**MEMCACHED:**

**Caching framewroks (in process cache)**

Java caching frameworks like EHCache and OSCache are essentially HashMap objects in your application code. Whenever you add a new object to the cache it will be stored in the memory of your application.   
  
This strategy works fine for storing small amounts of data, but it doesn't work for caching more than few gigabytes (GB).   
  
The designers of the Memcached server took a distributed architectural approach, which allows for system scalability. As a result, you can use Memcached to cache a huge amount of data.

**Using a traditional Java cache (in process cache)**

Using a traditional Java caching framework is quite easy, regardless of whether you choose an open source or commercial option.   
For an open source framework such as EHCache or OSCache, you would need to download the binaries and add necessary JAR files to the classpath of your application.   
  
You might also need to create a configuration file, which you would use to configure the size of the cache, disk offload, and so on.   
  
For a caching framework that came bundled with an application server you typically would not have to download any additional JARs because they would be bundled with the software.

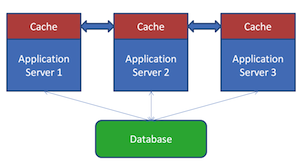
[[](http://www.javaworld.com/javaworld/jw-04-2012/Images/memcached1-fig1.png)](http://www.javaworld.com/javaworld/jw-04-2012/Images/memcached1-fig1.png" \o "Figure 1. Architecture of traditional Java caching (click to enlarge))

Figure 1. Architecture of traditional Java caching (click to enlarge)

After adding support for the caching framework in your application, you could start using it by creating a CacheManager object and getting and setting cache entries in it.   
Under the hood, the caching framework would create theCacheManager objects in the same JVM where your application was running. Every time you added a cache entry, that object would also be added to some type of hashtable maintained by the caching framework.

**Distributed Caches**

If your application server were running on multiple nodes, then you might also want support for distributed caching. In a distributed cache system, when you add an object in cache on AppServer1, that object is also available on AppServer2 and AppServer3. Traditional Java caches use *replication* for distributed caching, meaning that when you add a cache entry on AppServer1 it is automatically replicated to the other app servers in your system. As a result, the entry will be available on all of your nodes.

**Using Memcached**

In order to use Memcached for caching you must first download and install the Memcached server for the platform of your choice. Once you've installed the Memcached server it will listen on either a TCP or UDP port for caching calls.

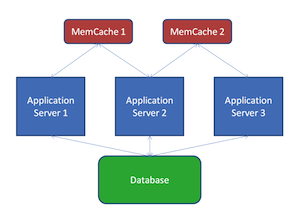
[[](http://www.javaworld.com/javaworld/jw-04-2012/Images/memcached1-fig2.png)](http://www.javaworld.com/javaworld/jw-04-2012/Images/memcached1-fig2.png" \o "Figure 2. Architecture of Memcached (click to enlarge))

Figure 2. Architecture of Memcached (click to enlarge)

Next, you'll download a [Java client](http://code.google.com/p/memcached/wiki/Clients#Java) for Memcached and add the client JARs to your application.

**Set/load entries Serialization e Deserialization**:

After that, you can create a Memcached client object and start calling its method to get and set cache entries.   
When you add an object to the cache, the Memcached client will take that object, serialize it, and send a byte array to the Memcached server for storage.   
At that point, the cached object might be garbage collected from the JVM where your application is running.

When you need that cached object, you can call the Memcached client's get() method. The client will take theget request, serialize it, and send it to the Memcached server. The Memcached server will use the request to look up the object from the cache. Once it has the object, it will return the byte array back to the Memcached client. The Memcached client object will then take the byte array and deserialize it to create the object and return it to your application.

Even if your application is running on more than one application server, all of them can point to the same Memcached server and use it for getting and setting cache entries.

**Client sa dove salva/trova entries**

If you have more than one Memcached server, the servers won't know about each other.

Instead, you'll configure your Memcached client so that it knows all the available Memcached servers.   
For example, if your application creates a Java object on AppServer1 and calls the set() method of Memcached, then the Memcached client will figure out which Memcached server that entry goes to.   
It will then start communicating with that Memcached server only.   
Likewise, when your code in AppServer2 or AppServer3 tries to get an entry, the Memcached client will first figure out which server that entry is stored on, and then communicate with that server only.

**Memcached client logic**

In its default configuration, the Memcached client uses very simple logic to select the server for a get or set operation. When you make a get() or set() call, the client takes the cache key and call its hashCode() method to get an integer such as 11. It then takes that number and divides it by number of available Memcached servers, say two. It then takes the value of the remainder, which is 1 in this case. The cache entry will go to Memcached server 1. This simple algorithm ensures that the Memcached client on each of your application servers always chooses the same server for a given cache key.

**Java Clients**