IMT Atlantique
Technopôle de Brest-Iroise
29238 Brest Cedex 3
www.imt-atlantique.fr

IMT Atlantique – UE CALC

Plateforme d'échecs distribuée avec RabbitMQ

Marion Boulin

Date d'édition : Décembre 2025
Version : 1.0

Sommaire

1. Introduction **1**

2. Service Producteur (producer.py) **2**

2.1. Importation des bibliothèques 2

2.2. Constantes de configuration 2

2.3. Variables globales de contrôle 2

2.4. Publication des événements 3

2.5. Réception des événements 3

2.6. Traitement d'un coup validé 3

2.7. Traitement de la fin de partie 4

2.8. Fonction principale 4

2.9. Connexion et abonnement à RabbitMQ 4

2.10. Boucle principale de jeu 4

2.11. Calcul et proposition d'un coup 5

2.12. Arrêt du service 5

3. Service de Validation (validation_service.py) **6**

3.1. Importation des bibliothèques 6

3.2. Constantes de configuration 6

3.3. Plateau officiel de la partie 6

3.4. Publication des événements 6

3.5. Réception des événements RabbitMQ 7

3.6. Gestion du démarrage d'une nouvelle partie 7

3.7. Réception d'un coup proposé 8

3.8. Vérification de la légalité du coup 8

3.9. Application et diffusion d'un coup validé 8

3.10. Détection de la fin de partie 8

3.11. Connexion et abonnement à RabbitMQ 8

3.12. Lancement du service 9

4. Service d'Analyse (analysis_service.py) **10**

4.1. Importation des bibliothèques 10

4.2. Constantes de configuration 10
4.3. Initialisation du plateau local 10
4.4. Initialisation du moteur Stockfish 10
4.5. Connexion à RabbitMQ 11
4.6. Déclaration de l'exchange 11
4.7. Création et liaison de la file RabbitMQ 11
4.8. Réception et traitement des événements 11
4.9. Gestion du démarrage d'une nouvelle partie 12
4.10. Analyse d'un coup validé 12
4.11. Analyse de la position par Stockfish 12
4.12. Interprétation du score 12
4.13. Détection d'un mat forcé 13
4.14. Évaluation matérielle 13
4.15. Affichage d'une interprétation humaine 13
4.16. Gestion de la fin de partie 13
4.17. Abonnement et démarrage de l'écoute 13

5. Service Spectateur (spectator_service.py) 15
5.1. Importation des bibliothèques 15
5.2. Initialisation et message de démarrage 15
5.3. Constantes de configuration 15
5.4. Plateau local du spectateur 15
5.5. Initialisation de l'affichage graphique 16
5.6. Définition des symboles des pièces 16
5.7. Fonction de dessin du plateau 16
5.7.1. Dessin des cases 16
5.7.2. Dessin des pièces 16
5.8. Mise en forme de l'affichage 16
5.9. Réception des événements RabbitMQ 17
5.10. Décodage et interprétation des événements 17
5.11. Gestion du démarrage d'une nouvelle partie 17
5.12. Affichage d'un coup validé 17
5.13. Gestion de la fin de partie 18

5.14. Fonction principale 18
5.15. Connexion et abonnement à RabbitMQ 18
5.16. Déclaration et liaison de la file 18
5.17. Abonnement aux messages 19
5.18. Démarrage de l'écoute 19
5.19. Gestion des erreurs et arrêt du service 19

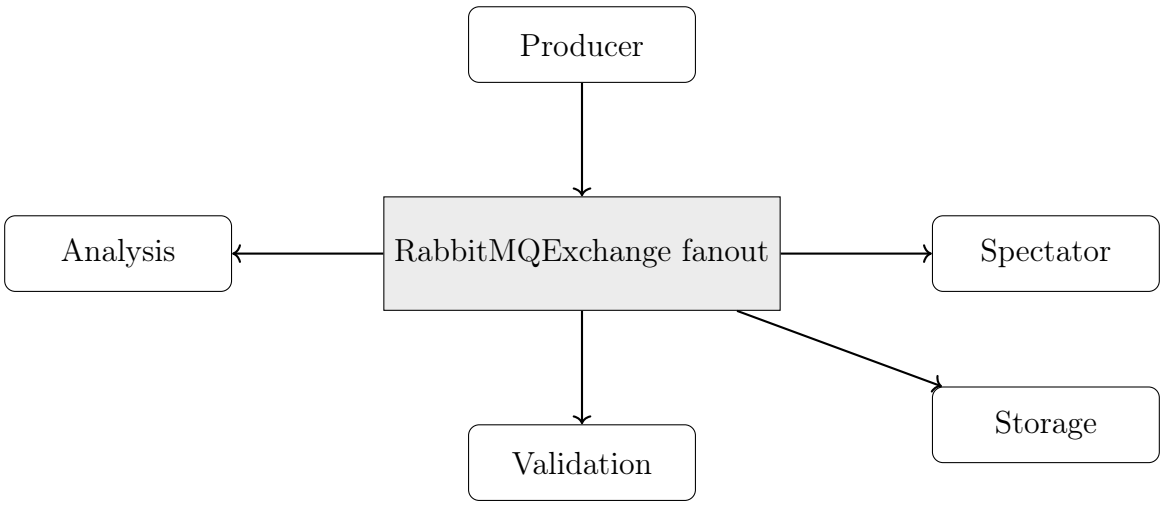
1. Introduction

Dans le cadre de l'évaluation de l'UE CALC, ce projet consiste à mettre en place une application distribuée à l'aide d'un middleware de messages.

Le middleware choisi est RabbitMQ, car il permet une communication simple et fiable entre plusieurs programmes indépendants. Il facilite le découplage des composants, la communication asynchrone et permet de continuer à fonctionner même si un service s'arrête temporairement.

Le cas d'étude retenu est une application d'échecs distribuée. Ce choix permet de mettre en évidence des échanges fréquents entre services, le respect de règles strictes et la synchronisation des données, ce qui correspond bien aux objectifs du projet. Et c'est aussi un jeu que j'affectionne.

L'application est composée de plusieurs services indépendants, chacun ayant un rôle précis. Tous les échanges passent par RabbitMQ :
- Le Producer envoie les coups proposés.
- Le service Validation reçoit ces coups, vérifie qu'ils sont légaux et diffuse les coups validés.
- Les services Analysis, Spectator et Storage reçoivent ensuite ces informations pour analyser la partie, l'afficher et la sauvegarde



2. Service Producteur (producer.py)

Le service `Producer` est responsable de la génération automatique des coups d'échecs dans le système distribué. Il utilise le moteur Stockfish afin de calculer les meilleurs coups possibles, puis les propose au reste du système par l'intermédiaire d'événements RabbitMQ. Il est important de souligner que ce service ne possède aucun rôle décisionnel concernant la légalité des coups ou la fin de la partie, ces responsabilités étant exclusivement assurées par le service de validation.

Le code analysé correspond au fichier `producer.py` fourni dans le projet turn0file4.

2.1. Importation des bibliothèques

Le code commence par l'importation des bibliothèques nécessaires au fonctionnement du service, comme illustré dans l'extrait suivant.

```
import pika
import json
import time
import chess
import chess.engine
from datetime import datetime, timezone
```

La bibliothèque `pika` permet la communication avec RabbitMQ. La bibliothèque `json` est utilisée pour la sérialisation des messages. Le module `chess` fournit la gestion du plateau et des coups, tandis que `chess.engine` permet l'interfaçage avec Stockfish. Enfin, `datetime` est utilisé afin d'horodater précisément chaque événement publié.

2.2. Constantes de configuration

Les paramètres globaux du service sont définis sous forme de constantes.

```
RABBITMQ_HOST = "localhost"
EXCHANGE_NAME = "chess.events"
STOCKFISH_PATH = r"C:\...\stockfish.exe"
```

Ces constantes définissent respectivement l'adresse du serveur RabbitMQ, le nom de l'exchange commun à l'ensemble des services, ainsi que le chemin absolu vers l'exécutable Stockfish utilisé pour le calcul des coups.

2.3. Variables globales de contrôle

Le service maintient un état interne minimal, illustré ci-dessous.

```
board = chess.Board()
game_over = False
waiting_validation = False
```

Le plateau `board` est strictement local au Producer et ne représente pas l'état officiel de la partie. La variable `game_over` indique si la partie est terminée, tandis que

`waiting_validation` empêche le service de proposer plusieurs coups successifs sans validation intermédiaire.

2.4. Publication des événements

La fonction `publish` permet de diffuser des événements vers RabbitMQ.

```
def publish(channel, event_type, payload):
    message = {
        "event_type": event_type,
        "timestamp": datetime.now(timezone.utc).isoformat(),
        "payload": payload
    }

    channel.basic_publish(
        exchange=EXCHANGE_NAME,
        routing_key="",
        body=json.dumps(message)
    )
```

Chaque message contient un type d'événement, un horodatage UTC et une charge utile. L'utilisation d'un exchange de type `fanout` garantit que tous les services reçoivent chaque événement publié.

2.5. Réception des événements

La fonction `on_message` est enregistrée comme callback RabbitMQ.

```
def on_message(ch, method, properties, body):
    event = json.loads(body)
    event_type = event.get("event_type")
    payload = event.get("payload", {})
```

Cette fonction est invoquée automatiquement lors de la réception d'un message. Elle extrait le type d'événement ainsi que les données associées.

2.6. Traitement d'un coup validé

Lorsqu'un événement `move_validated` est reçu, le coup est appliqué au plateau local.

```
if event_type == "move_validated":
    move = chess.Move.from_uci(payload["uci"])
    board.push(move)
    waiting_validation = False
```

Cette opération synchronise le plateau local du Producer avec l'état officiel validé par le service d'arbitrage.

2.7. Traitement de la fin de partie

La fin de partie est signalée par l'événement `game_ended`.

```
elif event_type == "game_ended":
    print("Partie termin e :", payload.get("result"))
    game_over = True
```

À partir de ce moment, la boucle principale du Producer est interrompue.

2.8. Fonction principale

La fonction `main` constitue le point d'entrée du service.

```
def main():
    engine = chess.engine.SimpleEngine.popen_uci(STOCKFISH_PATH)
```

Le moteur Stockfish est démarré via l'interface UCI, puis configuré comme suit.

```
engine.configure({
    "Skill Level": 20,
    "Threads": 4,
    "Hash": 256
})
```

Cette configuration permet d'obtenir un moteur performant exploitant le multithreading et une mémoire de calcul étendue.

2.9. Connexion et abonnement à RabbitMQ

Le service établit une connexion avec le broker RabbitMQ, déclare l'exchange commun de type `fanout`, puis crée une file temporaire exclusive. Cette file est liée à l'exchange afin de recevoir l'ensemble des événements diffusés dans le système. Le service s'abonne ensuite à cette file en enregistrant la fonction `on_message` comme callback de traitement des messages.

2.10. Boucle principale de jeu

Le Producer fonctionne dans une boucle principale tant que la partie n'est pas terminée.

```
while not game_over:
    if waiting_validation:
        channel.connection.process_data_events(time_limit=0.2)
        time.sleep(0.05)
        continue
```

Cette logique garantit que le service reste réactif aux messages RabbitMQ tout en évitant une consommation excessive de ressources.

2.11. Calcul et proposition d'un coup

Lorsque le Producer est autorisé à jouer, il calcule un coup à l'aide de Stockfish.

```
result = engine.play(
    board,
    chess.engine.Limit(time=0.7)
)

move = result.move
publish(channel, "move_proposed", {"uci": move.uci()})
waiting_validation = True
```

Le coup calculé est ensuite proposé au système via un événement `move_proposed`.

2.12. Arrêt du service

À la fin de la partie, le service libère proprement ses ressources.

```
engine.quit()
connection.close()
```

Cette étape garantit un arrêt propre du moteur Stockfish et de la connexion RabbitMQ.

3. Service de Validation (validation_service.py)

Le service de validation joue un rôle central dans l'architecture du système d'échecs distribué. Il agit comme l'unique arbitre officiel de la partie en cours. Contrairement aux autres services, il maintient le plateau de vérité, vérifie la légalité des coups proposés, applique uniquement les coups autorisés et détecte les conditions de fin de partie. Aucun autre service n'est autorisé à modifier l'état officiel du jeu.

Le code analysé correspond au fichier `validation_service.py` fourni dans le projet turn0file3.

3.1. Importation des bibliothèques

Le service commence par importer les bibliothèques nécessaires à son fonctionnement.

```
import pika
import json
import chess
from datetime import datetime, timezone
```

La bibliothèque `pika` permet la communication avec le broker RabbitMQ. La bibliothèque `json` est utilisée pour encoder et décoder les messages échangés entre les services. Le module `chess` fournit les outils nécessaires à la gestion du plateau, à la validation des coups et à la détection des états de fin de partie. Enfin, le module `datetime` permet d'associer un horodatage précis à chaque événement publié.

3.2. Constantes de configuration

Les paramètres globaux du service sont définis sous forme de constantes.

```
RABBITMQ_HOST = "localhost"
EXCHANGE_NAME = "chess.events"
```

Ces constantes indiquent respectivement l'adresse du serveur RabbitMQ et le nom de l'exchange commun utilisé pour la diffusion des événements dans tout le système. L'utilisation d'un exchange unique facilite la synchronisation de l'ensemble des services.

3.3. Plateau officiel de la partie

```
board = chess.Board()
```

Cette variable représente le plateau officiel de la partie. Il s'agit de la seule source de vérité du système. Tous les coups validés sont appliqués sur ce plateau, et aucun autre service ne dispose d'un accès direct permettant de le modifier. Cette approche garantit la cohérence globale de l'état du jeu.

3.4. Publication des événements

La fonction `publish` est responsable de la diffusion des événements validés vers RabbitMQ.

```
def publish(channel, event_type, payload):
    message = {
        "event_type": event_type,
        "timestamp": datetime.now(timezone.utc).isoformat(),
        "payload": payload
    }

    channel.basic_publish(
        exchange=EXCHANGE_NAME,
        routing_key="",
        body=json.dumps(message)
    )
```

Chaque événement publié contient trois éléments essentiels : le type d'événement, un horodatage UTC assurant la traçabilité, et une charge utile contenant les données associées. L'utilisation d'un exchange de type `fanout` permet de diffuser automatiquement chaque événement à tous les services abonnés.

3.5. Réception des événements RabbitMQ

La fonction `on_message` est enregistrée comme callback auprès de RabbitMQ. Elle est invoquée automatiquement à chaque réception d'un message.

```
def on_message(channel, method, properties, body):
    global board

    message = json.loads(body)
    event_type = message.get("event_type")
    payload = message.get("payload", {})
```

Cette fonction commence par décoder le message JSON reçu. Elle extrait ensuite le type d'événement et les données associées. Le mot-clé `global` permet de modifier le plateau officiel partagé par l'ensemble du service.

3.6. Gestion du démarrage d'une nouvelle partie

Lorsqu'un événement `game_started` est reçu, le plateau officiel est réinitialisé.

```
if event_type == "game_started":
    board.reset()
    print("Nouvelle partie valid e")
    return
```

Cette opération garantit que toutes les parties commencent à partir d'un état initial identique. Le retour immédiat empêche tout traitement supplémentaire pour cet événement.

3.7. Réception d'un coup proposé

Lorsque le service reçoit un événement `move_proposed`, il vérifie la légalité du coup proposé.

```
if event_type == "move_proposed":
    move = chess.Move.from_uci(payload["uci"])
```

Le coup est converti depuis son format UCI en un objet manipulable par la bibliothèque `chess`.

3.8. Vérification de la légalité du coup

```
if move not in board.legal_moves:
    print("Coup ill gal ignor  :", payload["uci"])
    return
```

Cette vérification garantit que seules les règles officielles des échecs sont appliquées. Si le coup est illégal, il est ignoré et aucun événement n'est publié, ce qui empêche toute désynchronisation du système.

3.9. Application et diffusion d'un coup validé

Lorsque le coup est légal, il est appliqué au plateau officiel et diffusé aux autres services.

```
board.push(move)
print("Coup valid  :", payload["uci"])
publish(channel, "move_validated", {"uci": payload["uci"]})
```

Cette étape synchronise l'ensemble des services sur le même état de jeu.

3.10. Détection de la fin de partie

Après chaque coup validé, le service vérifie si la partie est terminée.

```
if board.is_checkmate():
    publish(channel, "game_ended", {"result": "checkmate"})
elif board.is_stalemate() or board.can_claim_threefold_repetition()
   :
    publish(channel, "game_ended", {"result": "draw"})
```

Le service est capable de détecter un échec et mat ainsi que certaines conditions de partie nulle. Dans ces cas, un événement `game_ended` est publié afin d'informer tous les autres services.

3.11. Connexion et abonnement à RabbitMQ

La fonction principale établit la connexion avec RabbitMQ et prépare le service à recevoir les événements.

```
connection = pika.BlockingConnection(
    pika.ConnectionParameters(RABBITMQ_HOST)
)

channel = connection.channel()
```

Le service déclare ensuite l'exchange commun et crée une file temporaire exclusive.

```
queue = channel.queue_declare(queue="", exclusive=True).method.
   queue

channel.queue_bind(
    exchange=EXCHANGE_NAME,
    queue=queue
)
```

La fonction on_message est enregistrée comme callback de traitement.

```
channel.basic_consume(
    queue=queue,
    on_message_callback=on_message,
    auto_ack=True
)
```

3.12. Lancement du service

```
print("Validation service pr t , en attente des    vnements   ")
channel.start_consuming()
```

Le service entre alors dans une phase d'écoute permanente et traite les événements tant que la connexion RabbitMQ est active.

4. Service d'Analyse (analysis_service.py)

Le service d'analyse est un service passif du système d'échecs distribué. Son rôle est exclusivement informatif : il rejoue la partie à partir des coups validés, analyse chaque position à l'aide du moteur Stockfish et affiche une évaluation compréhensible pour l'utilisateur. Il ne prend aucune décision, ne valide aucun coup et ne modifie jamais l'état officiel de la partie.

Le code analysé correspond au fichier `analysis_service.py` fourni dans le projet turn0file0.

4.1. Importation des bibliothèques

Le service commence par importer les bibliothèques nécessaires à son fonctionnement.

```
import pika
import json
import chess
import chess.engine
```

La bibliothèque `pika` permet la communication avec le broker RabbitMQ. La bibliothèque `json` est utilisée pour décoder les messages reçus. Le module `chess` permet de maintenir un plateau local et de rejouer les coups. Le module `chess.engine` permet d'interfacer le moteur d'analyse Stockfish.

4.2. Constantes de configuration

Les paramètres globaux du service sont définis sous forme de constantes.

```
RABBITMQ_HOST = "localhost"
EXCHANGE_NAME = "chess.events"
STOCKFISH_PATH = r"C:\...\stockfish.exe"
```

Ces constantes définissent respectivement l'adresse du serveur RabbitMQ, le nom de l'exchange commun utilisé pour la diffusion des événements et le chemin absolu vers l'exécutable Stockfish utilisé pour l'analyse des positions.

4.3. Initialisation du plateau local

```
board = chess.Board()
```

Le plateau `board` est un plateau strictement local au service d'analyse. Il est indépendant du plateau officiel maintenu par le service de validation. Il sert uniquement à rejouer les coups validés afin d'analyser la position courante.

4.4. Initialisation du moteur Stockfish

```
engine = chess.engine.SimpleEngine.popen_uci(STOCKFISH_PATH)
```

Le moteur Stockfish est lancé via l'interface UCI. Il est utilisé uniquement pour analyser les positions et fournir une évaluation numérique ou un mat forcé. Le service d'analyse ne demande jamais à Stockfish de proposer un coup.

4.5. Connexion à RabbitMQ

Le service établit une connexion avec le broker RabbitMQ et crée un canal de communication.

```
connection = pika.BlockingConnection(
    pika.ConnectionParameters(host=RABBITMQ_HOST)
)

channel = connection.channel()
```

Cette connexion permet au service de recevoir les événements diffusés par les autres composants du système.

4.6. Déclaration de l'exchange

```
channel.exchange_declare(
    exchange=EXCHANGE_NAME,
    exchange_type="fanout",
    durable=True
)
```

L'exchange est déclaré avec le type `fanout`. Ce type garantit que chaque événement publié est diffusé à tous les services abonnés, sans filtrage.

4.7. Création et liaison de la file RabbitMQ

```
queue = channel.queue_declare(queue="", exclusive=True).method.
   queue

channel.queue_bind(
    exchange=EXCHANGE_NAME,
    queue=queue
)
```

Le service crée une file temporaire et exclusive. Cette file est automatiquement supprimée lorsque le service s'arrête. Elle est liée à l'exchange afin de recevoir tous les événements diffusés.

4.8. Réception et traitement des événements

La fonction `on_message` est enregistrée comme callback pour le traitement des messages entrants.

```
def on_message(ch, method, properties, body):
    event = json.loads(body)
    event_type = event.get("event_type")
    payload = event.get("payload", {})
```

Cette fonction est appelée automatiquement à chaque réception d'un message RabbitMQ. Elle commence par décoder le message JSON et extraire le type d'événement ainsi que les données associées.

4.9. Gestion du démarrage d'une nouvelle partie

```
if event_type == "game_started":
    board.reset()
    print("Nouvelle partie    analyser")
    return
```

Lorsqu'un événement `game_started` est reçu, le plateau local est réinitialisé. Cela garantit que l'analyse commence toujours à partir d'une position initiale correcte.

4.10. Analyse d'un coup validé

```
if event_type == "move_validated":
    move = chess.Move.from_uci(payload["uci"])
    board.push(move)
```

Le coup validé est converti depuis son format UCI puis appliqué au plateau local. Cette opération permet au service de maintenir une représentation fidèle de la partie en cours.

4.11. Analyse de la position par Stockfish

```
info = engine.analyse(
    board,
    chess.engine.Limit(depth=12)
)
```

Le moteur Stockfish analyse la position courante à une profondeur fixée. Le résultat contient une évaluation détaillée de la position.

4.12. Interprétation du score

```
score = info["score"].relative
```

Le score est récupéré du point de vue du joueur au trait. Cette valeur peut représenter soit un avantage matériel, soit un mat forcé.

4.13. Détection d'un mat forcé

```
if score.is_mate():
    mate_in = score.mate()
```

Si Stockfish détecte un mat forcé, le service l'annonce clairement en indiquant le nombre de coups restants et le camp gagnant.

4.14. Évaluation matérielle

```
cp = score.score()
pions = abs(cp) / 100
```

Lorsque la position n'est pas un mat forcé, le score est exprimé en centipions. Il est ensuite converti en pions afin de fournir une évaluation plus lisible.

4.15. Affichage d'une interprétation humaine

```
if abs(cp) < 20:
    print("Position    quilibre   ")
elif cp > 0:
    print(f"Avantage BLANC : {pions:.2f} pion(s)")
else:
    print(f"Avantage NOIR : {pions:.2f} pion(s)")
```

Le service interprète le score numérique pour produire un affichage compréhensible indiquant une position équilibrée ou un avantage pour l'un des deux camps.

4.16. Gestion de la fin de partie

```
if event_type == "game_ended":
    print("Fin de partie :", payload.get("result"))
```

Lorsque la fin de partie est annoncée par le service de validation, le service d'analyse affiche simplement le résultat final. Il ne modifie aucun état interne supplémentaire.

4.17. Abonnement et démarrage de l'écoute

```
channel.basic_consume(
    queue=queue,
    on_message_callback=on_message,
    auto_ack=True
)

print("Analysis service pr t , en attente des    vnements   ")
channel.start_consuming()
```

Le service s'abonne aux messages RabbitMQ et entre dans une phase d'écoute permanente. Chaque événement reçu déclenche l'analyse correspondante.

5. Service Spectateur (spectator_service.py)

Le service spectateur est un service purement passif du système d'échecs distribué. Son objectif principal est d'afficher graphiquement l'état de la partie en cours à partir des coups validés diffusés via RabbitMQ. Il ne prend aucune décision, ne valide aucun coup et ne modifie jamais l'état officiel de la partie. Il se contente d'observer les événements et de représenter visuellement le plateau d'échecs.

Le code analysé correspond au fichier `spectator_service.py` fourni dans le projet turn0file1.

5.1. Importation des bibliothèques

Le service commence par importer les bibliothèques nécessaires à son fonctionnement.

```
import pika
import json
import chess
import matplotlib.pyplot as plt
import time
import sys
```

La bibliothèque `pika` permet la communication avec RabbitMQ. La bibliothèque `json` est utilisée pour décoder les messages reçus. Le module `chess` permet de maintenir un plateau local et de rejouer les coups validés. La bibliothèque `matplotlib` est utilisée pour l'affichage graphique du plateau. Les modules `time` et `sys` sont utilisés respectivement pour gérer les temporisations et l'arrêt propre du service.

5.2. Initialisation et message de démarrage

```
print("spectator_service d marr ")
```

Ce message permet d'indiquer visuellement que le service spectateur a bien été lancé.

5.3. Constantes de configuration

Les paramètres globaux du service sont définis sous forme de constantes.

```
RABBITMQ_HOST = "localhost"
EXCHANGE_NAME = "chess.events"
QUEUE_NAME = "spectator_service"
```

Ces constantes définissent respectivement l'adresse du broker RabbitMQ, le nom de l'exchange commun et le nom de la file utilisée par le service spectateur. Contrairement aux autres services, cette file est persistante afin de permettre une reconnexion propre.

5.4. Plateau local du spectateur

```
board = chess.Board()
```

Le plateau `board` est un plateau local utilisé uniquement pour l'affichage. Il est mis à jour uniquement à partir des coups validés reçus depuis RabbitMQ.

5.5. Initialisation de l'affichage graphique

```
plt.ion()
fig, ax = plt.subplots(figsize=(6, 6))
```

Le mode interactif de `matplotlib` est activé afin de permettre une mise à jour dynamique de l'affichage. Une figure et un axe sont ensuite créés pour dessiner le plateau d'échecs.

5.6. Définition des symboles des pièces

```
PIECE_SYMBOLS = {
    "P": "P", "R": "R", "N": "N", "B": "B", "Q": "Q", "K": "K",
    "p": "p", "r": "r", "n": "n", "b": "b", "q": "q", "k": "k",
}
```

Ce dictionnaire associe chaque type de pièce à un symbole textuel utilisé lors de l'affichage graphique du plateau.

5.7. Fonction de dessin du plateau

```
def draw_board():
```

La fonction `draw_board` est responsable du rendu graphique du plateau et des pièces.

5.7.1. Dessin des cases

```
for x in range(8):
    for y in range(8):
        color = "#f0d9b5" if (x + y) % 2 == 0 else "#b58863"
        ax.add_patch(plt.Rectangle((x, y), 1, 1, color=color))
```

Cette boucle dessine les 64 cases du plateau en alternant les couleurs claires et foncées selon la position.

5.7.2. Dessin des pièces

```
for square in chess.SQUARES:
    piece = board.piece_at(square)
```

Chaque case est inspectée afin de déterminer si une pièce y est présente. Si c'est le cas, un symbole textuel est affiché au centre de la case correspondante.

5.8. Mise en forme de l'affichage

```
ax.set_xlim(0, 8)
ax.set_ylim(0, 8)
ax.set_xticks([])
ax.set_yticks([])
ax.set_title("Spectateur - Partie en cours")
```

Ces instructions permettent de masquer les axes et d'ajouter un titre à la fenêtre graphique.

5.9. Réception des événements RabbitMQ

La fonction `on_message` est enregistrée comme callback pour traiter les messages entrants.

```
def on_message(ch, method, properties, body):
    global board
```

Le mot-clé `global` permet à la fonction de modifier le plateau local utilisé pour l'affichage.

5.10. Décodage et interprétation des événements

```
event = json.loads(body)
event_type = event.get("event_type")
payload = event.get("payload", {})
```

Le message JSON reçu est décodé afin d'identifier le type d'événement et les données associées.

5.11. Gestion du démarrage d'une nouvelle partie

```
if event_type == "game_started":
    board.reset()
    draw_board()
```

Lorsque le spectateur reçoit un événement `game_started`, le plateau local est réinitialisé et l'affichage est mis à jour.

5.12. Affichage d'un coup validé

```
elif event_type == "move_validated":
    move = chess.Move.from_uci(payload["uci"])
```

Le coup validé est converti depuis son format UCI.

```
if move in board.legal_moves:
    board.push(move)
    draw_board()
```

Le coup est appliqué au plateau local uniquement s'il est légal. Cela permet de sécuriser l'affichage contre des incohérences éventuelles.

5.13. Gestion de la fin de partie

```
elif event_type == "game_ended":
    print("Fin de partie :", payload.get("result"))
    draw_board()
```

Lorsque la fin de partie est annoncée, le service affiche le résultat et met à jour une dernière fois le plateau.

5.14. Fonction principale

```
def main():
```

La fonction principale maintient le service actif et gère les reconnexions éventuelles à RabbitMQ.

5.15. Connexion et abonnement à RabbitMQ

```
connection = pika.BlockingConnection(
    pika.ConnectionParameters(
        host=RABBITMQ_HOST,
        heartbeat=600,
        blocked_connection_timeout=300
    )
)
```

Ces paramètres permettent de maintenir une connexion stable même en cas de latence ou de blocage temporaire.

5.16. Déclaration et liaison de la file

```
channel.exchange_declare(
    exchange=EXCHANGE_NAME,
    exchange_type="fanout",
    durable=True
)

channel.queue_declare(
    queue=QUEUE_NAME,
    durable=True
)

channel.queue_bind(
    exchange=EXCHANGE_NAME,
```

```
    queue=QUEUE_NAME
)
```

Le service utilise une file durable afin de conserver les messages en cas de reconnexion.

5.17. Abonnement aux messages

```
channel.basic_consume(
    queue=QUEUE_NAME,
    on_message_callback=on_message,
    auto_ack=True
)
```

La fonction `on_message` est enregistrée comme callback pour le traitement des événements.

5.18. Démarrage de l'écoute

```
draw_board()
channel.start_consuming()
```

Le service entre alors dans une phase d'écoute permanente.

5.19. Gestion des erreurs et arrêt du service

```
except pika.exceptions.AMQPConnectionError as e:
    time.sleep(5)
except KeyboardInterrupt:
    sys.exit(0)
```

Ces blocs permettent au service de gérer proprement les interruptions et les indisponibilités temporaires du broker RabbitMQ.