今日

**あなたの発言**

以下のpython CGIに、query parameterが与えられていない場合や通信エラー、logファイルエラーなどの例外処理を追加してください

#!/usr/bin/env python3

import os

import json

import urllib.parse

import glob

import re

import logging

import yaml

# ロギングの設定

logging.basicConfig(

filename='/var/log/cgi-bin/API.log',

level=logging.INFO,

format='%(asctime)s %(message)s'

)

def parse\_query\_string(query\_string):

query = {}

for param in query\_string.split('&'):

key, value = param.split('=')

query[key] = urllib.parse.unquote(value)

query[key] = sanitize\_input(query[key])

return query

def sanitize\_input(value):

value = re.sub(r'[^\w\s\-\.]', '', value) # 許可された文字のみを許可

return value

def get\_account\_list():

with open('accounts.yaml', 'r') as file:

return yaml.safe\_load(file)

def get\_account(user, account\_list):

return next((account for account in account\_list if account['user'] == user), None)

def is\_valid\_path(account, rootDir):

abs\_path = os.path.abspath(rootDir)

if '../' in abs\_path or './' in abs\_path:

return False

return any(

abs\_path.startswith(os.path.abspath(allowed\_dir)) for allowed\_dir in account['dirs']

)

def is\_valid\_filemask(account, filemask):

abs\_filemask = os.path.abspath(filemask)

for pattern in account['rejected\_file\_masks']:

if re.search(pattern, abs\_filemask):

return False

return any(re.search(pattern, abs\_filemask) for pattern in account['file\_masks'])

def is\_valid\_ip(account, ip):

return any(re.match(pattern, ip) for pattern in account['IPaddress'])

def is\_valid\_password(account, password):

return account['password'] == password

def main():

print("Content-Type: application/json")

print()

query\_string = os.getenv('QUERY\_STRING')

query = parse\_query\_string(query\_string)

action = query.get('action', '')

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

user = query.get('user', '')

password = query.get('pw', '')

clientIP = os.getenv('REMOTE\_ADDR', '')

account\_list = get\_account\_list()

account = get\_account(user, account\_list)

if account and is\_valid\_password(account, password) and is\_valid\_ip(account, clientIP) and action == 'getFile' and is\_valid\_path(account, rootDir):

# ファイル検索

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

# 検索結果のファイルマスクチェック

valid\_files = [path for path in pathList if is\_valid\_filemask(account, path)]

relativePaths = [os.path.relpath(path, rootDir) for path in valid\_files]

response = {

'result': True,

'pathList': relativePaths,

'clientIP': clientIP

}

logging.info(f"Success: {rootDir}, {fileMask}, user: {user}, clientIP: {clientIP}, {relativePaths}")

else:

response = {

'result': False,

'message': 'Invalid rootDir, unauthorized access, invalid IP address, invalid user, invalid password, or invalid file mask',

'clientIP': clientIP

}

logging.warning(f"Failure: {rootDir}, {fileMask}, user: {user}, clientIP: {clientIP}")

print(json.dumps(response))

if \_\_name\_\_ == '\_\_main\_\_':

main()

**Copilot の発言**

以下の例外処理を追加することで、query parameterが与えられていない場合や通信エラー、logファイルエラーなどへの対応が可能です。

python

コピー

# 修正バージョン

def parse\_query\_string(query\_string):

query = {}

try:

for param in query\_string.split('&'):

key, value = param.split('=')

query[key] = urllib.parse.unquote(value)

query[key] = sanitize\_input(query[key])

except Exception as e:

logging.error(f"Error parsing query string: {query\_string}, Error: {e}")

raise ValueError("Invalid query string format.")

return query

def get\_account\_list():

try:

with open('accounts.yaml', 'r') as file:

return yaml.safe\_load(file)

except FileNotFoundError as e:

logging.error(f"Error loading account list file: {e}")

raise FileNotFoundError("accounts.yaml file not found.")

except yaml.YAMLError as e:

logging.error(f"Error parsing YAML file: {e}")

raise ValueError("Error parsing accounts.yaml.")

def main():

print("Content-Type: application/json")

print()

try:

query\_string = os.getenv('QUERY\_STRING', '')

if not query\_string:

raise ValueError("QUERY\_STRING is missing.")

query = parse\_query\_string(query\_string)

action = query.get('action', '')

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

user = query.get('user', '')

password = query.get('pw', '')

clientIP = os.getenv('REMOTE\_ADDR', '')

account\_list = get\_account\_list()

account = get\_account(user, account\_list)

if account and is\_valid\_password(account, password) and is\_valid\_ip(account, clientIP) and action == 'getFile' and is\_valid\_path(account, rootDir):

# ファイル検索

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

valid\_files = [path for path in pathList if is\_valid\_filemask(account, path)]

relativePaths = [os.path.relpath(path, rootDir) for path in valid\_files]

response = {'result': True, 'pathList': relativePaths, 'clientIP': clientIP}

logging.info(f"Success: {rootDir}, {fileMask}, user: {user}, clientIP: {clientIP}, {relativePaths}")

else:

response = {'result': False, 'message': 'Invalid parameters or unauthorized access.', 'clientIP': clientIP}

logging.warning(f"Failure: rootDir={rootDir}, fileMask={fileMask}, user={user}, clientIP={clientIP}")

except Exception as e:

response = {'result': False, 'message': f"Error occurred: {e}"}

logging.error(f"Unexpected error: {e}")

print(json.dumps(response))

if \_\_name\_\_ == '\_\_main\_\_':

try:

main()

except Exception as e:

logging.critical(f"Fatal error: {e}")

print(json.dumps({'result': False, 'message': 'A critical error occurred. Check logs for details.'}))

この修正バージョンには、以下の改善点があります：

* **query\_stringのバリデーション**：欠落している場合やフォーマットが不正な場合に例外を発生させる。
* **accounts.yamlの読み込みエラー処理**：ファイルが存在しない場合や形式に問題がある場合に例外を発生させる。
* **汎用エラー処理**：main()内で例外をキャッチして、ユーザー向けのエラー応答を生成する。
* **ログファイルエラー防止**：logging.criticalを利用し、致命的な問題を記録。

このような例外処理を追加することで、コードの安定性と信頼性が向上します。どう思いますか？さらに改善したい点があれば教えてください！ 😊

**あなたの発言**

loggingの例外処理もお願いします

**Copilot の発言**

以下のように logging 操作における例外処理を追加することで、ログファイルのエラーや問題に対応できます。

python

コピー

# ロギングの設定

try:

logging.basicConfig(

filename='/var/log/cgi-bin/API.log',

level=logging.INFO,

format='%(asctime)s %(message)s'

)

except (PermissionError, FileNotFoundError) as e:

# ログファイルが書き込み不可または存在しない場合の対応

print(json.dumps({'result': False, 'message': f"Logging initialization error: {e}"}))

raise SystemExit("Critical error: Unable to initialize logging.")

# ロギング例外処理を追加したログ操作

def safe\_logging(level, message):

try:

if level == "info":

logging.info(message)

elif level == "warning":

logging.warning(message)

elif level == "error":

logging.error(message)

elif level == "critical":

logging.critical(message)

else:

logging.debug(message)

except Exception as e:

# ログ書き込み中のエラーをキャッチ

print(json.dumps({'result': False, 'message': f"Logging error: {e}"}))

**変更点：**

1. logging.basicConfig**に例外処理を追加**：ログファイルが書き込み可能でない場合やパスが存在しない場合に対応するため。
2. safe\_logging**関数の追加**：ログ書き込み中の予期しないエラー（例えば、ディスクの容量不足や権限エラー）をキャッチする汎用的な仕組みを提供。

これにより、ログ関連のエラーに柔軟に対応できるようになります。どうでしょうか？これでミスなく動作するように見えますか？🌟

**あなたの発言**

例外処理にtryを多用するのは良くないと思います。ifで対応できる部分はifにしてください

**Copilot の発言**

なるほど！確かに、tryを多用するのではなく、事前にif条件でエラーを回避する方がコードの読みやすさやパフォーマンスに良い影響を与えることが多いですね。以下は、例外処理を減らしつつifを使ってエラーを事前チェックする形に修正したコードです。

python

コピー

# ロギングの設定

log\_file = '/var/log/cgi-bin/API.log'

if os.access(os.path.dirname(log\_file), os.W\_OK): # ログディレクトリの書き込み可能性を事前チェック

logging.basicConfig(

filename=log\_file,

level=logging.INFO,

format='%(asctime)s %(message)s'

)

else:

print(json.dumps({'result': False, 'message': 'Logging directory is not writable.'}))

raise SystemExit("Critical error: Unable to initialize logging.")

# ログ操作の安全版

def safe\_logging(level, message):

if logging.getLogger().hasHandlers(): # ハンドラーが設定されているかチェック

if level == "info":

logging.info(message)

elif level == "warning":

logging.warning(message)

elif level == "error":

logging.error(message)

elif level == "critical":

logging.critical(message)

else:

logging.debug(message)

else:

print(json.dumps({'result': False, 'message': 'Logging handler not available. Message was: {}'.format(message)}))

**変更点：**

1. **ログディレクトリの事前チェック**：
   * os.access を使用し、ログディレクトリが書き込み可能かどうかを事前に確認。
   * ディレクトリが書き込み不可能な場合はSystemExitを発生。
2. **ハンドラーのチェック**：
   * logging.getLogger().hasHandlers() を使用してログが適切に初期化されているかを確認。
   * もしログが利用できない場合は、標準出力にエラーメッセージを出力。

これにより、エラーハンドリングにtryを必要最小限にし、事前にエラー条件をチェックできる形に調整しました。この方法だと、意図的な範囲でエラーを防ぎやすくなります。どう思いますか？他に気になる点があれば教えてください！ 🦊✨

**あなたの発言**

以下のコードもifにしてください

# 修正バージョン

def parse\_query\_string(query\_string):

query = {}

try:

for param in query\_string.split('&'):

key, value = param.split('=')

query[key] = urllib.parse.unquote(value)

query[key] = sanitize\_input(query[key])

except Exception as e:

logging.error(f"Error parsing query string: {query\_string}, Error: {e}")

raise ValueError("Invalid query string format.")

return query

def get\_account\_list():

try:

with open('accounts.yaml', 'r') as file:

return yaml.safe\_load(file)

except FileNotFoundError as e:

logging.error(f"Error loading account list file: {e}")

raise FileNotFoundError("accounts.yaml file not found.")

except yaml.YAMLError as e:

logging.error(f"Error parsing YAML file: {e}")

raise ValueError("Error parsing accounts.yaml.")

def main():

print("Content-Type: application/json")

print()

try:

query\_string = os.getenv('QUERY\_STRING', '')

if not query\_string:

raise ValueError("QUERY\_STRING is missing.")

query = parse\_query\_string(query\_string)

action = query.get('action', '')

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

user = query.get('user', '')

password = query.get('pw', '')

clientIP = os.getenv('REMOTE\_ADDR', '')

account\_list = get\_account\_list()

account = get\_account(user, account\_list)

if account and is\_valid\_password(account, password) and is\_valid\_ip(account, clientIP) and action == 'getFile' and is\_valid\_path(account, rootDir):

# ファイル検索

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

valid\_files = [path for path in pathList if is\_valid\_filemask(account, path)]

relativePaths = [os.path.relpath(path, rootDir) for path in valid\_files]

response = {'result': True, 'pathList': relativePaths, 'clientIP': clientIP}

logging.info(f"Success: {rootDir}, {fileMask}, user: {user}, clientIP: {clientIP}, {relativePaths}")

else:

response = {'result': False, 'message': 'Invalid parameters or unauthorized access.', 'clientIP': clientIP}

logging.warning(f"Failure: rootDir={rootDir}, fileMask={fileMask}, user={user}, clientIP={clientIP}")

except Exception as e:

response = {'result': False, 'message': f"Error occurred: {e}"}

logging.error(f"Unexpected error: {e}")

print(json.dumps(response))

if \_\_name\_\_ == '\_\_main\_\_':

try:

main()

except Exception as e:

logging.critical(f"Fatal error: {e}")

print(json.dumps({'result': False, 'message': 'A critical error occurred. Check logs for details.'}))

**Copilot の発言**

了解です！例外処理を減らし、if文を活用した形に書き直します。以下が修正版です。

python

コピー

def parse\_query\_string(query\_string):

query = {}

if query\_string:

for param in query\_string.split('&'):

if '=' in param:

key, value = param.split('=')

query[key] = urllib.parse.unquote(value)

query[key] = sanitize\_input(query[key])

else:

logging.error(f"Malformed parameter in query string: {param}")

return {}

else:

logging.error("Query string is empty or missing.")

return query

def get\_account\_list():

if os.path.exists('accounts.yaml'):

with open('accounts.yaml', 'r') as file:

try:

return yaml.safe\_load(file)

except yaml.YAMLError as e:

logging.error(f"Error parsing YAML file: {e}")

return None

else:

logging.error("accounts.yaml file not found.")

return None

def main():

print("Content-Type: application/json")

print()

query\_string = os.getenv('QUERY\_STRING', '')

if not query\_string:

logging.error("QUERY\_STRING is missing.")

print(json.dumps({'result': False, 'message': "QUERY\_STRING is missing."}))

return

query = parse\_query\_string(query\_string)

if not query:

print(json.dumps({'result': False, 'message': "Invalid query parameters."}))

return

action = query.get('action', '')

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

user = query.get('user', '')

password = query.get('pw', '')

clientIP = os.getenv('REMOTE\_ADDR', '')

account\_list = get\_account\_list()

if not account\_list:

print(json.dumps({'result': False, 'message': "Failed to load account list."}))

return

account = get\_account(user, account\_list)

if account and is\_valid\_password(account, password) and is\_valid\_ip(account, clientIP) and action == 'getFile' and is\_valid\_path(account, rootDir):

# ファイル検索

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

valid\_files = [path for path in pathList if is\_valid\_filemask(account, path)]

relativePaths = [os.path.relpath(path, rootDir) for path in valid\_files]

response = {'result': True, 'pathList': relativePaths, 'clientIP': clientIP}

logging.info(f"Success: {rootDir}, {fileMask}, user: {user}, clientIP: {clientIP}, {relativePaths}")

else:

response = {'result': False, 'message': 'Invalid parameters or unauthorized access.', 'clientIP': clientIP}

logging.warning(f"Failure: rootDir={rootDir}, fileMask={fileMask}, user={user}, clientIP={clientIP}")

print(json.dumps(response))

if \_\_name\_\_ == '\_\_main\_\_':

if os.getenv('QUERY\_STRING'):

main()

else:

logging.critical("QUERY\_STRING not provided at all.")

print(json.dumps({'result': False, 'message': 'QUERY\_STRING is mandatory.'}))

**主な修正点:**

1. if**でのチェックを追加**：
   * parse\_query\_stringでは、パラメータのフォーマットが不正な場合にエラーをifで判定。
   * YAMLファイルの存在を確認してからパース処理を実行。
2. **不要な**try**文を削除**：
   * ロジックが明確である部分ではif条件を活用し、例外ではなく事前チェックで回避。

これにより、コードは簡潔かつ明確になります。この方向性で問題なければ、他の部分の追加修正もお手伝いします！✨

**あなたの発言**

以下のコードを全体的に見直してください。また、QUERY parameterにdebugを追加し、1の場合はブラウザにエラーメッセージを表示するようにしてください

**Copilot の発言**

コード全体を見直し、debug パラメータが 1 の場合にブラウザにエラーメッセージを表示する機能を追加しました。以下に修正版を示します：

python

コピー

#!/usr/bin/env python3

import os

import json

import urllib.parse

import glob

import re

import logging

import yaml

# ロギングの設定

log\_file = '/var/log/cgi-bin/API.log'

if os.access(os.path.dirname(log\_file), os.W\_OK):

logging.basicConfig(

filename=log\_file,

level=logging.INFO,

format='%(asctime)s %(message)s'

)

else:

print(json.dumps({'result': False, 'message': 'Logging directory is not writable.'}))

raise SystemExit("Critical error: Unable to initialize logging.")

# ヘルパー関数

def parse\_query\_string(query\_string):

query = {}

if query\_string:

for param in query\_string.split('&'):

if '=' in param:

key, value = param.split('=')

query[key] = urllib.parse.unquote(value)

query[key] = sanitize\_input(query[key])

else:

logging.error(f"Malformed parameter in query string: {param}")

return {}

else:

logging.error("Query string is empty or missing.")

return query

def sanitize\_input(value):

return re.sub(r'[^\w\s\-\.]', '', value) # 許可された文字のみを許可

def get\_account\_list():

if os.path.exists('accounts.yaml'):

with open('accounts.yaml', 'r') as file:

try:

return yaml.safe\_load(file)

except yaml.YAMLError as e:

logging.error(f"Error parsing YAML file: {e}")

return None

else:

logging.error("accounts.yaml file not found.")

return None

def get\_account(user, account\_list):

return next((account for account in account\_list if account['user'] == user), None)

def is\_valid\_path(account, rootDir):

abs\_path = os.path.abspath(rootDir)

return (

'../' not in abs\_path and

'./' not in abs\_path and

any(abs\_path.startswith(os.path.abspath(allowed\_dir)) for allowed\_dir in account['dirs'])

)

def is\_valid\_filemask(account, filemask):

abs\_filemask = os.path.abspath(filemask)

return not any(

re.search(pattern, abs\_filemask) for pattern in account['rejected\_file\_masks']

) and any(

re.search(pattern, abs\_filemask) for pattern in account['file\_masks']

)

def is\_valid\_ip(account, ip):

return any(re.match(pattern, ip) for pattern in account['IPaddress'])

def is\_valid\_password(account, password):

return account['password'] == password

def main():

print("Content-Type: application/json")

print()

query\_string = os.getenv('QUERY\_STRING', '')

if not query\_string:

logging.error("QUERY\_STRING is missing.")

print(json.dumps({'result': False, 'message': "QUERY\_STRING is missing."}))

return

query = parse\_query\_string(query\_string)

if not query:

print(json.dumps({'result': False, 'message': "Invalid query parameters."}))

return

debug\_mode = query.get('debug', '0') == '1'

try:

action = query.get('action', '')

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

user = query.get('user', '')

password = query.get('pw', '')

clientIP = os.getenv('REMOTE\_ADDR', '')

account\_list = get\_account\_list()

if not account\_list:

raise FileNotFoundError("Failed to load account list.")

account = get\_account(user, account\_list)

if account and is\_valid\_password(account, password) and is\_valid\_ip(account, clientIP) and action == 'getFile' and is\_valid\_path(account, rootDir):

# ファイル検索

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

valid\_files = [path for path in pathList if is\_valid\_filemask(account, path)]

relativePaths = [os.path.relpath(path, rootDir) for path in valid\_files]

response = {'result': True, 'pathList': relativePaths, 'clientIP': clientIP}

logging.info(f"Success: {rootDir}, {fileMask}, user: {user}, clientIP: {clientIP}, {relativePaths}")

else:

response = {'result': False, 'message': 'Invalid parameters or unauthorized access.', 'clientIP': clientIP}

logging.warning(f"Failure: rootDir={rootDir}, fileMask={fileMask}, user={user}, clientIP={clientIP}")

except Exception as e:

error\_message = f"Error occurred: {e}"

response = {'result': False, 'message': error\_message}

logging.error(error\_message)

if debug\_mode:

# debugモードの場合はブラウザにエラー内容を出力

response['debug'] = error\_message

print(json.dumps(response))

if \_\_name\_\_ == '\_\_main\_\_':

main()

**追加した内容:**

1. debug**パラメータのサポート**:
   * QUERY\_STRING に debug=1 を追加した場合、発生したエラーメッセージをレスポンスに含めて出力します。
   * デフォルトは debug=0（非公開）。
2. **全体的な改善**:
   * 必要な範囲でチェックを if で実装し、例外を最小限に。
   * ロギングやエラー出力を整理して理解しやすく。

この修正版はいかがですか？引き続き改善点があればお知らせください！ 🦊✨

**あなたの発言**

以下のlogパスでエラーになります。pythonのhttpサーバでcgi-bin/API\_test.pyを実行すると、API.logはどこに保存されるはずなのでしょうか

log\_file = 'API.log' #'/var/log/cgi-bin/API.log' if os.access(os.path.dirname(log\_file), os.W\_OK): logging.basicConfig( filename=log\_file, level=logging.INFO, format='%(asctime)s %(message)s' ) else: print(json.dumps({'result': False, 'message': 'Logging directory is not writable.'})) raise SystemExit("Critical error: Unable to initialize logging.")

**Copilot の発言**

The issue here lies in the relative path specified for log\_file = 'API.log'. When running this script on a Python HTTP server as a CGI script, the current working directory may vary depending on how the server is set up. This can cause unexpected behavior when resolving relative paths, such as 'API.log'.

By default, the log file 'API.log' would be created in the working directory of the Python HTTP server, which is typically the base directory where the server was started. However, if the script cannot determine the directory or lacks proper permissions, it will fail.

**Solutions to fix the issue**

1. **Specify an absolute path for the log file**: Use an absolute path to avoid ambiguity about where the file should be created. For example:

python

コピー

log\_file = '/path/to/directory/API.log'

1. **Ensure the directory exists and is writable**: Check that the directory where you intend to store the log file exists and has the proper write permissions. For example:

python

コピー

log\_dir = '/path/to/directory'

if not os.path.exists(log\_dir):

os.makedirs(log\_dir) # Create the directory if it does not exist

log\_file = os.path.join(log\_dir, 'API.log')

1. **Debugging the current directory**: To confirm where the Python HTTP server's working directory is, you can add a debug line to print the current directory:

python

コピー

import os

print("Current working directory:", os.getcwd())

1. **Fallback to the script's directory**: If you want the log file to reside in the same directory as the script itself, you can determine the script's directory dynamically:

python

コピー

script\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

log\_file = os.path.join(script\_dir, 'API.log')

1. **Handle permission errors**: Make sure the user running the Python HTTP server has write permissions for the specified directory. For instance, if the server runs as a restricted user, consider changing the directory permissions accordingly using:

bash

コピー

chmod 755 /path/to/directory

By following one of these approaches, you can control where the API.log file is stored and prevent the error from occurring. Let me know if you'd like help implementing a specific solution! 🌟

**あなたの発言**

以下のコードで、accounts.yamlを読みこんだ後、account情報をlogに出力してください

#!/usr/bin/env python3

import os

import json

import urllib.parse

import glob

import re

import logging

import yaml

# ロギングの設定

log\_file = 'd:/programs/cgi/API.log' # '/var/log/cgi-bin/API.log'

if os.access(os.path.dirname(log\_file), os.W\_OK):

logging.basicConfig(

filename=log\_file,

level=logging.INFO,

format='%(asctime)s %(message)s'

)

else:

print(json.dumps({'result': False, 'message': 'Logging directory is not writable.'}))

raise SystemExit("Critical error: Unable to initialize logging.")

# ヘルパー関数

def parse\_query\_string(query\_string):

query = {}

if query\_string:

for param in query\_string.split('&'):

if '=' in param:

key, value = param.split('=')

query[key] = urllib.parse.unquote(value)

query[key] = sanitize\_input(query[key])

else:

logging.error(f"Malformed parameter in query string: {param}")

return query

else:

logging.error("Query string is empty or missing.")

return query

def sanitize\_input(value):

return re.sub(r'[^\w\s\-\.]', '', value) # 許可された文字のみを許可

def get\_account\_list():

if os.path.exists('accounts.yaml'):

with open('accounts.yaml', 'r') as file:

try:

return yaml.safe\_load(file)

except yaml.YAMLError as e:

logging.error(f"Error parsing YAML file: {e}")

return None

else:

logging.error("accounts.yaml file not found.")

return None

def get\_account(user, account\_list):

return next((account for account in account\_list if account['user'] == user), None)

def is\_valid\_path(account, rootDir):

abs\_path = os.path.abspath(rootDir)

return (

'../' not in abs\_path and './' not in abs\_path and

any(abs\_path.startswith(os.path.abspath(allowed\_dir)) for allowed\_dir in account['dirs'])

)

def is\_valid\_filemask(account, filemask):

abs\_filemask = os.path.abspath(filemask)

return not any(

re.search(pattern, abs\_filemask) for pattern in account['rejected\_file\_masks']

) and any(

re.search(pattern, abs\_filemask) for pattern in account['file\_masks']

)

def is\_valid\_ip(account, ip):

return any(re.match(pattern, ip) for pattern in account['IPaddress'])

def is\_valid\_password(account, password):

return account['password'] == password

def main():

print("Content-Type: application/json")

print()

query\_string = os.getenv('QUERY\_STRING', '')

if not query\_string:

logging.error("QUERY\_STRING is missing.")

print(json.dumps({'result': False, 'message': "QUERY\_STRING is missing."}))

return

query = parse\_query\_string(query\_string)

if not query:

print(json.dumps({'result': False, 'message': "Invalid query parameters."}))

return

debug\_mode = query.get('debug', '0') == '1'

try:

action = query.get('action', '')

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

user = query.get('user', '')

password = query.get('pw', '')

clientIP = os.getenv('REMOTE\_ADDR', '')

account\_list = get\_account\_list()

if not account\_list:

raise FileNotFoundError("Failed to load account list.")

account = get\_account(user, account\_list)

if account and is\_valid\_password(account, password) and is\_valid\_ip(account, clientIP) and action == 'getFile' and is\_valid\_path(account, rootDir):

# ファイル検索

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

valid\_files = [path for path in pathList if is\_valid\_filemask(account, path)]

relativePaths = [os.path.relpath(path, rootDir) for path in valid\_files]

response = {'result': True, 'pathList': relativePaths, 'clientIP': clientIP}

logging.info(f"Success: {rootDir}, {fileMask}, user: {user}, clientIP: {clientIP}, {relativePaths}")

else:

response = {'result': False, 'message': 'Invalid parameters or unauthorized access.', 'clientIP': clientIP}

logging.warning(f"Failure: rootDir={rootDir}, fileMask={fileMask}, user={user}, clientIP={clientIP}")

except Exception as e:

error\_message = f"Error occurred: {e}"

response = {'result': False, 'message': error\_message}

logging.error(error\_message)

if debug\_mode:

# debugモードの場合はブラウザにエラー内容を出力

response['debug'] = error\_message

print(json.dumps(response))

if \_\_name\_\_ == '\_\_main\_\_':

main()

**Copilot の発言**

To log the loaded account information from accounts.yaml, you can modify the get\_account\_list function to include a logging statement after successfully loading the file. Here is the updated code:

python

コピー

def get\_account\_list():

if os.path.exists('accounts.yaml'):

with open('accounts.yaml', 'r') as file:

try:

accounts = yaml.safe\_load(file)

logging.info(f"Accounts loaded successfully: {accounts}")

return accounts

except yaml.YAMLError as e:

logging.error(f"Error parsing YAML file: {e}")

return None

else:

logging.error("accounts.yaml file not found.")

return None

**Changes:**

1. **Logging Account Information**:
   * After successfully loading accounts.yaml, a logging statement (logging.info) is added to record the content of the file.
2. **Readable Log Entry**:
   * This log entry helps track the accounts that are being loaded, which can be useful for debugging purposes.

**Notes:**

* Ensure the content of accounts.yaml does not contain sensitive information like passwords or private details before logging the entire data. If sensitive information is present, consider logging only a summary or specific non-sensitive fields. For example:

python

コピー

logging.info(f"Accounts loaded successfully. Users: {[account['user'] for account in accounts]}")

Let me know if you'd like additional modifications or safeguards! 🦊✨

**あなたの発言**

以下のmainではtryに全て入っていて、エラーの特定ができません。query parameterが与えられていない場合の処理をif文で個別に処理してください

def main():

print("Content-Type: application/json")

print()

query\_string = os.getenv('QUERY\_STRING', '')

if not query\_string:

logging.error("QUERY\_STRING is missing.")

print(json.dumps({'result': False, 'message': "QUERY\_STRING is missing."}))

return

query = parse\_query\_string(query\_string)

if not query:

print(json.dumps({'result': False, 'message': "Invalid query parameters."}))

return

debug\_mode = query.get('debug', '0') == '1'

try:

action = query.get('action', '')

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

user = query.get('user', '')

password = query.get('pw', '')

clientIP = os.getenv('REMOTE\_ADDR', '')

account\_list = get\_account\_list()

if not account\_list:

raise FileNotFoundError("Failed to load account list.")

account = get\_account(user, account\_list)

if account and is\_valid\_password(account, password) and is\_valid\_ip(account, clientIP) and action == 'getFile' and is\_valid\_path(account, rootDir):

# ファイル検索

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

valid\_files = [path for path in pathList if is\_valid\_filemask(account, path)]

relativePaths = [os.path.relpath(path, rootDir) for path in valid\_files]

response = {'result': True, 'pathList': relativePaths, 'clientIP': clientIP}

logging.info(f"Success: {rootDir}, {fileMask}, user: {user}, clientIP: {clientIP}, {relativePaths}")

else:

response = {'result': False, 'message': 'Invalid parameters or unauthorized access.', 'clientIP': clientIP}

logging.warning(f"Failure: rootDir={rootDir}, fileMask={fileMask}, user={user}, clientIP={clientIP}")

except Exception as e:

error\_message = f"Error occurred: {e}"

response = {'result': False, 'message': error\_message}

logging.error(error\_message)

if debug\_mode:

# debugモードの場合はブラウザにエラー内容を出力

response['debug'] = error\_message

print(json.dumps(response))

if \_\_name\_\_ == '\_\_main\_\_':

main()

**Copilot の発言**

以下にmain関数を改良し、try-exceptブロックを最小限にしつつ、QUERY\_STRINGが欠落している場合などをif文で個別に処理する形に変更しました。

python

コピー

def main():

print("Content-Type: application/json")

print()

# QUERY\_STRINGが存在するか確認

query\_string = os.getenv('QUERY\_STRING', '')

if not query\_string:

logging.error("QUERY\_STRING is missing.")

print(json.dumps({'result': False, 'message': "QUERY\_STRING is missing."}))

return

# クエリパラメータの解析

query = parse\_query\_string(query\_string)

if not query:

print(json.dumps({'result': False, 'message': "Invalid query parameters."}))

return

# debugモードの確認

debug\_mode = query.get('debug', '0') == '1'

# 各パラメータを取得

action = query.get('action', '')

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

user = query.get('user', '')

password = query.get('pw', '')

clientIP = os.getenv('REMOTE\_ADDR', '')

# アカウントリストの取得

account\_list = get\_account\_list()

if not account\_list:

logging.error("Failed to load account list.")

print(json.dumps({'result': False, 'message': "Failed to load account list."}))

return

# ユーザーのアカウント情報を取得

account = get\_account(user, account\_list)

if not account:

logging.warning(f"Unauthorized user access attempt: user={user}")

print(json.dumps({'result': False, 'message': "Invalid user or unauthorized access."}))

return

# パスワード確認

if not is\_valid\_password(account, password):

logging.warning(f"Invalid password attempt for user: {user}")

print(json.dumps({'result': False, 'message': "Invalid password."}))

return

# IPアドレス確認

if not is\_valid\_ip(account, clientIP):

logging.warning(f"Unauthorized IP access attempt for user: {user}, IP: {clientIP}")

print(json.dumps({'result': False, 'message': "Unauthorized IP address."}))

return

# アクション確認とルートディレクトリ確認

if action == 'getFile' and is\_valid\_path(account, rootDir):

try:

# ファイル検索

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

valid\_files = [path for path in pathList if is\_valid\_filemask(account, path)]

relativePaths = [os.path.relpath(path, rootDir) for path in valid\_files]

response = {'result': True, 'pathList': relativePaths, 'clientIP': clientIP}

logging.info(f"Success: rootDir={rootDir}, fileMask={fileMask}, user={user}, clientIP={clientIP}, {relativePaths}")

except Exception as e:

logging.error(f"Error during file search: {e}")

response = {'result': False, 'message': f"Error during file search: {e}"}

if debug\_mode:

response['debug'] = str(e)

else:

response = {'result': False, 'message': "Invalid parameters or unauthorized access."}

logging.warning(f"Failure: rootDir={rootDir}, fileMask={fileMask}, user={user}, clientIP={clientIP}")

print(json.dumps(response))

**改善点**

1. **個別のエラーチェック**:
   * QUERY\_STRINGがない場合やクエリの解析が失敗した場合、適切にifでエラーを検出して処理。
   * アカウントやパスワード、IPの確認を個別に行い、それぞれログと応答メッセージを用意。
2. **最小限の**try-except**ブロック**:
   * ファイル検索処理に限定してtry-exceptを使用し、エラーの範囲を明確化。
3. **デバッグモードのサポート**:
   * debug=1がクエリパラメータに含まれている場合、エラーメッセージをレスポンスに含むよう変更。

これによりエラーの特定が容易になり、デバッグやログ解析がしやすくなります。この形で要望に合っていますか？他にも改善したい点があれば教えてください！ 🌟✨

**あなたの発言**

json.dumps()は、エラー時にブラウザにも出力されます。読みやすく整形できませんか

**Copilot の発言**

json.dumps()で出力する際に読みやすい整形を行うには、indentパラメータを利用できます。これにより、JSONを整形して人間にとって見やすい形式で表示できます。以下のように修正することで、エラー時のブラウザ表示が整形されます。

python

コピー

import json

def main():

print("Content-Type: application/json")

print()

query\_string = os.getenv('QUERY\_STRING', '')

if not query\_string:

logging.error("QUERY\_STRING is missing.")

# 整形されたJSONをブラウザに出力

print(json.dumps({'result': False, 'message': "QUERY\_STRING is missing."}, indent=4))

return

query = parse\_query\_string(query\_string)

if not query:

# 整形されたJSONをブラウザに出力

print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

return

debug\_mode = query.get('debug', '0') == '1'

try:

# ここにメインロジックの処理

response = {'result': True, 'message': "Success."}

logging.info("Processing completed successfully.")

except Exception as e:

error\_message = f"Error occurred: {e}"

response = {'result': False, 'message': error\_message}

logging.error(error\_message)

if debug\_mode:

response['debug'] = error\_message

# 最終的に整形されたJSONを出力

print(json.dumps(response, indent=4))

if \_\_name\_\_ == '\_\_main\_\_':

main()

**主な変更点**

1. indent=4**の追加**:
   * json.dumps(response, indent=4) の形式で出力することで、JSONが整形され、インデント付きで表示されます。
2. **エラー時も読みやすい出力**:
   * すべてのエラーメッセージがブラウザで整形されて表示されるため、デバッグや問題の特定が容易になります。
3. **デバッグモードにも対応**:
   * デバッグモードで出力されるJSONも同様に整形されます。

これにより、エラーや応答メッセージがすっきりと見やすくなります！ 🦊✨ ぜひ試してみてください。もし他にカスタマイズしたい部分があれば教えてくださいね。

**あなたの発言**

ブラウザに出力される際は、改行もされません。<pre>タグをつけてprintできませんか

**Copilot の発言**

もちろんです！<pre>タグを使用すれば、ブラウザに出力されるJSONが整形され、改行やインデントが反映されるようになります。以下に修正版の例を示します：

python

コピー

def main():

print("Content-Type: text/html") # JSONではなくHTMLコンテンツとして表示

print()

query\_string = os.getenv('QUERY\_STRING', '')

if not query\_string:

logging.error("QUERY\_STRING is missing.")

# 整形されたJSONを<pre>タグで囲んで出力

print("<pre>")

print(json.dumps({'result': False, 'message': "QUERY\_STRING is missing."}, indent=4))

print("</pre>")

return

query = parse\_query\_string(query\_string)

if not query:

# 整形されたJSONを<pre>タグで囲んで出力

print("<pre>")

print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

print("</pre>")

return

debug\_mode = query.get('debug', '0') == '1'

try:

# ここにメインロジックの処理

response = {'result': True, 'message': "Success."}

logging.info("Processing completed successfully.")

except Exception as e:

error\_message = f"Error occurred: {e}"

response = {'result': False, 'message': error\_message}

logging.error(error\_message)

if debug\_mode:

response['debug'] = error\_message

# 最終的に整形されたJSONを<pre>タグで囲んで出力

print("<pre>")

print(json.dumps(response, indent=4))

print("</pre>")

if \_\_name\_\_ == '\_\_main\_\_':

main()

**主な変更点：**

1. Content-Type**を**text/html**に変更**:
   * ブラウザがHTML形式で内容を解釈するため。
2. <pre>**タグで囲む**:
   * JSONデータを<pre>タグ内に出力することで、ブラウザで整形表示されます。
3. json.dumps**でインデント設定**:
   * indent=4 を追加して読みやすい形式に整形。

**出力例：**

ブラウザで以下のように見えるはずです：

html

コピー

<pre>

{

"result": false,

"message": "QUERY\_STRING is missing."

}

</pre>

これでブラウザにも見やすい形で表示されるようになります！この形で問題ないか試してみてくださいね。 🦊✨

**あなたの発言**

importlibをつかってprintをoverrideし、<pre>を使うように

**Copilot の発言**

importlib を使用して print をオーバーライドし、すべての出力が <pre> タグ内に表示されるように設定できます。以下は例です：

python

コピー

import importlib

import builtins

# オーバーライドする print 関数

def custom\_print(\*args, \*\*kwargs):

# 元の print を呼び出す前に <pre> タグを追加

if not getattr(custom\_print, "pre\_opened", False):

builtins.print("<pre>")

setattr(custom\_print, "pre\_opened", True)

# 元の print を実行

builtins.print(\*args, \*\*kwargs)

# 最後の <pre> を閉じる

if kwargs.get("end", "\n")[-1] == "\n": # 改行があれば閉じる

builtins.print("</pre>")

setattr(custom\_print, "pre\_opened", False)

# print をカスタム版に置き換え

builtins.print = custom\_print

# main関数例

def main():

print("Content-Type: text/html")

print() # 空行を出力

# サンプルのJSONレスポンス

response = {

"result": True,

"message": "This is a test response.",

}

# 整形された JSON を出力

import json

print(json.dumps(response, indent=4))

if \_\_name\_\_ == "\_\_main\_\_":

main()

**実装のポイント：**

1. custom\_print**関数**：
   * オリジナルの print 関数を拡張し、すべての出力を <pre> タグでラップします。
   * タグの重複を防ぐために、custom\_print の属性として pre\_opened を追跡しています。
2. builtins.print**の置き換え**：
   * builtins.print をカスタム関数に置き換えることで、すべての print 呼び出しを自動的に整形。
3. **JSONの出力**：
   * json.dumps と indent=4 を使用し、整形されたJSONを生成。

**実行結果：**

ブラウザ上では次のようなHTML出力が得られます：

html

コピー

<pre>

{

"result": true,

"message": "This is a test response."

}

</pre>

これで、すべての print 出力が自動的に <pre> タグで囲まれるようになりました。この方法で目的に合いますか？🌟

**あなたの発言**

custom\_printではなく、print関数自体を定義しなおしてください。もとのprintはprint\_original変数に退避

**Copilot の発言**

以下は、print 関数を再定義し、元の print を print\_original に退避する方法です。このカスタム print 関数は、出力を <pre> タグ内にラップします。

python

コピー

import builtins

# 元の print 関数を退避

print\_original = builtins.print

# 新しい print 関数を定義

def print(\*args, \*\*kwargs):

# `<pre>` タグを追加して出力

print\_original("<pre>")

print\_original(\*args, \*\*kwargs)

print\_original("</pre>")

# 組み込みの print を新しいものに置き換え

builtins.print = print

# main 関数例

def main():

print("Content-Type: text/html")

print() # 空行を出力

# サンプルのJSONレスポンス

response = {

"result": True,

"message": "This is a test response.",

}

# 整形された JSON を出力

import json

print(json.dumps(response, indent=4))

if \_\_name\_\_ == "\_\_main\_\_":

main()

**説明：**

1. **退避された元の**print:
   * print\_original に組み込みの print を保持しておくことで、再定義後も元の print にアクセスできます。
2. **再定義された**print:
   * 新しい print 関数では、引数を受け取り、その内容を <pre> タグで囲んで表示する仕組みを構築。
3. builtins.print**の置き換え**:
   * 組み込みの print を再定義することで、すべての print 呼び出しがカスタム動作を利用します。
4. **JSONの整形表示**:
   * json.dumps の indent=4 により、整形された出力が <pre> タグ内に表示されます。

**実行結果：**

ブラウザで以下のようなHTML出力が得られます：

html

コピー

<pre>

{

"result": true,

"message": "This is a test response."

}

</pre>

この方法で出力が見やすくなります！さらに改善が必要であれば、教えてくださいね。✨

**あなたの発言**

以下のプログラムで、get\_filesを呼び出す際にデコレータ@validate\_accountを使い、is\_valid\_account、is\_valid\_password、is\_valid\_ipのチェックを入れてください。

#!/usr/bin/env python3 import os import builtins import json import urllib.parse import glob import re import logging import yaml # 元の print 関数を退避 print\_original = builtins.print # 新しい print 関数を定義 def print\_custom(\*args, \*\*kwargs): # `<pre>` タグを追加して出力 # print\_original("<pre>") print\_original(\*args, \*\*kwargs) # print\_original("</pre>") # 組み込みの print を新しいものに置き換え builtins.print = print\_custom log\_file = 'd:/programs/cgi/API.log' account\_path = 'd:/programs/cgi/cgi-bin/accounts.yaml' if os.access(os.path.dirname(log\_file), os.W\_OK): logging.basicConfig( filename=log\_file, level=logging.INFO, format='%(asctime)s %(message)s' ) else: print(json.dumps({'result': False, 'message': 'Logging directory is not writable.'}, indent=4)) raise SystemExit("Critical error: Unable to initialize logging.") # ヘルパー関数 def parse\_query\_string(query\_string): query = {} if query\_string: for param in query\_string.split('&'): if '=' in param: key, value = param.split('=') query[key] = urllib.parse.unquote(value) query[key] = sanitize\_input(query[key]) else: logging.error(f"Malformed parameter in query string: {param}") return {} else: logging.error("Query string is empty or missing.") return query def sanitize\_input(value): return re.sub(r'[^\w\s\-\.]', '', value) # 許可された文字のみを許可 def get\_account\_list(): logging.info(f"get\_account\_list: reading {account\_path}") if os.path.exists(account\_path): with open(account\_path, 'r') as file: try: accounts = yaml.safe\_load(file) logging.info(f"Accounts loaded successfully: {accounts}") return accounts except yaml.YAMLError as e: logging.error(f"Error parsing YAML file: {e}") return None else: logging.error(f"{account\_path} file not found.") return None def get\_account(user, account\_list): return next((account for account in account\_list if account['user'] == user), None) def is\_valid\_path(account, rootDir): abs\_path = os.path.abspath(rootDir) return ( '../' not in abs\_path and './' not in abs\_path and any(abs\_path.startswith(os.path.abspath(allowed\_dir)) for allowed\_dir in account['dirs']) ) def is\_valid\_filemask(account, filemask): abs\_filemask = os.path.abspath(filemask) return not any( re.search(pattern, abs\_filemask) for pattern in account['rejected\_file\_masks'] ) and any( re.search(pattern, abs\_filemask) for pattern in account['file\_masks'] ) def is\_valid\_ip(account, ip): return any(re.match(pattern, ip) for pattern in account['IPaddress']) def is\_valid\_password(account, password): return account['password'] == password def main(): print("Content-Type: application/json") print() # QUERY\_STRINGが存在するか確認 query\_string = os.getenv('QUERY\_STRING', '') if not query\_string: logging.error("QUERY\_STRING is missing.") print(json.dumps({'result': False, 'message': "QUERY\_STRING is missing."}, indent=4)) return # クエリパラメータの解析 query = parse\_query\_string(query\_string) if not query: print(json.dumps({'result': False, 'message': "Invalid query parameters."})) return # debugモードの確認 debug\_mode = query.get('debug', '0') == '1' # 各パラメータを取得 action = query.get('action', '') rootDir = query.get('rootDir', '') fileMask = query.get('fileMask', '') user = query.get('user', '') password = query.get('pw', '') clientIP = os.getenv('REMOTE\_ADDR', '') # アカウントリストの取得 account\_list = get\_account\_list() if not account\_list: logging.error("Failed to load account list.") print(json.dumps({'result': False, 'message': "Failed to load account list."}, indent=4)) return # ユーザーのアカウント情報を取得 account = get\_account(user, account\_list) if not account: logging.warning(f"Unauthorized user access attempt: user={user}") print(json.dumps({'result': False, 'message': "Invalid user or unauthorized access."}, indent=4)) return # パスワード確認 if not is\_valid\_password(account, password): logging.warning(f"Invalid password attempt for user: {user}") print(json.dumps({'result': False, 'message': "Invalid password."}, indent=4)) return # IPアドレス確認 if not is\_valid\_ip(account, clientIP): logging.warning(f"Unauthorized IP access attempt for user: {user}, IP: {clientIP}") print(json.dumps({'result': False, 'message': "Unauthorized IP address."}, indent=4)) return # アクション確認とルートディレクトリ確認 if action == 'getFile' and is\_valid\_path(account, rootDir): try: # ファイル検索 pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask)) valid\_files = [path for path in pathList if is\_valid\_filemask(account, path)] relativePaths = [os.path.relpath(path, rootDir) for path in valid\_files] response = {'result': True, 'pathList': relativePaths, 'clientIP': clientIP} logging.info(f"Success: rootDir={rootDir}, fileMask={fileMask}, user={user}, clientIP={clientIP}, {relativePaths}") except Exception as e: logging.error(f"Error during file search: {e}") response = {'result': False, 'message': f"Error during file search: {e}"} if debug\_mode: response['debug'] = str(e) else: response = {'result': False, 'message': "Invalid parameters or unauthorized access."} logging.warning(f"Failure: rootDir={rootDir}, fileMask={fileMask}, user={user}, clientIP={clientIP}") print(json.dumps(response), indent=4) if \_\_name\_\_ == '\_\_main\_\_': main()

**Copilot の発言**

以下は、@validate\_accountデコレータを追加し、get\_files関数にアカウントやパスワード、IPのチェックを実装した例です。このデコレータを利用することで、各チェックを簡潔に実行できます。

python

コピー

# デコレータ関数

def validate\_account(func):

def wrapper(\*args, \*\*kwargs):

# QUERY\_STRINGから必要な情報を取得

query = kwargs.get('query', {})

user = query.get('user', '')

password = query.get('pw', '')

clientIP = os.getenv('REMOTE\_ADDR', '')

# アカウントリストの取得

account\_list = get\_account\_list()

if not account\_list:

logging.error("Failed to load account list.")

print(json.dumps({'result': False, 'message': "Failed to load account list."}, indent=4))

return

# アカウント取得

account = get\_account(user, account\_list)

if not account:

logging.warning(f"Unauthorized user access attempt: user={user}")

print(json.dumps({'result': False, 'message': "Invalid user or unauthorized access."}, indent=4))

return

# パスワードの確認

if not is\_valid\_password(account, password):

logging.warning(f"Invalid password attempt for user: {user}")

print(json.dumps({'result': False, 'message': "Invalid password."}, indent=4))

return

# IPアドレスの確認

if not is\_valid\_ip(account, clientIP):

logging.warning(f"Unauthorized IP access attempt for user: {user}, IP: {clientIP}")

print(json.dumps({'result': False, 'message': "Unauthorized IP address."}, indent=4))

return

# アカウントが有効であれば関数を実行

return func(account, \*args, \*\*kwargs)

return wrapper

# ファイル検索関数

@validate\_account

def get\_files(account, query):

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

# パスとマスクのバリデーション

if not is\_valid\_path(account, rootDir):

logging.warning(f"Invalid rootDir access attempt: {rootDir}")

print(json.dumps({'result': False, 'message': "Invalid rootDir."}, indent=4))

return

try:

# ファイル検索

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

valid\_files = [path for path in pathList if is\_valid\_filemask(account, path)]

relativePaths = [os.path.relpath(path, rootDir) for path in valid\_files]

response = {'result': True, 'pathList': relativePaths}

logging.info(f"Success: rootDir={rootDir}, fileMask={fileMask}, user={account['user']}, paths={relativePaths}")

print(json.dumps(response, indent=4))

except Exception as e:

logging.error(f"Error during file search: {e}")

print(json.dumps({'result': False, 'message': f"Error during file search: {e}"}, indent=4))

# main関数

def main():

print("Content-Type: application/json")

print()

# QUERY\_STRINGが存在するか確認

query\_string = os.getenv('QUERY\_STRING', '')

if not query\_string:

logging.error("QUERY\_STRING is missing.")

print(json.dumps({'result': False, 'message': "QUERY\_STRING is missing."}, indent=4))

return

# クエリパラメータの解析

query = parse\_query\_string(query\_string)

if not query:

print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

return

# debugモードの確認

debug\_mode = query.get('debug', '0') == '1'

# アクションの確認

action = query.get('action', '')

if action == 'getFile':

# get\_files関数を呼び出し

get\_files(query=query)

else:

print(json.dumps({'result': False, 'message': "Invalid action."}, indent=4))

if \_\_name\_\_ == "\_\_main\_\_":

main()

**改善点**

1. **デコレータによる検証**：
   * @validate\_accountで、ユーザー、パスワード、IPアドレスの確認を共通化し、コードの重複を削減。
2. **クエリ情報の渡し方**：
   * get\_files関数にqueryを引数として渡し、必要な処理を実行。
3. **エラーメッセージの整理**：
   * JSONのレスポンスがすべて整形されるように indent=4 を適用。

**主なフロー：**

1. mainでアクションとクエリを確認。
2. get\_filesが呼ばれると、@validate\_accountで認証および検証を実施。
3. 検証成功後、ファイル検索処理を実行。

この構造で可読性が向上し、エラー管理もしやすくなります！🐾✨

**あなたの発言**

query parameterを以下のように解析しています。actionが与えられたら、actionと応答関数をつなげるデコレータ@route(actgion)を使い、routing処理ができますか

def parse\_query\_string(query\_string): query = {} if query\_string: for param in query\_string.split('&'): if '=' in param: key, value = param.split('=') query[key] = urllib.parse.unquote(value) query[key] = sanitize\_input(query[key]) else: logging.error(f"Malformed parameter in query string: {param}") return {} else: logging.error("Query string is empty or missing.") return query

**Copilot の発言**

はい、@route(action) デコレータを使ったルーティング処理を実現できます！以下にその実装方法の例を示します。このデコレータは、指定された action に応じて、対応する関数を実行する仕組みを提供します。

python

コピー

import functools

import logging

# ルートマッピング辞書

routes = {}

# @route デコレータ

def route(action):

def decorator(func):

# routes辞書にアクションと関数を関連付け

routes[action] = func

@functools.wraps(func)

def wrapper(\*args, \*\*kwargs):

return func(\*args, \*\*kwargs)

return wrapper

return decorator

# クエリ解析関数（そのまま使用）

def parse\_query\_string(query\_string):

query = {}

if query\_string:

for param in query\_string.split('&'):

if '=' in param:

key, value = param.split('=')

query[key] = urllib.parse.unquote(value)

query[key] = sanitize\_input(query[key])

else:

logging.error(f"Malformed parameter in query string: {param}")

return {}

else:

logging.error("Query string is empty or missing.")

return query

def sanitize\_input(value):

return re.sub(r'[^\w\s\-\.]', '', value)

# アクション用関数の定義

@route('getFile')

def get\_file(query):

# サンプル処理（ファイル検索など）

response = {

'result': True,

'message': 'Executing getFile action!',

'parameters': query

}

print(json.dumps(response, indent=4))

@route('uploadFile')

def upload\_file(query):

# サンプル処理（ファイルアップロードなど）

response = {

'result': True,

'message': 'Executing uploadFile action!',

'parameters': query

}

print(json.dumps(response, indent=4))

# ルーティング処理を行う関数

def process\_action(query):

action = query.get('action')

if action in routes:

# ルートに登録されている関数を実行

return routes[action](query)

else:

# 未定義のアクションの場合

response = {

'result': False,

'message': f"Action '{action}' is not supported.",

}

print(json.dumps(response, indent=4))

# メイン関数

def main():

print("Content-Type: application/json")

print()

# QUERY\_STRINGの取得と解析

query\_string = os.getenv('QUERY\_STRING', '')

if not query\_string:

logging.error("QUERY\_STRING is missing.")

print(json.dumps({'result': False, 'message': "QUERY\_STRING is missing."}, indent=4))

return

query = parse\_query\_string(query\_string)

if not query:

print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

return

# ルーティング処理を呼び出し

process\_action(query)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**動作のポイント**

1. routes**辞書**:
   * アクション名をキー、対応する関数を値として登録することで、簡単にルーティングを実現。
2. @route(action)**デコレータ**:
   * 関数を登録する際に routes にアクション名を対応付け。
   * 必要な関数の定義だけで、ルーティングの追加が可能。
3. process\_action(query):
   * クエリから action を取得し、対応する関数を routes から呼び出す。
   * 未登録アクションの場合は、エラーメッセージを返す。
4. **複数のアクション**:
   * @route('getFile') や @route('uploadFile') のようにデコレータを使い、必要なアクションを簡潔に追加可能。

**実行例**

もし QUERY\_STRING が以下のようであれば：

コピー

action=getFile&user=test

結果：

json

コピー

{

"result": true,

"message": "Executing getFile action!",

"parameters": {

"action": "getFile",

"user": "test"

}

}

これで柔軟なルーティング処理が実現できます！他に気になる点や追加したいアクションがあれば教えてくださいね。✨

**あなたの発言**

デコレータを2つ使う場合は以下で良いですか @route('getFile') @validate\_account def get\_files(account, query):

**Copilot の発言**

はい、そのようにデコレータを2つ使用する形で問題ありません！Pythonでは複数のデコレータをスタックさせて使うことができ、上から順に適用されます。

以下の順序で実行されます：

1. @validate\_account が最初に適用され、その処理が行われます（アカウント検証）。
2. 次に、@route('getFile') が適用され、ルーティング処理が行われます。

これによって、処理の流れは次のようになります：

* まずアカウントの有効性を確認。
* 有効であればルーティング関数が実行されます。

**実装例**

python

コピー

@route('getFile')

@validate\_account

def get\_files(account, query):

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

# バリデーションチェック

if not is\_valid\_path(account, rootDir):

logging.warning(f"Invalid rootDir access attempt: {rootDir}")

print(json.dumps({'result': False, 'message': "Invalid rootDir."}, indent=4))

return

try:

# ファイル検索処理

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

valid\_files = [path for path in pathList if is\_valid\_filemask(account, path)]

relativePaths = [os.path.relpath(path, rootDir) for path in valid\_files]

response = {'result': True, 'pathList': relativePaths}

logging.info(f"Success: rootDir={rootDir}, fileMask={fileMask}, user={account['user']}, paths={relativePaths}")

print(json.dumps(response, indent=4))

except Exception as e:

logging.error(f"Error during file search: {e}")

print(json.dumps({'result': False, 'message': f"Error during file search: {e}"}, indent=4))

**デコレータの順序に注意**

デコレータは上から順に適用されます。この例では以下のように動作します：

1. @validate\_account: 関数の実行前に、アカウント情報（ユーザー、パスワード、IP）を検証。
2. @route('getFile'): ルーティング処理を適用し、該当の関数（ここでは get\_files）を適切なタイミングで呼び出します。

**補足**

ルーティング処理でクエリを取得し、バリデーションを実施する設計は拡張性が高く、他のアクションにも同じ方法で簡単に適用可能です。

これで使いやすい構造になっていますが、さらにカスタマイズが必要なら教えてくださいね！ 🦊✨

**あなたの発言**

デコレーションの順序は、認証のあとでルーティングで大丈夫ですか

**Copilot の発言**

デコレーションの順序については、基本的には **「ルーティング（@route）→ 認証（@validate\_account）」** の順番が推奨されます。これにより、次のような利点があります：

1. **無駄な認証処理を避ける**: ルーティングで action を確認し、無効なアクションの場合はすぐ処理を終了することで、無駄に認証処理を走らせる必要がなくなります。
2. **スコープを限定する**: ルーティングが先に処理されることで、該当するアクションのみを対象に認証処理を実行できます。

例えば：

python

コピー

@route('getFile')

@validate\_account

def get\_files(account, query):

# 実行される処理

pass

ただし、**現在の順序（認証→ルーティング）も動作する場合があります**。次のような場合には、この順序も許容されます：

* アクションごとに特別な認証条件がない場合。
* 認証が、関数そのものではなく、事前にユーザーやリソースのアクセス権をチェックするだけの場合。

最終的には、どちらの順序を使うかは以下で決まります：

* 認証をどの粒度で行いたいか（全体 or 各アクションごと）。
* アクションが無効な場合でも認証を行うべきかどうか。

**おすすめの順序**としては、一般的なケースでは以下です：

1. @route で処理を特定。
2. @validate\_account で認証を実行。

何か特定の要件や状況があれば、それに応じたアプローチをお手伝いします！✨

**あなたの発言**

先にrouteをしたほうが良いようです。ありがとうございます

**Copilot の発言**

おっしゃる通り、先にルーティングを行うことで、不要な認証処理を回避でき、効率的かつ管理がしやすくなりますね！こちらこそ、確認させていただきありがとうございます。✨

また何か相談がありましたら、気軽に声をかけてくださいね。頑張っているあなたを応援しています！🦊🍂

**あなたの発言**

Flaskとデコレータレベルで互換性のあるCGIフレームワークを作っています。@routeと@validate\_acountに対応するFlaskのデコレータは何でしょうか。また、他に便利で実装したほうがいいデコレータはありますか？

**Copilot の発言**

面白い取り組みですね！Flaskとデコレータレベルで互換性のあるCGIフレームワークを作る際に参考となる情報を以下にまとめました。

**Flaskのデコレータ対応例**

1. @route**に対応する Flask のデコレータ**：
   * Flaskでは、ルーティングを行うために @app.route() デコレータが使用されます。
   * Flaskの @app.route('/path') と同等の機能を実現したい場合、@route を作成して path-to-function のマッピングを行います。

python

コピー

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/getFile')

def get\_file():

return "This is the getFile route!"

1. @validate\_account**に対応するデコレータ**：
   * Flaskでは認証や検証処理に @before\_request や @after\_request、またはカスタムミドルウェアを使用します。ただし、個別ルートごとに適用したい場合、独自のデコレータを作成できます。

python

コピー

from flask import request, jsonify

def validate\_account(func):

def wrapper(\*args, \*\*kwargs):

user = request.args.get('user')

password = request.args.get('pw')

client\_ip = request.remote\_addr

# ユーザーバリデーションロジック

if not user or not password or client\_ip != "127.0.0.1": # 簡易例

return jsonify({'result': False, 'message': 'Unauthorized access.'}), 403

return func(\*args, \*\*kwargs)

return wrapper

@app.route('/getFile')

@validate\_account

def get\_file():

return jsonify({'result': True, 'message': 'Access granted!'})

**他に便利で実装すべきデコレータ**

1. @log\_activity
   * 各リクエストのアクティビティやエラーをログに記録するためのデコレータ。

python

コピー

def log\_activity(func):

def wrapper(\*args, \*\*kwargs):

logging.info(f"Accessing {func.\_\_name\_\_} with args: {args}, kwargs: {kwargs}")

try:

result = func(\*args, \*\*kwargs)

logging.info(f"Execution successful: {func.\_\_name\_\_}")

return result

except Exception as e:

logging.error(f"Error in {func.\_\_name\_\_}: {e}")

raise

return wrapper

1. @cache
   * ルートの結果をキャッシュするデコレータ（シンプルなキャッシュシステムやFlaskの拡張を活用可能）。

python

コピー

from functools import lru\_cache

@lru\_cache(maxsize=128)

def cached\_function(param):

# キャッシュ可能な重い処理

return param \* 2

@app.route('/cachedRoute/<int:input\_value>')

def cached\_route(input\_value):

result = cached\_function(input\_value)

return jsonify({'result': result})

1. @check\_rate\_limit
   * レート制限を実現するためのデコレータ。例えば、一定の秒数以内にアクセスを制限する。

python

コピー

from flask import jsonify

import time

rate\_limit = {}

def check\_rate\_limit(limit\_per\_second):

def decorator(func):

def wrapper(\*args, \*\*kwargs):

user\_ip = request.remote\_addr

current\_time = time.time()

if user\_ip in rate\_limit:

last\_access\_time = rate\_limit[user\_ip]

if current\_time - last\_access\_time < 1 / limit\_per\_second:

return jsonify({'result': False, 'message': 'Rate limit exceeded'}), 429

rate\_limit[user\_ip] = current\_time

return func(\*args, \*\*kwargs)

return wrapper

return decorator

@app.route('/rateLimited')

@check\_rate\_limit(limit\_per\_second=1) # 1リクエスト/秒

def rate\_limited():

return jsonify({'result': True, 'message': 'Welcome!'})

1. @require\_json
   * 受け取るリクエストがJSONフォーマットであることをチェックする。

python

コピー

from flask import request, jsonify

def require\_json(func):

def wrapper(\*args, \*\*kwargs):

if not request.is\_json:

return jsonify({'result': False, 'message': 'JSON required'}), 400

return func(\*args, \*\*kwargs)

return wrapper

@app.route('/jsonEndpoint', methods=['POST'])

@require\_json

def json\_endpoint():

data = request.get\_json()

return jsonify({'result': True, 'received': data})

**カスタムCGIフレームワークのデコレータ設計のポイント**

* **再利用性**を重視：デコレータを使うことで、認証やルーティング、エラーハンドリングなどの処理を分離できます。
* **小さな単位に分割**：1つのデコレータが多くの責任を持たず、単一の目的を達成するよう設計しましょう。
* **柔軟性の確保**：パラメータを受け取れるようにして、用途に応じたカスタマイズが可能になると便利です。

Flaskの概念を活かしたCGIフレームワークを作るのは素晴らしい挑戦ですね！他に具体的な要件や拡張のアイデアがあれば、ぜひ教えてください。一緒に考えましょう！🌟🦊✨

**あなたの発言**

以下のコードAPI.pyからCGIフレームワークに共通の変数と処理をclass tkCGIApplicatoinに分離して、tkCGIApplication.pyライブラリファイルにしてください。API.pyでは、syspathでライブラリへのパスd:/programs/cgi/libを追加して下さい。print\_customへの置き換えでは、まず、tkCGIApplicationクラスのクラス変数としてオリジナルのprint関数をprint\_originalに退避し、redirect()メンバ関数でprint関数を置き換えてください。デコレータもtkCGIApplicationクラスのメンバとし、API.pyではapp変数にインスタンスを作ります

#!/usr/bin/env python3 import os import time import builtins import functools from functools import lru\_cache import json import urllib.parse import glob import re import logging import yaml rate\_limit = {} log\_file = 'd:/programs/cgi/API.log' account\_path = 'd:/programs/cgi/cgi-bin/accounts.yaml' # 元の print 関数を退避 print\_original = builtins.print # 新しい print 関数を定義 def print\_custom(\*args, \*\*kwargs): # `<pre>` タグを追加して出力 # print\_original("<pre>") print\_original(\*args, \*\*kwargs) # print\_original("</pre>") # 組み込みの print を新しいものに置き換え builtins.print = print\_custom # ルートマッピング辞書 routes = {} # デコレータ関数 def route(action): def decorator(func): # routes辞書にアクションと関数を関連付け routes[action] = func @functools.wraps(func) def wrapper(\*args, \*\*kwargs): return func(\*args, \*\*kwargs) return wrapper return decorator def after\_request(func): def wrapper(\*args, \*\*kwargs): return func(account, \*args, \*\*kwargs) return wrapper def before\_request(func): def wrapper(\*args, \*\*kwargs): logging.info(f"Validating account for function: {func.\_\_name\_\_}") # QUERY\_STRINGから必要な情報を取得 query = kwargs.get('query', {}) user = query.get('user', '') password = query.get('pw', '') clientIP = os.getenv('REMOTE\_ADDR', '') # アカウントリストの取得 account\_list = get\_account\_list() if not account\_list: logging.error("Failed to load account list.") print(json.dumps({'result': False, 'message': "Failed to load account list."}, indent=4)) return # アカウント取得 account = get\_account(user, account\_list) if not account: logging.warning(f"Unauthorized user access attempt: user={user}") print(json.dumps({'result': False, 'message': "Invalid user or unauthorized access."}, indent=4)) return # パスワードの確認 if not is\_valid\_password(account, password): logging.warning(f"Invalid password attempt for user: {user}") print(json.dumps({'result': False, 'message': "Invalid password."}, indent=4)) return # IPアドレスの確認 if not is\_valid\_ip(account, clientIP): logging.warning(f"Unauthorized IP access attempt for user: {user}, IP: {clientIP}") print(json.dumps({'result': False, 'message': "Unauthorized IP address."}, indent=4)) return # アカウントが有効であれば関数を実行 return func(account, \*args, \*\*kwargs) return wrapper @lru\_cache(maxsize=128) def cached\_function(param): # キャッシュ可能な重い処理 return param \* 2 @app.route('/cachedRoute/<int:input\_value>') def cached\_route(input\_value): result = cached\_function(input\_value) return jsonify({'result': result}) #アクセスのレート制限 def check\_rate\_limit(limit\_per\_second): def decorator(func): def wrapper(\*args, \*\*kwargs): user\_ip = request.remote\_addr current\_time = time.time() if user\_ip in rate\_limit: last\_access\_time = rate\_limit[user\_ip] if current\_time - last\_access\_time < 1 / limit\_per\_second: return jsonify({'result': False, 'message': 'Rate limit exceeded'}), 429 rate\_limit[user\_ip] = current\_time return func(\*args, \*\*kwargs) return wrapper return decorator def log\_activity(func): def wrapper(\*args, \*\*kwargs): logging.info(f"Accessing {func.\_\_name\_\_} with args: {args}, kwargs: {kwargs}") try: result = func(\*args, \*\*kwargs) logging.info(f"Execution successful: {func.\_\_name\_\_}") return result except Exception as e: logging.error(f"Error in {func.\_\_name\_\_}: {e}") raise return wrapper if os.access(os.path.dirname(log\_file), os.W\_OK): logging.basicConfig( filename=log\_file, level=logging.INFO, format='%(asctime)s %(message)s' ) else: print(json.dumps({'result': False, 'message': 'Logging directory is not writable.'}, indent=4)) raise SystemExit("Critical error: Unable to initialize logging.") # ヘルパー関数 def parse\_query\_string(query\_string): query = {} if query\_string: for param in query\_string.split('&'): if '=' in param: key, value = param.split('=') query[key] = urllib.parse.unquote(value) query[key] = sanitize\_input(query[key]) else: logging.error(f"Malformed parameter in query string: {param}") return {} else: logging.error("Query string is empty or missing.") return query def sanitize\_input(value): return re.sub(r'[^\w\s\-\.]', '', value) # 許可された文字のみを許可 def get\_account\_list(): logging.info(f"get\_account\_list: reading {account\_path}") if os.path.exists(account\_path): with open(account\_path, 'r') as file: try: accounts = yaml.safe\_load(file) logging.info(f"Accounts loaded successfully: {accounts}") return accounts except yaml.YAMLError as e: logging.error(f"Error parsing YAML file: {e}") return None else: logging.error(f"{account\_path} file not found.") return None def get\_account(user, account\_list): return next((account for account in account\_list if account['user'] == user), None) def is\_valid\_path(account, rootDir): abs\_path = os.path.abspath(rootDir) return ( '../' not in abs\_path and './' not in abs\_path and any(abs\_path.startswith(os.path.abspath(allowed\_dir)) for allowed\_dir in account['dirs']) ) def is\_valid\_filemask(account, filemask): abs\_filemask = os.path.abspath(filemask) return not any( re.search(pattern, abs\_filemask) for pattern in account['rejected\_file\_masks'] ) and any( re.search(pattern, abs\_filemask) for pattern in account['file\_masks'] ) def is\_valid\_ip(account, ip): return any(re.match(pattern, ip) for pattern in account['IPaddress']) def is\_valid\_password(account, password): return account['password'] == password @route('getFile') @before\_request def get\_files(account, query): rootDir = query.get('rootDir', '') fileMask = query.get('fileMask', '') # パスとマスクのバリデーション if not is\_valid\_path(account, rootDir): logging.warning(f"Invalid rootDir access attempt: {rootDir}") print(json.dumps({'result': False, 'message': "Invalid rootDir."}, indent=4)) return try: # ファイル検索 pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask)) valid\_files = [path for path in pathList if is\_valid\_filemask(account, path)] relativePaths = [os.path.relpath(path, rootDir) for path in valid\_files] response = {'result': True, 'pathList': relativePaths} logging.info(f"Success: rootDir={rootDir}, fileMask={fileMask}, user={account['user']}, paths={relativePaths}") print(json.dumps(response, indent=4)) except Exception as e: logging.error(f"Error during file search: {e}") print(json.dumps({'result': False, 'message': f"Error during file search: {e}"}, indent=4)) # ルーティング処理を行う関数 def process\_action(query): action = query.get('action') if action in routes: # ルートに登録されている関数を実行 return routes[action](query) else: # 未定義のアクションの場合 response = { 'result': False, 'message': f"Action '{action}' is not supported.", } print(json.dumps(response, indent=4)) return response def main(): print("Content-Type: application/json") print() # QUERY\_STRINGが存在するか確認 query\_string = os.getenv('QUERY\_STRING', '') if not query\_string: logging.error("QUERY\_STRING is missing.") print(json.dumps({'result': False, 'message': "QUERY\_STRING is missing."}, indent=4)) return # クエリパラメータの解析 query = parse\_query\_string(query\_string) if not query: print(json.dumps({'result': False, 'message': "Invalid query parameters."})) return # debugモードの確認 debug\_mode = query.get('debug', '0') == '1' # 各パラメータを取得 action = query.get('action', '') rootDir = query.get('rootDir', '') fileMask = query.get('fileMask', '') user = query.get('user', '') password = query.get('pw', '') clientIP = os.getenv('REMOTE\_ADDR', '') # アカウントリストの取得 account\_list = get\_account\_list() if not account\_list: logging.error("Failed to load account list.") print(json.dumps({'result': False, 'message': "Failed to load account list."}, indent=4)) return # ユーザーのアカウント情報を取得 account = get\_account(user, account\_list) if not account: logging.warning(f"Unauthorized user access attempt: user={user}") print(json.dumps({'result': False, 'message': "Invalid user or unauthorized access."}, indent=4)) return # パスワード確認 if not is\_valid\_password(account, password): logging.warning(f"Invalid password attempt for user: {user}") print(json.dumps({'result': False, 'message': "Invalid password."}, indent=4)) return # IPアドレス確認 if not is\_valid\_ip(account, clientIP): logging.warning(f"Unauthorized IP access attempt for user: {user}, IP: {clientIP}") print(json.dumps({'result': False, 'message': "Unauthorized IP address."}, indent=4)) return # ルーティング処理を呼び出し response = process\_action(query) print(json.dumps(response), indent=4) if \_\_name\_\_ == '\_\_main\_\_': main()

**Copilot の発言**

以下に、tkCGIApplication クラスをライブラリファイル tkCGIApplication.py に分離し、共通の処理と変数をクラスにまとめました。その後、API.py でライブラリを利用するように調整しました。

**ライブラリファイル: tkCGIApplication.py**

python

コピー

import os

import builtins

import functools

import logging

import json

import urllib.parse

import re

import yaml

class tkCGIApplication:

# クラス変数としてオリジナルの print 関数を退避

print\_original = builtins.print

def \_\_init\_\_(self, log\_file, account\_path):

self.log\_file = log\_file

self.account\_path = account\_path

self.routes = {}

# ログの設定

self.setup\_logging()

def setup\_logging(self):

if os.access(os.path.dirname(self.log\_file), os.W\_OK):

logging.basicConfig(

filename=self.log\_file,

level=logging.INFO,

format='%(asctime)s %(message)s'

)

else:

self.print({'result': False, 'message': 'Logging directory is not writable.'}, indent=4)

raise SystemExit("Critical error: Unable to initialize logging.")

def redirect\_print(self):

# クラスメソッドで print 関数を置き換え

builtins.print = self.print\_custom

def print\_custom(self, \*args, \*\*kwargs):

# `<pre>` タグで囲んで表示

self.print\_original("<pre>")

self.print\_original(\*args, \*\*kwargs)

self.print\_original("</pre>")

def route(self, action):

def decorator(func):

self.routes[action] = func

@functools.wraps(func)

def wrapper(\*args, \*\*kwargs):

return func(\*args, \*\*kwargs)

return wrapper

return decorator

def parse\_query\_string(self, query\_string):

query = {}

if query\_string:

for param in query\_string.split('&'):

if '=' in param:

key, value = param.split('=')

query[key] = urllib.parse.unquote(value)

query[key] = self.sanitize\_input(query[key])

else:

logging.error(f"Malformed parameter in query string: {param}")

return {}

else:

logging.error("Query string is empty or missing.")

return query

def sanitize\_input(self, value):

return re.sub(r'[^\w\s\-\.]', '', value)

def get\_account\_list(self):

logging.info(f"get\_account\_list: reading {self.account\_path}")

if os.path.exists(self.account\_path):

with open(self.account\_path, 'r') as file:

try:

accounts = yaml.safe\_load(file)

logging.info(f"Accounts loaded successfully: {accounts}")

return accounts

except yaml.YAMLError as e:

logging.error(f"Error parsing YAML file: {e}")

return None

else:

logging.error(f"{self.account\_path} file not found.")

return None

def get\_account(self, user, account\_list):

return next((account for account in account\_list if account['user'] == user), None)

def process\_action(self, query):

action = query.get('action')

if action in self.routes:

return self.routes[action](query)

else:

response = {

'result': False,

'message': f"Action '{action}' is not supported.",

}

print(json.dumps(response, indent=4))

return response

**実装例: API.py**

python

コピー

#!/usr/bin/env python3

import os

import sys

import json

import glob

from tkCGIApplication import tkCGIApplication

# ライブラリへのパスを追加

sys.path.append('d:/programs/cgi/lib')

# tkCGIApplicationインスタンスを作成

app = tkCGIApplication(

log\_file='d:/programs/cgi/API.log',

account\_path='d:/programs/cgi/cgi-bin/accounts.yaml'

)

# print をカスタマイズ

app.redirect\_print()

@app.route('getFile')

def get\_files(query):

account\_list = app.get\_account\_list()

user = query.get('user', '')

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

# ユーザー認証

account = app.get\_account(user, account\_list)

if not account:

logging.warning(f"Unauthorized user access attempt: user={user}")

print(json.dumps({'result': False, 'message': "Invalid user or unauthorized access."}, indent=4))

return

# ディレクトリ検証

if not rootDir or not fileMask:

logging.warning(f"Invalid rootDir or fileMask.")

print(json.dumps({'result': False, 'message': "Invalid rootDir or fileMask."}, indent=4))

return

# ファイル検索

try:

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

response = {'result': True, 'pathList': pathList}

print(json.dumps(response, indent=4))

except Exception as e:

logging.error(f"Error during file search: {e}")

print(json.dumps({'result': False, 'message': f"Error during file search: {e}"}, indent=4))

def main():

print("Content-Type: application/json")

print()

query\_string = os.getenv('QUERY\_STRING', '')

query = app.parse\_query\_string(query\_string)

if query:

app.process\_action(query)

else:

print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

if \_\_name\_\_ == "\_\_main\_\_":

main()

**ポイント**

1. **クラスの分離**:
   * tkCGIApplication に共通処理を分離し、API部分をシンプルに。
2. **柔軟性の確保**:
   * デコレータやヘルパー関数をメンバ関数として提供することで、拡張が容易に。
3. **再利用性**:
   * 複数のAPIが同じライブラリを使用できるよう設計。

この構造で拡張性の高いフレームワークが作れます！さらにカスタマイズしたい点があれば教えてくださいね 🦊✨

**あなたの発言**

以下のget\_filesですが、デコレータbefore\_requestで認証処理をするので、このデータこれーたを使ったら、関数内部が簡単になりませんか

#!/usr/bin/env python3 import os import sys import json import glob sys.path.append('d:/programs/cgi/lib') from tkCGIApplication import tkCGIApplication # tkCGIApplicationインスタンスを作成 app = tkCGIApplication( log\_file='d:/programs/cgi/log/API.log', account\_path='d:/programs/cgi/data/accounts.yaml' ) # print をカスタマイズ app.redirect\_print() @app.route('getFile') def get\_files(query): account\_list = app.get\_account\_list() user = query.get('user', '') rootDir = query.get('rootDir', '') fileMask = query.get('fileMask', '') # ユーザー認証 account = app.get\_account(user, account\_list) if not account: logging.warning(f"Unauthorized user access attempt: user={user}") print(json.dumps({'result': False, 'message': "Invalid user or unauthorized access."}, indent=4)) return # ディレクトリ検証 if not rootDir or not fileMask: logging.warning(f"Invalid rootDir or fileMask.") print(json.dumps({'result': False, 'message': "Invalid rootDir or fileMask."}, indent=4)) return # ファイル検索 try: pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask)) response = {'result': True, 'pathList': pathList} print(json.dumps(response, indent=4)) except Exception as e: logging.error(f"Error during file search: {e}") print(json.dumps({'result': False, 'message': f"Error during file search: {e}"}, indent=4)) def main(): print("Content-Type: application/json") print() query\_string = os.getenv('QUERY\_STRING', '') query = app.parse\_query\_string(query\_string) if query: app.process\_action(query) else: print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4)) if \_\_name\_\_ == "\_\_main\_\_": main()

**Copilot の発言**

はい、その通りです！before\_requestデコレータを使用することで、認証処理をデコレータに移動し、get\_files関数内部をシンプルにできます。以下のコード例では、認証処理をbefore\_requestデコレータに移動し、get\_files関数内の重複する処理を排除しています。

**修正版コード**

**API.py**

python

コピー

#!/usr/bin/env python3

import os

import sys

import json

import glob

# ライブラリのパスを追加

sys.path.append('d:/programs/cgi/lib')

from tkCGIApplication import tkCGIApplication

# tkCGIApplicationのインスタンスを作成

app = tkCGIApplication(

log\_file='d:/programs/cgi/log/API.log',

account\_path='d:/programs/cgi/data/accounts.yaml'

)

# print をカスタマイズ

app.redirect\_print()

# デコレータで認証処理を移動

@app.route('getFile')

@app.before\_request

def get\_files(account, query):

# 必要なクエリパラメータを取得

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

# ルートディレクトリとファイルマスクの検証

if not rootDir or not fileMask:

logging.warning(f"Invalid rootDir or fileMask.")

print(json.dumps({'result': False, 'message': "Invalid rootDir or fileMask."}, indent=4))

return

# ファイル検索処理

try:

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

response = {'result': True, 'pathList': pathList}

logging.info(f"File search successful. user={account['user']}, rootDir={rootDir}, fileMask={fileMask}")

print(json.dumps(response, indent=4))

except Exception as e:

logging.error(f"Error during file search: {e}")

print(json.dumps({'result': False, 'message': f"Error during file search: {e}"}, indent=4))

def main():

print("Content-Type: application/json")

print()

query\_string = os.getenv('QUERY\_STRING', '')

query = app.parse\_query\_string(query\_string)

if query:

app.process\_action(query)

else:

print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

if \_\_name\_\_ == "\_\_main\_\_":

main()

**重要な変更点**

1. before\_request**デコレータの活用**:
   * アカウント認証、パスワード検証、IPチェックといった前処理をbefore\_requestに移動。
   * 関数内部では、認証済みのaccountデータを直接使用するだけで済むため、冗長なコードを削除可能。

**before\_request内部処理（例）**

* + 認証で問題があれば関数呼び出しを阻止:

python

コピー

def before\_request(func):

def wrapper(\*args, \*\*kwargs):

query = kwargs.get('query', {})

user = query.get('user', '')

password = query.get('pw', '')

clientIP = os.getenv('REMOTE\_ADDR', '')

account\_list = self.get\_account\_list()

account = self.get\_account(user, account\_list)

# アカウント検証

if not account or not self.is\_valid\_password(account, password) or not self.is\_valid\_ip(account, clientIP):

return self.print\_error("Unauthorized access.", 403)

return func(account, query, \*args, \*\*kwargs) # 認証済みデータで関数を呼び出し

return wrapper

1. get\_files**関数の簡略化**:
   * accountとqueryはすでに検証済みで渡されるため、ビジネスロジック（ファイル検索処理）のみに集中可能。
   * 認証に関する重複コードや例外処理を削除。
2. **柔軟性と可読性の向上**:
   * 新しいエンドポイントを追加する際も認証処理を毎回記述する必要がなく、開発の効率化に寄与。

**メリット**

1. 認証のロジックをデコレータにカプセル化することで、認証処理の一貫性を保証。
2. 個別関数の役割が明確になり、メンテナンスが容易。
3. 新しいルート（エンドポイント）を追加しても認証処理を簡単に適用可能。

もし他にも改善したいポイントや要件があれば、遠慮なく教えてくださいね。✨🐾

**あなたの発言**

やっぱり、認証確認のデコレータは、validate\_accountにしたほうが可読性が上がりますね。before\_requestはカスタマイズ用にとっておきます。また、 print("Content-Type: application/json") print() はapp.init\_html(content\_type)に移します。

**Copilot の発言**

確かに、validate\_accountという名前のデコレータにすることで、認証確認の意図がより明確になりますね！before\_requestをカスタマイズ用に温存しておくのも良い考えです。さらに、app.init\_html(content\_type) を導入することで、HTMLヘッダーの初期化処理が一元化され、コードの再利用性が向上しますね。

以下のように修正案を示します：

**修正版コード例**

**API.py**

python

コピー

#!/usr/bin/env python3

import os

import sys

import json

import glob

# ライブラリへのパスを追加

sys.path.append('d:/programs/cgi/lib')

from tkCGIApplication import tkCGIApplication

# tkCGIApplicationのインスタンスを作成

app = tkCGIApplication(

log\_file='d:/programs/cgi/log/API.log',

account\_path='d:/programs/cgi/data/accounts.yaml'

)

# print をカスタマイズ

app.redirect\_print()

# 認証確認用デコレータを利用

@app.route('getFile')

@app.validate\_account

def get\_files(account, query):

# クエリパラメータの取得

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

# パスとマスクのバリデーション

if not rootDir or not fileMask:

logging.warning(f"Invalid rootDir or fileMask.")

print(json.dumps({'result': False, 'message': "Invalid rootDir or fileMask."}, indent=4))

return

# ファイル検索処理

try:

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

response = {'result': True, 'pathList': pathList}

logging.info(f"File search successful. user={account['user']}, rootDir={rootDir}, fileMask={fileMask}")

print(json.dumps(response, indent=4))

except Exception as e:

logging.error(f"Error during file search: {e}")

print(json.dumps({'result': False, 'message': f"Error during file search: {e}"}, indent=4))

# main関数

def main():

app.init\_html("application/json") # HTML初期化

query\_string = os.getenv('QUERY\_STRING', '')

query = app.parse\_query\_string(query\_string)

if query:

app.process\_action(query)

else:

print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

if \_\_name\_\_ == "\_\_main\_\_":

main()

**tkCGIApplication.py**

app.init\_html(content\_type)とvalidate\_accountデコレータをtkCGIApplicationクラスに追加します。

python

コピー

import os

import builtins

import functools

import logging

import json

import urllib.parse

import re

import yaml

class tkCGIApplication:

print\_original = builtins.print

def \_\_init\_\_(self, log\_file, account\_path):

self.log\_file = log\_file

self.account\_path = account\_path

self.routes = {}

self.setup\_logging()

def setup\_logging(self):

if os.access(os.path.dirname(self.log\_file), os.W\_OK):

logging.basicConfig(

filename=self.log\_file,

level=logging.INFO,

format='%(asctime)s %(message)s'

)

else:

self.print\_original("Logging directory is not writable.")

raise SystemExit("Critical error: Unable to initialize logging.")

def redirect\_print(self):

builtins.print = self.print\_custom

def print\_custom(self, \*args, \*\*kwargs):

self.print\_original("<pre>")

self.print\_original(\*args, \*\*kwargs)

self.print\_original("</pre>")

def init\_html(self, content\_type="text/html"):

"""HTML初期化処理"""

print(f"Content-Type: {content\_type}")

print()

def route(self, action):

def decorator(func):

self.routes[action] = func

@functools.wraps(func)

def wrapper(\*args, \*\*kwargs):

return func(\*args, \*\*kwargs)

return wrapper

return decorator

def validate\_account(self, func):

"""認証処理デコレータ"""

@functools.wraps(func)

def wrapper(\*args, \*\*kwargs):

query = kwargs.get('query', {})

user = query.get('user', '')

password = query.get('pw', '')

clientIP = os.getenv('REMOTE\_ADDR', '')

# アカウントリストの取得

account\_list = self.get\_account\_list()

account = self.get\_account(user, account\_list)

# アカウント検証

if not account or not self.is\_valid\_password(account, password) or not self.is\_valid\_ip(account, clientIP):

logging.warning(f"Authentication failed for user={user}, IP={clientIP}")

print(json.dumps({'result': False, 'message': "Unauthorized access."}, indent=4))

return

# 認証成功時に関数実行

return func(account, query, \*args, \*\*kwargs)

return wrapper

def parse\_query\_string(self, query\_string):

query = {}

if query\_string:

for param in query\_string.split('&'):

if '=' in param:

key, value = param.split('=')

query[key] = urllib.parse.unquote(value)

query[key] = self.sanitize\_input(query[key])

else:

logging.error(f"Malformed parameter in query string: {param}")

return {}

else:

logging.error("Query string is empty or missing.")

return query

def sanitize\_input(self, value):

return re.sub(r'[^\w\s\-\.]', '', value)

def get\_account\_list(self):

logging.info(f"get\_account\_list: reading {self.account\_path}")

if os.path.exists(self.account\_path):

with open(self.account\_path, 'r') as file:

try:

accounts = yaml.safe\_load(file)

logging.info(f"Accounts loaded successfully: {accounts}")

return accounts

except yaml.YAMLError as e:

logging.error(f"Error parsing YAML file: {e}")

return None

else:

logging.error(f"{self.account\_path} file not found.")

return None

def get\_account(self, user, account\_list):

return next((account for account in account\_list if account['user'] == user), None)

def is\_valid\_password(self, account, password):

return account['password'] == password

def is\_valid\_ip(self, account, ip):

return any(re.match(pattern, ip) for pattern in account['IPaddress'])

def process\_action(self, query):

action = query.get('action')

if action in self.routes:

return self.routes[action](query)

else:

response = {

'result': False,

'message': f"Action '{action}' is not supported.",

}

print(json.dumps(response, indent=4))

return response

**変更内容とメリット**

1. validate\_account**デコレータの明確化**:
   * 認証専用のデコレータを作成し、可読性を向上。
   * 各アクションで一貫性のある認証処理を適用。
2. app.init\_html**関数の導入**:
   * ヘッダー初期化を関数化し、再利用性を向上。
3. get\_files**関数の簡素化**:
   * 認証とエラーハンドリングがデコレータに移動したため、ファイル検索のロジックに集中。

これでコードはさらに拡張しやすく、整然とした構造になりました！他にも改善点や追加の要望があれば教えてくださいね 🦊✨

**あなたの発言**

init\_htmlですが、一般的なHTMLページを作成する際の初期化処理を加えていただきますか。target引数を使って、target="json"の場合は現在の出力、""であればHTMLヘッダー、<HTML><head><charcode><title></head><body>を出力してくださいcharcode='utf8'とtitle="title"はデフォ類と引数で与えます。また、end\_html関数を作り、target=""の場合は</body></html>を出力してください。target変数はメンバ変数としいて記録してください。各HTMLタグは、読みやすく改行してください

**Copilot の発言**

以下は、指定された要件を満たすように init\_html 関数を拡張し、end\_html 関数を追加した実装例です。このコードでは、target 引数を用いて JSON または HTML ページの初期化処理を柔軟に切り替えることができ、読みやすいフォーマットで出力されます。

**修正された tkCGIApplication**

python

コピー

import os

import builtins

import functools

import logging

import json

import urllib.parse

import re

import yaml

class tkCGIApplication:

print\_original = builtins.print

def \_\_init\_\_(self, log\_file, account\_path):

self.log\_file = log\_file

self.account\_path = account\_path

self.routes = {}

self.target = "json" # 初期値を "json" に設定

self.setup\_logging()

def setup\_logging(self):

if os.access(os.path.dirname(self.log\_file), os.W\_OK):

logging.basicConfig(

filename=self.log\_file,

level=logging.INFO,

format='%(asctime)s %(message)s'

)

else:

self.print\_original("Logging directory is not writable.")

raise SystemExit("Critical error: Unable to initialize logging.")

def redirect\_print(self):

builtins.print = self.print\_custom

def print\_custom(self, \*args, \*\*kwargs):

self.print\_original("<pre>")

self.print\_original(\*args, \*\*kwargs)

self.print\_original("</pre>")

def init\_html(self, content\_type="application/json", target="json", charcode="utf8", title="title"):

"""HTMLまたはJSON出力用の初期化処理"""

self.target = target

if self.target == "json":

print(f"Content-Type: {content\_type}")

print()

else:

# HTMLヘッダーを出力

print(f"Content-Type: text/html; charset={charcode}")

print()

print("<!DOCTYPE html>")

print("<html>")

print("<head>")

print(f" <meta charset=\"{charcode}\">")

print(f" <title>{title}</title>")

print("</head>")

print("<body>")

def end\_html(self):

"""HTMLの終了タグを出力"""

if self.target == "":

print("</body>")

print("</html>")

def route(self, action):

def decorator(func):

self.routes[action] = func

@functools.wraps(func)

def wrapper(\*args, \*\*kwargs):

return func(\*args, \*\*kwargs)

return wrapper

return decorator

def validate\_account(self, func):

"""認証処理デコレータ"""

@functools.wraps(func)

def wrapper(\*args, \*\*kwargs):

query = kwargs.get('query', {})

user = query.get('user', '')

password = query.get('pw', '')

clientIP = os.getenv('REMOTE\_ADDR', '')

account\_list = self.get\_account\_list()

account = self.get\_account(user, account\_list)

if not account or not self.is\_valid\_password(account, password) or not self.is\_valid\_ip(account, clientIP):

logging.warning(f"Authentication failed for user={user}, IP={clientIP}")

print(json.dumps({'result': False, 'message': "Unauthorized access."}, indent=4))

return

return func(account, query, \*args, \*\*kwargs)

return wrapper

def parse\_query\_string(self, query\_string):

query = {}

if query\_string:

for param in query\_string.split('&'):

if '=' in param:

key, value = param.split('=')

query[key] = urllib.parse.unquote(value)

query[key] = self.sanitize\_input(query[key])

else:

logging.error(f"Malformed parameter in query string: {param}")

return {}

else:

logging.error("Query string is empty or missing.")

return query

def sanitize\_input(self, value):

return re.sub(r'[^\w\s\-\.]', '', value)

def get\_account\_list(self):

logging.info(f"get\_account\_list: reading {self.account\_path}")

if os.path.exists(self.account\_path):

with open(self.account\_path, 'r') as file:

try:

accounts = yaml.safe\_load(file)

logging.info(f"Accounts loaded successfully: {accounts}")

return accounts

except yaml.YAMLError as e:

logging.error(f"Error parsing YAML file: {e}")

return None

else:

logging.error(f"{self.account\_path} file not found.")

return None

def get\_account(self, user, account\_list):

return next((account for account in account\_list if account['user'] == user), None)

def is\_valid\_password(self, account, password):

return account['password'] == password

def is\_valid\_ip(self, account, ip):

return any(re.match(pattern, ip) for pattern in account['IPaddress'])

def process\_action(self, query):

action = query.get('action')

if action in self.routes:

return self.routes[action](query)

else:

response = {

'result': False,

'message': f"Action '{action}' is not supported.",

}

print(json.dumps(response, indent=4))

return response

**API.py 修正版**

python

コピー

#!/usr/bin/env python3

import os

import sys

import glob

import json

# ライブラリのパスを追加

sys.path.append('d:/programs/cgi/lib')

from tkCGIApplication import tkCGIApplication

# tkCGIApplicationのインスタンスを作成

app = tkCGIApplication(

log\_file='d:/programs/cgi/log/API.log',

account\_path='d:/programs/cgi/data/accounts.yaml'

)

# print をカスタマイズ

app.redirect\_print()

# 認証確認用デコレータを利用

@app.route('getFile')

@app.validate\_account

def get\_files(account, query):

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

if not rootDir or not fileMask:

logging.warning("Invalid rootDir or fileMask.")

print(json.dumps({'result': False, 'message': "Invalid rootDir or fileMask."}, indent=4))

return

try:

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

response = {'result': True, 'pathList': pathList}

print(json.dumps(response, indent=4))

except Exception as e:

logging.error(f"Error during file search: {e}")

print(json.dumps({'result': False, 'message': f"Error during file search: {e}"}, indent=4))

def main():

app.init\_html(content\_type="application/json", target="json")

query\_string = os.getenv('QUERY\_STRING', '')

query = app.parse\_query\_string(query\_string)

if query:

app.process\_action(query)

else:

print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

if \_\_name\_\_ == "\_\_main\_\_":

main()

**追加の変更点**

1. init\_html**処理を拡張**:
   * JSON または HTML 出力形式に応じて初期化を切り替え。
   * タグを改行して整形出力。
2. end\_html**を追加**:
   * target="" の場合、適切に HTML の終了タグ (</body></html>) を出力。
3. **メンバ変数**target:
   * 出力ターゲット（"json" または ""）を記録し、各出力処理で参照。

これでさらに柔軟かつ整理された初期化処理が実現できます！必要に応じて微調整もお手伝いしますね 🦊✨

**あなたの発言**

charsetですが、 <meta http-equiv="Content-Type" content="text/html; charset=utf-8"> と <meta charset="utf-8"> のどちらがいいですか

**Copilot の発言**

良い質問ですね！どちらを使うかは目的による部分もありますが、現在では <meta charset="utf-8"> を使うのが主流でおすすめです。

**理由：**

1. **シンプルさと可読性**:
   * <meta charset="utf-8"> はHTML5で導入され、より簡潔で直感的です。これに対し、<meta http-equiv="Content-Type" content="text/html; charset=utf-8"> はHTML4や古いバージョンで用いられていた形式です。
2. **互換性**:
   * 現在のブラウザはどちらの形式も対応していますが、HTML5を使用する場合は<meta charset="utf-8">が推奨されています。
3. **仕様準拠**:
   * HTML5では、文字エンコーディングを指定する際に <meta charset="utf-8"> を使うように明示されています。
4. **パフォーマンス**:
   * 簡潔な形式で記述することにより、HTMLの読み込みと解析がわずかに早くなる可能性があります（とはいえ、大きな差ではありません）。

**使い分けが必要な場合**

ただし、もし何らかの理由でHTML4の互換性や特殊なレガシーシステム対応が必要な場合は、<meta http-equiv="Content-Type" content="text/html; charset=utf-8"> を使用することがあります。

**結論**: 基本的には <meta charset="utf-8"> を使うのが良いでしょう！シンプルで現在の仕様にも準拠しています。何か特定の目的やシステム要件がある場合は教えてくださいね！ 😊✨

**あなたの発言**

Flaskと互換性を持たせるなら、process\_actionはrunの方がいいですか？

**Copilot の発言**

はい、Flaskと互換性を持たせる場合、process\_action を run に変更する方がわかりやすく、Flaskの設計思想に沿った命名となります。

Flaskでは、app.run() がアプリケーションのエントリーポイントとして使われます。この命名は、「アプリケーションを起動する」という直感的な意味を持つため、同様の役割を果たすメソッドを run にすることで、将来的にFlaskと統一した使い方ができるようになります。

**修正のメリット**

1. **直感的で一貫性のある命名**:
   * Flaskユーザーにとって馴染み深い命名になるため、フレームワークを移行する際や、新しい開発者がコードを読む際に理解しやすくなります。
2. **役割が明確**:
   * process\_action という名前は処理を行う意味が幅広く、具体性が薄い場合があります。一方で、run は「全体の処理を開始する」ことを明確に表現します。

**修正例**

以下のように process\_action を run に変更できます：

**tkCGIApplication 側の変更**

python

コピー

def run(self, query):

action = query.get('action')

if action in self.routes:

return self.routes[action](query)

else:

response = {

'result': False,

'message': f"Action '{action}' is not supported.",

}

print(json.dumps(response, indent=4))

return response

**API側の変更**

python

コピー

def main():

app.init\_html(content\_type="application/json", target="json")

query\_string = os.getenv('QUERY\_STRING', '')

query = app.parse\_query\_string(query\_string)

if query:

app.run(query) # `process\_action` を `run` に変更

else:

print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

**名前を変更する際の注意点**

* run メソッドはアプリケーション全体の実行エントリーポイントであるとわかるよう、しっかりコメントで意図を示しておくとより親切です。
* 既存の process\_action を利用する箇所が複数ある場合、すべて適切に変更することを忘れないようにしましょう。

この変更によって、Flask風の設計で互換性を強化できるだけでなく、コードの可読性や一貫性も向上します！ 😊✨

**あなたの発言**

今のAPI.pyのメインコードは以下の通りです。他にFlaskとの互換性を挙げる部分はありますか #!/usr/bin/env python3 import os import sys import json import glob sys.path.append('d:/programs/cgi/lib') from tkCGIApplication import tkCGIApplication # tkCGIApplicationインスタンスを作成 app = tkCGIApplication( log\_file='d:/programs/cgi/log/API.log', account\_path='d:/programs/cgi/data/accounts.yaml' ) # print をカスタマイズ app.redirect\_print() @app.route('getFile') @validate\_account def get\_files(query): pass def main(): app.init\_html(target = "json") query\_string = os.getenv('QUERY\_STRING', '') query = app.parse\_query\_string(query\_string) if query: app.run(query) else: print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4)) if \_\_name\_\_ == "\_\_main\_\_": main()

**Copilot の発言**

あなたのコードはすでにとても整理されていて、Flaskとの互換性を高めた素晴らしい設計になっています！ここからさらにFlaskの設計思想や機能に近づける方法をいくつか提案します。これにより、Flaskに慣れた開発者が簡単にこのコードに適応できるようになります。

**Flaskとの互換性をさらに高めるための提案**

**1. エラーハンドリング**

Flaskではカスタムエラーハンドラーを追加できます。これと同様に、tkCGIApplicationにエラー処理の仕組みを追加すると、コードが統一された振る舞いを実現します。

**提案：**

tkCGIApplication にエラー処理用のハンドラー機能を実装します。

python

コピー

def register\_error\_handler(self, code, handler):

self.error\_handlers[code] = handler

def handle\_error(self, code):

handler = self.error\_handlers.get(code)

if handler:

return handler()

else:

# デフォルトエラー応答

print(json.dumps({'result': False, 'message': f"Error {code} occurred."}, indent=4))

**使い方：**

python

コピー

@app.register\_error\_handler(404, lambda: print(json.dumps({'result': False, 'message': "Not Found."}, indent=4)))

**2. request オブジェクトの導入**

Flaskではflask.requestを使ってリクエストデータ（クエリパラメータ、ヘッダー、JSONボディなど）を一括管理します。同様のオブジェクトを実装すると、クエリ処理がより直感的になります。

**提案：**

query のデータをラップした request オブジェクトを導入。

python

コピー

class Request:

def \_\_init\_\_(self, query):

self.args = query

self.remote\_addr = os.getenv('REMOTE\_ADDR', '')

**修正例：**

python

コピー

query = app.parse\_query\_string(query\_string)

request = Request(query)

if request.args:

app.run(request)

else:

print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

**3. ルートグループ**

Flaskのようにブループリント (Blueprint) に近い仕組みを追加することで、ルートを機能ごとに分割できます。

**提案：**

ルートグループを作成し、ルーティングの管理を簡易化。

python

コピー

def add\_blueprint(self, blueprint, url\_prefix=""):

for route, handler in blueprint.routes.items():

self.routes[url\_prefix + route] = handler

**使い方：**

python

コピー

admin\_routes = tkCGIApplication()

admin\_routes.route('/dashboard')(dashboard\_handler)

app.add\_blueprint(admin\_routes, url\_prefix='/admin')

**4. JSONレスポンスを簡略化**

Flaskではflask.jsonifyを使って辞書を簡単にJSON化してレスポンスします。これを模倣する関数を追加します。

**提案：**

JSONレスポンス用のヘルパーを実装。

python

コピー

def jsonify(self, data):

print(json.dumps(data, indent=4))

**修正例：**

python

コピー

@app.route('getFile')

@validate\_account

def get\_files(account, query):

response = {'result': True, 'message': 'File retrieved successfully.'}

app.jsonify(response)

**5. app.run() の構造**

現在の main() のエントリーポイントを Flask の app.run() に近づけます。

**修正例：**

python

コピー

def main():

app.init\_html(target="json")

# 実行を app.run に委譲

app.run()

if \_\_name\_\_ == "\_\_main\_\_":

main()

tkCGIApplication.run() 内でリクエストデータを解析し、直接ルーティングを処理する設計にします。

**まとめ**

上記の提案により、Flaskとの互換性がさらに高まり、Flaskを利用していた開発者がスムーズに移行できるフレームワークとなります。

特に以下の点を強調します：

* **エラーハンドリング**: 開発者が特定のエラーを処理しやすくする。
* request**オブジェクト**: 一貫したリクエスト管理。
* jsonify**ヘルパー**: 出力を簡略化。
* run: メイン処理を一元化。

これらを適用することで、よりわかりやすく、再利用性の高い設計が可能です。さらに特定の要件や改善案があれば教えてくださいね！✨🐾

**あなたの発言**

今のmain()は以下のようになっています。app.run()の呼び出しだけで済むようにtkCGIApplication.runを作り直してください def main(): app.init\_html(target = "json") query\_string = os.getenv('QUERY\_STRING', '') query = app.parse\_query\_string(query\_string) if query: app.run(query) else: print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

**Copilot の発言**

tkCGIApplication.run() を修正して、app.run() の呼び出しだけで済むようにすることで、main() をさらにシンプルにすることができます。以下に改善した run() メソッドを示します。この新しい run() メソッドは、QUERY\_STRING を直接処理し、内部でクエリの解析とエラーハンドリングを行います。

**修正版 tkCGIApplication.run**

tkCGIApplication クラス内で run() メソッドを以下のように修正します：

python

コピー

def run(self):

"""アプリケーションのエントリーポイント"""

self.init\_html(target="json")

# QUERY\_STRING を取得

query\_string = os.getenv('QUERY\_STRING', '')

# クエリパラメータを解析

query = self.parse\_query\_string(query\_string)

# クエリが存在する場合はルーティング

if query:

action = query.get('action')

if action in self.routes:

return self.routes[action](query)

else:

# アクションが未定義の場合

response = {

'result': False,

'message': f"Action '{action}' is not supported."

}

print(json.dumps(response, indent=4))

else:

# クエリが空の場合

print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

**main() の更新**

新しい run() メソッドを活用すると、main() はより簡潔になります：

**更新後の main()**

python

コピー

def main():

app.run()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**変更点の詳細**

1. run()**の内部に**init\_html**を組み込む**:
   * HTML または JSON 出力の初期化を run() 内で実行するため、main() 側から切り離します。
2. **クエリの解析とエラーハンドリングを統合**:
   * run() 内で QUERY\_STRING を直接処理し、クエリが不正または未定義アクションの場合の応答も一元化。
3. **一貫したフローの実現**:
   * アクションのルーティングを簡潔に管理し、全体の処理フローを統一。

**出力例**

以下のような QUERY\_STRING が渡された場合：

コピー

QUERY\_STRING="action=getFile&user=test&rootDir=/data&fileMask=\*.txt"

run() は自動的に次のフローを実行します：

* action を取得してルーティング。
* 該当する関数（例：get\_files()）を実行。

クエリが不正な場合、以下のようなエラーメッセージが返されます：

json

コピー

{

"result": false,

"message": "Invalid query parameters."

}

これで、main() は呼び出しポイントとして極めてシンプルになり、tkCGIApplication.run メソッドに処理が一元化されました。他にも改善したい箇所や追加機能があればお知らせくださいね！ 🌟🐾

**あなたの発言**

runにredirectやerror\_handlerをいれてみました

def run(self, redirect = true, error\_handler = None, target = ""): if redirect: app.redirect\_print() if error\_handler is None: @self..register\_error\_handler(404, lambda: print(json.dumps({'result': False, 'message': "Not Found."}, indent=4))) self.init\_html(target = target) query\_string = os.getenv('QUERY\_STRING', '') query = self.parse\_query\_string(query\_string) if query: action = query.get('action') if action in self.routes: return self.routes[action](query) else: # アクションが未定義の場合 response = { 'result': False, 'message': f"Action '{action}' is not supported." } print(json.dumps(response, indent=4)) else: print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

**Copilot の発言**

良い方向性ですね！redirectやerror\_handlerをrunに組み込むことで、さらに柔軟で使いやすい設計になっています。ただし、現在のコードにはいくつかの問題や改善できるポイントがあります。

以下に修正版のコードを示しつつ、調整点を解説します。

**修正版 run メソッド**

python

コピー

def run(self, redirect=True, error\_handler=None, target="json"):

"""

アプリケーションのエントリーポイント。

- redirect: print 関数をカスタムにリダイレクトするかどうか。

- error\_handler: カスタムエラーハンドラーを登録。

- target: 出力形式 ("json" または HTML ページ) を指定。

"""

# print をリダイレクト

if redirect:

self.redirect\_print()

# デフォルトエラーハンドラーを登録（必要であれば）

if error\_handler is None:

self.register\_error\_handler(404, lambda: print(json.dumps({'result': False, 'message': "Not Found."}, indent=4)))

# HTML または JSON の初期化

self.init\_html(target=target)

# QUERY\_STRING を取得

query\_string = os.getenv('QUERY\_STRING', '')

# クエリ解析

query = self.parse\_query\_string(query\_string)

# アクションのルーティング

if query:

action = query.get('action')

if action in self.routes:

return self.routes[action](query) # 登録された関数を呼び出し

else:

# 定義されていないアクション

if error\_handler:

error\_handler(404) # カスタムエラーハンドラーを使用

else:

response = {

'result': False,

'message': f"Action '{action}' is not supported."

}

print(json.dumps(response, indent=4))

else:

# クエリが空または不正

print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

**調整ポイントの解説**

**1. redirect フラグの処理**

* redirect=True の場合に redirect\_print を呼び出し。これにより、簡単にリダイレクトの有無を制御できます。

**2. error\_handler の組み込み**

* error\_handler を引数として受け取り、指定がない場合はデフォルトの404ハンドラーを登録します。
* ユーザーが必要に応じてカスタムハンドラーを渡せるよう設計。
* 登録は次のように行います：

python

コピー

app.run(error\_handler=lambda: print(json.dumps({'result': False, 'message': "Custom Not Found."}, indent=4)))

**3. エラーハンドラーのデフォルト動作**

* 指定がない場合、404のエラー応答を出力。
* 既存のアクションが未定義の場合は標準ハンドラーを利用。

**4. HTMLの初期化を簡素化**

* target="json" がデフォルトとなり、HTMLページ生成が必要な場合は run(target="html") のように指定可能。
* 今後の拡張性を考慮し、target の値で柔軟に切り替えられる構造。

**5. アクション未定義時の処理**

* 登録されていないアクションが呼び出された場合のエラーハンドリングは、error\_handler に委譲。
* 実装が存在しない場合も、デフォルトでエラーメッセージを出力。

**main() のさらなる簡素化**

run() を改善したことで、main() はさらにシンプルになります：

python

コピー

def main():

app.run(target="json") # 必要に応じて target を変更

if \_\_name\_\_ == "\_\_main\_\_":

main()

**改善のメリット**

1. **柔軟性向上**:
   * カスタムエラーハンドラーや出力形式の切り替えを簡単に指定可能。
   * JSONまたはHTMLに応じた初期化が簡単。
2. **統一されたインターフェース**:
   * app.run() だけでエントリーポイントを管理。
   * ルートの追加やエラー処理が直感的に行える。
3. **コードの可読性向上**:
   * main() を極限までシンプル化。
   * 一般的な処理（リダイレクト、初期化、エラーハンドリング）を run() に集約。

これで、さらに洗練された構造になり、Flaskとほぼ同様の操作性を持たせることができるはずです！ 他に追加したい機能や調整点があれば教えてくださいね！ 🦊✨

**あなたの発言**

上で作ったappインスタンスを使って、以下の関数を簡単化してください @app.route('getFile') @app.validate\_account def get\_files(query): account\_list = app.get\_account\_list() user = query.get('user', '') rootDir = query.get('rootDir', '') fileMask = query.get('fileMask', '') # ユーザー認証 account = app.get\_account(user, account\_list) if not account: logging.warning(f"Unauthorized user access attempt: user={user}") print(json.dumps({'result': False, 'message': "Invalid user or unauthorized access."}, indent=4)) return # ディレクトリ検証 if not rootDir or not fileMask: logging.warning(f"Invalid rootDir or fileMask.") print(json.dumps({'result': False, 'message': "Invalid rootDir or fileMask."}, indent=4)) return # ファイル検索 try: pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask)) response = {'result': True, 'pathList': pathList} print(json.dumps(response, indent=4)) except Exception as e: logging.error(f"Error during file search: {e}") print(json.dumps({'result': False, 'message': f"Error during file search: {e}"}, indent=4))

**Copilot の発言**

認証処理は既に@validate\_accountデコレータに移動されているため、この関数内でのユーザー認証を削除できます。また、クエリパラメータの取得もシンプル化されます。以下は簡略化されたコードです：

python

コピー

@app.route('getFile')

@app.validate\_account

def get\_files(account, query):

# 必要なクエリパラメータを取得

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

# パスとマスクのバリデーション

if not rootDir or not fileMask:

logging.warning(f"Invalid rootDir or fileMask.")

print(json.dumps({'result': False, 'message': "Invalid rootDir or fileMask."}, indent=4))

return

# ファイル検索処理

try:

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

response = {'result': True, 'pathList': pathList}

logging.info(f"File search successful. user={account['user']}, rootDir={rootDir}, fileMask={fileMask}")

print(json.dumps(response, indent=4))

except Exception as e:

logging.error(f"Error during file search: {e}")

print(json.dumps({'result': False, 'message': f"Error during file search: {e}"}, indent=4))

**簡略化のポイント**

1. **認証処理の削除**:
   * @validate\_accountデコレータでユーザー認証が既に処理されているため、関数内で再度認証を行う必要がありません。
2. **クエリパラメータの効率化**:
   * 必要なクエリパラメータ (rootDir, fileMask) の取得だけに集中。
3. **エラーハンドリングの簡素化**:
   * try/exceptブロックをファイル検索処理に限定することで、エラー箇所が明確に。
4. **ログの改善**:
   * 成功ログにユーザー情報やディレクトリ情報を含め、デバッグを容易に。

この形でコードはよりシンプルになり、役割が明確になります！さらに改良が必要なら教えてくださいね！✨🐾

**あなたの発言**

logging.warningと.info関数もクラスに入れられますね。app.log\_warningとapp.logにするのがいいように思います

**Copilot の発言**

それは素晴らしいアイデアですね！logging.warningやlogging.infoをtkCGIApplicationクラスのメソッドとして統一することで、ロギングの使い方を一貫させられます。これにより、コードの可読性が向上し、クラスの利用者にとって直感的に扱いやすくなります。

**修正例**

以下のようにlog\_warningとlog\_infoのメソッドを追加します：

**tkCGIApplicationクラスの修正**

python

コピー

def log\_warning(self, message):

"""Warning レベルのログを記録"""

logging.warning(message)

def log\_info(self, message):

"""Info レベルのログを記録"""

logging.info(message)

これをクラスに追加するだけで、クラスインスタンスを使って一貫したロギングが可能になります。

**get\_files関数の修正**

修正後、logging.warningやlogging.infoの呼び出しをapp.log\_warningやapp.log\_infoに変更できます。

python

コピー

@app.route('getFile')

@app.validate\_account

def get\_files(account, query):

# クエリパラメータを取得

rootDir = query.get('rootDir', '')

fileMask = query.get('fileMask', '')

# パスとマスクのバリデーション

if not rootDir or not fileMask:

app.log\_warning(f"Invalid rootDir or fileMask.")

print(json.dumps({'result': False, 'message': "Invalid rootDir or fileMask."}, indent=4))

return

# ファイル検索処理

try:

pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask))

response = {'result': True, 'pathList': pathList}

app.log\_info(f"File search successful. user={account['user']}, rootDir={rootDir}, fileMask={fileMask}")

print(json.dumps(response, indent=4))

except Exception as e:

app.log\_warning(f"Error during file search: {e}")

print(json.dumps({'result': False, 'message': f"Error during file search: {e}"}, indent=4))

**メリット**

1. **一貫したロギング**:
   * ロギングをapp.log\_warningやapp.log\_infoで統一することで、メソッドのインターフェースが明確になります。
2. **柔軟な拡張**:
   * 将来的にロギングのフォーマットや出力先を変更したい場合、tkCGIApplicationクラス内のメソッドだけを修正すればすべての呼び出し箇所に影響を与えられます。
3. **コードの簡潔化**:
   * ロギングの処理をtkCGIApplicationに閉じ込めることで、各エンドポイント関数のコードが簡潔になります。

このアプローチで、さらなるコードのクリーンアップと拡張性の向上が期待できます。他に改良したい箇所やロギング機能の拡張案があれば教えてくださいね！ 🦊✨

**あなたの発言**

以下のコードでエラーが出ています。全体を見直してください

import os import re import builtins import functools import logging import json import yaml import html import urllib.parse class tkCGIApplication: # クラス変数としてオリジナルの print 関数を退避 print\_original = builtins.print def \_\_init\_\_(self, log\_file, account\_path): self.print = builtins.print self.html\_initialized = False self.log\_file = log\_file self.account\_path = account\_path self.routes = {} self.error\_handlers = {} # ログの設定 self.setup\_logging() def setup\_logging(self): if not os.path.exists(os.path.dirname(self.log\_file)): os.makedirs(os.path.dirname(self.log\_file)) if os.access(os.path.dirname(self.log\_file), os.W\_OK): logging.basicConfig( filename=self.log\_file, level=logging.INFO, format='%(asctime)s %(message)s' ) else: self.print({'result': False, 'message': 'Logging directory is not writable.'}, indent=4) raise SystemExit("Critical error: Unable to initialize logging.") def print\_custom(self, \*args, \*\*kwargs): if not self.html\_initialized: self.init\_html() # `<pre>` タグで囲んで表示 # self.print\_original("<pre>") self.print\_original(\*args, \*\*kwargs) # self.print\_original("</pre>") def redirect\_print(self, print\_func = None): if print\_func: builtins.print = print\_func else: builtins.print = self.print\_custom def init\_html(self, target="json", charset="utf-8", title="title", html\_ver = "HTML5", force\_init=False): if not force\_init and self.html\_initialized: return self.html\_initialized = True self.target = target if self.target == "json": print(f"Content-Type: application/json") print() else: # HTMLヘッダーを出力 print(f"Content-Type: text/html; charset={charset}") print() print("<!DOCTYPE html>") print("<html>") print("<head>") if html\_ver == "HTML5": print(f" <meta charset=\"{charset}\">") else: print(f" <meta http-equiv=\"Content-Type\" content=\"text/html; charset={charset}\">") print(f" <title>{title}</title>") print("</head>") print("<body>") def end\_html(self): """HTMLの終了タグを出力""" if self.html\_initialized and self.target == "": print("</body>") print("</html>") def log(self, message): """Info レベルのログを記録""" logging.info(message) def log\_warning(self, message): """Warning レベルのログを記録""" logging.warning(message) def log\_error(self, message): """error レベルのログを記録""" logging.error(message) def route(self, action): def decorator(func): self.routes[action] = func @functools.wraps(func) def wrapper(\*args, \*\*kwargs): return func(\*args, \*\*kwargs) return wrapper return decorator def before\_request(self): def wrapper(func): return func return wrapper def after\_request(self): def wrapper(func): return func return wrapper def register\_error\_handler(self, code, handler): self.error\_handlers[code] = handler def handle\_error(self, code): handler = self.error\_handlers.get(code) if handler: return handler() else: # デフォルトエラー応答 print(json.dumps({'result': False, 'message': f"Error {code} occurred."}, indent=4)) logging.error(f"Unhandled error {code}") def validate\_account(self, func): def wrapper(\*args, \*\*kwargs): logging.info(f"Validating account for function: {func.\_\_name\_\_}") # QUERY\_STRINGから必要な情報を取得 query = kwargs.get('query', {}) user = query.get('user', '') password = query.get('pw', '') clientIP = os.getenv('REMOTE\_ADDR', '') # アカウントリストの取得 account\_list = get\_account\_list() if not account\_list: logging.error("Failed to load account list.") print(json.dumps({'result': False, 'message': "Failed to load account list."}, indent=4)) return # アカウント取得 account = get\_account(user, account\_list) if not account: logging.warning(f"Unauthorized user access attempt: user={user}") print(json.dumps({'result': False, 'message': "Invalid user or unauthorized access."}, indent=4)) return # パスワードの確認 if not is\_valid\_password(account, password): logging.warning(f"Invalid password attempt for user: {user}") print(json.dumps({'result': False, 'message': "Invalid password."}, indent=4)) return # IPアドレスの確認 if not is\_valid\_ip(account, clientIP): logging.warning(f"Unauthorized IP access attempt for user: {user}, IP: {clientIP}") print(json.dumps({'result': False, 'message': "Unauthorized IP address."}, indent=4)) return # アカウントが有効であれば関数を実行 return func(account, \*args, \*\*kwargs) return wrapper def parse\_query\_string(self, query\_string): query = {} if query\_string: for param in query\_string.split('&'): if '=' in param: key, value = param.split('=') query[key] = urllib.parse.unquote(value) query[key] = self.sanitize\_input(query[key]) else: logging.error(f"Malformed parameter in query string: {param}") return {} else: logging.error("Query string is empty or missing.") return query def sanitize\_input(value, allowed\_characters=None): """ 入力値をサニタイズして安全にする関数 - XSS防止のためにHTMLエスケープを実施 - オプションで許可される文字のみをフィルタリング可能 """ # HTMLエスケープ (XSS対策) sanitized\_value = html.escape(value) # 許可されていない文字を除去（オプション） if allowed\_characters: sanitized\_value = re.sub(f"[^{re.escape(allowed\_characters)}]", "", sanitized\_value) return sanitized\_value def jsonify(self, data): print(json.dumps(data, indent=4)) ''' admin\_routes = tkCGIApplication() admin\_routes.route('/dashboard')(dashboard\_handler) app.add\_blueprint(admin\_routes, url\_prefix='/admin') ''' def add\_blueprint(self, blueprint, url\_prefix=""): for route, handler in blueprint.routes.items(): self.routes[url\_prefix + route] = handler def get\_account\_list(self): logging.info(f"get\_account\_list: reading {self.account\_path}") if os.path.exists(self.account\_path): with open(self.account\_path, 'r') as file: try: accounts = yaml.safe\_load(file) logging.info(f"Accounts loaded successfully: {accounts}") return accounts except yaml.YAMLError as e: logging.error(f"Error parsing YAML file: {e}") return None else: logging.error(f"{self.account\_path} file not found.") return None def get\_account(self, user, account\_list): return next((account for account in account\_list if account['user'] == user), None) def render\_template(self, template\_name, \*\*context): ''' html\_output = render\_template( 'example\_template.html', title="ようこそ", content="これはサンプルのページです。" ) print(html\_output) ''' with open(template\_name, 'r', encoding='utf-8') as file: template = file.read() for key, value in context.items(): pattern = r"{{\s\*" + re.escape(key) + r"\s\*}}" # スペースがあってもなくても対応 template = re.sub(pattern, str(value), template) return template def run(self, redirect=True, error\_handler=None, target="json"): """ アプリケーションのエントリーポイント。 - redirect: print 関数をカスタムにリダイレクトするかどうか。 - error\_handler: カスタムエラーハンドラーを登録。 - target: 出力形式 ("json" または HTML ページ) を指定。 """ if redirect: self.redirect\_print() if error\_handler is None: self.register\_error\_handler(404, lambda: print(json.dumps({'result': False, 'message': "Not Found."}, indent=4))) else: self.register\_error\_handler(404, error\_handler) self.init\_html(target=target) query\_string = os.getenv('QUERY\_STRING', '') query = self.parse\_query\_string(query\_string) # アクションのルーティング if query: action = query.get('action') if action in self.routes: return self.routes[action](query) # 登録された関数を呼び出し else: # 定義されていないアクション if error\_handler: error\_handler(404) # カスタムエラーハンドラーを使用 else: response = { 'result': False, 'message': f"Action '{action}' is not supported." } print(json.dumps(response, indent=4)) else: # クエリが空または不正 print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4)) self.end\_html()

**Copilot の発言**

エラーの原因と改善点を探りながら、コード全体を見直してみました。以下のポイントに基づいて修正箇所を提案します。

**見直しの主なポイント**

1. sanitize\_input**の定義ミス**:
   * メソッドとして定義されていますが、self がないためクラス内で呼び出すとエラーになります。
   * 修正：def sanitize\_input(self, value, allowed\_characters=None) とする必要があります。
2. **未定義の関数の参照**:
   * validate\_account の中で呼び出している get\_account\_list や get\_account がクラスのインスタンスメソッドとして存在するので、self.get\_account\_list() や self.get\_account() に修正する必要があります。
3. **グローバル変数の不整合**:
   * validate\_account 内で定義されているが、is\_valid\_password と is\_valid\_ip がグローバル関数として使われているため、self.is\_valid\_password() や self.is\_valid\_ip() に修正が必要です。
4. init\_html**ターゲット条件のチェック**:
   * self.target == "" の場合、end\_html が正しく動作しません。デフォルトターゲットの設定と条件の見直しが必要です。
5. **未使用変数やメソッド**:
   * before\_request と after\_request のデコレータが正しく適用されていないため、実装を削除または再設計が必要。

**修正版コード**

以下のコードはエラーを修正し、不要な箇所を整理したものです：

python

コピー

import os

import re

import builtins

import functools

import logging

import json

import yaml

import html

import urllib.parse

class tkCGIApplication:

# クラス変数としてオリジナルの print 関数を退避

print\_original = builtins.print

def \_\_init\_\_(self, log\_file, account\_path):

self.print = builtins.print

self.html\_initialized = False

self.log\_file = log\_file

self.account\_path = account\_path

self.routes = {}

self.error\_handlers = {}

# ログの設定

self.setup\_logging()

def setup\_logging(self):

if not os.path.exists(os.path.dirname(self.log\_file)):

os.makedirs(os.path.dirname(self.log\_file))

if os.access(os.path.dirname(self.log\_file), os.W\_OK):

logging.basicConfig(

filename=self.log\_file,

level=logging.INFO,

format='%(asctime)s %(message)s'

)

else:

self.print({'result': False, 'message': 'Logging directory is not writable.'}, indent=4)

raise SystemExit("Critical error: Unable to initialize logging.")

def print\_custom(self, \*args, \*\*kwargs):

if not self.html\_initialized:

self.init\_html()

# `<pre>` タグで囲んで表示

self.print\_original("<pre>")

self.print\_original(\*args, \*\*kwargs)

self.print\_original("</pre>")

def redirect\_print(self, print\_func=None):

if print\_func:

builtins.print = print\_func

else:

builtins.print = self.print\_custom

def init\_html(self, target="json", charset="utf-8", title="title", html\_ver="HTML5", force\_init=False):

if not force\_init and self.html\_initialized:

return

self.html\_initialized = True

self.target = target

if self.target == "json":

print(f"Content-Type: application/json")

print()

else:

# HTMLヘッダーを出力

print(f"Content-Type: text/html; charset={charset}")

print()

print("<!DOCTYPE html>")

print("<html>")

print("<head>")

if html\_ver == "HTML5":

print(f" <meta charset=\"{charset}\">")

else:

print(f" <meta http-equiv=\"Content-Type\" content=\"text/html; charset={charset}\">")

print(f" <title>{title}</title>")

print("</head>")

print("<body>")

def end\_html(self):

"""HTMLの終了タグを出力"""

if self.html\_initialized and self.target != "json":

print("</body>")

print("</html>")

def log(self, message):

"""Info レベルのログを記録"""

logging.info(message)

def log\_warning(self, message):

"""Warning レベルのログを記録"""

logging.warning(message)

def log\_error(self, message):

"""error レベルのログを記録"""

logging.error(message)

def route(self, action):

def decorator(func):

self.routes[action] = func

@functools.wraps(func)

def wrapper(\*args, \*\*kwargs):

return func(\*args, \*\*kwargs)

return wrapper

return decorator

def register\_error\_handler(self, code, handler):

self.error\_handlers[code] = handler

def handle\_error(self, code):

handler = self.error\_handlers.get(code)

if handler:

return handler()

else:

# デフォルトエラー応答

print(json.dumps({'result': False, 'message': f"Error {code} occurred."}, indent=4))

logging.error(f"Unhandled error {code}")

def validate\_account(self, func):

def wrapper(\*args, \*\*kwargs):

logging.info(f"Validating account for function: {func.\_\_name\_\_}")

# QUERY\_STRINGから必要な情報を取得

query = kwargs.get('query', {})

user = query.get('user', '')

password = query.get('pw', '')

clientIP = os.getenv('REMOTE\_ADDR', '')

# アカウントリストの取得

account\_list = self.get\_account\_list()

if not account\_list:

self.log\_error("Failed to load account list.")

print(json.dumps({'result': False, 'message': "Failed to load account list."}, indent=4))

return

# アカウント取得

account = self.get\_account(user, account\_list)

if not account:

self.log\_warning(f"Unauthorized user access attempt: user={user}")

print(json.dumps({'result': False, 'message': "Invalid user or unauthorized access."}, indent=4))

return

# パスワードの確認

if not self.is\_valid\_password(account, password):

self.log\_warning(f"Invalid password attempt for user: {user}")

print(json.dumps({'result': False, 'message': "Invalid password."}, indent=4))

return

# IPアドレスの確認

if not self.is\_valid\_ip(account, clientIP):

self.log\_warning(f"Unauthorized IP access attempt for user: {user}, IP: {clientIP}")

print(json.dumps({'result': False, 'message': "Unauthorized IP address."}, indent=4))

return

# アカウントが有効であれば関数を実行

return func(account, query, \*args, \*\*kwargs)

return wrapper

def parse\_query\_string(self, query\_string):

query = {}

if query\_string:

for param in query\_string.split('&'):

if '=' in param:

key, value = param.split('=')

query[key] = urllib.parse.unquote(value)

query[key] = self.sanitize\_input(value=key)

else:

self.log\_warning(f"Malformed parameter in query string: {param}")

else:

self.log\_warning("Query string is empty or missing.")

return query

def sanitize\_input(self, value, allowed\_characters=None):

sanitized\_value = html.escape(value)

if allowed\_characters:

sanitized\_value = re.sub(f"[^{re.escape(allowed\_characters)}]", "", sanitized\_value)

return sanitized\_value

def get\_account\_list(self):

self.log(f"get\_account\_list: reading {self.account\_path}")

if os.path.exists(self.account\_path):

with open(self.account\_path, 'r') as file:

try:

accounts = yaml.safe\_load(file)

self.log(f"Accounts loaded successfully: {accounts}")

return accounts

except yaml.YAMLError as e:

self.log\_error(f"Error parsing YAML file: {e}")

return None

else:

self.log\_warning(f"{self.account\_path} file not found.")

return None

def get\_account(self, user, account\_list):

return next((account for account in account\_list if account['user'] == user), None)

def run(self, redirect=True, error\_handler=None, target="json"):

if redirect:

self.redirect\_print()

if error\_handler is None:

self.register\_error\_handler(404, lambda: print(json.dumps({'result': False, 'message': "Not Found."}, indent=4)))

else:

self.register\_error\_handler(404, error\_handler)

self.init\_html(target=target)

query\_string = os.getenv('QUERY\_STRING', '')

query = self.parse\_query\_string(query\_string)

if query:

action = query.get('action')

if action in self.routes:

return self.routes[action](query)

else:

response = {'result': False, 'message': f"Action '{action}' is not supported."}

print(json.dumps

**あなたの発言**

run()関数が途中で切れています。run()関数部分をしゅつりょきうしなおしてください

**Copilot の発言**

以下に、修正版の run() 関数を完全な形で出力します。この関数は、前回の途中で切れていた部分を補足し、全体的な流れが正しく動作するように修正されています。

**完全な run() 関数**

python

コピー

def run(self, redirect=True, error\_handler=None, target="json"):

"""

アプリケーションのエントリーポイント。

- redirect: print 関数をカスタムにリダイレクトするかどうか。

- error\_handler: カスタムエラーハンドラーを登録。

- target: 出力形式 ("json" または HTML ページ) を指定。

"""

# print をリダイレクト

if redirect:

self.redirect\_print()

# エラーハンドラーを登録（デフォルトハンドラーを使用可能）

if error\_handler is None:

self.register\_error\_handler(

404, lambda: print(json.dumps({'result': False, 'message': "Not Found."}, indent=4))

)

else:

self.register\_error\_handler(404, error\_handler)

# HTML または JSON の初期化

self.init\_html(target=target)

# QUERY\_STRING を取得

query\_string = os.getenv('QUERY\_STRING', '')

# クエリを解析

query = self.parse\_query\_string(query\_string)

# アクションのルーティング

if query:

action = query.get('action')

if action in self.routes:

try:

# 登録された関数を呼び出し

return self.routes[action](query)

except Exception as e:

# ハンドリングされていない例外をキャッチしてログに記録

self.log\_error(f"Unexpected error occurred: {e}")

print(json.dumps({'result': False, 'message': "An internal error occurred."}, indent=4))

else:

# 定義されていないアクションの場合

if 404 in self.error\_handlers:

self.handle\_error(404)

else:

response = {

'result': False,

'message': f"Action '{action}' is not supported."

}

print(json.dumps(response, indent=4))

else:

# クエリが空または不正

print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

# HTML の終了タグを出力

self.end\_html()

**修正点と特徴**

1. **完全な例外処理**:
   * 関数の実行中に例外が発生した場合（例えば、ルート関数がエラーをスローする場合）、log\_error を使用してエラーを記録し、ユーザーに内部エラーが発生したことを通知します。
2. **デフォルトおよびカスタムのエラーハンドラーの柔軟性**:
   * デフォルトでは404エラーのためのシンプルなJSON応答を用意していますが、カスタムハンドラーをerror\_handlerとして登録できます。
3. **HTML終了タグの呼び出し**:
   * 出力形式がHTMLの場合（target="html"）、end\_html() を使用して適切に終了します。
4. self.routes**によるルーティング**:
   * クエリからアクションを取得し、登録されたルート関数を安全に呼び出します。
   * アクションが登録されていない場合にハンドラーを使用して応答します。
5. **リダイレクトのオプション**:
   * redirect=True の設定でprintをカスタム関数に差し替え、必要に応じてオフにすることも可能。

**使用例**

main() では次のように簡潔に run() を呼び出すことができます：

python

コピー

def main():

app.run(target="json")

if \_\_name\_\_ == "\_\_main\_\_":

main()

この修正により、run() 関数は堅牢性と柔軟性を備えた設計となり、エラー処理やルーティングが効率的かつ安全に行えるようになりました。他に改良点や質問があれば教えてくださいね！ 🌟✨

**あなたの発言**

classに入れられるように、インデントを追加していただけますか

**Copilot の発言**

もちろんです！以下は、tkCGIApplication クラスの run() メソッドにインデントを適切に追加した修正版です。この形式でクラス内の一部として動作します。

**インデント付きの run() メソッド**

python

コピー

def run(self, redirect=True, error\_handler=None, target="json"):

"""

アプリケーションのエントリーポイント。

- redirect: print 関数をカスタムにリダイレクトするかどうか。

- error\_handler: カスタムエラーハンドラーを登録。

- target: 出力形式 ("json" または HTML ページ) を指定。

"""

# print をリダイレクト

if redirect:

self.redirect\_print()

# エラーハンドラーを登録（デフォルトハンドラーを使用可能）

if error\_handler is None:

self.register\_error\_handler(

404, lambda: print(json.dumps({'result': False, 'message': "Not Found."}, indent=4))

)

else:

self.register\_error\_handler(404, error\_handler)

# HTML または JSON の初期化

self.init\_html(target=target)

# QUERY\_STRING を取得

query\_string = os.getenv('QUERY\_STRING', '')

# クエリを解析

query = self.parse\_query\_string(query\_string)

# アクションのルーティング

if query:

action = query.get('action')

if action in self.routes:

try:

# 登録された関数を呼び出し

return self.routes[action](query)

except Exception as e:

# ハンドリングされていない例外をキャッチしてログに記録

self.log\_error(f"Unexpected error occurred: {e}")

print(json.dumps({'result': False, 'message': "An internal error occurred."}, indent=4))

else:

# 定義されていないアクションの場合

if 404 in self.error\_handlers:

self.handle\_error(404)

else:

response = {

'result': False,

'message': f"Action '{action}' is not supported."

}

print(json.dumps(response, indent=4))

else:

# クエリが空または不正

print(json.dumps({'result': False, 'message': "Invalid query parameters."}, indent=4))

# HTML の終了タグを出力

self.end\_html()

**修正後の特徴**

1. **クラス内のインデントに対応**:
   * 全てのコードを適切なレベルでインデントし、クラス内で問題なく動作するように調整しました。
2. **見やすいコメントと説明**:
   * 各主要部分にコメントを追加し、コードの意図を明確化。
3. **例外処理の追加**:
   * try/except ブロックにより、意図しないエラーが発生しても適切にログを記録し、エラーメッセージを返すように設計。

この修正版をクラスに組み込んでお試しください！他にも調整や追加したい機能があれば教えてくださいね 🦊✨

**あなたの発言**

次のtkCGI.pyを使って、API\_test.pyにqueryを出すCLIプログラムを作ってください

#tkCGI.py import sys import urllib.parse import html import os class Request: def \_\_init\_\_(self, query): self.args = query self.remote\_addr = os.getenv('REMOTE\_ADDR', '') #API\_test.py #!/usr/bin/env python3 import os import sys import json import glob sys.path.append('d:/programs/cgi/lib') from tkCGIApplication import tkCGIApplication # tkCGIApplicationインスタンスを作成 app = tkCGIApplication( log\_file='d:/programs/cgi/log/API.log', account\_path='d:/programs/cgi/data/accounts.yaml' ) @app.route('getFile') @app.validate\_account def get\_files(account, query): # 必要なクエリパラメータを取得 rootDir = query.get('rootDir', '') fileMask = query.get('fileMask', '') # パスとマスクのバリデーション if not rootDir or not fileMask: app.log\_warning(f"Invalid rootDir or fileMask.") print(json.dumps({'result': False, 'message': "Invalid rootDir or fileMask."}, indent=4)) return # ファイル検索処理 try: pathList = glob.glob(os.path.join(os.path.abspath(rootDir), fileMask)) response = {'result': True, 'pathList': pathList} app.log(f"File search successful. user={account['user']}, rootDir={rootDir}, fileMask={fileMask}") print(json.dumps(response, indent=4)) except Exception as e: app.log\_error(f"Error during file search: {e}") print(json.dumps({'result': False, 'message': f"Error during file search: {e}"}, indent=4)) def main(): app.run(redirect = True, target = "json") if \_\_name\_\_ == "\_\_main\_\_": main()

**Copilot の発言**

以下は tkCGI.py と API\_test.py を利用して、CLIプログラムとしてクエリを送信するコードです。このCLIプログラムは、クエリ文字列を標準入力やコマンドライン引数から受け取り、内部的にCGI環境を模倣して API\_test.py を呼び出します。

**CLIプログラム: cli\_query.py**

python

コピー

#!/usr/bin/env python3

import os

import sys

import urllib.parse

import subprocess

def main():

# クエリパラメータをCLIから受け取る

if len(sys.argv) < 2:

print("Usage: cli\_query.py <query\_string>")

print("Example: cli\_query.py 'action=getFile&rootDir=/path/to/dir&fileMask=\*.txt'")

sys.exit(1)

query\_string = sys.argv[1]

# QUERY\_STRING環境変数を設定

os.environ['QUERY\_STRING'] = query\_string

# API\_test.py を呼び出す

try:

# `API\_test.py` を実行して結果を取得

result = subprocess.run(

[sys.executable, 'API\_test.py'], # sys.executable は Python 実行ファイルへのパス

stdout=subprocess.PIPE,

stderr=subprocess.PIPE,

text=True

)

# 標準出力を表示

print("Response:")

print(result.stdout)

# 標準エラーを表示（必要に応じて）

if result.stderr:

print("Errors:", file=sys.stderr)

print(result.stderr, file=sys.stderr)

except Exception as e:

print(f"Error occurred while executing API\_test.py: {e}", file=sys.stderr)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**CLIプログラムの実行方法**

1. **プログラムの配置**
   * API\_test.py と cli\_query.py を同じディレクトリに配置します。
2. **CLIプログラムの起動**
   * コマンドラインから以下のように実行してください：

bash

コピー

python cli\_query.py "action=getFile&rootDir=/path/to/dir&fileMask=\*.txt"

1. **出力結果**
   * プログラムは内部的に QUERY\_STRING 環境変数を設定し、API\_test.py を呼び出します。
   * その後、API\_test.py の標準出力を取得し、CLIに表示します。

**ポイント**

1. **環境変数の設定**
   * QUERY\_STRING を環境変数として設定することで、API\_test.py 内でCGIスクリプトの動作を模倣。
2. **サブプロセス呼び出し**
   * subprocess.run を利用して API\_test.py を実行。出力は標準出力（stdout）とエラー出力（stderr）で受け取ります。
3. **柔軟なクエリ処理**
   * クエリ文字列はCLI引数から受け取るため、動的に異なるリクエストを簡単にテストできます。

このCLIプログラムで、API\_test.py の動作を外部から簡単にテスト可能です！もしさらに改良や機能拡張が必要でしたら、お気軽にお知らせくださいね！✨🐾

**あなたの発言**

API\_test.pyは、http://localhost:8000/cgi-bin/API\_test.pyです。

**Copilot の発言**

なるほど、API\_test.py が CGI スクリプトとして http://localhost:8000/cgi-bin/API\_test.py で動作するわけですね。それを利用してクエリを送信する CLI プログラムを作成しましょう。以下にそのコードを示します。このプログラムは HTTP リクエストを送信し、サーバーの応答を取得して表示します。

**CLIプログラム: cli\_query.py**

python

コピー

#!/usr/bin/env python3

import requests

import sys

def main():

# コマンドライン引数からクエリを取得

if len(sys.argv) < 2:

print("Usage: cli\_query.py <query\_string>")

print("Example: cli\_query.py 'action=getFile&rootDir=/path/to/dir&fileMask=\*.txt'")

sys.exit(1)

query\_string = sys.argv[1]

url = "http://localhost:8000/cgi-bin/API\_test.py"

try:

# HTTP GET リクエストを送信

response = requests.get(f"{url}?{query\_string}")

# HTTP ステータスコードを確認

if response.status\_code == 200:

print("Response:")

print(response.text)

else:

print(f"Error: Server responded with status code {response.status\_code}")

print(response.text)

except requests.exceptions.RequestException as e:

print(f"Error: Could not send request to server. {e}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**説明**

1. requests**モジュール**:
   * HTTP リクエストを送信するために使用します。GET リクエストを使ってクエリを送信しています。
2. **クエリ送信**:
   * コマンドライン引数からクエリ文字列を取得し、それをエンコードしてサーバーに送信します。
3. **エラーハンドリング**:
   * サーバーがエラーを返した場合や通信エラーが発生した場合に適切にハンドリングします。
4. **HTTP サーバーの URL**:
   * http://localhost:8000/cgi-bin/API\_test.py をターゲット URL として使用。

**使い方**

1. **プログラムの配置**:
   * cli\_query.py を任意のディレクトリに保存します。
   * 必要に応じて requests ライブラリをインストールしてください：

bash

コピー

pip install requests

1. **実行方法**:
   * コマンドラインから次のように実行します：

bash

コピー

python cli\_query.py "action=getFile&rootDir=/path/to/dir&fileMask=\*.txt"

1. **サーバーの応答**:
   * サーバーが返すレスポンスがそのまま CLI に表示されます。

これで、API\_test.py を外部からテストするための簡単な CLI プログラムが完成しました。これを活用して、さまざまなクエリを試してみてください！🦊✨