Application-level Caching Patterns Catalog

Jhonny Mertz and Ingrid Nunes

Our qualitative study allowed us to understand how developers deal with application-level caching in their applications, by explaining design, implementation and maintenance choices. Our findings and observations were used as foundation to the provision of practical guidance for developers with respect to caching. Based on our study, we derived caching patterns, which can be used by developers to help them design, implement and manage cache. Caching patterns are classified into categories, explained below. The proposed patterns are summarized in Table 1.

Design. Support to design decisions associated with application-level caching.

Implementation. Support to implementation issues of application-level caching, by providing solutions and guidance at the code level.

Maintenance. Support to performance analysis and improvement of application-level caching.

Table 1: Caching Pattern Classification.

Pattern	**Classification**	**Intent**
Asynchronous Loading	Implementation	Design a mediator to asynchronously deal with caching.
Cacheability	Design	Provide an intuitive process to decide whether to cache or not particular data.
Data Expiration	Design and Maintenance	Given cacheable content, provide an intuitive process to choose a consistency management approach based on data specificities.
Name Assignment	Implementation	Ensure a unique key and keep track of the content cached.

Next, we present in detail our patterns and their components, which comprise a template for a caching pattern catalog. These components are (i) a *classification*, (ii) the pattern *intent*, (iii) the *problem* involved, (iv) the *solution* proposed, and (v) an *example*.

Contents

1 Asynchronous Loading

Classification: Implementation

Intent

Design a mediator to asynchronously deal with caching.

Synchronous caching operations can affect client-response time and blocks the request processing thread.

Problem Under certain circumstances the cost of populating the cache is very expensive (data provided by third-party systems via web services or the result of a heavy calculation process, and others). In such a scenario whenever the cache is invalidated (by automatic expiration or invalidation) the request processing is blocked while getting fresh data, which affects client-response time.

Solution Load the cache asynchronously with a separate thread or by using a batch process.

Rules of thumb

- At initial population of the cache, mainly for large caches, load the cache asynchronously with a separate thread or by using a batch process.
- At runtime, when the cache is invalidated, repopulate it in a background thread and then hide the cost of data retrieval to the end-users.
- It is strongly recommended the use of a third-party library or frameworks which already provides cache basic operations in an async way. There are options for the most of programming languages and cache providers.

Example The method in the following code example shows an implementation of the Cache-aside pattern based on asynchronous processing. An object is identified by using an ID as the key. The `asyncGet` method uses this key and attempts to retrieve an item from the cache. If a matching item is found, it is returned. If there is no match in the cache, it should retrieve the object from a data store, adds it to the cache, and then returns it (the code that actually retrieves the data from the data store has been omitted because it is data store dependent). Note that the load from cache has a timeout, in order to ensure that the cache interaction do not block the processing.

```
  // Get a cache client
  // it can be a trird-party library or an implemented module
  // this facade should provide at least get, set and remove methods
  Cache cache = Cache.getInstance();

  public List<Product> getProducts() {

      List<Product> products = null;
      Future<Object> f = (List<Product>) cache.asyncGet("products");
      try {
  // Try to get a value, for up to 5 seconds, and cancel if it
  // doesn't return
  products = f.get(5, TimeUnit.SECONDS);

      // throws expecting InterruptedException, ExecutionException
      // or TimeoutException
      } catch (Exception e) {
    // Since we don't need this, go ahead and cancel the operation.
    // This is not strictly necessary, but it'll save some work on
    // the server.  It is okay to cancel it if running.
    f.cancel(true);
    // Do other timeout related stuff
      }

      if (products == null) {
    products = getProductsFromDB();

    // updates into cache should not block the request
    // return the user request as soon as possible
    cache.asyncSet("products", products);
      }

      return products;
  }
```

```
public Product getProduct(String id) {

  Product product = null;
  Future<Object> f = (Product) cache.asyncGet("product" + id);
  try {
    product = f.get(5, TimeUnit.SECONDS);
  } catch (Exception e) {
      f.cancel(true);
  }

  if (products == null) {
      product = getProductFromDB(id);

      // updates into cache should not block the request
      // return the user request as soon as possible
      cache.asyncSet("product" + id, products);
  }

  return product;
}
```

The code below demonstrates how to invalidate an object in the cache when the value is changed by the application. The code updates the original data store and then removes the cached item from the cache by calling the `asyncDelete` method, specifying the key.

The order of the steps in this sequence is important. If the item is removed before the cache is updated, there is a small window of opportunity for a client application to fetch the data (because it is not found in the cache) before the item in the data store has been changed, resulting in the cache containing stale data.

```
public void updateProduct(Product product) {
  updateProductIntoDB(product);
  cache.asyncDelete("products");

  //optionally, it is possible to update the data into cache
  cache.asyncSet("product" + id, product);
}

public void deleteProduct(String id) {
  deleteProductFromDB(id);
  cache.asyncDelete("products");
  cache.asyncDelete("product" + id);
}
```

2 Cacheability

Classification: Design

Intent Provide an intuitive process to decide whether to cache or not particular data.

Problem Cache has limited size, so it is important to use the available space to cache data that maximizes the benefits provided to the application. Otherwise, it can end up reducing application performance instead of improving it, consuming more cache memory and at the same time suffering from cache misses, where the data is not getting served from cache but is fetched from the source.

Solution Even though there are many criteria that contribute for identifying the level of data cacheability, there is a subset that would confirm this decision regardless of the values of the other criteria. Changeability is the first criterion that should be analyzed while selecting cacheable data, then usage frequency, shareability, computation complexity, and cache properties should be considered.

Figure 1 expresses a flowchart of the reasoning process to decide whether to cache data, based on the observation of data and cache properties. All criteria are tightly related to the application specificities and should be specified by the developer.

Rules of thumb

(a) Despite being frequently used, user-specific data are not shareable and may not bring the benefit of caching, being usually avoided by developers. In this case, a specific session component is used to keep and retrieve user sessions.

(b) If the data changes frequently, it should not be immediately discarded from cache. An evaluation of the performance benefits of caching against the cost of building the cache should be done. Caching frequently changing data can provide benefits if slightly stale data is allowed.

(c) Expensive spots (when much processing is required to retrieve or create data) are bottlenecks that directly affect application performance and should be cached, even though it can increase complexity and responsibilities to deal with. Methods with high latency or that consists of a large call stack are some examples of this situation and opportunities for caching.

In addition, we list content properties that should be avoided, which do not convey the influence factors in a good way and lead to problems such as cache trashing.

(a) User-specific data. Avoid caching content that varies depending on the particularities of the request, unless weak consistency is acceptable. Otherwise, the cache can end up being fulfilled with small and less beneficial objects. As result, the caching component achieves its maximum capacity earlier and is flushed or replaced many times in a brief period, which is cache thrashing.

(b) Highly time-sensitive data. Content that changes more than is used should not be cached given that it will not take advantage from caching. The cost of implementing and designing an efficient consistency policy may not be compensate.

(c) Large-sized objects. Unless the size of the cache is large enough, do not cache large objects, it will probably result in a cache trashing problem, where the caching component is flushed or replaced many times in a short period.

Example We list some typical scenarios where data should be cached and also give explanations based on the criteria presented.

(a) Headlines. In most cases, headlines are shared by multiple users and updated infrequently.

(b) Dashboards. Usually, much data need to be gathered across several application modules and manipulated to build a summarized information about the application.

(c) Catalogs. Catalogs need to be updated at specific intervals, are shared across the application, and manipulated before sending the content to the client.

(d) Metadata/configuration. Settings that do not frequently change, such as country/state lists, external resource addresses, logic/branching settings and tax definitions.

(e) Historical datasets for reports. Costly to retrieve or create and does not need to change frequently.

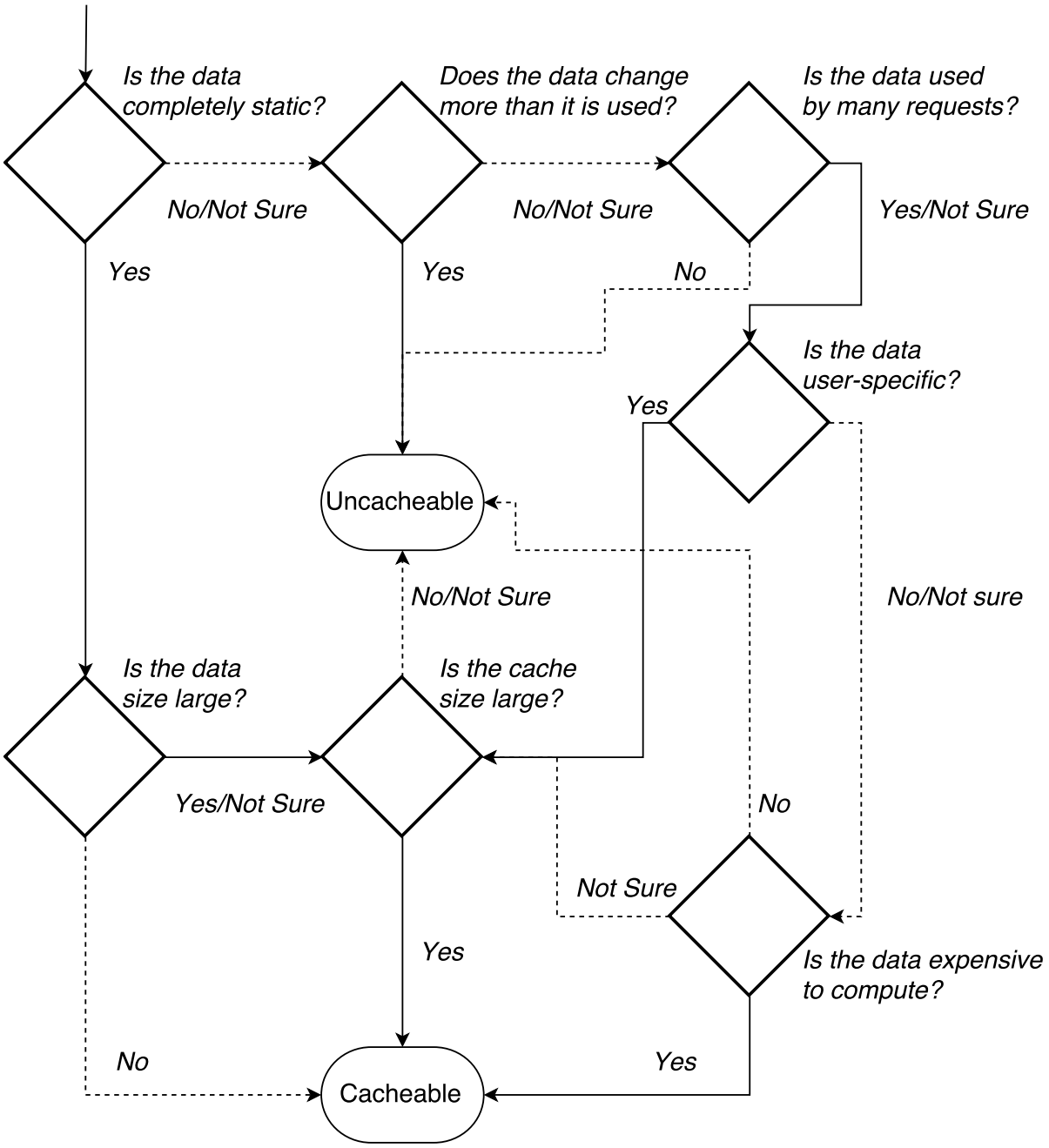

Figure 1: Cacheability Flowchart.

3 Data Expiration

Classification: Design and Maintenance

Intent Given the cacheable content, provide an intuitive process to decide a consistency management approach based on data specificities.

Problem It is usually 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 up to date as far as possible, but can also detect and handle situations that arise when the data in the cache has become stale. An inappropriate expiration policy may result in frequent invalidation of the cached data, which negates the benefits of caching.

Solution Every piece of cached data is already potentially stale, and a good trade-off between performance benefits and cost of invalidation approaches should be achieved. Its necessary to determine the appropriate time interval to refresh data, and design a notification process to indicate that the cache needs refreshing. If the data is held too long, it runs the risk of using stale data, and if it was expired too frequently, it could affect performance.

Deciding on the expiration algorithm that is right for the scenario includes the following possibilities:

- Heuristic-based. Traditional algorithms such as least recently used (LRU) and least frequently used (LFU) can be used.
- Absolute expiration after a fixed interval. Expiration based on a pre-defined time-to-live (TTL), applied to every content.
- Invalidation. Caching expiration based on a change in an external dependency, such as modifications in the data by users actions.
- Flushing. Cleaning up the cache if a resource threshold (such as a memory limit) is reached.

Figure 2 expresses a flowchart with the reasoning process to decide the appropriate consistency approach, based on observation of data properties. Changeability is the first property that should be analyzed while deciding, then staleness level and the amount of operations and dependencies related to the data should be considered. All properties are tightly related to the application specificities and should be defined by developer.

Rules of thumb

- While deciding the best consistency approach, it is important to measure the staleness degree and the lifetime of cached data.
- Frequently changed data is easily managed when associated with a TTL.
- Infrequently changed data provide more benefits when cached for long periods, thus manual invalidations or replacement are recommended.
- Determining how often is the cached information allowed to be wrong and work with weak consistency can be easier than defining a hard-to-maintain invalidation process. However, if the expiration period is too short, objects will expire too quickly, and it will reduce the benefits of using the cache. On the other hand, if the expiration period is too long, it risks the data becoming stale.
- Do not give all your keys the same TTLs, so they do not all expire at the same time. Doing this ensures that you do not get spikes of requests trying to make requests to your database because the cache keys have expired simultaneously.
- If the lifetime is dependent on how frequently the data is used, traditional heuristic-based eviction policies are right choices.
- If you frequently expire the cache to keep in synchronization with the rapidly changing data, you might end up using more system resources such as CPU, memory, and network.

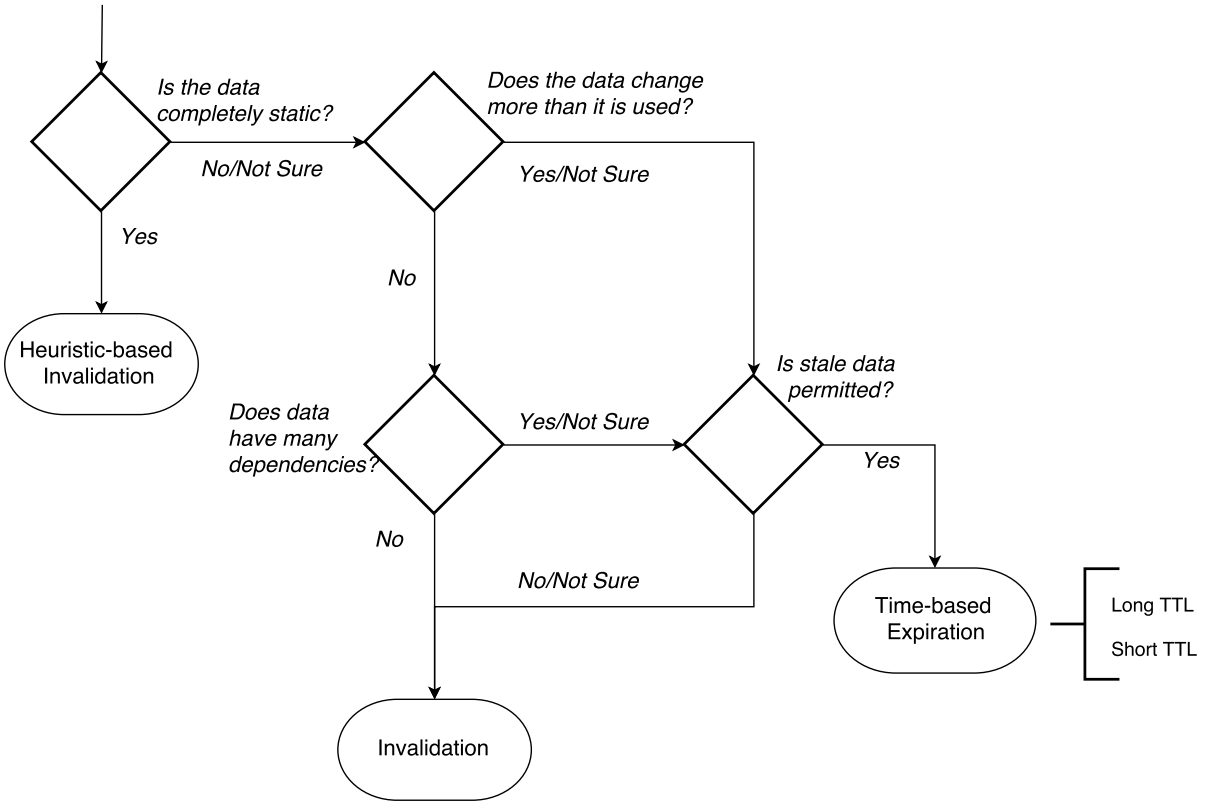

Figure 2: Data Expiration Flowchart.

- If the data does change frequently, you should evaluate the acceptable time limit during which stale data can be served to the user.
- Even if the data is quite volatile and changes, for example, every two minutes, the application can still take advantage from caching. For instance, if 20 clients are requesting the same data in a 2-minute interval, it is saving at least 20 round trips to the server by caching the data.
- Do not 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, do not make the expiration period so long that the cached data is likely to become stale.
- Most caches adopt a LRU policy for selecting items to evict, but this may be customizable. Configure the global expiration property and other properties of the cache, and the expiration property of each cached item, to help ensure that the cache is cost effective. It may not always be 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 may be beneficial to retain this item in cache at the expense of more frequently accessed but less costly items.
- Cache services typically evict data on a LRU basis, but you can usually override this policy and prevent items from being evicted. However, if you adopt this approach, you risk your cache exceeding the memory that it has available, and an application that attempts to add an item to the cache will fail with an exception.

Example Consider a stock ticker, which shows the stock quotes. Although the stock rates are continuously updated, the stock ticker can safely be removed or even updated after a fixed time interval of some minutes.

4 Name Assignment

Classification: Implementation

Intent

- Ensure a unique key.
- Keep track of the content cached.

Problem Keys are important to keep track of the content cached while debugging or when it is necessary to invalidate and delete stale data from cache, in the case of changes in the source of information.

Solution When choosing a cache key, you should ensure that it is unique to the object being cached, and that it appropriately varies by any contextual values.

Rules of thumb:

- A unique key can be simple strings or more complex types like hashes, lists, or sets. The content can be identified by method signature and individual identification. Moreover, a tag-based identification should be used to group related content.
- If the object being cached relies on the current user (perhaps via HttpContext.Current.User), then the cache key may include a variable that uniquely identifies that user.

Example The key can be scoped to its particular function area, and be formatted with varying parameters.

```
var key = string.Format("MyClass.MyMethod:{0}:{1}", myParam1, myParam2);
```

```
class ShopCore extends ObjectModel
{
    public static function getCompleteListOfShopsID()
    {
        $cache_id = 'Shop::getCompleteListOfShopsID';

        if (!Cache::isStored($cache_id)) {
            $list = array();

            //database load

            Cache::store($cache_id, $list);
            return $list;
        }
    }
}
```