import os

import requests

import base64

import json

from PyQt6.QtWidgets import (

    QDialog, QVBoxLayout, QHBoxLayout, QListWidget, QPushButton,

    QFileDialog, QMessageBox, QLabel, QProgressBar

)

from PyQt6.QtCore import Qt, QThread, pyqtSignal

from config import GITHUB\_TOKEN, REPO\_OWNER, REPO\_NAME

class PartageWorker(QThread):

    progress\_updated = pyqtSignal(int)

    operation\_completed = pyqtSignal(bool, str)

    def \_\_init\_\_(self, operation, \*args):

        super().\_\_init\_\_()

        self.operation = operation

        self.args = args

        self.MAX\_FILE\_SIZE = 100 \* 1024 \* 1024  # 100MB limite

    def run(self):

        try:

            if self.operation == "download":

                self.\_download\_file(\*self.args)

            elif self.operation == "upload":

                self.\_upload\_file(\*self.args)

        except Exception as e:

            self.operation\_completed.emit(False, str(e))

    def \_download\_file(self, file\_name, download\_path):

        try:

            self.progress\_updated.emit(5)

            # Configuration des headers

            headers = {

                "Authorization": f"token {GITHUB\_TOKEN}",

                "Accept": "application/vnd.github.v3.raw"

            }

            # URL de téléchargement direct (raw)

            download\_url = f"https://raw.githubusercontent.com/{REPO\_OWNER}/{REPO\_NAME}/main/Partage/{file\_name}"

            # Téléchargement avec stream

            self.progress\_updated.emit(10)

            with requests.get(download\_url, headers=headers, stream=True, timeout=30) as r:

                r.raise\_for\_status()

                # Vérification de la taille

                total\_size = int(r.headers.get('content-length', 0))

                if total\_size > self.MAX\_FILE\_SIZE:

                    raise ValueError(f"Fichier trop volumineux ({total\_size//(1024\*1024)}MB > {self.MAX\_FILE\_SIZE//(1024\*1024)}MB max)")

                # Écriture par chunks avec progression

                self.progress\_updated.emit(20)

                downloaded = 0

                with open(download\_path, 'wb') as f:

                    for chunk in r.iter\_content(chunk\_size=8192):  # 8KB chunks

                        if self.isInterruptionRequested():

                            raise Exception("Téléchargement annulé")

                        if chunk:  # filter out keep-alive chunks

                            f.write(chunk)

                            downloaded += len(chunk)

                            progress = 20 + int(70 \* (downloaded / total\_size))

                            self.progress\_updated.emit(progress)

            self.progress\_updated.emit(95)

            # Vérification finale

            if os.path.getsize(download\_path) == total\_size:

                self.operation\_completed.emit(True, f"Téléchargement réussi: {file\_name} ({total\_size//1024} KB)")

            else:

                raise Exception("Téléchargement incomplet")

        except requests.exceptions.RequestException as e:

            self.operation\_completed.emit(False, f"Erreur réseau: {str(e)}")

        except Exception as e:

            # Nettoyage en cas d'erreur

            if os.path.exists(download\_path):

                os.remove(download\_path)

            self.operation\_completed.emit(False, f"Erreur de téléchargement: {str(e)}")

    def \_upload\_file(self, file\_path, file\_name):

        try:

            self.progress\_updated.emit(5)

            # Vérification de la taille

            file\_size = os.path.getsize(file\_path)

            if file\_size > self.MAX\_FILE\_SIZE:

                raise ValueError(f"Fichier trop volumineux ({file\_size//(1024\*1024)}MB > {self.MAX\_FILE\_SIZE//(1024\*1024)}MB max)")

            # 1. Vérifier si le fichier existe déjà

            self.progress\_updated.emit(10)

            url = f"https://api.github.com/repos/{REPO\_OWNER}/{REPO\_NAME}/contents/Partage/{file\_name}"

            headers = {

                "Authorization": f"token {GITHUB\_TOKEN}",

                "Accept": "application/vnd.github.v3+json"

            }

            sha = None

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

            if response.status\_code == 200:

                sha = response.json().get('sha')

            # 2. Lecture par blocs avec progression

            self.progress\_updated.emit(20)

            chunks = []

            total\_read = 0

            with open(file\_path, 'rb') as f:

                while True:

                    if self.isInterruptionRequested():

                        raise Exception("Upload annulé")

                    chunk = f.read(1024\*1024)  # 1MB chunks

                    if not chunk:

                        break

                    chunks.append(chunk)

                    total\_read += len(chunk)

                    progress = 20 + int(40 \* (total\_read / file\_size))

                    self.progress\_updated.emit(progress)

            # 3. Encodage base64

            self.progress\_updated.emit(65)

            content = b''.join(chunks)

            encoded\_content = base64.b64encode(content).decode('utf-8')

            # 4. Préparation requête

            self.progress\_updated.emit(75)

            data = {

                "message": f"Ajout du fichier {file\_name}",

                "content": encoded\_content

            }

            if sha:

                data["sha"] = sha

            # 5. Envoi

            self.progress\_updated.emit(85)

            response = requests.put(url, headers=headers, json=data, timeout=60)

            response.raise\_for\_status()

            self.progress\_updated.emit(100)

            self.operation\_completed.emit(True, f"Fichier uploadé: {file\_name} ({file\_size//1024} KB)")

        except requests.exceptions.RequestException as e:

            self.operation\_completed.emit(False, f"Erreur réseau: {str(e)}")

        except Exception as e:

            self.operation\_completed.emit(False, f"Erreur d'upload: {str(e)}")

class PartageDialog(QDialog):

    def \_\_init\_\_(self, parent=None):

        super().\_\_init\_\_(parent)

        self.setWindowTitle("Partage de fichiers")

        self.setMinimumSize(700, 500)

        self.local\_dir = "Partage"

        os.makedirs(self.local\_dir, exist\_ok=True)

        self.current\_worker = None

        self.init\_ui()

        self.load\_github\_files()

    def init\_ui(self):

        layout = QVBoxLayout()

        # Titre

        title = QLabel("Gestion des fichiers partagés")

        title.setAlignment(Qt.AlignmentFlag.AlignCenter)

        title.setStyleSheet("font-weight: bold; font-size: 16px;")

        layout.addWidget(title)

        # Liste des fichiers GitHub

        self.github\_list = QListWidget()

        self.github\_list.setSelectionMode(QListWidget.SelectionMode.SingleSelection)

        layout.addWidget(QLabel("Fichiers disponibles sur GitHub:"))

        layout.addWidget(self.github\_list)

        # Boutons

        btn\_layout = QHBoxLayout()

        self.download\_btn = QPushButton("Télécharger")

        self.download\_btn.clicked.connect(self.download\_file)

        btn\_layout.addWidget(self.download\_btn)

        self.upload\_btn = QPushButton("Importer")

        self.upload\_btn.clicked.connect(self.upload\_file)

        btn\_layout.addWidget(self.upload\_btn)

        self.refresh\_btn = QPushButton("Actualiser")

        self.refresh\_btn.clicked.connect(self.load\_github\_files)

        btn\_layout.addWidget(self.refresh\_btn)

        self.download\_btn.setFixedHeight(30)

        self.upload\_btn.setFixedHeight(30)

        self.refresh\_btn.setFixedHeight(30)

        layout.addLayout(btn\_layout)

        # Barre de progression

        self.progress = QProgressBar()

        self.progress.setVisible(False)

        layout.addWidget(self.progress)

        # Status label

        self.status\_label = QLabel()

        self.status\_label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        layout.addWidget(self.status\_label)

        self.setLayout(layout)

    def load\_github\_files(self):

        if not all([GITHUB\_TOKEN, REPO\_OWNER, REPO\_NAME]):

            QMessageBox.warning(self, "Configuration manquante",

                              "Veuillez configurer les informations GitHub dans config.py")

            return

        try:

            self.status\_label.setText("Chargement de la liste des fichiers...")

            url = f"https://api.github.com/repos/{REPO\_OWNER}/{REPO\_NAME}/contents/Partage"

            headers = {"Authorization": f"token {GITHUB\_TOKEN}"}

            response = requests.get(url, headers=headers, timeout=10)

            response.raise\_for\_status()

            files = [item['name'] for item in response.json() if item['type'] == 'file']

            self.github\_list.clear()

            self.github\_list.addItems(files)

            self.status\_label.setText(f"{len(files)} fichiers disponibles")

        except Exception as e:

            QMessageBox.critical(self, "Erreur", f"Impossible de charger les fichiers: {str(e)}")

            self.status\_label.setText("Erreur de chargement")

    def download\_file(self):

        selected\_items = self.github\_list.selectedItems()

        if not selected\_items:

            QMessageBox.warning(self, "Aucune sélection", "Veuillez sélectionner un fichier à télécharger")

            return

        file\_name = selected\_items[0].text()

        save\_path = os.path.join(self.local\_dir, file\_name)

        # Vérifier si le fichier existe déjà localement

        if os.path.exists(save\_path):

            reply = QMessageBox.question(

                self,

                "Fichier existant",

                f"Le fichier {file\_name} existe déjà. Voulez-vous le remplacer?",

                QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No

            )

            if reply == QMessageBox.StandardButton.No:

                return

        # Lancer le téléchargement

        self.progress.setVisible(True)

        self.progress.setValue(0)

        self.set\_buttons\_enabled(False)

        self.status\_label.setText(f"Téléchargement de {file\_name}...")

        self.current\_worker = PartageWorker("download", file\_name, save\_path)

        self.current\_worker.progress\_updated.connect(self.progress.setValue)

        self.current\_worker.operation\_completed.connect(self.on\_operation\_complete)

        self.current\_worker.start()

    def upload\_file(self):

        file\_path, \_ = QFileDialog.getOpenFileName(

            self,

            "Sélectionner un fichier à importer",

            "",

            "Tous les fichiers (\*)"

        )

        if not file\_path:

            return

        file\_name = os.path.basename(file\_path)

        file\_size = os.path.getsize(file\_path)

        # Vérifier la taille

        if file\_size > 100 \* 1024 \* 1024:  # 100MB

            QMessageBox.warning(self, "Fichier trop volumineux",

                              "Les fichiers de plus de 100MB ne sont pas supportés")

            return

        # Lancer l'upload

        self.progress.setVisible(True)

        self.progress.setValue(0)

        self.set\_buttons\_enabled(False)

        self.status\_label.setText(f"Envoi de {file\_name} ({file\_size//1024} KB)...")

        self.current\_worker = PartageWorker("upload", file\_path, file\_name)

        self.current\_worker.progress\_updated.connect(self.progress.setValue)

        self.current\_worker.operation\_completed.connect(self.on\_operation\_complete)

        self.current\_worker.start()

    def on\_operation\_complete(self, success, message):

        self.progress.setVisible(False)

        self.set\_buttons\_enabled(True)

        self.status\_label.setText(message)

        if success:

            QMessageBox.information(self, "Succès", message)

            self.load\_github\_files()  # Rafraîchir la liste

        else:

            QMessageBox.critical(self, "Erreur", message)

    def set\_buttons\_enabled(self, enabled):

        self.download\_btn.setEnabled(enabled)

        self.upload\_btn.setEnabled(enabled)

        self.refresh\_btn.setEnabled(enabled)

    def closeEvent(self, event):

        """Arrêter les opérations en cours lors de la fermeture"""

        if self.current\_worker and self.current\_worker.isRunning():

            self.current\_worker.requestInterruption()

            self.current\_worker.wait()

        event.accept()

Analyse détaillée du programme partage\_interface.py

Je vais analyser ce programme ligne par ligne et vous fournir une explication détaillée. Voici l'analyse complète :

Structure globale du programme

Le programme est une interface PyQt6 permettant de partager des fichiers via GitHub. Il se compose de deux classes principales :

1. PartageWorker - Un QThread qui gère les opérations de téléchargement et d'upload en arrière-plan
2. PartageDialog - Une fenêtre d'interface utilisateur pour interagir avec le système de partage

Analyse détaillée

Importations

import os

import requests *# Pour les requêtes HTTP*

import base64 *# Pour encoder/décoder les fichiers en base64*

import json *# Pour traiter les réponses JSON*

from PyQt6.QtWidgets import ( *# Composants d'interface*

QDialog, QVBoxLayout, QHBoxLayout, QListWidget, QPushButton,

QFileDialog, QMessageBox, QLabel, QProgressBar

)

from PyQt6.QtCore import Qt, QThread, pyqtSignal *# Fonctionnalités Qt de base*

from config import GITHUB\_TOKEN, REPO\_OWNER, REPO\_NAME *# Configuration*

Classe PartageWorker

class PartageWorker(QThread):

progress\_updated = pyqtSignal(int) *# Signal pour la progression*

operation\_completed = pyqtSignal(bool, str) *# Signal pour le résultat*

def \_\_init\_\_(self, operation, \*args):

super().\_\_init\_\_()

self.operation = operation *# "download" ou "upload"*

self.args = args *# Arguments passés*

self.MAX\_FILE\_SIZE = 100 \* 1024 \* 1024 *# Limite de 100MB*

Méthode run()

def run(self):

try:

if self.operation == "download":

self.\_download\_file(\*self.args)

elif self.operation == "upload":

self.\_upload\_file(\*self.args)

except Exception as e:

self.operation\_completed.emit(False, str(e))

Méthode \_download\_file()

def \_download\_file(self, file\_name, download\_path):

try:

self.progress\_updated.emit(5) *# Progression initiale*

*# Configuration des headers avec le token GitHub*

headers = {

"Authorization": f"token {GITHUB\_TOKEN}",

"Accept": "application/vnd.github.v3.raw"

}

*# URL de téléchargement direct*

download\_url = f"https://raw.githubusercontent.com/{REPO\_OWNER}/{REPO\_NAME}/main/Partage/{file\_name}"

*# Téléchargement avec stream pour les gros fichiers*

self.progress\_updated.emit(10)

with requests.get(download\_url, headers=headers, stream=True, timeout=30) as r:

r.raise\_for\_status()

*# Vérification taille du fichier*

total\_size = int(r.headers.get('content-length', 0))

if total\_size > self.MAX\_FILE\_SIZE:

raise ValueError(f"Fichier trop volumineux...")

*# Écriture par morceaux avec progression*

self.progress\_updated.emit(20)

downloaded = 0

with open(download\_path, 'wb') as f:

for chunk in r.iter\_content(chunk\_size=8192): *# Morceaux de 8KB*

if self.isInterruptionRequested(): *# Vérifie si annulation*

raise Exception("Téléchargement annulé")

if chunk: *# Filtre les chunks vides*

f.write(chunk)

downloaded += len(chunk)

progress = 20 + int(70 \* (downloaded / total\_size))

self.progress\_updated.emit(progress)

self.progress\_updated.emit(95)

*# Vérification finale*

if os.path.getsize(download\_path) == total\_size:

self.operation\_completed.emit(True, f"Téléchargement réussi...")

else:

raise Exception("Téléchargement incomplet")

Méthode \_upload\_file()

def \_upload\_file(self, file\_path, file\_name):

try:

self.progress\_updated.emit(5)

*# Vérification taille du fichier*

file\_size = os.path.getsize(file\_path)

if file\_size > self.MAX\_FILE\_SIZE:

raise ValueError(f"Fichier trop volumineux...")

*# 1. Vérifier si le fichier existe déjà sur GitHub*

self.progress\_updated.emit(10)

url = f"https://api.github.com/repos/{REPO\_OWNER}/{REPO\_NAME}/contents/Partage/{file\_name}"

headers = {

"Authorization": f"token {GITHUB\_TOKEN}",

"Accept": "application/vnd.github.v3+json"

}

sha = None

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

if response.status\_code == 200: *# Fichier existe*

sha = response.json().get('sha') *# SHA nécessaire pour mise à jour*

*# 2. Lecture par blocs avec progression*

self.progress\_updated.emit(20)

chunks = []

total\_read = 0

with open(file\_path, 'rb') as f:

while True:

if self.isInterruptionRequested():

raise Exception("Upload annulé")

chunk = f.read(1024\*1024) *# Morceaux de 1MB*

if not chunk:

break

chunks.append(chunk)

total\_read += len(chunk)

progress = 20 + int(40 \* (total\_read / file\_size))

self.progress\_updated.emit(progress)

*# 3. Encodage base64*

self.progress\_updated.emit(65)

content = b''.join(chunks)

encoded\_content = base64.b64encode(content).decode('utf-8')

*# 4. Préparation requête*

self.progress\_updated.emit(75)

data = {

"message": f"Ajout du fichier {file\_name}",

"content": encoded\_content

}

if sha: *# Si fichier existe déjà*

data["sha"] = sha

*# 5. Envoi*

self.progress\_updated.emit(85)

response = requests.put(url, headers=headers, json=data, timeout=60)

response.raise\_for\_status()

self.progress\_updated.emit(100)

self.operation\_completed.emit(True, f"Fichier uploadé...")

Classe PartageDialog

class PartageDialog(QDialog):

def \_\_init\_\_(self, parent=None):

super().\_\_init\_\_(parent)

self.setWindowTitle("Partage de fichiers")

self.setMinimumSize(700, 500) *# Taille minimale*

self.local\_dir = "Partage" *# Dossier local*

os.makedirs(self.local\_dir, exist\_ok=True) *# Crée le dossier si inexistant*

self.current\_worker = None *# Référence au worker actuel*

self.init\_ui() *# Initialise l'interface*

self.load\_github\_files() *# Charge la liste des fichiers*

Méthode init\_ui()

def init\_ui(self):

layout = QVBoxLayout() *# Layout principal vertical*

*# Titre*

title = QLabel("Gestion des fichiers partagés")

title.setAlignment(Qt.AlignmentFlag.AlignCenter)

title.setStyleSheet("font-weight: bold; font-size: 16px;")

layout.addWidget(title)

*# Liste des fichiers GitHub*

self.github\_list = QListWidget()

self.github\_list.setSelectionMode(QListWidget.SelectionMode.SingleSelection)

layout.addWidget(QLabel("Fichiers disponibles sur GitHub:"))

layout.addWidget(self.github\_list)

*# Boutons*

btn\_layout = QHBoxLayout() *# Layout horizontal pour les boutons*

self.download\_btn = QPushButton("Télécharger")

self.download\_btn.clicked.connect(self.download\_file)

btn\_layout.addWidget(self.download\_btn)

self.upload\_btn = QPushButton("Importer")

self.upload\_btn.clicked.connect(self.upload\_file)

btn\_layout.addWidget(self.upload\_btn)

self.refresh\_btn = QPushButton("Actualiser")

self.refresh\_btn.clicked.connect(self.load\_github\_files)

btn\_layout.addWidget(self.refresh\_btn)

*# Taille fixe pour les boutons*

self.download\_btn.setFixedHeight(30)

self.upload\_btn.setFixedHeight(30)

self.refresh\_btn.setFixedHeight(30)

layout.addLayout(btn\_layout)

*# Barre de progression*

self.progress = QProgressBar()

self.progress.setVisible(False) *# Cachée par défaut*

layout.addWidget(self.progress)

*# Label de statut*

self.status\_label = QLabel()

self.status\_label.setAlignment(Qt.AlignmentFlag.AlignCenter)

layout.addWidget(self.status\_label)

self.setLayout(layout)

Méthode load\_github\_files()

def load\_github\_files(self):

if not all([GITHUB\_TOKEN, REPO\_OWNER, REPO\_NAME]): *# Vérifie la config*

QMessageBox.warning(self, "Configuration manquante",

"Veuillez configurer les informations GitHub dans config.py")

return

try:

self.status\_label.setText("Chargement de la liste des fichiers...")

url = f"https://api.github.com/repos/{REPO\_OWNER}/{REPO\_NAME}/contents/Partage"

headers = {"Authorization": f"token {GITHUB\_TOKEN}"}

response = requests.get(url, headers=headers, timeout=10)

response.raise\_for\_status()

*# Filtre seulement les fichiers (pas les dossiers)*

files = [item['name'] for item in response.json() if item['type'] == 'file']

self.github\_list.clear()

self.github\_list.addItems(files)

self.status\_label.setText(f"{len(files)} fichiers disponibles")

except Exception as e:

QMessageBox.critical(self, "Erreur", f"Impossible de charger les fichiers: {str(e)}")

self.status\_label.setText("Erreur de chargement")

Méthode download\_file()

def download\_file(self):

selected\_items = self.github\_list.selectedItems()

if not selected\_items: *# Aucune sélection*

QMessageBox.warning(self, "Aucune sélection", "Veuillez sélectionner un fichier à télécharger")

return

file\_name = selected\_items[0].text()

save\_path = os.path.join(self.local\_dir, file\_name)

*# Vérifie si le fichier existe déjà*

if os.path.exists(save\_path):

reply = QMessageBox.question(

self,

"Fichier existant",

f"Le fichier {file\_name} existe déjà. Voulez-vous le remplacer?",

QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No

)

if reply == QMessageBox.StandardButton.No:

return

*# Configure l'interface pendant le téléchargement*

self.progress.setVisible(True)

self.progress.setValue(0)

self.set\_buttons\_enabled(False)

self.status\_label.setText(f"Téléchargement de {file\_name}...")

*# Crée et démarre le worker*

self.current\_worker = PartageWorker("download", file\_name, save\_path)

self.current\_worker.progress\_updated.connect(self.progress.setValue)

self.current\_worker.operation\_completed.connect(self.on\_operation\_complete)

self.current\_worker.start()

Méthode upload\_file()

def upload\_file(self):

*# Ouvre un dialogue de sélection de fichier*

file\_path, \_ = QFileDialog.getOpenFileName(

self,

"Sélectionner un fichier à importer",

"",

"Tous les fichiers (\*)"

)

if not file\_path: *# Annulation*

return

file\_name = os.path.basename(file\_path)

file\_size = os.path.getsize(file\_path)

*# Vérifie la taille*

if file\_size > 100 \* 1024 \* 1024: *# 100MB*

QMessageBox.warning(self, "Fichier trop volumineux",

"Les fichiers de plus de 100MB ne sont pas supportés")

return

*# Configure l'interface pendant l'upload*

self.progress.setVisible(True)

self.progress.setValue(0)

self.set\_buttons\_enabled(False)

self.status\_label.setText(f"Envoi de {file\_name} ({file\_size//1024} KB)...")

*# Crée et démarre le worker*

self.current\_worker = PartageWorker("upload", file\_path, file\_name)

self.current\_worker.progress\_updated.connect(self.progress.setValue)

self.current\_worker.operation\_completed.connect(self.on\_operation\_complete)

self.current\_worker.start()

Autres méthodes

python

Copy

Download

def on\_operation\_complete(self, success, message):

self.progress.setVisible(False)

self.set\_buttons\_enabled(True)

self.status\_label.setText(message)

if success:

QMessageBox.information(self, "Succès", message)

self.load\_github\_files() *# Rafraîchit la liste*

else:

QMessageBox.critical(self, "Erreur", message)

def set\_buttons\_enabled(self, enabled):

self.download\_btn.setEnabled(enabled)

self.upload\_btn.setEnabled(enabled)

self.refresh\_btn.setEnabled(enabled)

def closeEvent(self, event):

"""Arrête les opérations en cours lors de la fermeture"""

if self.current\_worker and self.current\_worker.isRunning():

self.current\_worker.requestInterruption()

self.current\_worker.wait()

event.accept()

Points clés à retenir

1. **Architecture**:
   * Séparation claire entre l'interface (PartageDialog) et le traitement (PartageWorker)
   * Utilisation de QThread pour les opérations longues (évite de bloquer l'interface)
2. **Fonctionnalités GitHub**:
   * Utilisation de l'API GitHub pour gérer les fichiers
   * Téléchargement via l'URL raw
   * Upload via l'API avec encodage base64
3. **Gestion des fichiers**:
   * Limite de taille à 100MB
   * Téléchargement par morceaux avec progression
   * Gestion des conflits (fichiers existants)
4. **Interface utilisateur**:
   * Barre de progression
   * Feedback visuel
   * Gestion des erreurs

Conseils pour développer un programme similaire

1. **Structure**:
   * Toujours séparer interface et traitement
   * Utiliser des threads pour les opérations longues
2. **API externe**:
   * Bien lire la documentation de l'API utilisée
   * Gérer les limites et quotas
   * Implémenter une gestion robuste des erreurs
3. **Interface**:
   * Donner un feedback visuel pendant les opérations
   * Bloquer les interactions pendant les traitements
   * Fournir des messages d'erreur clairs
4. **Sécurité**:
   * Ne pas hardcoder les tokens/secrets
   * Valider les entrées utilisateur
   * Gérer proprement les interruptions

Pour développer un programme similaire, vous pourriez par exemple:

* Créer un système de partage utilisant Dropbox ou Google Drive à la place de GitHub
* Ajouter des fonctionnalités comme la compression avant upload
* Implémenter un système de synchronisation bidirectionnelle

Je vais générer un fichier Word téléchargeable contenant cette analyse.

**1. Architecture Globale**

Le programme repose sur deux composants principaux qui communiquent entre eux via des signaux PyQt :

**a.**PartageWorker**(Traitement en arrière-plan)**

* Hérite de QThread pour exécuter des tâches longues sans bloquer l'interface.
* Émet des signaux pour :
  + progress\_updated : Mise à jour de la progression (0-100%).
  + operation\_completed : Résultat final (succès/échec + message).

**b.**PartageDialog**(Interface Utilisateur)**

* Gère les interactions utilisateur (boutons, sélections).
* Instancie PartageWorker et connecte ses signaux aux slots appropriés.

**2. Mécanisme de Téléchargement**

**Séquence détaillée :**

1. **Initialisation**
   * L'utilisateur sélectionne un fichier dans QListWidget.
   * Vérification de l'existence locale du fichier (avec demande de confirmation pour écrasement).
2. **Configuration de la requête**

headers = {

"Authorization": f"token {GITHUB\_TOKEN}",

"Accept": "application/vnd.github.v3.raw" *# Format brut (non JSON)*

}

* + L'URL utilise le domaine raw.githubusercontent.com pour un téléchargement direct.

1. **Streaming du fichier**
   * Utilisation de stream=True dans requests.get() pour éviter de charger tout le fichier en mémoire.
   * Lecture par morceaux de 8 Ko :

for chunk in r.iter\_content(chunk\_size=8192):

f.write(chunk)

downloaded += len(chunk)

*# Calcul de la progression (20% à 90%)*

progress = 20 + int(70 \* (downloaded / total\_size))

1. **Vérification finale**
   * Compare la taille téléchargée avec content-length pour détecter les corruptions.

**3. Mécanisme d'Upload**

**Étapes clés :**

1. **Pré-vérification**
   * Vérifie la taille du fichier (<100 Mo).
   * Récupère le sha du fichier existant (pour les mises à jour) via une requête GET.
2. **Lecture et encodage**
   * Découpage en blocs de 1 Mo pour permettre des mises à jour de progression :

chunk = f.read(1024\*1024) *# 1 Mo*

chunks.append(chunk)

progress = 20 + int(40 \* (total\_read / file\_size))

* + Encodage Base64 obligatoire pour l'API GitHub :

encoded\_content = base64.b64encode(content).decode('utf-8')

1. **Requête PUT**
   * Envoie les métadonnées + contenu encodé :

data = {

"message": f"Ajout du fichier {file\_name}",

"content": encoded\_content,

"sha": sha *# Seulement pour les mises à jour*

}

requests.put(url, headers=headers, json=data)

**4. Gestion des Erreurs**

**Stratégies mises en place :**

* **Réseau** : Timeout de 30s pour les téléchargements, 60s pour les uploads.
* **Taille des fichiers** : Blocage dès le départ si >100 Mo.
* **Interruptions** :

if self.isInterruptionRequested():

raise Exception("Opération annulée")

* **Nettoyage** : Suppression des fichiers incomplets en cas d'échec.

**Exemple de gestion d'erreur dans l'interface :**

except requests.exceptions.RequestException as e:

self.operation\_completed.emit(False, f"Erreur réseau: {str(e)}")

**5. Détails Techniques Importants**

**a. Progression**

* Les pourcentages sont répartis pour refléter les étapes :
  + **Téléchargement** :
    - 0-20% : Préparation + vérifications.
    - 20-90% : Transfert des données.
    - 90-100% : Validation finale.

**b. Sécurité**

* Le token GitHub est toujours transmis via les headers, jamais dans l'URL.
* Les fichiers locaux sont stockés dans un dossier dédié (./Partage/).

**c. Performances**

* Utilisation de chunk\_size optimaux :
  + 8 Ko pour le téléchargement (équilibre entre vitesse et réactivité).
  + 1 Mo pour l'upload (réduit le nombre d'appels à base64.b64encode).

**6. Extensions Possibles**

1. **Compression** :  
   Utiliser zlib pour compresser les fichiers avant upload :

python

Copy

Download

import zlib

compressed\_content = zlib.compress(content)

encoded\_content = base64.b64encode(compressed\_content).decode('utf-8')

1. **Synchronisation** :  
   Ajouter un système pour détecter les modifications locales/serveur via des timestamps.
2. **Journalisation** :  
   Logger les opérations dans un fichier pour le débogage :

python

Copy

Download

import logging

logging.basicConfig(filename='partage.log', level=logging.INFO)

**7. Points de Vigilance**

* **Limites de GitHub** :
  + Taille max par fichier : 100 Mo (limite de l'API).
  + Quota de requêtes : 5 000 req/h pour un token personnel.
* **Blocages UI** :  
  Toujours vérifier que les signaux/slots sont correctement connectés pour éviter des freezes.

Cette analyse approfondie révèle comment le programme allie robustesse (gestion d'erreurs, streaming) et convivialité (barre de progression, feedback utilisateur). Pour un projet similaire, vous pourriez remplacer GitHub par une autre API (comme AWS S3) en conservant la même architecture.