Le but de ce projet est de créer un système de gestion de portefeuille boursier. Le système permettra à l'utilisateur de spécifier une liste d'actifs et leurs pondérations respectives. Le système récupèrera ensuite les données de marché pour chacun de ces actifs et calculera le risque global du portefeuille. Le système est composé de plusieurs entités, agrégats et contextes. Les entités incluent des classes pour représenter des symboles, des prix et des volumes, tandis que les agrégats comprennent des classes pour représenter un portefeuille d'actifs et des données de marché pour ces actifs. Le contexte principal est ApplicationContext, qui est responsable de l'initialisation des autres contextes, notamment PortfolioContext et RiskManagementContext. PortfolioContext est responsable de la récupération des données de marché et de la création d'un portefeuille, tandis que RiskManagementContext est responsable du calcul du risque global du portefeuille.

Le système utilise également un modèle d'entrepôts (repository) pour accéder aux données de portefeuille et de marché. En outre, le système utilise la programmation asynchrone pour améliorer les performances lors de la récupération des données de marché.

Pour utiliser le système, l'utilisateur peut simplement créer une instance de ApplicationContext et appeler la méthode calculate_portfolio_risk sur son RiskManagementContext. Le système retournera alors une estimation du risque global du portefeuille basée sur les données de marché récupérées pour chaque actif et leurs pondérations respectives.

Section 1 : Threading

Lisez le code suivant avant de répondre les questions

```
import random
import sqlite3
import threading
import time


market_data_list = [
    {'symbol': 'AAPL', 'price': 150.0, 'volume': 1000000},
    {'symbol': 'GOOGL', 'price': 2500.0, 'volume': 500000},
    {'symbol': 'AAPL', 'price': 155.0, 'volume': 1200000},
    {'symbol': 'GOOGL', 'price': 2525.0, 'volume': 550000},
    {'symbol': 'AAPL', 'price': 157.0, 'volume': 1500000},
    {'symbol': 'GOOGL', 'price': 2550.0, 'volume': 600000},
]


def create_market_data():
    conn = sqlite3.connect('market_data.db')
```

```
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS market_data (
            symbol TEXT,
            price REAL,
            volume INTEGER,
            timestamp BIGINT
        )
    ''')
    conn.commit()
    conn.close()


def update_market_data(event: threading.Event, waiting_time: float = 60.0):
    create_market_data()
    conn = sqlite3.connect('market_data.db')
    c = conn.cursor()
    while True:
        for data in market_data_list:
            # to do
            time.sleep(random.uniform(0.1, 1))
        event.wait(waiting_time)



if __name__ == '__main__':
    t_event = threading.Event()
    update_market_data(t_event, 60)
```

1. Qu'est-ce que fait ce code ?
2. Quels sont les modules importés dans ce code ?
3. Y-a-t-il un problème dans la méthode create_market_data() ? Comment le fixer ?
4. Qu'est-ce que la liste "market_data_list" contient ?
5. Que fait la fonction "create_market_data" ?
6. Que fait l'appel lors d'arriver event.wait(waiting_time) (ou threading.Event().wait(60)) ?
7. Comment compléter "update_market_data" ?
8. Quel est le but de l'instruction "if name == 'main' "?
9. Comment met-on la tâche update_market_data() au fond ?

Le code complet est dans le module market_data_sqlite3.py

Section 2 : Flask RestX

Nous créons un module qui implémente une API Flask pour récupérer des données de marché pour une entreprise spécifique. Il utilise SQLite pour stocker les données de marché, et Flask RestX pour gérer les endpoints de l'API. Lorsqu'un utilisateur envoie une requête GET avec un symbole d'entreprise spécifique, l'API interroge la base de données SQLite pour récupérer les dernières données de marché pour ce symbole. Si les données sont disponibles, elles sont renvoyées sous forme de réponse HTTP. Si aucune donnée n'est disponible pour le symbole spécifié, une réponse d'erreur 404 est renvoyée. L'API utilise également Flask RestX pour définir un modèle pour les données de marché, qui comprend les champs 'symbol', 'price' et 'volume'. Ce modèle est utilisé pour valider les données d'entrée et pour fournir une documentation claire de l'API.

Nous créons un module market_data_restx_api.py

1. Importez les modules nécessaires : sqlite3, Flask, et flask_restx.
2. Créez une instance de l'application Flask.
3. Créez une instance de l'API avec une version, un titre et une description.
4. Créez un namespace pour les endpoints de l'API internes pour les données de marché.
5. Définissez un modèle pour les données de marché avec les champs 'symbol', ' price' et 'volume'.
6. Créez une classe MarketData qui hérite de la classe Resource.
7. Ajoutez une méthode get_market_data qui prend en paramètre un symbole et qui retourne les dernières données de marché pour ce symbole.
8. Ajoutez une méthode get à la classe MarketData qui utilise la méthode get_market_data pour récupérer les données de marché pour un symbole spécifique et qui retourne les données sous forme de réponse HTTP.
9. Définissez une condition if **name** == '**main**' pour lancer l'application Flask en mode debug.

Section 3 : Domain Driven Design

C'est le moment de créer le projet qui consiste en la création d'un système de récupération et de stockage de données de marché pour différents actifs financiers tels que des actions ou des devises. En utilisant la conception en Domain Driven Design (DDD), nous pouvons identifier les différents objets qui sont impliqués dans le processus et les organiser en fonction de leur rôle. Voici les étapes du développement en DDD : (N.B. n'oubliez pas les mises à jour chez les tests unitaires)

1. Identifiez et classifiez les éléments dans DDD

- Symbole (Symbol) : représente le symbole de l'actif financier (ex: AAPL pour Apple et GOOGL pour Google)
- Prix (Price) : représente le prix actuel de l'actif financier
- Volume (Volume) : représente le volume d'actions échangées
- Actif financier (Asset) : représente l'actif financier en lui-même. Il est identifié par son symbole et possède un prix et un volume.
- Poids (Weight) : représente le poids d'actifs
- Les données de marché (MarketData) : représente une triple de Symbole, Prix, Volume pour la simplification
- Portefeuille (Portfolio) : représente une liste d'actifs financiers. Il va nous permettre de manipuler plusieurs actifs financiers en même temps. Par exemple calculer leurs poids en total
- Factory de donnée de marché (MarketDataFactory) : fournit une méthode pour créer des objets MarketData à partir de données brutes.
- Entrepôt de données de portefeuille (PortfolioRepository) : fournit un accès à la persistance des données à un portefeuille en donnant les poids aux actifs dedans.
- Entrepôt de données de marché (MarketDataRepository) : fournit un accès à la persistance des données RESTFUL à un actif par son symbole.
- Service de données de marché (MarketDataService) : fournit une méthode pour récupérer des données de marché pour une liste d'actifs en utilisant une API HTTP.
- Événement de modification de données de marché (MarketDataFetched): est déclenché chaque fois que des données de marché sont récupérées avec succès pour un actif.
- Contexte de portefeuille (PortfolioContext) : contient des références aux classes PortfolioRepository, MarketDataRepository, MarketDataFactory et MarketDataFetchedHandler. Elle fournit des méthodes pour récupérer les données de marché pour les actifs dans le portefeuille, calculer le risque du portefeuille et accéder aux données de marché pour un symbole donné.
- Contexte de gestion de risque (RiskManagementContext) : contient une référence à PortfolioContext. Elle fournit une méthode pour calculer le risque du portefeuille.
- Contexte d'application (ApplicationContext): contient des références à toutes les classes précédentes. Elle fournit une méthode run() qui exécute l'application.

D'ailleurs, en termes de la carte de contextes (context map), elle montre comment les contextes délimités interagissent les uns avec les autres dans un système. Il peut inclure divers types de relations, telles que des partenariats, des relations client-fournisseur, des relations amont-aval ou des relations de noyau partagé. La carte de contexte peut également inclure des modèles de gestion de la communication et de l'intégration entre des

contextes délimités, tels que des couches anti-corruption, conformiste, un service d'hôte ouvert ou un langage publié.

Quelle est la carte de contexte pour les contextes dans notre projet ?

2. À quoi servent les libs suivantes à importer dans le projet :

```
from dataclasses import dataclass

import aiohttp
import asyncio
```

3. Implémentez les objets de valeur (values.py) et les objets d'entité (entities.py) dont par ailleurs, une interface est demandée à compléter :

```
from dataclasses import dataclass, field

from .values import Price, Symbol, Weight, Volume
from ...utils.numbers import get_id

@dataclass
class Entity:
    id: int = field(default_factory=get_id().__next__)

    def __eq__(self, other):
        # to do
```

4. Complétez la classe Portfolio (aggregates.py)

```
@dataclass
class Portfolio(Entity):
    assets: List[Asset] = None

    def total_weight(self):
        # to do
```

5. Complétez la classe MarketDataFactory (factories.py)

```
class MarketDataFactory:
    @classmethod
    def create_market_data(cls, symbol: str, price: float, volume: float):
        #  to do
```

6. Créez une propriété portfolio du récupérateur de données de portefeuille ainsi que celle de db_connector (repositories.py)

```
# Repository
class PortfolioRepository:
 def __init__(self, db_connector=None):
     self._db_connector = db_connector
     self._portfolio = self.get()
```

```
def get(self):
    logger.info("DB Connection for Portfolio fetch")
    logger.debug(self.db_connector or "Mock Portfolio from DB")
    return Portfolio(assets=[
        Asset(symbol=Symbol('AAPL'), weight=Weight(0.5)),
        Asset(symbol=Symbol('GOOGL'), weight=Weight(0.5))])
```

7. Complétez la classe MarketDataRepository (repositories.py)

```
class MarketDataRepository:
    @classmethod
    async def get_market_data(cls, session, symbol):
        # to do

        return MarketDataFactory.create_market_data(
             symbol=symbol,
             price=response_json['price'],
             volume=response_json['volume'])
```

8. Complétez la classe MarketDataService (market_data.py)

```
class MarketDataService:
    @classmethod
    async def get_market_data_for_assets(cls, assets):
        # to do
```

9. Complétez la classe PortfolioContext (contexts.py) en supposant que

```
# Evénements
class MarketDataFetched:
    def __init__(self, symbol: Symbol, price: Price, volume: Volume,
                 timestamp: float = utc_now_tsp):
        self.symbol = symbol
        self.price = price
        self.volume = volume
        self.timestamp = timestamp


class PortfolioContext:
    def __init__(self, portfolio_repository: PortfolioRepository,
                 market_data_service: MarketDataService,
                 market_data_factory: MarketDataFactory,
                 market_data_fetched_handler):
         #  to do

    async def fetch_market_data(self):
         self.portfolio = self.portfolio_repository.portfolio
         #  to do
```

```
            for md in market_data:
                self.market_data[md.symbol.name] = md
                self._market_data_fetched_handler(MarketDataFetched(
                    md.symbol, md.price, md.volume))

    def get_market_data(self, symbol: Symbol):
        #  to do

    def calculate_risk(self) -> float:
        return sum(asset.weight.value
                   * self.market_data[asset.symbol.name].price.value
                   / self.market_data[asset.symbol.name].volume.value
                   for asset in self.portfolio.assets) / self.total_weight()

    def total_weight(self):
        return self.portfolio.total_weight()
```

10. Complétez la classe RiskManagementContext (contexts.py)

```
class RiskManagementContext:
   def __init__(self, portfolio_context: PortfolioContext):
       self.portfolio_context = portfolio_context

   def calculate_portfolio_risk(self) -> float:
       #  to do
```

11. Complétez la classe ApplicationContext (contexts.py) pour que l'on puisse exécuter asyncio.run(ApplicationContext().run())

```
class ApplicationContext:
    def __init__(self):
        self.portfolio_repository = PortfolioRepository()
        self.market_data_repository = MarketDataService()
        self.market_data_factory = MarketDataFactory()
        self.market_data_fetched_handler = self.on_market_data_fetched
        self.portfolio_context = PortfolioContext(
            portfolio_repository=self.portfolio_repository,
            market_data_service=self.market_data_repository,
            market_data_factory=self.market_data_factory,
            market_data_fetched_handler=self.market_data_fetched_handler)
        self.risk_management_context = RiskManagementContext(
            portfolio_context=self.portfolio_context)

    @classmethod
    def on_market_data_fetched(cls, event: MarketDataFetched):
        logger.info(
            f"Received event: {event.symbol.name}, {event.price.value}, "
            f"{event.volume.value}")
```

```
        async def run(self):
            #  to do
```

Annexes :

Structure arborescente du code :

```
|-- solutions
|   |-- __init__.py
|   |-- applications
|   |   |-- __init__.py
|   |   |-- globals
|   |   |   |-- __init__.py
|   |   |   `-- contexts.py
|   |   |-- market_data
|   |   |   |-- __init__.py
|   |   |   |-- events.py
|   |   |   `-- repositories.py
|   |   |-- portfolios
|   |   |   |-- __init__.py
|   |   |   |-- contexts.py
|   |   |   |-- events.py
|   |   |   |-- factories.py
|   |   |   `-- repositories.py
|   |   `-- risks
|   |       |-- __init__.py
|   |       `-- contexts.py
|   |-- domains
|   |   |-- __init__.py
|   |   |-- models
|   |   |   |-- __init__.py
|   |   |   |-- aggregates.py
|   |   |   |-- entities.py
|   |   |   `-- values.py
|   |   `-- services
|   |       |-- __init__.py
|   |       `-- market_data.py
|   `-- infrastructures
|   |   |-- __init__.py
|   |   |-- main.py
|   |   |-- external_services
|   |   |   |-- __init__.py
|   |   |   `-- market_data_sqlite3.py
|   |   `-- internal_services
|   |       |-- __init__.py
```

```
|   |       `-- market_data_restx_api.py
|   `-- utils
|       |-- __init__.py
|       |-- datetimes.py
|       |-- loggers.py
|       `-- numbers.py
`-- tests
    |-- __init__.py
    |-- applications
    |   |-- __init__.py
    |   |-- globals
    |   |   |-- __init__.py
    |   |   `-- test_contexts.py
    |   |-- market_data
    |   |   |-- __init__.py
    |   |   |-- test_events.py
    |   |   `-- test_repositories.py
    |   |-- portfolios
    |   |   |-- __init__.py
    |   |   |-- test_contexts.py
    |   |   |-- test_events.py
    |   |   |-- test_factories.py
    |   |   `-- test_repositories.py
    |   `-- risks
    |       `-- test_contexts.py
    |-- domains
    |   |-- __init__.py
    |   |-- models
    |   |   |-- __init__.py
    |   |   |-- test_aggregates.py
    |   |   |-- test_entities.py
    |   |   `-- test_values.py
    |   `-- services
    |   |   |-- __init__.py
    |       `-- test_market_data.py
    |-- infrastructures
    |   |-- __init__.py
    |   |-- external_services
    |   |   |-- __init__.py
    |   |   `-- test_market_data_sqlite3.py
    |   `-- internal_services
    |   |   |-- __init__.py
    |       `-- test_market_data_restx_api.py
    `-- utils
        |-- __init__.py
        |-- test_datetimes.py
```

```
|-- test_loggers.py
`-- test_numbers.py
```