Raport Testów Wydajności
Aplikacja Metri

15 grudnia 2025

Spis treści

1 Zakres i uruchamianie

Niniejszy raport dokumentuje wyniki testów wydajności kluczowych fragmentów logiki aplikacji Metri. Celem benchmarków jest ocena czasu wykonania operacji, które mogą mieć istotny wpływ na odczuwalną szybkość działania aplikacji przy pracy na większych zbiorach danych.

Zakres obejmuje następujące obszary (bez warstwy GUI i bez operacji dyskowych):

- **Renderowanie piosenek** – generowanie widoków tekstowych z wewnętrznym formatowaniem (np. tagi `<b>`, `<i>`, `<code>`) na potrzeby podglądu utworów (moduł `display_func`).
- **Parsowanie treści** – konwersja surowych tekstów piosenek do ustrukturyzowanego formatu JSON (moduł `jsonify_func`).
- **Operacje na songbooku** – filtrowanie, wyszukiwanie i agregacja metadanych dla dużej liczby piosenek (moduł `song_func`).

Benchmarki uruchamiane są za pomocą `pytest-benchmark` i oznaczone markerem `perf`, dzięki czemu domyślnie nie są wykonywane podczas zwykłego `pytest`. Aby wykonać wyłącznie testy wydajności, należy użyć polecenia:

```
.venv\Scripts\python.exe -m pytest tests/test_perf.py -m perf --benchmark-only
```

2 Metodologia

- Narzędzie: `pytest-benchmark` (domyślne ustawienia: min_rounds=5, max_time=1s, timer=`time.perf_counter`).
- Dane syntetyczne wstrzykiwane przez `monkeypatch` (brak I/O plików, brak GUI), aby mierzyć samą logikę.
- Każdy benchmark wykonuje wiele iteracji (`Rounds`) – wyniki raportowane jako średni czas oraz OPS (operacje/s).
- Środowisko: Windows, Python 3.13.2, wirtualne środowisko projektu.

3 Opis scenariuszy

Poniższe benchmarki odzwierciedlają typowe i skrajne przypadki użycia aplikacji.

- **test_display_html_many_songs** – generowanie sformatowanego tekstu (z tagami w stylu HTML, np. `<b>`, `<i>`, `<code>`) dla 200 piosenek. Scenariusz odwzorowuje sytuację, w której użytkownik intensywnie przegląda utwory, a aplikacja musi wielokrotnie budować złożony widok w oknie desktopowym.
- **test_display_plaintext_many_songs** – generowanie prostego, dwuwierszowego widoku tekstowego (tekst + akord) dla 200 piosenek. Mierzy czas działania uproszczonej ścieżki bez HTML, wykorzystywanej np. do eksportu lub prostych podglądów.

- **test_jsonify_auto_bulk** – parsowanie 300 surowych bloków tekstu (sekcje, akordy, liryki) do struktury JSON. Scenariusz odpowiada hurtowemu importowi lub konwersji większej liczby piosenek z formatu tekstowego do wewnętrznego modelu aplikacji.
- **test_add_very_long_song_jsonify** – parsowanie pojedynczej, bardzo długiej piosenki (setki linii tekstu). Odzwierciedla sytuację, w której użytkownik dodaje rozbudowany utwór, a aplikacja musi zachować płynność działania.
- **test_add_dense_chords_song_jsonify** – parsowanie długiej piosenki z gęsto rozmieszczonymi i złożonymi akordami (np. `Asus4/D`, `Gmbadd9/F`, `Bbmaj7`). Scenariusz obciąża przede wszystkim regexy i logikę wykrywania akordów, reprezentując „najtrudniejsze" przypadki harmoniczne.
- **test_filter_songs_large_dataset** – filtrowanie i sortowanie 1000 piosenek po tytule, języku i tagach. Jest to przybliżenie pracy z większym, ale nadal typowym songbookiem użytkownika.
- **test_get_tags_large_dataset** – agregacja i deduplikacja tagów z 1000 piosenek. Test mierzy koszt budowania list pomocniczych (np. do filtrów po tagach) dla średniej wielkości biblioteki.
- **test_filter_songs_huge_dataset** – to samo zapytanie filtrujące wykonane na 10 000 piosenek. Scenariusz sprawdza, jak rośnie czas odpowiedzi wraz z rozmiarem kolekcji i czy algorytmy filtrowania skalują się liniowo.
- **test_list_many_songs_mapping** – przekształcenie listy 10 000 piosenek do uproszczonej struktury (id, tytuł, wykonawca, język, tagi), która mogłaby zostać przekazana do warstwy UI. Test mierzy koszt samego przygotowania danych, bez rysowania widoku.

4 Wyniki (15.12.2025, Windows, Python 3.13.2)

Benchmark	Średni czas	OPS (1/średni)	Rounds
oprule extbfBenchmark	**Średni czas**	**OPS (1/średni)**	**Rounds**
exttttest_get_tags_large_dataset	∼ 0.218 ms	∼ 4585	4720
exttttest_filter_songs_large_dataset	∼ 0.387 ms	∼ 2582	1421
exttttest_display_plaintext_many_songs	∼ 0.570 ms	∼ 1754	1746
exttttest_add_dense_chords_song_jsonify	∼ 1.056 ms	∼ 947	824
exttttest_add_very_long_song_jsonify	∼ 1.092 ms	∼ 915	866
exttttest_display_html_many_songs	∼ 3.921 ms	∼ 255	251
exttttest_filter_songs_huge_dataset	∼ 4.100 ms	∼ 244	259
exttttest_list_many_songs_mapping	∼ 4.259 ms	∼ 235	177
exttttest_jsonify_auto_bulk	∼ 8.266 ms	∼ 121	101

Tabela 1: Podsumowanie benchmarków wydajności (dane syntetyczne, wstrzykiwane przez `monkeypatch`)

5 Uwagi

- Dane testowe są syntetyczne; celem jest porównanie względne między wersjami (regresje/przyspieszenia).
- Benchmarki są oznaczone markerem `perf`; bez opcji `-benchmark-only` zostaną pominięte.
- Wymagany pakiet: `pytest-benchmark` (dodany do `requirements.txt`).
- Jeśli potrzebujesz porównań między gałęziami/wersjami, użyj wbudowanej komendy: `pytest-benchmark compare <plik1> <plik2>`.
- Dalsze rozszerzenia: dodać scenariusz pełnego startu aplikacji (czas inicjalizacji GUI) oraz prosty test opóźnienia MIDI po mocku `pygame`.

6 Wnioski

- Czysta logika jest bardzo lekka: operacje na 1000 piosenek (tagi/filtrowanie) mieszczą się w ułamkach milisekundy na wywołanie.
- Render tekstowy/HTML dla 200 piosenek zamyka się w pojedynczych milisekundach na piosenkę; to nie wygląda na wąskie gardło przy typowym użyciu.
- Parsowanie surowych bloków (`jsonify_func`) jest wolniejsze od renderu, ale nadal $< 10\,\text{ms}$ na blok — akceptowalne dla wsadowych operacji.
- Realne czasy w aplikacji wzrosną o narzut I/O i GUI; jeśli zauważysz lagi, zmierz konkretny scenariusz końcowy (z plikami/oknem) i profiluj, zamiast optymalizować w ciemno.
- Progi orientacyjne: render HTML $\leq 5\,\text{ms}$/piosenka, parsowanie $\leq 10\,\text{ms}$/blok; przekroczenie tych wartości w realnych scenariuszach to sygnał do profilowania.