Java HW - T-Mobile

Aleš Prokop

28. ledna 2026

Návrh a implementace Rate Limiteru

Základní design - single node

1. Použil bych jednoduchý interface, ve kterém by hlavní metoda fungovala tak, že by jen rozhodla, jestli klienta pustíme, nebo musí počkat a odmítneme ho. Metoda by tedy jen vracela `boolean`, který reprezentuje, jestli je access allowed nebo ne. Argument této metody je `clientID`, které zaručí, že pro každého klienta spravedlivě rozhodneme, jestli projde nebo ne, vzhledem k počtu jeho požadavků. Tato metoda by šla upravit, pokud bychom měli například nějaké prioritní klienty a mohli bychom zavést buď priority, nebo `cost` (cenu požadavku), pokud by například nějaká volání byla dražší.

2. Hlavní myšlenka algoritmu je následující. Budeme mít uložený počet klientů, které momentálně můžeme pustit, to můžeme reprezentovat například "tokeny". Tento počet závisí na tom, kolik klientů chceme pustit každou vteřinu k naší API službě. Budeme tedy mít aktuální počet tokenů uložený v našem vědru (`Bucket`). Každý request potom odečte počet tokenů, kolik stojí jeho přístup (v případě, že máme ceny), respektive jeden token, pokud ceny nemáme. Pokud ten počet tokenů v tomto vědru ještě je, odečteme je. Pokud ne, rovnou odmítneme přístup k naší REST API službě. Tokeny se doplňují. Opět, tokeny můžeme doplňovat několika způsoby. Jeden z nich je doplňovat tokeny každou vteřinu, ale lepší způsob je doplňovat tokeny při každém accessu, podle toho, kolik času uběhlo. Rychlost doplňování opět záleží na tom, jak rychle chceme požadavky pouštět. Tohle řešení je lepší, než například sliding window z toho důvodu, že při sliding window bychom se mohli trefit přesně na rozhraní a povolit například dvojnásobek volání.

3. Zde už je dobré myslet na to, že se může stát, že více klientů bude přistupovat ke kyblíkům najednou. Vzhledem k tomu, že potřebujeme rychlé vyhledávání dle ID, tak hash mapa je nejlepší volbou. V Javě existuje `ConcurrentHashMap`, která zaručí, že se nerozbije přidávání a vyhledávání uživatelů při více přístupech najednou.

4. Pro tento bod jsme už udělali trochu práce v bodě 3. Ale může se nám ještě rozbít samotné vědro, to zatím není nijak chráněné. Zde máme několik možností:

    - Nejsnazší metoda je použít klíčové slovo `synchronized`, které bude pouštět vlákna jedno po druhém. Můžeme buď zamknout celou aplikaci, což by ale bylo relativně pomalé, nebo můžeme zamknout jen instanci `Bucket` pro daného uživatele.
    - Další možností by teoreticky bylo použít atomické proměnné, které fungují dobře. Nicméně, pro atomickou změnu více hodnot najednou musíme použít nový objekt (immutable state). Místo toho, abychom ukládali čísla přímo do vědra, budeme ukládat referenci na stav vědra. To nám potom dovolí použít `AtomicReference` a atomické operace, které jsou rychlejší než zámky.

Distribuovaný systém (Multi-node)

1. Problém je ten, že pro uživatele limit funguje separátně na každém serveru. To znamená, že uživatel může posílat více požadavků, než jsme mu dovolili. Pro příklad, mějme například uživatele Aleše. Uživatel Aleš má povolených 10 požadavků za sekundu. Máme 5 serverů. Aleš pošle 50 požadavků a load balancer je zrovna rozdělí po 10 požadavcích na každý server, protože každý server má svoje počítadlo, tak všechny požadavky projdou i když by neměly.

2. Tento problém můžeme vyřešit následovně. Počet požadavků a stav nebude na každém serveru zvlášť, ale sjednotíme ho. Použijeme `Redis`, který nám v podstatě nahradí naši hash mapu. Aplikace pošle ID klienta a požadovanou kapacitu a na Redisu se dopočítá logika a vrátí `boolean`, jestli je access povolený nebo není.

Integrace a produkční nasazení

1. Limity klientů si budeme udržovat v nějakém souboru, například typu `json`. V tomto souboru bude `ClientID` a limit. Tím můžeme před startem aplikace měnit jejich limity. Pokud bychom chtěli aplikace nerestartovat, mohli bychom implementovat nějaký watcher, který by nám řekl, že se změnil soubor a pokud ano, tak aktualizuje hodnoty, které jsou v naší hash mapě.

2. K testování musíme použít několik věcí. Nejdříve musíme použít simulaci času, abychom nemuseli při testování čekat, až tento čas reálně uběhne. Vícevláknové chování budeme testovat pomocí `ExecutorService` a poté všechna vlákna pustíme najednou pomocí `CountDownLatch`. Tím můžeme testovat i vysokou zátěž.

Vyřešení cyklických závislostí

Cyklické závislosti

Obecně jsou cyklické závislosti špatný objektový návrh. Mezi řešení patří refaktorizace kódu, která je doporučená. Například můžeme společnou logiku vytáhnout do "nadřazené"třídy. Ve springu se to dá vyřešit pomocí `@Lazy` anotaci. Tahle anotace udělá to, že se třída inicializuje až když je vyžádána, místo toho, aby se inicializovala při startu aplikace.