Jak cacheować aplikacje w Django

Michał Przyłucki

November 2024

1 Abstract

Projektowanie nowoczesnych aplikacji stawia przed deweloperami wiele wyzwań. Jednym z nich jest problem skalowalności aplikacji. Projektując produkt, już na pierwszych etapach powinniśmy wziąć pod uwagę możliwość wystąpieniu wielu zapytań w jednostce czasu. Należy przygotować na to serwer oraz aplikację.

Jednym ze sposobów radzenia sobie z problemem skalowalności, czy ogólnie problemem obciążenia, jest używanie mechanizmu cache'owania.

W przypadku aplikacji opartych na REST API często zdarza się, że przy każdym odświeżeniu strony backend wykonuje identyczne zapytania do bazy danych, aby pobrać te same informacje, które następnie przekazuje frontendowi. Znacznie bardziej optymalnym rozwiązaniem jest przechowywanie często wykorzystywanych danych w pamięci podręcznej (np. Redis) lub w specjalnie przygotowanej bazie pośredniej. Pozwala to ograniczyć liczbę bezpośrednich zapytań do głównej bazy danych, zwiększyć wydajność aplikacji, skrócić czas odpowiedzi serwera i tym samym poprawić doświadczenie użytkownika.

Omówiony przykład mechanizmu cache'owania zostanie przedstawiony na podstawie aplikacji Django oraz Django Rest Framework, który pomaga w tworzeniu aplikacji opartej o technologię Rest API.

2 Czym jest redis?

Logo redis

Redis to niezwykle szybka, open-source'owa baza danych, która przetrzymuje pary klucz-wartość i która działa w pamięci operacyjnej RAM. Z tych powodów, jest idealny do pracy jako pamięć podręczna cache, ale ma też wiele innych ciekawych zastosowań.

Niektóre z nich to:

- kolejka zadań
- licznik
- model publikacji i subskrypcji MQTT - system publikowania i subskrybowania wiadomości, np. do powiadomień w czasie rzeczywistym.
- utrzymywanie sesji użytkownika
- lekka baza NoSQL

3 Co to jest cache'owanie?

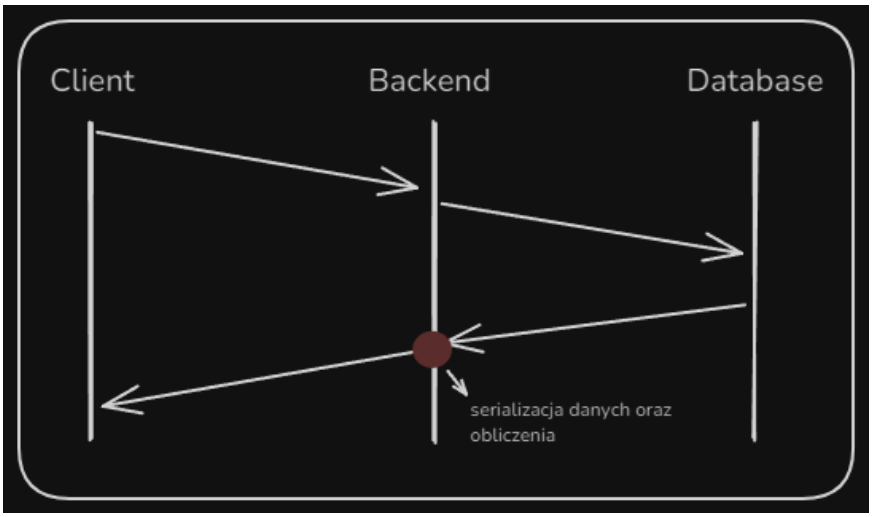
Cache'owanie to proces polegający na zapisywaniu danych, efektów obliczeń, informacji oraz wszystkiego tego co może zostać użyte ponownie w dedykowanej bazie danych. Przy projektowaniu rozwiązań webowych, zazwyczaj stosowane są takie technologie jak Redis, memcached czy MongoDB.

Cache to pamięć podręczna, czyli taka, do której aplikacja posiada bardzo szybki dostęp. Zapisywane są w niej rzeczy, które już zostały obliczone, pozyskane, i których później można użyć ponownie bez ponownego obliczenia i pobierania, co czasami bywa kosztowne.

Należy jednak pamiętać, że korzystając z mechanizmu cache'owania musimy zapewnić też mechanizmy inwalidacji cache'owania. Bez tego użytkownicy mogą otrzymywać nieaktualne dane.

Standardowy schemat wymiany informacji pomiędzy klientem, backendem, a bazą danych:

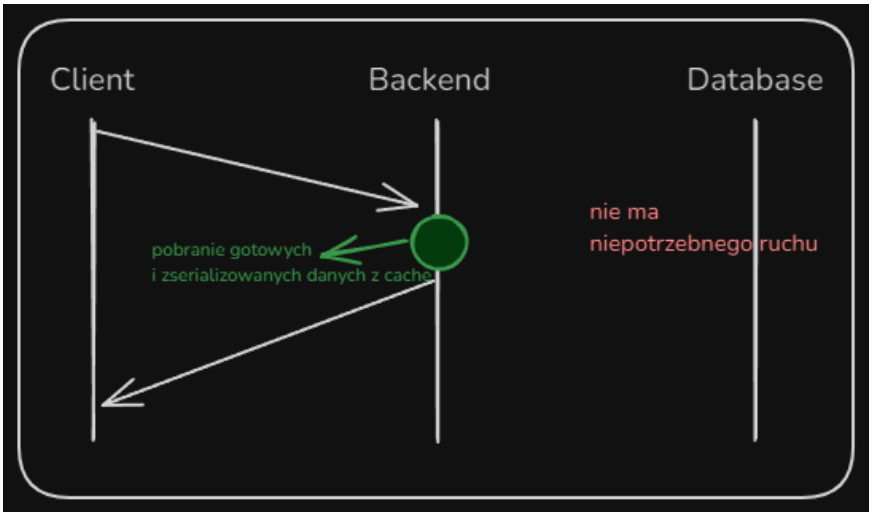
abc



Standardowy schemat wymiany danych

Przy implementacji mechanizmów cache'owania ograniczamy ilość zapytań do bazy danych oraz powtarzalne skomplikowane obliczenia, np. wynikające z logiki biznesowej.

Schemat wymiany informacji z użyciem mechanizmu cache'owania:



Użycie mechanizmu cache'owanie

4 Kiedy cache'ować?

Cache'ować można praktycznie wszystko, tylko, że cache'owanie wszystkiego nie zawsze ma sens. W przypadku aplikacji z Restful API można cache'ować zserializowane dane, całe treści odpowiedzi HTTP, nawet całe widoki. Ważne, żeby zrozumieć, że cache'owanie ma szczególny sens w przypadku danych, które rzadko ulegają zmianie (lub kontrolujemy zmianiające się dane przez mechanizm inwalidacji cache).

Idealny przykład cacheowania: Mamy tabelę "Awarie", która jest zasilana danymi codziennie o północy. Jest to tabela stricte dla administratorów, nie musimy zgłębiać teraz jej pochodzenia.

Dostęp do danych w tej tabeli znajduje się w zakładce "Maintenance" w naszej aplikacji. Po wejściu do tej zakładki wyświetlane są wszystkie zarejestrowane awarie.

To idealne miejsce do implementacji przechowywania informacji w pamięci podręcznej cache. Dzięki temu, użytkownicy odwiedzający zakładkę "Maintenance" nie muszą odpytywać bazy danych przy każdym odświeżeniu strony. Są one pobierane z pamięci cache. Co noc po zasileniu tabeli danymi, pamięć cache jest czyszczona (przynajmniej ten fragment z awariami), w celu uaktualnienia. Dzięki temu już drugie zapytanie po dane do zakładki "Maintenance" będzie pobierane z cache, zamiast tworzyć zbędne zapytania do bazy danych.

To oczywiście przerysowany przykład, ale pozwala zrozumieć filozofię cache'owania, zwłaszcza w kontekście aplikacji webowych.

Cache'owanie danych, które często ulegają zmianie mija się z celem, chyba, że zastosujemy dodatkowe mechanizmy, np. inwalidacji pamięci podręcznej. Ten artykuł opisuje właśnie takie zastosowanie cache, które nie wymaga, aby dane, których dotyczy cache'owanie nie zmianiały się zbyt

często.

Najpierw omówmy jednak proces cache'owania w aplikacjach wykorzystujących framework Django.

5 Cache'owanie w Django

W przykładzie została użyta baza danych redis.

Pierwsze co będzie potrzebne to sama baza danych. W zależności od infrastruktury instalacja takiej bazy danych może się różnić. W przykładzie zostanie użyty kontener z obrazem redisa.

Do pliku docker-compose.yaml została dodana usługa redis:

```
redis:
        image: redis:7
        container_name: redis_cache
        restart: always
```

Cały gotowy plik docker-compose.yaml:

```
services:
  db:
    image: postgres:13
    volumes:
      - ./data/db:/var/lib/postgresql/data
    environment:
      - POSTGRES_DB=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
  backend:
    build: ./backend
    command: bash -c "python manage.py runserver 0.0.0.0:8000"
    volumes:
      - ./backend:/app
    ports:
      - "8000:8000"
    environment:
      - POSTGRES_NAME=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
    depends_on:
```

```
      - db
  redis:
    image: redis:7
    container_name: redis_cache
    restart: always
```

Następnie należy poinformować aplikację Django, że cache'owanie będzie opierało się o bazę danych redisa. Aby to zrobić należy zainstalować moduł 'django-redis'.

```
        pip install django-redis
```

Do pliku 'settings.py' (głównego pliku konfiguracyjnego aplikacji) należy również dodać następującą treść:

```
CACHES = {
        "default": {
                "BACKEND": "django_redis.cache.RedisCache",
                "LOCATION": "redis://redis:6379/0",
                "OPTIONS": {
                        "CLIENT_CLASS": "django_redis.client.
                           DefaultClient",
                }
        }
}
```

6 Manualne cache'owanie

W celu zobrazowania metody cache'owania, został zaimplementowany następujący widok.

```
from rest_framework import generics
from rest_framework.response import Response
from main.models import Person
from main.serializers import PersonSerializer
from django.core.cache import cache


class PersonAPI(generics.ListAPIView, generics.CreateAPIView
    ):
        queryset = Person.objects
        serializer_class = PersonSerializer

        def get(self, request):
                qs = Person.objects.all()
                data_from_cache = cache.get('
                    person_view_data')
                if data_from_cache:
                        return Response(data_from_cache)
                else:
                        serializer = self.serializer_class(
                            qs, many=True)
                        cache.set("person_view_data",
                            serializer.data)
                        return Response(serializer.data)
```

6.1 Wyjaśnienie

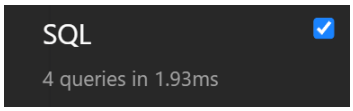
Dane próbują być pobrane z pamięci podręcznej cache. Jeśli ta operacja zostanie zakończona sukcesem, pobrane dane zostają zwrócone w odpowiedzi HTTP. Jeśli to się nie udaje - czyli dane nie istnieją w cache - zostaje wykonana normalna operacja serializacji. Jednocześnie zserializowane dane zostają zapisane w redisie, co przyda się przy wywołaniu zapytania po raz kolejny.

6.2 Django-debug-toolbar

Dodatkowo zostało użyte narzędzie ‘django-debug-toolbar‘. Pozwoli ono na sprawdzenie jak wiele zapytań SQL zostało wykonanych do bazy danych.

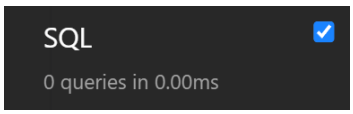
Link do narzędzia django-debug-toolbar: https://django-debug-toolbar.readthedocs.io/en/latest/

Przy pierwszym uruchomieniu zapytania:



Widoczne zapytanie do bazy danych

Przy każdym kolejnym:



Brak zapytania do bazy danych

Jak widać na screenach, mechanizm cache’owania działa poprawnie. Pierwsze wywołanie widoku generuje zapytania do bazy danych, a dane zostają zapisane w redisie. Każde kolejne wywołanie widoku pobiera dane bezpośrednio z pamięci podręcznej cache co nie wymaga wykonywania zapytań do głównej bazy danych.

Oczywiście nie ma potrzeby ręcznej implementacji pobierania oraz ustawiania danych w pamięci podręcznej, tak jak to zostało zrobione w przedstawionym widoku. Z pomocą przychodzą nam gotowe klasy i funkcje w postaci dekoratorów. W przypadku aplikacji używających ‘rest-framework‘ cała dokumentacja znajduje się tutaj: https://www.django-rest-framework.org/api-guide/caching/.

Bardzo podobny efekt uzyskamy używając dekoratora @method_decorator(cache_page(60 * 60 * 2))

```
        from rest_framework import generics
        from main.models import Person
        from main.serializers import PersonSerializer
        from django.utils.decorators import method_decorator
        from django.views.decorators.cache import cache_page


        class PersonAPI(generics.ListAPIView, generics.CreateAPIView
            ):
                queryset = Person.objects
```

```
        serializer_class = PersonSerializer

        @method_decorator(cache_page(60 * 60 * 2))
        def get(self, request):
                return super().get(request)
```

Co jednak jeśli dane ulegną zmianie? Prowadzi to do dużego problemu, ponieważ user końcowy może dostać nieaktualne dane. Tutaj z pomocą przychodzi nam mechanizm inwalidacji cache.

7 Inwalidacja cache

Dane przetrzymywane w pamięci podręcznej cache muszą zostać usunięte w odpowiednim momencie. W innym przypadku userzy dostawaliby ciągle te same dane, bez względu na zmiany. Zazwyczaj cacheowanie jest ustawiane na konkretny interwał czasowy, na przykład 5 minut. Dzięki temu user, który długo wykonuje jakieś operacje w naszej aplikacji nie musi za każdym razem pobierać danych od nowa.

Pomysł przedstawiony w tym artykule jest nieco inny. Sugeruje on używanie mechanizmu cache bez limitu czasowego, natomiast wymusza bardzo rygorystyczne zasady stosowania mechanizmu inwalidacji cache. Kluczową rolę w tym procesie odgrywają sygnały Django, takie jak post_save i post_delete, które odpowiadają za aktualizację pamięci podręcznej po modyfikacji lub usunięciu instancji modelu.

```
from django.db.models.signals import post_save, post_delete
from django.dispatch import receiver
from django.core.cache import cache
from main.models import Person


@receiver(post_save, sender=Person)
def invalidate_cache_on_save(sender, instance, **kwargs):
        cache.delete('person_view_data')

@receiver(post_delete, sender=Person)
def invalidate_cache_on_delete(sender, instance, **kwargs):
        cache.delete('person_view_data')
```

Dzięki tak skonfigurowanemu mechanizmowi inwalidacji danych, możemy używać pamięci podręcznej bez limitu czasu, ponieważ każda zmiana danych wymusi usunięcie skorelowanej z widok-

iem pamięci podręcznej. Dzięki temu tylko pierwsze wywołanie widoku po zmianie danych wywoła łańcuch operacji takich jak:

- zapytanie do bazy danych
- paginacja
- serializacja danych
- przygotowanie odpowiedzi.

Każde następne wywołanie widoku skorzysta już z pamięci podręcznej, co może znacząco wpłynąć na czas oczekiwania na odpowiedź serwera.

8 Podsumowanie zachowania pamięci podręcznej cache

Implementacja pamięci podręcznej, takiej jak Redis, niesie ze sobą zarówno korzyści, jak i potencjalne zagrożenia. Nie należy jej stosować automatycznie przy każdym widoku ani dla wszystkich typów danych. Cache nie jest uniwersalnym rozwiązaniem na wszelkie problemy związane z wydajnością, obciążeniem serwerów czy szybkością działania aplikacji.

Mimo to, odpowiednio zaimplementowana pamięć podręczna może znacząco usprawnić działanie systemu, zwłaszcza w kontekście:

- optymalizowania obciążenia baz danych,
- optymalizowania obciążenia serwerów,
- minimalizowania powtarzalności wykonywania tych samych czasochłonnych obliczeń,
- zarządzania i ograniczania zbędnego ruchu
- zwiększania wydajności aplikacji/systemu
- zmniejszania czasu oczekiwania na odpowiedź serwera

9 Bilbiografia

- Artykuł o cache: https://wildasoftware.pl/post/czym-jest-cache-jak-pomaga-przyspieszyc-oprogramowanie
- Rest Framework Cache: https://www.django-rest-framework.org/api-guide/caching/
- django-redis: https://github.com/jazzband/django-redis
- Django-debug-toolbar: https://django-debug-toolbar.readthedocs.io/en/latest/
- Redis: https://redis.io/