# 1. /telegram\_planfix\_integration\_work\_in\_progress/main.py  
  
import asyncio  
import secrets  
import uuid  
from concurrent.futures import ThreadPoolExecutor  
from enum import Enum  
from functools import lru\_cache, partial  
from typing import Dict, Union, Tuple, List, Any  
  
from flask import Flask, request, jsonify, Response  
from telethon import TelegramClient, events  
  
from config import telegram\_api\_id, telegram\_api\_hash  
from flask\_logging import logger  
from telegram\_integration\_planfix import send\_message\_to\_planfix, send\_telegram\_message\_to\_planfix  
  
  
app = Flask(\_\_name\_\_)  
  
  
executor = ThreadPoolExecutor()  
  
  
# Декоратор lru\_cache для функции get\_clients, который кэширует результаты вызовов  
@lru\_cache(maxsize=None) # None означает неограниченный размер кэша  
def get\_clients() -> Dict:  
 *"""Get clients dictionary."""* return {}  
  
  
# Получение интернируемого объекта словаря клиентов  
clients = get\_clients()  
  
# Настройка Telethon с использованием TelegramClient  
client = TelegramClient('bot\_session', telegram\_api\_id, telegram\_api\_hash)  
  
  
# Определение перечисления (Enum) ClientStatus, представляющего статусы клиента  
class ClientStatus(Enum):  
 ONLINE = 'Online'  
 OFFLINE = 'Offline'  
  
  
# Определение перечисления (Enum) MessageType, представляющего типы сообщений  
class MessageType(Enum):  
 TELEGRAM = 'Telegram'  
 PLANFIX = 'Planfix'  
  
  
# Обработчик ошибок для исключений типа Exception  
@app.errorhandler(Exception)  
def handle\_error(error: Exception) -> Tuple[Response, int]:  
 *"""Handle errors.  
  
 Args:  
 error (Exception): The exception object.  
  
 Returns:  
 Tuple[Response, int]: A tuple containing a JSON response and an HTTP status code.  
 """* # Логирование ошибки  
 logger.error(f"An error occurred: {error}")  
 # Возврат JSON-ответа с описанием ошибки и кодом состояния 500 (Internal Server Error)  
 return jsonify({'error': str(error)}), 500  
  
  
# Асинхронная функция для запуска процесса polling с использованием Telethon  
async def run\_polling() -> None:  
 *"""Run Telethon polling."""* try:  
 logger.info("Starting Telethon polling...")  
 # Асинхронное подключение к Telegram  
 await client.connect()  
 # Асинхронный запуск polling до отключения  
 await client.run\_until\_disconnected()  
 except Exception as error:  
 logger.error(f"Error in Telethon polling: {error}")  
  
  
# Асинхронная функция для верификации ключа сессии клиента  
async def verify\_session\_key(client\_data: Dict[str, Union[str, Dict]]) -> bool:  
 *"""Verify session key.  
  
 Args:  
 client\_data (Dict[str, Union[str, Dict]]): Client data containing the session key.  
  
 Returns:  
 bool: True if the session key is valid, False otherwise.  
 """* # Получение сохраненного ключа из данных клиента  
 stored\_key = client\_data["session\_key"]  
 # Извлечение текущего ключа из данных клиента  
 key = client\_data.get("session\_key")  
 # Возвращение результата сравнения ключей  
 return stored\_key == key  
  
  
# Асинхронная функция для валидации наличия обязательных ключей в данных  
async def validate\_data(data: Dict[str, Union[str, Dict]], required\_keys: List[str]):  
 *"""Validate data keys.  
  
 Args:  
 data (Dict[str, Union[str, Dict]]): Data to be validated.  
 required\_keys (List[str]): List of required keys.  
  
 Raises:  
 ValueError: Raised if any required key is missing in the data.  
 """* # Проверка наличия всех обязательных ключей в данных  
 if not all(key in data for key in required\_keys):  
 # В случае отсутствия ключей вызывается исключение ValueError  
 raise ValueError("Missing required keys in data")  
  
  
# Асинхронная функция для валидации наличия обязательных ключей в данных сообщения  
async def validate\_message\_data(message\_data: Dict[str, Union[str, Dict]], required\_keys: List[str]):  
 *"""Validate message data keys.  
  
 Args:  
 message\_data (Dict[str, Union[str, Dict]]): Message data to be validated.  
 required\_keys (List[str]): List of required keys.  
  
 Raises:  
 ValueError: Raised if any required key is missing in the message data.  
 """* # Проверка наличия всех обязательных ключей в данных сообщения  
 if not all(key in message\_data for key in required\_keys):  
 # В случае отсутствия ключей вызывается исключение ValueError  
 raise ValueError("Missing required keys in messageData")  
  
  
# Асинхронная функция для генерации ключа сессии клиента  
async def generate\_client\_session\_key(length: int = 32) -> str:  
 *"""Generate client session key.  
  
 Args:  
 length (int): The length of the session key.  
  
 Returns:  
 str: The generated client session key.  
 """* # Проверка, что длина ключа положительна  
 if length <= 0:  
 raise ValueError("Длина ключа сессии должна быть положительным целым числом.")  
 # Генерация шестнадцатеричного токена указанной длины  
 return secrets.token\_hex(length)  
  
  
# Асинхронная функция для генерации уникального идентификатора клиента  
async def generate\_client\_id() -> str:  
 *"""Generate client ID.  
  
 Returns:  
 str: The generated client ID.  
 """* # Генерация уникального идентификатора клиента  
 return str(uuid.uuid4())  
  
  
# Асинхронная функция для создания нового клиента  
async def create\_client() -> Dict[str, Union[str, Dict]]:  
 *"""Create a new client.  
  
 Returns:  
 Dict[str, Union[str, Dict]]: Data of the new client.  
 """* # Генерация уникального идентификатора и ключа сессии  
 client\_id = await generate\_client\_id()  
 session\_key = await generate\_client\_session\_key()  
 # Возвращение данных нового клиента в виде словаря  
 return {"client\_id": client\_id, "session\_key": session\_key, "telegram\_to\_planfix": {}}  
  
  
# Асинхронная функция для получения данных клиента  
async def get\_client\_data() -> Dict[str, Union[str, Dict]]:  
 *"""Get client data.  
  
 Returns:  
 Dict[str, Union[str, Dict]]: Data of the client.  
 """* # Проверка, что словарь клиентов не пустой  
 if not clients:  
 # Генерация нового клиента и добавление его в словарь  
 client\_id = generate\_client\_id()  
 clients[client\_id] = await create\_client()  
  
 # Получение первого ключа словаря клиентов  
 client\_id = list(clients.keys())[0]  
 # Получение данных клиента по ключу  
 client\_data = clients[client\_id]  
  
 # Проверка валидности ключа сессии  
 if not await verify\_session\_key(client\_data):  
 # В случае невалидного ключа вызывается исключение ValueError  
 raise ValueError("Invalid session key")  
  
 # Возврат данных клиента в виде словаря  
 return client\_data  
  
  
# Обработчик событий нового сообщения (исходящего и входящего)  
@client.on(events.NewMessage(outgoing=True, incoming=True))  
async def handle\_text\_message(event: events.NewMessage.Event) -> None:  
 *"""Handle incoming and outgoing text messages.  
  
 Arguments:  
 - event (events.NewMessage.Event): The event representing a new message, either incoming or outgoing.  
  
 Returns:  
 None  
  
 This function handles incoming and outgoing text messages, logging relevant information and processing the messages  
 accordingly. It uses the Flask application context and includes debug logs for message details. The handling  
 includes checking for the presence of the 'id' attribute, message text, and dispatching messages to Planfix.  
 """* # С использованием контекста приложения Flask  
 with app.app\_context():  
 try:  
 # Отладочные логи для вывода информации о сообщении и его типе  
 logger.debug(f"Message: {event.message}")  
 logger.debug(f"Type of message: {type(event.message)}")  
  
 # Проверка наличия атрибута 'id' у сообщения  
 if hasattr(event, 'message') and hasattr(event.message, 'id'):  
 logger.debug(f"Message ID: {event.message.id}")  
 else:  
 logger.warning("Message or message has no 'id' attribute")  
  
 # Проверка наличия текста в сообщении  
 if event.message.text:  
 message = event.message.text  
 logger.debug(f"Message text: {event.message.text}")  
 logger.debug(f"Type of message text: {type(event.message.text)}")  
  
 # Получение данных клиента  
 client\_data = await get\_client\_data()  
 logger.debug(f"Client data: {client\_data}")  
  
 try:  
 # Проверка наличия атрибута 'id' у сообщения  
 if hasattr(event, 'message') and hasattr(event.message, 'id'):  
 logger.debug(f"Message ID: {event.message.id}")  
 else:  
 logger.warning("Message or message has no 'id' attribute")  
 except Exception as id\_error:  
 logger.error(f"Error checking 'id' attribute: {id\_error}")  
  
 # Обработка входящего сообщения  
 if not event.message.out:  
 logger.debug("Handling incoming message.")  
 await send\_telegram\_message\_to\_planfix(event, client\_data, event.message.id)  
 else:  
 logger.debug("Handling outgoing message.")  
 # Добавьте свою логику обработки исходящего сообщения здесь  
 else:  
 logger.warning("Message has no text")  
  
 except Exception as error:  
 # Обработка ошибок при обработке текстового сообщения  
 logger.error(f"Error handling text message: {error}")  
  
  
# Обработчик маршрута /webhook с поддержкой различных HTTP-методов  
@app.route('/webhook', methods=['POST', 'GET', 'DELETE', 'PUT'])  
def handle\_webhook() -> Tuple[Response, int]:  
 *"""Handle webhook requests.  
  
 Arguments:  
 None  
  
 Returns:  
 Tuple[Response, int]  
  
 This function handles HTTP requests to the '/webhook' route, supporting various HTTP methods (POST, GET, DELETE,  
 PUT). It determines the appropriate asynchronous coroutine based on the request method and executes it, returning  
 the result as a tuple containing a Flask Response object and an HTTP status code.  
 """* try:  
 # Получение текущего цикла событий asyncio  
 loop = asyncio.get\_event\_loop()  
  
 # Определение асинхронной корутины в зависимости от метода HTTP-запроса  
 if request.method == 'POST':  
 coro = handle\_post(request.json)  
 elif request.method == 'GET':  
 coro = handle\_get()  
 elif request.method == 'DELETE':  
 coro = handle\_delete()  
 elif request.method == 'PUT':  
 coro = handle\_put(request.json)  
 else:  
 # Обработка неверного метода запроса  
 response: Response = jsonify({'error': 'Invalid request method'})  
 return response, 400  
  
 # Запуск асинхронной корутины и возврат результата  
 result: Any = loop.run\_until\_complete(loop.create\_task(coro))  
 return result if isinstance(result, tuple) else (result, 200)  
  
 except Exception as error:  
 # Обработка и возврат ошибки в случае исключения  
 response: Response = jsonify({'error': str(error)})  
 return response, 500  
  
  
# Асинхронная функция для обработки входящих запросов  
async def handle\_request(data: Dict[str, Union[str, Dict]], send\_function) -> Union[  
 Dict[str, Union[str, Dict]], Tuple[Dict[str, str], int]]:  
 *"""Handle incoming requests.  
  
 Arguments:  
 - data (Dict[str, Union[str, Dict]]): The data representing the incoming request.  
 - send\_function: The function responsible for sending the message to Planfix.  
  
 Returns:  
 Union[Dict[str, Union[str, Dict]], Tuple[Dict[str, str], int]]  
  
 This asynchronous function handles incoming requests, validates the required data, retrieves client data, and sends  
 the message to Planfix. It returns a dictionary with a successful status or an error message along with the HTTP  
 status code.  
 """* try:  
 # Проверка наличия обязательных ключей в данных запроса  
 required\_keys = ['messageData', 'telegram\_message\_id']  
 await validate\_data(data, required\_keys)  
  
 # Получение данных о сообщении из запроса  
 message\_data = data['messageData']  
 # Проверка наличия обязательных ключей в данных сообщения  
 required\_message\_keys = ['text', 'telegram\_message\_id']  
 await validate\_message\_data(message\_data, required\_message\_keys)  
  
 # Получение данных клиента  
 client\_data, success\_response, error\_response = get\_client\_data()  
 if client\_data is None:  
 logger.error("Client data not found")  
 # Возврат словаря с ошибкой в случае отсутствия данных клиента  
 return error\_response  
  
 # Получение текста сообщения  
 message\_text = message\_data.get('text')  
 logger.debug(f"Message text: {message\_text}")  
  
 # Вызов функции отправки события в Planfix и получение ID сообщения  
 planfix\_message\_id = await send\_function(message\_data, client\_data, message\_data['telegram\_message\_id'])  
 logger.debug(f"Planfix message ID: {planfix\_message\_id}")  
  
 # Инициализация словаря, если не существует  
 if 'telegram\_to\_planfix' not in client\_data:  
 client\_data['telegram\_to\_planfix'] = {}  
  
 # Сохранение соответствия ID сообщения в Telegram и Planfix  
 client\_data['telegram\_to\_planfix'][message\_data['telegram\_message\_id']] = planfix\_message\_id  
 # Обновление статуса клиента  
 client\_data['status'] = ClientStatus.ONLINE  
 logger.info("Successfully handled request")  
  
 # Возврат словаря с успешным ответом  
 return {'status': client\_data['status'], 'token': client\_data['token']}  
 except Exception as error:  
 logger.error(f"An error occurred in handle\_request: {error}")  
 # Возврат словаря с ошибкой и статусом 500  
 return {'error': str(error)}, 500  
  
  
# Асинхронная функция для обработки PUT-запросов  
async def handle\_put(data: Dict[str, Union[str, Dict]]) -> Union[  
 Dict[str, Union[str, Dict]], Tuple[Dict[str, str], int]]:  
 *"""Handle PUT requests.  
  
 Arguments:  
 - data (Dict[str, Union[str, Dict]]): The data representing the incoming PUT request.  
  
 Returns:  
 Union[Dict[str, Union[str, Dict]], Tuple[Dict[str, str], int]]  
  
 This asynchronous function handles incoming PUT requests by calling the handle\_request function with the appropriate  
 send function. It returns a dictionary with a successful status or an error message along with the HTTP status  
 code.  
 """* # Вызов функции обработки входящего запроса с использованием функции отправки в Planfix  
 return await handle\_request(data, send\_message\_to\_planfix)  
  
  
# Асинхронная функция для обработки POST-запросов  
async def handle\_post(data: Dict[str, Union[str, Dict]]) -> Union[  
 Dict[str, Union[str, Dict]], Tuple[Dict[str, str], int]]:  
 *"""Handle POST requests.  
  
 Arguments:  
 - data (Dict[str, Union[str, Dict]]): The data representing the incoming POST request.  
  
 Returns:  
 Union[Dict[str, Union[str, Dict]], Tuple[Dict[str, str], int]]  
  
 This asynchronous function handles incoming POST requests by running the handle\_request function in a separate  
 thread. It returns a dictionary with a successful status or an error message along with the HTTP status code.  
 """* # Получение текущего цикла событий  
 loop = asyncio.get\_event\_loop()  
 # Вызов функции обработки входящего запроса в отдельном потоке  
 return await loop.run\_in\_executor(executor, partial(handle\_request, data, send\_telegram\_message\_to\_planfix))  
  
  
# Асинхронная функция для обработки GET-запросов  
async def handle\_get() -> Union[Dict[str, Union[str, Dict]], Tuple[Dict[str, str], int]]:  
 *"""Handle GET requests.  
  
 Arguments:  
 None  
  
 Returns:  
 Union[Dict[str, Union[str, Dict]], Tuple[Dict[str, str], int]]  
  
 This asynchronous function handles incoming GET requests, retrieves client data, and returns a dictionary with a  
 successful status or an error message along with the HTTP status code.  
 """* try:  
 # Получение данных клиента и ответов на успех и ошибку  
 client\_data, success\_response, error\_response = get\_client\_data()  
 if client\_data is None:  
 # Возврат словаря с ошибкой в случае отсутствия данных клиента  
 return error\_response  
  
 # Возврат словаря с успешным ответом, содержащим статус клиента  
 return success\_response({'status': client\_data['status']})  
 except Exception as error:  
 # Возврат словаря с ошибкой и статусом 500 в случае исключения  
 return {'error': str(error)}, 500  
  
  
# Асинхронная функция для обработки DELETE-запросов  
async def handle\_delete() -> Union[Dict[str, Union[str, Dict]], Tuple[Dict[str, str], int]]:  
 *"""Handle DELETE requests.  
  
 Arguments:  
 None  
  
 Returns:  
 Union[Dict[str, Union[str, Dict]], Tuple[Dict[str, str], int]]  
  
 This asynchronous function handles incoming DELETE requests, sets the client status to OFFLINE, and returns a  
 dictionary with a successful status or an error message along with the HTTP status code.  
 """* try:  
 # Получение данных клиента и ответов на успех и ошибку  
 client\_data, success\_response, error\_response = get\_client\_data()  
 if client\_data is None:  
 # Возврат словаря с ошибкой в случае отсутствия данных клиента  
 return error\_response  
  
 # Установка статуса клиента в OFFLINE  
 client\_data['status'] = ClientStatus.OFFLINE  
  
 # Возврат словаря с успешным ответом, содержащим статус клиента  
 return success\_response({'status': client\_data['status']})  
 except Exception as error:  
 # Возврат словаря с ошибкой и статусом 500 в случае исключения  
 return {'error': str(error)}, 500  
  
  
# Блок выполнения кода при запуске программы  
if \_\_name\_\_ == '\_\_main\_\_':  
 try:  
 # Запуск обработчика входящих сообщений от бота Telegram  
 asyncio.get\_event\_loop().run\_until\_complete(run\_polling())  
 # Запуск веб-приложения Flask на порту 5000  
 app.run(port=5000)  
 except KeyboardInterrupt:  
 # Обработка прерывания по Ctrl+C  
 logger.info("Program interrupted by user")  
 except Exception as e:  
 # Обработка неожиданной ошибки  
 logger.error(f"An unexpected error occurred: {e}")

**#########################################################################################################**

# 2./telegram\_planfix\_integration\_work\_in\_progress/telegram\_integration\_planfix.py  
  
from typing import Optional  
  
import aiohttp  
from aiocache import SimpleMemoryCache  
from telethon import TelegramClient  
  
from config import planfix\_api\_url, planfix\_api\_key, telegram\_api\_id, telegram\_api\_hash  
from flask\_logging import logger  
  
# Асинхронный кэш для хранения данных о сообщениях  
messages: SimpleMemoryCache = SimpleMemoryCache()  
  
  
# Асинхронная функция для отправки сообщения из Telegram в Planfix и обработки ответа  
async def send\_telegram\_message\_to\_planfix(update, client\_data, telegram\_message\_id) -> None:  
 *"""Sends a message from Telegram to Planfix and handles the response.  
  
 Args:  
 update: Telethon update object.  
 client\_data: Client data.  
 telegram\_message\_id: Telegram message ID.  
  
 Returns:  
 None  
 """* try:  
 # Проверка наличия объекта update или update.message  
 if update is None or update.message is None:  
 logger.warning("Received None in update or update.message in send\_telegram\_message\_to\_planfix")  
 return  
  
 logger.debug(f"Full update: {update}")  
 logger.debug(f"Client data: {client\_data}")  
 logger.debug(f"Telegram message ID: {telegram\_message\_id}")  
  
 # Проверка наличия текста в сообщении  
 if hasattr(update.message, 'text') and update.message.text:  
 message\_text = update.message.text  
 logger.debug(f"Type of message\_text: {type(message\_text)}, Message text: {message\_text}")  
 else:  
 logger.warning("No text attribute found in update.message")  
 return  
  
 # Отправка сообщения в Planfix и получение идентификатора сообщения  
 planfix\_message\_id = await send\_message\_to\_planfix(client\_data, message\_text)  
 logger.info(f"Planfix message ID: {planfix\_message\_id}")  
  
 # Сохранение соответствия идентификаторов сообщений между Telegram и Planfix  
 await messages.set(planfix\_message\_id, {  
 "telegram\_message\_id": telegram\_message\_id,  
 "telegram\_chat\_id": update.message.chat\_id  
 })  
 logger.info(f"Mapping saved for Planfix message ID: {planfix\_message\_id}")  
  
 # Идентификатор чата в Telegram  
 telegram\_chat\_id = update.message.chat\_id  
 # Вызов асинхронной функции отправки ответа из Planfix в Telegram  
 await send\_response\_to\_telegram(planfix\_message\_id, telegram\_chat\_id)  
 except Exception as e:  
 await handle\_error(e)  
  
  
# Асинхронная функция для отправки ответа из Planfix в Telegram  
async def send\_response\_to\_telegram(planfix\_message\_id, telegram\_chat\_id) -> None:  
 *"""Sends a response from Planfix to Telegram.  
  
 Args:  
 planfix\_message\_id: Planfix message ID.  
 telegram\_chat\_id: Telegram chat ID.  
  
 Returns:  
 None  
 """* try:  
 # Получение текста ответа из Planfix асинхронным образом  
 response\_text\_task = get\_planfix\_response(planfix\_message\_id)  
  
 # Использование клиента Telegram для отправки ответа  
 async with TelegramClient('session\_name', telegram\_api\_id, telegram\_api\_hash) as telegram\_client:  
 response\_text = await response\_text\_task  
  
 # Отправка текста ответа в указанный чат Telegram  
 await telegram\_client.send\_message(telegram\_chat\_id, response\_text)  
 logger.info(f"Sent response to Telegram chat {telegram\_chat\_id}")  
 except Exception as e:  
 logger.error(f"Error sending response to Telegram: {e}")  
 await handle\_error(e)  
  
  
# Асинхронная функция для получения идентификатора сообщения в Telegram из соответствия  
async def get\_telegram\_id\_from\_mapping(planfix\_message\_id) -> Optional[int]:  
 *"""Gets the Telegram message ID from the mapping.  
  
 Args:  
 planfix\_message\_id: Planfix message ID.  
  
 Returns:  
 Optional[int]: Telegram message ID if found, otherwise None.  
 """* try:  
 # Получение данных из асинхронного кеша по идентификатору сообщения в Planfix  
 result = await messages.get(planfix\_message\_id, {})  
  
 # Извлечение идентификатора сообщения в Telegram из полученных данных  
 telegram\_message\_id = result.get("telegram\_message\_id", None)  
 logger.debug(f"Telegram Message ID retrieved successfully. Planfix Message ID: {planfix\_message\_id}, "  
 f"Telegram Message ID: {telegram\_message\_id}")  
 return telegram\_message\_id  
 except Exception as e:  
 logger.error(f"Error in get\_telegram\_id\_from\_mapping: {e}")  
 await handle\_error(e)  
  
  
# Асинхронная функция для получения идентификатора чата в Telegram из соответствия  
async def get\_chat\_id\_by\_message(telegram\_message\_id) -> Optional[int]:  
 *"""Gets the Telegram chat ID from the mapping.  
  
 Args:  
 telegram\_message\_id: Telegram message ID.  
  
 Returns:  
 Optional[int]: Telegram chat ID if found, otherwise None.  
 """* try:  
 # Получение данных из асинхронного кеша по идентификатору сообщения в Telegram  
 result = await messages.get(telegram\_message\_id, {})  
  
 # Извлечение идентификатора чата в Telegram из полученных данных  
 telegram\_chat\_id = result.get("telegram\_chat\_id", None)  
 logger.debug(  
 f"Chat ID retrieved successfully. Telegram Message ID: {telegram\_message\_id}, "  
 f"Telegram Chat ID: {telegram\_chat\_id}")  
 return telegram\_chat\_id  
 except Exception as e:  
 logger.error(f"Error in get\_chat\_id\_by\_message: {e}")  
 await handle\_error(e)  
  
  
# Асинхронная функция для отправки сообщения в Planfix  
async def send\_message\_to\_planfix(client\_data, message) -> int:  
 *"""Sends a message to Planfix.  
  
 Args:  
 client\_data: Client data.  
 message: Text of the message.  
  
 Returns:  
 int: Planfix message ID.  
 """* try:  
 # Установка заголовков для авторизации в Planfix  
 headers = {"Authorization": f"Bearer {planfix\_api\_key}"}  
  
 # Подготовка данных для отправки  
 data = {"client\_id": client\_data["client\_id"], "message": message}  
  
 # Использование aiohttp для отправки запроса POST к Planfix API  
 async with aiohttp.ClientSession() as session:  
 async with session.post(f"{planfix\_api\_url}/messages", headers=headers, json=data) as response:  
 # Проверка статуса ответа  
 response.raise\_for\_status()  
 # Получение идентификатора сообщения из ответа Planfix API  
 response\_json = await response.json()  
 planfix\_message\_id = response\_json["id"]  
 logger.debug(f"Planfix message sent successfully. Message ID: {planfix\_message\_id}")  
 return planfix\_message\_id  
 except aiohttp.ClientError as e:  
 logger.error(f"Error in send\_message\_to\_planfix: {e}")  
 await handle\_error(e)  
  
  
# Асинхронная функция для получения ответа от Planfix по идентификатору сообщения  
async def get\_planfix\_response(message\_id) -> str:  
 *"""Gets the response from Planfix by message ID.  
  
 Args:  
 message\_id: Planfix message ID.  
  
 Returns:  
 str: Text of the Planfix response.  
 """* try:  
 # Установка заголовков для авторизации в Planfix  
 headers = {"Authorization": f"Bearer {planfix\_api\_key}"}  
  
 # Использование aiohttp для отправки запроса GET к Planfix API  
 async with aiohttp.ClientSession() as session:  
 async with session.get(f"{planfix\_api\_url}/messages/{message\_id}", headers=headers) as response:  
 # Проверка статуса ответа  
 response.raise\_for\_status()  
 # Извлечение текста ответа из JSON-ответа  
 response\_text = (await response.json())["text"]  
 logger.debug(f"Planfix response text for message ID {message\_id}: {response\_text}")  
 return response\_text  
 except aiohttp.ClientError as e:  
 logger.error(f"Error in get\_planfix\_response for message ID {message\_id}: {e}")  
 await handle\_error(e)  
  
  
# Асинхронная функция для обработки ошибки  
async def handle\_error(error) -> None:  
 *"""Handles an error.  
  
 Args:  
 error: Error object.  
  
 Returns:  
 None  
 """* if error is not None:  
 logger.error(f"An error occurred: {error}")  
 else:  
 logger.warning("Received None as an error object in handle\_error")

**#########################################################################################################**

# 3./telegram\_planfix\_integration\_work\_in\_progress/config.py  
  
import os  
  
from dotenv import load\_dotenv  
  
# Загрузка переменных среды из файла .env  
load\_dotenv()  
  
  
telegram\_api\_id = '22870332'  
telegram\_api\_hash = 'fafcdb82d4f83d15be0fbd6457f6cd8e'  
telegram\_bot\_token = '6204244389:AAHpf4iebKMkRBXOlZbivaGYnaDTH7cwtlQ'  
planfix\_api\_url = 'https://api.planfix.com/xml/'  
planfix\_api\_key = '0c8271c614a138a01b31c9309208a732'  
  
  
  
# telegram\_api\_id = os.getenv('TELEGRAM\_API\_ID')  
# telegram\_api\_hash = os.getenv('TELEGRAM\_API\_HASH')  
# telegram\_bot\_token = os.getenv('TELEGRAM\_BOT\_TOKEN')  
# planfix\_api\_url = os.getenv('PLANFIX\_API\_URL')  
# planfix\_api\_key = os.getenv('PLANFIX\_API\_KEY')  
  
if not telegram\_api\_id or not telegram\_api\_hash or not telegram\_bot\_token or not planfix\_api\_url or not planfix\_api\_key:  
 print("Ошибка: Пожалуйста, укажите все необходимые переменные окружения "  
 "(TELEGRAM\_API\_ID, TELEGRAM\_API\_HASH, TELEGRAM\_BOT\_TOKEN, PLANFIX\_API\_URL, PLANFIX\_API\_KEY).")

**#########################################################################################################**

# 4./pythonProject\_1/flask\_logging.py  
  
import logging  
  
# Уровень логирования (DEBUG - наиболее подробный уровень)  
leve1 = logging.DEBUG  
  
# Формат записей лога  
# %(asctime)s: Это подставляемый параметр, который будет заменен на текущую дату и время в формате, указанном в  
# настройках логирования. asctime расшифровывается как "as available time", и s указывает, что это строковое  
# представление времени.  
  
# |%(filename)s: Здесь filename представляет имя файла, в котором было выполнено логирование. %s означает, что это  
# строковый параметр.  
  
# |%(lineno)04d: lineno представляет номер строки в исходном коде, откуда было выполнено логирование. %04d указывает на  
# форматирование числа: 4 знака, дополняемые нулями впереди (если необходимо).  
  
# -%(levelname)-5s: levelname представляет уровень логирования (например, DEBUG, INFO, WARNING и т.д.). %5s указывает,  
# что это строка шириной в 5 символов, выравненная по правому краю. Дополнительный минус (-) указывает на выравнивание  
# по левому краю.  
  
# - |: Это просто символы, добавленные для создания разделителей между разными частями лог-записи.  
  
# %(message)s: Это место, где выводится само сообщение лога. %s здесь также означает строковый параметр.  
format1 = '%(asctime)s |%(filename)s |%(lineno)04d-%(levelname)-5s| - | %(message)s |'  
  
# Настройка логирования для записи в файл 'val.log'  
# filemode='a' - режим добавления (append), новые записи добавляются в конец файла  
logging.basicConfig(filename='val.log', format=format1, filemode='a', level=leve1)  
  
# Создание обработчика для вывода логов в консоль  
console\_handler = logging.StreamHandler()  
  
# Установка уровня логирования для обработчика консоли  
console\_handler.setLevel(leve1)  
  
# Создание форматтера для форматирования записей лога  
formatter = logging.Formatter(format1)  
  
# Привязка форматтера к обработчику консоли  
console\_handler.setFormatter(formatter)  
  
# Добавление обработчика консоли к корневому логгеру  
logging.getLogger().addHandler(console\_handler)  
  
# Получение ссылки на корневой логгер  
logger = logging.getLogger()  
  
# Вывод информационного сообщения в лог  
logger.info('hello')