Solutions TP - Mocking et Fixtures

Structure des fichiers de solution

```
tp_mocking/
├── weather_service.py     # Service météo complet
├── test_weather.py        # Tests complets avec mocks
└── solutions.md           # Ce fichier
```

Solutions complètes

Fichier `weather_service.py` - Version complète

```python
import requests
import json
from datetime import datetime

def get_temperature(city):
    """Récupère la température d'une ville via une API"""
    try:
        url = f"http://api.openweathermap.org/data/2.5/weather"
        params = {
            'q': city,
            'appid': 'fake_api_key',
            'units': 'metric'
        }

        response = requests.get(url, params=params)

        if response.status_code == 200:
            data = response.json()
            return data['main']['temp']
        else:
            return None

    except requests.exceptions.RequestException:
        # Gestion des erreurs réseau
        return None

def save_weather_report(city, filename="weather_log.json"):
    """Récupère la météo et la sauvegarde dans un fichier"""

    # 1. Récupérer la température
    temp = get_temperature(city)
    if temp is None:
        return False
```

```
    # 2. Créer le rapport
    report = {
        'city': city,
        'temperature': temp,
        'timestamp': datetime.now().isoformat()
    }

    # 3. Sauvegarder dans le fichier
    try:
        # Lire le fichier existant
        with open(filename, 'r') as f:
            reports = json.load(f)
    except FileNotFoundError:
        reports = []

    reports.append(report)

    with open(filename, 'w') as f:
        json.dump(reports, f)

    return True
```

Fichier `test_weather.py` - Version complète

```
import unittest
from unittest.mock import patch, Mock, mock_open
import requests.exceptions
from weather_service import get_temperature, save_weather_report

class TestWeather(unittest.TestCase):

    def setUp(self):
        """Fixture : prépare les données avant chaque test"""
        self.sample_weather_data = {
            'main': {
                'temp': 25.5
            }
        }
        self.test_city = "Paris"

    @patch('weather_service.requests.get')
    def test_get_temperature_success(self, mock_get):
        """Premier test avec mock - SOLUTION COMPLÈTE"""

        # 1. Créer un objet Mock pour simuler la réponse HTTP
        fake_response = Mock()
        fake_response.status_code = 200

        # 2. Configurer les données JSON que l'API retournerait
        fake_response.json.return_value = self.sample_weather_data
```

```
        # 3. Configurer le mock pour retourner notre fake_response
        mock_get.return_value = fake_response

        # 4. Tester la fonction
        result = get_temperature(self.test_city)

        # 5. Vérifier le résultat
        self.assertEqual(result, 25.5)

        # 6. Vérifier que requests.get a été appelé correctement
        expected_url = "http://api.openweathermap.org/data/2.5/weather"
        expected_params = {
            'q': self.test_city,
            'appid': 'fake_api_key',
            'units': 'metric'
        }
        mock_get.assert_called_once_with(expected_url, params=expected_params)

    @patch('weather_service.requests.get')
    def test_get_temperature_city_not_found(self, mock_get):
        """Test quand la ville n'existe pas - SOLUTION"""

        # Créer un Mock qui retourne status_code = 404
        fake_response = Mock()
        fake_response.status_code = 404

        # Configurer mock_get.return_value
        mock_get.return_value = fake_response

        # Tester get_temperature("VilleInexistante")
        result = get_temperature("VilleInexistante")

        # Vérifier que le résultat est None
        self.assertIsNone(result)

        # Vérifier que l'appel API a bien été fait
        mock_get.assert_called_once()

    @patch('weather_service.requests.get')
    def test_get_temperature_network_error(self, mock_get):
        """Test quand il y a une erreur réseau - SOLUTION"""

        # Configurer le mock pour lever une exception
        mock_get.side_effect = requests.exceptions.RequestException("Network
error")

        # Tester que la fonction gère l'exception
        result = get_temperature("Paris")

        # Vérifier que la fonction retourne None en cas d'erreur
        self.assertIsNone(result)

        # Vérifier que l'exception a bien été gérée (pas de crash)
        mock_get.assert_called_once()
```

```
    @patch('weather_service.requests.get')
    def test_multiple_cities(self, mock_get):
        """Test plusieurs villes avec une seule méthode - SOLUTION"""

        cities_and_temps = [
            ("Paris", 25.0),
            ("Londres", 18.5),
            ("Tokyo", 30.2)
        ]

        for city, expected_temp in cities_and_temps:
            with self.subTest(city=city):
                # Configurer le mock pour cette ville
                fake_response = Mock()
                fake_response.status_code = 200
                fake_response.json.return_value = {'main': {'temp':
expected_temp}}
                mock_get.return_value = fake_response

                # Tester get_temperature(city)
                result = get_temperature(city)

                # Vérifier le résultat
                self.assertEqual(result, expected_temp)

class TestWeatherReport(unittest.TestCase):

    def setUp(self):
        """Fixture pour les tests de rapport météo"""
        self.test_city = "Paris"
        self.test_temp = 20.5
        self.fixed_timestamp = "2024-01-01T12:00:00"

    @patch('weather_service.datetime')
    @patch('builtins.open', new_callable=mock_open, read_data='[]')
    @patch('weather_service.get_temperature')
    def test_save_weather_report_success(self, mock_get_temp, mock_file,
mock_datetime):
        """Test sauvegarde rapport météo - SOLUTION COMPLÈTE"""

        # Configurer mock_get_temp pour retourner 20.5
        mock_get_temp.return_value = self.test_temp

        # Configurer mock_datetime pour retourner une date fixe
        mock_datetime.now.return_value.isoformat.return_value =
self.fixed_timestamp

        # Appeler save_weather_report
        result = save_weather_report(self.test_city)

        # Vérifier que le résultat est True
        self.assertTrue(result)
```

```
        # Vérifier que get_temperature a été appelé avec "Paris"
        mock_get_temp.assert_called_once_with(self.test_city)

        # Vérifier que le fichier a été ouvert en lecture puis en écriture
        self.assertEqual(mock_file.call_count, 2)

        # Vérifier les appels de fichier
        calls = mock_file.call_args_list
        self.assertEqual(calls[0][0][0], "weather_log.json")  # Premier appel
(lecture)
        self.assertEqual(calls[0][0][1], "r")
        self.assertEqual(calls[1][0][0], "weather_log.json")  # Deuxième appel
(écriture)
        self.assertEqual(calls[1][0][1], "w")

    @patch('weather_service.datetime')
    @patch('builtins.open', new_callable=mock_open, read_data='[]')
    @patch('weather_service.get_temperature')
    def test_save_weather_report_api_failure(self, mock_get_temp, mock_file,
mock_datetime):
        """Test quand l'API météo échoue - SOLUTION"""

        # Configurer mock_get_temp pour retourner None (échec API)
        mock_get_temp.return_value = None

        # Appeler save_weather_report
        result = save_weather_report(self.test_city)

        # Vérifier que le résultat est False
        self.assertFalse(result)

        # Vérifier que get_temperature a été appelé
        mock_get_temp.assert_called_once_with(self.test_city)

        # Vérifier que le fichier N'A PAS été ouvert (pas de sauvegarde)
        mock_file.assert_not_called()

if __name__ == '__main__':
    unittest.main()
```

Explications détaillées des solutions

Partie 2 : Premier mock (Étape 3)

Solution du test `test_get_temperature_success`

```
@patch('weather_service.requests.get')
def test_get_temperature_success(self, mock_get):
    # 1. Créer un Mock pour la réponse HTTP
    fake_response = Mock()
```

```
        fake_response.status_code = 200
        fake_response.json.return_value = self.sample_weather_data

        # 2. Configurer le mock
        mock_get.return_value = fake_response

        # 3. Tester
        result = get_temperature(self.test_city)

        # 4. Vérifications
        self.assertEqual(result, 25.5)
        mock_get.assert_called_once_with(expected_url, params=expected_params)
```

Explications étape par étape :

1. `@patch('weather_service.requests.get')` :
   - Remplace temporairement `requests.get` par un mock
   - Le mock est injecté comme paramètre `mock_get`
   - **Pourquoi** : Évite l'appel réseau réel
2. `fake_response = Mock()` :
   - Crée un objet factice qui simule une réponse HTTP
   - **Pourquoi** : `requests.get()` retourne un objet Response
3. `fake_response.status_code = 200` :
   - Simule une réponse HTTP réussie
   - **Pourquoi** : Notre fonction teste `if response.status_code == 200`
4. `fake_response.json.return_value = {...}` :
   - Définit ce que retourne `response.json()`
   - **Pourquoi** : Notre fonction appelle `response.json()['main']['temp']`
5. `mock_get.return_value = fake_response` :
   - Fait que `requests.get()` retourne notre fake_response
   - **Pourquoi** : Contrôle total sur ce que reçoit notre fonction
6. **Vérifications avec** `assert_called_once_with()` :
   - Vérifie que l'API a été appelée avec les bons paramètres
   - **Pourquoi** : S'assurer que notre fonction fait bien l'appel attendu

Partie 3 : Tests d'erreur (Étapes 4-5)

Solution du test `test_get_temperature_city_not_found`

```
@patch('weather_service.requests.get')
def test_get_temperature_city_not_found(self, mock_get):
    fake_response = Mock()
    fake_response.status_code = 404
    mock_get.return_value = fake_response

    result = get_temperature("VilleInexistante")

    self.assertIsNone(result)
```

Justification :

- **Cas limite important** : Ville inexistante (erreur 404)
- **Test du comportement** : Fonction doit retourner `None` si status ≠ 200
- **Contrôle des erreurs** : Vérifier que l'application gère les échecs API

Solution du test `test_get_temperature_network_error`

```
@patch('weather_service.requests.get')
def test_get_temperature_network_error(self, mock_get):
    mock_get.side_effect = requests.exceptions.RequestException("Network error")

    result = get_temperature("Paris")

    self.assertIsNone(result)
```

Explications :

- `side_effect` : Fait lever une exception au lieu de retourner une valeur
- **Test de robustesse** : Vérifie que l'application ne crash pas
- **Gestion d'erreur** : Force l'ajout du `try/except` dans le code source

Partie 4 : Fixtures (Étape 6)

Solution du `setUp()`

```
def setUp(self):
    self.sample_weather_data = {
        'main': {'temp': 25.5}
    }
    self.test_city = "Paris"
```

Avantages des fixtures :

- **DRY (Don't Repeat Yourself)** : Évite la duplication de données
- **Maintenance** : Changement centralisé des données de test

- **Lisibilité** : Tests plus courts et focalisés sur la logique

Partie 5 : Fonction complexe (Étapes 7-8)

Solution du test `test_save_weather_report_success`

```
@patch('weather_service.datetime')
@patch('builtins.open', new_callable=mock_open, read_data='[]')
@patch('weather_service.get_temperature')
def test_save_weather_report_success(self, mock_get_temp, mock_file,
mock_datetime):
```

Explications des 3 mocks :

1. `@patch('weather_service.get_temperature')` :
   - Mock la fonction `get_temperature()` utilisée dans `save_weather_report()`
   - **Pourquoi** : Isolation - on ne teste que la logique de sauvegarde
   - **Configuration** : `mock_get_temp.return_value = 20.5`
2. `@patch('builtins.open', new_callable=mock_open, read_data='[]')` :
   - Mock les opérations de fichier (lecture/écriture)
   - `read_data='[]'` : Simule un fichier contenant une liste JSON vide
   - **Pourquoi** : Éviter de créer de vrais fichiers pendant les tests
3. `@patch('weather_service.datetime')` :
   - Mock l'horodatage pour avoir des tests déterministes
   - **Configuration** : `mock_datetime.now().isoformat.return_value = "2024-01-01T12:00:00"`
   - **Pourquoi** : Tests reproductibles avec timestamp fixe

Vérifications multiples :

```
# Vérifier le succès
self.assertTrue(result)

# Vérifier l'appel de dépendance
mock_get_temp.assert_called_once_with(self.test_city)

# Vérifier les opérations fichier
self.assertEqual(mock_file.call_count, 2)  # Lecture puis écriture
```

Partie 6 : Tests paramétrés (Étape 9)

Solution du test `test_multiple_cities`

```python
@patch('weather_service.requests.get')
def test_multiple_cities(self, mock_get):
    cities_and_temps = [("Paris", 25.0), ("Londres", 18.5), ("Tokyo", 30.2)]

    for city, expected_temp in cities_and_temps:
        with self.subTest(city=city):
            # Configuration spécifique pour chaque ville
            fake_response = Mock()
            fake_response.status_code = 200
            fake_response.json.return_value = {'main': {'temp': expected_temp}}
            mock_get.return_value = fake_response

            result = get_temperature(city)
            self.assertEqual(result, expected_temp)
```

Avantages des `subTest` :

- **Isolation** : Un échec sur Paris n'empêche pas de tester Londres
- **Diagnostic** : Indique exactement quelle ville a échoué
- **Efficacité** : Un seul test pour plusieurs cas similaires

---

Bonnes pratiques appliquées

1. **Isolation des tests**

- Chaque test utilise des mocks pour ses dépendances
- Aucun appel réseau réel
- Pas de fichiers créés sur le disque

2. **Tests déterministes**

- Timestamps fixes avec mock de `datetime`
- Réponses API prévisibles
- Résultats reproductibles

3. **Structure des tests**

- **Arrange** : Configuration des mocks et données
- **Act** : Appel de la fonction à tester
- **Assert** : Vérification des résultats et interactions

4. **Couverture complète**

- **Cas nominal** : API fonctionne, fichier existe
- **Cas d'erreur** : API en panne, ville inexistante
- **Cas limites** : Fichier inexistant, erreur réseau

5. **Nommage explicite**

- `test_fonction_cas_specifique`
- Docstrings descriptives
- Variables avec noms clairs (`fake_response`, `expected_temp`)

Comparaison avant/après mocks

❌ Sans mocks (problématique)

```
def test_get_temperature_paris(self):
    temp = get_temperature("Paris")
    self.assertIsNotNone(temp)  # Test faible
```

Problèmes :

- Dépend d'internet
- Résultats variables selon la météo réelle
- Impossible de tester les cas d'erreur
- Lent et peu fiable

☑ Avec mocks (solution)

```
@patch('weather_service.requests.get')
def test_get_temperature_success(self, mock_get):
    fake_response = Mock()
    fake_response.status_code = 200
    fake_response.json.return_value = {'main': {'temp': 25.5}}
    mock_get.return_value = fake_response

    result = get_temperature("Paris")

    self.assertEqual(result, 25.5)
    mock_get.assert_called_once()
```

Avantages :

- Tests rapides et fiables
- Contrôle total des scénarios
- Possibilité de tester tous les cas d'erreur
- Isolation parfaite des dépendances

Extensions possibles

Tests plus avancés

```
    # Test avec retry automatique
    @patch('weather_service.requests.get')
    def test_get_temperature_retry_on_failure(self, mock_get):
        # Premier appel échoue, deuxième réussit
        mock_get.side_effect = [
            requests.exceptions.RequestException(),
            Mock(status_code=200, json=lambda: {'main': {'temp': 22.0}})
        ]

        result = get_temperature_with_retry("Paris", max_retries=2)
        self.assertEqual(result, 22.0)
        self.assertEqual(mock_get.call_count, 2)

    # Test avec cache
    @patch('weather_service.requests.get')
    def test_temperature_caching(self, mock_get):
        mock_response = Mock(status_code=200, json=lambda: {'main': {'temp': 20.0}})
        mock_get.return_value = mock_response

        # Deux appels successifs
        temp1 = get_temperature_cached("Paris")
        temp2 = get_temperature_cached("Paris")

        # L'API ne doit être appelée qu'une fois (cache)
        mock_get.assert_called_once()
        self.assertEqual(temp1, temp2)
```

Ces solutions montrent comment les mocks permettent de tester des comportements complexes de manière simple et fiable.