożliwiają aplikacjom śledzenie zmian w drzewie węzłów. W ten sposób aplikacje mogą reagować na zmiany w systemie w czasie rzeczywistym.
4. Bezpieczeństwo - takie jak uwierzytelnianie i autoryzacja, aby chronić dane i zasoby systemu przed nieuprawnionym dostępem.
5. Wydajność i skalowalność - zaprojektowany w taki sposób, aby zapewnić wysoką wydajność i skalowalność w systemach rozproszonych. Może obsługiwać duże ilości węzłów i dane, utrzymując jednocześnie wysoką wydajność operacji.

1.4. Broker

Broker w kontekście Apache Kafka to serwer, który działa jako pośrednik w przesyłaniu wiadomości między producentami (nadawcami) a konsumentami (odbiorcami). Odpowiada za przechowywanie i zarządzanie tematami, do których producenci wysyłają wiadomości, oraz umożliwia konsumentom odczytanie tych wiadomości. Brokerzy są kluczowymi komponentami w architekturze Kafka, umożliwiającymi skalowalność, replikację danych i niezawodną komunikację w czasie rzeczywistym.

1.5. Topic

W Apache Kafka, topic (temat) jest podstawową jednostką organizacyjną, która służy do kategoryzowania i przechowywania wiadomości. Temat jest nazwą identyfikującą logiczny strumień danych, do którego producenci wysyłają wiadomości, a konsumenci je odbierają.

Temat można rozumieć jako nazwaną kolejność wiadomości, gdzie każda wiadomość ma przypisaną wartość klucza (opcjonalnie) oraz dane. Producent wysyła wiadomości do konkretnego tematu, a Kafka przechowuje te wiadomości w poszczególnych partycjach. Każda partycja tematu jest uporządkowanym strumieniem danych, a Kafka utrzymuje i zarządza replikacją partycji w celu zapewnienia niezawodności i odporności na awarie.

Tematy w Kafka są elastyczne i można nimi zarządzać, takie jak tworzenie, usuwanie, konfigurowanie i skalowanie. Mogą być wykorzystywane do różnych zastosowań, takich jak strumieniowanie danych, przesyłanie wiadomości, logowanie zdarzeń i wiele innych.

Dzięki tematowi Kafka zapewnia wysoką wydajność i skalowalność w obszarze przetwarzania strumieniowego i komunikacji między aplikacjami w czasie rzeczywistym.

1.6. Docker

Docker to otwarta platforma do wirtualizacji kontenerowej, która umożliwia pakowanie, wdrażanie i uruchamianie aplikacji oraz ich zależności w izolowanych środowiskach zwanych kontenerami.

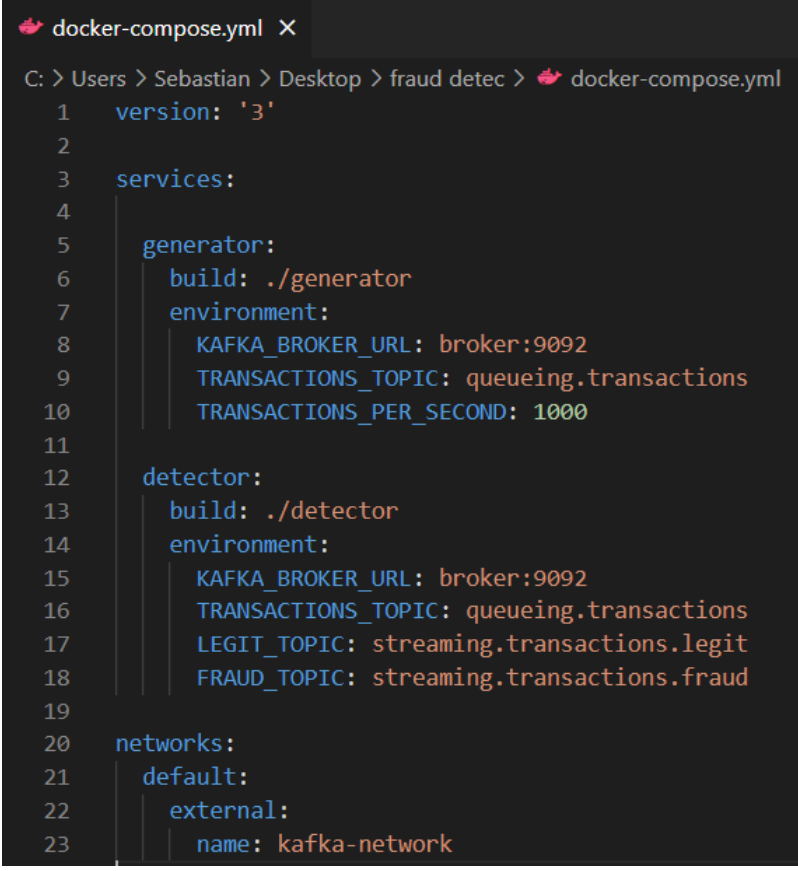
Kontenery Docker są lekkie, przenośne i niezależne od systemu operacyjnego, co oznacza, że aplikacje mogą być uruchamiane w dowolnym środowisku, które ma zainstalowany Docker. Kontenery zapewniają izolację zasobów, takich jak pamięć, procesor czy sieć, co umożliwia uruchomienie wielu aplikacji na jednym fizycznym lub wirtualnym serwerze bez konfliktów i interferencji.

Docker wykorzystuje technologię kontenerów opartą na jądrze systemu operacyjnego, co sprawia, że kontenery są bardziej wydajne i mają mniejsze wymagania sprzętowe w porównaniu do tradycyjnych wirtualnych maszyn. Dzięki temu, Docker jest popularnym narzędziem w środowiskach deweloperskich i produkcyjnych, umożliwiając łatwe tworzenie, testowanie i dostarczanie aplikacji w izolowanych i niezawodnych kontenerach.

Dodatkowo, Docker dostarcza narzędzia i funkcje zarządzania kontenerami, takie jak Docker Compose, które umożliwiają definiowanie i uruchamianie wielu kontenerów jako część kompleksowej aplikacji. Pozwala to na łatwe skonfigurowanie i zarządzanie wieloma usługami oraz tworzenie złożonych środowisk deweloperskich i produkcyjnych.

Docker stał się popularnym narzędziem w branży IT, przyspieszając proces wdrażania aplikacji, zwiększając skalowalność i ułatwiając zarządzanie infrastrukturą. Dzięki Dockerowi, aplikacje mogą być uruchamiane w izolowanych kontenerach, co sprawia, że są bardziej niezawodne, elastyczne i łatwiejsze do przenoszenia między różnymi środowiskami.

2. Konfiguracja Dockera



```
1 version: '3'
2
3 services:
4   generator:
5     build: ./generator
6     environment:
7       KAFKA_BROKER_URL: broker:9092
8       TRANSACTIONS_TOPIC: queueing.transactions
9       TRANSACTIONS_PER_SECOND: 1000
10
11   detector:
12     build: ./detector
13     environment:
14       KAFKA_BROKER_URL: broker:9092
15       TRANSACTIONS_TOPIC: queueing.transactions
16       LEGIT_TOPIC: streaming.transactions.legit
17       FRAUD_TOPIC: streaming.transactions.fraud
18
19 networks:
20   default:
21     external:
22       name: kafka-network
```

Rysunek 1 Konfiguracja docker-compose.yml

Ten kod to plik konfiguracyjny dla Docker Compose. Definiuje dwie usługi - "generator" i "detector", które będą uruchomione w środowisku Docker. Każda usługa ma swoje zmienne środowiskowe i korzysta z zdefiniowanej sieci "kafka-network". Usługa "generator" generuje transakcje finansowe, a usługa "detector" analizuje te transakcje w celu wykrywania oszustw.

```

1  version: '3'
2
3  services:
4
5      zookeeper:
6          image: confluentinc/cp-zookeeper:latest
7          environment:
8              ZOOKEEPER_CLIENT_PORT: 2181
9              ZOOKEEPER_TICK_TIME: 2000
10
11     broker:
12         image: confluentinc/cp-kafka:latest
13         depends_on:
14             - zookeeper
15         environment:
16             KAFKA_BROKER_ID: 1
17             KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
18             KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://broker:9092
19             KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
20
21     networks:
22         default:
23             external:
24                 name: kafka-network
25

```

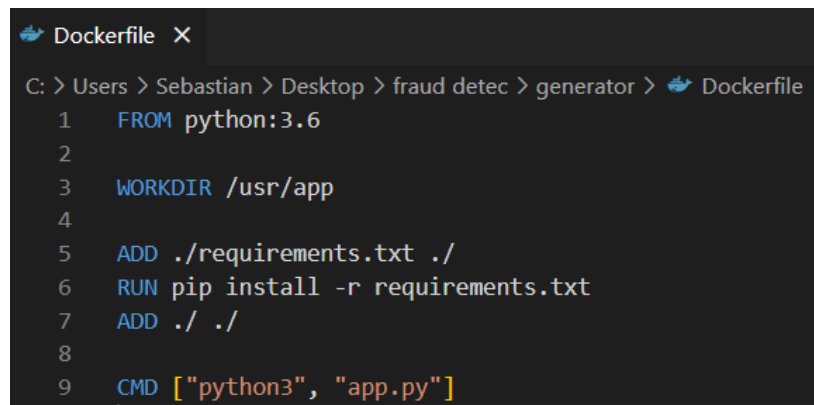
Rysunek 2 Plik konfiguracyjny docker-compose.kafka.yml

Plik konfiguracyjny w formacie YAML dla narzędzia Docker Compose, ma za zadanie opisywać zestaw usług, które mają być uruchomione w środowisku Docker. Wersja Docker Compose jest określona jako '3', co oznacza korzystanie z trzeciej wersji składni i funkcji Compose.

Kod definiuje dwie usługi: "zookeeper" i "broker". Każda usługa korzysta z obrazu Docker: "confluentinc/cp-zookeeper:latest" dla ZooKeeper i "confluentinc/cp-kafka:latest" dla brokera Kafka. Usługa "zookeeper" ma zdefiniowane zmienne środowiskowe, takie jak ZOOKEEPER_CLIENT_PORT i ZOOKEEPER_TICK_TIME.

Usługa "broker" zależy od usługi "zookeeper" (określonej w sekcji depends_on) i ma również zdefiniowane zmienne środowiskowe, zmienne te dotyczą konfiguracji brokera.

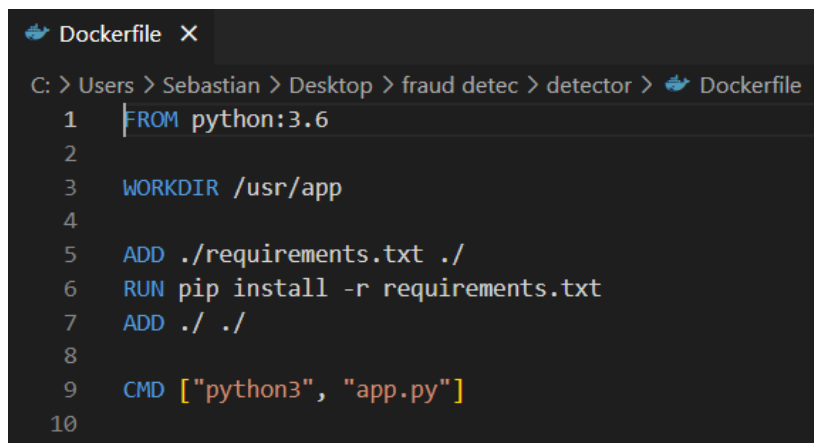
Następnie, definiuje sieć o nazwie "default" i określa, że jest to zewnętrzna sieć o nazwie "kafka-network". To umożliwia usługom dostęp do tej sieci i zbudowanie oraz uruchomienie dwóch usług - ZooKeeper i broker Kafka - w środowisku Docker, przy użyciu obrazów dostarczonych przez Confluent.



```
Dockerfile X
C: > Users > Sebastian > Desktop > fraud detec > generator > Dockerfile
1 FROM python:3.6
2
3 WORKDIR /usr/app
4
5 ADD ./requirements.txt ./
6 RUN pip install -r requirements.txt
7 ADD ./ ./
8
9 CMD ["python3", "app.py"]
```

Rysunek 3 Konfiguracja dockera dla generatora

Ten plik Dockerfile buduje obraz kontenera dla aplikacji Python. Na podstawie obrazu Pythona 3.6, kopiowane są pliki aplikacji i zależności z lokalnego środowiska. Następnie, zainstalowane są zależności z pliku "requirements.txt", a ostatecznie uruchamiana jest aplikacja poprzez wykonanie pliku "app.py".

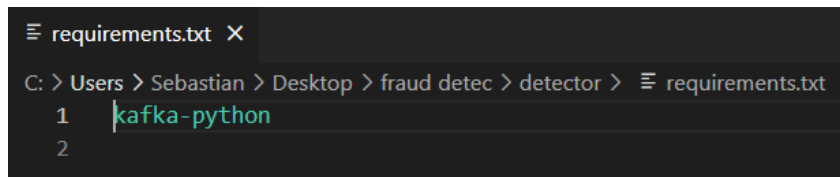


```
Dockerfile X
C: > Users > Sebastian > Desktop > fraud detec > detector > Dockerfile
1 FROM python:3.6
2
3 WORKDIR /usr/app
4
5 ADD ./requirements.txt ./
6 RUN pip install -r requirements.txt
7 ADD ./ ./
8
9 CMD ["python3", "app.py"]
10
```

Rysunek 4 Konfiguracja dockera dla detektora

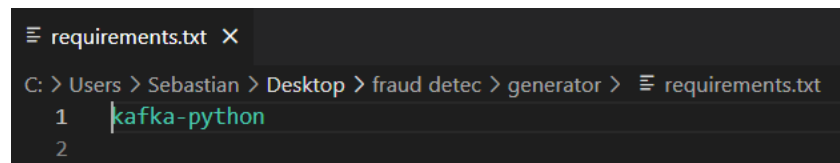
Działa tak samo jak w przypadku generatora opisanego w tekście powyżej.

3. Pliki tekstowe requirements



```
requirements.txt X
C: > Users > Sebastian > Desktop > fraud detec > detector > requirements.txt
1 kafka-python
2
```

Rysunek 5 Plik requirements.txt dla detektora



```
requirements.txt X
C: > Users > Sebastian > Desktop > fraud detec > generator > requirements.txt
1 kafka-python
2
```

Rysunek 6 Plik requirements.txt dla generatora

Pliki "requirements.txt" w Dockerze są używane do zarządzania zależnościami bibliotek i pakietów Pythona dla aplikacji. Zawierają listę bibliotek, które są wymagane do poprawnego działania aplikacji.

Podczas procesu budowania obrazu kontenera, Docker będzie odczytywał zawartość pliku "requirements.txt" i instalował wymienione w nim biblioteki i ich odpowiednie wersje. Dzięki temu, wewnątrz kontenera będą dostępne wszystkie niezbędne zależności, aby aplikacja mogła działać prawidłowo.

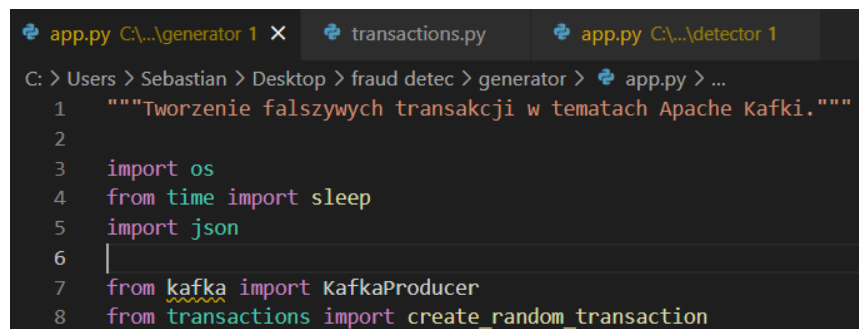
Plik "requirements.txt" jest powszechnie stosowany w społeczności deweloperów Pythona jako sposób na zarządzanie zależnościami projektu. W kontekście Dockera, plik ten jest wykorzystywany podczas budowania obrazu, aby zapewnić, że wszystkie wymagane biblioteki są dostępne wewnątrz kontenera.

4. Wykorzystanie Apache Kafka w języku Python

4.1. Potrzebne biblioteki i moduły

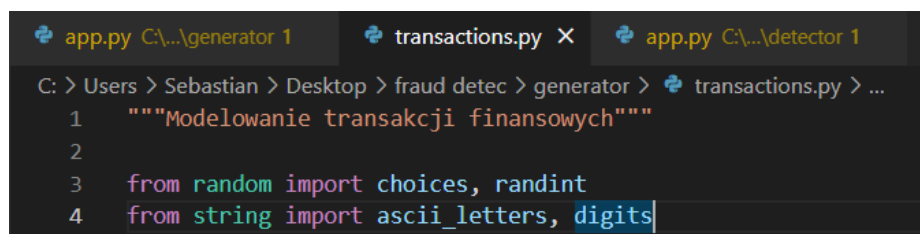
1. Os - dostarcza funkcje do interakcji z systemem operacyjnym. W tym przypadku, może być używana do operacji na środowisku, takich jak dostęp do zmiennych środowiskowych.
2. Time (moduł sleep) - zawiera różne funkcje związane z czasem. W tym kodzie, importowany jest tylko moduł sleep, który pozwala na opóźnienie wykonywania programu przez określony czas.

3. Json - dostarcza funkcje do pracy z formatem JSON (JavaScript Object Notation). Umożliwia parsowanie danych JSON oraz ich serializację i deserializację.
4. Kafka (moduł KafkaProducer) - klient Apache Kafka dla języka Python. W tym kodzie, importowany jest moduł KafkaProducer, który umożliwia wysyłanie wiadomości do klastra Kafka.
5. Kafka (moduł KafkaConsumer) - klient Apache Kafka dla języka Python, umożliwiający interakcję z systemem Kafka. Pozwala na tworzenie konsumentów Kafka, którzy mogą subskrybować tematy i odbierać wiadomości przesyłane do tych tematów.
6. Transactions (moduł create_random_transaction) - lokalny moduł zawierający funkcję create_random_transaction. Prawdopodobnie zawiera logikę generowania losowych transakcji.
7. Random (funkcja choices, funkcja randint) - zawiera funkcje do generowania losowych danych. W tym kodzie, importowane są funkcje choices i randint, które mogą być wykorzystane do generowania losowych wartości.
8. String (stała ascii_letters, stała digits) - zawiera stałe związane z ciągami znaków. W tym kodzie, importowane są stałe ascii_letters (zawierające wszystkie litery alfabetu) i digits (zawierające wszystkie cyfry), które mogą być użyte do generowania losowych ciągów znaków.



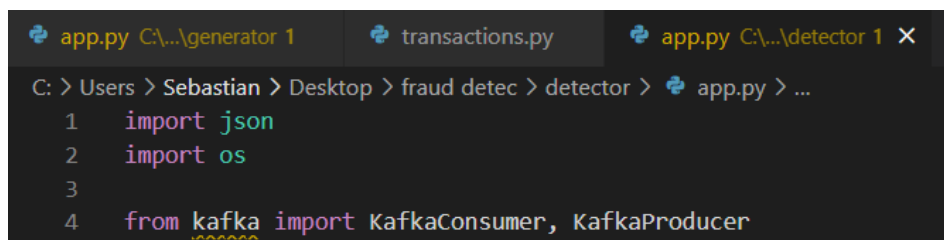
```
app.py C:\...\generator 1 X transactions.py app.py C:\...\detector 1
C: > Users > Sebastian > Desktop > fraud detec > generator > app.py > ...
1  """Tworzenie fałszywych transakcji w tematach Apache Kafki."""
2
3  import os
4  from time import sleep
5  import json
6
7  from kafka import KafkaProducer
8  from transactions import create_random_transaction
```

Rysunek 7 Import bibliotek dla generatora w pliku app.py



```
app.py C:\...\generator 1 transactions.py X app.py C:\...\detector 1
C: > Users > Sebastian > Desktop > fraud detec > generator > transactions.py > ...
1  """Modelowanie transakcji finansowych"""
2
3  from random import choices, randint
4  from string import ascii_letters, digits
```

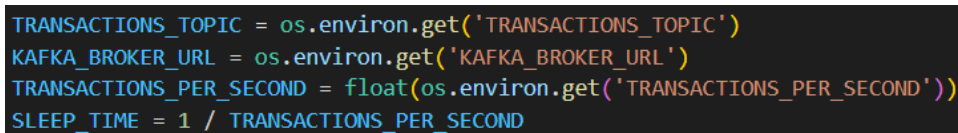
Rysunek 8 Import bibliotek dla generatora, potrzebnych do modelowania transakcji finansowych



```
app.py C:\...\generator 1 transactions.py app.py C:\...\detector 1 X
C: > Users > Sebastian > Desktop > fraud detec > detector > app.py > ...
1 import json
2 import os
3
4 from kafka import KafkaConsumer, KafkaProducer
```

Rysunek 9 Import bibliotek dla detektora potrzebnych do wykrywania podejrzanych transakcji

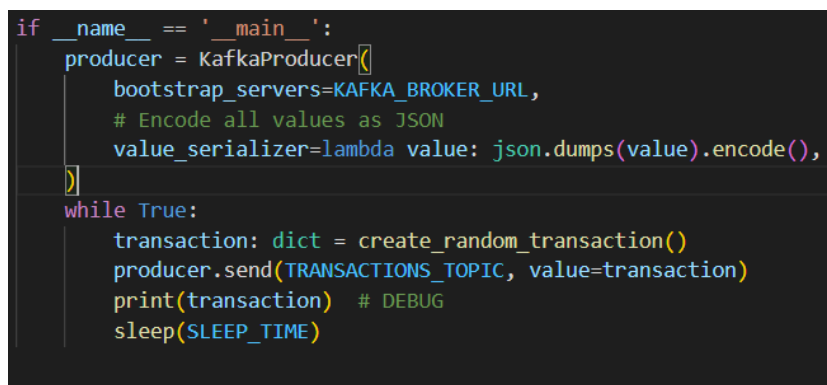
4.2. Tworzenie fałszywych informacji w generatorze



```
TRANSACTIONS_TOPIC = os.environ.get('TRANSACTIONS_TOPIC')
KAFKA_BROKER_URL = os.environ.get('KAFKA_BROKER_URL')
TRANSACTIONS_PER_SECOND = float(os.environ.get('TRANSACTIONS_PER_SECOND'))
SLEEP_TIME = 1 / TRANSACTIONS_PER_SECOND
```

Rysunek 10 Ustawienia zmiennych środowiskowych i obliczanie czasu opóźnienia między transakcjami.

Kod pobiera ustawienia zmiennych środowiskowych dotyczących tematu transakcji, adresu brokera Kafka oraz liczby transakcji na sekundę. Następnie przekształca wartość liczby transakcji na sekundę na czas opóźnienia między transakcjami. Te wartości mogą być wykorzystane w dalszej części kodu do konfiguracji i kontrolowania logiki przetwarzania transakcji w aplikacji.



```
if __name__ == '__main__':
    producer = KafkaProducer(
        bootstrap_servers=KAFKA_BROKER_URL,
        # Encode all values as JSON
        value_serializer=lambda value: json.dumps(value).encode(),
    )
    while True:
        transaction: dict = create_random_transaction()
        producer.send(TRANSACTIONS_TOPIC, value=transaction)
        print(transaction) # DEBUG
        sleep(SLEEP_TIME)
```

Rysunek 11 Tworzenie producenta Kafka i wysyłanie losowe transakcje do tematu.

W tej części kodu, jeśli plik jest uruchamiany bezpośrednio jako program, tworzony jest obiekt `KafkaProducer` z użyciem podanych parametrów:

- `bootstrap_servers=KAFKA_BROKER_URL` - Określa adresy serwerów brokerów Kafka, do których klient Kafka będzie się łączył.
- `value_serializer=lambda value: json.dumps(value).encode()` - Ustawia funkcję do kodowania wartości przesyłanych przez producenta na format JSON.

Następnie w pętli nieskończonej tworzona jest losowa transakcja przy pomocy funkcji `create_random_transaction()`, utworzona transakcja jest wysyłana przez producenta do tematu `TRANSACTIONS_TOPIC` za pomocą metody `producer.send()`.

Transakcja jest wyświetlana na konsoli w celach debugowania, a wykonywanie programu jest wstrzymywane na określony czas opóźnienia `SLEEP_TIME` przy użyciu funkcji `sleep()` z modułu `time`.

W skrócie, ta część kodu tworzy i konfiguruje producenta Kafka, a następnie w pętli wysyła losowe transakcje do określonego tematu Kafka, z uwzględnieniem czasu opóźnienia między transakcjami.

4.3. Modelowanie danych finansowych

```
account_chars: str = digits + ascii_letters
```

Rysunek 12 Generowanie ciągu znaków do numeru konta

Kod definiuje zmienną `account_chars` jako ciąg znaków, który zawiera cyfry (`digits`) oraz małe i duże litery (`ascii_letters`). Zmienna ta jest wykorzystywana do generowania znaków, które mogą być użyte w numerach kont lub identyfikatorach kont w aplikacji.

```
def _random_account_id() -> str:
    """Zwracanie losowego numeru konta złożonego z 12 znaków."""
    return ''.join(choices(account_chars, k=12))
```

Rysunek 13 Losowo generowany numer konta

Funkcja `_random_account_id()` zwraca losowo wygenerowany numer konta, składający się z 12 znaków. Wykorzystuje funkcję `choices()` z biblioteki `random`, aby wybrać losowe znaki zdefiniowane w zmiennej `account_chars` i łączy je w jeden ciąg znaków przy użyciu metody `join()`.

```
def _random_amount() -> float:
    """Zwracanie losowej sumy pieniędzy pomiędzy 1.00 a 1000.00."""
    return randint(100, 100000) / 100
```

Rysunek 14 Generowanie losowej kwoty do transakcji

Funkcja `_random_amount()` zwraca losową kwotę pieniędzy w formacie zmiennoprzecinkowym (typ `float`) między 1.00 a 1000.00. Wykorzystuje funkcję `randint()` z biblioteki `random`, aby wygenerować losową liczbę całkowitą z zakresu od 100 do 100000, a następnie dzieli ją przez 100, aby przekształcić ją w kwotę zmiennoprzecinkową.

```
def create_random_transaction() -> dict:
    """Tworzenie fałszywej, losowej transakcji."""
    return {
        'source': _random_account_id(),
        'target': _random_account_id(),
        'amount': _random_amount(),
        'currency': 'USD',
    }
```

Rysunek 15 Tworzenie fałszywej transakcji dla źródła

Funkcja `create_random_transaction()` tworzy fałszywą, losową transakcję poprzez generowanie losowych wartości dla kluczy `'source'` (źródło), `'target'` (cel), `'amount'` (kwota) oraz ustawienie wartości klucza `'currency'` (waluta) na `'USD'`. Wykorzystuje funkcje `_random_account_id()` do generowania losowych numerów kont oraz `_random_amount()` do generowania losowych kwot pieniędzy. Zwraca transakcję jako słownik (typ `dict`).

4.4. Wykrywanie fałszywej transakcji

```
KAFKA_BROKER_URL = os.environ.get('KAFKA_BROKER_URL')
TRANSACTIONS_TOPIC = os.environ.get('TRANSACTIONS_TOPIC')
LEGIT_TOPIC = os.environ.get('LEGIT_TOPIC')
FRAUD_TOPIC = os.environ.get('FRAUD_TOPIC')
```

Rysunek 16 Odczytywanie wartości zmiennych środowiskowych dla konfiguracji Kafka i detektora.

Kod odczytuje wartości zmiennych środowiskowych za pomocą funkcji `os.environ.get()` i przypisuje je do odpowiadających zmiennych. Wartości te są odczytywane zmiennych o nazwach `'KAFKA_BROKER_URL'`, `'TRANSACTIONS_TOPIC'`, `'LEGIT_TOPIC'` i `'FRAUD_TOPIC'`. Zmienne są wykorzystywane w aplikacji do konfiguracji połączenia z brokerem Kafka oraz do określenia nazw tematów transakcji, tematów prawidłowych transakcji (`legit`) i tematów oszustwowych transakcji (`fraud`).


```
def is_suspicious(transaction: dict) -> bool:
    """Sprawdzenie, czy transakcja jest podejrzana"""
    return transaction['amount'] >= 900
```

Rysunek 17 Sprawdzanie, czy transakcja jest podejrzana

Funkcja `is_suspicious()` sprawdza, czy transakcja jest podejrzana, przyjmując transakcję jako argument w postaci słownika. Jej implementacja polega na porównaniu wartości klucza 'amount' (kwota) w transakcji z wartością 900. Jeśli kwota transakcji jest większa lub równa 900, funkcja zwraca True, w przeciwnym przypadku zwraca False.

```
if __name__ == '__main__':
    consumer = KafkaConsumer(
        TRANSACTIONS_TOPIC,
        bootstrap_servers=KAFKA_BROKER_URL,
        value_deserializer=lambda value: json.loads(value),
    )
    producer = KafkaProducer(
        bootstrap_servers=KAFKA_BROKER_URL,
        value_serializer=lambda value: json.dumps(value).encode(),
    )
    for message in consumer:
        transaction: dict = message.value
        topic = FRAUD_TOPIC if is_suspicious(transaction) else LEGIT_TOPIC
        producer.send(topic, value=transaction)
        print(topic, transaction)
```

Rysunek 18 Tworzenie konsumenta i producenta Kafka, przetwarzanie transakcji i aktualizacja tematu.

W głównej części kodu, jeśli jest uruchamiana jako skrypt, tworzone są obiekty konsumenta (consumer) i producenta (producer) Kafka. Konsument jest skonfigurowany do subskrybowania tematu transakcji (TRANSACTIONS_TOPIC) z podanym adresem brokera (KAFKA_BROKER_URL) oraz deserializacji wartości jako dane JSON. Producent jest skonfigurowany do korzystania z tego samego adresu brokera i serializowania wartości jako dane JSON. Następnie w pętli `for` iteruje się przez każdą wiadomość otrzymaną przez konsumenta. Wartość wiadomości jest przypisywana do zmiennej `transaction` jako słownik. Następnie na podstawie oceny podejrzaności transakcji (`is_suspicious(transaction)`) określany jest temat, do którego zostanie wysłana transakcja. Jeśli transakcja jest podejrzana, temat będzie `FRAUD_TOPIC`, w przeciwnym razie będzie to `LEGIT_TOPIC`. Transakcja zostaje wysłana do odpowiedniego tematu przez producenta, a następnie wypisany jest temat oraz transakcja na konsolę.

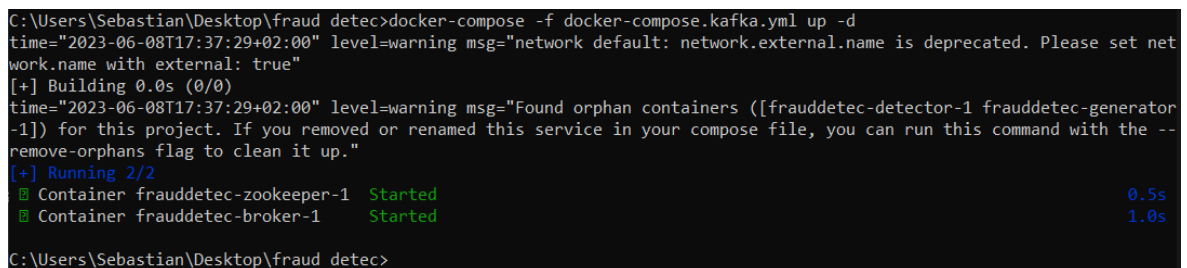
5. Uruchamianie Dockera za pomocą wiersza poleceń lub terminala

Aby kod działał poprawnie wystarczy utworzyć sieć Docker o nazwie "kafka-network", aby umożliwić komunikację między klastrami Kafka a aplikacjami:

\$ docker network create kafka-network

Następnie należy uruchomić lokalny klaster Kafka (działa on w tle):

\$ docker-compose -f docker-compose.kafka.yml up -d



```
C:\Users\Sebastian\Desktop\fraud detec>docker-compose -f docker-compose.kafka.yml up -d
time="2023-06-08T17:37:29+02:00" level=warning msg="network default: network.external.name is deprecated. Please set net
work.name with external: true"
[+] Building 0.0s (0/0)
time="2023-06-08T17:37:29+02:00" level=warning msg="Found orphan containers ([frauddetec-detector-1 frauddetec-generator
-1]) for this project. If you removed or renamed this service in your compose file, you can run this command with the --
remove-orphans flag to clean it up."
[+] Running 2/2
   Container frauddetec-zookeeper-1   Started      0.5s
   Container frauddetec-broker-1     Started      1.0s
C:\Users\Sebastian\Desktop\fraud detec>
```

Rysunek 19 Uruchamianie klastra lokalnego Kafka

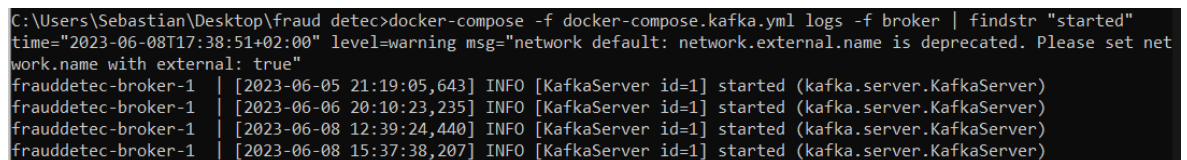
Sprawdzenie, czy klaster jest uruchomiony i działa (należy poczekać na pojawienie się komunikatu "started"):

Dla terminala Linuxa:

\$ docker-compose -f docker-compose.kafka.yml logs -f broker | grep "started"

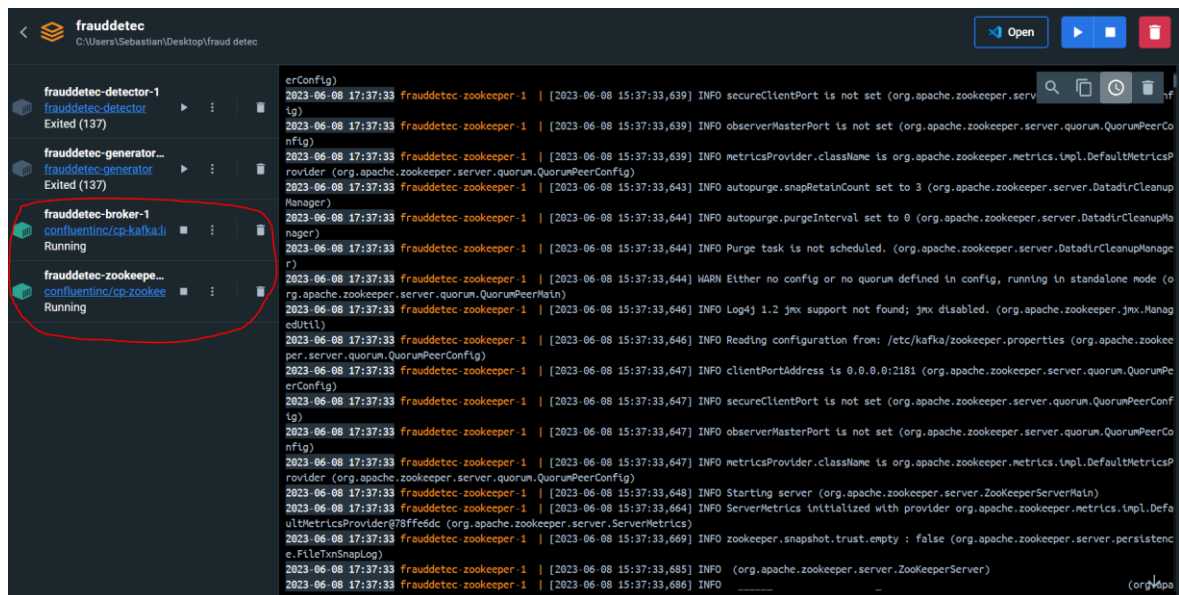
Dla wiersza poleceń Windows:

docker-compose -f docker-compose.kafka.yml logs -f broker | findstr "started"



```
C:\Users\Sebastian\Desktop\fraud detec>docker-compose -f docker-compose.kafka.yml logs -f broker | findstr "started"
time="2023-06-08T17:38:51+02:00" level=warning msg="network default: network.external.name is deprecated. Please set net
work.name with external: true"
frauddetec-broker-1 | [2023-06-05 21:19:05,643] INFO [KafkaServer id=1] started (kafka.server.KafkaServer)
frauddetec-broker-1 | [2023-06-06 20:10:23,235] INFO [KafkaServer id=1] started (kafka.server.KafkaServer)
frauddetec-broker-1 | [2023-06-08 12:39:24,440] INFO [KafkaServer id=1] started (kafka.server.KafkaServer)
frauddetec-broker-1 | [2023-06-08 15:37:38,207] INFO [KafkaServer id=1] started (kafka.server.KafkaServer)
```

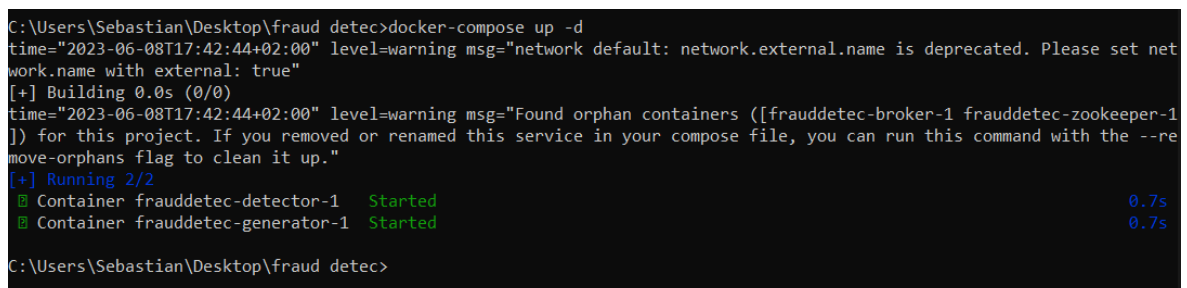
Rysunek 20 Sprawdzenie działania klastra, oraz historia uruchomień



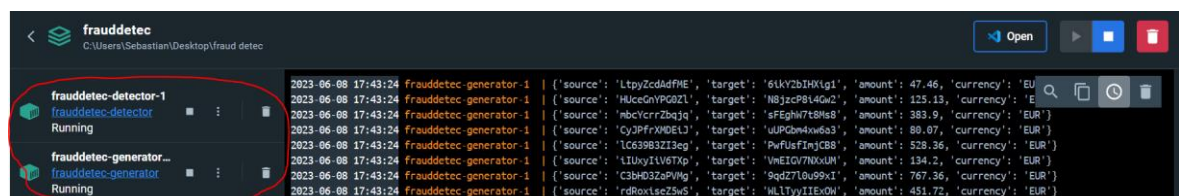
Rysunek 21 Uruchomiony ZooKeeper oraz Broker

Uruchamianie generator transakcji oraz detektor oszustw (będzie działać w tle), należy to wykonać w nowym oknie terminala/wiersza poleceń:

\$ docker-compose up -d



Rysunek 22 Uruchamianie generatora transakcji oraz detektora



Rysunek 23 Uruchomiony generator oraz detektor

Wyświetlanie strumienia transakcji w temacie T (opcjonalnie, można dodać **--from-beginning**, aby rozpocząć od początku)

\$ docker-compose -f docker-compose.kafka.yml exec broker kafka-console-consumer --bootstrap-server localhost:9092 --topic T

```
C:\Users\Sebastian\Desktop\fraud detec>docker-compose -f docker-compose.kafka.yml exec broker kafka-console-consumer --bootstrap-server localhost:9092 --topic T --from-beginning
time="2023-06-08T17:45:48+02:00" level=warning msg="network default: network.external.name is deprecated. Please set network.name with external: true"
```

Rysunek 24 Strumień transakcji w temacie T

Powstałe tematy:

- **queuing.transactions**: losowo wygenerowane transakcje do przetworzenia

```
2023-06-08 17:48:40 frauddetec-generator-1 | {'source': 'i30XC1l4KGzb', 'target': 'NzhCPioCGUsh', 'amount': 165.97, 'currency': 'EUR'}
2023-06-08 17:48:40 frauddetec-generator-1 | {'source': 'I43E58ifCARw', 'target': 'tJirj5w3540k', 'amount': 721.77, 'currency': 'EUR'}
2023-06-08 17:48:40 frauddetec-generator-1 | {'source': '5wmNQbPxoeP3', 'target': 'mgt5XHf1Zxmf', 'amount': 954.92, 'currency': 'EUR'}
```

Rysunek 25 Losowo wygenerowane transakcje

- **streaming.transactions.legit**: prawidłowe transakcje

```
2023-06-08 17:48:48 frauddetec-detector-1 | streaming.transactions.legit {'source': '03Hec9NBKs1Y', 'target': '0yEWLnTFxTv7', 'amount': 488.27, 'currency': 'EUR'}
2023-06-08 17:48:48 frauddetec-detector-1 | streaming.transactions.legit {'source': 'HnzCX2a7La6a', 'target': 'PBxgDATGm0o', 'amount': 863.81, 'currency': 'EUR'}
2023-06-08 17:48:48 frauddetec-detector-1 | streaming.transactions.legit {'source': '9djTeMwDFCmN', 'target': '1MpjPNOifknR', 'amount': 54.52, 'currency': 'EUR'}
```

Rysunek 26 Transakcje wykryte jako prawidłowe

- **streaming.transactions.fraud**: podejrzane transakcje

```
2023-06-08 17:48:49 frauddetec-detector-1 | streaming.transactions.legit {'source': 'LpDBhqqB0282', 'target': 'cg3VB9nNahPa', 'amount': 148.48, 'currency': 'EUR'}
2023-06-08 17:48:49 frauddetec-detector-1 | streaming.transactions.fraud {'source': 'ukOt4x6Pu6V4', 'target': 'UGp4WfJdguQG', 'amount': 957.55, 'currency': 'EUR'}
2023-06-08 17:48:49 frauddetec-detector-1 | streaming.transactions.legit {'source': 'I7vS0p6JLSN1', 'target': 'c6C4ZcdqL9oU', 'amount': 875.34, 'currency': 'EUR'}
```

Rysunek 27 Transakcja wykryta jako podejrzana pośród prawidłowych

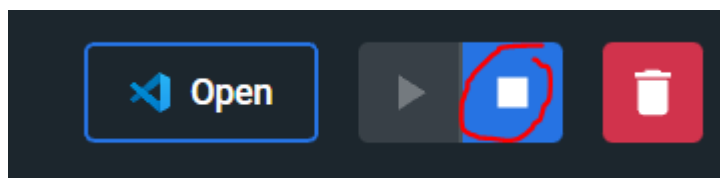
Przykładowo zwrócona wiadomość transakcji:

```
{"source": "Zj2Khk9LiO9q", "target": "JSm8hjs9u2UW", "amount": 456.87, "currency": "USD"}
```

```
2023-06-08 17:47:07 frauddetec-generator-1 | {'source': 'sL7f10aCr0D1', 'target': 'dJhLneqECbnD', 'amount': 234.52, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-generator-1 | {'source': 'uV35UgKT2sK0', 'target': 'NwbGG6epTSkS', 'amount': 306.91, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-generator-1 | {'source': 'QDX9UW4JRHFn', 'target': 'Vs2yALy1kWi7', 'amount': 504.62, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-generator-1 | {'source': 'ANRD2c3z1j1V', 'target': 'FALW2nUX53A6', 'amount': 109.55, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-generator-1 | {'source': 'TZLeEQK1VCPD', 'target': 'EPWcTcTlLYe', 'amount': 765.86, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-generator-1 | {'source': 'ge0qpB8X9bLL', 'target': 'IOw7RwIypht', 'amount': 963.65, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-generator-1 | {'source': 'fYDEBx8TcFPZ', 'target': 'U0N3In8Uky2j', 'amount': 774.11, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-generator-1 | {'source': 'LB9REUVRxwlb', 'target': 'rwjrpypghfQZ', 'amount': 642.07, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-generator-1 | {'source': 'ZFK1v9VKXfSp', 'target': 'ZQ8ytVK1UJ00', 'amount': 634.59, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-generator-1 | {'source': 'WSbhBh3BvWly', 'target': 'qbMA2meSSbNr', 'amount': 404.91, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-detector-1 | streaming.transactions.legit {'source': 'FmG54kgNoDuE', 'target': 'KxpBHS4MDxpI', 'amount': 592.67, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-detector-1 | streaming.transactions.legit {'source': 'uecRIDEcnqLu', 'target': '4EyJkbvUAHgI', 'amount': 97.61, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-detector-1 | streaming.transactions.legit {'source': 'h9HpyFIdkLcq', 'target': 'kGXetnrMMfxN', 'amount': 358.21, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-detector-1 | streaming.transactions.legit {'source': '4ddqsTueNc3M', 'target': '3yIdc0A5x4dM', 'amount': 204.55, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-detector-1 | streaming.transactions.legit {'source': 'XSoJMAN1EuKs', 'target': '4Y5TdTPK89MG', 'amount': 160.41, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-detector-1 | streaming.transactions.legit {'source': '02Jrv1NJYhlp', 'target': 'UUQ0ehpDzdVu', 'amount': 411.23, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-detector-1 | streaming.transactions.legit {'source': 'SMaY9uFT4Ya1', 'target': 'eUXTS6EnD0S1', 'amount': 301.45, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-detector-1 | streaming.transactions.legit {'source': 'e3aSyxP5R9yF', 'target': 'Dj2HAK10ZZXq', 'amount': 221.21, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-detector-1 | streaming.transactions.legit {'source': 'KmKws9ZahdzU', 'target': 'PINuWTKASbpl', 'amount': 693.89, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-detector-1 | streaming.transactions.legit {'source': 'I11SeBnhyAkt', 'target': 'xKv7EX9qvcTI', 'amount': 702.71, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-detector-1 | streaming.transactions.legit {'source': 'sL7f10aCr0D1', 'target': 'dJhLneqECbnD', 'amount': 234.52, 'currency': 'EUR'}
2023-06-08 17:47:07 frauddetec-detector-1 | streaming.transactions.legit {'source': 'uV35UgKT2sK0', 'target': 'NwbGG6epTSkS', 'amount': 306.91, 'currency': 'EUR'}
```

Rysunek 28 Generowane i przetwarzane informacje

Aby zatrzymać generator transakcji i detektor oszustw wystarczy zamknąć okna wiersza poleceń/terminala oraz w dockerze kliknąć przycisk w prawym górnym rogu:



Rysunek 29 Przycisk do zatrzymania procesów

```
1970-01-01 01:00:00 frauddetec-generator-1 exited with code 0
1970-01-01 01:00:00 frauddetec-generator-1 exited with code 137
1970-01-01 01:00:00 frauddetec-detector-1 exited with code 0
1970-01-01 01:00:00
```

Rysunek 30 Informacja, że proces został zamknięty prawidłowo

Informacja dodatkowa:

Podczas kopiowania poleceń należy pamiętać, że w system Windows w przeciwieństwie do Linuxa **NIE obsługuje** znaku \$, który oznacza, że użytkownik ma standardowe uprawnienia użytkownika.

6. Podsumowanie

Projekt "Aplikacja do przetwarzania transakcji finansowych z wykorzystaniem Apache Kafka" skupiał się na stworzeniu niezawodnej i skalowalnej aplikacji, która wykorzystuje narzędzia takie jak Apache Kafka, Python i Apache ZooKeeper. Wdrażając aplikację w środowisku kontenerowym za pomocą Docker, osiągnięte zostały izolacja, skalowalność i elastyczność. Wykorzystanie Apache Kafka umożliwiło przesyłanie i przetwarzanie danych transakcyjnych w czasie rzeczywistym, co jest kluczowe dla instytucji finansowych. Tworzenie dedykowanej sieci (kafka-network) wewnątrz kontenerów pozwoliło na efektywną komunikację między komponentami aplikacji. Projekt ten dostarcza solidne fundamenty dla stabilnej i wydajnej aplikacji do przetwarzania transakcji finansowych, która może być łatwo replikowana i zarządzana w środowiskach produkcyjnych.

7. Spis rysunków

Rysunek 1 Konfiguracja docker-compose.yml.....	9
Rysunek 2 Plik konfiguracyjny docker-compose.kafka.yml	10
Rysunek 3 Konfiguracja dockera dla generatora	11
Rysunek 4 Konfiguracja dockera dla detektora	11
Rysunek 5 Plik requirements.txt dla detektora	12
Rysunek 6 Plik requirements.txt dla generatora	12
Rysunek 7 Import bibliotek dla generatora w pliku app.py	13
Rysunek 8 Import bibliotek dla generatora, potrzebnych do modelowania transakcji finansowych	13
Rysunek 9 Import bibliotek dla detektora potrzebnych do wykrywania podejrzanych transakcji	14
Rysunek 10 Ustawienia zmiennych środowiskowych i obliczanie czasu opóźnienia między transakcjami.	14
Rysunek 11 Tworzenie producenta Kafka i wysyłanie losowe transakcje do tematu.	14
Rysunek 12 Generowanie ciągu znaków do numeru konta	15
Rysunek 13 Losowo generowany numer konta	15

Rysunek 14 Generowanie losowej kwoty do transakcji	16
Rysunek 15 Tworzenie fałszywej transakcji dla źródła.....	16
Rysunek 16 Odczytywanie wartości zmiennych środowiskowych dla konfiguracji Kafka i detektora.....	16
Rysunek 17 Sprawdzanie, czy transakcja jest podejrzana	17
Rysunek 18 Tworzenie konsumenta i producenta Kafka, przetwarzanie transakcji i aktualizacja tematu.	17
Rysunek 19 Uruchamianie klastra lokalnego Kafka.....	18
Rysunek 20 Sprawdzenie działania klastra, oraz historia uruchomień.....	18
Rysunek 21 Uruchomiony ZooKeeper oraz Broker	19
Rysunek 22 Uruchamianie generatora transakcji oraz detektora.....	19
Rysunek 23 Uruchomiony generator oraz detektor	19
Rysunek 24 Strumień transakcji w temacie T	20
Rysunek 25 Losowo wygenerowane transakcje	20
Rysunek 26 Transakcje wykryte jako prawidłowe	20
Rysunek 27 Transakcja wykryta jako podejrzana pośród prawidłowych.....	20
Rysunek 28 Generowane i przetwarzane informacje	21
Rysunek 29 Przycisk do zatrzymania procesów	21
Rysunek 30 Informacja, że proces został zamknięty prawidłowo.....	21

8. Użyte materiały

1. <https://towardsdatascience.com/getting-started-with-apache-kafka-in-python-604b3250aa05>
2. <https://docs.docker.com/desktop/install/windows-install/>
3. <https://docs.docker.com/desktop/install/linux-install/>
4. <https://kafka-python.readthedocs.io/en/master/>
5. https://en.wikipedia.org/wiki/Apache_Kafka
6. https://en.wikipedia.org/wiki/Apache_ZooKeeper