## **1. What is caching?**

Caching is a mechanism to enhance the performance of a system. It is a temporary memory that lies between the application and the persistent database. Cache memory stores recently used data items in order to reduce the number of hits to any data source

## **2. Types of cache**

In general, cache can be seen of following types.

#### **2.1. In-memory caching**

This is the most frequently used area where caching is used extensively to increase performance of the application. In-memory caches such as Memcached and Radis are key-value stores between your application and your data storage. Since the data is held in RAM, it is much faster than typical databases where data is stored on disk.

RAM is more limited than disk, so cache invalidation algorithms such as least recently used (LRU) can help invalidate ‘cold’ entries and keep ‘hot’ data in RAM. Memcached is in-memory caching where Redis is more advanced which allows us to backup and restore facility as well as it is distributed caching tool where we can manage caching in distributed clusters.

#### **2.2. Database caching**

Your database usually includes some level of caching in a default configuration, optimized for a generic use case. Tweaking these settings for specific usage patterns can further boost performance. One popular in this area is first level cache of Hibernate or any ORM frameworks.

#### **2.3. Web server caching**

Reverse proxies and caches such as [Varnish](https://varnish-cache.org/index.html) can serve static and dynamic content directly. Web servers can also cache requests, returning responses without having to contact application servers. In today’s API age, this option is a viable if we want to cache API responses in web server level.

#### **2.4. CDN caching**

Caches can be located on the client side (OS or browser), server side, or in a distinct cache layer.

## **3. Spring boot cache annotations**

Spring framework provides cache abstraction api for different cache providers. The usage of the API is very simple, yet very powerful. Today we will see the annotation based Java configuration on caching. Note that we can achieve similar functionality through XML configuration as well.

Cache Hit ratio

Algorithm

#### **3.1. @EnableCaching**

It enables Spring’s annotation-driven cache management capability. In spring boot project, we need to add it to the boot application class annotated with @SpringBootApplication. Spring provides one concurrent hashmap as default cache, but we can override CacheManager to register external cache providers as well easily.

#### **3.2. @Cacheable**

It is used on the method level to let spring know that the response of the method are cacheable. Spring manages the request/response of this method to the cache specified in annotation attribute. For example, @Cacheable ("cache-name1", “cache-name2”).

@Cacheable annotation has more options. Like we can specify the key of the cache from the request of the method. If nothing specified, spring uses all the class fields and use those as cache key (mostly HashCode) to maintain caching but we can override this behavior by providing key information.

76351376

| @Cacheable(value="books", key="#isbn")  public Book findStoryBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)  {  //retrieve from DB 100 ms delay  }    @Cacheable(value="books", key="#isbn.rawNumber")  public Book findStoryBook (ISBN isbn, boolean checkWarehouse, boolean includeUsed)    @Cacheable(value="books", key="T(classType).hash(#isbn)")  public Book findStoryBook (ISBN isbn, boolean checkWarehouse, boolean includeUsed) |
| --- |

We can also use conditional caching as well. For example,

| @Cacheable(value="book", condition="#name.length < 50")  public Book findStoryBook (String name) |
| --- |

#### **3.3. @CachePut**

Sometimes we need to manipulate the cacheing manually to put (update) cache before method call. This will allow us to update the cache and will also allow the method to be executed. The method will always be executed and its result placed into the cache (according to the @CachePut options).

It supports the same options as @Cacheable and should be used for cache population rather then method flow optimization.

Note that using @CachePut and @Cacheable annotations on the same method is generally discouraged because they have different behaviors. While the latter causes the method execution to be skipped by using the cache, the former forces the execution in order to execute a cache update.

This leads to unexpected behavior and with the exception of specific corner-cases (such as annotations having conditions that exclude them from each other), such declarations should be avoided.

#### **3.4. @CacheEvict**

It is used when we need to evict (remove) the cache previously loaded of master data. When CacheEvict annotated methods will be executed, it will clear the cache.

We can specify key here to remove cache, if we need to remove all the entries of the cache then we need to use allEntries=true. This option comes in handy when an entire cache region needs to be cleared out – rather then evicting each entry (which would take a long time since it is inefficient), all the entries are removed in one operation.

#### **3.5. @Caching**

This annotation is required when we need both CachePut and CacheEvict at the same time.

## **4. How to register a cache engine with spring boot**

Spring boot provides integration with following cache providers. Spring boot does the auto configuration with default options if those are present in class path and we have enabled cache by @EnableCaching in the spring boot application.

* JCache (JSR-107) (EhCache 3, Hazelcast, Infinispan, and others)
* EhCache 2.x
* Hazelcast
* Infinispan
* Couchbase
* Redis
* Caffeine
* Simple cache
* Guava Cache

We can override specific cache behaviors in Spring boot by overriding the cache provider specific settings – for example-

| spring.cache.infinispan.config=infinispan.xml |
| --- |

For details information this we can see the official spring boot documentation [here](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-caching.html)

## **5. Spring boot caching example**

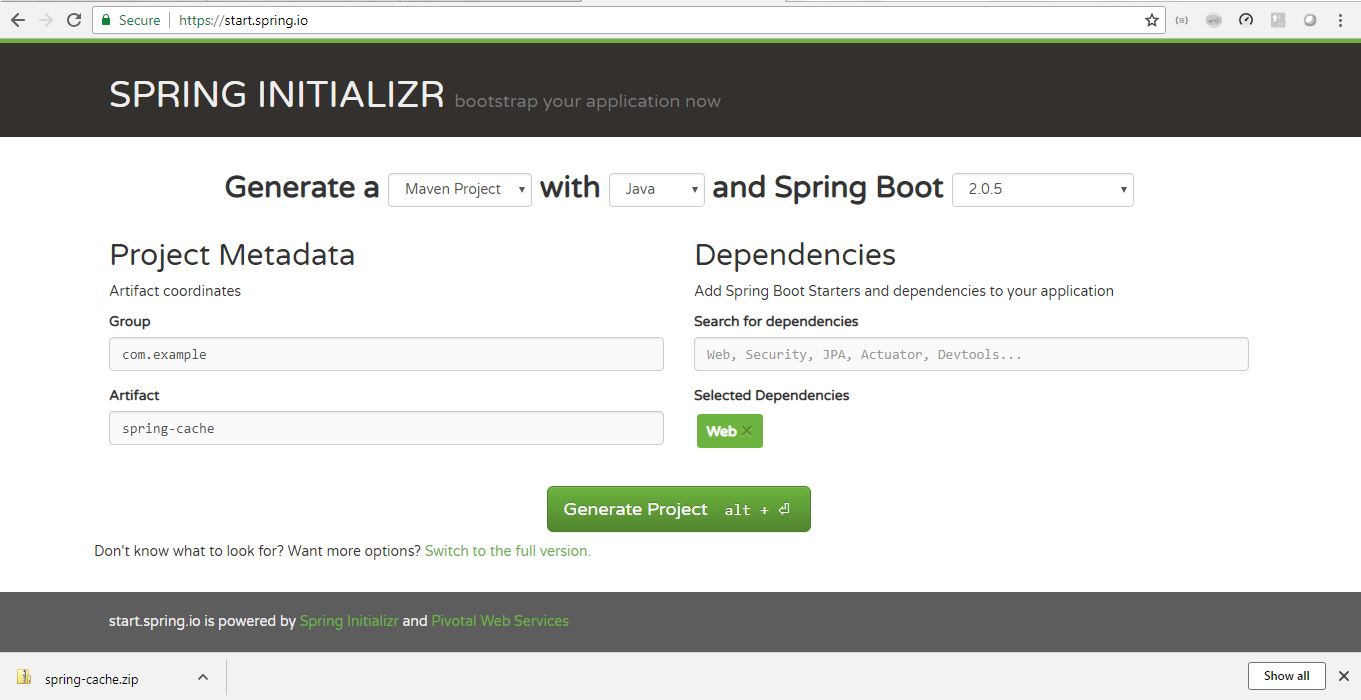
In this spring boot cahce configuration example, we will see how we can *enable default caching in spring boot* and will also enable caching for one of the business method. Finally we will test that the application performance on the repeated call of same method.

We will simulate the delay in the actual method call by using Thread.sleep() method to feel the effect of cache. So let’s follow the simple steps in creating project and testing.

#### **5.1 Create Spring Boot project**

Create one simple spring boot project named spring-cache with spring-boot-web dependency for hosting this in web server.

To do this we need to go to *https://start.spring.io/* and give the maven coordinates and select dependencies. Download the zip file containing the skeleton project. Then we need to import that in eclipse once unzipped in suitable folder. Do the initial mvn clean install to download all the required dependencies to local repository.

**Spring Boot Project Creation**

#### **5.2 Create HTTP GET REST API**

Create one REST service which will be a search service using GET request. Our main target is to cache the response of the method in the service layer where we will introduce an intentional delay to simulate the actual backend service call to get the result. In the first hit, the response will be delayed as we will have some simulated delay in the application, but in the subsequent calls, we will get much faster response.