QuickCache Analysis

QuickNode Interview Written Assignment

Next are some potential issues I found regarding the current PoC:

- On storage I: I/O are costly operations and making them against the server file system (FS) will make the situation worse.
- On storage II: nothing assures me that JSON files are going to be stored contiguously, thus, they can be pretty dispersed on the disk. Accessing different physical positions on a disk is costly.
- On storage III: FS usually splits files into blocks and blocks are reserved for that specific file so on-time data will use too many blocks - reducing memory capacity.

- On coding practices I: string "*filePath*" is being constructed 3 times instead of one time on the public function "*get*" - this enhances code comprehension and avoids using resources to recreate a string that does not change during the request.
- On coding practices II: there is no consistency on functions signatures, public function "*get*" returns a "*Promise<ReadonlyArray<JsonFragment> | null>*" but then functions "*readCachedFile*" and "*abi*" return different than that, only "*fetchContractAbiIfNotCached*" returns the same - this is critical for code comprehension and type verification.

- On coding practices III: there is no value type verification on functions
  - "*abi*": you might be parsing something that is not a valid JSON;
  - "*readCachedFile*": again, the file might exists and might be stored as .json but it could be a no valid JSON;
- On coding practices IV: no error handling on functions:
  - "*abi*": "*etherscanProvider.fetch*" returns a Promise, if an error arises it's not handled - it's the same for "*promises.stat*" and "*promises.writeFile*";
  - "*readCacheFile*": if no file exists while performing reading, no exception is caught;

Next are some recommendations to enhance functionality regarding the current PoC:

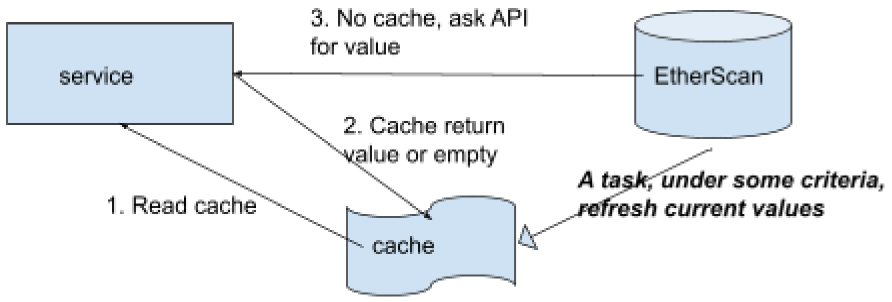
- On storage I: a Key-Value structure is the best fit for this situation since the ABI's address determines the ABI itself. These structures are optimized for direct access to the data through its associated key.
- On storage II: either in memory or within a DB (like Redis or Memcached), Key-Value structures are designed to be contiguous on physical memory to optimize access. Ideally, all operations in a Key-Value structure should be close to O(1), whereas on FS it would be on average near to O(logN) depending on the structure to use (for example trees).

- On coding practices I: no chaining of Promises is made, which would allow better control of the program sequence depending on the result of the Promises. Also, it gives the viewer of the code a better understanding of it. Flow control is paramount.

- On architecture and design I: needs to be cache-resilient, meaning, a failure on cache access should direct request to EtherscanApiClient.
- On architecture and design II: a strategy for frequently/more accessed data is needed to not fulfill the cache with *not-so-much-used* data - in other words, an eviction policy through some algorithm is needed although consideration is needed for those Values whose size is quite large or the time-to-access them is slow (meaning, no eviction is applied)
    - Eviction Algorithm: some of them include Most Recently Used, Least Recently Used, Least Frequently Used, FiFo, among others
- On architecture and design III: with solutions like Redis or Memcached you avoid the issue of losing data with in-memory approaches, multiple nodes can be deployed providing high availability and resilience (though in a total collapse, fall back to EtherscanApiClient should be applied)

And finally, next are some suggestions if we consider Proxy Contracts*:

- An Expiration Policy: a TTL is needed for updates on present data, avoiding the service of stale data;
- A different Architecture and Design: A refresh-ahead (task) pattern could help in keeping up-to-date data, usually before users request them. This in combination with the applied cache-aside pattern would be something like the following:


3. No cache, ask API for value
service
EtherScan
2. Cache return value or empty
A task, under some criteria, refresh current values
1. Read cache
cache

---

* Proxy Contracts: an ABI can ***<u>virtually</u>*** change, so for example a user can still request the ABI to the same known address but this address actually contains a proxy contract which then calls whatever address it has defined in its code (thus, it can point to newer versions) and retrieves its ABI.