N-WAY, SET ASSOCIATIVE CACHE

Library for n-way cache implementation

Bernardoni, Mirko
mirko.bernarodni@gmail.com

Table of Contents

Purpose 4

Design requirements: 4

Cache summary 5

- Cache Algorithms 5
  - Least Recently Used (LRU) 6
  - Most Recently Used (MRU) 6

Library design 7

- Memory structure 7
- Concurrency management 9
- Behind the scenes for retrieving a value 9

Library usage 11

- Applicability 11
- Population 11
  - From a CacheLoader 11
  - Inserted Directly 12
- Eviction 12
  - LRU Algorithm 12
  - MRU Algorithm 12
  - LRU Expired Algorithm 13
  - Custom Algorithm 14
- Listeners 14
  - Removal Listeners 14
  - Miss Listener 14
  - Cached Listener 15

Class design 16

Example: User id 18

- REST API 18
- WEB Frontend 18

Getting started and requirements 19

Prerequisite 19

- Build from the source code (Make the jar) 19
- Generate the website/reports 19
- Clean the target folder 19
- How to run the example 20

Purpose

Provide a class design for an N-way, set-associative cache.

Design requirements:

- The cache itself is entirely in memory (i.e. it does not communicate with a backing store or use any I/O)
- The client interface should be type-safe for keys and values and allow for both the keys and values to be of an arbitrary type (e.g., strings, integers, classes, etc.). For a given instance of a cache all keys will be the same type and all values will be the same type.
- Design the interface as a library to be distributed by clients. Assume that the client doesn't have source code to your library and that internal data structures aren't exposed to the client.
- The design should allow for any replacement algorithm to be implemented by the client. Please provide the LRU and MRU algorithms as part of your solution.
- Example use case: As an in-memory cache on an application server, storing data associated with user id, in order to avoid a database dip for every request.

Cache summary

A very important factor in determining the effectiveness of the cache relates to how the cache is mapped to the system memory. What this means in brief is that there are many different ways to allocate the storage in our cache to the memory addresses it serves. Let's take as an example a system with 512 KB of L2 cache and 64 MB of main memory. The burning question is: how do we decide how to divvy up the 16,384 address lines in our cache amongst the "huge" 64 MB of memory?

There are three different ways that this mapping can generally be done. The choice of mapping technique is so critical to the design that the cache is often named after this choice:

- **Direct Mapped Cache**: The simplest way to allocate the cache to the system memory is to determine how many cache lines there are (16,384 in our example) and just chop the system memory into the same number of chunks. Then each chunk gets the use of one cache line. This is called direct mapping. So if we have 64 MB of main memory addresses, each cache line would be shared by 4,096 memory addresses (64 M divided by 16 K).
- **Fully Associative Cache**: Instead of hard-allocating cache lines to particular memory locations, it is possible to design the cache so that any line can store the contents of any memory location. This is called fully associative mapping.
- **N-Way Set Associative Cache**: "N" here is a number, typically 2, 4, 8 etc. This is a compromise between the direct mapped and fully associative designs. In this case the cache is broken into sets where each set contains "N" cache lines, let's say 4. Then, each memory address is assigned a set, and can be cached in any one of those 4 locations within the set that it is assigned to. In other words, within each set the cache is associative, and thus the name.
  This design means that there are "N" possible places that a given memory location may be in the cache. The tradeoff is that there are "N" times as many memory locations competing for the same "N" lines in the set.
  Let's suppose in our example that we are using a 4-way set associative cache. So instead of a single block of 16,384 lines, we have 4,096 sets with 4 lines in each. Each of these sets is shared by 16,384 memory addresses (64 M divided by 4 K) instead of 4,096 addresses as in the case of the direct mapped cache. So there is more to share (4 lines instead of 1) but more addresses sharing it (16,384 instead of 4,096).

Conceptually, the direct mapped and fully associative caches are just "special cases" of the N-way set associative cache. You can set "N" to 1 to make a "1-way" set associative cache. If you do this, then there is only one line per set, which is the same as a direct mapped cache because each memory address is back to pointing to only one possible cache location. On the other hand, suppose you make "N" really large; say, you set "N" to be equal to the number of lines in the cache (16,384 in our example). If you do this, then you only have one set, containing all of the cache lines, and every memory location points to that huge set. This means that any memory address can be in any line, and you are back to a fully associative cache.

Cache Algorithms

In computing, cache algorithms (also frequently called replacement algorithms or replacement policies) are optimizing instructions – or algorithms – that a computer program or a hardware-maintained structure can follow, in order to manage a cache of information stored on the computer.

When the cache is full, the algorithm must choose which items to discard to make room for the new ones.

Least Recently Used (LRU)

Discards the least recently used items first. This algorithm requires keeping track of what was used when, which is expensive if one wants to make sure the algorithm always discards ''the'' least recently used item.

General implementations of this technique require keeping ''age bits'' for cache-lines and track the ''Least Recently Used'' cache-line based on age-bits. In such an implementation, every time a cache-line is used, the age of all other cache-lines changes. LRU is actually a family of caching algorithms with members including, 2Q by Theodore Johnson and Dennis Shasha and LRU/K by Pat O'Neil, Betty O'Neil and Gerhard Weikum.

Most Recently Used (MRU)

Discards, in contrast to LRU, the most recently used items first. In findings presented at the 11th VLDB conference, Chou and Dewitt noted that ''When a file is being repeatedly scanned in a Looping Sequential reference pattern, MRU is the best replacement algorithm.'' Subsequently other researchers presenting at the 22nd VLDB conference noted that for random access patterns and repeated scans over large datasets (sometimes known as cyclic access patterns) MRU cache algorithms have more hits than LRU due to their tendency to retain older data. MRU algorithms are most useful in situations where the older an item is, the more likely it is to be accessed.

Library design

Memory structure

The cache is divided into sets where each set contains "N" cache entries. In order to provide concurrency access in read and write from multiple threads the set structure is called bag.

Each bag contains a double linked list of Cache entry and a read write lock object.
The read write object control the concurrent access.
The CacheEntry contains the key, value, last access time, creation time and CacheStatus.
CacheStatus define if the entry will be deleted on the next eviction or not.
Graphically a Cache bag is represented as below:

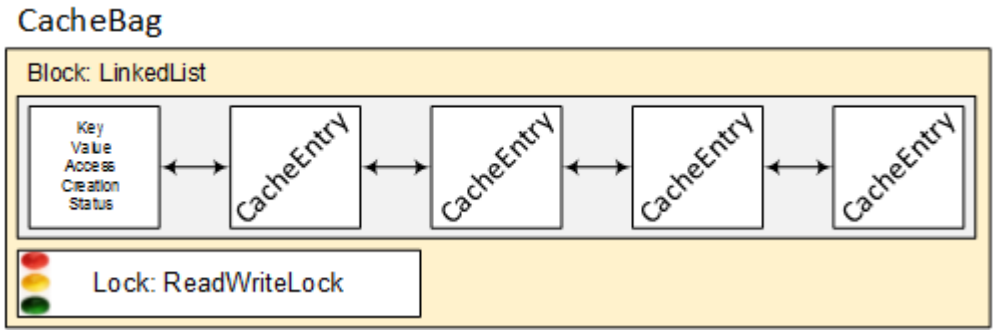


Figure 1: CacheBag

All the CacheBag are inserted in the cacheBags list. This list is immutable and allocated during the cache initialization.

With also the CacheBag representation the diagrams become:

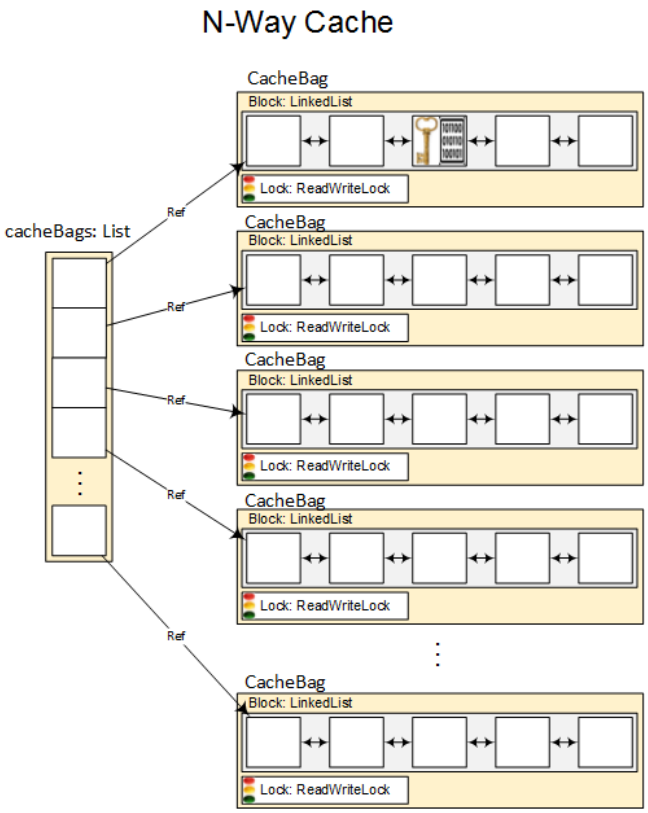


Figure 2: cacheBags

Concurrency management

The concurrency is managed only at level of each independent CacheBag object.

This is possible because the cacheBags List is immutable and defined at the nWay cache initialization.

The usage of a LinkedList allow to use a write lock only when the CacheEntry in the Block is added or deleted. Any other operation require the read lock only.

The usage of a concurrent list has been avoided because the class is synchronizing every access without making distinction between read and write.

Read locks are used when:

- An immutable copy of the Block is created in order to give to the CacheEviction implementation
- A key is searched inside the current Block

Write locks are used when:

- The entries in the block are deleted as result of the eviction
- A new entry is created

No lock are necessary when:

- Access time is updated (the variable is declared as volatile)
- The entry value is updated (the variable is declared as volatile)
- Any read to the entry object
- Getting access to the CacheBag object

Behind the scenes for retrieving a value

The main operation of a cache is retrieving a value. For retrieving a value you must have a key.

1. It is used the key.hash function for having an integer value. This number is used to determine what is the key position in the cacheBags list via a mod operation
2. The right cacheBag is retrieved
3. A sequential search is applied to the block using the key.equals function for find the right Cache Entry
4. If a cache entry has been found then the value is returned to the user
5. An entry is not present then the CacheLoader is invoked
6. CacheLoader returns the value from some slow access memory (database, filesystem, network…)
7. A new Entry is created with the current key and the value loaded
8. The Entry is added in the block. (LinkedList always add at the end of the structure)
9. The value is returned

In case of the Block size is equals or bigger than “N” an eviction is called at the beginning of step 7.

Eviction is going to:

1. Create an immutable list from the block. This operation is done via the guava immutable list class that is “smart” enough to avoid a physical copy if not necessary
2. Eviction is called

3. Every CacheEntry that has status to DELETED is going to be removed from the block
4. If the size of the block is major that the double of “N” an OutOfMemoryException is generated
5. A new Entry is created with the current key and the value loaded

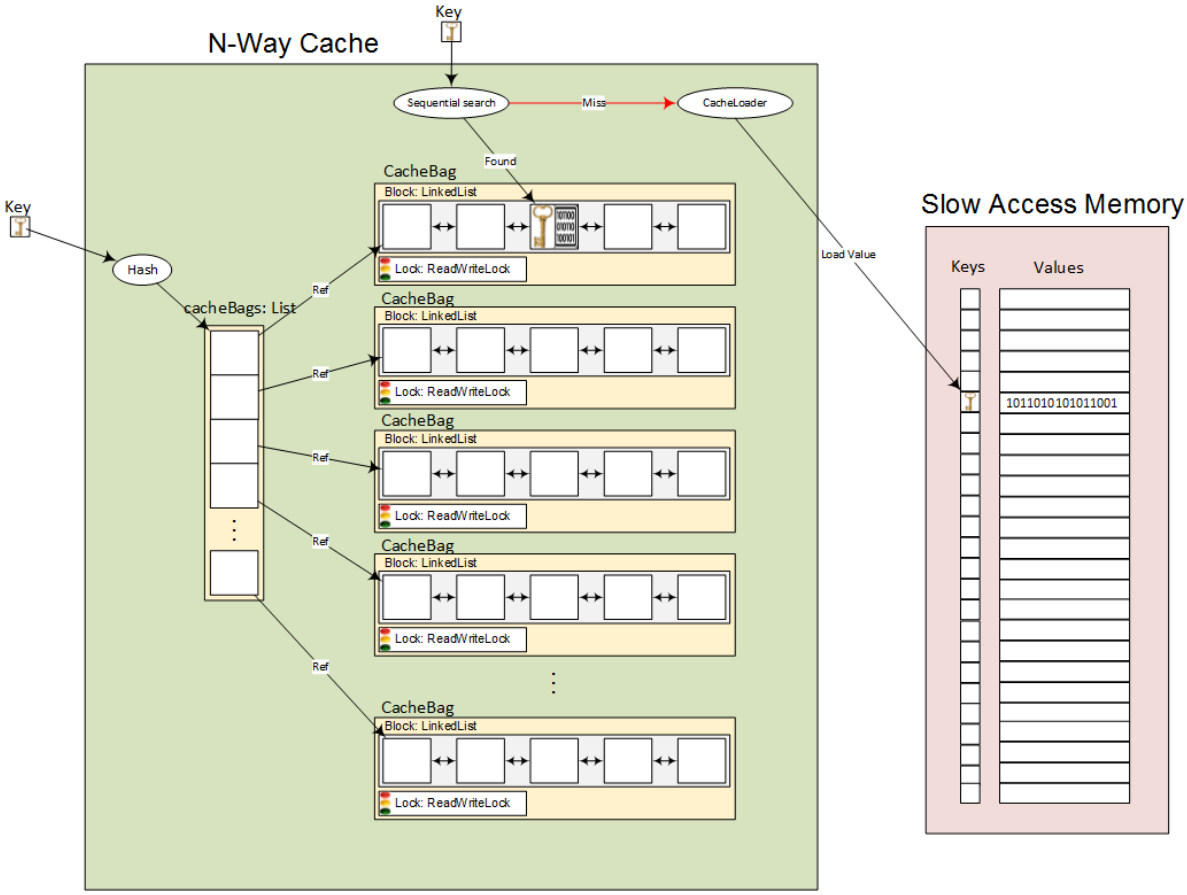


Figure 3: Get a value

Library usage

Example

```
Cache<Integer, String> myCache = new NWayCacheBuilder<>()
    .build(key -> {return myDataDAO.load(key)});
```

Applicability

Caches are tremendously useful in a wide variety of use cases. For example, you should consider using caches when a value is expensive to compute or retrieve, and you will need its value on a certain input more than once.

A Cache is similar to ConcurrentMap, but not quite the same. The most fundamental difference is that aConcurrentMap persists all elements that are added to it until they are explicitly removed. A Cache on the other hand is generally configured to evict entries automatically, in order to constrain its memory footprint. In some cases a Cache can be useful even if it doesn't evict entries, due to its automatic cache loading.

Generally, the n-way caching is applicable whenever:

- You are willing to spend some memory to improve speed.
- You expect that keys will sometimes get queried more than once.
- Your cache will not need to store more data than what would fit in RAM. (Cache is local to a single run of your application. It is not storing data in files, or on outside servers.) If each of these apply to your use case, then the N-Way cache could be right for you!

Obtaining a Cache is done using the NWayCacheBuilder builder pattern as demonstrated by the example code above, but customizing your cache is the interesting part.

Population

The first question to ask yourself about your cache is: is there some sensible default function to load or compute a value associated with a key? If so, you should use a CacheLoader. Elements can be inserted directly, usingCache.put, but automatic cache loading is preferred as it makes it easier to reason about consistency across all cached content.

From a CacheLoader

Creating a CacheLoader is typically as easy as implementing the method Value load(Key key) throws Exception. So, for example, you could create a Cache with the following code:

```
Cache<Key, Graph> myCache = new NWayCacheBuilder<>()
    .build(
        new CacheLoader<Key, Graph>() {
            public Graph load(Key key) throws Exception {
                return createExpensiveGraph(key);
            }
   });
...
```

```
try {
    return graphs.get(key);
} catch (Exception e) {
    throw new OtherException(e.getCause());
}
```

The canonical way to query a Cache is with the method get(K). This will either return an already cached value, or else use the cache's CacheLoader to atomically load a new value into the cache. Because CacheLoader might throw an Exception, Cache.get(K) throws Exception.

Inserted Directly

Values may be inserted into the cache directly with Cache.put(key, value). This overwrites any previous entry in the cache for the specified key.

Eviction

The cold hard reality is that we almost certainly don't have enough memory to cache everything we could cache. You must decide: when is it not worth keeping a cache entry? N-Way cache provides three algorithms to do so: LRU, MRU, LRU Expired. In addition it is possible to write your own eviction algorithm.

The eviction algorithm is not going to physically delete any cache entry. The deletion process is managed by the cache implementation itself. Instead the eviction is marking the entries as DELETED.

Each cache block contains the entries in creation order (the older are first), this is guarantee by the Cache implementation.

LRU Algorithm

The class LRUAlgorithm implements of a simple version of LRU algorithm.

This implementation deletes only the oldest LRUAlgorithm.entriesToDelete entries from the current block.

Example:

Remember that each cache block contains the entries in creation order (the older are first)

```
Memory block = [ 1 -> "first", 5 -> "apple", 2 -> "red", 10 -> "table", 3->
"orange" ]
```

```
entriesToDelete = 3
```

After the eviction:

```
Memory block = [ 10 -> "table", 3-> "orange" ]
```

Usage:

```
Cache<Integer, String> myCache = new NWayCacheBuilder<>()
    .LRUEviction() // Or nothing because it is the default
.build(key -> ... });
```

MRU Algorithm

The class MRUAlgorithm implements of a simple version of MRU algorithm.

This implementation deletes only the latest LMRUAlgorithm.entriesToDelete entries from the current block.
In other words it is just the opposite than LRU.

Example:
Remember that each cache block contains the entries in creation order (the older are first)

```
Memory block = [ 1 -> "first", 5 -> "apple", 2 -> "red", 10 -> "table", 3-> "orange" ]

entriesToDelete = 3
```

After the eviction:

```
Memory block = [ 1 -> "first", 5 -> "apple" ]
```

Usage:

```
Cache<Integer, String> myCache = new NWayCacheBuilder<>()

    .MRUEviction()

.build(key -> ... });
```

LRU Expired Algorithm

The class LRUExpiredAlgorithm implements a LRU algorithm based on invalidating the entries not used for a while.
The differences from simple LRU are:

- Eviction is based on the expired entries (time based). An entry is expired when the access time is too old.
  The time is expressed in milliseconds.
- Guarantee at least one eviction. If no element are expired the oldest one is deleted.

Eviction entry calculation:

```
Expiration Time = Entry Access time + expiration

if Expiration Time < Current time then

    mark for deletion current entry
```

Example:

```
Memory block = [ 1 -> ("first", AccessTime: 1000) , 5 -> ("apple", AccessTime: 11000),

                 2 -> ("red", ("apple", AccessTime: 10010), 10 -> ("table", AccessTime: 100),

                 3-> ("orange",  AccessTime: 10005) ]

expiration = 5000

currentTime = 12000
```

After the eviction:

```
Memory block = [ 5 -> ("apple", AccessTime: 11000),  2 -> ("red", ("apple", AccessTime: 10010)]
```

Usage:

```
Cache<Integer, String> myCache = new NWayCacheBuilder<>()
    .LRUExpiredEviction()
.build(key -> ... });
```

Custom Algorithm

Creating a custom eviction algorithm is typically as easy as implementingCacheEviction.eviction(List<CacheEntry<Key, Value>> block). The follow point are to take in consideration:

- Each cache block contains the entries in creation order (the older are first), this is guarantee by the Cache implementation.
- The block is immutable. It is not possible to add or delete entries.
- CacheEntry.status is used to determine if the entry has to be deleted or not (DELETED, ACTIVE)

Usage:

```
Cache<Integer, String> myCache = new NWayCacheBuilder<>()
    .customEviction(block -> your beautiful eviction algorithm)
.build(key -> ... });
```

Listeners

Three different types of listeners are provided in order to take actions or collect cache information: removal, cached, miss.
Is it possible to add and create more than one listener per type. The cache implementation guarantee to call all of them.

Removal Listeners

```
RemovalListener is called after removing an entry from the cache.
Cache<Integer, String> myCache = new NWayCacheBuilder<>()
          .build(key -> {return ...)});
RemovalListener<Key, DatabaseConnection> removalListener = new RemovalListener<Key,
DatabaseConnection>() {
    public void onRemoval(RemovalNotification<Key, DatabaseConnection> removal) {
        DatabaseConnection conn = removal.getValue();
        conn.close(); // tear down properly
    }
};
```

myCache.addRemovalListener(removalListener);

Miss Listener

MissListener is called when a entry is requested and it is not found in the cache (miss).

```
Cache<Integer, String> myCache = new NWayCacheBuilder<>()
          .build(key -> {return ...)});
```

```
MissListener<Key> missListener = new MissListener<Key>() {
    public void onMiss(Key) {
        collectMissStatistics();
    }
};
myCache.addMissListener(missListener);
```

Cached Listener

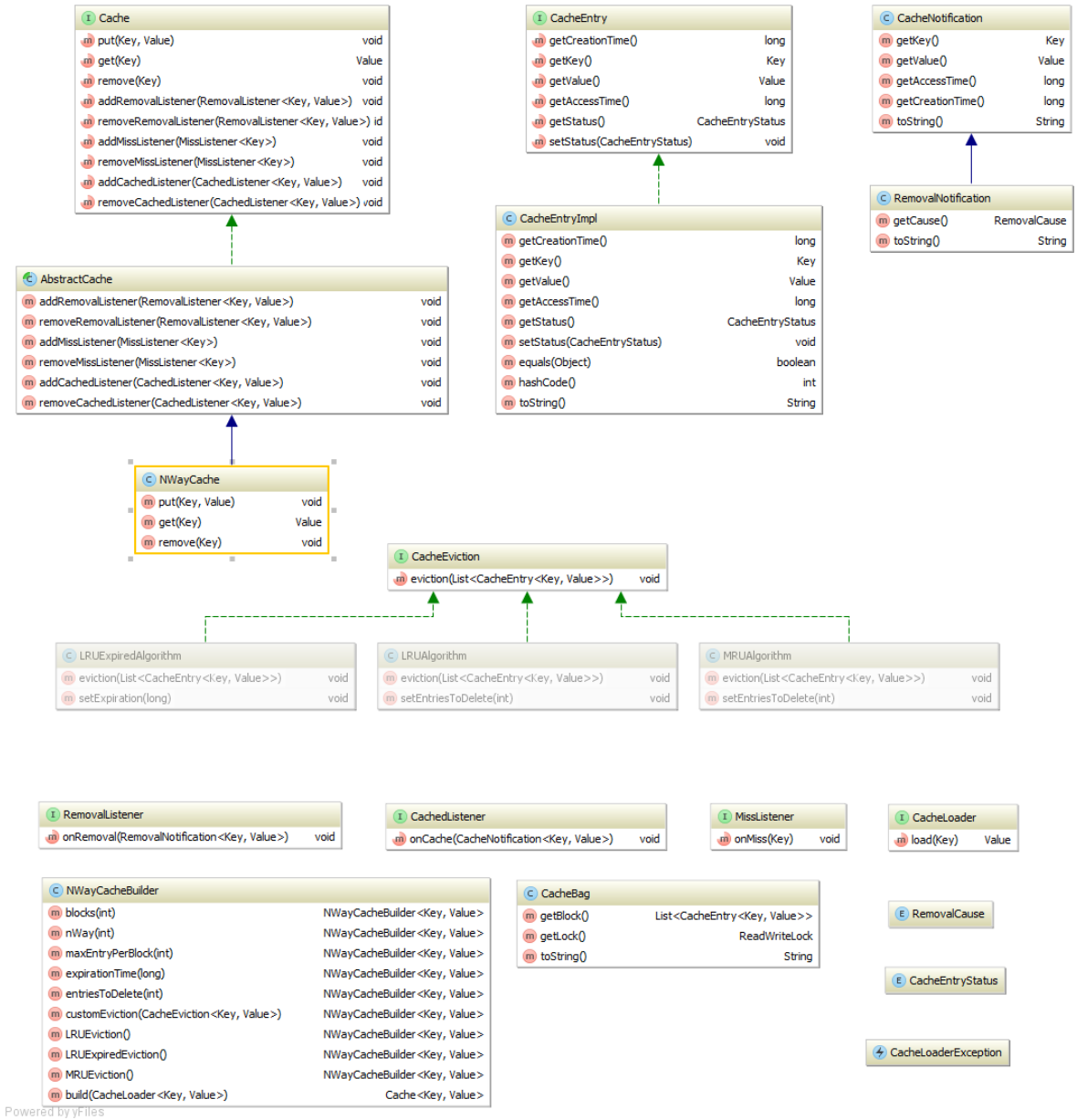
CachedListener is called when a entry is requested and it is in the cache. No load is necessary for retrieving the value.
Please note that this listener can slow down the cache performance. It is strongly suggested to implement the method in a separate thread.

```
Cache<Integer, String> myCache = new NWayCacheBuilder<>()
          .build(key -> {return ...)});
CachedListener<Key> cachedListener = new MissListener<Key>() {
    public void onCache(CacheNotification<Key, Value> notification) {
        collectCacheStatistics(notification);
    }
};
myCache.addCachedListener(cachedListener);
```

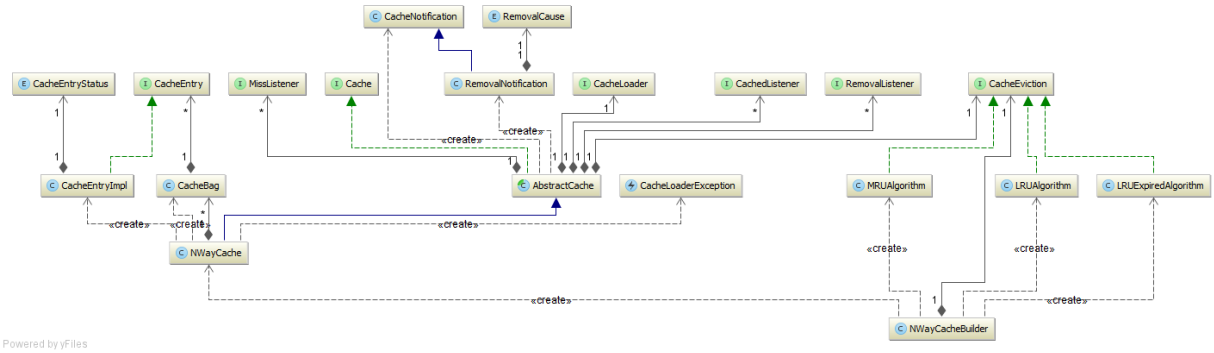
Class design

The follow diagrams is showing the public class and interfaces that the final user is allowed to use or implement with all the methods.

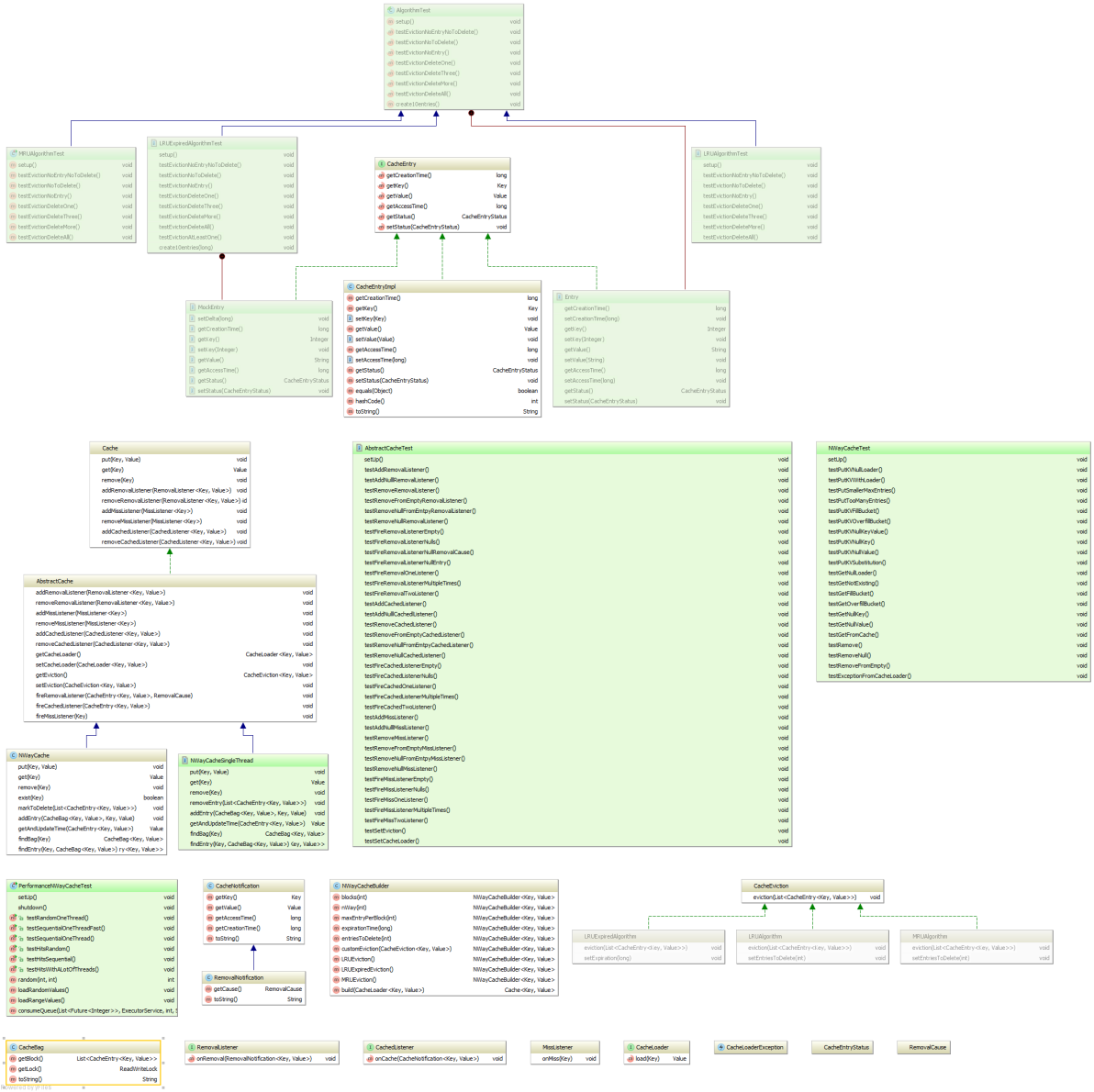
AbstractCache and NWayCache are declared with the access modifier package in order to avoid that the client can instantiate without using the NWayCacheBuilder (builder pattern).



The follow diagrams is showing the public class and interfaces that the final user is allowed to use and their dependences.



The follow diagram is giving the full overview of all the classes, including the “internal” ones and the tests



Example: User id

The use case realized is an in-memory cache on an application server, storing data associated with user id, in order to avoid a database dip for every request.

The code is dived in two parts a set of REST API for managing the user id and a web fronted that consume the API

REST API

The REST API is developed using Spring boot with Tomcat as application server, Hibernate as JPA implementation and H2 as the database (it is in memory).

REST web service specification:

Method	URL	Input Body	Return Body	Description
`GET`	/users/{id}	Not required	User JSON object	Retrieve information about the user with {id} `HTTP 404` code returned if the {id} is not found
`GET`	/users	Not required	Array of user JSON objects	Retrieve all the users
`POST`	/users	User JSON object	Nothing	Create a new user
`POST`	/users/load	Not required	Nothing	Load 50 users
`PUT`	/users/{id}	User JSON object	Nothing	Update an existing user with {id} `HTTP 404` code returned if the {id} is not found
`DELETE`	/users/{id}	Not required	Nothing	Delete an existing user with {id} `HTTP 404` code returned if the {id} is not found

REST API Structure

The API structure is divided in three layers: DAO, Service, Controller.

- **DAO**: Is implemented by Spring Data. The only necessary code is defining the DAO interface `UserDAO`
- **Service**: The N Way Cache is used on this layer in order to avoid unnecessary call to database. This layer is accessible via the `UserServices` interface. Its implementation is done via `CachedUserServices`.
- **Controller**: `RestController`is defining the endpoints for the REST api. It calls the `UserService` for delegating the API implementation.

`User` class describe the model of a user.
`Application` class is the entry point for Spring boot in order to launch H2 in memory database and the tomcat application server.
`NotFoundException` is raised when a user is not found for a id.

WEB Frontend

The WEB frontend is an addons to the REST api in order to have an easy way to interrogate them.

The web fronted is developed using AngularJs and Bootstrap for the CSS.
It is an implementation of a basic CRUD operation around a user.

Getting started and requirements

N-Way cache is a java 8 library that uses maven.

Prerequisite

- JDK 8 in the path (http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html)
- `JAVA_HOME` set to JDK 8 (example: `set JAVA_HOME=C:\Program Files\Java\jdk1.8.0_45`)
- `Maven 3.3.3` installed in the path (https://maven.apache.org/download.cgi)
- You can test your java environment configuration launching the follow command

```
mvn -v
```

the output is something like

```
C:\git>mvn -v
Apache Maven 3.3.3 (7994120775791599e205a5524ec3e0dfe41d4a06; 2015-04-22T12:57:37+01:00)
Maven home: C:\git\apache-maven-3.3.3\bin\..
Java version: 1.8.0_45, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.8.0_45\jre
Default locale: en_GB, platform encoding: Cp1252
OS name: "windows 7", version: "6.1", arch: "amd64", family: "dos"
```

Build from the source code (Make the jar)

- Open the console/shell
- Enter in the `nway` folder
- Execute `mvn package`

The jar will be found in the target sub-folder nway-1.0-SNAPSHOT.jar

Mvn package is going to run all the unit tests.

Generate the website/reports

- Open the console/shell
- Enter in the `nway` folder
- Execute `mvn site`

The site is created in the target\site\

Clean the target folder

- Open the console/shell
- Enter in the `nway` folder
- Execute `mvn clean`

The target subfolder is not existing anymore

How to run the example

- Open the console/shell
- Enter in the `nway` folder
- Execute `mvn install`
- Enter in the `nway-example` folder
- Execute `mvn package`
- Enter in `target`
- Execute `java -jar nway-example-1.0.0-SNAPSHOT.jar`

Now in about 10 seconds the application servers is running at port 8080.

- Open your browser at http://localhost:8080