UDC 004.9
DOI https://doi.org/10.32782/2663-5941/2023.1/20

Falkevych V.G.
Zaporizhzhia National University

Lisniak A.O.
Zaporizhzhia National University

METHODOLOGY OF CACHE INVALIDATION IN MICROSERVICES ARCHITECTURE OF THE WEB APPLICATIONS

In the article are considered existing approaches, methods of using cashing in scalable web systems and decisions proposed in related works. Created a methodology of cache invalidation in microservices architecture. The object of researching is a process of creating a methodology of cache invalidation. Methods of researching are based on modeling cache management processes and services cooperation. Defined requirements for cache invalidation systems: splitting of area responsibilities, invalidation only part of caching, prevention of repeating cache invalidation by time, non-blocking operation. Described all steps of the system`s components cooperation that includes API gateway, cache service, cache, application services. Described examples of using an application with cache service for keeping cache fresh. In the article are included 1 figure with schematic components of the application architecture, 2 sequence diagrams with steps of system elements cooperation in time. Defined main principles in the microservices architecture with cache system: cache invalidation abstraction from the inside services to the cache service by creation cache schema for every services; describing rules of preparation API responses; requesting to the cache from services only through the cache service; using API gateway as a router between services; changing cache schema should not affect to the work of the cache service. Researched base necessary rules for cache invalidation: cache schema, response schema (or response prepare handler), action. Cache schema describes an object as value from the service depend on type and name of the request; response handler or response schema – are rules or ready for using function (after decoding) for preparing response before client receives response. Action – is a type of command that should be done (створити, оновити, видалити).

Key words: *cache, PASS, Redis, Memcashed, ElastiCache, Memorystore, API, cache invalidation, microservice.*

Problem definition. Nowadays the understanding of using microservices approaches in the development of scalable projects have become a necessity. Code splitting, areas of responsibility, business logic and views is a common using practice for modern developers. Users become more templated, systems are more demanded of resources. Ways of optimization are improved everyday, same as technologies which allow us to achieve these goals [1]. Using caching methodologies and different ways allow to decrease loading for servers, help to make systems more effective and scalable, reduce count of requests to databases and catching deadlocks, time for making selection data for next calculation before response [2]. A key-value storage is one of the best ways to get access to useful data without heavy selection demand of resources. But nothing is taken nowhere and does not lead anywhere. Every improvement resolving some field of problem brings another inconvenience. For instance, managing cache storage, making additional infrastructure which allows it to work with this storage, scaling storage, mechanisms of synchronization with databases. Another inconvenience we can face with – is getting expired data.

Analysis of recent research and publications. Considering cache systems and approaches can be highlighted the most popular and common using of them: Memcached or Redis. Cache – is a key-value storage which keeps data in memory without saving it like persistent files [3]. It allows you to avoid a process of reading/writing data from files to RAM. So Redis or Memcached allows you to get stored data as fast as RAM memory can do it. Many companies offer ready solutions like cloud caching like PASS (platform as service), for example, Amazon ElastiCache, Google Memorystore etc.

Describe the principle as cache works. When an application tries to get some data from a database, make a request to the cache. If the data by key exists, return it. In another way return data from the database and store in the cache to further requests. Saving data in cache is necessary to set up time for caching. It can be a timestamp with the date when cache by key was

written [4]. Having timestamp different approaches can be implemented for keeping data in the cache fresh. For instance, cache can be invalidated by comparing actual time with timestamp when a request is made by an application. Another way implies using planning tools like CRON for invalidation cache bypassing all keys step by step comparing actual date with timestamp [5].

First way of cache invalidation can seem more preferable than bypassing all values. But it can also have disadvantages. To point them consider using caching in MVC systems (model view controller). Model is responsible for getting data. It can be data from a database or data from API. Writing models developers should make decisions about what data should be stored in cache, what data can be updated [5]. Similar approach can be proposed in modern front-end development using UI building libraries like React, Angular, VueJS on the rope with local cache management using Apollo GraphQL. In both cases developers should make decisions about which data should be updated in cache. It would be good to have an opportunity for flexible tools for cashing. But returning to the principle of code and duties splitting, the logic of cashing and business logic intertwines in the code [6, p. 136].

The main reason why developers make decisions about the necessity of updating cache is that no one wants to get expired data. Should be another way for working with cache. Microsoft has offered to use NCache instead Redis for caching. This technology allows synchronization with databases [2]. Developers should not describe caching mechanisms in their code that allow them to follow the fifth rule of SOLID principle (Dependency Inversion. Depend upon abstractions, not concretics).

Take a look at NCache in detail. There are some available features using this technology:

1. Pub/Sub Messaging;
2. cache distribution;
3. SQL searching and grouping;
4. database synchronization.

The publishing-subscribing messaging pattern allows senders of messages, called publishers, to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers. The events are published outside the microservice, to the NCache message broker [2]. Each subscriber microservice contains an event handler to handle the appropriate event once the publisher microservice has published it.

NCache being In-memory distributed cache allows microservices to provide scalability and handle a larger number of transactions.

When so much data is being stored in cache by microservices for their app data caching needs, then having the ability to quickly find relevant data through SQL searching or grouping make it very easy to process it [1].

Another feature proposed by Microsoft is database synchronization. When the database receives some changes, the specific field in the cache also updates. To use this opportunity it is necessary to set up a connection with the database first using API and describe procedures of cache and database synchronization, which consist of next steps:

1. creating a dependency with stored procedure;
2. getting an item from the database;
3. generation a unique id for the item;
4. creating a new cache item and adding dependency to it.

Using database synchronization mechanism it is possible to describe all entities that should be updated in cache when database updates. It avoids using expired cache and supports splitting responsibilities comparing the way developers should make decisions when cache should be updated [7, p. 14]. The main disadvantage in using this approach is binding to certain databases and stack of technologies. If a developer changes NCache to Redis or Memcashed he loses an opportunity to use database synchronization. It is necessary to have a unified mechanism of keeping cache fresh [8, p. 43].

The main purpose of this article is considering existing approaches, methods of using cashing in scalable web systems and decisions proposing in related works. Will be found and offered a methodology of cache invalidation in microservices architecture. The object of researching is a process of creating a methodology of cache invalidation. Methods of researching are based on modeling cache management processes and services cooperation.

Main research material. Moving ahead in considering systems and technologies for caching becomes obvious that cache should be invalidated to prevent working with expired data. Developers should follow the fifth principle of SOLID (dependency inversion) and split areas of responsibilities [9, p. 230].

One of the ways to do that is creating an independent mechanism for cache invalidation using cache service and providing cache schema from the services.

Define requirements for cache invalidation methodology:

– splitting of area responsibilities;
– invalidation only part of caching;
– prevention repeating cache invalidation by time;
– non-blocking operation.

According to the inverse dependency principle, cache invalidation implementation should not depend on microservices logic like microservices logic and vice versa [10, p. 179].

Invalidation mechanism should not affect cache that should not be updated. Simultaneously all levels of caching should be invalidated by time. If the part of cache is invalidated by event, the timestamp when cache was rewritten should be updated also. An operation for cache updating is sent after service has been worked and should obey similar rules. If a request to the service and next response initiates the cache updating, finite users should not await when cache updates. Another similar operation that can be sent by service at the same time when cache updating should not rewrite or interrupt to work for the system [11, p. 1156].

To achieve these requirements, take a look at figure 1. The diagram is described here, has next components:

– API gateway;
– cache service;
– cache;
– application services.

API gateway receives all requests from the client and redirects them to services. Services should have schemas that describe a memoization value in the cache. That schemas can be changed and cache invalidation continues working correctly. All services get cache using API gateway through cache service for connection to the cache system. Using cache schema cache service can update, add or delete memoized values. At the same time when the first service responds, the data is sent from the service can be used in another service without waiting for cache service updates.

Cache service – is an essence that has responsibilities:

– updating cache using service`s cache schema with data;
– getting data from cache for service.

A process of updating cache should be in parallel with working application services. That means no one services and API gateway should not wait when cache will be updated [12, p. 157].

Consider in detail how the cache invalidation system works (figure 1). For instance, there is a request that creates a new book in the database. After this action happens we would like to add this book to the cache without rewriting all books. Making a POST request /books/create by client through API gateway the request is redirected to the service responsible for this route and method [13, p. 54].

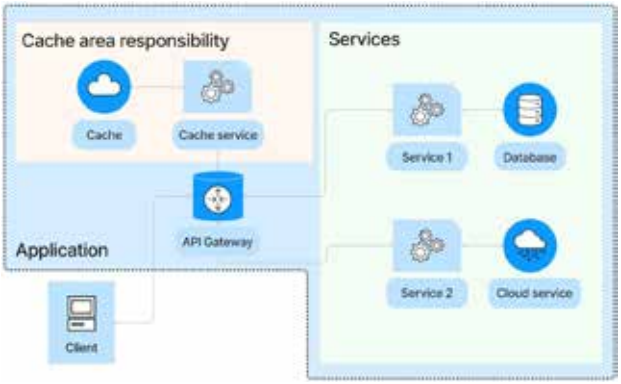

Fig. 1. application architecture with a cache system

Next step – is processing input data by Book service and creating a new book in the database. After that the service responds to the API gateway and provides a cache schema with data about the created book. The gateway makes routing to the service responsible for cache invalidation and provides new data and the schema. Cache service reads the schema and connects with cache and calculates updates depending on input data [14, p. 28; 15, p. 519].

The schema for caching should include comprehensive information about type of request (create, update, delete), structure for caching and also can be provided rules for data normalization before response is returned to the client. That means the API should not include overhead information in the response about the action that happened. But at the same time the cache service needs data to work with, however this data is not to be used in the response. Describing rules for preparation response we can move the logic of preparation of the data for response on another level of abstraction. For example, it is possible to create a service, responsible only for data preparation before response received by client [16, p. 89; 17, p. 52].

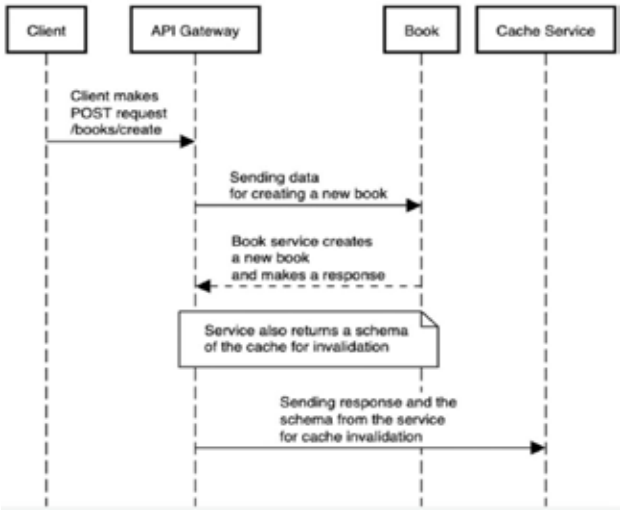

Fig. 2. A process of cache updating using cache service

That service receives data, rules for data preparation, makes preparation and responses to the client through API gateway [18, p. 360].

Cache updating process can take time and there is not any guarantee that this process is finished before the next request is called. Considering this aspect of asynchronous mechanism cache invalidation and working services during one request, it is necessary to create a queue in the cache service and keep processes here. Only after the first process finishes, the next can be started [19, p. 375].

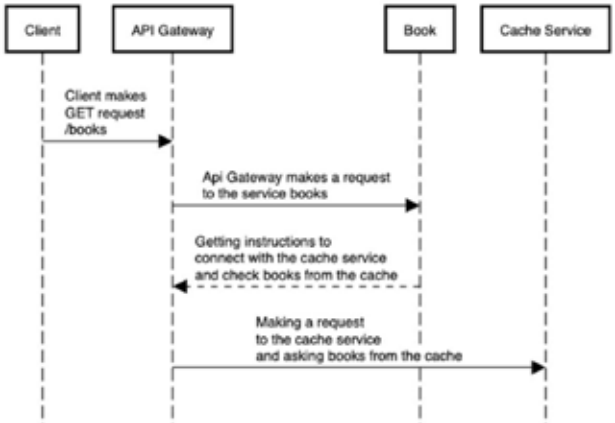

Fig. 3. A process of getting a list of books from the cache

Take a look at the next example: we make a GET request/books for getting a list of books (figure 3). First step – is making a request from client to the API gateway and it makes route to the Book service. This service makes a request to the cache service for checking books in the cache through API gateway. If books are found in the cache, the Book service creates a response. Otherwise this service makes a request from the database and gets books from it. Responding without schema of cache invalidation API gateway understands that response is finite and it is not necessary to call cache service [20, p. 87].

Considering peculiarities of working services with cache must be mentioned working services with each other how data can be delivered from service to service. For example, an author has private books and additional fields that can not be delivered to the not authorized user and the author wants to add new characteristic values to one of his books. In the case when we should be routed to the Auth service (a service responsible for authorization), get user`s data (if credentials succeed) and get books as authorized author from another service, the Auth service calls the Book service through API gateway. Then the last service creates a new characteristic to the one of the author`s book, prepares rules for normalizing the response to the client, sending schema for the cache invalidation [21, p. 169].

Conclusions. Considered existing approaches, methods of using cashing in scalable web systems and decisions are proposed in related works. Created a methodology of cache invalidation in microservices architecture. Have formulated the requirements for cache invalidation methodology consists of:

– splitting of area responsibilities;
– invalidation only part of caching;
– prevention repeating cache invalidation by time;
– non-blocking operation.

Have proposed a full flow of cache invalidation. Considered components of the application architecture with the cache invalidation mechanism:

– API gateway;
– cache service;
– cache;
– application services.

Described examples of using an application with cache service for keeping cache fresh. In summary, are defined main principles in the microservices architecture with cache system: abstraction from the services to the cache service by creation cache schema for every services; defined rules of preparation API response after a service responses; requesting to the cache from services only through the cache service; using API gateway as a router between services; changing cache schema should not affect to the work of the cache service.

Bibliography:

1. Scale Microservices Performance with Distributed Caching. URL: https://www.alachisoft.com/blogs/scale-microservices-performance-with-distributed-caching.
2. Cache Data Dependency on Database. URL: https://www.alachisoft.com/resources/docs/ncache/prog-guide/notification-based-dependencies.html.
3. Кеширование данных между микросервисами в бессерверной архитектуре. URL: https://habr.com/ru/post/651829.
4. Кэширование в облачном приложении. URL: https://learn.microsoft.com/ru-ru/dotnet/architecture/cloud-native/azure-caching.
5. Кеширование в Laravel: основы плюс tips&tricks. URL: https://habr.com/ru/post/463495.
6. Charan, P.S.B., Varshitha, G., Lashya, A., Varma, U.S.R. and Madhusudhan, D., REDIS: IN MEMORY DATA STORE. 2022. № 5. P. 132–138.

7. Gupta, P., Zeldovich, N. and Madden, S., 2011. A trigger-based middleware cache for ORMs. In Middleware 2011: ACM/IFIP/USENIX 12th International Middleware Conference, Lisbon, Portugal, December 12-16, 2011. Proceedings 12 (pp. 329-349). Springer Berlin Heidelberg.

8. Ghandeharizadeh, S. and Shayandeh, S., 2007, April. Greedy cache management techniques for mobile devices. In 2007 IEEE 23rd International Conference on Data Engineering Workshop (pp. 39-48). IEEE.

9. Katsaros, D. and Manolopoulos, Y., 2003. Cache management for Web-powered databases. In Web-Powered Databases (pp. 203-244). IGI Global.

10. Liu, Q. and Yuan, H., 2019. A High Performance Memory Key-Value Database Based on Redis. J. Comput., 14(3), pp.170-183.

11. Carra, D. and Michiardi, P., 2014, June. Memory partitioning in memcached: An experimental performance analysis. In 2014 IEEE International Conference on Communications (ICC) (pp. 1154-1159). IEEE.

12. Gan, Y. and Delimitrou, C., 2018. The architectural implications of cloud microservices. IEEE Computer Architecture Letters, 17(2), pp.155-158.

13. Pavlenko, A., Askarbekuly, N., Megha, S. and Mazzara, M., 2020. Micro-frontends: application of microservices to web front-ends. J. Internet Serv. Inf. Secur., 10(2), pp.49-66.

14. Brown, K. and Woolf, B., 2016, October. Implementation patterns for microservices architectures. In Proceedings of the 23rd Conference on Pattern Languages of Programs (pp. 1-35).

15. Sriraman, A., Dhanotia, A. and Wenisch, T.F., 2019, June. Softsku: Optimizing server architectures for microservice diversity@ scale. In Proceedings of the 46th International Symposium on Computer Architecture (pp. 513-526).

16. Familiar, B., 2015. Microservices, IoT, and Azure (pp. 69-93). Berkeley, CA: Apress.

17. Bucchiarone, A., Dragoni, N., Dustdar, S., Larsen, S.T. and Mazzara, M., 2018. From monolithic to microservices: An experience report from the banking domain. Ieee Software, 35(3), pp.50-55.

18. Guo, D., Wang, W., Zeng, G. and Wei, Z., 2016, March. Microservices architecture based cloudware deployment platform for service computing. In 2016 IEEE Symposium on Service-Oriented System Engineering (SOSE) (pp. 358-363).

19. Munonye, K. and Martinek, P., 2020, June. Evaluation of Data Storage Patterns in Microservices Archicture. In 2020 IEEE 15th International Conference of System of Systems Engineering (SoSE) (pp. 373-380). IEEE.

20. Parekh, J., Moroney, A., Golani, L. and Shankarmani, R., A Timestamp based Novel Caching Mechanism for Distributed Web Systems. International Journal of Computer Applications, 975, p.87-88.

21. Fajardo, M.E., 2020, November. Building Microservices for Scalability and Availability: Step by Step, from Beginning to End. In New Perspectives in Software Engineering: Proceedings of the 9th International Conference on Software Process Improvement (CIMPS 2020) (Vol. 1297, p. 169). Springer Nature.

Фалькевич В.Г., Лісняк А.О. МЕТОДОЛОГІЯ ІНВАЛІДАЦІЇ КЕШУ У МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ ВЕБ ДОДАТКІВ

У статті розглянуто наявні підходи, методи використання кешування в масштабованих веб-системах та рішення, запропоновані в супутніх роботах. Створено методологію інвалідації кешу в мікросервесній архітектурі. Об'єктом дослідження є процес створення методології оновлення кешу. Методи дослідження базуються на моделюванні процесів управління кеш-пам'яттю та взаємодії сервісів. Визначено вимоги до систем інвалідації кешу: розподіл зобов'язаностей, інвалідація лише частини кешу, запобігання повторному оновленню кешу за часом, неблокуюча операція. Описано всі етапи взаємодії компонентів системи, що включають API шлюз, кеш-сервіс, кеш-пам'ять, сервіси додатку. Розглянуто приклади використання додатку та кеш-сервісу для підтримки актуальності кеш-системи. У статтю включено один рисунок зі схематичними компонентами архітектури додатку, дві діаграми послідовності з етапами взаємодії елементів системи у часі під час виконання та обробки запиту. Визначено основні принципи в архітектурі мікросервісів із кеш-системою: абстрагування інвалідації кешу від внутрішніх сервісів до кеш-сервісу шляхом створення схеми кешу для кожного сервісу; опис правил підготовки відповідей API; запит до кешу від служб лише через кеш-сервіс; використання API-шлюзу як маршрутизатора між сервісами; зміна схеми кешу не повинна впливати на роботу сервісу кешу. Досліджено базові необхідні правила для інвалідації кешу: схема кешу, схема відповіді (або обробник даних для підготовки відповіді), дія. Схема кешу описує об'єкт данних з поточного сервісу з наявного запиту, що зберігатиметься у кеш-системі; обробник відповіді або схема відповіді – це правила або готова до використання функція (після декодування) для підготовки відповіді до відправлення клієнту. Дія як тип команди, яку необхідно виконати при оновленні кешу сервісом після отримання відповіді (створити, оновити, видалити).

Ключові слова: *кеш, PASS, Redis, Memcashed, ElastiCache, Memorystore, API, інвалідація кешу, мікросервіс.*