CACHES

DIGITAL EXPERIENCE MANAGER 7.1.1

SUMMARY

1 INTRODUCTION

The cache has been refactored in Jahia Digital Experience Manager.

The HTML Cache in Jahia Digital Experience Manager are coded as rendering filters.

The default filter is the “AggregateCacheFilter” class.

This filter cache each module separately and then aggregates all modules on rendering time to deliver the full page to the client.

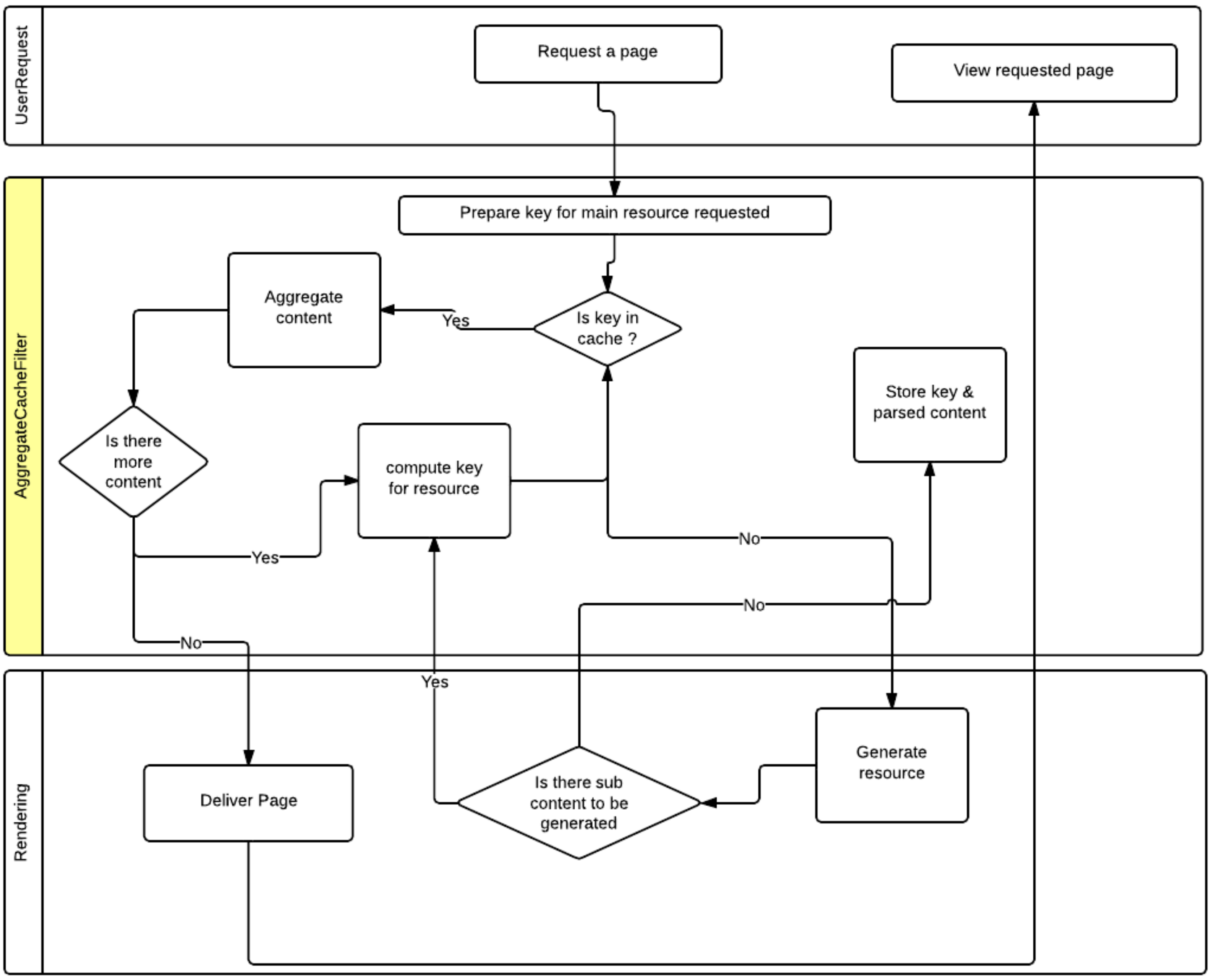
As an example imagine a "last news module" that display the latest 5 news of a site.

The aggregate on rendering will ask for the module last news for the rendering of each discrete news, those news will be cached separately and the last news module will only contains its own html and references to the displayed news.

When subsequent rendering is asked the last news modules search for the expected news in the cache. If they are found it aggregates the content in the output, otherwise it asks for rendering only the missing part.

This means that your news can be cached for hours but your last news request be cache for only 5-10mn (depending on the frequency of updates on your site (**no cache is never a good idea**)), older new will be delivered from the cache and only the new one will fully rendered by the engine.

1 CACHE PROCESS IN DIGITAL EXPERIENCE MANAGER



2 WHAT CACHE FRAMEWORK IS DIGITAL EXPERIENCE MANAGER USING?

We are using EHCache in its version 2.8.1.

You can configure it in the file WEB-INF/classes/ehcache-jahia.xml or

WEB-INF/classes/ehcache-jahia_cluster.xml.

3 INVALIDATION OR EXPIRATION?

Digital Experience Manager is using both modes for its caches, by default a fragment will have its expiration set to 30 minutes.

If during this time the element is updated or deleted or child node is added/removed the element will be invalidated from the cache on the fly.

4 OVERRIDING THE DEFAULT EXPIRATION?

You can override the default expiration in two ways.

The easiest and more end-user friendly method, is to allow the users to specify the expiration time directly from the end user interface.

User need specific permissions to access to this parameter.

To enable manual setup of expiration delay in the engines, you must apply the mixin type *jmix:cache* to the targeted object definition.

```
[jnt:lastNews] > jnt:content, jmix:list, mix:title, jmix:queryContent, jmix:cache
 - maxNews (long) = 10 indexed=no
 - filter (reference, category[autoSelectParent=false])
```

You can also have an hardcoded expiration on a per template basis in a template properties file.

Example you can create the jnt_banner/html/banner.properties file in default module to make banner cached only 30s.

```
#Make banner non cacheable
cache.expiration = 30
```

Expiration delays are expressed in seconds.

Note that if you have an alternative view on your content, you can specify a properties file for this view that will override the default ones if present.

For instance, the jnt_user/html/user.welcome.properties file in default module overrides the cache for the "welcome" view of the user module.

5 AUTOMATIC/MANUAL MANAGEMENT OF DEPENDENCY FOR AN ELEMENT

Dependencies of an html element define for which nodes updates this element should be flushed. The system tries to handle most of the dependencies by itself. Automatically the system detects implicit dependencies like parent/childs.

If an existing child is updated, only his html fragment will be flushed from the cache. If we create or delete a new child, the system will also flush the parent html fragment. So the system handles automatically all standard parent/child relations.

Now if you have some bound components in your page, the system handles it automatically by making those elements dependent of the bound component for the key computing.

The system will also parse your html to find all the links you have in your module html to other nodes (useful for rich text where your editors will have entered links to pages or contents you couldn't know in advance) and define the corresponding dependencies.

This parsing is executed by the CacheUrlDependenciesParserFilter.

So if in your templates you have defined a template for a news object that add a rateable module bound to this news, then the cache will reflect that by caching the rendering of you rateable module per main resource and adding a dependency to the news.

This way we avoid to display the same rateable module for all news, but we have one per news.

You can have some of those dependencies set using the template properties file.

You can also define directly in your script file (jsps, etc.) the dependencies you want to add to your fragment. As an example, we can look at the comments component that you can bind to any object in Digital Experience Manager. This comments components works in two parts.

First part is the display of the form to add a comment, second part is the display of the comments list. This form on first submission will create a subfolder under the main resource called comments.

So on the creation of the first comment the display list will be correctly flushed as the system as automatically created a dependency between the fragment and the main resource.

By adding a child under the main resource (comments node) we will flush all html fragments having a dependency to the main resource.

But for subsequent submission of new comments, we do not update anymore the main resource but only the subnode comments, so in our script we have to tell the system to flush this html fragment when the main resource subnode comments is updated.

```
<jcr:node var="comments" path="${boundComponent.path}/comments"/>
<c:if test="${not empty comments}">
   <template:addCacheDependency node="${comments}"/>
   <template:module node="${comments}" />
</c:if>
```

Here what you have to keep in mind is that if your script loads another node than the current node or the bound one then you will have to add a dependency manually.

You can also define dependencies based on some regular expression, this is really useful for html fragment that are using search queries to display content. As a rule, if you are using a query and that query have a constraint on descendant nodes or children node then you should have a regexp dependency on that path

Here an example from the blogs application.

```
<query:definition var="result"
            statement="select * from [jnt:blogPost] as blogPost  where
    isdescendantnode(blogPost, ['${renderContext.mainResource.node.path}']) order by
    blogPost.[jcr:lastModified] desc" limit="20"/>


<template:addCacheDependency
    flushOnPathMatchingRegexp="\\\\Q${renderContext.mainResource.node.path}\\\\E/.*/c
    omments/.*"/>
```

This fragment will be flushed for any change on any nodes down to two sub level of the main resource.

The \\\\Q and \\\\E are here to define an escape sequence so that whatever the path value it will be interpreted literally (<u>This should be put in all your regexp encapsulating an unknown path</u>).

6 CACHING IN MODES OTHER THAN LIVE

The AggregateCacheFilter is configured to only work for the live workspace in live mode.

From 6.6.1.5 onwards Jahia provides a second filter, which can be used also to cache fragments in the default workspace to be used in other modes (e.g. contribute, edit or wise mode).

This is the ExpiringCacheFilter, which only caches fragment, which have overridden the default expiration value. The fragments are thus only flushed on expiration and not by resolving dependencies like it is done in the AggregateCacheFilter.

This cache is not activated out-of-the-box (except for wise mode if the Jahia Wise templates are installed). In order to activate it for contribute mode, you need to define the filter in a Spring configuration file, like this:

```
<bean id="contributeModeCacheFilter"
    class="org.jahia.services.render.filter.cache.ExpiringCacheFilter">
   <property name="disabled" value="${disableOutputCache:false}"/>
   <property name="priority" value="16" />
   <property name="description" value="Module content caching filter for contribute
    mode (caches only content with expiration set)." />
   <property name="cacheProvider" ref="ModuleCacheProvider"/>
   <property name="skipOnTemplateTypes" value="json"/>
   <property name="skipOnConfigurations" value="include,wrapper,option"/>
   <property name="applyOnModes" value="contribute"/>
   <property name="generatorQueue" ref="moduleGeneratorQueue"/>
   <property name="moduleParamsProperties">
       <map key-type="java.lang.String" value-type="java.lang.String">
           <entry key="j:subNodesView" value="subNodesView"/>
       </map>
   </property>
</bean>
```

7 CACHE KEY GENERATION

The cache key is generated by multiple implementations of CacheKeyPartGenerator .

The core defines different key parts like "workspace", "language", "node path", "template", "templateType", "acls", "queryString", which are usually enough to create a unique key for each fragment.

The default generators are defined in the file applicationcontext-cache.xml.

```
<bean id="cacheKeyGenerator"
    class="org.jahia.services.render.filter.cache.DefaultCacheKeyGenerator">
        <property name="partGenerators">
            <list>
                <bean
    class="org.jahia.services.render.filter.cache.LanguageCacheKeyPartGenerator"/>
                <bean
    class="org.jahia.services.render.filter.cache.PathCacheKeyPartGenerator"/>
                <bean
    class="org.jahia.services.render.filter.cache.TemplateCacheKeyPartGenerator"/>
                <bean
    class="org.jahia.services.render.filter.cache.TemplateTypeCacheKeyPartGenerator"/
    >
                <ref bean="aclCacheKeyPartGenerator"/>
                <bean
    class="org.jahia.services.render.filter.cache.ContextCacheKeyPartGenerator"/>
                <bean
    class="org.jahia.services.render.filter.cache.WrappedCacheKeyPartGenerator"/>
                <bean
    class="org.jahia.services.render.filter.cache.CustomCacheKeyPartGenerator"/>
                <bean
    class="org.jahia.services.render.filter.cache.QueryStringCacheKeyPartGenerator"/>
                <bean
    class="org.jahia.services.render.filter.cache.TemplateNodesCacheKeyPartGenerator"
    />
                <bean
    class="org.jahia.services.render.filter.cache.ResourceIDCacheKeyPartGenerator"/>
                <bean
    class="org.jahia.services.render.filter.cache.InAreaCacheKeyPartGenerator"/>
                <bean
    class="org.jahia.services.render.filter.cache.SiteCacheKeyPartGenerator"/>
                <bean
    class="org.jahia.services.render.filter.cache.ModuleParamsCacheKeyPartGenerator"/
    >
                <bean
    class="org.jahia.services.render.filter.cache.TemplateDefaultViewCacheKeyPartGene
    rator"/>
            </list>
        </property>
   </bean>
```

However, if you need to customize the key by adding specific values, a spring bean implementing CacheKeyPartGenerator can be added to any module, and will impact all keys generated for the cache.

Each key that is generated looks like that:

```
en@@/sites/ACMESPACE/contents/projects-news/news_36-
    1@@medium@@html@@privileged%2Csite-
    administrator:%2Fsites%2FACMESPACE|@@module@@false@@@@{}@@@@998a823d-f275-4a10-
    aef4-0ae9d1ea5677@@@@ACMESPACE:null@@{}@@
```

Each parameter of the key is separated by @@ so be sure to avoid this symbols inside your values, empty part are possible.

Keys are generated during the prepare and execute phase of the AggregateCacheFilter and should give the same result on each call, otherwise Jahia will log a warning like “Key generation does not give the same result after execution…” with the involved keys. This issue can lead to an overhead in cache generation as fragments might not be found in the cache.

Here an example of a very simple part, the LanguageCacheKeyPartGenerator:

```
public class LanguageCacheKeyPartGenerator implements CacheKeyPartGenerator {
   @Override
   public String getKey() {
       return "language";
   }


   @Override
   public String getValue(Resource resource, RenderContext renderContext, Properties
    properties) {
       return resource.getLocale().toString();
   }


   @Override
   public String replacePlaceholders(RenderContext renderContext, String keyPart) {
       return keyPart;
   }


}
```

As you can see this one return the locale of the current resource as a String when the key is generated (getValue method is called in the prepare and execute phase). The replacePlaceholders method is called by Jahia when finding subfragment in a fragment, we then asked each key part to generate the right value if needed and then check if the key is found in the cache (compute key for resource phase in the diagram).

Another example is the path part of the key that can take into account the main resource requested by the user:

```
public class PathCacheKeyPartGenerator implements CacheKeyPartGenerator {
   public static final String MAIN_RESOURCE_KEY = "_mr_";


   @Override
   public String getKey() {
       return "path";
   }


   @Override
   public String getValue(Resource resource, RenderContext renderContext, Properties
    properties) {
       StringBuilder s = new StringBuilder(resource.getNode().getPath());
       if ("true".equals(properties.getProperty("cache.mainResource"))) {
           s.append(MAIN_RESOURCE_KEY);
       }
       return s.toString();
   }


   @Override
   public String replacePlaceholders(RenderContext renderContext, String keyPart) {
       return StringUtils.replace(keyPart, MAIN_RESOURCE_KEY,
               renderContext.getMainResource().getNode().getCanonicalPath() +
    renderContext.getMainResource().getResolvedTemplate());
   }
}
```

In this one every time we find a fragment in a content we replace the value of _mr_ in the path by the path of the current main resource and then jahia check in the cache if this fragment has already been generated and its still in the cache or needs to be generated (expired or non existing).

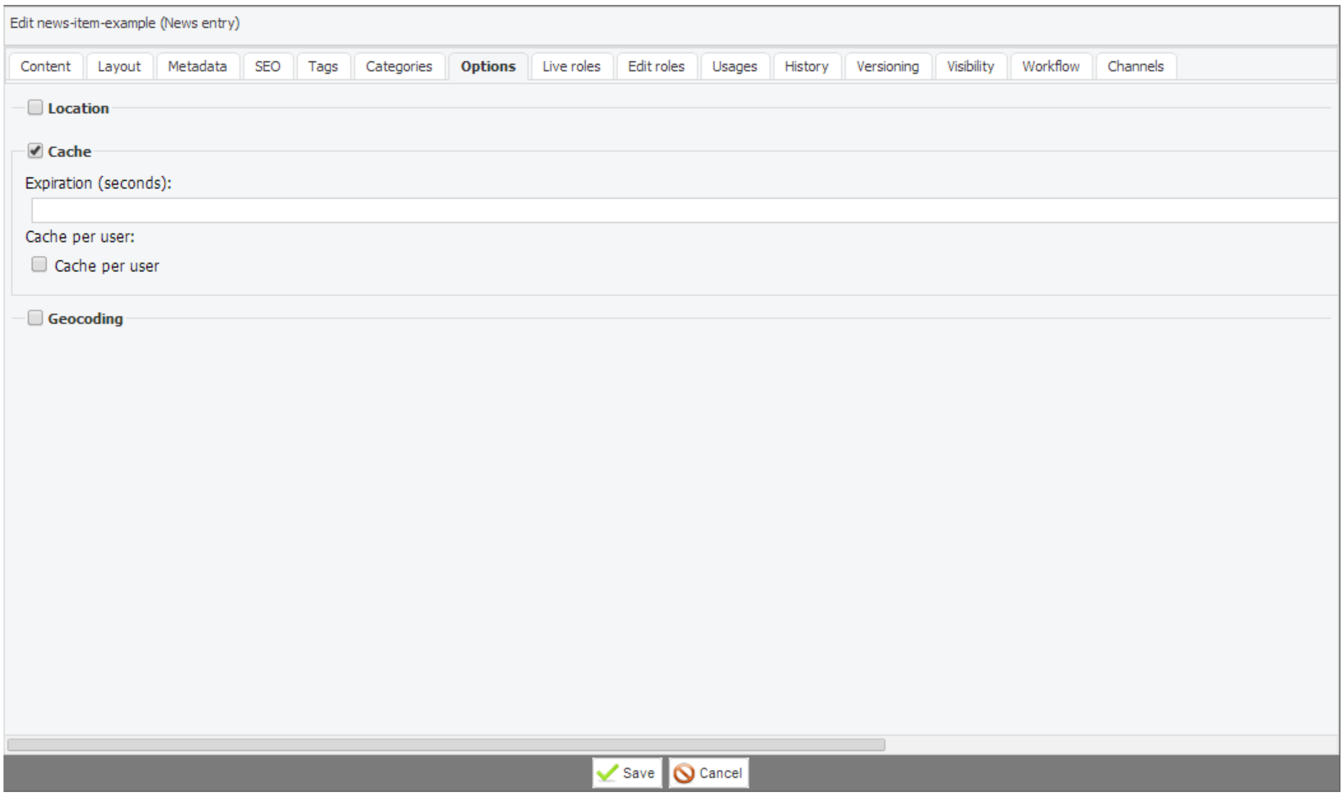
7.1 ACLS IN CACHE KEYS

Upon a request if content is not cached we search for all the acls that can apply for the user directly. Then we look up all the acls of the groups in its membership list. We use all this to build a map of acls per path as acls are applied for a path.

Groups and Users Acls are stored in local cache that are flushed when an acl is updated on the platform.

7.2 CUSTOM ELEMENTS IN KEYS

The custom cache key part allows you to put some element in the request in an attribute named “module.cache.additional.key”. This element has to be present in the request in the prepare phase of the AgregateCacheFilter so it means it has to be set before (higher priority filter, like the ChannelFilter for example, which is using that mechanism to switch cache between different channels). The value of the custom parts are return as is in the replacePlaceholders phase, so those value can not depends on some request parameters?


Edit news-item-example (News entry)

Options tab:

- [ ] Location
- [✓] Cache
  - Expiration (seconds):
  - Cache per user: [ ] Cache per user
- [ ] Geocoding