Cache Plugin - Reference Documentation

Authors: Jeff Brown, Burt Beckwith

Version: 1.1.8

Table of Contents

1 Introduction To The Cache Plugin

The Grails Cache plugin provides powerful and easy to use caching functionality to Grails applications and

The plugin makes significant use of the caching abstraction provided by Spring 3.1. This user guide will f of that functionality specifically within the context of a Grails application. For information on the under Official Spring Documentation.

1.1 Change log

Version 1.0.1 - October 28, 2012

Version 1.0.0 - July 4, 2012

Version 1.0.0.RC1 - May 22, 2012

Version 1.0.0.M2 - May 12, 2012

2 Usage

The cache plugin adds Spring bean method call, controller action, and GSP page fragment and ten applications. You configure one or more caches in `Config.groovy` and/or one or more Groovy artifac in CacheConfig.groovy (for example FooCacheConfig.groovy, BarCacheConfig.groovy, and these can `grails-app/conf` (or a subdirectory if in a package) using an implementation-specific DSL, and an Spring beans (typically Grails services) or controllers) to be cached. You can also wrap GSP sections cached templates.

There are three annotations; Cacheable, CachePut, and CacheEvict. You use @Cacheable to mark a m check the cache for a pre-existing result, or generate a new result and cache it. Use @CachePut to mark a r always be evaluated and store its result in the cache regardless of existing cache values. And use @Cac (either fully or partially) to force the re-evaluation of previously cached results. The annotations are base the same name from Spring (Cacheable, CachePut, and CacheEvict) and support the same syntax bu functionality in the future.

When using distributed caching (such as ehcache with distributed cache enabled, or redis with multiple in running against one redis instance), all classes that use annotation caching or XML caching should method. The hash code of the object with the method marked as being cacheable is included in the cac `hashCode` implementation will vary each time the application is run. Overriding `hashCode` ensures applications will appropriately share cache keys.

This 'core' cache plugin uses an in-memory implementation where the caches and cache manager are `java.util.concurrent.ConcurrentMap`. This is fine for testing and possibly for low-traff consider using one of the extension plugins if you need clustering, disk storage, persistence betv configurability of features like time-to-live, maximum cache size, etc. Currently the extension plugins cache-redis, and cache-gemfire.

2.1 Configuration

Config.groovy and artifact files

The caching configuration can be specified in `Config.groovy` or *CacheConfig.groovy files. B `environments` blocks for environment-specific configuration, and you can specify the loading orde overriding values. One example of this might be a plugin that specifies a known load order, allowing you t your file and override some or all of the plugin's configuration.

There are a few configuration options for the plugin; these are specified in `Config.groovy`.

Property	Default	Description
grails.cache.enabled	`true`	Whether to enable the plugin
grails.cache.proxyTargetClass	`false`	From the Spring Javadoc: "By defau as JDK proxies. This may cause so injecting objects as concrete classe To overcome this restriction `proxy-target-class` attribute result in class-based proxies being cr
grails.cache.aopOrder	Ordered.LOWEST_PRECEDENCE	From the Spring docs: "Defines the that is applied to beans annotate @CacheEvict. No specified orderi subsystem determines the order of th
grails.cache.clearAtStartup	`false`	Whether to clear all caches at startup
grails.cache.keyGenerator	"customCacheKeyGenerator"	Replace the key generator with your i s `grails.cache.keyGenerator` , you would declare som `myCacheKeygen(my.company` resources.groovy. Note: MyKey KeyGenerator.

2.2 Cache DSL

The cache implementation used by this plugin is very simple, so there aren't many configuration options (implementation for example, where you have fine-grained control over features like overflowing to di maximum size of caches, etc.) So there aren't many supported options in the cache configuration DSL, alt parser is lenient and just logs warnings if you specify options that aren't understood. This lets you shar applications that use different plugins.

Since there is no way to configure "time to live" with this plugin, all cached items have no t remain cached until either the JVM restarts (since the backing store is in-memory) or t partially or fully cleared (by calling a method or action annotated with @Cac programmatically).

You specify the cache configuration in `Config.groovy` under the `grails.cache.config` key, for

```
grails.cache.config = {
   cache {
      name 'message'
   }
   cache {
      name 'maps'
   }
}
```

or in a *CacheConfig.groovy file in the `grails-app/conf` directory under the `config` key, for exam

```
config = {
   cache {
      name 'message'
   }
   cache {
      name 'maps'
   }
}
```

Both of these will create two caches, one with name "message" and one with name "maps". You can also DSLs and they will be ignored, for example:

```
grails.cache.config = {
   cache {
      name 'message'
      eternal false
      overflowToDisk true
      maxElementsInMemory 10000
      maxElementsOnDisk 10000000
   }
   cache {
      name 'maps'
   }
}
```

This configuration results in the same caches as the simpler one.

Order

You can configure your cache definitions to be loaded before or after others by setting the `order` attrib higher numbers are loaded later, so these can override previously-configured values, although there is caches or cache attributes, only adding or overriding:

```
order = 2000

config = {
   cache {
      name 'message'
   }
   cache {
      name 'maps'
   }
}
```

2.3 Annotations

The Cacheable and CacheEvict annotations provided by the plugin have counterparts with the same names the Spring documentation for their usage and allowed syntax.

Service method caching

Given this simple service, you can see that the `getMessage` method is configured to cache the results ir
The `title` parameter will be used as the cache key; if there were multiple parameters they would be co
you can always specify the key using the Spring SpEL support. The `save` method is configured as one t
the cache. There is no need to clear the entire cache in this case; instead any previously cached item with t
will be replaced with the current `Message` instance.

```
package com.yourcompany

import grails.plugin.cache.CacheEvict
import grails.plugin.cache.Cacheable

class MessageService {

@Cacheable('message')
   Message getMessage(String title) {
      println 'Fetching message'
      Message.findByTitle(title)
   }

@CachePut(value='message', key='#message.title')
   void save(Message message) {
      println "Saving message $message"
      message.save()
   }

@CacheEvict(value='message', key='#message.title')
   void delete(Message message) {
      println "Deleting message $message"
      message.delete()
   }
}
```

Note that you could also use @CacheEvict for the `save` method, which would remove the old cached
current value.

This service works with the `Message` domain class:

```
package com.yourcompany

class Message implements Serializable {

private static final long serialVersionUID = 1

String title
   String body

String toString() {
      "$title: $body"
   }
}
```

Note that for in-memory cache implementations it's not required that the objects being cached implement
you use an implementation that uses Java serialization (for example the Redis plugin, or the Ehcach
configured clustered caching) you must implement `Serializable`.

To test this out, be sure to define a `"message"` cache in `Config.groovy` and save and retrieve `Mes`
service. There are `println` statements but you can also turn on SQL logging to watch the database acces
instances that aren't cached yet, and you shouldn't see database access for cached values.

Controller action caching

In addition to caching Spring bean return values, you can also cache responses for web requests using th that since caching is implemented only for methods (Spring creates a proxy for your cached class in the s transactional proxy to start, commit, and roll back transactions for transactional Grails services) so yo closures. This doesn't fail silently; your controller class will not compile since the annotations are only a methods; since Closures are fields, the annotations aren't valid.

For example, in this controller the `lookup` action will use the `"message"` cache, so the first time you c the output from the `println` statement but subsequent calls won't execute and you'll see the cached res call the `evict` action the entire cache will be cleared (because of the `allEntries=true` attribute):

```
package com.yourcompany

import grails.plugin.cache.CacheEvict
import grails.plugin.cache.Cacheable

class TestController {

@Cacheable('message')
   def lookup() {
      // perform some expensive operations
      println "called 'lookup'"
   }

@CacheEvict(value='message', allEntries=true)
   def evict() {
      println "called 'evict'"
   }
}
```

Caching of dynamically scaffolded actions is not supported. If the scaffolding templates a with `grails install-templates` and cache related annotations are added to met controller template, those annotations will only be relevant to generated scaffolding, n scaffolding.

If you can't use annotations

Annotations aren't required, they're just the most convenient approach for configuration. If you like semantics in `grails-app/conf/spring/resources.groovy` (or `resources.xml` if you useful if you want to apply caching but can't edit the code to add annotations (for example if you have com

This Spring BeanBuilder DSL code will configure the same behavior as the two annotations in the exampl

```
beans = {

xmlns cache: 'http://www.springframework.org/schema/cache'
   xmlns aop: 'http://www.springframework.org/schema/aop'

cache.'advice'(id: 'messageServiceCacheAdvice',
                  'cache-manager': 'grailsCacheManager') {
      caching(cache: 'message') {
         cacheable(method: 'getMessage')
         'cache-evict'(method: 'save', key: '#message.title')
      }
   }

// apply the cacheable behavior to MessageService
   aop.config {
      advisor('advice-ref': 'messageServiceCacheAdvice',
              pointcut: 'execution(* com.yourcompany.MessageService.*(..))')
   }
}
```

2.4 CacheManager

The plugin registers an instance of the CacheManager iterface as the `grailsCacheManager` Spring b using dependency injection.

The most common method you would call on the `grailsCacheManager` is `getCache(String r` instance programmatically. This shouldn't be needed often however. From the `Cache` instance you can a cache implementation using `cache.getNativeCache()`.

3 GSP Cache Tags

The plugin provides GSP tags which are useful for caching the result of evaluating sections of markup. result of evaluating sections of markup to be cached so subsequent renderings of the same markup do markup being evaluated again.

See the documentation for the block and render tags for more details.

4 Grails Cache Admin Service

The plugin provides a service named `GrailsCacheAdminService` which supports various methods f

4.1 Clearing Caches

There are methods in GrailsCacheAdminService for clearing the caches used by the block and render tags.

```
class ReportingController {

def grailsCacheAdminService

def report() {
        // clear the cache used by the blocks tag…
        grailsCacheAdminService.clearBlocksCache()

// clear the cache used by the render tag…
        grailsCacheAdminService.clearTemplatesCache()

…
    }
}
```

5 Implementation Details

All of the plugin's classes are designed for extensibility; the classes are all public, and fields and meth protected. Consider subclassing existing classes to reuse as much as possible instead of completely rewritir

Cache manager

The core cache plugin registers a `grailsCacheManager` Spring bean, and the extension plugins repla creates and manages caches for that implementation. The default implementation `grails.plugin.cache.GrailsConcurrentMapCacheManager` which `grails.plugin.cache.GrailsConcurrentMapCache` as its cache implementat `java.util.concurrent.ConcurrentHashMap` to store cached values.

You can customize the cache manager by replacing the `grailsCacheManager` Spring bean in `res` your own; either subclass `GrailsConcurrentMapCacheManager` (e.g. t `createConcurrentMapCache()` method) or by implementing the `grails.plugin.cache.G` interface.

Controller caching

The controller caching is implemented with a filter registered as `grailsCacheFilter` in `web.xml` Spring bean of the same name. The implementation `grails.plugin.cache.web.filter.simple.MemoryPageFragmentCachingFilter`.

The content that is cached is the response generated by GSP (or directly by the controller if programmatically) before Sitemesh applies its template(s).

Key generation

Controller caching uses a key generator, a class that im `grails.plugin.cache.web.filter.WebKeyGenerator` interface (by `grails.plugin.cache.web.filter.DefaultWebKeyGenerator`). This is reg `webCacheKeyGenerator` Spring bean, so customizing the key generation is simply a r `DefaultWebKeyGenerator` or re-implementing the interface and registering your own `webCacheK` `resources.groovy`.

Fragment caching

You can cache partial GSP page sections with the `<cache:block>` tag. You can specify a key when general unnecessary. This is because the block will be rendered with its own Closure, and the default ke name. This is unique since the closures aren't re-used; for example these two blocks will be cached indepe GSP:

```
<cache:block>
foo
</cache:block>

<cache:block>
bar
</cache:block>
```

You can cache the content of templates with the `<cache:render>` tag. You can specify a key when u `block` tag, it's in general unnecessary because the default key is the full template class name.

Service caching

You can cache the return value of a service method by annotating it with `Cacheable`.

Key generation

The default implementation of the `org.springframework.cache.interceptor.KeyGenerat` for service method calls is `org.springframework.cache.interceptor.DefaultKeyGener` if there is no `key` attribute specified in the annotation for the method. It generates a numeric key, with the

```
public Object generate(Object target, Method method, Object… params) {
   if (params.length == 1) {
      return (params[0] == null ? 53 : params[0]);
   }
if (params.length == 0) {
      return 0;
   }
int hashCode = 17;
   for (Object object : params) {
      hashCode = 31 * hashCode + (object == null ? 53 : object.hashCode());
   }
   return hashCode;
}
```

This is very generic and somewhat risky, since two no-arg methods that use the same cache will store va (0), and different methods with similar signatures can easily generate the same key for different return va specify the `key` attribute in the annotation, or use separate caches.

DSL parsing

The cache plugin's DSL is very basic; only the cache name can be specified. But you could extend customized the cache or cache manager implementation, although a new plugin would probably make mor `grailsCacheConfigLoader` Spring bean in `resources.groovy`. The default in `grails.plugin.cache.ConfigLoader`.

Annotation SpEL expression evaluator

You can extend or customize what is SpEL expressions are supported by re-defining the `webExpress` bean in `resources.groovy`. The default implementation is an `grails.plugin.cache.web.filter.ExpressionEvaluator`.