

CACHES

DIGITAL EXPERIENCE MANAGER 7.2

TABLE OF CONTENT

1	INTRODUCTION	3
2	CACHE PROCESS IN DIGITAL EXPERIENCE MANAGER	5
2.1	Rendering scripts execution with legacy implementation	5
2.2	Rendering scripts execution with new implementation	6
2.3	Detailed flow of aggregation/Cache.....	7
3	WHAT CACHE FRAMEWORK IS DIGITAL EXPERIENCE MANAGER USING?	9
4	INVALIDATION OR EXPIRATION?	10
5	OVERRIDING THE DEFAULT EXPIRATION?	11
6	AUTOMATIC/MANUAL MANAGEMENT OF DEPENDENCY FOR AN ELEMENT	12
7	FRAGMENT KEY GENERATION.....	14
7.1	Type of keys	14
7.2	Cache Key Part Generator	14
7.3	ACLS in cache keys	18
7.4	Custom elements in keys.....	19
7.5	New implementation specificities	20

1 INTRODUCTION

In Jahia Digital Experience Manager, the HTML Cache is implemented with rendering filters caching each module separately and then aggregating all the 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.

Jahia Digital Experience Manager 7.2 provides a new implementation for the aggregation and cache mechanism, based on two filters (AggregateFilter and CacheFilter) and uses a new aggregation flow.

Why a new implementation for Digital Experience Manager 7.2 ? Because the legacy implementation has one defect which is that a parent fragment is put in cache only when the sub fragments are rendered, and cached.

This simple sentence resume by itself why it was necessary to change this.

As an example, imagine a user (user 1) is requesting a page of the site, this page is using a template named "2 columns". 2 seconds after another user (user 2) is requesting another page of the site, this page is using the same template ("2 columns"). For some reason the page requested by user 1 is taking 2min to be displayed, because this page contains somewhere one fragment doing an external call to an external API and there is some connection issue.

The effect of this is that user 2 is blocked during 2 min, because user 1 is generating the template fragment. and the template fragment will be put in cache only when all the sub fragments of the page are rendered and cached.

You can imagine worst scenarios with a lot of users.

With the new implementation, this can't happen anymore because now a parent fragment is put in cache directly, then the sub fragments are rendered and cached.

2 CACHE PROCESS IN DIGITAL EXPERIENCE MANAGER

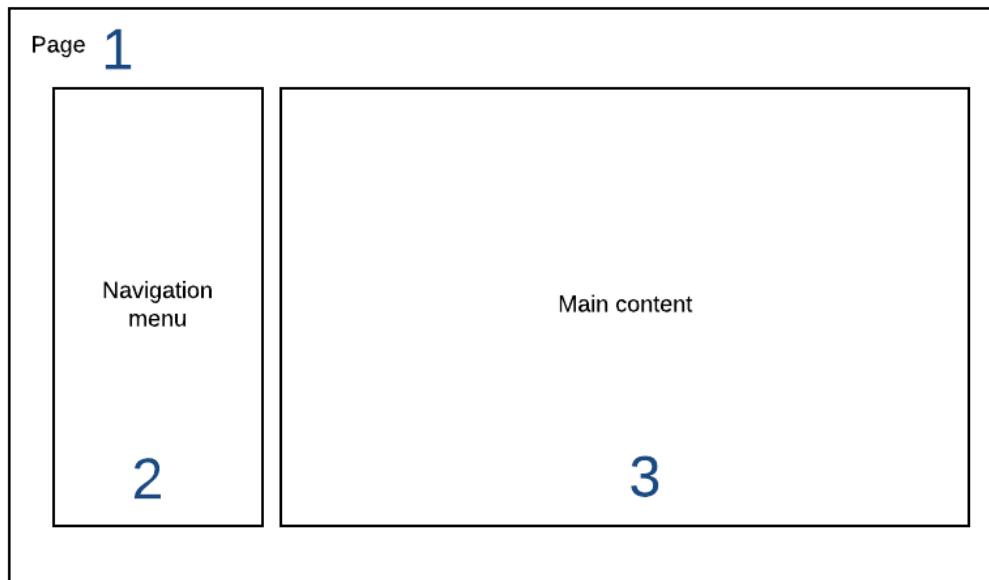
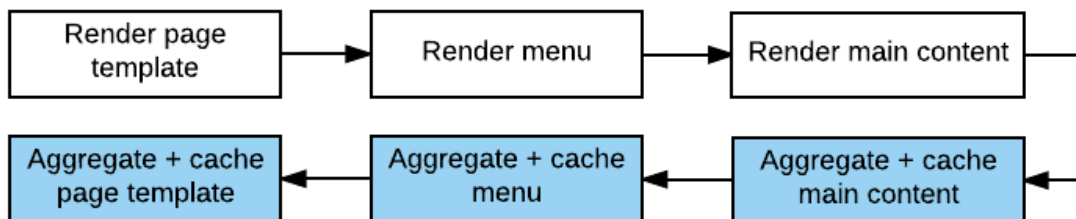


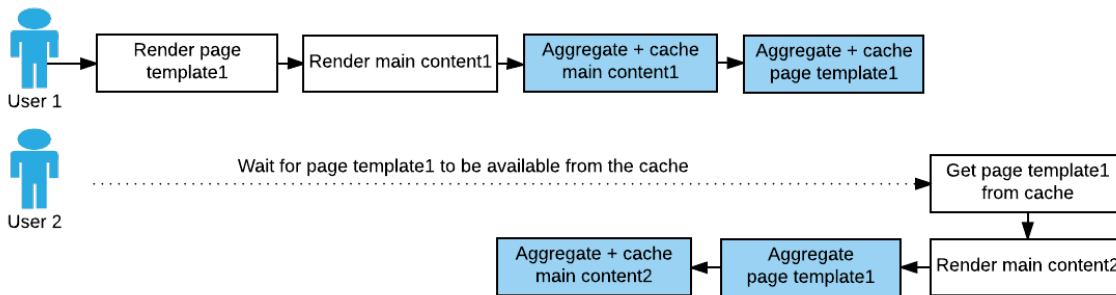
Figure 1 Example of page structure

2.1 RENDERING SCRIPTS EXECUTION WITH LEGACY IMPLEMENTATION

The legacy cache implementation rendered all the fragments of the page before aggregating them and storing them in the cache. This implementation is great because the fragment aggregation was done at the end of the rendering pipeline, thus allowing to have a single page context for the whole process. It made things easier to save parameters in the page context and access them in other execution scripts.



The downside of this method is that while the entire page is rendered, no fragment gets stored in the cache. It is not a problem for a single user on the platform, but can become an issue when concurrent users access shared resources on the site. (In the above example, page templates and navigation menus are shared resources).

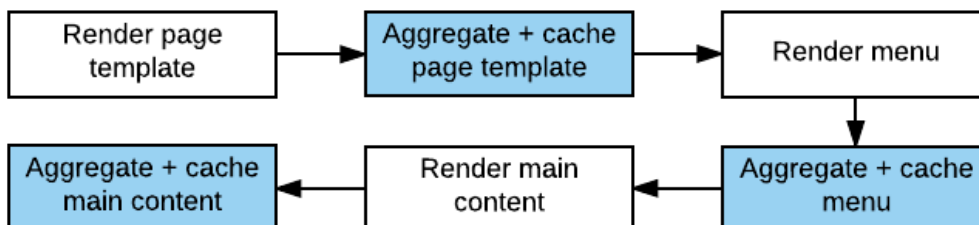


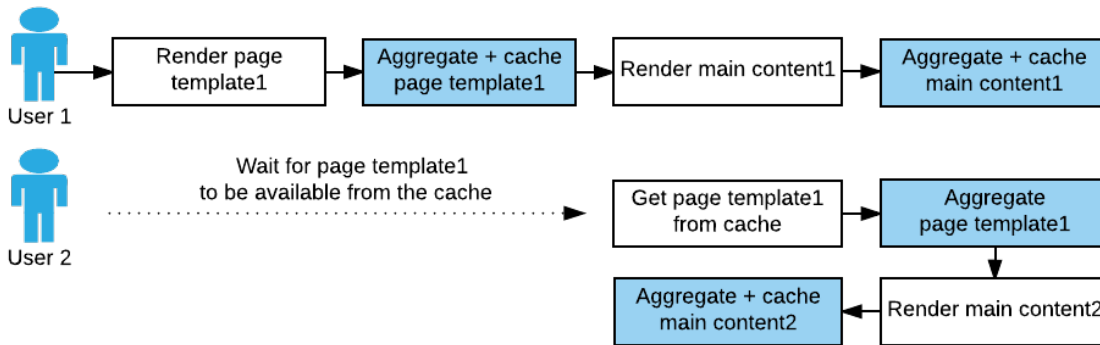
This strategy is not ideal from a performance standpoint. It was decided to move away from it and allow a new implementation in DX 7.2.

2.2 RENDERING SCRIPTS EXECUTION WITH NEW IMPLEMENTATION

Instead of caching and aggregating fragment at the end of the rendering pipeline, each fragment is now cached and aggregated right after it is generated. It is then made available for all requests right away.

Now, since the aggregation is done after each fragment generation, variables cannot be passed implicitly between views anymore. (Fragments aren't aggregated on the fly anymore, but extracted from the cache as they are needed, thus losing variables passed implicitly)





2.3 DETAILED FLOW OF AGGREGATION/CACHE

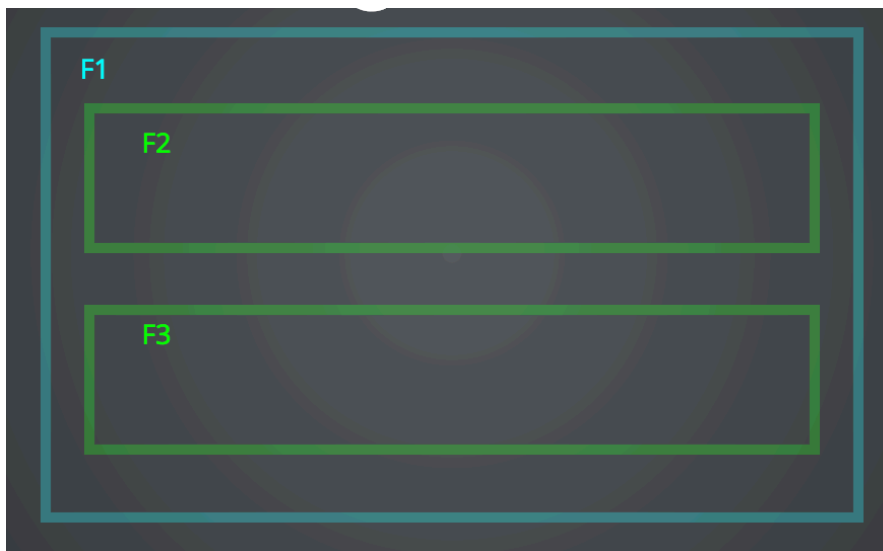


Figure 2 Example of page structure

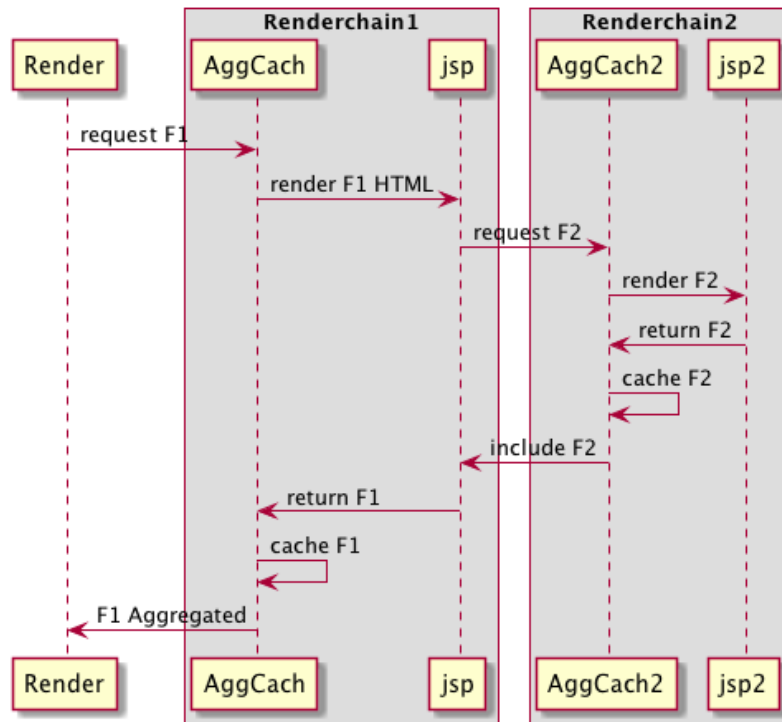


Figure 3 Legacy implementation flow (using one render filter)

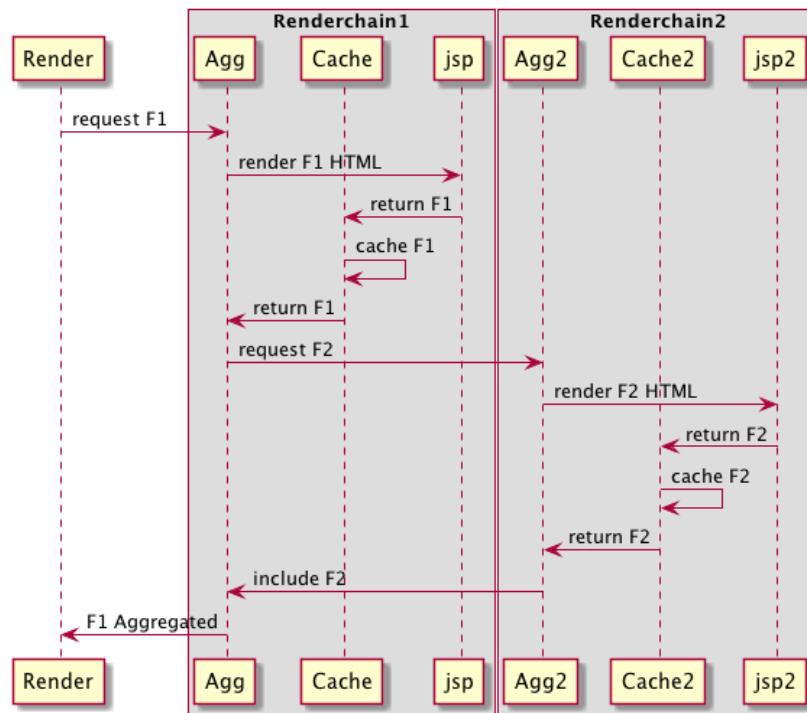


Figure 4 New implementation flow (using two render filter)

3 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.

4 INVALIDATION OR EXPIRATION?

Digital Experience Manager is using both modes for its caches, by default a fragment will have its expiration set to 4 hours.

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.

5 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 a 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.

6 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
  flushOnPathMatchingRegex="\\\\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 (This should be put in all your regexp encapsulating an unknown path).

7 FRAGMENT KEY GENERATION

Originally called cache keys, they are more fragment keys now. Because they are used by the Aggregate filter to identify fragments.

7.1 TYPE OF KEYS

- **Fragment key:** This is the identity of the fragment without the context.
- **Fragment final key:** The complete identity of a fragment, the final key is the result of (Fragment key + the current context of rendering). For example the same fragment key can result in multiple different final keys depending on the context of the render request. (users logged or not, parameters, etc.)

7.2 CACHE KEY PART GENERATOR

These classes are used to add new entries to fragment keys, and fragment final keys. In order to do that you can implement **CacheKeyPartGenerator**.

It contains two methods:

- **getValue():** used to generate the key only when the parent fragment is not in cache. Parent fragment html contains the sub fragment key, we will try to find the fragment using this key before calling the `getValue()` to rebuild it.

To summarize:

- build the fragment key
- only when parent fragment is not in cache
- support heavy operations like reading nodes from JCR
- **replacePlaceholders():** used to build the final key. When the fragment key is retrieved from parent fragment or have already been constructed. This method is called just after to identify the fragment based on the current context of execution.

This method is used to replace the previous value in the key, by something that could potentially differentiate the fragment based on the context, even if the initial fragment key is the same.

To summarize:

- Build the fragment final key
- Always called in all the case
- Do not support heavy operations, avoid JCR read here.

Example of best practice for CacheKeyPartGenerator implementation:

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

    @Override
    public String getValue(Resource resource, RenderContext renderContext,
        Properties properties) {
        // read the node to detect if the resource need to be contextual
        // will be call only one time
        if (resource.getNode().isNodeType("jnt:contextualizedNode")) {
            return "contextual";
        }
        return "notContextual";
    }

    @Override
    public String replacePlaceholders(RenderContext renderContext, String keyPart)
    {
        // apply the contextual changes in the final key to differentiate the
        fragments
        if("contextual".equals(keyPart)) {
            return renderContext.isLoggedIn() ? "logged" : "notLogged";
        }
        return keyPart;
    }
}
```

In the previous example the node of type "jnt:contextualizedNode" will have 2 possible different final key depending on the context (user logged or not)

So the best practice for the **CacheKeyPartGenerator** implementation is:

- avoid JCR read in replacePlaceholders (because call every time)
- use this two methods to decouple the logic of your cache key part generators (what is contextual ? what is not contextual ? do you need to read nodes ?)

Example of implementations:

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="${org.jahia.aclCacheKeyPartGenerator.implementation: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.AjaxCacheKeyPartGenerator"/>
      <ref bean="areaResourceCacheKeyPartGenerator"/>
    </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%2Csiteadministrator:%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 is possible.

Keys are generated during the prepare and execute phase of the **AggregateFilter** and should give the same result on each call, otherwise DXM 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 is 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 returns the locale of the current resource as a String when the key is generated. And the “replacePlaceholders” is just returning the same value, because there is nothing else to do. We could use the replacePlaceholders() to resolve the language, but we prefer to use the getValue() here because the language is the only information to resolved and mainly because if a parent resource is in language “EN” the child resource will also be in “EN”. The “replacePlaceholders” is called in all the case, so here we use the “getValue” because this one is only called when the key is built for the given resource, this way we reduce the code executed when fragments are in cache.

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();
    }

    public String getPath(String key) {
        return StringUtils.replace(key, MAIN_RESOURCE_KEY, "");
    }

    @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, we use the token “_mr_” returned by the `getValue()` and replaced by the `replacePlaceholders()`. A same node can be displayed in different context, with different main resource, that’s why this operation is done in the `replacePlaceholders()`. We also read the properties in the `getValue()` to know if the current fragment have the properties `cache.mainresource`.

7.3 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.

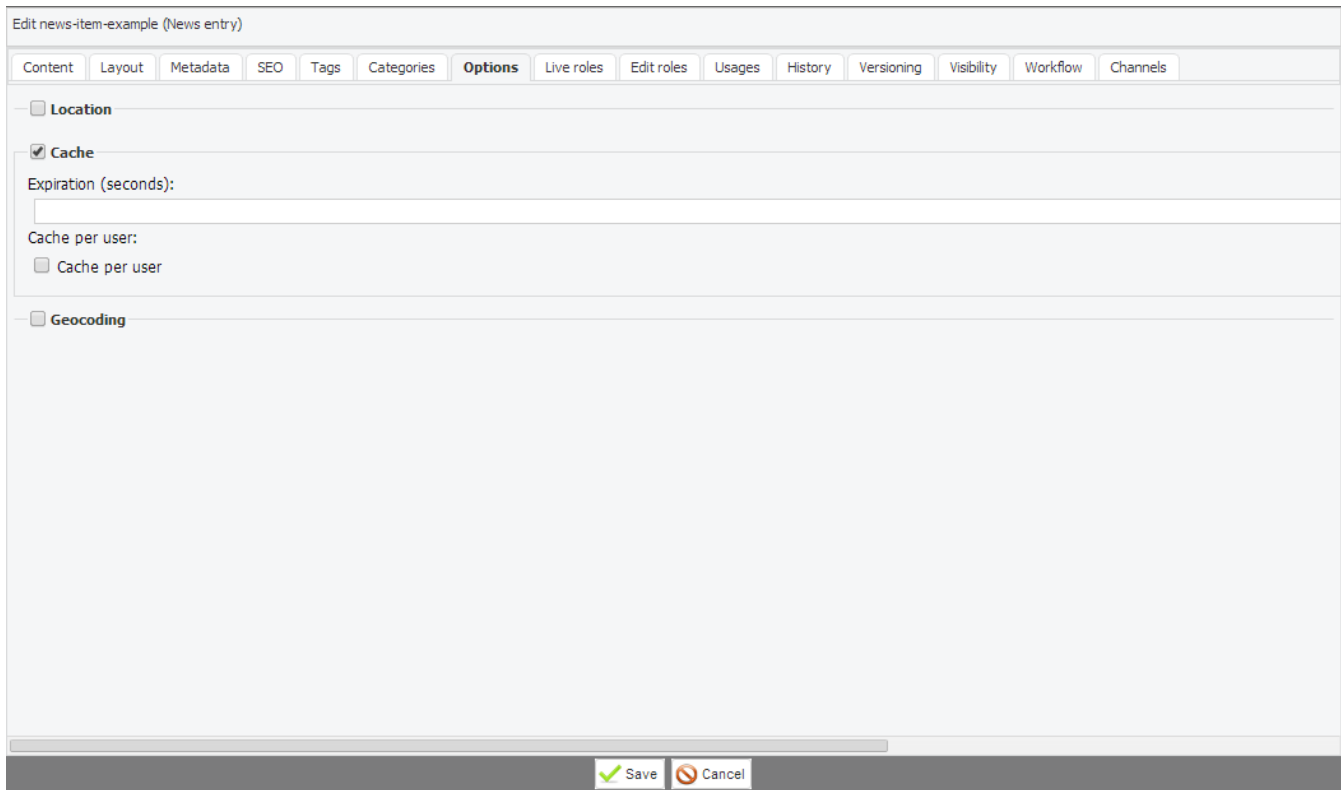
Groups and Users ACLs are stored in local cache that are flushed when an ACL is updated on the platform.

Improvement on new implementation:

- ACL in the keys were taking too much space, ending with very big keys. This have been fixed in 7.2
- New global optimization configs for ACLs key part are available:
 - **org.jahia.aclCacheKeyPartGenerator.usePerUser** : replace the ACLs in the fragment key by the user key of the current user, very fast to execute, but memory consuming, because a lot of cache entries will be created depending on the number of users on the platform. **Default value: FALSE**
 - **org.jahia.aclCacheKeyPartGenerator.useGroupsSignature** : best optimization, activated by default since 7.2. Internally cache all the principals users/groups that have ACLs, and append them to the key if it's needed. Faster than 7.1 implementation and fragment keys shortened. **Default value: TRUE**
- View properties:
 - **cache.perUser**: same effect as **org.jahia.aclCacheKeyPartGenerator.usePerUser** but only for the current fragment.
 - **cache.useGroupSignature**: same effect as **org.jahia.aclCacheKeyPartGenerator.useGroupsSignature** but only for the current fragment.

7.4 CUSTOM ELEMENTS IN KEYS

The custom cache key part allows you putting some elements 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 **AggregateFilter** 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 returned as is in the **replacePlaceholders** phase.



7.5 NEW IMPLEMENTATION SPECIFICITIES

7.5.1 Flow specificities:

In the new implementation, rendering scripts (JSPs and other scripts) are executed independently and not on a single page context anymore. This optimization is part of the new DXM 7.2 cache strategy as it allows to decouple the execution of each single fragment on a page, hence avoiding to lock a whole tree of fragments that could be used by concurrent HTTP requests (e.g other users on the platform).

7.5.2 Variables passed implicitly

Variables passed implicitly between view when using `<template:module>` tags aren't considered best practices and DXM provides alternative solutions. This pattern has a chance to do not work like it used to when using legacy aggregate/cache implementation.

Code that might play poorly with the new implementation was already generating some issues before the HTML cache optimization. It won't work at all now.

Default view:

```
<template:module node="{currentNode}" view="detailView"/>
<%
  // This is a simplified version of what actually happens in real life, but think
  // of variables
  // initialized in Render Filters for instance.
  // Note that this kind of code actually generated issues before DX 7.2, but it
  // will work even less
  // now each fragment generation is decoupled
  request.getSession().setAttribute("myVar", request.getAttribute("myVar"));
%>
myVar : {myVar}
```

Detail view (detailView):

```
<%
  // This is a simplified version of what actually happens in real life, but think
  // of variables
  // initialized in Render Filters for instance.
  // Note that this kind of code actually generated issues before DX 7.2, but it
  // will work even less
  // now each fragment generation is decoupled
  request.setAttribute("myVar", "Value of myVar");
%>
```

That code won't work because `template:module` isn't generating the fragment on the fly, but will rather generate a placeholder to be replaced with the actual HTML fragment later. The whole default view will now be executed before the `detailView` is called.

If for some reason, you need to make the previous case work with the new implementation, it's still possible to disable the aggregation. Multiple ways are available to achieve that:

- In a Render Filter, set the **AggregateFilter.SKIP_AGGREGATION** request attribute to true
- In the properties file associated to a view, define **skip.aggregation=true**
- In a view, set the **skipAggregation** attribute to `template:module` to true

By doing that, `template:module` will generate the fragment on the fly and will return once the `detailView` is executed. keep in mind that doing this will merge both fragment in the cache entry (instead to have two cache entries, there will be only one containing both fragment already aggregated)

Or you can use the Interface `RenderContextTuner`.

7.5.3 RenderContextTuner

It's a new interface related to the new implementation, it allows serializing data into the fragment key to be able to restore it if this fragment needs to generate and the parent is in cache.

Scenario example:

- Parent fragment A is in cache
- Sub fragment B have been flushed
- When the page is requested by a visitor, the fragment A is served by the cache, but fragment B has to be generated. But there is an issue with this, fragment A was setting request attribute in the request for fragment B.

To be able to handle such case, a Cache key part generator can implement the interface **RenderContextTuner**. And here is how the previous scenario will be handled:

```
@Override
public Object prepareContextForContentGeneration(String value, Resource resource,
    RenderContext renderContext) {
    if (fragmentA) {
        // set the request attribute for the next render chain
        renderContext.getRequest().setAttribute(requestAttrFragmentB,
            deserialize(value));
        return value;
    }
    return null;
}

@Override
public void restoreContextAfterContentGeneration(String value, Resource resource,
    RenderContext renderContext, Object original) {
    if (fragmentA && original != null) {
        renderContext.getRequest().removeAttribute(requestAttrFragmentB);
    }
}
```

The above two methods are provided by the **RenderContextTuner** interface. the method “**prepareContextForContentGeneration**” is called before the render chain starts to generate fragment B, so the data is put in request and will be used by Fragment B JSP. The method “**restoreContextAfterContentGeneration**” is called just after the render chain finished to render fragment B, this way we can clean the request.

So, using `RenderContextTuner`, it's now possible to store data in the fragment key, and reinject this data to be able to render fragments correctly, when data need to transit between different level of fragments, this is considered as the best practice now.

The core is providing some usage of the `RenderContextTuner`, example of the `AreaResourceCacheKeyPartGenerator`:

```
@Override
public Object prepareContextForContentGeneration(String value, Resource resource,
    RenderContext renderContext) {
    if (!isDisabled() && StringUtils.isNotEmpty(value)) {
        // set the request attribute for the next render chain
        renderContext.getRequest().setAttribute(SAVED_AREA_PATH, value);
        return value;
    }
    return null;
}

@Override
public void restoreContextAfterContentGeneration(String value, Resource resource,
    RenderContext renderContext, Object original) {
    if (!isDisabled() && original != null) {
        renderContext.getRequest().removeAttribute(SAVED_AREA_PATH);
    }
}
```

This key part generator oversees storing the area path if the fragment is under an area, this is mainly used by content list that need to get the properties of the area. The content list is under the area, if the area is in cache but the content list not, the area data will be lost.

That's why the area path is serialized into the fragment key, this way we can re inject it before generating content list fragment and clean the injected data after.

7.5.4 JCR Node read before Cache Filter

The new implementation has been designed to use the render chain capabilities. Now Render filters that have priorities inferior to the Cache filter are executed even if the fragment is in cache. This is a big difference compared to the previous implementation.

Due to the HTML cache refactoring, some changes were made to the Render Filters with a priority lower than 16.

- **Legacy implementation**, filters with a priority lower than 16 that apply to a view (that does not belong to the main resource) which is cached are not executed.

- **New implementation**, filters with a priority lower than 16 that apply to a view which is cached are executed each time.

Custom Render Filters with a priority lower than 16 need to be analyzed since chances are they will get executed much more often than before.

It is not considered as a good practice to do backend calls during a render chain execution before the HTML CacheFilter is called. To discourage developers from doing so, a WARN message has been introduced in the logs when JCR read operations are performed in a Render Filter with a priority lower than 16.

Since Render Filters with a priority lower than 16 are now called each time a fragment is displayed, read operations in such case will become costlier. **Developers are encouraged to move the JCR Read and Write operations to Filters executed after the caching takes place.**

7.5.5 Render filters priorities:

The priority of Render Filters is now a Long variable, instead of an Integer. This allows to be more accurate in render filter placement.

Instead of a monolithic **AggregateCacheFilter** , two new filters have been created: **AggregateFilter** and **CacheFilter**. A new filter was also created to measure the rendering time of each individual fragment. For legacy purposes, **AggregateCacheFilter** still exists and can be activated instead of the new implementation.

16.0	AggregateFilter	New implementation: Aggregate filter, aggregates contents by resolving sub fragments
16.0	AggregateCacheFilter	Legacy implementation
16.1	MetricsLoggingFilter	Calls the logging service to log the display of a resource. Also initializes profiling information. Disabled by default in 7.2.

16.5	CacheFilter	New implementation: Cache filter, provides the html result (either from the cache or by generating it) of the fragment and caches it if necessary
16.8	NodeAttributesFilter	NodeAttributesFilter Sets request parameters related to the JCR node; separated from the BaseAttributesFilter to avoid reading the node from JCR before the cache filter