Cache-Aside pattern

Cache for Redis

Load data on demand into a cache from a data store. This can improve performance and also helps to maintain consistency between data held in the cache and data in the underlying data store.

Context and problem

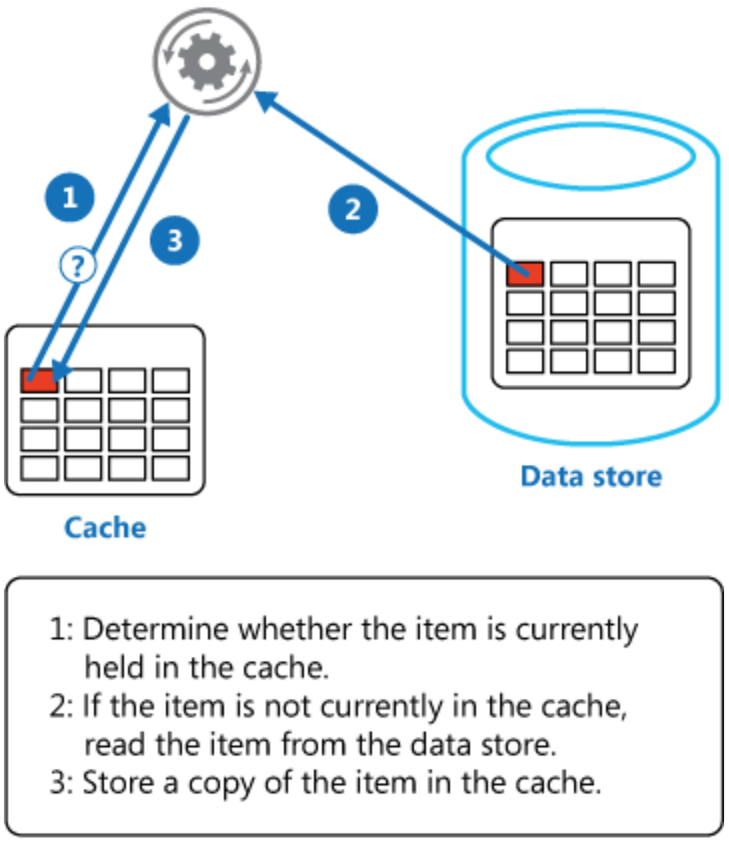
Applications use a cache to improve repeated access to information held in a data store. However, it's impractical to expect that cached data will always be completely consistent with the data in the data store. Applications should implement a strategy that helps to ensure that the data in the cache 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.


1
?
3
2
Cache
Data store
1: Determine whether the item is currently held in the cache.
2: If the item is not currently in the cache, read the item from the data store.
3: Store a copy of the item 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.

Example

In Microsoft Azure you can use Azure Cache for Redis to create a distributed cache that can be shared by multiple instances of an application.

This following code examples use the StackExchange.Redis client, which is a Redis client library written for .NET. To connect to an Azure Cache for Redis instance, call the static `ConnectionMultiplexer.Connect` method and pass in the connection string. The method returns a `ConnectionMultiplexer` that represents the connection. One approach to sharing a `ConnectionMultiplexer` instance in your application is to have a static property that returns a connected instance, similar to the following example. This approach provides a thread-safe way to initialize only a single connected instance.

C#

```csharp
private static ConnectionMultiplexer Connection;

// Redis connection string information
private static Lazy<ConnectionMultiplexer> lazyConnection = new
Lazy<ConnectionMultiplexer>(() =>
{
    string cacheConnection =
ConfigurationManager.AppSettings["CacheConnection"].ToString();
    return ConnectionMultiplexer.Connect(cacheConnection);
});

public static ConnectionMultiplexer Connection =>
lazyConnection.Value;
```

The `GetMyEntityAsync` method in the following code example shows an implementation of the Cache-Aside pattern. This method retrieves an object from the cache using the read-through approach.

An object is identified by using an integer ID as the key. The `GetMyEntityAsync` method tries to retrieve an item with this key from the cache. If a matching item is found, it's returned. If there's no match in the cache, the `GetMyEntityAsync` method retrieves the object from a data store, adds it to the cache, and then returns it. The code that actually reads the data from the data store is not shown here, because it depends on the data store. Note that the cached item is configured to expire to prevent it from becoming stale if it's updated elsewhere.

C#

```
// Set five minute expiration as a default
private const double DefaultExpirationTimeInMinutes = 5.0;

public async Task<MyEntity> GetMyEntityAsync(int id)
{
  // Define a unique key for this method and its parameters.
  var key = $"MyEntity:{id}";
  var cache = Connection.GetDatabase();

  // Try to get the entity from the cache.
  var json = await cache.StringGetAsync(key).ConfigureAwait(false);
  var value = string.IsNullOrWhiteSpace(json)
                ? default(MyEntity)
                : JsonConvert.DeserializeObject<MyEntity>(json);

  if (value == null) // Cache miss
  {
    // If there's a cache miss, get the entity from the original store
and cache it.
    // Code has been omitted because it is data store dependent.
    value = ...;

    // Avoid caching a null value.
    if (value != null)
    {
      // Put the item in the cache with a custom expiration time that
      // depends on how critical it is to have stale data.
      await cache.StringSetAsync(key,
JsonConvert.SerializeObject(value)).ConfigureAwait(false);
      await cache.KeyExpireAsync(key,
TimeSpan.FromMinutes(DefaultExpirationTimeInMinutes)).ConfigureAwait(f
alse);
    }
  }

  return value;
}
```

> The examples use Azure Cache for Redis to access the store and retrieve information from the cache. For more information, see Using Azure Cache for Redis and How to

> create a Web App with Azure Cache for Redis.

The `UpdateEntityAsync` method shown 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.

C#

```csharp
public async Task UpdateEntityAsync(MyEntity entity)
{
    // Update the object in the original data store.
    await this.store.UpdateEntityAsync(entity).ConfigureAwait(false);

    // Invalidate the current cache object.
    var cache = Connection.GetDatabase();
    var id = entity.Id;
    var key = $"MyEntity:{id}"; // The key for the cached object.
    await cache.KeyDeleteAsync(key).ConfigureAwait(false); // Delete
this key from the cache.
}
```

> **Note**
>
> The order of the steps is important. Update the data store *before* removing the item from the cache. If you remove the cached item first, there is a small window of time when a client might fetch the item before the data store is updated. That will result in a cache miss (because the item was removed from the cache), causing the earlier version of the item to be fetched from the data store and added back into the cache. The result will be stale cache data.

Related resources

The following information might be relevant when implementing this pattern:

- Caching Guidance. Provides additional information on how you can cache data in a cloud solution, and the issues that you should consider when you implement a cache.

- Data Consistency Primer. Cloud applications typically use data that's spread across data stores. Managing and maintaining data consistency in this environment is a critical aspect of the system, particularly the concurrency and availability issues that can arise. This primer describes issues about consistency across distributed data, and summarizes how an application can implement eventual consistency to maintain the availability of data.