**A beginner’s guide to Cache synchronization strategies**

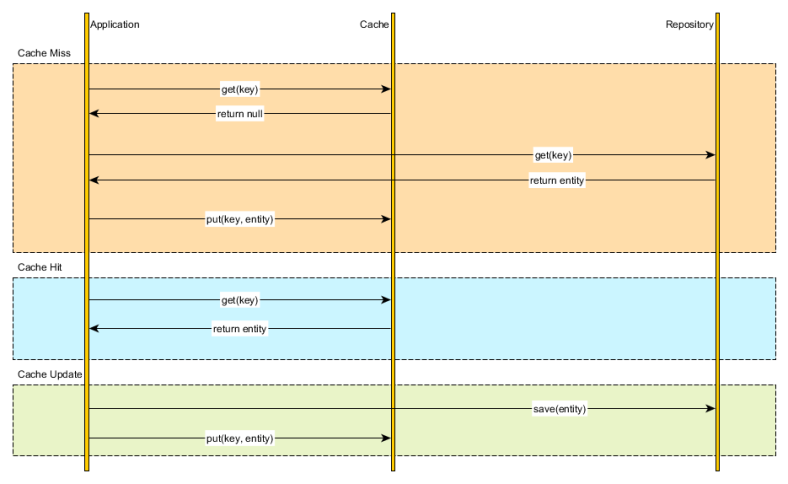
**Introduction**

A [system of record](http://en.wikipedia.org/wiki/System_of_record) is the authoritative data source when information is scattered among various data providers. When we introduce a [caching solution](https://vladmihalcea.com/2015/04/16/things-to-consider-before-jumping-to-enterprise-caching/), we automatically duplicate our data. To avoid inconsistent reads and data integrity issues, it’s very important to synchronize the database and the cache (whenever a change occurs in the system).

There are various ways to keep the cache and the underlying database in sync and this article will present some of the most common cache synchronization strategies.

**Cache-aside (**Repository is update first then cache**)**

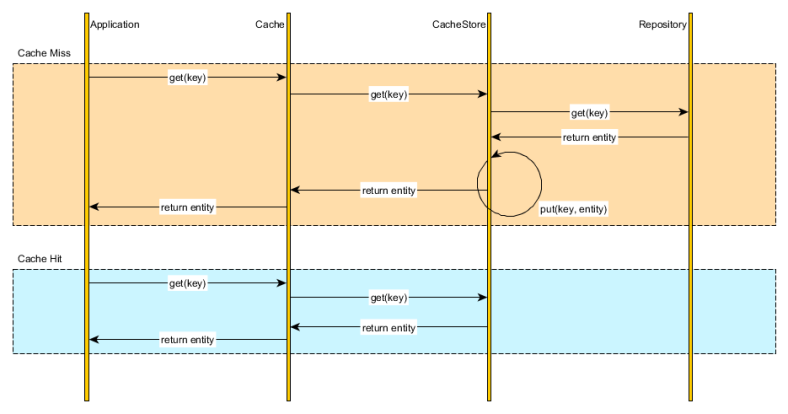
The application code can manually manage both the database and the cache information. The application logic inspects the cache before hitting the database and it updates the cache after any database modification.

[](https://vladmihalcea.files.wordpress.com/2015/04/cacheaside.png)

Mixing caching management and application is not very appealing, especially if we have to repeat these steps in every data retrieval method. Leveraging an [Aspect-Oriented caching interceptor](https://vladmihalcea.com/2014/12/01/spring-request-level-memoization/) can mitigate the cache leaking into the application code, but it doesn’t exonerate us from making sure that both the database and the cache are properly synchronized.

**Read-through**

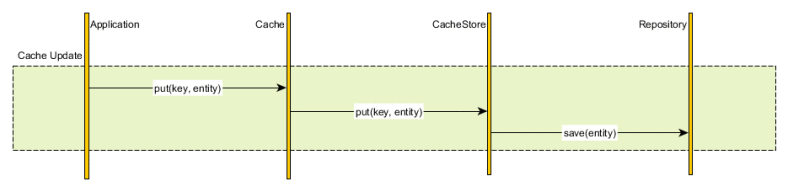
Instead of managing both the database and the cache, we can simply delegate the database synchronization to the cache provider. All data interactions is therefore done through the cache abstraction layer.

[](https://vladmihalcea.files.wordpress.com/2015/04/cachereadthrough.png)

Upon fetching a cache entry, the Cache verifies the cached element availability and loads the underlying resource on our behalf. The application uses the cache as the *system of record* and the cache is able to auto-populate on demand.

**Write-through (This happen synchronously)**

Analogous to the *read-through* data fetching strategy, the cache can update the underlying database every time a cache entry is changed.

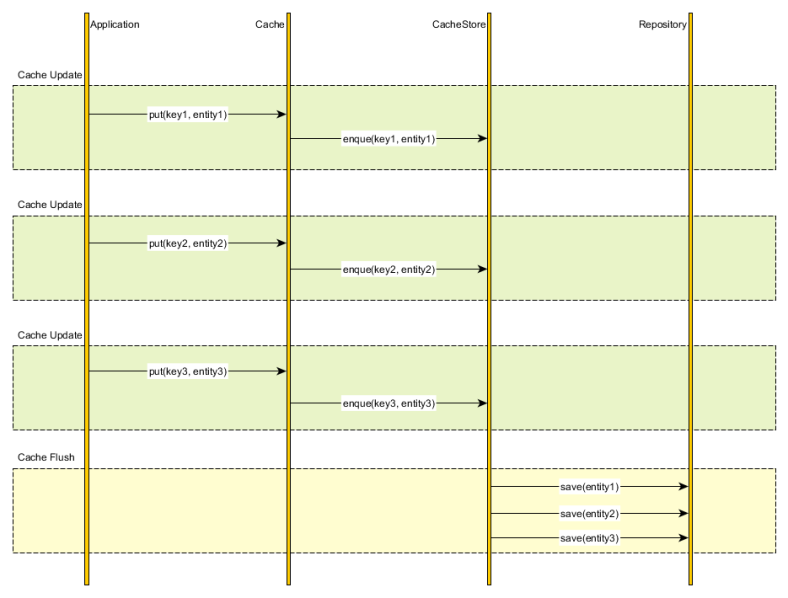
[](https://vladmihalcea.files.wordpress.com/2015/04/cachewritethrough.png)

Although the database and the cache are updated synchronously, we have the liberty of choosing the transaction boundaries according to our current business requirements.

* If strong consistency is mandatory and the cache provider offers an [XAResource](http://docs.oracle.com/javaee/7/api/javax/transaction/xa/XAResource.html) we can then enlist the cache and the database in the same global transaction. The database and the cache are therefore updated in a [single atomic unit-of-work](https://vladmihalcea.com/2014/01/05/a-beginners-guide-to-acid-and-database-transactions/)
* If consistency can be weaken, we can update the cache and the database sequentially, without using a global transaction. Usually the cache is changed first and if the database update fails, the cache can use a compensating action to roll-back the current transaction changes

**Write-behind caching (This happen asynchronously)**

If strong consistency is not mandated, we can simply enqueue the cache changes and periodically flush them to the database.

[](https://vladmihalcea.files.wordpress.com/2015/04/cachewritebehind.png)

This strategy is employed by the [Java Persistence](https://vladmihalcea.com/2014/08/07/a-beginners-guide-to-jpahibernate-flush-strategies/)[*EntityManager*](http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html) (first-level cache), all [entity state transitions](https://vladmihalcea.com/2014/07/30/a-beginners-guide-to-jpahibernate-entity-state-transitions/) being flushed towards the end of the current running transaction (or when a query is issued)

*Update repository first then cache 🡪 Cache Aside (Application has the responsibility to keep the cache up-to-date)*

*Update cache first then repository 🡪 Write Behind (done asynchronously. Cache has the responsibility to update the cache)  
  
Update simultaneously both at cache & repository 🡪 Write Through (Cache store has the responsibility to perform write on the database when there is a write or there is cache miss to update itself)*

<https://vladmihalcea.com/2015/04/20/a-beginners-guide-to-cache-synchronization-strategies/>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Cache-Aside pattern

## **Context and problem**

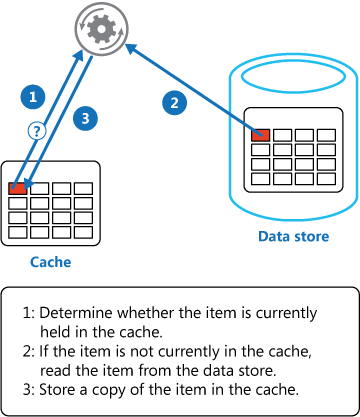
Applications use a cache to improve repeated access to information held in a data store. However, it's impractical to expect that cached data will always be completely consistent with the data in the data store. Applications should implement a strategy that helps to ensure that the data in the cache is as up-to-date as possible, but can also detect and handle situations that arise when the data in the cache has become stale.

## **Solution**

Many commercial caching systems provide read-through and write-through/write-behind operations. In these systems, an application retrieves data by referencing the cache. If the data isn't in the cache, it's retrieved from the data store and added to the cache. Any modifications to data held in the cache are automatically written back to the data store as well.

For caches that don't provide this functionality, it's the responsibility of the applications that use the cache to maintain the data.

An application can emulate the functionality of read-through caching by implementing the cache-aside strategy. This strategy loads data into the cache on demand. The figure illustrates using the Cache-Aside pattern to store data in the cache.



If an application updates information, it can follow the write-through strategy by making the modification to the data store, and by invalidating the corresponding item in the cache.

When the item is next required, using the cache-aside strategy will cause the updated data to be retrieved from the data store and added back into the cache.

## **Issues and considerations**

Consider the following points when deciding how to implement this pattern:

**Lifetime of cached data**. Many caches implement an expiration policy that invalidates data and removes it from the cache if it's not accessed for a specified period. For cache-aside to be effective, ensure that the expiration policy matches the pattern of access for applications that use the data. Don't make the expiration period too short because this can cause applications to continually retrieve data from the data store and add it to the cache. Similarly, don't make the expiration period so long that the cached data is likely to become stale. Remember that caching is most effective for relatively static data, or data that is read frequently.

**Evicting data**. Most caches have a limited size compared to the data store where the data originates, and they'll evict data if necessary. Most caches adopt a least-recently-used policy for selecting items to evict, but this might be customizable. Configure the global expiration property and other properties of the cache, and the expiration property of each cached item, to ensure that the cache is cost effective. It isn't always appropriate to apply a global eviction policy to every item in the cache. For example, if a cached item is very expensive to retrieve from the data store, it can be beneficial to keep this item in the cache at the expense of more frequently accessed but less costly items.

**Priming the cache**. Many solutions prepopulate the cache with the data that an application is likely to need as part of the startup processing. The Cache-Aside pattern can still be useful if some of this data expires or is evicted.

**Consistency**. Implementing the Cache-Aside pattern doesn't guarantee consistency between the data store and the cache. An item in the data store can be changed at any time by an external process, and this change might not be reflected in the cache until the next time the item is loaded. In a system that replicates data across data stores, this problem can become serious if synchronization occurs frequently.

**Local (in-memory) caching**. A cache could be local to an application instance and stored in-memory. Cache-aside can be useful in this environment if an application repeatedly accesses the same data. However, a local cache is private and so different application instances could each have a copy of the same cached data. This data could quickly become inconsistent between caches, so it might be necessary to expire data held in a private cache and refresh it more frequently. In these scenarios, consider investigating the use of a shared or a distributed caching mechanism.

## **When to use this pattern**

Use this pattern when:

* A cache doesn't provide native read-through and write-through operations.
* Resource demand is unpredictable. This pattern enables applications to load data on demand. It makes no assumptions about which data an application will require in advance.

This pattern might not be suitable:

* When the cached data set is static. If the data will fit into the available cache space, prime the cache with the data on startup and apply a policy that prevents the data from expiring.
* For caching session state information in a web application hosted in a web farm. In this environment, you should avoid introducing dependencies based on client-server affinity

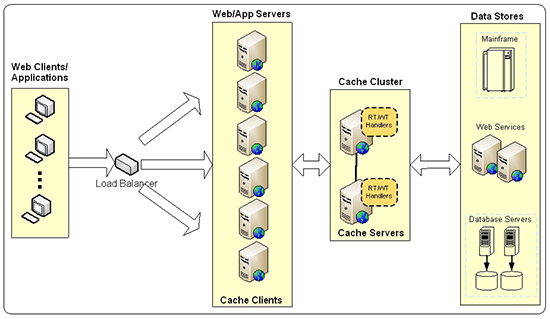
<https://docs.microsoft.com/en-us/azure/architecture/patterns/cache-aside>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Using Read-through & Write-through in Distributed Cache**

There are two main ways people use a distributed cache:

* **Cache-aside:** This is where application is responsible for reading and writing from the database and the cache doesn't interact with the database at all. The cache is "kept aside" as a faster and more scalable in-memory data store. The application checks the cache before reading anything from the database. And, the application updates the cache after making any updates to the database. This way, the application ensures that the cache is kept synchronized with the database.
* **Read-through/Write-through (RT/WT):** This is where the application treats cache as the main data store and reads data from it and writes data to it. The cache is responsible for reading and writing this data to the database, thereby relieving the application of this responsibility



##### Figure 1: Read-through/Write-through caching architecture.

### Benefits of Read-through & Write-through over Cache-aside

Cache-aside is a very powerful technique and allows you to issue complex database queries involving joins and nested queries and manipulate data any way you want. Despite that, Read-through/Write-through has various advantages over cache-aside as mentioned below:

* **Simplify application code:** In the cache-aside approach, your application code continues to have complexity and direct dependence on the database and even code duplication if multiple applications are dealing with the same data. Read-through/Write-through moves some of the data access code from your applications to the caching-tier. This dramatically simplifies your applications and abstracts away the database even more clearly.
* **Better read scalability with Read-through:** There are many situations where a cache-item expires and multiple parallel user threads end up hitting the database. Multiplying this with millions of cached-items and thousands of parallel user requests, the load on the database becomes noticeably higher. But, Read-through keeps cache-item in the cache while it is fetching the latest copy of it from the database. It then updates the cache-item. The end result is that the application never goes to the database for these cache-items and the database load is kept to the minimum.
* **Auto-refresh cache on expiration:** Read-through allows the cache to automatically reload an object from the database when it expires. This means that your application does not have to hit the database in peak hours because the latest data is always in the cache
* **Auto-refresh cache on database changes:** Read-through allows the cache to automatically reload an object from the database when its corresponding data changes in the database. This means that the cache is always fresh and your application does not have to hit the database in peak hours because the latest data is always in the cache
* **Better write performance with Write-behind:** In cache-aside, application updates the database directly synchronously. Whereas, a Write- behind lets your application quickly update the cache and return. Then it lets the cache update the database in the background.
* **Better database scalability with Write-behind:** With Write-behind, you can specify throttling limits so the database writes are not performed as fast as the cache updates and therefore the pressure on the database is not much. Additionally, you can schedule the database writes to occur during off-peak hours, again to minimize pressure.
* **Read-through/Write-through** is not intended to be used for all data access in your application. It is best suited for situations where you're either reading individual rows from the database or reading data that can directly map to an individual cache-item. It is also ideal for reference data that is meant to be kept in the cache for frequent reads even though this data changes periodically

<http://www.alachisoft.com/resources/articles/readthru-writethru-writebehind.html>