Boosting Performance with Data Caching

Cuprins

Cuprins

Cuprins

Cuprins

Proprietăți & Reguli
Extra în Python

Experiențele voastre cu Cache

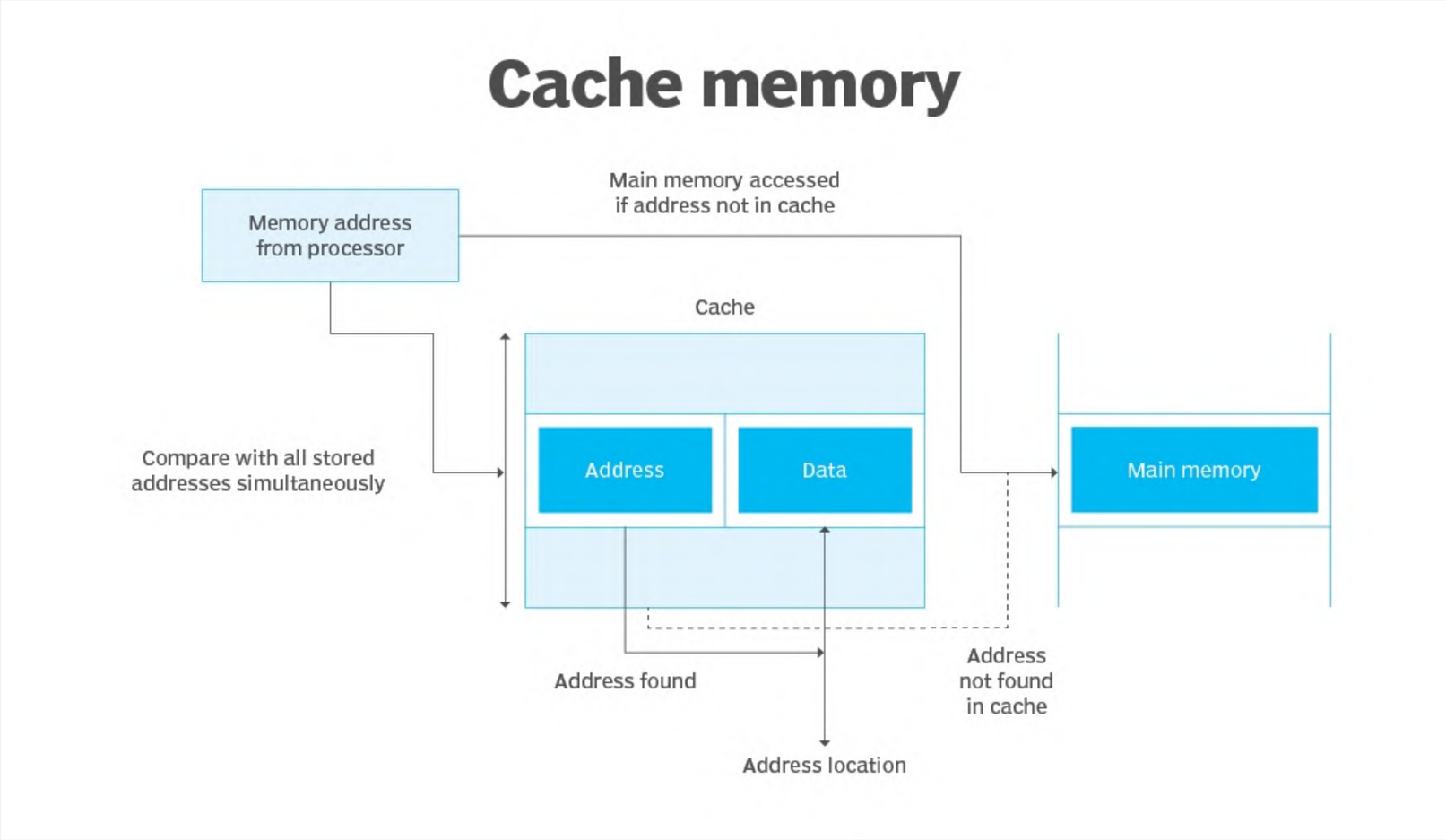
Ce este Data Caching? – Explicația conceptului de data caching

Caching este o tehnică utilizată pentru a îmbunătăți performanța.

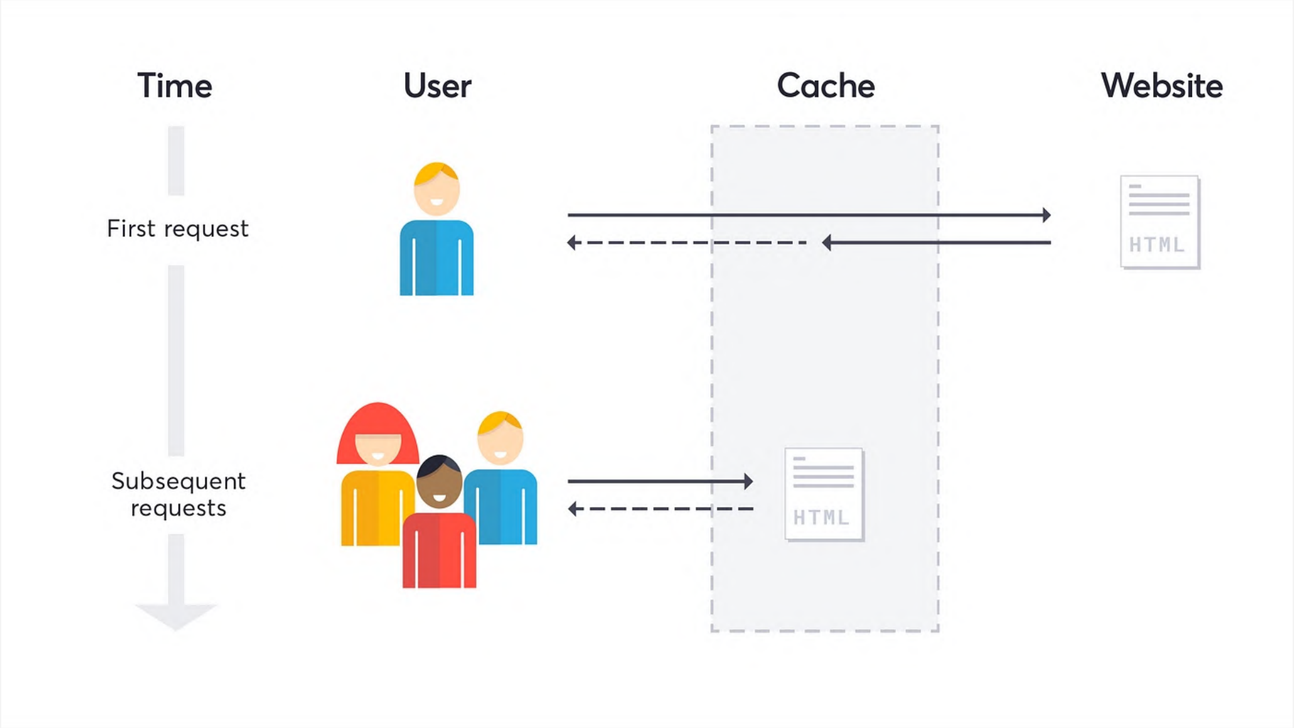
Ce este Data Caching? - Explicația conceptului de data caching

Ideea din spatele caching-ului constă în stocarea rezultatelor costisitoare într-o locație temporară numită cache, care poate fi situată în memorie, pe disc sau într-o locație remote.

Ce este Data Caching? – Explicația conceptului de data caching



Ce este Data Caching? – Explicația conceptului de data caching



Ce este Data Caching? - Explicația conceptului de data caching

Stocarea și reutilizarea rezultatelor apelurilor de funcții anterioare într-o aplicație este de obicei numită memorare.

Ce este Data Caching? - Explicația conceptului de data caching

Un exemplu binecunoscut este calculul șirului Fibonacci.

```
def fibonacci(n):
    if n == 0:
        result = 0
    elif n == 1:
        result = 1
    else:
        result = fibonacci(n - 1) + fibonacci(n - 2)
    return result
```

Ce este Data Caching? - Explicația conceptului de data caching

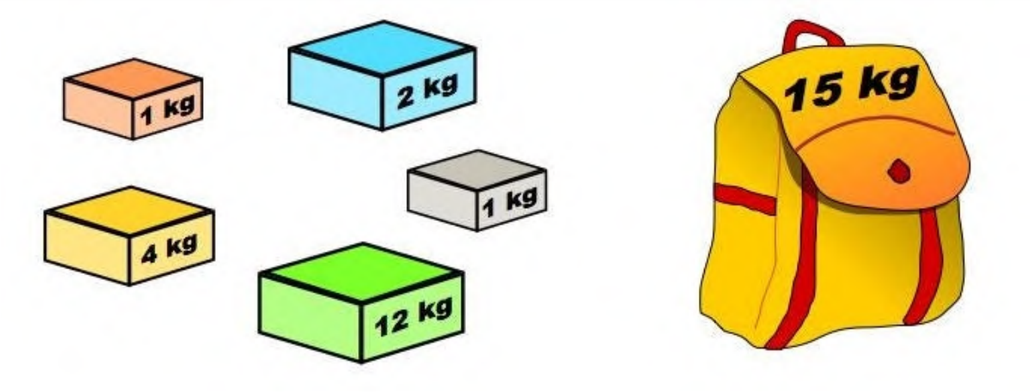
```
memo = {}
def fibonacci(n):
    if n in memo:
        return memo[n]
    if n == 0:
        result = 0
    elif n == 1:
        result = 1
    else:
        result = fibonacci(n - 1) + fibonacci(n - 2)
    memo[n] = result
    return result
```

```
fibonacci 30
0.08440682000946253538 seconds fibonacci_simple
0.00001116399653255939 seconds fibonacci_memo
```

Ce este Data Caching? – Explicația conceptului de data caching

Problema rucsacului (Knapsack problem).

```
21.665253389044664800l7 seconds knapsack_simple
0.0015308170113712549 2 seconds knapsack_memo
```



Caching local

Cache-ul local implică stocarea datelor într-o memorie cache care este locală aplicației sau procesului Python.

Caching local

Cache-ul local implică stocarea datelor într-o memorie cache care este locală aplicației sau procesului Python.

Este potrivit pentru date și rezultate de calcul temporare care trebuie să fie accesate rapid în cadrul aplicației Python.

Caching la nivel de sistem

Cache-ul la nivel de sistem poate implica stocarea datelor într-un spațiu de memorie mai larg, cum ar fi memoria RAM, sau pe disc.

Caching la nivel de sistem

Cache-ul la nivel de sistem poate implica stocarea datelor într-un spațiu de memorie mai larg, cum ar fi memoria RAM, sau pe disc.

Este potrivit pentru date și rezultate care trebuie împărtășite între mai multe aplicații sau procese Python, sau pentru date care trebuie să rămână disponibile chiar și după închiderea aplicației.

Ce este Data Caching? - Tipuri de caching

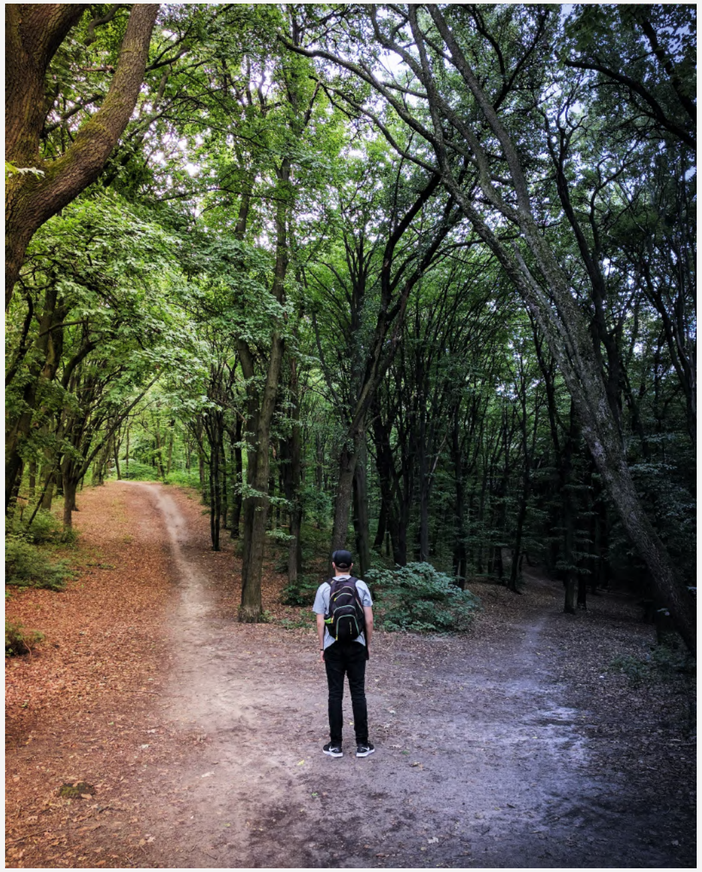
Alta clasificare

Application Server Cache
Distributed Cache
Global Cache

Ce este Data Caching? – Tipuri de caching

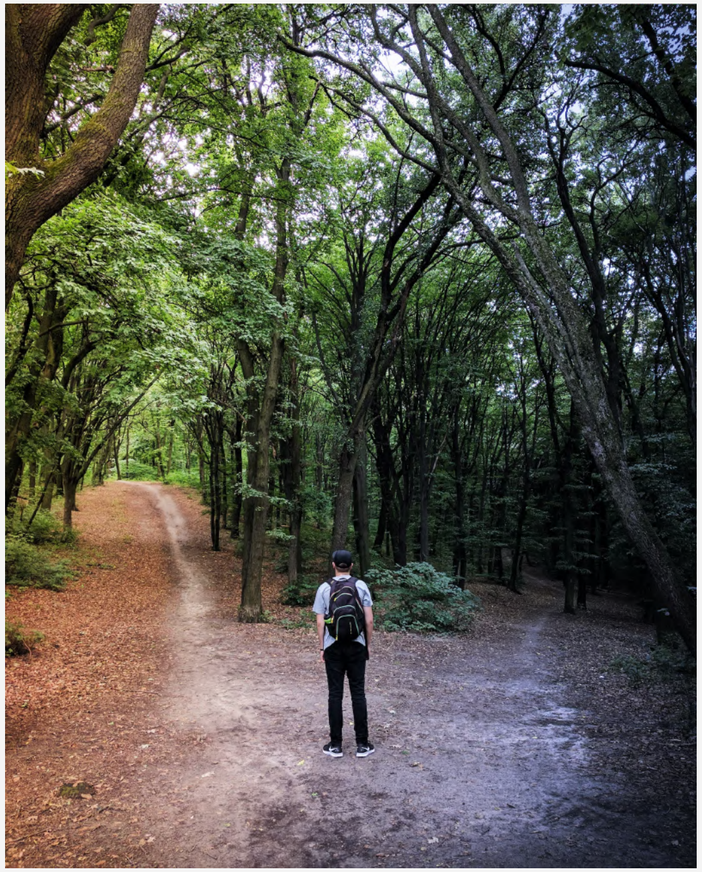
Ce este Data Caching? – Tipuri de caching

Alegem tipul de caching în funcție de cerințele și obiectivele impuse de aplicație

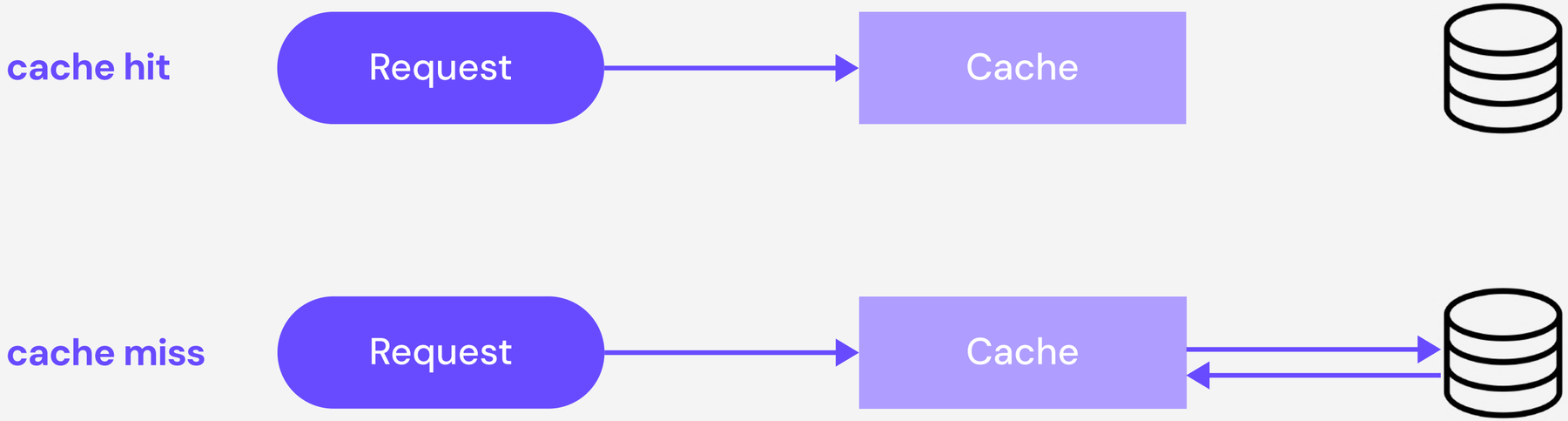


Ce este Data Caching? – Tipuri de caching

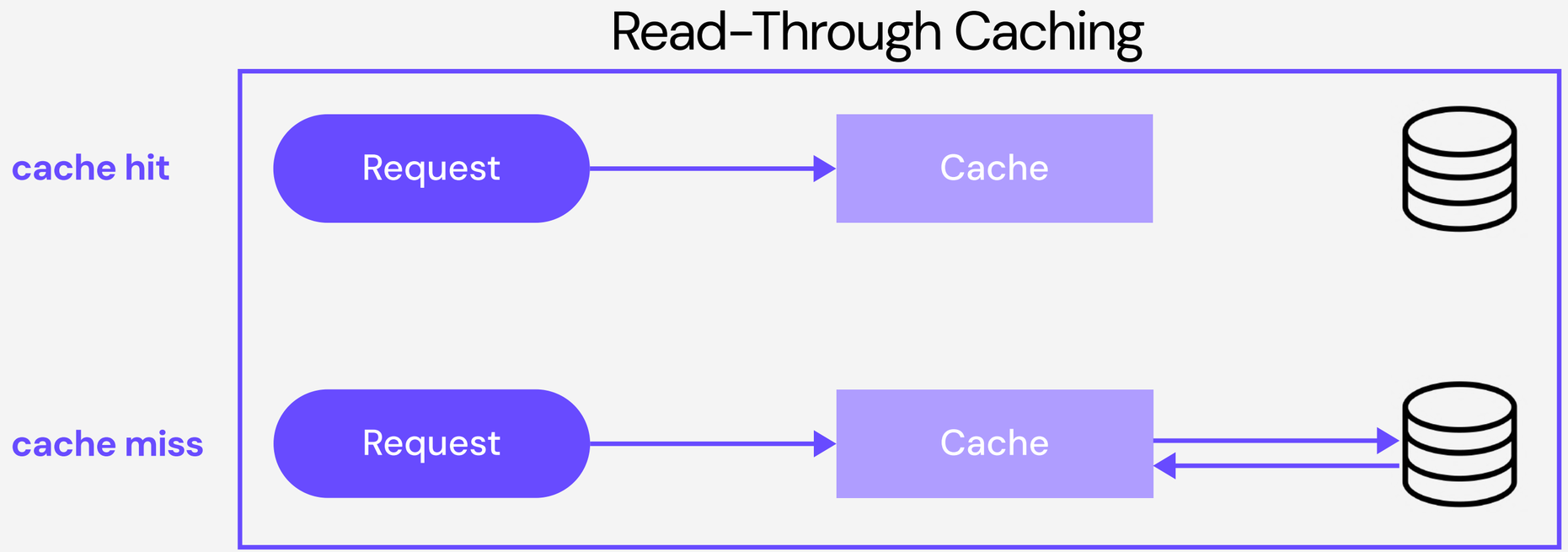
Alegem tipul de caching în funcție de cerințele și obiectivele impuse de aplicație



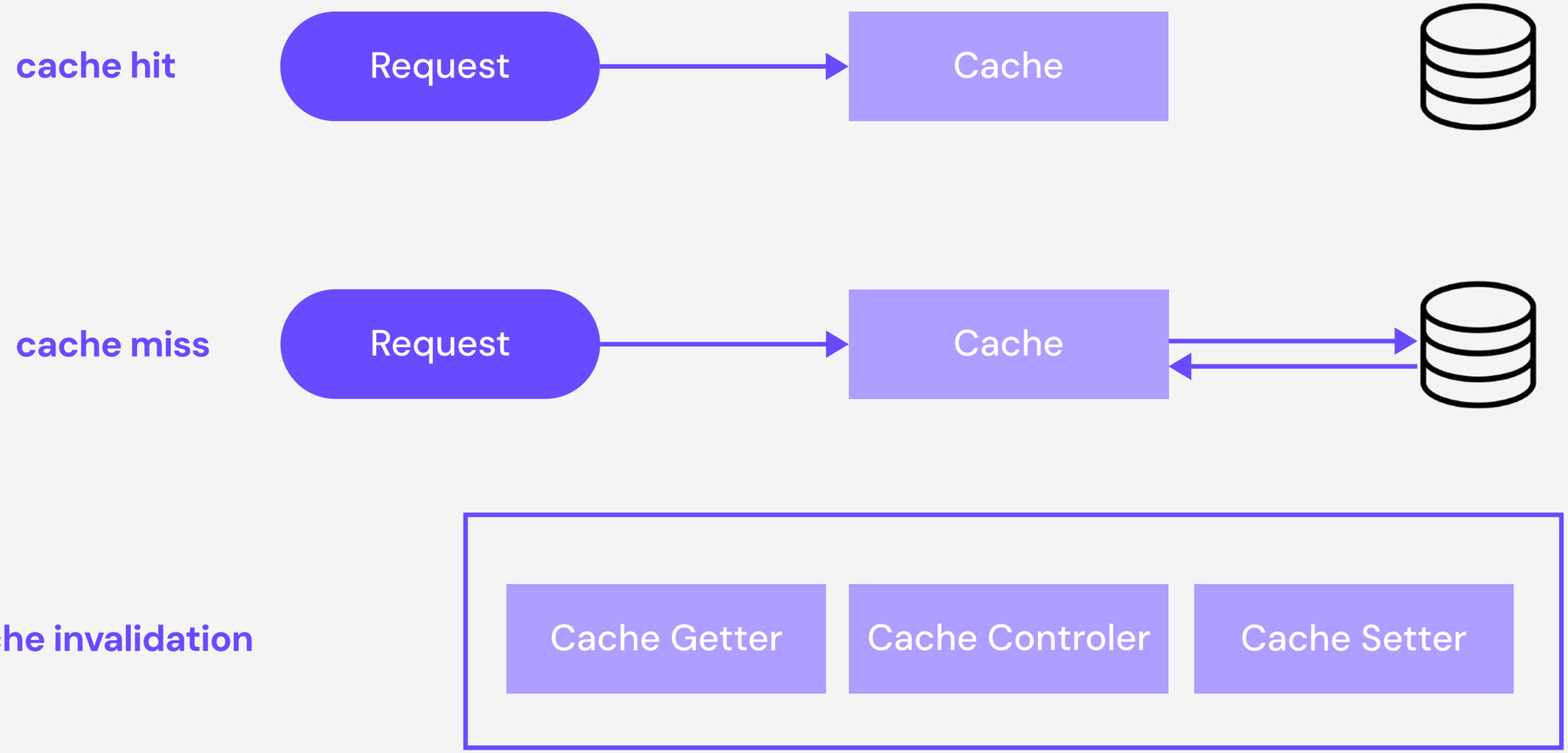
Citire cache



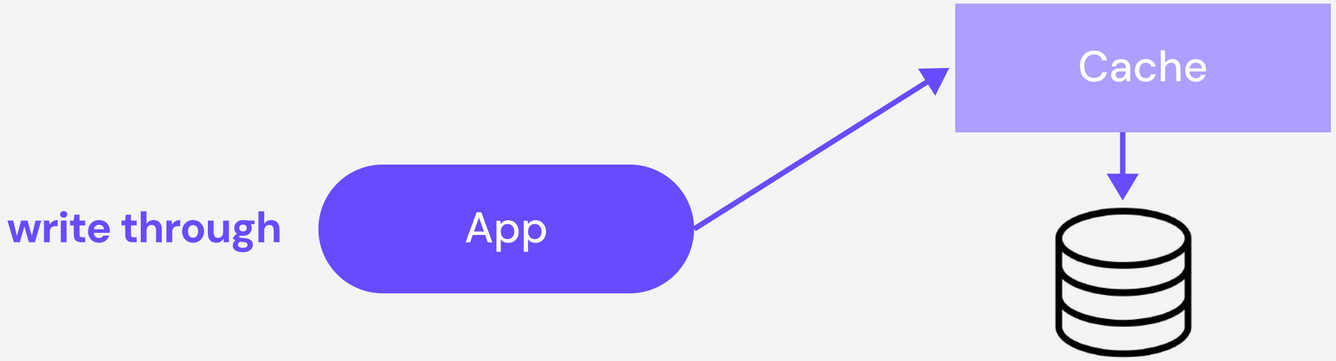
Citire cache



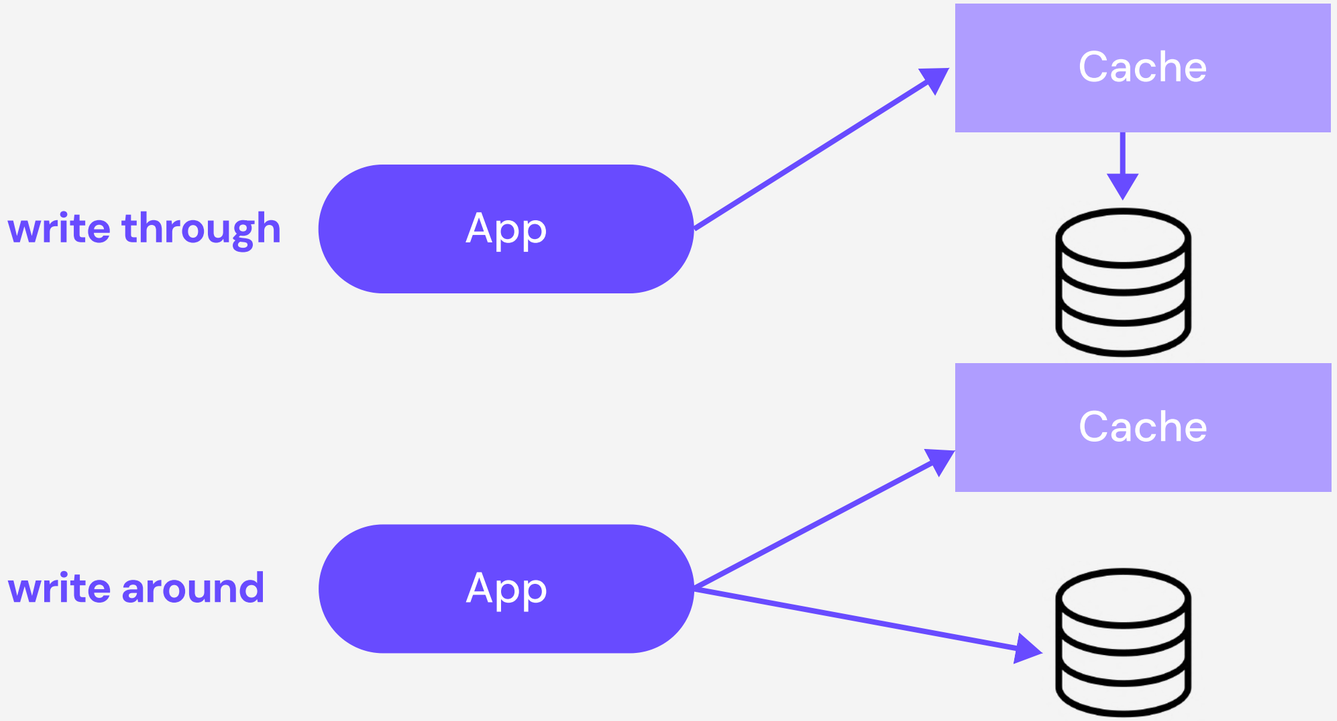
Citire cache



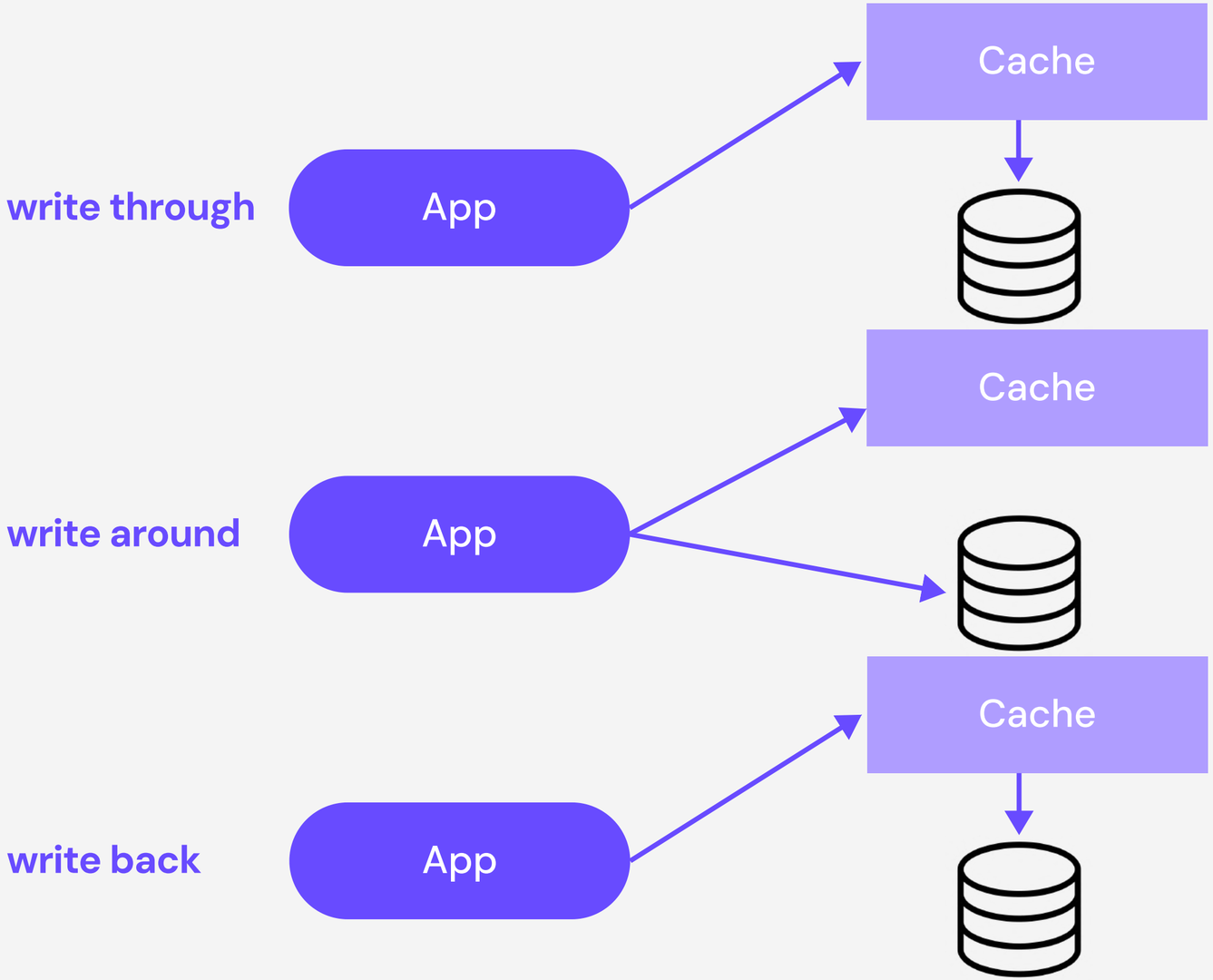
Scriere cache



Scriere cache



Scriere cache

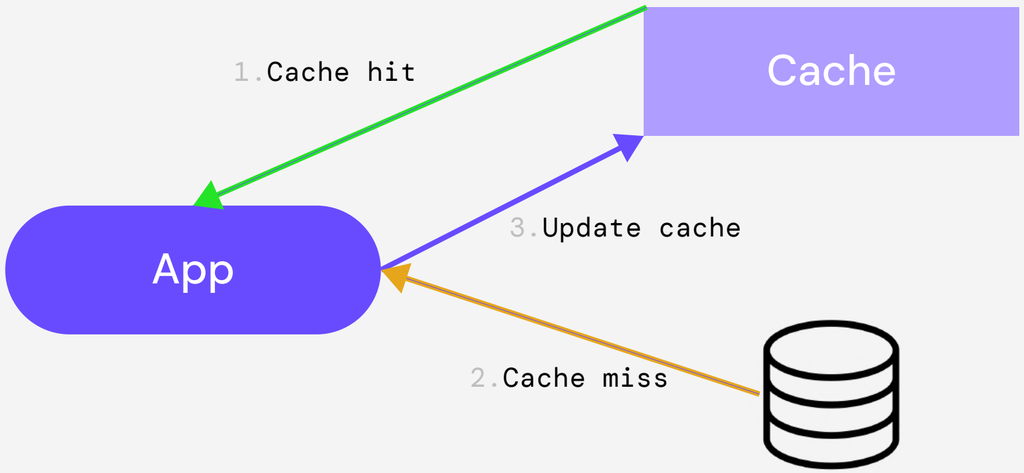


Cache-Aside

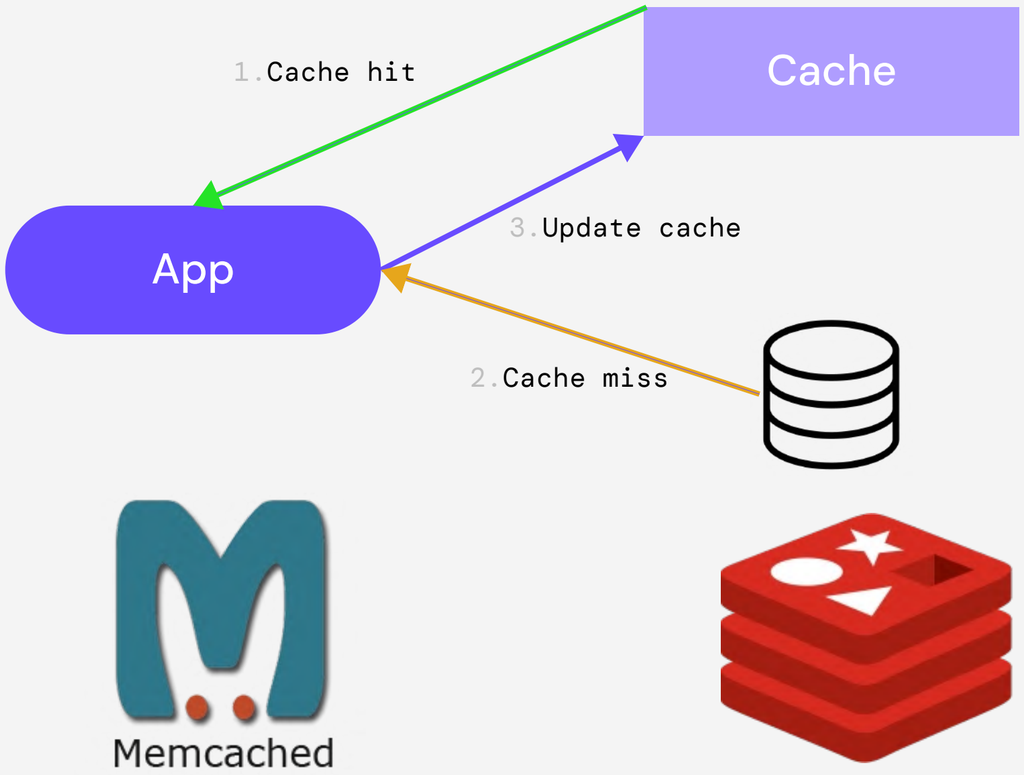
Aceasta este poate cea mai folosită abordare de stocare în cache, cel puțin în proiectele la care am lucrat.

Cache-ul se află pe o parte, iar aplicația vorbește direct atât cu memoria cache, cât și cu baza de date.

Cache-Aside



Cache-Aside



Cache-Aside

```
class CacheAsideWithRedis:
    def __init__(self, host='localhost', port=6379, db=0):
        self.redis_client = redis.StrictRedis(host=host, port=port, db=db)

    def get_from_cache(self, key):
        cached_data = self.redis_client.get(key)
        if cached_data is not None:
            return cached_data.decode('utf-8')
        else:
            return None

    def add_to_cache(self, key, value, expiration=3600):
        self.redis_client.setex(key, expiration, value)

    def remove_from_cache(self, key):
        if self.redis_client.exists(key):
            self.redis_client.delete(key)
```

```
class CacheAsideWithMemcached:
    def __init__(self, servers=['localhost:11211']):
        self.mc = memcache.Client(servers)

    def get_from_cache(self, key):
        cached_data = self.mc.get(key)
        if cached_data is not None:
            return cached_data
        else:
            return None

    def add_to_cache(self, key, value, expiration=3600):
        self.mc.set(key, value, expiration)

    def remove_from_cache(self, key):
        if self.mc.get(key) is not None:
            self.mc.delete(key)
```

Cache-Aside

sunt de obicei de uz general și funcționează cel mai bine pentru "read-heavy workloads"

"resilient to cache failures"

modelul de date din cache poate fi diferit de modelul de date din baza de date

Cache-Aside

Când se utilizează această metodă, cea mai comună strategie de scriere este să scrieți datele direct în baza de date.

Când se întâmplă acest lucru, memoria cache poate deveni inconsecventă cu baza de date.

Pentru a face față acestui lucru se va folosi **TTL**.

Strategii de caching - Impactul alegerii unei strategii de caching

Pași pentru alegerea unei strategii de caching optime

evaluează-ți cu atenție obiectivele

înțelegeți modelele de acces la date (citire/scriere)

alege cea mai bună strategie sau o combinație prin aplicare/comparare

Strategii de caching - Impactul alegerii unei strategii de caching

vs

Memcached - dacă ai nevoie de un sistem simplu de caching pentru a accelera accesul la date într-o manieră ușoară și de încredere

Redis - dacă ai nevoie de o platformă de stocare în memorie cu funcționalități avansate, persistență, scalabilitate și suport pentru diverse tipuri de date

Strategii de caching - Impactul alegerii unei strategii de caching

simplicitate și viteză
structură simplă
cache-aside

Strategii de caching - Impactul alegerii unei strategii de caching

funcționalitate bogată
durabilitate
replicare și clusterizare
cache-aside și alte strategii

Strategii de caching - Impactul alegerii unei strategii de caching

performanță îmbunătățită
reducerea încărcării surselor
economisirea resurselor
asigurarea disponibilității
gestionarea corectă a expirării datelor=
memorarea eficientă a datelor
consistență și actualizare

Strategii de caching - Impactul alegerii unei strategii de caching

date inutile în cache
o latență suplimentară
costuri mari

Cache eviction algorithms - Algorithms used in caching

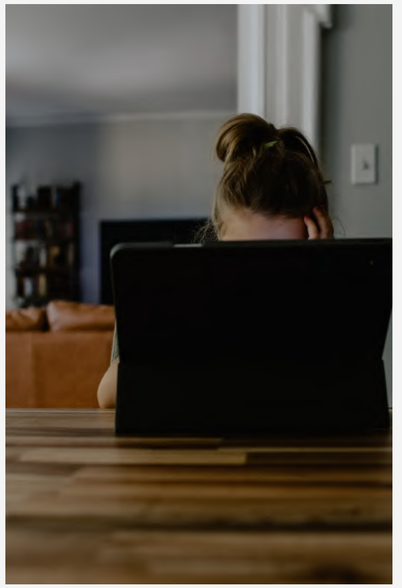
Un "cache eviction" a memoriei cache este o modalitate de a decide ce element să elimine atunci când memoria cache este plină.

Cache eviction algorithms - Algorithms used in caching

```
cache = {}
def expensive_operation(n):
    time.sleep(1)
    return n * 2

def cached_operation(n):
    if n in cache:
        return cache[n]
    result = expensive_operation(n)
    cache[n] = result
    return result
```

```
1.0001901050563901662 seconds - cached_operation
0.00000477093271911144 seconds - cached_operation
```



Cache eviction algorithms - Algorithms used in caching

LRU (Least Recently Used) este o strategie de gestionare a cache-ului care elimină cel mai vechi element atunci când cache-ul este plin.

Cache eviction algorithms - Algorithms used in caching

LFU (Least Frequently Used) elimină cel mai puțin accesate elemente din cache.

Cache eviction algorithms - Algorithms used in caching

MRU (Most Recently Used) este o strategie de înlocuire a elementelor într-un cache care elimină cel mai recent utilizate elemente atunci când cache-ul este plin.

Cache eviction algorithms - Algorithms used in caching

FIFO (First In, First Out) elimină cel mai vechi element din cache atunci când este necesară eliberarea de spațiu.

Cache eviction algorithms - Algorithms used in caching

TTLCache este un tip de cache care șterge automat elementele după o perioadă de timp specificată.

RRCache (Random Replacement Cache) elimină aleatoriu un element din cache atunci când este necesar să se facă loc pentru unul nou, fără a ține cont de ordinea sau frecvența accesărilor.

Cache eviction algorithms - Algorithms used in caching

Pentru fiecare algoritm prezentat veți găsi și o implementare în repo-ul shared.

https://github.com/aar0man/python-masterclass-23

Python built-in LRU algoritm

```
@lru_cache()
def expensive_operation(n):
    time.sleep(1)
    return n * 2
```

Cache eviction algorithms - Caching libraries

cachetools

```
@cached(cache={})
def expensive_operation(n):
    time.sleep(1)
    return n * 2

@cached(cache=LRUCache(maxsize=5))
def expensive_operation(n):
    time.sleep(1)
    return n * 2


@cached(cache=RRCache(maxsize=5))
def expensive_operation(n):
    time.sleep(1)
    return n * 2
```

```
@cached(cache=TTLCache(maxsize=5))
def expensive_operation(n):
    time.sleep(1)
    return n * 2

@cached(cache=LFUCache(maxsize=5))
def expensive_operation(n):
    time.sleep(1)
    return n * 2

@cached(cache=FIFOCache(maxsize=5))
def expensive_operation(n):
    time.sleep(1)
    return n * 2

@cached(cache=TLRUCache(maxsize=5))
def expensive_operation(n):
    time.sleep(1)
    return n * 2
```

django-cacheback

```
from cacheback.base import CachebackJob
```

cacheback_jobs.py

```
class MasterclassJob(CachebackJob):
    def fetch(self, *args, **kwargs):
        return Users.objects.all()
```

views.py

```
def masterclass_users_view(request):
    masterclass = MasterclassJob()
    data = masterclass.get()
```

settings.py

```
CACHEBACK_TASK_QUEUE = 'celery'
```

More about cache – Proprietăți & Reguli

valabilitate
consistență
eficiență

More about cache – Proprietăți & Reguli

găsiți blocajul de performanță

More about cache – Reguli

găsiți blocajul de performanță

verificați care variantă este mai rapidă: să obțineți datele din memoria cache sau prin executarea directă a logicii

More about cache – Reguli

găsiți blocajul de performanță

verificați care variantă este mai rapidă: să obțineți datele din memoria cache sau prin executarea directă a logicii

verificați memory footprint

More about cache - Reguli

găsiți blocajul de performanță

verificați care variantă este mai rapidă: să obțineți datele din memoria cache sau prin executarea directă a logicii

verificați memory footprint

decideți la ce nivel să aplicați cache-ul

More about cache – Reguli

nu salvați date senzitive în cache

More about cache - Extra în Python

```
class CachedAttribute:
    def __init__(self, method, name=None):
        self.method = method
        self.name = name or method.__name__

    def __get__(self, instance, cls):
        if instance is None:
            return self
        result = instance.__dict__[self.name] = self.method(instance)
        return result

class MyClass:
    @CachedAttribute
    def my_method(self):
        # Some expensive computation goes here
        return result
```

More about cache – Extra în Python

```
import asyncio

async def refresh_cache():
    while True:
        update_cache_with_fresh_data()
        await asyncio.sleep(3600)

loop = asyncio.get_event_loop()
loop.create_task(refresh_cache())
```

More about cache – Extra în Python

```
import pickle


with open('cache_file.pkl', 'wb') as f:
    pickle.dump(my_cache_data, f)


with open('cache_file.pkl', 'rb') as f:
    my_cache_data = pickle.load(f)
```

Boosting Performance with Data Caching - Lessons learned

este necesar să înțelegem datele aplicației noastre

trebuie să alegem o strategie potrivită care să ne permită scalarea

datele din cache trebuie să fie invalidate

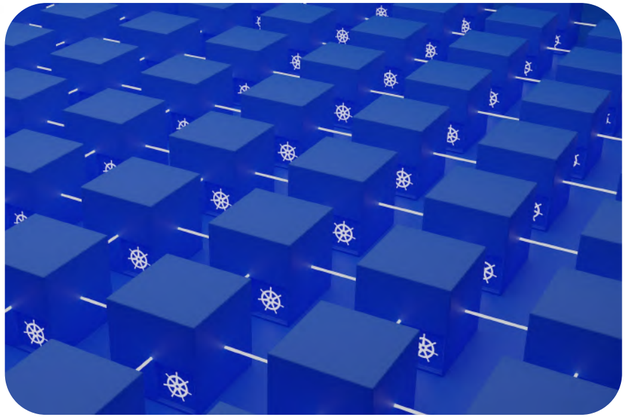
dimensiunea cache-ului este foarte importantă

alegeți nivelului de granularitate potrivit

atenție la caracterul datelor salvate în cache

Boosting Performance with Data Caching

Performance Optimization Techniques



Journey into Microservices