

30. Cache Abstraction

[Prev](#)

Part VI. Integration

[Next](#)

30. Cache Abstraction

30.1 Introduction

Since version 3.1, Spring Framework provides support for transparently adding caching into an existing Spring application. Similar to the [transaction](#) support, the caching abstraction allows consistent use of various caching solutions with minimal impact on the code.

As from Spring 4.1, the cache abstraction has been significantly improved with the support of [JSR-107 annotations](#) and more customization options.

30.2 Understanding the cache abstraction

Cache vs Buffer

The terms "buffer" and "cache" tend to be used interchangeably; note however they represent different things. A buffer is used traditionally as an intermediate temporary store for data between a fast and a slow entity. As one party would have to *wait* for the other affecting performance, the buffer alleviates this by allowing entire blocks of data to move at once rather than in small chunks. The data is written and read only once from the buffer. Furthermore, the buffers are *visible* to at least one party which is aware of it.

A cache on the other hand by definition is hidden and neither party is aware that caching occurs. It as well improves performance but does that by allowing the same data to be read multiple times in a fast fashion.

A further explanation of the differences between two can be found [here](#).

At its core, the abstraction applies caching to Java methods, reducing thus the number of executions based on the information available in the cache. That is, each time a *targeted* method is invoked, the abstraction will apply a caching behavior checking whether the method has been already executed for the given arguments. If it has, then the cached result is returned without having to execute the actual method; if it has not, then method is executed, the result cached and returned to the user so that, the next time the method is invoked, the cached result is returned. This way, expensive methods (whether CPU or IO bound) can be executed only once for a given set of parameters and the result reused without having to actually execute the method again. The caching logic is applied transparently without any interference to the invoker.



Important

Obviously this approach works only for methods that are guaranteed to return the same output (result) for a given input (or arguments) no matter how many times it is being executed.

Other cache-related operations are provided by the abstraction such as the ability to update the content of the cache or remove one of all entries. These are useful if the cache deals with data that can change during the course of the application.

Just like other services in the Spring Framework, the caching service is an abstraction (not a cache implementation) and requires the use of an actual storage to store the cache data - that is, the abstraction frees the developer from having to write the caching logic but does not provide the actual stores. This abstraction is materialized by the `org.springframework.cache.Cache` and `org.springframework.cache.CacheManager` interfaces.

There are a few implementations of that abstraction available out of the box: JDK

`java.util.concurrent.ConcurrentMap` based caches, EhCache, Gemfire cache, Guava caches and JSR-107 compliant caches. See Section 30.7, “Plugging-in different back-end caches” for more information on plugging in other cache stores/providers.





Important

The caching abstraction has no special handling of multi-threaded and multi-process environments as such features are handled by the cache implementation. .

If you have a multi-process environment (i.e. an application deployed on several nodes), you will need to configure your cache provider accordingly. Depending on your use cases, a copy of the same data on several nodes may be enough but if you change the data during the course of the application, you may need to enable other propagation mechanisms.

Caching a particular item is a direct equivalent of the typical get-if-not-found-then- proceed-and-put-eventually code blocks found with programmatic cache interaction: no locks are applied and several threads may try to load the same item concurrently. The same applies to eviction: if several threads are trying to update or evict

►  Search Documentation **DOCS SPRING 4.0** Live Online, 24 Nov 2014  For advanced features in that area, refer to the documentation of the cache provider that you are using for more details.

To use the cache abstraction, the developer needs to take care of two aspects:

- caching declaration - identify the methods that need to be cached and their policy
- cache configuration - the backing cache where the data is stored and read from

30.3 Declarative annotation-based caching

For caching declaration, the abstraction provides a set of Java annotations:

- `@Cacheable` triggers cache population
- `@CacheEvict` triggers cache eviction
- `@CachePut` updates the cache without interfering with the method execution
- `@Caching` regroups multiple cache operations to be applied on a method
- `@CacheConfig` shares some common cache-related settings at class-level

Let us take a closer look at each annotation:

30.3.1 @Cacheable annotation

As the name implies, `@Cacheable` is used to demarcate methods that are cacheable - that is, methods for whom the result is stored into the cache so on subsequent invocations (with the same arguments), the value in the cache is returned without having to actually execute the method. In its simplest form, the annotation declaration requires the name of the cache associated with the annotated method:

```
@Cacheable("books")
public Book findBook(ISBN isbn) {...}
```

In the snippet above, the method `findBook` is associated with the cache named `books`. Each time the method is called, the cache is checked to see whether the invocation has been already executed and does not have to be repeated. While in most cases, only one cache is declared, the annotation allows multiple names to be specified so that more than one cache are being used. In this case, each of the caches will be checked before executing the method - if at least one cache is hit, then the associated value will be returned:



All the other caches that do not contain the value will be updated as well even though the cached method was not actually executed.

```
@Cacheable({"books", "isbns"})
public Book findBook(ISBN isbn) {...}
```

Default Key Generation

Since caches are essentially key-value stores, each invocation of a cached method needs to be translated into a suitable key for cache access. Out of the box, the caching abstraction uses a simple `KeyGenerator` based on the following algorithm:

- If no params are given, return `SimpleKey.EMPTY`.
- If only one param is given, return that instance.
- If more the one param is given, return a `SimpleKey` containing all parameters.

This approach works well for most use-cases; As long as parameters have *natural keys* and implement valid `hashCode()` and `equals()` methods. If that is not the case then the strategy needs to be changed.

To provide a different *default* key generator, one needs to implement the `org.springframework.cache.interceptor.KeyGenerator` interface.



The default key generation strategy changed with the release of Spring 4.0. Earlier versions of Spring used a key generation strategy that, for multiple key parameters, only considered the `hashCode()` of parameters and not `equals()`; this could cause unexpected key collisions (see [SPR-10237](#) for background). The new `SimpleKeyGenerator` uses a compound key for such scenarios.

If you want to keep using the previous key strategy, you can configure the deprecated `org.springframework.cache.interceptor.DefaultKeyGenerator` class or create a custom hash-based `KeyGenerator` implementation.

Custom Key Generation Declaration

Since caching is generic, it is quite likely the target methods have various signatures that cannot be simply mapped on top of the cache structure. This tends to become obvious when the target method has multiple arguments out of which only some are suitable for caching (while the rest are used only by the method logic). For example:

```
@Cacheable("books")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

At first glance, while the two `boolean` arguments influence the way the book is found, they are no use for the cache. Further more what if only one of the two is important while the other is not?

For such cases, the `@Cacheable` annotation allows the user to specify how the key is generated through its `key` attribute. The developer can use SpEL to pick the arguments of interest (or their nested properties), perform operations or even invoke arbitrary methods without having to write any code or implement any interface. This is the recommended approach over the `default generator` since methods tend to be quite different in signatures as the code base grows; while the default strategy might work for some methods, it rarely does for all methods.

Below are some examples of various SpEL declarations - if you are not familiar with it, do yourself a favor and read [Chapter 8, Spring Expression Language \(SpEL\)](#):

```
@Cacheable(value="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@Cacheable(value="books", key="#isbn.rawNumber")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@Cacheable(value="books", key="T(someType).hash(#isbn)")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

The snippets above show how easy it is to select a certain argument, one of its properties or even an arbitrary (static) method.

If the algorithm responsible to generate the key is too specific or if it needs to be shared, you may define a custom `keyGenerator` on the operation. To do this, specify the name of the `KeyGenerator` bean implementation to use:

```
@Cacheable(value="books", keyGenerator="myKeyGenerator")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```



The `key` and `keyGenerator` parameters are mutually exclusive and an operation specifying both will result in an exception.

Default Cache Resolution

Out of the box, the caching abstraction uses a simple `CacheResolver` that retrieves the cache(s) defined at the operation level using the configured `CacheManager`.

To provide a different *default* cache resolver, one needs to implement the `org.springframework.cache.interceptor.CacheResolver` interface.

Custom cache resolution

The default cache resolution fits well for applications working with a single `CacheManager` and with no complex cache resolution requirements.

For applications working with several cache managers, it is possible to set the `cacheManager` to use per operation:

```
@Cacheable(value="books", cacheManager="anotherCacheManager")
public Book findBook(ISBN isbn) {...}
```

It is also possible to replace the `CacheResolver` entirely in a similar fashion as for [key generation](#). The resolution is requested for every cache operation, giving a chance to the implementation to actually resolve the cache(s) to use based on runtime arguments:

```
@Cacheable(cacheResolver="runtimeCacheResolver")
public Book findBook(ISBN isbn) {...}
```



Since Spring 4.1, the `value` attribute of the cache annotations are no longer mandatory since this particular information can be provided by the `CacheResolver` regardless of the content of the annotation.

Similarly to `key` and `keyGenerator`, the `cacheManager` and `cacheResolver` parameters are mutually exclusive and an operation specifying both will result in an exception as a custom `CacheManager` will be ignored by the `CacheResolver` implementation. This is probably not what you expect.

Conditional caching

Sometimes, a method might not be suitable for caching all the time (for example, it might depend on the given arguments). The cache annotations support such functionality through the `condition` parameter which takes a `SpEL` expression that is evaluated to either `true` or `false`. If `true`, the method is cached - if not, it behaves as if the method is not cached, that is executed every since time no matter what values are in the cache or what arguments are used. A quick example - the following method will be cached only if the argument `name` has a length shorter than 32:

```
@Cacheable(value="book", condition="#name.length < 32")
public Book findBook(String name)
```

In addition the `condition` parameter, the `unless` parameter can be used to veto the adding of a value to the cache. Unlike `condition`, `unless` expressions are evaluated *after* the method has been called. Expanding on the previous example - perhaps we only want to cache paperback books:

```
@Cacheable(value="book", condition="#name.length < 32", unless="#result.hardback")
public Book findBook(String name)
```

Available caching SpEL evaluation context

Each `SpEL` expression evaluates again a dedicated `context`. In addition to the build in parameters, the framework provides dedicated caching related metadata such as the argument names. The next table lists the items made available to the context so one can use them for key and conditional computations:

Table 30.1. Cache SpEL available metadata

Name	Location	Description	Example
methodName	root object	The name of the method being invoked	<code>#root.methodName</code>
method	root object	The method being invoked	<code>#root.method.name</code>
target	root object	The target object being invoked	<code>#root.target</code>

targetClass	root object	The class of the target being invoked	<code>#root.targetClass</code>
args	root object	The arguments (as array) used for invoking the target	<code>#root.args[0]</code>
caches	root object	Collection of caches against which the current method is executed	<code>#root.caches[0].name</code>
argument name	evaluation context	Name of any of the method argument. If for some reason the names are not available (ex: no debug information), the argument names are also available under the <code>a<#arg></code> where <code>#arg</code> stands for the argument index (starting from 0).	<code>iban</code> or <code>a0</code> (one can also use <code>p0</code> or <code>p<#arg></code> notation as an alias).
result	evaluation context	The result of the method call (the value to be cached). Only available in <code>unless</code> expressions, <code>cache put</code> expressions (to compute the <code>key</code>), or <code>cache evict</code> expressions (when <code>beforeInvocation</code> is <code>false</code>).	<code>#result</code>

30.3.2 @CachePut annotation

For cases where the cache needs to be updated without interfering with the method execution, one can use the `@CachePut` annotation. That is, 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 than method flow optimization:

```
@CachePut(value="book", key="#isbn")
public Book updateBook(ISBN isbn, BookDescriptor descriptor)
```



Important

Note that using `@CachePut` and `@Cacheable` annotations on the same method is generally strongly 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 declaration should be avoided. Note also that such condition should not rely on the result object (i.e. the `#result` variable) as these are validated upfront to confirm the exclusion.

30.3.3 @CacheEvict annotation

The cache abstraction allows not just population of a cache store but also eviction. This process is useful for removing stale or unused data from the cache. Opposed to `@Cacheable`, annotation `@CacheEvict` demarcates methods that perform cache *eviction*, that is methods that act as triggers for removing data from the cache. Just like its sibling, `@CacheEvict` requires specifying one (or multiple) caches that are affected by the

action, allows a custom cache and key resolution or a condition to be specified but in addition, features an extra parameter `allEntries` which indicates whether a cache-wide eviction needs to be performed rather than just an entry one (based on the key):

```
@CacheEvict(value="books", allEntries=true)
public void loadBooks(InputStream batch)
```

This option comes in handy when an entire cache region needs to be cleared out - rather than evicting each entry (which would take a long time since it is inefficient), all the entries are removed in one operation as shown above. Note that the framework will ignore any key specified in this scenario as it does not apply (the entire cache is evicted not just one entry).

One can also indicate whether the eviction should occur after (the default) or before the method executes through the `beforeInvocation` attribute. The former provides the same semantics as the rest of the annotations - once the method completes successfully, an action (in this case eviction) on the cache is executed. If the method does not execute (as it might be cached) or an exception is thrown, the eviction does not occur. The latter (`beforeInvocation=true`) causes the eviction to occur always, before the method is invoked - this is useful in cases where the eviction does not need to be tied to the method outcome.

It is important to note that void methods can be used with `@CacheEvict` - as the methods act as triggers, the return values are ignored (as they don't interact with the cache) - this is not the case with `@Cacheable` which adds/updates data into the cache and thus requires a result.

30.3.4 @Caching annotation

There are cases when multiple annotations of the same type, such as `@CacheEvict` or `@CachePut` need to be specified, for example because the condition or the key expression is different between different caches. Unfortunately Java does not support such declarations however there is a workaround - using an *enclosing* annotation, in this case, `@Caching`. `@Caching` allows multiple nested `@Cacheable`, `@CachePut` and `@CacheEvict` to be used on the same method:

```
@Caching(evict = { @CacheEvict("primary"), @CacheEvict(value="secondary", key="#p0") })
public Book importBooks(String deposit, Date date)
```

30.3.5 @CacheConfig annotation

So far we have seen that caching operations offered many customization options and these can be set on an operation basis. However, some of the customization options can be tedious to configure if they apply to all operations of the class. For instance, specifying the name of the cache to use for every cache operation of the class could be replaced by a single class-level definition. This is where `@CacheConfig` comes into play.

```
@CacheConfig("books")
public class BookRepositoryImpl implements BookRepository {

    @Cacheable
    public Book findBook(ISBN isbn) {...}
}
```

`@CacheConfig` is a class-level annotation that allows to share the cache names, the custom `KeyGenerator`, the custom `CacheManager` and finally the custom `CacheResolver`. Placing this annotation on the class does not turn on any caching operation.

An operation-level customization will always override a customization set on `@CacheConfig`. This gives therefore three levels of customizations per cache operation:

- Globally configured, available for `CacheManager`, `KeyGenerator`
- At class level, using `@CacheConfig`
- At the operation level

30.3.6 Enable caching annotations

It is important to note that even though declaring the cache annotations does not automatically trigger their actions - like many things in Spring, the feature has to be declaratively enabled (which means if you ever suspect caching is to blame, you can disable it by removing only one configuration line rather than all the annotations in your code).

To enable caching annotations add the annotation `@EnableCaching` to one of your `@Configuration` classes:

```
@Configuration
@EnableCaching
public class AppConfig {
}
```

Alternatively for XML configuration use the `cache:annotation-driven` element:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cache="http://www.springframework.org/schema/cache"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/cache http://www.springframework.org/schema/cache

       <cache:annotation-driven />

</beans>
```

Both the `cache:annotation-driven` element and `@EnableCaching` annotation allow various options to be specified that influence the way the caching behavior is added to the application through AOP. The configuration is intentionally similar with that of `@Transactional`:

Table 30.2. Cache annotation settings

XML Attribute	Annotation Attribute	Default	Description
<code>cache-manager</code>	N/A (See <code>CachingConfigurer</code> javadocs)	<code>cacheManager</code>	Name of cache manager to use. A default <code>CacheResolver</code> will be initialized behind the scenes with this cache manager (or <code>cacheManager</code> if not set). For more fine-

				grained management of the cache resolution, consider setting the <i>cache-resolver</i> attribute.
cache-resolver	N/A (See <code>CachingConfigurer</code> javadocs)	A <code>SimpleCacheResolver</code> using the configured <code>cacheManager</code> .		The bean name of the <code>CacheResolver</code> that is to be used to resolve the backing caches. This attribute is not required, and only needs to be specified as an alternative to the <i>cache-manager</i> attribute.
key-generator	N/A (See <code>CachingConfigurer</code> javadocs)	<code>SimpleKeyGenerator</code>		Name of the custom key generator to use.
error-handler	N/A (See <code>CachingConfigurer</code> javadocs)	<code>SimpleCacheErrorHandler</code>		Name of the custom cache error handler to use. By default, any exception throw during a cache related operations are thrown back at the client.
mode	mode	proxy		The default mode "proxy" processes annotated beans to be proxied using Spring's AOP framework (following proxy semantics, as discussed above, applying to method calls coming in through the proxy only). The alternative mode "aspectj" instead weaves the affected classes with Spring's AspectJ caching aspect, modifying the target class byte code to apply to any kind of method call. AspectJ weaving requires <code>spring-aspects.jar</code> in

proxy-target-class

proxyTargetClass

false

the classpath as well as load-time weaving (or compile-time weaving) enabled. (See [the section called “Spring configuration”](#) for details on how to set up load-time weaving.)

Applies to proxy mode only. Controls what type of caching proxies are created for classes annotated with the `@Cacheable` or `@CacheEvict` annotations. If the `proxy-target-class` attribute is set to `true`, then class-based proxies are created. If `proxy-target-class` is `false` or if the attribute is omitted, then standard JDK interface-based proxies are created. (See [Section 9.6, “Proxying mechanisms”](#) for a detailed examination of the different proxy types.)

order

order

Ordered.LOWEST_PRECEDENCE

Defines the order of the cache advice that is applied to beans annotated with `@Cacheable` or `@CacheEvict`. (For more information about the rules related to ordering of AOP advice, see [the section called “Advice ordering”](#).) No specified ordering means that the AOP subsystem determines the order of



`<cache:annotation-driven/>` only looks for `@Cacheable/@CachePut/@CacheEvict/@Caching` on beans in the same application context it is defined in. This means that, if you put `<cache:annotation-driven/>` in a `WebApplicationContext` for a `DispatcherServlet`, it only checks for beans in your controllers, and not your services. See [Section 17.2, “The DispatcherServlet”](#) for more information.

Method visibility and cache annotations

When using proxies, you should apply the cache annotations only to methods with *public* visibility. If you do annotate protected, private or package-visible methods with these annotations, no error is raised, but the annotated method does not exhibit the configured caching settings. Consider the use of AspectJ (see below) if you need to annotate non-public methods as it changes the bytecode itself.



Spring recommends that you only annotate concrete classes (and methods of concrete classes) with the `@Cache*` annotation, as opposed to annotating interfaces. You certainly can place the `@Cache*` annotation on an interface (or an interface method), but this works only as you would expect it to if you are using interface-based proxies. The fact that Java annotations are *not inherited from interfaces* means that if you are using class-based proxies (`proxy-target-class="true"`) or the weaving-based aspect (`mode="aspectj"`), then the caching settings are not recognized by the proxying and weaving infrastructure, and the object will not be wrapped in a caching proxy, which would be decidedly *bad*.



In proxy mode (which is the default), only external method calls coming in through the proxy are intercepted. This means that self-invocation, in effect, a method within the target object calling another method of the target object, will not lead to an actual caching at runtime even if the invoked method is marked with `@Cacheable` - considering using the aspectj mode in this case.

30.3.7 Using custom annotations

Custom annotation and AspectJ

This feature only works out-of-the-box with the proxy-based approach but can be enabled with a bit of extra effort using AspectJ.

The `spring-aspects` module defines an aspect for the standard annotations only. If you have defined your own annotations, you also need to define an aspect for those. Check `AnnotationCacheAspect` for an example.

The caching abstraction allows you to use your own annotations to identify what method triggers cache population or eviction. This is quite handy as a template mechanism as it eliminates the need to duplicate cache annotation declarations (especially useful if the key or condition are specified) or if the foreign imports (`org.springframework`) are not allowed in your code base. Similar to the rest of the `stereotype` annotations,

`@Cacheable`, `@CachePut`, `@CacheEvict` and `@CacheConfig` can be used as meta-annotations, that is annotations that can annotate other annotations. To wit, let us replace a common `@Cacheable` declaration with our own, custom annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Cacheable(value="books", key="#isbn")
public @interface SlowService {
}
```

Above, we have defined our own `SlowService` annotation which itself is annotated with `@Cacheable` - now we can replace the following code:

```
@Cacheable(value="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

with:

```
@SlowService
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

Even though `@SlowService` is not a Spring annotation, the container automatically picks up its declaration at runtime and understands its meaning. Note that as mentioned above, the annotation-driven behavior needs to be enabled.

30.4 JCache (JSR-107) annotations

Since the Spring Framework 4.1, the caching abstraction fully supports the JCache standard annotations: these are `@CacheResult`, `@CacheEvict`, `@CacheRemove` and `@CacheRemoveAll` as well as the `@CacheDefaults`, `@CacheKey` and `@CacheValue` companions. These annotations can be used right the way without migrating your cache store to JSR-107: the internal implementation uses Spring’s caching abstraction and provides default `CacheResolver` and `KeyGenerator` implementations that are compliant with the specification. In other words, if you are already using Spring’s caching abstraction, you can switch to these standard annotations without changing your cache storage (or configuration, for that matter).

30.4.1 Features summary

For those who are familiar with Spring’s caching annotations, the following table describes the main differences between the Spring annotations and the JSR-107 counterpart:

Table 30.3. Spring vs. JSR-107 caching annotations

Spring	JSR-107	Remark
<code>@Cacheable</code>	<code>@CacheResult</code>	Fairly similar. <code>@CacheResult</code> can cache specific exceptions and force the execution of the method regardless of the content of the cache.
<code>@CachePut</code>	<code>@CachePut</code>	While Spring updates the cache with the result of the method invocation, JCache

requires to pass it as an argument that is annotated with `@CacheValue`. Due to this difference, JCache allows to update the cache before or after the actual method invocation.

`@CacheEvict`

`@CacheRemove`

Fairly similar. `@CacheRemove` supports a conditional evict in case the method invocation results in an exception.

`@CacheEvict(allEntries=true)`

`@CacheRemoveAll`

See `@CacheRemove`.

`@CacheConfig`

`@CacheDefaults`

Allows to configure the same concepts, in a similar fashion.

JCache has the notion of `javax.cache.annotation.CacheResolver` that is identical to the Spring's `CacheResolver` interface, except that JCache only supports a single cache. By default, a simple implementation retrieves the cache to use based on the name declared on the annotation. It should be noted that if no cache name is specified on the annotation, a default is automatically generated, check the javadoc of `@CacheResult#cacheName()` for more information.

`CacheResolver` instances are retrieved by a `CacheResolverFactory`. It is possible to customize the factory per cache operation:

```
@CacheResult(value="books", cacheResolverFactory=MyCacheResolverFactory.class)
public Book findBook(ISBN isbn)
```



For all referenced *classes*, Spring tries to locate a bean with the given type. If more than one match exists, a new instance is created and can use the regular bean lifecycle callbacks such as dependency injection.

Keys are generated by a `javax.cache.annotation.CacheKeyGenerator` that serves the same purpose as Spring's `KeyGenerator`. By default, all method arguments are taken into account unless at least one parameter is annotated with `@CacheKey`. This is similar to Spring's [custom key generation declaration](#). For instance these are identical operations, one using Spring's abstraction and the other with JCache:

```
@Cacheable(value="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@CacheResult(cacheName="books")
public Book findBook(@CacheKey ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

The `CacheKeyResolver` to use can also be specified on the operation, in a similar fashion as the `CacheResolverFactory`.

JCache can manage exceptions thrown by annotated methods: this can prevent an update of the cache but it can also cache the exception as an indicator of the failure instead of calling the method again. Let's assume that `InvalidIsbnNotFoundException` is thrown if the structure of the ISBN is invalid. This is a permanent failure, no book could ever be retrieved with such parameter. The following caches the exception so that further calls with the same, invalid ISBN, throws the cached exception directly instead of invoking the method again.

```
@CacheResult(cacheName="books", exceptionCacheName="failures"
    cachedExceptions = InvalidIsbnNotFoundException.class)
public Book findBook(ISBN isbn)
```

30.4.2 Enabling JSR-107 support

Nothing specific needs to be done to enable the JSR-107 support alongside Spring's declarative annotation support. Both `@EnableCaching` and the `cache:annotation-driven` element will enable automatically the JCache support if both the JSR-107 API and the `spring-context-support` module are present in the classpath.



Depending of your use case, the choice is basically yours. You can even mix and match services using the JSR-107 API and others using Spring's own annotations. Be aware however that if these services are impacting the same caches, a consistent and identical key generation implementation should be used.

30.5 Declarative XML-based caching

If annotations are not an option (no access to the sources or no external code), one can use XML for declarative caching. So instead of annotating the methods for caching, one specifies the target method and the caching directives externally (similar to the declarative transaction management [advice](#)). The previous example can be translated into:

```
<!-- the service we want to make cacheable -->
<bean id="bookService" class="x.y.service.DefaultBookService"/>

<!-- cache definitions -->
<cache:advice id="cacheAdvice" cache-manager="cacheManager">
    <cache:caching cache="books">
        <cache:cacheable method="findBook" key="#isbn"/>
        <cache:cache-evict method="loadBooks" all-entries="true"/>
    </cache:caching>
</cache:advice>

<!-- apply the cacheable behavior to all BookService interfaces -->
<aop:config>
    <aop:advisor advice-ref="cacheAdvice" pointcut="execution(* x.y.BookService.*(..))"/>
</aop:config>

<!-- cache manager definition omitted -->
```

In the configuration above, the `bookService` is made cacheable. The caching semantics to apply are encapsulated in the `cache:advice` definition which instructs method `findBooks` to be used for putting data into the cache while method `loadBooks` for evicting data. Both definitions are working against the `books` cache.

The `aop:config` definition applies the cache advice to the appropriate points in the program by using the AspectJ pointcut expression (more information is available in [Chapter 9, Aspect Oriented Programming with Spring](#)). In the example above, all methods from the `BookService` are considered and the cache advice applied to them.

The declarative XML caching supports all of the annotation-based model so moving between the two should be fairly easy - further more both can be used inside the same application. The XML based approach does not touch the target code however it is inherently more verbose; when dealing with classes with overloaded methods that are targeted for caching, identifying the proper methods does take an extra effort since the `method` argument is not a good discriminator - in these cases, the AspectJ pointcut can be used to cherry pick the target methods and apply the appropriate caching functionality. However through XML, it is easier to apply a package/group/interface-wide caching (again due to the AspectJ pointcut) and to create template-like definitions (as we did in the example above by defining the target cache through the `cache:definitions` `cache` attribute).

30.6 Configuring the cache storage

Out of the box, the cache abstraction provides several storages integration. To use them, one needs to simply declare an appropriate `CacheManager` - an entity that controls and manages `Cache`s and can be used to retrieve these for storage.

30.6.1 JDK ConcurrentMap-based Cache

The JDK-based `Cache` implementation resides under `org.springframework.cache.concurrent` package. It allows one to use `ConcurrentHashMap` as a backing `Cache` store.

```
<!-- simple cache manager -->
<bean id="cacheManager" class="org.springframework.cache.support.SimpleCacheManager">
  <property name="caches">
    <set>
      <bean class="org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean" p:
      <bean class="org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean" p:
    </set>
  </property>
</bean>
```

The snippet above uses the `SimpleCacheManager` to create a `CacheManager` for the two nested `ConcurrentMapCache` instances named *default* and *books*. Note that the names are configured directly for each cache.

As the cache is created by the application, it is bound to its lifecycle, making it suitable for basic use cases, tests or simple applications. The cache scales well and is very fast but it does not provide any management or persistence capabilities nor eviction contracts.

30.6.2 EhCache-based Cache

The EhCache implementation is located under `org.springframework.cache.ehcache` package. Again, to use it, one simply needs to declare the appropriate `CacheManager`:

```
<bean id="cacheManager"
  class="org.springframework.cache.ehcache.EhCacheCacheManager" p:cache-manager-ref="ehcach

<!-- EhCache Library setup -->
<bean id="ehcache"
  class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean" p:config-location="eh
```


This setup bootstraps the ehcache library inside Spring IoC (through the `ehcache` bean) which is then wired into the dedicated `CacheManager` implementation. Note the entire ehcache-specific configuration is read from `ehcache.xml`.

30.6.3 Guava Cache

The Guava implementation is located under `org.springframework.cache.guava` package and provides access to several features of Guava.

Configuring a `CacheManager` that creates the cache on demand is straightforward:

```
<bean id="cacheManager"
      class="org.springframework.cache.guava.GuavaCacheManager"/>
```

It is also possible to provide the caches to use explicitly. In that case, only those will be made available by the manager:

```
<bean id="cacheManager" class="org.springframework.cache.guava.GuavaCacheManager">
  <property name="caches">
    <set>
      <value>default</value>
      <value>books</value>
    </set>
  </property>
</bean>
```

The Guava `CacheManager` also supports custom `CacheBuilder` and `CacheLoader`. See the [Guava documentation](#) for more information about those.

30.6.4 GemFire-based Cache

GemFire is a memory-oriented/disk-backed, elastically scalable, continuously available, active (with built-in pattern-based subscription notifications), globally replicated database and provides fully-featured edge caching. For further information on how to use GemFire as a `CacheManager` (and more), please refer to the [Spring Data GemFire reference documentation](#).

30.6.5 JSR-107 Cache

JSR-107 compliant caches can also be used by Spring's caching abstraction. The JCache implementation is located under `org.springframework.cache.jcache` package.

Again, to use it, one simply needs to declare the appropriate `CacheManager`:

```
<bean id="cacheManager"
      class="org.springframework.cache.jcache.JCacheCacheManager"
      p:cache-manager-ref="jCacheManager"/>

<!-- JSR-107 cache manager setup -->
<bean id="jCacheManager" .../>
```

30.6.6 Dealing with caches without a backing store

Sometimes when switching environments or doing testing, one might have cache declarations without an actual backing cache configured. As this is an invalid configuration, at runtime an exception will be thrown since the caching infrastructure is unable to find a suitable store. In situations like this, rather than removing the cache declarations (which can prove tedious), one can wire in a simple, dummy cache that performs no caching - that is, forces the cached methods to be executed every time:

```
<bean id="cacheManager" class="org.springframework.cache.support.CompositeCacheManager">
  <property name="cacheManagers">
    <list>
      <ref bean="jdkCache"/>
      <ref bean="gemfireCache"/>
    </list>
  </property>
  <property name="fallbackToNoOpCache" value="true"/>
</bean>
```

The `CompositeCacheManager` above chains multiple `CacheManager`s and additionally, through the `fallbackToNoOpCache` flag, adds a *no op* cache that for all the definitions not handled by the configured cache managers. That is, every cache definition not found in either `jdkCache` or `gemfireCache` (configured above) will be handled by the no op cache, which will not store any information causing the target method to be executed every time.

30.7 Plugging-in different back-end caches

Clearly there are plenty of caching products out there that can be used as a backing store. To plug them in, one needs to provide a `CacheManager` and `Cache` implementation since unfortunately there is no available standard that we can use instead. This may sound harder than it is since in practice, the classes tend to be simple *adapters* that map the caching abstraction framework on top of the storage API as the `ehcache` classes can show. Most `CacheManager` classes can use the classes in `org.springframework.cache.support` package, such as `AbstractCacheManager` which takes care of the boiler-plate code leaving only the actual *mapping* to be completed. We hope that in time, the libraries that provide integration with Spring can fill in this small configuration gap.

30.8 How can I set the TTL/TTI/Eviction policy/XXX feature?

Directly through your cache provider. The cache abstraction is... well, an abstraction not a cache implementation. The solution you are using might support various data policies and different topologies which other solutions do not (take for example the JDK `ConcurrentHashMap`) - exposing that in the cache abstraction would be useless simply because there would no backing support. Such functionality should be controlled directly through the backing cache, when configuring it or through its native API.

[Prev](#)[Up](#)[Next](#)[29. Dynamic language support](#)[Home](#)[Part VII. Appendices](#)