1: tests/conftest.py

```python
import sys
from pathlib import Path

TEST_DIR = Path(__file__)
ROOT_DIR = TEST_DIR.parents[1]

sys.path.append(str(ROOT_DIR))
```

2: tests/__init__.py

3: tests/domains/__init__.py

4: tests/domains/models/__init__.py

5: tests/domains/models/test_aggregates.py

```python
from solutions.domains.models.aggregates import Portfolio
from solutions.domains.models.entities import Asset
from solutions.domains.models.values import Symbol, Weight


class TestPortfolio:
    def test_total_weight(self):
        symbol1 = Symbol("AAPL")
        symbol2 = Symbol("GOOGL")
        weight1 = Weight(0.5)
        weight2 = Weight(0.3)
        assets = [Asset(symbol=symbol1, weight=weight1),
                  Asset(symbol=symbol2, weight=weight2)]
        portfolio = Portfolio(assets=assets)
        assert portfolio.total_weight() == weight1.value + weight2.value
```

6: tests/domains/models/test_entities.py

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

```
class TestAsset:
    def test_init(self):
        symbol = Symbol("AAPL")
        weight = Weight(0.5)
        asset = Asset(symbol=symbol, weight=weight)
        assert asset.symbol == symbol
        assert asset.weight == weight

    def test_equality(self):
        symbol = Symbol("AAPL")
        weight = Weight(0.5)
        asset = Asset(symbol=symbol, weight=weight)
        asset2 = Asset(symbol=symbol, weight=weight)
        assert asset != asset2


class TestMarketData:
    def test_init(self):
        symbol = Symbol('AAPL')
        price = Price(100.0)
        volume = Volume(100.0)
        md = MarketData(symbol=symbol, price=price, volume=volume)
        assert md.symbol == symbol
        assert md.price == price
        assert md.volume == volume

    def test_equality(self):
        symbol = Symbol('AAPL')
        price = Price(100.0)
        volume = Volume(100.0)
        md = MarketData(symbol=symbol, price=price, volume=volume)
        md2 = MarketData(symbol=symbol, price=price, volume=volume)
        assert md != md2
```

7: tests/domains/models/test_values.py

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


class TestSymbol:
    def test_init(self):
        symbol = Symbol('AAPL')
        assert symbol.name == 'AAPL'

    def test_equality(self):
        symbol1 = Symbol('AAPL')
        symbol2 = Symbol('AAPL')
        symbol3 = Symbol('GOOGL')
        assert symbol1 == symbol2
        assert symbol1 != symbol3


class TestPrice:
    def test_init(self):
        price = Price(100.0)
        assert price.value == 100.0

    def test_equality(self):
        price1 = Price(100.0)
        price2 = Price(100.0)
        price3 = Price(200.0)
        assert price1 == price2
        assert price1 != price3


class TestVolume:
    def test_init(self):
        volume = Volume(100.0)
        assert volume.value == 100.0

    def test_equality(self):
        volume1 = Volume(100.0)
        volume2 = Volume(100.0)
        volume3 = Volume(200.0)
        assert volume1 == volume2
        assert volume1 != volume3


class TestWeight:
    def test_init(self):
        weight = Weight(0.5)
        assert weight.value == 0.5

    def test_equality(self):
        weight1 = Weight(0.5)
        weight2 = Weight(0.5)
        weight3 = Weight(0.7)
        assert weight1 == weight2
        assert weight1 != weight3
```

8: tests/domains/services/__init__.py

9: tests/domains/services/test_market_data.py

```
import pytest

from solutions.applications.market_data.repositories import \
    MarketDataRepository
from solutions.domains.models.entities import MarketData, Asset
from solutions.domains.services.market_data import MarketDataService
from solutions.domains.models.values import Price, Symbol, Volume, Weight


class TestMarketDataService:

    @pytest.mark.asyncio
    async def test_get_market_data_for_assets(self, monkeypatch):
        # Define a list of Asset objects
        assets = [
            Asset(symbol=Symbol("AAPL"), weight=Weight(0.5)),
            Asset(symbol=Symbol("GOOGL"), weight=Weight(0.5))
        ]

        # Define a list of MarketData objects to be returned by the mock
        market_data_list = [
            MarketData(symbol=Symbol("AAPL"), price=Price(120),
                       volume=Volume(11000)),
            MarketData(symbol=Symbol("GOOGL"), price=Price(250),
                       volume=Volume(22000))
        ]

        # Define a mock for the get_market_data method
        async def mock_get_market_data(session, sym_name):
            print(session, sym_name)
            for market_data in market_data_list:
                if market_data.symbol.name == sym_name:
                    return market_data
        monkeypatch.setattr(MarketDataRepository, 'get_market_data',
                            mock_get_market_data)
        # Create an instance of MarketDataService
        market_data_service = MarketDataService()

        # Call the get_market_data_for_assets method and get the result
        result = await market_data_service.get_market_data_for_assets(assets)

        # Check that the result is a list of MarketData objects
        assert isinstance(result, list)
        assert all(isinstance(item, MarketData) for item in result)

        # Check that the result contains the expected MarketData objects
        for asset in assets:
            symbol_name = asset.symbol.name
            expected_market_data = next(
                md for md in market_data_list if md.symbol.name == symbol_name)
            actual_market_data = next(
                md for md in result if md.symbol.name == symbol_name)
            assert actual_market_data == expected_market_data
```

10: tests/utils/test_numbers.py

```
from solutions.utils.numbers import get_id


def test_get_id():
    gen = get_id()
    uuid1 = next(gen)
    uuid2 = next(gen)
    assert uuid1 != uuid2, "Generated UUIDs should be different"
```

11: tests/utils/__init__.py

12: tests/utils/test_loggers.py

13: tests/utils/test_datetimes.py

```
import datetime

from solutions.utils.datetimes import monotonic_time, utc_now


def test_utc_now():
    assert isinstance(utc_now, datetime.datetime), \
        "utc_now should be a datetime object"
    assert utc_now.tzinfo == datetime.timezone.utc, \
        "utc_now should be timezone-aware and set to UTC"


def test_monotonic_time():
    assert isinstance(monotonic_time, float), \
        "monotonic_time should be a float"
```

14: tests/infrastructures/__init__.py

15: tests/infrastructures/internal_services/test_market_data_restx_api.py

```
from unittest.mock import patch, MagicMock

import pytest

from solutions.infrastructures.internal_services.market_data_restx_api import \
    app, MarketData


class TestWebServices:
    @pytest.fixture(scope='module')
    def client(self):
        with app.test_client() as client:
            yield client

    @patch('sqlite3.connect')
    def test_get_market_data_with_data(self, mock_connect):
        mock_cursor = MagicMock()
        mock_data = ('AAPL', 134.5, 100000)
        mock_cursor.fetchone.return_value = mock_data
        mock_connect.return_value.cursor.return_value = mock_cursor

        data = MarketData.get_market_data('AAPL')
        assert data == {'symbol': 'AAPL', 'price': 134.5, 'volume': 100000}

        mock_connect.assert_called_once_with('market_data.db')
        mock_cursor.execute.assert_called_once_with(
            "SELECT * FROM market_data WHERE symbol = 'AAPL' "
            "ORDER BY timestamp DESC LIMIT 1")
        mock_cursor.fetchone.assert_called_once_with()

    @patch('sqlite3.connect')
    def test_get_market_data_without_data(self, mock_connect):
        mock_cursor = MagicMock()
        mock_data = None
        mock_cursor.fetchone.return_value = mock_data
        mock_connect.return_value.cursor.return_value = mock_cursor

        data = MarketData.get_market_data('MSFT')
        assert data == mock_data

        mock_connect.assert_called_once_with('market_data.db')
        mock_cursor.execute.assert_called_once_with(
            "SELECT * FROM market_data WHERE symbol = 'MSFT' "
            "ORDER BY timestamp DESC LIMIT 1")
        mock_cursor.fetchone.assert_called_once_with()

    def test_get(self, client, monkeypatch):
        mock_data = {'symbol': 'AAPL', 'price': 134.5, 'volume': 100000}
        mock_get_market_data = MagicMock(return_value=mock_data)

        monkeypatch.setattr(MarketData, 'get_market_data',
                            mock_get_market_data)

        response = client.get('/internal/marketdata/AAPL')
        assert response.status_code == 200
        assert response.json == mock_data

        mock_get_market_data.assert_called_once_with('AAPL')

    def test_get_no_data(self, monkeypatch, client):
        mock_get_market_data = MagicMock(return_value=None)

        monkeypatch.setattr(MarketData, 'get_market_data',
                            mock_get_market_data)

        response = client.get('/internal/marketdata/MSFT')
        assert response.status_code == 404
        assert response.json == {
            'message': 'Aucune donnée pour le symbole MSFT. '
                       'You have requested this URI [/internal/'
                       'marketdata/MSFT] but did you mean /'
                       'internal/marketdata/<string:symbol> ?'}

        mock_get_market_data.assert_called_once_with('MSFT')
```

16: tests/infrastructures/internal_services/__init__.py

17: tests/infrastructures/external_services/test_market_data_sqlite3.py

```
from unittest.mock import Mock

import pytest

from solutions.infrastructures.external_services import \
    market_data_sqlite3 as bases


@pytest.fixture(scope='function')
def mock_sqlite3(monkeypatch):
    conn_mock = Mock()
    cursor_mock = Mock()
    conn_mock.cursor.return_value = cursor_mock
    monkeypatch.setattr('sqlite3.connect', lambda x: conn_mock)
    return conn_mock


def test_create_market_data(mock_sqlite3):
    bases.create_market_data()

    mock_sqlite3.cursor.assert_called_once_with()
    mock_sqlite3.cursor().execute.assert_called_once_with('''
            CREATE TABLE IF NOT EXISTS market_data (
                symbol TEXT,
                price REAL,
                volume INTEGER,
                timestamp BIGINT
            )
        ''')
    mock_sqlite3.commit.assert_called_once_with()
    mock_sqlite3.close.assert_called_once_with()


def test_update_market_data(monkeypatch):
    def mock_create_market_data():
        pass

    def mock_sleep(seconds):
        print(seconds)

    monkeypatch.setattr(bases, 'create_market_data', mock_create_market_data)
    monkeypatch.setattr(bases.time, 'sleep', mock_sleep)

    event = bases.threading.Event()

    def mock_update_market_data(m_event, waiting_time=0):
        for data in bases.market_data_list:
            print(m_event, waiting_time, data)

    monkeypatch.setattr(bases, 'update_market_data', mock_update_market_data)

    test_thread = bases.threading.Thread(target=bases.update_market_data,
                                         args=(event,))
    test_thread.start()
    bases.time.sleep(0.1)
    event.set()
    test_thread.join()
```

18: tests/infrastructures/external_services/__init__.py

19: tests/applications/__init__.py

20: tests/applications/globals/__init__.py

21: tests/applications/globals/test_contexts.py

```
import pytest
```

```
from solutions.applications.globals.contexts import ApplicationContext
from solutions.applications.portfolios.factories import MarketDataFactory
from solutions.applications.portfolios.repositories import PortfolioRepository
from solutions.applications.risks.contexts import PortfolioContext, \
    RiskManagementContext
from solutions.domains.services.market_data import MarketDataService


class TestApplicationContext:
    @pytest.fixture
    def application_context(self):
        return ApplicationContext()

    def test_portfolio_repository(self, application_context):
        assert isinstance(application_context.portfolio_repository,
                          PortfolioRepository)

    def test_market_data_repository(self, application_context):
        assert isinstance(application_context.market_data_repository,
                          MarketDataService)

    def test_market_data_factory(self, application_context):
        assert isinstance(application_context.market_data_factory,
                          MarketDataFactory)

    def test_market_data_fetched_handler(self, application_context):
        assert callable(application_context.market_data_fetched_handler)

    def test_portfolio_context(self, application_context):
        assert isinstance(application_context.portfolio_context,
                          PortfolioContext)

    def test_risk_management_context(self, application_context):
        assert isinstance(application_context.risk_management_context,
                          RiskManagementContext)
```

22: tests/applications/market_data/__init__.py

23: tests/applications/market_data/test_repositories.py

```
from unittest.mock import AsyncMock, MagicMock
```

```
import aiohttp
import pytest
from aiohttp import ContentTypeError, ClientResponse

from solutions.applications.market_data.repositories import \
    MarketDataRepository


class TestMarketDataRepository:
    @pytest.mark.asyncio
    async def test_get_market_data_error(self):
        # Create a mock response object with status code 404
        response = AsyncMock(status=404)
        response.json = AsyncMock(return_value={})

        # Create a mock session object that returns the response object
        session = AsyncMock(spec=aiohttp.ClientSession)
        async with self.context_manager_mock() as manager_mock:
            manager_mock.__aenter__.return_value = response
            session.get = AsyncMock(return_value=manager_mock)

            # Call the get_market_data method with the mock session & a symbol
            result = await MarketDataRepository.get_market_data(session,
                                                                "AAPL")

            # Assert that the result is None
            assert result is None

    @pytest.mark.asyncio
    async def test_get_market_data_content_type_error(self):
        # Create a mock response object with content type 'text/html'
        response = MagicMock()
        response.status = 200
        response.json = AsyncMock(
            side_effect=ContentTypeError(response,
                                         (MagicMock(spec=ClientResponse),)))

        # Create a mock session object that returns the response object
        session = AsyncMock(spec=aiohttp.ClientSession)
        async with self.context_manager_mock() as manager_mock:
            manager_mock.__aenter__.return_value = response
            session.get = AsyncMock(return_value=manager_mock)
        # Call the get_market_data method with the mock session & a symbol
        result = await MarketDataRepository.get_market_data(session,
                                                             "AAPL")

        # Assert that the result is None
        assert result is None

    @pytest.mark.asyncio
    async def test_get_market_data_expected_value(self):
        # Create a mock response
        response = MagicMock(status=200)
        response.json = AsyncMock(
            return_value={"symbol": "AAPL", "price": 200.0})

        # Create a mock session
        session = AsyncMock(spec=aiohttp.ClientSession)
        async with self.context_manager_mock() as manager_mock:
            manager_mock.__aenter__.return_value = response
            session.get = AsyncMock(return_value=manager_mock)

            # Call the get_market_data method with the mock session & a symbol
            result = await MarketDataRepository.get_market_data(session,
                                                                 "AAPL")

            # Assert that the function returned the expected result
            assert result == {"symbol": "AAPL", "price": 200.0}

    @classmethod
    def context_manager_mock(cls):
        # Create a mock context manager object
        manager_mock = MagicMock()

        # Add an __aenter__ method that returns the object itself
        manager_mock.__aenter__ = AsyncMock(return_value=manager_mock)

        # Add an __aexit__ method that does nothing
        manager_mock.__aexit__ = AsyncMock()

        return manager_mock
```

24: tests/applications/market_data/test_events.py

25: tests/applications/portfolios/__init__.py

26: tests/applications/portfolios/test_repositories.py

```
from unittest.mock import Mock

from solutions.applications.portfolios.repositories import PortfolioRepository
from solutions.domains.models.aggregates import Portfolio
from solutions.domains.models.values import Symbol, Weight


class TestPortfolioRepository:
    def test_init(self):
        portfolio_repository = PortfolioRepository()
        assert len(portfolio_repository.portfolio.assets) == 2

    def test_get(self, monkeypatch):
        # Mock the db_connector to return a dummy value
        mock_db_connector = Mock(return_value='dummy value')
        monkeypatch.setattr(PortfolioRepository, "db_connector",
                            mock_db_connector)

        # Create the repository instance
        repo = PortfolioRepository()

        # Call the get method and check that it returns the expected portfolio
        portfolio = repo.get()
        assert isinstance(portfolio, Portfolio)
        assert len(portfolio.assets) == 2
        assert portfolio.assets[0].symbol == Symbol('AAPL')
        assert portfolio.assets[0].weight == Weight(0.5)
        assert portfolio.assets[1].symbol == Symbol('GOOGL')
        assert portfolio.assets[1].weight == Weight(0.5)
```

27: tests/applications/portfolios/test_contexts.py

```
from unittest.mock import Mock

import pytest

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 PortfolioContext
from solutions.domains.models.aggregates import Portfolio
from solutions.domains.models.entities import Asset, MarketData
from solutions.domains.models.values import Price, Symbol, Volume, Weight
from solutions.domains.services.market_data import MarketDataService


class TestPortfolioContext:
    @pytest.fixture
    def portfolio_repository(self):
        return PortfolioRepository()

    @pytest.fixture
    def market_data_service(self):
        return Mock(spec=MarketDataService)

    @pytest.fixture
    def market_data_factory(self):
        return Mock(spec=MarketDataFactory)

    @pytest.fixture
    def market_data_fetched_handler(self):
        return Mock(spec=MarketDataFetched)

    @pytest.fixture
    def portfolio_context(self, portfolio_repository, market_data_service,
                          market_data_factory, market_data_fetched_handler):
        return PortfolioContext(
            portfolio_repository=portfolio_repository,
            market_data_service=market_data_service,
            market_data_factory=market_data_factory,
            market_data_fetched_handler=market_data_fetched_handler
        )

    @pytest.mark.asyncio
    async def test_fetch_market_data(self, portfolio_context,
                                     market_data_service,
                                     market_data_fetched_handler):
        # Create mock market data responses
        aapl_market_data = Mock(spec=MarketData)
        aapl_market_data.symbol = Symbol("AAPL")
        aapl_market_data.price = Price(200.0)
        aapl_market_data.volume = Volume(500.0)
        googl_market_data = Mock(spec=MarketData)
        googl_market_data.symbol = Symbol("GOOGL")
        googl_market_data.price = Price(300.0)
        googl_market_data.volume = Volume(600.0)

        # Set up mock market data service to return the mock market data
        # responses
        market_data_service.get_market_data_for_assets.return_value = [
            aapl_market_data,
            googl_market_data,
        ]

        # Call the fetch_market_data method to test
        await portfolio_context.fetch_market_data()

        # Check that the market data was fetched and stored correctly
        assert portfolio_context.get_market_data(
            Symbol("AAPL")) == aapl_market_data
        assert portfolio_context.get_market_data(
            Symbol("GOOGL")) == googl_market_data
        assert len(portfolio_context.market_data) == 2
        assert market_data_fetched_handler.call_count == 2

    def test_get_market_data(self, portfolio_context):
        portfolio_context.market_data = {
            'AAPL': MarketData(symbol=Symbol('AAPL'), price=Price(130.0),
                               volume=Volume(1000)),
            'GOOGL': MarketData(symbol=Symbol('GOOGL'), price=Price(2500.0),
                                volume=Volume(500))
        }
        assert portfolio_context.get_market_data(Symbol('AAPL')) == \
               portfolio_context.market_data['AAPL']
        assert portfolio_context.get_market_data(Symbol('GOOGL')) == \
               portfolio_context.market_data['GOOGL']

    def test_calculate_risk(self, portfolio_context):
        portfolio_context.portfolio = Portfolio(assets=[
            Asset(symbol=Symbol('AAPL'), weight=Weight(0.5)),
            Asset(symbol=Symbol('GOOGL'), weight=Weight(0.5))
        ])
        portfolio_context.market_data = {
            'AAPL': MarketData(symbol=Symbol('AAPL'), price=Price(130.0),
                               volume=Volume(1000)),
            'GOOGL': MarketData(symbol=Symbol('GOOGL'), price=Price(2500.0),
                    volume=Volume(500))
        }
        assert portfolio_context.calculate_risk() == pytest.approx(2.565,
                                                                rel=1e-4)

    def test_total_weight(self, portfolio_context):
        portfolio_context.portfolio = Portfolio(assets=[
            Asset(symbol=Symbol('AAPL'), weight=Weight(0.5)),
            Asset(symbol=Symbol('GOOGL'), weight=Weight(0.5))
        ])
        assert portfolio_context.total_weight() == 1.0
```

28: tests/applications/portfolios/test_factories.py

```
from solutions.applications.portfolios.factories import MarketDataFactory


class TestMarketDataFactory:
    def test_create_market_data(self):
        symbol = "AAPL"
        price = 100.0
        volume = 200.0
        market_data = MarketDataFactory.create_market_data(
            symbol=symbol, price=price, volume=volume)
        assert market_data.symbol.name == symbol
        assert market_data.price.value == price
        assert market_data.volume.value == volume
```

29: tests/applications/portfolios/test_events.py

```
from solutions.applications.market_data.events import MarketDataFetched
from solutions.domains.models.values import Price, Symbol, Volume


class TestMarketDataFetched:
    def test_market_data_fetched(self):
        symbol = Symbol('AAPL')
        price = Price(100)
        volume = Volume(1000)
        event = MarketDataFetched(symbol, price, volume)
        assert event.symbol == symbol
        assert event.price == price
        assert event.volume == volume
```

30: tests/applications/risks/__init__.py

31: tests/applications/risks/test_contexts.py

```python
from unittest.mock import Mock

import pytest

from solutions.applications.portfolios.factories import MarketDataFactory
from solutions.applications.portfolios.repositories import PortfolioRepository
from solutions.applications.risks.contexts import PortfolioContext, \
    RiskManagementContext
from solutions.domains.models.aggregates import Portfolio
from solutions.domains.models.entities import Asset, MarketData
from solutions.domains.models.values import Price, Symbol, Volume, Weight
from solutions.domains.services.market_data import MarketDataService


class TestRiskManagementContext:
    @pytest.fixture
    def market_data_service(self):
        return Mock(spec=MarketDataService)

    @pytest.fixture
    def market_data_factory(self):
        return Mock(spec=MarketDataFactory)

    @pytest.fixture
    def portfolio_repository(self):
        return Mock(spec=PortfolioRepository)

    @pytest.fixture
    def portfolio_context(self, portfolio_repository, market_data_service,
                          market_data_factory):
        return PortfolioContext(
            portfolio_repository=portfolio_repository,
            market_data_service=market_data_service,
            market_data_factory=market_data_factory,
            market_data_fetched_handler=None)

    @pytest.fixture
    def risk_management_context(self, portfolio_context):
```

```
    return RiskManagementContext(portfolio_context)

def test_calculate_portfolio_risk(self, risk_management_context):
    aapl = Asset(symbol=Symbol('AAPL'), weight=Weight(0.5))
    googl = Asset(symbol=Symbol('GOOGL'), weight=Weight(0.5))
    risk_management_context.portfolio_context.portfolio = Portfolio(
        assets=[aapl, googl])
    risk_management_context.portfolio_context.market_data = {
        'AAPL': MarketData(symbol=Symbol('AAPL'), price=Price(100.0),
                           volume=Volume(200.0)),
        'GOOGL': MarketData(symbol=Symbol('GOOGL'), price=Price(200.0),
                            volume=Volume(100.0))
    }
    assert risk_management_context.calculate_portfolio_risk() == 1.25
```