Dokumentacja Projektu Steam Dashboard (PySide6 + Asyncio)

18 października 2025

Spis treści

1 Wprowadzenie

Projekt `Custom-Steam-Dashboard` to aplikacja pulpitu zbudowana w Pythonie z wykorzystaniem biblioteki graficznej **PySide6** (Qt) oraz asynchronicznego programowania (**asyncio** i **qasync**) do pobierania danych z różnych API związanych ze Steam (statystyki graczy, promocje, nowości).

2 Struktura Folderów

Projekt ma logiczną strukturę pakietów, która oddziela warstwę interfejsu użytkownika (UI) od warstwy logiki biznesowej i dostępu do danych (Core/Services).

```
Custom-Steam-Dashboard-main/
          app/
                core/
                      services/
                            deals_api.py         # API do promocji (CheapShark)
                            steam_api.py         # API Steam Store i Web API
                            _base_http.py        # Bazowa klasa HTTP z asynchronicznym
    klientem
                            __init__.py          # Definiuje 'services' jako pakiet
                      __init__.py              # Definiuje 'core' jako pakiet
                ui/
                      home_view.py             # Widok główny/startowy aplikacji
                      __init__.py              # Definiuje 'ui' jako pakiet
                main.py                      # Główny punkt wejścia
                main_window.py               # Główne okno aplikacji
                __init__.py                  # Definiuje 'app' jako pakiet (może być
    opcjonalny)
          .env.example
          .gitignore
          LICENSE
          pyproject.toml
          requirements.txt
```

Listing 1: Struktura Katalogów

3 Szczegółowa Analiza Plików

3.1 `main.py` (Punkt Wejścia)

Plik odpowiedzialny za inicjalizację aplikacji Qt i zintegrowanie jej z pętlą zdarzeń **asyncio** za pomocą biblioteki **qasync**.

```python
import sys
import asyncio
from PySide6.QtWidgets import QApplication
from main_window import MainWindow
from qasync import QEventLoop, run
# ...

async def main_coro(app, window):
    """Główna korutyna, która uruchamia i utrzymuje okno."""
    window.show()
    future = asyncio.Future()

    def on_quit():
        # Ustawienie wyniku Future (kończy await future)
        if not future.done():
            future.set_result(True)

    # KLUCZOWE: Łączymy sygnał o zamykaniu aplikacji (QApplication) z naszą funkcją
    app.aboutToQuit.connect(on_quit)

    # Czekamy, aż future zostanie zakończone (czyli aplikacja się zamknie)
    try:
        await future
    except asyncio.CancelledError:
        pass
```

```
26
27      return None
28
29  def main():
30      # 2. Utworzenie i ustawienie pętli zdarzeń QEventLoop
31      app = QApplication(sys.argv)
32      loop = QEventLoop(app)
33      asyncio.set_event_loop(loop)
34      # ...
35      # 4. Użycie qasync.run() do uruchomienia głównej korutyny
36      sys.exit(run(main_coro(app, window)))
37  # ...
```

Listing 2: main.py (Kluczowe fragmenty)

- `main_coro`: Jest to główna korutyna, która zarządza cyklem życia okna. Wykorzystuje `asyncio.Future` połączone z sygnałem `app.aboutToQuit` do utrzymania asynchronicznej pętli zdarzeń do momentu zamknięcia okna GUI.
- `QEventLoop(app)`: Jest to kluczowy element integracji. Zastępuje standardową pętlę zdarzeń Pythona, pozwalając na jednoczesne zarządzanie zdarzeniami Qt (GUI) i operacjami `asyncio` (I/O sieciowe).
- `qasync.run()`: Uruchamia asynchroniczną pętlę zdarzeń i czeka na zakończenie korutyny `main_coro`, co jest mechanizmem startowym aplikacji.

3.2 `main_window.py` (Główne Okno)

Definiuje główne okno aplikacji dziedziczące po `QMainWindow`. Zarządza podstawowym układem, paskiem narzędzi (`QToolBar`) i widokami (za pomocą `QStackedWidget`).

```
1   from PySide6.QtWidgets import QMainWindow, QWidget, QVBoxLayout, QStackedWidget,
        QToolBar
2   # ...
3   from ui.home_view import HomeView
4
5   class MainWindow(QMainWindow):
6       def __init__(self):
7           super().__init__()
8           self.setWindowTitle("Steam Dashboard")
9           self.setMinimumSize(1000, 800)
10          # ...
11          self.stack = QStackedWidget() # Umożliwia przełączanie widoków
12          self.home_view = HomeView()
13          self.stack.addWidget(self.home_view)
14          self._init_toolbar()
15
16      def _init_toolbar(self):
17          toolbar = QToolBar("Menu")
18          self.addToolBar(toolbar)
19
20          home_action = QAction("Home", self)
21          home_action.triggered.connect(lambda: self.stack.setCurrentWidget(self.home_view
                ))
22          toolbar.addAction(home_action)
23
24          refresh_action = QAction("Odśwież", self)
25          refresh_action.triggered.connect(self.refresh_current_view)
26          toolbar.addAction(refresh_action)
27
28      def refresh_current_view(self):
29          widget = self.stack.currentWidget()
30          # Wywołuje metodę refresh() na aktywnym widoku
31          if hasattr(widget, "refresh"):
32              widget.refresh()
```

Listing 3: main_window.py (Kluczowe fragmenty)

- `QStackedWidget`: Pozwala na efektywne zarządzanie wieloma widokami, wyświetlając tylko jeden na raz (obecnie `HomeView`).

- `QToolBar`: Implementuje pasek narzędzi z akcjami do nawigacji (`Home`) i odświeżania.
- `refresh_current_view`: Implementuje uniwersalny mechanizm odświeżania, sprawdzając, czy aktualnie wyświetlany widżet posiada metodę `refresh()`, i ją wywołuje. Jest to standardowy wzorzec w aplikacjach wielowidokowych.

3.3 `ui/home_view.py` (Widok Główny)

Główny widok użytkownika, który asynchronicznie pobiera i wyświetla dane z API Steam oraz CheapShark w trzech sekcjach: aktywni gracze, najlepsze promocje i nadchodzące gry.

```python
import asyncio
from PySide6.QtWidgets import QWidget, QVBoxLayout, QLabel, QListWidget, QFrame
from PySide6.QtCore import QTimer, Qt
from core.services.steam_api import SteamStoreClient

# ... TOP_GAME_APP_IDS (lista gier do sprawdzenia liczby graczy)

class HomeView(QWidget):
    def __init__(self):
        super().__init__()
        # ... Inicjalizacja trzech sekcji (QListWidget, QLabel, QFrame)

        # Inicjalizacja: Odśwież dane po załadowaniu GUI
        QTimer.singleShot(100, self.refresh)

    def refresh(self):
        """Odświeża dane we wszystkich sekcjach asynchronicznie."""
        # ... Czyści listy i dodaje komunikat "Ładowanie danych..."

        # KLUCZOWE: Uruchomienie asynchronicznej operacji w tle
        asyncio.create_task(self._load_data_async())

    async def _load_data_async(self):
        """Asynchroniczne pobieranie danych z API."""
        try:
            async with SteamStoreClient(timeout=15.0) as client:

                # 1. LIVE GAMES COUNT (Wywołuje osobną korutynę)
                await self._load_top_live_games(client)

                # 2. BEST DEALS (Pobiera z 'specials')
                best_deals = await client._get_featured_category_items("specials", cc="
                    pl", lang="pl", limit=15)
                # ... Wyświetlanie promocji z ceną i procentem zniżki

                # 3. BEST UPCOMING RELEASES
                top = await client.get_coming_soon(limit=15)
                # ... Wyświetlanie nazw nadchodzących gier

        except Exception as e:
            # Centralna obsługa błędów API
            # ... Wyświetlenie błędu we wszystkich listach

    async def _load_top_live_games(self, client: SteamStoreClient):
        """Pobiera dane o liczbie graczy dla predefiniowanych gier i nazwach."""

        # Jednoczesne pobranie statystyk i nazw
        player_tasks = [client.get_number_of_current_players(appid) for appid in
            TOP_GAME_APP_IDS]
        detail_tasks = [client.get_app_details(appid, cc="pl", lang="pl") for appid in
            TOP_GAME_APP_IDS]

        player_counts = await asyncio.gather(*player_tasks, return_exceptions=True)
        game_details = await asyncio.gather(*detail_tasks, return_exceptions=True)
        # ... Łączenie, sortowanie i formatowanie wyników
```

Listing 4: ui/home_view.py (Kluczowe fragmenty)

- `asyncio.create_task()`: Zapewnia, że operacje pobierania danych są uruchamiane **współbieżnie w tle** w pętli zdarzeń, dzięki czemu GUI pozostaje responsywne.

- `async with SteamStoreClient`: Użycie menedżera kontekstu gwarantuje bezpieczne zarządzanie zasobami sieciowymi.
- `asyncio.gather()`: Użyte w `_load_top_live_games` do drastycznego skrócenia czasu ładowania. Wszystkie żądania statystyk i detali dla 15 gier są wysyłane **równolegle**.
- `QListWidget`: Jest głównym widżetem do wyświetlania dynamicznie ładowanych list informacji.

3.4 `core/services/_base_http.py` (Bazowy Klient HTTP)

Zawiera klasę bazową dla wszystkich klientów API. Wykorzystuje **httpx** do asynchronicznych żądań HTTP i **tenacity** do automatycznego ponawiania połączeń w przypadku błędów.

```python
import httpx
from tenacity import retry, stop_after_attempt, wait_exponential,
    retry_if_exception_type
# ...

class BaseAsyncService:
    """Współdzielony asynchroniczny klient HTTP z obsługą ponawiania i menedżera
        kontekstu."""

    def __init__(self, *, timeout: httpx.Timeout | float = _DEFAULT_TIMEOUT):
        self._client = httpx.AsyncClient(http2=True, timeout=timeout)

    async def __aenter__(self):
        return self

    async def __aexit__(self, exc_type, exc, tb) -> None:
        await self.aclose() # Automatyczne zamykanie klienta httpx

    @retry(
        reraise=True,
        stop=stop_after_attempt(3), # Maksymalnie 3 próby
        wait=wait_exponential(multiplier=0.5, min=0.5, max=4), # Czas oczekiwania rośnie
            wykładniczo
        retry=retry_if_exception_type(httpx.HTTPError), # Ponawia tylko błędy HTTP
    )
    async def _get_json(self, url: str, *, params: Optional[Dict[str, Any]] = None,
        headers: Optional[Dict[str, str]] = None) -> Any:
        # ...
        resp = await self._client.get(url, params=safe_params, headers=headers)
        resp.raise_for_status() # Wyrzuca błąd dla kodów 4xx/5xx
        return resp.json()
```

Listing 5: core/services/_base_http.py (Kluczowe elementy)

- `httpx.AsyncClient`: Wybrany klient HTTP zapewniający natywną obsługę `async/await`.
- `__aenter__`/`__aexit__`: Implementuje asynchroniczny menedżer kontekstu, umożliwiający bezpieczne użycie `async with`.
- `@retry`: Dekorator z biblioteki `tenacity` zapewnia automatyczną, niezawodną obsługę błędów sieciowych i serwerowych poprzez ponawianie żądań.

3.5 `core/services/steam_api.py` (Klient Steam API)

Implementuje interfejsy do pobierania danych ze Steam Store i Steam Web API. Wykorzystuje **Pydantic** do modelowania otrzymanych danych.

```python
from __future__ import annotations
# ...
from pydantic import BaseModel, ConfigDict, Field
from ._base_http import BaseAsyncService # POPRAWIONY IMPORT WZGLĘDNY
# ... Definicje modeli Pydantic (PlayerCount, AppDetails, FeaturedItem, itp.)
# ... Definicje interfejsów ISteamStore, ISteamWebApi

class SteamStoreClient(BaseAsyncService, ISteamStore):
    # ...
```

```python
    async def get_number_of_current_players(self, appid: int) -> PlayerCount:
        url = "https://api.steampowered.com/ISteamUserStats/GetNumberOfCurrentPlayers/v1/"
        # ... Zwraca PlayerCount

    async def get_app_details(self, appid: int, cc: str = "us", lang: str = "en") -> Optional[AppDetails]:
        url = "https://store.steampowered.com/api/appdetails"
        # ... Pobiera szczegóły gier (w tym nazwy i ceny)

    async def _get_featured_category_items(self, key: str, *, cc: str, lang: str, limit: int) -> List[FeaturedItem]:
        # Wewnętrzna metoda do pobierania list gier (promocje, nowości, nadchodzące)
        # ...

class SteamWebApiClient(BaseAsyncService, ISteamWebApi):
    # Klient wymagający klucza API (do danych prywatnych, np. owned_games)
    # ...
```

Listing 6: core/services/steam_api.py (Kluczowe fragmenty)

- **Pydantic**: Używany do definiowania ścisłych struktur danych (np. `PlayerCount`, `AppDetails`, `FeaturedItem`). Zapewnia to walidację i bezpieczeństwo danych.
- **Poprawka Importu**: Kluczową zmianą, która naprawiła błąd kompilacji, było użycie **importu względnego**: `from ._base_http import BaseAsyncService`.
- `ISteamStore`: Interfejs do publicznie dostępnych danych sklepu (wykorzystywany w `HomeView`).

3.6 `core/services/deals_api.py` (Klient CheapShark API)

Implementuje interfejs `IDealsApi` do pobierania danych o promocjach gier z CheapShark (API, które gromadzi najlepsze zniżki).

```python
from ._base_http import BaseAsyncService # Poprawny import względny
# ... Definicja modelu Pydantic Deal
# ... Definicja interfejsu IDealsApi

class DealsApiClient(BaseAsyncService, IDealsApi):
    BASE_URL = "https://www.cheapshark.com/api/1.0"

    async def get_current_deals(self, limit: int = 50, min_discount: int = 30) -> List[Deal]:
        url = f"{self.BASE_URL}/deals"
        params = {
            "onSale": 1,
            "pageSize": max(1, min(500, int(limit))),
            "sortBy": "Savings", # Sortowanie po największej oszczędności
        }
        data = await self._get_json(url, params=params)
        # ... Parsowanie i walidacja danych do listy Deal
        return deals
    # ... Inne metody API Deals
```

Listing 7: core/services/deals_api.py (Kluczowe fragmenty)

- `DealsApiClient`: Dziedziczy z `BaseAsyncService`, wykorzystując mechanizmy ponawiania i kontekstowego zamykania połączenia.
- `get_current_deals`: Pobiera najbardziej aktualne promocje, sortując je domyślnie według oszczędności (`sortBy: "Savings"`) i filtrując minimalny procent zniżki.
- `Deal`: Model Pydantic dostosowuje niestandardowe nazwy pól z CheapShark (np. `steamAppID`, `salePrice`) do standardowej struktury danych.