Seminarium 9: Cache

Wstęp

Wszystkie zadania wykorzystują to repozytorium.

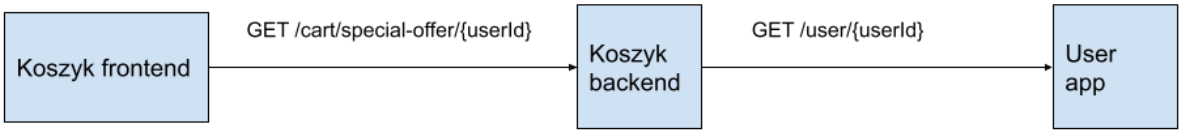
Zadanie 1

Przygotowanie techniczne zadania:

1. Uruchomić infrastrukturę z repozytorium wg instrukcji na branchu zad1.
2. Uruchomić testy JMetera, który wygeneruje nam ruch.
3. Poczekać 5 minut na zebranie metryk.

Mamy sklep internetowy, na którym w koszyku chcemy wyświetlić informacje o specjalnej ofercie - zniżce dla naszych lojalnych klientów. Zniżka ma być zależna od wewnętrznego systemu ocen klienta (0-100). Im lepsza ocena tym większa zniżka.

Rozwiązanie wygląda następująco.



Po wdrożeniu pojawiły się problemy wydajnościowe z usługą *user-app*. Generujemy duży ruch z nowej funkcjonalności. Ponadto z powodu zwiększonego ruchu *user-app* nie wyrabia się w zadanym SLA.

By uniknąć takiego problemu postanowiono zaimplementować cache do klienta *user-app.* Do implementacji wykorzystano Caffeine cache. Po zapisie do cache, każdy z elementów jest przechowywany przez maksymalnie 30s.

Pytania:

1. Ile elementów przechowuje cache w pamięci?
2. Ile requestów jest mniej do serwisu *user-app* po implementacji cache? Ile requestów wysyłamy do *user-app*?
3. O ile szybciej odpowiadamy na zapytania bez cache vs z cache?
4. Jaka jest efektywność wykorzystania cache?

Zadanie 2

Zwiększony ruch

By przygotować się do zadania:

1. Przejdź do brancha zad2a
2. Zatrzymaj testy jmeter.
3. Uruchom testy jmeter z tego brancha.

W związku ze zbliżającym się grudniem ruch na naszym sklepie internetowym zwiększa się dwukrotnie.

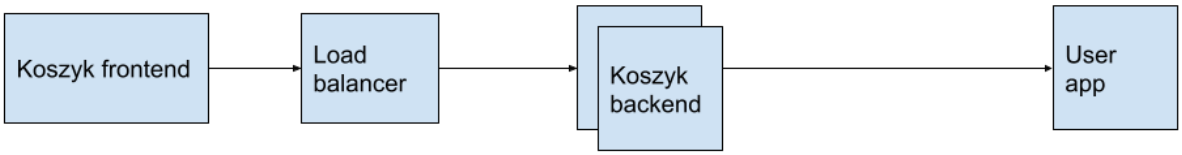
1. Jak zmieniła się efektywność cache?
2. Jak można podnieść wydajność cache? Jakie widzisz wady takiego rozwiązania.
3. Jak zarządzamy pamięcią cache? Jakie znacie algorytmy wybierania elementu do usunięcia z cache? Jakie mają wady i zalety takie rozwiązania?
4. (Bonus) Jak caffeine cache wybiera element do usunięcia z cache?

Skalowanie instancji

By przygotować się do zadania:

1. Przejdź do brancha zad2b
2. Zatrzymaj testy jmetera
3. Zatrzymaj instancje (ctrl + c lub docker compose down)
4. Uruchom instancje ponownie *docker compose up -d*
5. Uruchom ponownie testy jmetera.
6. Poczekaj kilka minut na zebranie statystyk.

By lepiej obsłużyć takie obciążenie skalujemy się instancjami koszyka do 2. Ruch pomiędzy instancjami rozkłada Load balancer za pomocą algorytmu round robin.



Pytania

1. Jak zmieniły się metryki cache? Czy cache pracuje tak samo efektywnie jak przy jednej instancji? Dlaczego?
2. Jak można poprawić efektywność cache?

Load balancer: sticky sessions

By przygotować się do zadania:

1. Przejdź do brancha zad2c
2. Zatrzymaj testy jmetera
3. Zatrzymaj instancje (ctrl + c lub docker compose down)
4. Uruchom instancje ponownie *docker compose up -d*
5. Uruchom ponownie testy jmetera.
6. Poczekaj kilka minut na zebranie statystyk.

By poprawić efektywność działania cache wprowadzano inny algorytm load balancera: hashowanie po URL.

1. Jak zmieniła się efektywność cache po wprowadzonych zmianach?
2. Jakie wady ma takie rozwiązanie?

Zadanie 3

By przygotować się do zadania:

1. Przejdź do brancha zad3
2. Zatrzymaj testy jmetera
3. Zatrzymaj instancje (ctrl + c lub docker compose down)
4. Uruchom instancje ponownie *docker compose up -d*
5. Uruchom ponownie testy jmetera
6. Poczekaj kilka minut na zebranie statystyk.

Po wprowadzeniu nowego algorytmu load balancera zaobserwowano nierówne obciążenie ruchu między instancjami. W związku z powyższym algorytm LB został zmieniony na round robin. Aby zachować podobną efektywność cache postanowiono użyć jeden współdzielony cache dla obu instancji. Jego implementacją będzie redis.

Redis jest skalowalną bazą/cache in-memory. Z naszej perspektywy ważne jest, że jest zoptymalizowany do szybkich operacji pobierania i zapisywania danego klucza. Innym ważną funkcją jest możliwość ustawienia *expiry date* na danym kluczu. Rozmiarem takiego cache sterujemy poprzez ustawienie rozmiaru pamięci jaką chcemy przeznaczyć na cache.

1. Porównaj efektywność współdzielonego cache?
2. Jakie to rozwiązanie ma wady i zalety?
3. Rozważ jakie zyski może przynieść wprowadzenie dwu poziomowego cache (L1: Caffeine cache i L2: redis).

4. (Bonus) Jakie algorytmy usuwania elementów z Redis'a znasz? Jakie mają wady i zalety?

Do dalszej lektury

- Artykuł twórcy Caffeine Cache na temat Window TinyLFU (algorytm usuwania elementów z cache). Dużo też można się dowiedzieć o innych algorytmach stosowanych w cachowaniu i problemach jakie napotykamy podczas implementacji. Po przeczytaniu nigdy nie spojrzysz na cache jak na zwykłą HashMapę.
- Dokumentacja redis na temat polityk usuwania elementów z cache. Tutaj już widać jak proste algorytmy typu LFU (Least frequently used) i LRU (Least recently used) zaczynają się komplikować przy implementacji rozproszonego cache'a.
- Tutaj o algorytmie usuwania nieaktualnych elementów z memcached. Jest to alternatywa dla redis'a. Oferuje proste struktury danych i operacje na nich, ale w przeciwieństwie do Redisa działa wielowątkowo.
- Opis consitent caching w memcached dzięki, któremu możemy skalować się na wiele node'ów po stronie klienta memcached