International Journal of Computer Trends and Technology
ISSN: 2231-2803 / https://doi.org/10.14445/22312803/IJCTT-V68I8P102

Volume 68 Issue 8, 14-20, August 2020
© 2020 Seventh Sense Research Group®

Original Article

Caching in Web Applications

Shobhit Chauhan

Northwestern Mutual, United States of America

Received Date: 16 June 2020
Revised Date: 28 July 2020
Accepted Date: 31 July 2020

Abstract *- Last two decades have seen tremendous growth in Web-based applications. This growth, combined with advancement in web technologies, has fundamentally transformed user's expectations of services and products provided by commercial entities. Today's customer is flooded with choices quite literally at their fingertips. It's no surprise then that user experience in web applications is a key parameter that drives the technology initiatives in any company. One of the key components of user experience is the performance of the system. As page load time goes from one second to 10 seconds, the probability of a mobile site visitor bouncing increases by 123%.[1]. Businesses have only a few seconds to engage the customer online, or they run the risk of losing the business. Performance Engineering of any web application is therefore critical to the success of any business in today's digitized world. Within the landscape of performance engineering, caching holds a prominent, permanent and respectable position. Any web application aimed at improving the user experience implements caching in some shape or form. Caching improves the responsiveness of any website and thereby improves the user experience. This paper looks at the basics of caching and how caching can be implemented in today's technological landscape.*

Keywords *- Caching, Performance Engineering, Website Optimization, User Experience*

INTRODUCTION

Caching is the mechanism by which information is temporarily saved for future use. The information can be in myriad forms – images, text, audio files, video files, data and instructions in L1/L2 cache etc. Cache work on the principle of "locality of reference", particularly temporal locality. Temporal locality states that if the particular element has been asked for, then the same element will be asked for in the near future. When the system request information for the first time, the call is made to the origin server, which holds the master copy of the asset. This information is then sent back to the user. At the same time, this information is updated in an additional storage location closer to the user. This location is referred to as a cache. When systems request for the same information or resource again, a copy of the requested resource is served from the cache instead of making a call to the originating source. This has several advantages: faster availability of the resource, improvement in the usage of network bandwidth, decrease in the load on the origin server, improved uptime of the server, among others. At the same time, caching strategies need to be properly designed and configured. Resources should be cached only till the time they are valid. Stale information can be as bad as no information. Stale information can lead to a bad customer experience or, worse, crash the applications altogether. Technological processes are expected to be very agile. Companies perform multiple production deployments within a day, which can lead to changes in the content of a page multiple times within a day. In the absence of a sound caching strategy, these changes can break the page and adversely affect customer experience.

I. TERMINOLOGIES

Following terminologies will come in handy when understanding caching and its role in web applications

A. Client-Server

The Client-Server model is an architecture in which a central server tier is responsible for serving requests for resources from many clients. In the age of containerization and cloud computing, there are no central servers as such. An application spread across multiple instances in different parts of the world is considered a single server application tier as they behave as a single entity and perform the same functions.

B. Origin Server

The origin server is the location of the master copy of the content. The copy of resources on this server serves as the source of truth for all the tiers which cache the content. An origin server can be a single file, a folder containing multiple files or can be a set of servers together acting as a unified cache.

C. Private Cache

Private Cache is the Cache dedicated to a single entity. Entities can vary depending on the tier. For example: in the case of an end-user, a browser cache is considered as a private cache since it serves a single entity – browser instance; in the case of an application instance, an in-memory store for data or cache providers caching return value of methods can serve as a private cache for that instance.

D. Shared Cache

Shared Cache is the Cache that stores resources that can be used by multiple entities. For example, Content Delivery Networks or ISP created caches are shared across multiple users in the region covered by that CDN or ISP.

E. Cache Hit Ratio

The cache hit ratio is the key metric to measure the effectiveness of Cache. It is defined as the ratio of the number of requests served from Cache to the total number of requests made to the Cache. A high cache hit ratio means that a large number of requests are being served from Cache, which is a desirable outcome.

F. Cache replacement

All caches are essentially saved on a storage system and have finite capacity. Cache replacement is a feature through which resources in the Cache are replaced to make way for newer data.

Key criteria for consideration for cache eviction are:

- frequency – relies on how frequently is an item requested.
- Recency – relies on how recently was the item requested
- Frecency – this is a combination of frequency and recency.

The most popular algorithm for cache replacement is LRU (Least Recently Used). Adaptive Replacement Cache (ARC) is considered a better alternative to LRU and is gaining some traction in developer communities these days. It is considered to have better performance than LRU [2]. This is accomplished by keeping track of both frequently used and recently used pages plus a cache directory.

G. Hot, Warm and Cold Cache

Caches start in a cold state, which means that they are empty with no objects stored in them, resulting in cache misses. As Cache start receiving requests, it starts updating itself with cacheable objects. This state of Cache is termed a warm state. A cache is in a hot state when it has all cacheable objects stored and up-to-date.

H. Freshness

Whether an object can be served in response to a query is determined by its freshness. An object is returned to the client only when the object is within a freshness time frame - measured by time-to-live (TTL) value. Time-to-live defines the maximum time till which an object can be served to the client.

I. Cache Invalidation

Cache invalidation is the process of invalidating the contents in the Cache before the expiration of the content as determined by the caching policy. When the original content on the server is changed, all the copies of the content saved on different tiers need to be updated. Different tiers employ different mechanisms to update their caches.

On the Browser side. Asset names are appended with uuids. Whenever an asset needs to be invalidated in Cache, a new uuid is appended to the asset's name. The client application now calls the assets with updated uuid to get the latest copy.

Application-level caches which reside within the application instance is invalidated when the application instance restarts. In the case of an external cache, specific methods need to be written and invoked to clear the Cache. Cache providers provide clear() methods which can clear all or part of the Cache.

Oftentimes, application owners do not have control over proxy caches and DB plan caches. These are, therefore, difficult to be invalidated. Different strategies are used to work around these caches. For example, to bypass the plan query cache in the database, special inbuilt functions are used to force specific plans on a SQL query. Proxy caches are bypassed through the usage of the cache-control header as private. Private header ensures that assets can be cached only on the browser.

J. Cache Validation

Cache validation is the process of validating the resource in the Cache against the resource at the origin server. It ensures that the cached version is the latest version of the resource, which can be delivered to the end customer. Making a call to the origin server to get the items is an expensive operation. It costs network bandwidth, increases the load on the origin server and adds to latency in obtaining the assets. Latency will further increase as the size of assets increases. By performing cache validation of the resource, these costs can be reduced, and performance can be significantly improved. As cache validation is performed, the asset's expiration time gets updated as well. So, once cache validation is performed, the asset behaves just like original cached content, which can be directly delivered to the customer without any validation from the origin server. Example: Etags(Entity Tags) are often used by web applications as means to effective caching strategy and cache validation. When a cacheable asset is first requested by the client, a call is made to the origin server, an asset with Etag and expires header is delivered to the client and Cache is updated with the content.

a) If the client requests for this asset before the expiration of cached content, the asset is directly delivered from the Cache.

b) If the client requests for the asset after the expiration date, a call is made to the origin server with the assigned ETag to check if the asset has been modified. Assets are considered the same if the etag of the asset at the origin and etag of the asset in the cache match. In this case, http status code 304 (not modified) is returned by the server. If Etags do not match, then a new asset with the latest header information is sent to the client and Cache is updated with a new asset. This saves in latency and bandwidth since a lighter version of the call is made to the origin, and contents are downloaded only in case of a change.

K. *Distributed Cache*

Distributed Cache, as the name suggests, is the Cache that is distributed across multiple servers. This allows the Cache to grow in size horizontally. As more and more companies move towards the containerized platform and towards cloud computing, distributed Cache forms a key component of their architecture. One of the key considerations in containerization and cloud-based architecture is the transient nature of application instances. Containers can fail any time; new instances can get created anytime either to replace the failed instance or as part of auto-scaling. In either of the cases, Cache within the instance will either be destroyed with the instance or will be in a cold state and will require priming. This will result in Cache becoming ineffective. A persistent location is therefore required for the cache component of any web application. Distributed Cache fulfils this requirement. Another use case created by containerization and cloud computing is the preference for stateless instances. Applications need to find a way to maintain a state of various sessions. Distributed Cache helps with maintaining web session data. This allows for application containers to be stateless while the overall application can still create and maintain states for different sessions.

II. CACHING TIERS IN WEB APPLICATIONS

Any content in a web application passes through multiple tiers before reaching the end-user. Mechanisms exist to cache resources at each of these tiers. Most efficient caching strategies utilize most if not all the tiers to improve the performance of the application and provide a better user experience.

A. *Client or Browser cache*

The browser maintains a cache that stores cacheable assets on the hard drive of the user's system. This has many benefits. It helps with back/forward navigation, viewing the page at a later date without making the calls for the assets to the server again. It can help with offline browsing of the content as well. The application lets the browser know which assets can be cached and how to maintain them through various headers (Cache-control, expires, eTags etc.). The best candidates of caching are static assets, e.g. CSS files, javascript files, images, videos, fonts, SVG files, libraries (e.g. jquery) files, bootstrap files etc. In addition to static assets, dynamic assets can be cached as well. This requires careful consideration of the data being cached. Only data that is not expected to change for some significant amount of time should be cached. For example description of a product and its price is not expected to change within a day. Such data can be cached for a period of 24 hours. With the usage of Etags and UUIDs, cached content can be validated, and the Number of calls to origin for such dynamic content can be reduced further.

Browser caches are very powerful and can help improve the performance considerably since it is closest to the end-user and has the least latency. Through the use of various http headers and careful study of application behaviour, a robust caching strategy can be developed, which can lead to a great customer experience. Browser is one of the only caches on which end customer has some form of control. A user can delete the entire Cache or delete parts of Cache whenever the user wants.

B. *Proxy caches*

As the asset travels through the network, it passes through various proxy servers. These servers have the capabilities to cache the content as it passes through the servers. This content can then be served to a larger user group without making the call to origin servers. ISPs (Internet Service Providers) and large enterprises create and maintain these caching servers. A proxy behaves like both a client and a server. It acts like a server to clients and like a client to servers. A proxy receives and processes requests from clients, and then it forwards those requests to origin servers.[3]. These caches are out of the control of application owners and end-users. In addition to this, there is limited support available for cache control through http headers. Since these are out of the control of application owners, but they do exist, the impact of such caches need to be considered when developing a caching strategy. Consider an asset that gets modified at the origin and whose browser cache is invalidated, but proxy cache has not yet been invalidated. In this case, when a request is made for the asset from the browser, stale content can be served by the proxy server. The application should have a way of bypassing the proxy server in such cases. Oftentimes, creating an asset with a different name (by using UUID) is used to bypass the proxy cache.

C. *Content Delivery Networks*

Content Delivery Network is a network that consists of geographically distributed servers aimed at delivering the content to the user from the location closest to the user irrespective of the location of the origin server. Before IaaS (Infrastructure as a Service) became the norm, applications were hosted by central servers in data centres. The user base of the applications could be geographically distributed. So, a client running in London (Europe) could be asking for an asset that was hosted in Los Angeles (North America). This led to considerable latency issues. CDNs proved to be of great solution to this problem. CDNs act as trusted overlay networks that offer high-performance delivery of common Web objects, static data, and rich multimedia content by distributing content load among servers that are close to the clients. CDN benefits include reduced origin server load, reduced latency for end-users, and increased throughput [4]. Different CDN providers implement caching functionalities in different ways, but the most common of these is the usage of Edge Serving. Edge serving is where a CDN will provide a network of geographically distributed servers that, in theory, will reduce time to load by moving the serving of the content closer to the end-user. This is called edge serving because the serving of the content has been pushed to the edge of the networks [5].

With the rise of IaaS, hosting servers in single data centres have been replaced by instances serving content from different geographical locations. However, the majority of application hosting is still region-specific. So, CDN is still helpful. In fact, all major cloud providers provide CDN as an offering in their Product suite (AWS CloudFront, GCP Cloud CDN, Azure CDN).

D. Distributed In-Memory Caching systems

Distributed in-memory cache systems form another cache layer in front of the application layer. In typical cloud architecture, this tier can be placed between the API gateway and application layer. This tier can be used for caching data on behalf of the applications layer, e.g. results of a method call depending on query params. This Cache can be used by the application tier as additional cache storage storing results of method calls or the results of most frequently executed function calls. In addition to this, it is used to cache small-sized database tables, which serve as reference tables in applications, thus helping applications to avoid a call to the database reducing latency of the call, load on the database and improving throughput. This caching layer is also used for storing web session information which is shared across various application instances.

E. Application caching

All major application frameworks provide support for caching. For example, Nodejs provides node-cache and Spring framework provides an abstraction for caching that supports different cache providers. Each time a given method is invoked, the cache abstraction checks whether the method has been invoked with the same arguments. If the method has been executed earlier, the cached results are returned. If not, then the method is executed, results returned to the caller and cached. When the method is invoked again with the same values, results can be retrieved from the Cache instead of the application going through expensive IO (e.g. calls to external systems and database) or CPU (e.g. heavy computations) intensive calls every time[6]. Data access objects can also be cached in this way. This helps with a reduction in latency and better utilization of resources and add to cost optimization in cloud applications where charges are dependent on the number of resources used.

F. Database caching

Several strategies are employed for fast retrieval of results from databases – standard SQL optimization, indexing etc. Database engines have an inbuilt mechanism that use various algorithms to improve the performance of query plans. With metadata collected through the execution of query plans, database engines optimize the query plans and cache these plans for future query executions. However, response times will still be limited by response times of disk IO and network latency. In addition to this, oftentimes, the database is a central resource (there are distributed databases available as well, but those are still not widely used) which is accessed by multiple instances of the application, and as such, it is important that load on database servers should be reduced as much as possible to provide high availability. Database caching is a combination of multiple strategies

- Caching at the application tier of various data access objects and data transfer objects.
- Caching of data in in-memory data caches
- Integrated Cache inbuilt in database engines, e.g. MySQL has global query cache, MSSQL has Cursor Library cache, and Amazon Aurora offers integrated Cache managed within the database engine. Each engine comes with its own unique features and implementation, but the idea is to reduce the workload on the database, improve latency and minimize calls to the database for frequently accessed data.

III. DEVELOPING A CACHING STRATEGY

A. Identifying assets to be cached

The caching strategy should consider the following with respect to assets:

a) Usage of asset

Caches are finite resources and should be used judiciously. A good candidate for a cacheable asset is one that is read frequently and updated infrequently. Usage of an asset determines how frequently an asset is called by clients. Her clients refer to the entire user base of the applications. Consider 2 pages of a website – one page gets a large volume of traffic (e.g. Home page), and the second page which receives significantly low traffic (e.g. Press Release page). Any asset which is part of the first page is a better candidate for caching than the second page because it will be called more and will provide benefits of caching to a larger user base.

b) Frequency of change in the content of the asset

Stale data can be almost as bad as no data if a user takes actions based on the content of the data. It's imperative that the latest data is available for the user. For example: If a user plans to use a financial product based on the interest rate provided by the institution, then the application should ensure that only currently valid data is provided to the end-user. If data becomes stale frequently, then it is better to make the call to the origin than to cache it and run the risk of serving stale data. Applications should also consider backward compatibility when dealing with cacheable data. A new version of the application which cannot handle the old data format can cause a bad user experience or, worse, crash the application when presented with cache data. Applications should always consider the effects of new versions with older data formats and should have built-in resilience against such failures.

c) Size of assets

Several choices need to be made when considering caching and the size of the assets. When a large number of small-sized assets are cached, it will lead to less number of

calls to the origin and free up the resources (threads, connections etc.) both on the client-side as well as on the server-side. At the same time, a small number of large-sized assets are better candidates for caching if they take a long time at the origin to process. For example, consider an application aggregating and processing the large volume of monthly data of all employees in a particular region and which returns a large-sized response. Here overall latency of downloading the asset will include the time to process the request by a server application and the time to transfer all the packets of the response. Packets count and consequently time to transfer the contents will increase as the size of the asset increases. If the client serves the content from Cache instead of making a call to the origin, it will save bandwidth, free up resources on client and server, and result in a reduction of latency considerably.
Study of the usage patterns and comprehensive performance testing can provide data points which can then help with comparison of cost-to-benefit analysis of such conflicting cases.

d) Type of asset

Static assets are considered the best candidates for caching provided they perform well on other indices – size, usage and change frequency. Examples of static assets can be CSS files, javascript files, logos, fonts, images etc. which are used for rendering the static potion of web pages. On the server-side, applications logic often looks up various tables for reference data, e.g. Product Type Ids, Product sizes. Reference data that is not expected to be changed frequently but is used by the application for executing methods/functions frequently are good candidates for caching at the application tier or at in-memory caches. If reference data spans across multiple tables, then a method can be written to fetch the data from various tables. Results of this method call can be cached. Whenever the method is invoked next time, it will check the Cache first to provide the response. This will reduce the load on the database and reduce overall latency since network latency (from call to the database), disk latency (from disk IO to get the results), and query latency (time taken to query the response) has been replaced by a single call to an in-memory cache.
Careful analysis needs to be performed before caching dynamic assets. The following questions should be answered before caching dynamic contents:

- How frequently does the content of the asset change?
- What will be the impact if the user is presented with stale data? Will it break any calls on the current page or subsequent pages? Are the safeguards in place to avoid users getting served with stale data?
- Is the data critical for the user to take the next steps on the site?
- Can HTTP headers (Etags, cache-control: no-cache, max-age=0) be used to validate the contents?

B. Identifying tiers to be used

Any data passes through multiple tiers before it is delivered to the end-user. Each of these tiers has the capability to cache data. A sound caching strategy considers all tiers and uses them depending on their strengths. Two key ideas related to tiers should be kept in mind:

- Closer the tier is to the end-user, the more beneficial it is.
For example: if a call is triggered from a browser and is satisfied by the browser cache, this will provide the maximum benefit of caching.
- Farther the tier from application owner, less control application owner has on the Cache For example, an application owner has more control over the application tier than on the browser cache.
If an asset changes within the expiration period, the application owner can clear the entire Cache on application instances (closer to the application owner). But on the browser (farther from the application owner), Cache cannot be cleared (less control). The owner will need to use the invalidation process so that the user uses content other than the one already cached.

C. Identifying expiration criteria, validation and invalidation strategies

Business priorities change, technologies change, deployment does not go as planned, all or parts of the code may need to be rolled back, unplanned changes happen, content becomes out-of-date/invalid before it was expected to be. All of these are commonplace in IT systems. Applications should always consider that such surprises can happen and, as such, should have the agility and ability to handle them effectively, oftentimes, on very short notice. Every cacheable asset should have an expiration date or expiration criteria.
The caching strategy should have a process of validating and invalidating every cached asset on every tier. In the absence of such safeguards with no ability to control behaviour, applications can run unpredictably when changes happen.
Additionally, careful attention needs to be paid when working with caching and HTTP headers.
For example: Consider an application that catches the username or account Id or product preferences of the user. But the application doesn't set the cache-control to private. This information can now be cached on the proxy server. If now some other user uses the same application and same proxy, this user will be provided with the personal information (username or account Id or product preferences) of the first user. Besides this being a huge security concern, it is a terrible user experience for a second user who is unable to access his own account. This problem is further exacerbated by the fact that proxy cache is mostly out of the control of both end-user and application owner.

It is imperative that every asset has expiration criteria and mechanisms exist for validation and invalidation. The validation approach can be synchronous or asynchronous. Synchronous validation of an object is done at object request time. With asynchronous validation, the client periodically checks its cached objects, identifies those objects that require validation, and validates them proactively without waiting for a request [3].

D. General considerations

Caching helps once the data is available in the Cache. But when the user calls for the asset the first time, there will be some delay since data is not available in the Cache yet. In addition to this, shared caches like proxy caches or CDN can be region-specific. This means that unless a user within a region doesn't call for an asset, it won't be available for other users in the region. This leads to a situation where it takes some time before shared Cache is able to help a larger user base with cache benefits. However, with carefully designed applications, some of the data can be made available to the user instantaneously. This can be done by using some or all of the following techniques:

a) Priming the Cache

Instead of waiting for a call to trigger the process of caching, the application can pre-emptively build up the Cache. This priming is applicable to tiers under control of the application owner – database caching, application caching, distributed Cache. Application logic is created, which calls all the methods whose response needs to be cached. This logic is triggered during the start-up of the application. This way, some data is available in the Cache even before any user request has reached the application instance. This is termed priming the Cache.

However, priming the Cache is not without a catch. When priming the Cache, the time to start the application and make it available for processing user requests will increase. This will happen because start-up has a new step of priming the Cache now, which in itself may be composed of multiple calls to origin (database, files etc.). If left unchecked, this time can be considerably high, which leads to a high start-up time of application.

This can further affect the high availability and elasticity of the system adversely. One of the ways applications achieve high availability and elasticity is through auto-scaling. Autoscaling is triggered when an application needs additional instances to process the requests. Auto-scaling, therefore, sets up the instances, starts the application and make them available to the pool for processing of requests. An increase in start-up time will add to a delay in instances being available for processing of users' requests. This is an undesirable outcome, particularly when the system needs additional instances quickly. Comprehensive testing of the start-up process and the auto-scaling process is required to ascertain the correct amount of priming, which helps meet the SLAs.

b) Server Push

Server Push is a feature available to HTTP/2 compatible clients and servers. It is one of the key features coming out of HTTP/2 and is targeted towards better performance. Through Server Push, the server can send the files to the client even before the client has requested those files. For example, consider a client requests for a Product page from a retail website. The server sends the base HTML back to the client. The client needs to parse through the HTML to find what other assets it needs to request in order to render the page, which will take some time. But server application is aware of the essential assets required to render the page. The server can, therefore, pre-emptively and asynchronously send the required assets to the client. When the client actually parses the base HTML and calls for the asset, it is already available for the user, and the browser can begin rendering the page faster.

Server push can be used to send assets not only of the current page but also of subsequent pages that the user is likely to visit. For example: If a user has clicked in for the Login page, the user will most probably sign in and will be taken to the account landing page. Server push can provide assets both for the Login page and the account landing page.

IV. CONCLUSION

Caching forms the cornerstone of every type of architectural implementation- monolith or microservices, on-prem or cloud, front-end heavy or backend services, region-specific or global.

It's one of the most efficient and time-tested tools in performance engineering.

It helps businesses to provide good user experiences even with complex functionalities. And it helps engineering teams to avoid premature optimization when working with complex applications.

Web applications have undergone a huge transformation in the last two decades. It is the primary way in which the majority of people communicate with their environment now - services, products, social circles, information etc. Users have high expectations from these applications both in terms of features as well as performance. Software applications are becoming increasingly more complex in order to provide the user with richer and newer experiences. These factors together throw in great challenges towards technology to provide great experiences without losing out on performance. Caching in web applications is one of the many ways in which technology has been able to meet such challenges.

REFERENCES

[1] DanielAn, https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/, February 2018

[2] N. Megiddo and D. S. Modha, Outperforming LRU with an adaptive replacement cache algorithm, in *Computer*, 37(4) (2004)58-65. doi: 10.1109/MC.2004.1297303.

[3] Duane Wessels, Web Caching. O'Reilly Media Inc, (2001).

[4] A. Vakali and G. Pallis, Content delivery networks: status and trends," in IEEE Internet Computing, 7(6)(2003)68-74. doi: 10.1109/MIC.2003.1250586.

[5] Tom Barker, Intelligent Caching, O'Reilly Media Inc. 2017

[6] https://docs.spring.io/spring/docs/current/spring framework reference/integration.html#cache

[7] Oliver Spatscheck, Michael Rabinovich, Web Caching and Replication, Addison-Wesley Professional, 2001.

[8] Banerjee Krishnadas, Das Anupam, Dhawa Sagar Kumar, Dr Roy Chowdhury Dilip, Dr Das Anirban Application of Nature-Inspired Algorithms for Decision Making of Small Businessmen International Journal of Engineering Trends and Technology 66(3) (2018)129-132.