__init__.py

__init__.py

values.py

```
from dataclasses import dataclass


@dataclass(frozen=True)
class Symbol:
    name: str


@dataclass(frozen=True)
class Price:
    value: float


@dataclass(frozen=True)
class Volume:
    value: float


@dataclass(frozen=True)
class Weight:
    value: float
```

__init__.py

aggregates.py

```
from dataclasses import dataclass
from typing import List

from solutions.domains.models.entities import Asset, Entity
```

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

    def total_weight(self):
        return sum(asset.weight.value for asset in self.assets)
```

entities.py

```
from dataclasses import dataclass, field

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


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

    def __eq__(self, other):
        if not isinstance(other, Entity):
            return False
        return self.id == other.id


# Objets d'Entité
@dataclass
class Asset(Entity):
    symbol: Symbol = None
    weight: Weight = None


@dataclass
class MarketData(Entity):
    symbol: Symbol = None
    price: Price = None
    volume: Volume = None
```

__init__.py

market_data.py

```
import asyncio

import aiohttp as aiohttp

from solutions.applications.market_data.repositories import \
    MarketDataRepository


class MarketDataService:
    @classmethod
    async def get_market_data_for_assets(cls, assets):
        async with aiohttp.ClientSession(trust_env=True) as session:
            tasks = [
                asyncio.create_task(
                    MarketDataRepository.get_market_data(
                        session, asset.symbol.name)) for
                asset in assets]
            return await asyncio.gather(*tasks)
```

loggers.py

```
import datetime
from pathlib import Path

from loguru import logger

log_file = Path(__file__).parents[2] / "logs" /\
           "app_risks_" \
           f"{datetime.datetime.now().strftime('%Y-%m-%d_%H-%M-%S')}.log"
log_file.parent.mkdir(parents=True, exist_ok=True)

# logger.add(sys.stdout, format="{time} {level} {message}", colorize=True)
logger.add(log_file, format="{time} {level} {message}", colorize=True)
```

datetimes.py

```
import datetime
import time

utc_now = datetime.datetime.utcnow().replace(tzinfo=datetime.timezone.utc)
utc_now_tsp: int = int(utc_now.timestamp())
```

```
monotonic_time = time.monotonic()

if __name__ == '__main__':
    print(
        f"Monotonic UTC now: {utc_now.isoformat()} or {utc_now_tsp} "
        f"({monotonic_time:.3f} seconds since epoch)")
```

__init__.py

numbers.py

```
import uuid


def get_id():
    while True:
        yield uuid.uuid4()
```

__init__.py

main.py

```
import asyncio
import threading

from solutions.infrastructures.external_services.market_data_sqlite3 import update_market_data
from solutions.infrastructures.internal_services.market_data_restx_api import app

if __name__ == '__main__':
    t_event = threading.Event()
    t_extern = threading.Thread(target=update_market_data, args=(t_event, 60))
    t_extern.start()
    t_intern = threading.Thread(target=app.run, kwargs={'debug': False})
    t_intern.start()

    t_extern.join()
    t_intern.join()
```

__init__.py

market_data_restx_api.py

```
import sqlite3

from flask import Flask
from flask_restx import Resource, fields, Api


app: Flask = Flask(__name__)

api = Api(app, version='1.0', title='Market Data API',
          description='API pour récupérer les données de marché')

ns = api.namespace('marketdata',
                   description='Endpoints pour les données de marché')

market_data_model = api.model('MarketData', {
    'symbol': fields.String(required=True,
                            description="Le symbole de l'entreprise"),
    'price': fields.Float(required=True,
                          description="Le prix actuel de l'action"),
    'volume': fields.Integer(required=True,
                             description="Le volume d'actions échangées")
})


@ns.route('/<string:symbol>')
class MarketData(Resource):
    name = 'Market Data'

    @classmethod
    def get_market_data(cls, symbol):
        conn = sqlite3.connect('market_data.db')
        try:
            c = conn.cursor()
            c.execute(f"SELECT * FROM market_data WHERE symbol = '{symbol}' "
                      "ORDER BY timestamp DESC LIMIT 1")
            data = c.fetchone()
        finally:
            conn.close()
        if not data:
            return None
```

```
        return {'symbol': data[0], 'price': data[1], 'volume': data[2]}

    def get(self, symbol):
        print(self.name)
        data = self.get_market_data(symbol)
        if not data:
            return {'message': f'Aucune donnée pour le symbole {symbol}'}, 404
        return data, 200


if __name__ == '__main__':
    app.run(debug=True)
```

market_data_sqlite3.py

```
import random
import sqlite3
import threading
import time

lock = threading.Lock()

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():
    with lock:
        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:
            c.execute(f"INSERT INTO market_data VALUES "
                      f"('{data['symbol']}', {data['price']}, "
                      f"{data['volume']}, (strftime('%s', 'now') * 1000))")
            conn.commit()
            time.sleep(random.uniform(0.1, 1))
        print(f"update in {waiting_time} seconds...")
        event.wait(waiting_time)


if __name__ == '__main__':
    t_event = threading.Event()
    t = threading.Thread(target=update_market_data, args=(t_event, 60))
    t.start()
    t.join()
```

__init__.py

__init__.py

__init__.py

contexts.py

```
import asyncio
from datetime import datetime

from solutions.applications.portfolios.contexts import PortfolioContext
from solutions.applications.market_data.events import MarketDataFetched
```

```
from solutions.applications.portfolios.factories import MarketDataFactory
from solutions.applications.portfolios.repositories import PortfolioRepository
from solutions.applications.risks.contexts import RiskManagementContext
from solutions.domains.services.market_data import MarketDataService
from solutions.utils.loggers import logger


# Contexte global ApplicationContext
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} on "
            f"{datetime.fromtimestamp(event.timestamp)}")

    async def run(self):
        await self.portfolio_context.fetch_market_data()
        portfolio_risk = \
            self.risk_management_context.calculate_portfolio_risk()
        logger.info(f"Portfolio risk: {portfolio_risk}")


if __name__ == '__main__':
    asyncio.run(ApplicationContext().run())
```

events.py

```
from solutions.domains.models.values import Price, Symbol, Volume
from solutions.utils.datetimes import utc_now_tsp
```

```
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
```

__init__.py

repositories.py

```
import aiohttp

from solutions.applications.portfolios.factories import MarketDataFactory


class MarketDataRepository:
    @classmethod
    async def get_market_data(cls, session, symbol):
        url = f"http://localhost:5000/marketdata/{symbol}"
        async with session.get(url) as response:
            if response.status != 200:
                print(response)
                # Handle error response here
                return
            try:
                response_json = await response.json()
            except aiohttp.ContentTypeError:
                print(response_json)
                # Handle unexpected response here
                return
            return MarketDataFactory.create_market_data(
                symbol=symbol,
                price=response_json['price'],
                volume=response_json['volume'])
```

__init__.py

factories.py

```
from solutions.domains.models.entities import MarketData
from solutions.domains.models.values import Price, Symbol, Volume


class MarketDataFactory:
    @classmethod
    def create_market_data(cls, symbol: str, price: float, volume: float):
        return MarketData(symbol=Symbol(symbol), price=Price(price),
                          volume=Volume(volume))
```

repositories.py

```
from solutions.domains.models.aggregates import Portfolio
from solutions.domains.models.entities import Asset
from solutions.domains.models.values import Symbol, Weight
from solutions.utils.loggers import logger


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

    @property
    def portfolio(self):
        return self._portfolio

    @property
    def db_connector(self):
        return self._db_connector

    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))])

    def find(self, *args, **kwargs):
        ...

    def add(self, *args, **kwargs):
```

```
        ...

    def remove(self, *args, **kwargs):
        ...

    def save(self, *args, **kwargs):
        ...
```

contexts.py

```
from solutions.applications.market_data.events import MarketDataFetched
from solutions.applications.portfolios.factories import MarketDataFactory
from solutions.applications.portfolios.repositories import PortfolioRepository
from solutions.domains.models.values import Symbol
from solutions.domains.services.market_data import MarketDataService


# Contexte borné PortfolioContext
class PortfolioContext:
    def __init__(self, portfolio_repository: PortfolioRepository,
                 market_data_service: MarketDataService,
                 market_data_factory: MarketDataFactory,
                 market_data_fetched_handler):
        self.portfolio_repository = portfolio_repository
        self.market_data_service = market_data_service
        self.market_data_factory = market_data_factory
        self.portfolio = None
        self.market_data = {}
        self._market_data_fetched_handler = market_data_fetched_handler

    async def fetch_market_data(self):
        self.portfolio = self.portfolio_repository.portfolio
        market_data = \
            await self.market_data_service.get_market_data_for_assets(
                self.portfolio.assets)
        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):
        return self.market_data[symbol.name]

    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()
```

__init__.py

contexts.py

```
from solutions.applications.portfolios.contexts import PortfolioContext


# Contexte borné RiskManagementContext
class RiskManagementContext:
    def __init__(self, portfolio_context: PortfolioContext):
        self.portfolio_context = portfolio_context

    def calculate_portfolio_risk(self) -> float:
        return self.portfolio_context.calculate_risk()
```