Jahia 5 Performance Tuning Guide

5.0SP3

Jahia

9 route des Jeunes, CH-1227, Carouge Switzerland

http://www.jahia.com > The company web site
http://www.jahia.net > The community web site

VERSION

This table records the versions of this document and their last updates

Version	Author	Date	Modifications
1.0	Serge Huber	2007-06-26	Initial document
2.0	Serge Huber	2007-09-14	New sections on selecting configurations for deployment
5.0SP3	Serge Huber	2007-10-08	Preparing for simultaneous release with Jahia 5.0SP3

SUMMARY

Page

Version 1

Summary 2

Introduction 1
Users types 1
Effects on Jahia system load 1

Jahia Caches 3
Introduction 3
Choosing output cache deployment 3
HTML cache 5
HTML cache – expiration only mode 7
Container HTML cache 7
ESI (Edge-Side Includes), deactivated in 5.0SP3, being rewritten for 5.1 8
ESI – expiration mode 9
OSCache 10
Rule of thumb for choosing between output cache deployments : 11
Output cache compatibility matrix 11
Configuring output cache 12
Full-page HTML cache parameters 12
Configuring ESI 13
Time-based invalidation mode 13

Clustering deployment 14
Clustering overhead 14
Jahia "Browsing & Editing" nodes 15
Processing server 15
Indexing server 16
How to size the cluster deployment 16
Rules of thumb to size the cluster deployment 17
Cluster configuration 17

Testing performance 18
Configuration 18
On Jahia 18
On Tomcat 19
On the database 19
Before each test 19
Restart each computer running Jahia 19
Warm up pass 20
Tools 20
Extracting test input data 20

Case studies 22
Introduction 22
Small Jahia site 22
Medium Jahia site 23

Large Jahia site 25
Scaling out the configuration 27
Fail-over configurations 28
Template fine-tuning 30
During development 30
In production 32
Jahia Tuning 34
During development 34
In production 34
Portlets fine-tuning 47
During development 47
In production 47
Database Tuning 48
During development 48
In production 48
Other useful tools 50

INTRODUCTION

This document presents various deployment and fine tuning tips, in order to achieve the best possible performance when deploying high traffic and large data Jahia installations.

In order to better understand how to fine-tune Jahia for the best possible performance, we will first give a quick overview of the architecture of the application. Jahia is composed of many layers, and a proper understanding of each of them will help you build and setup the best system for deployment.

USERS TYPES

Users types define different types of load on Jahia content management installations, and we will introduce them here so that we can better explain how to deploy and configure Jahia according to the expected amount of user types.

- Anonymous users : these users are people simply navigating through the site, without any type of personalization associated with them.

- Logged-in, authentified users : recognized users of the Jahia system, which will see possibly different content than anonymous users, and will have different options for personalizing their experience on the site. This can include user profile information that is being displayed or used to personalize the page display.

- Content editors : these are a sub-class of the authentified users that are allowed to enter EDIT mode and modify content on the site. They may also control access to content if they have the permissions to do so.

- Content validators : these users are similar to content editors, except that their behavior is a little different. They mostly review content and publish it once it has been approved.

- Administrators : in charge of administrating the site, or maintaining the complete Jahia installation. They can create new sites, setup replication of content, etc.

Effects on Jahia system load

- Anonymous users are in effect the users that affect Jahia's load the least, as they are only viewing content that is identical to all. They

share the same username called “guest”, which is part of the “guest” group.

- Authentified users may introduce load because the generated HTML might be different amongst users, meaning more memory will be needed to handle them. Page generation load will be more important as they do not see them same content. This is especially true for the full-page HTML output cache system, which cases a copy of the whole page for each user. The container HTML cache and ESI sub systems are more efficient for handling this type of user. Such users do not have major influences over load on the processing server or the indexation server.

- Content editors generate and modify content. They introduce an edition load because they can enter EDIT mode (and therefore the output cache will be different for them), but they also introduce an additional load as database writes will happen when they modify or add new content. They also have an effect on the indexing sub-system when uploading files, as well as load on the processing server when doing operations such as copy & paste or import / export.

- Content validators mostly affect the processing and indexing servers as they publish the modifications to the LIVE mode.

- Administrators introduce the most varied load, depending on the operations they are performing. Site creation and deletion are also load expensive operations, as well as site replication that are similar to import/export operations.

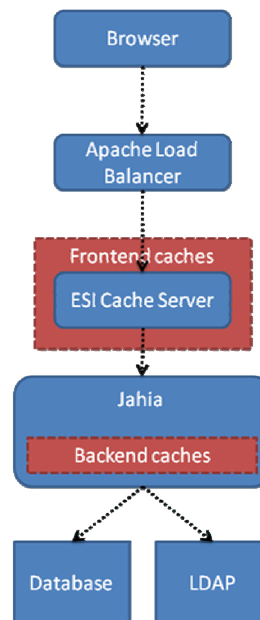
JAHIA CACHES

INTRODUCTION

Jahia uses caches to improve the performance of its input/output operations (such as reading from /writing to the database or generating the resulting HTML page). We can separate these caches into two categories:

- Backend caches : these cache layers mostly concern optimizing Jahia's internal operations when communicating with the database, or internal work such as creating objects, accessing LDAP servers, etc.

- Frontend caches, also known as output caches or HTML caches : these caches are located in the request chain as close as possible to the HTML browser in order to serve HTML mostly from the caches instead of generating load on the Jahia backend. This leaves the backend free for other operations.



The above illustration explains where the two different caches are located in the request flow.

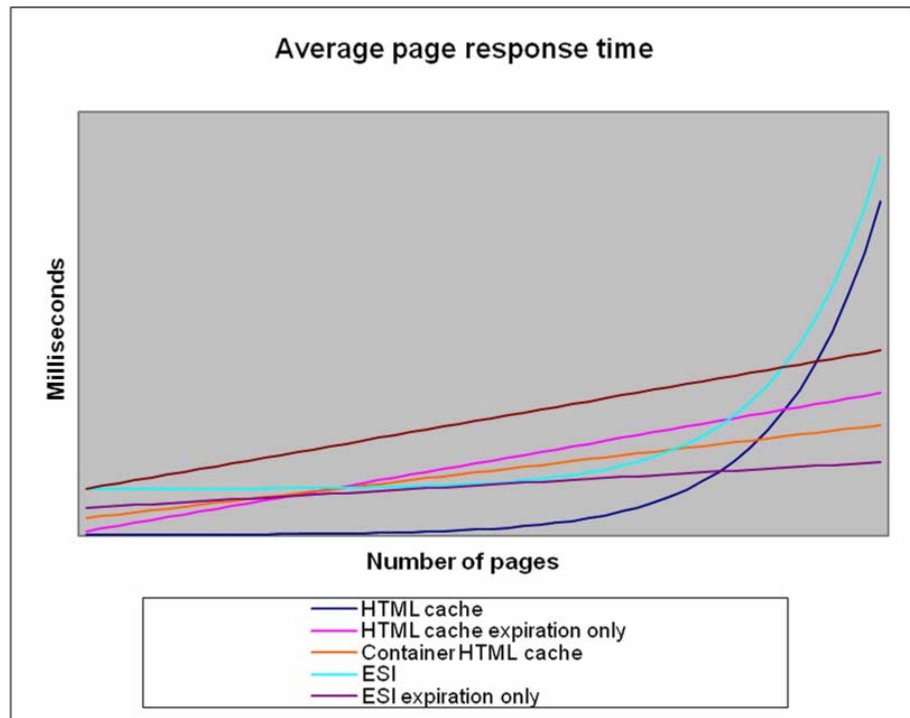
CHOOSING OUTPUT CACHE DEPLOYMENT

Jahia is quite powerful in terms of performance, and offers multiple possibilities in order to cache the output of a generated page, so that subsequent requests for the page will be very fast. In best case scenarios, page response time can be below measurable milliseconds, so it is clearly an interesting topic to think about when looking at deployment possibilities.

Jahia also uses other caches, notably to optimize the performance of the back-end services when accessing the database, but these will not be presented in this section. Check out the database and Jahia fine-tuning tips later in this document for more information.

Jahia basically offers 4 different output cache solutions:

- HTML cache
- Container HTML cache (starting from Jahia 5.1)
- ESI cache
- OSCache



Disclaimer : this graph is for informational purposes only, and doesn't represent real performance testing data

As we can see in the above graph, the theoretical evolution of response times versus the number of pages will help you select the best cache implementation for your deployment. For small sites, the HTML cache is the best, while for large sites, the ESI server in expiration-only configuration is clearly the most performing cache implementation.

We will now detail each cache implementation, as to give an idea what their advantages / disadvantages are.

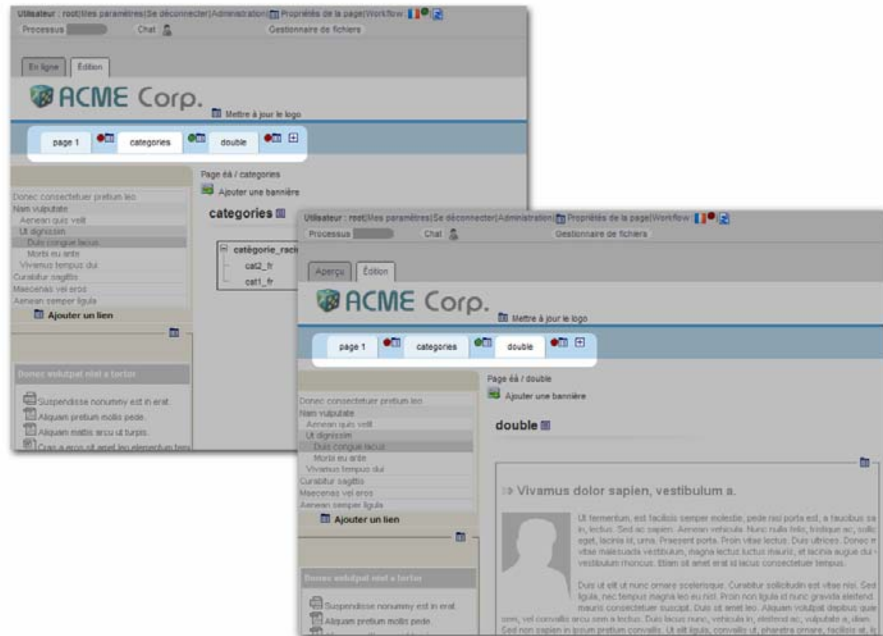
HTML cache

The basic integrated output cache in Jahia is called the "HTML cache", and caches the whole HTML rendered for a specific page. Since Jahia pages may be different by type of users (because of access controls at the container level), by type of navigation modes (live/edit/preview) and by language, we must actually store multiple HTML versions of the same page in the cache.

As an example, let's say you have a site that contains 1000 pages, that has 500 authentified users (not using the same guest/anonymous profile) that are all able to edit content and that contains 5 languages, the total combination of entries that will be stored in the HTML cache could be as high as 500 x 5 x 4 (navigation modes) x 1000 = 10'000'000. If we average the size of an HTML page at 80kBytes, the total weight of the HTML cache would be 768GB (!). Of course this is assuming that each user has visited all the pages of the site which is rarely the case, and that each user has full editing rights on the whole site, which is even less likely. Nevertheless, this cache can quickly grow in size, assuming that entries are not flushed too often because of content updates.

But it is evident that as this cache is integrated into the same JVM as the rest of the Jahia server, the size of this cache can quickly become problematic. Properly adjusting the size this cache is also difficult because it largely depends on the type of expected traffic.

Moreover, as Jahia allows for content to be re-used on multiple pages, when we update this content, we will need to invalidate the HTML cache for all the target pages. In order to keep track of which content is stored on which page, Jahia uses an internal structure called "JahiaLinks" which stores information as to which content is stored on which page. In order to better illustrate this, let's have a look at the following schema :



As we can see, both pages "categories" and "double" display the same navigation container list (as in Jahia even navigation are built using a container list). As the choice of displaying this container list is done in the templates, Jahia also tracks this usage in the database, in the "JahiaLinks" (aka as cross-reference table) table, because it needs to be able to access reference information without dispatching to the template. If more pages also display the same container list, the number of cross-references will grow. This structure is built on demand, so each time a page displays a new content object, the structure in the database must be updated, and therefore database write operations can occur when we are simply displaying a page, which in turn will slow down the rendering of the page.

Advantages :

- Extremely fast when in cache, can't be faster as we are directly serving HTML from memory
- Integrated in the Jahia server, no need to setup a distinct cache proxy server
- Invalidation is handled by Jahia's content management system, not need to take care of it at the template developer or user level, not is there any waiting time for new content to appear.
- Immediately invalidated when a change occurs

Disadvantages :

- Can consume a lot of memory which could lead, if the cache is not limited to out of memory errors.

- "JahiaLinks" cross-reference table can grow really large, and exponentially on large sites which can lead to severe performance issues on large sites.

HTML cache – expiration only mode

The expiration-only HTML cache is a configuration variation of the HTML output cache. Basically when this mode is activated, it means that the "JahiaLinks" cross-reference system will no longer be used to keep track of which pages need invalidating when a content object is modified, and invalidations will only be based on a time-based policy. This configuration is valid only for the LIVE mode navigation. EDIT, PREVIEW and COMPARE modes, as they need to always be up-to-date, will never be cached. So if you have lots of content editors, this option will put more load on the back-end system that will have to regenerate the page each time.

Advantages :

- Scales better than HTML cache on large sites

- Consumes less memory (since we don't cache EDIT, PREVIEW and COMPARE navigation modes)

- Much faster page generation on large sites since we don't have to maintain cross-reference data

Disadvantages :

- EDIT mode becomes slower

- LIVE content is not immediately available after a page has been published but only when the expiration time has been reached.

Container HTML cache (Jahia 5.1 and later only)

Starting from Jahia 5.1, we will introduce a new HTML cache, that, instead of caching the entire HTML page, automatically caches the output of a single Jahia container.

This cache is targeted at Jahia deployments that will be doing a lot of content re-use, which is often the case for top-level container navigation menus that usually always display the top-level pages on each page. If an editor were to change the name of a top-level page when using the full-page HTML output cache, Jahia would have to invalidate the whole cache, as all the pages display the top-level menu. This is the worst case scenario for the full-page HTML cache. Another consequence of using the full-page HTML cache is

that the cross-references for this navigation content object must be maintained on all the pages, and as the site grows, so does the database table containing the references. This directly impacts the performance of the page generation as the references must be updated for each page, and as the number of pages grows, the loading of the references from the database becomes slower.

The container HTML cache is based on the assumption that we will no longer cache the full-page, but only container "blocks" of HTML. If these blocks are re-used on multiple pages, the same cached entry will be used. This is very similar to what the ESI cache server does in a more general way with fragments. Contrary to the ESI technology, the template developer cannot himself define what the fragment would be on each page.

If we take the same example as in the full-page HTML output cache example, when we update a container that's displayed on multiple pages, with the container HTML cache we will only flush the container entry and no other cache entry will be flushed, not even the other containers in the same container list.

Another advantage of the container HTML cache implementation is that it is able to share cache entries between users, by calculating the group of users that have the same permissions on content. This is also known in Jahia terminology as "ACL groups".

Advantages :

- Good performer on sites which re-use content on multiple pages, for example in the case of a top-level navigation menu displayed on each page
- Scales up well to large content sites

Disadvantages :

- Slightly slower than best-case full-page HTML output cache, as Jahia needs to aggregate the container fragments in order to recompose the full page.

ESI (Edge-Side Includes), deactivated in 5.0SP3, being rewritten for 5.1

ESI, also known as Edge-Side Includes (http://www.esi.org), is a caching technology that was developed to solve the problem of caching output in highly dynamic web pages. In such pages, we often only want to cache HTML fragments of the page, instead of the complete rendered HTML, as with each request the displayed fragments may change. This is also based on the assumption that fragments will be re-used, amongst different users or pages, and therefore will provide a performance benefit by generating a specific fragment only once. It also introduces the possibility of having lighter

cache proxy servers serving the "edge" content without requiring all the complex back-end architecture of a full-fledged Jahia server.

From a technical point of view, ESI uses a structure similar to the "JahiaLinks" (or cross-references) to keep track of which fragments contain which container lists, in order to send the ESI caching server invalidation messages when a container is updated. The tracking of these relationships is done through an AOP system implemented using the Aspectwerkz implementation. This implementation introduces some limitations on deployment on specific application servers.

As this last aspect was causing performance and deployment problems, it is currently de-activated in Jahia 5.0SP3 and being rewritten for Jahia 5.1, using the new container HTML cache feature.

Advantages :

- Standalone caching server, doesn't use any memory from the Jahia server

- In future versions, could be clustered

- Encourages fragment re-use between users, groups, pages

- Relatively straight-forward to integrate into templates

Disadvantages :

- Requires support for Aspectwerkz in web application server, as it relies on this library to perform it's invalidation tracking. Another side effect is a partial slow-down on the JDK 1.4 because it doesn't have built-in support for code instrumentation, which is a requirement of aspect oriented systems such as Aspectwerkz.

- Traffic between caching servers and Jahia servers can introduce in some cases high loads on the Jahia servers (if a lot of fragment requests happen simultaneously)

- More complicated to deploy than other Jahia output cache solutions

- Scalability issues in current implementation when the number of Jahia servers increases due to replication of the invalidation tracking information (which keeps track of which fragment contains which content objects).

ESI – expiration mode

In order to solve some of the problems in the current implementation of the tracking structure, Jahia 5.0 Service Pack 3 introduces a new mode for the ESI

caching server that caches fragments only in live mode, and invalidates them using time-based rules instead of invalidation messages coming from the Jahia server. This new mode is similar to the HTML expiration-only mode that was introduced in Jahia 5.0 Service Pack 2.

Advantages :

- Same as ESI
- No need for an invalidation tracking structure that is stored on the Jahia back-end server and eats up precious memory
- No scalability problems

Disadvantages :

- Time-based expiration. Published content is not immediately visible on caching server
- EDIT mode is no longer cached, meaning that surfing in this mode will be a little slower

The future of the ESI implementation is to provide an implementation similar to the container HTML cache system, but that will be able to run in standalone mode on a dedicated caching server.

OSCache

The last solution for output HTML caching is the introduction of a legacy caching library. It is targeted at integrators that will develop and integrate into their own Jahia templates their custom logic to personalize page generation. This could also include aggregating outside content from RSS feeds, portlets, or other sources. The goal is to cache the result so that subsequent requests will be much faster. For this we recommend using the OSCache JSP tags that offer custom control of caching. Invalidation must be handled either through time-based expiration or through custom logic implemented by the integrator. An example of a custom invalidation implementation could be done using Jahia event listeners, so that invalidation could be performed even on Jahia content modifications.

For more information on integrating OSCache, please check the following URL : http://www.opensymphony.com/oscache/wiki/JSP%20Tags.html

It should also be noted that it is perfectly possible to mix OSCache integration with another output cache system, such as the full-page HTML cache or ESI, but one must be careful about the invalidation then, as it will be more complicated to handle because of the intermix between the two caching systems.

Advantages :

- Flexible solution for custom output caching
- Easy to use if a time-based expiration solution is acceptable

Disadvantages :

- Not integrated with Jahia content modifications, must be done manually by the integrator if needed
- Not a remote caching solution like ESI. The cache will consume the memory of a Jahia server which could lead to out of memory issues if the expiration policy is not properly setup.

Rule of thumb for choosing between output cache deployments :

Site type	Preferred output cache
Small "static" sites	HTML cache
Large "static" sites	HTML cache with time-based expiration mode
Small dynamic sites and personalization	Container HTML cache
Large dynamic sites with high load and personalization	ESI
Custom page rendering or portlets in pages	OSCache

Output cache compatibility matrix

The following table explains the output caches that may be activated simultaneously.

	HTML	HTML time expiration	Container HTML (5.1)	ESI (5.1)	ESI time expiration	OSCache
HTML	x					x*
HTML time expiration		x				x*
Container HTML (5.1)			x	x***	x	x
ESI (5.1)			x***	x		x**
ESI time expiration			x		x	x**
OSCache	x*	x*	x	x**	x**	x

*=cache will need to be deactivated on the page that has the fragments

**=not really interesting for performance, use default ESI functionality instead

***=not optional, required

Configuring output cache

The output cache is mostly configured in the tomcat/webapps/jahia/WEB-INF/etc/config/jahia.properties file, except for the case of the ESI server where configuration must be done on it's server (see section later in this document for ESI setup).

Full-page HTML cache parameters

```
# The output (HTML) cache may also be controlled in more detail with the
# following parameters.
outputCacheActivated                                         =   false
# the following value is in milliseconds, set to -1 for no time expiration
outputCacheDefaultExpirationDelay                            =   -1
# The following setting is designed to be used for large sites (10'000
# pages and more), and will switch the output cache to a expiration-only
# mode. This means that all pages in live mode will not be invalidated
# immediately when content is published, but only after the expiration
# of the cache entry. This also deactivates the output cache in EDIT
# mode, which might have a performance impact. Also, this deactivates
# the generation of the JahiaLink HTML references building, which is
```

```
# a performance problem when sites reach large sizes. So if your
# site is getting large, it is recommended that you switch this
# variable to true and that you set a reasonable value for the
# outputCacheDefaultExpirationDelay.
outputCacheExpirationOnly                              =   false
```

CONFIGURING ESI

ESI can be configured in two ways :

- Using time-based expiration of fragments, in which case new content will only be available after a certain time for validated elements, but all the other navigation modes are non-cached. This mode is by far the one that can handle the biggest load in LIVE mode, because the server doesn't need to do anything to track content invalidation. Unfortunately there is a slowdown in EDIT/PREVIEW and COMPARE modes as these are not cached.

- (Jahia 5.1, deactivated in 5.0SP3) Immediate invalidation upon content modification. This mode uses structures on the Jahia server so it can eat up a little more memory as well as a little more CPU, but it ensures that all modes are cached, therefore offering better overall performance.

Time-based invalidation mode

In order to launch ESI using the new time-based invalidation mode, you must run ESI without the Aspectwerkz framework so be sure to start your Jahia server using jahia.sh or jahia.bat and no longer the jahia_Esi.sh or jahia_Esi.bat (now deprecated)

In the ESI server configuration file (tomcat/webapps/ROOT/WEB-INF/data.xml) is configured to not cache EDIT/PREVIEW/COMPARE modes (as defined in the pass through rules)

The expiration of content will only happen once the elapsed time has been reached. The default value is set to 3600s (1 hour) this could be easily changed in jahia.properties or jahia.skeleton

In order to migrate an existing installation, after installing all the new versions of ESI and Jahia, in the jahia.properties the following variable must be set to this value :

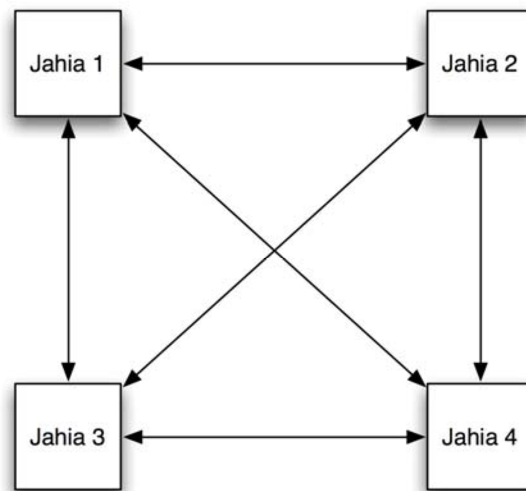
```
esiCacheDefaultExpirationDelay = 3600
```

CLUSTERING DEPLOYMENT

Deploying Jahia in a cluster is a very powerful way of distributing CPU and memory load to handle larger traffic sites. We will explain in this section the different types of possible deployments, as well as how to choose whether to use them or not.

CLUSTERING OVERHEAD

Although clustering is a good solution to scale Jahia, it is not completely transparent in terms of performance impact. As Jahia uses internal caches to speed up its operations, cache coherency must be maintained throughout all the nodes of the cluster so that, when a node invalidates a cache entry (for example because it has modified it), the other nodes are informed, and in turn must invalidate their own cache entry. This communication overhead is only present when Jahia is configured in cluster. If we have a look at it graphically, and can be illustrated as follow :



As we can see in the above illustration, each node communicates with all the others, in a truly peer-to-peer fashion. The nice thing about this is that nodes are completely independent, and if one of them fails the system can still work. By default the cluster configuration uses UDP multicast packets, which reduces the need to actually connect to all the nodes. This requires a solid configuration on the routing equipment, which is quite often a bit problematic. In order to avoid this problem, a TCP configuration mode is also available, as well as even other types of transport. Jahia uses for this the

JGroups clustering library. More information about the capabilities of this library may be found here : http://www.jgroups.org .

As the number of nodes in a cluster deployment grows, so does the traffic between the machines, so it is important to remember that there is an overhead due to the communication between all the machines (in the form of invalidation messages). It is therefore crucial that the networking hardware between the machines does not interfere with the traffic, and that everything is setup properly so that the performance impact stays minimal. Jahia already optimizes the traffic of messages between the nodes in order to regroup them and only send one packet at the end of each request. The receiving nodes then must unpack the list of invalidations and process them. Basically this traffic should stay in the 10% CPU overhead range, and this is mostly related to the number of nodes as well as the performance of the CPU and the JVM on each node, so if a large cluster is planned, the machines should be properly dimensioned in terms of processing power.

JAHIA "BROWSING & EDITING" NODES

Jahia "browsing & editing" nodes are cluster nodes that can be used to either browse Jahia content or edit it. This is the most common usage of Jahia nodes, and therefore it is interesting to have multiple instances of these in order to distribute the load.

It is also possible, through Apache Web server rewrite rules, to separate edit and browsing loads by, for example, redirecting all edit mode navigation and editing to specific machines. Support for this type of separation will be improved in Jahia 5.0 Service Pack 3 and further versions. It should be noted that the clustering setup requires the configuration of "sticky" sessions, as Jahia does not currently support session distribution, and each user session must be always directed to the same node.

Browsing is usually not a heavy operation, but can become more important if personalization is used or if portlets are deployed on a Jahia site. Also, as Jahia browsing nodes can also serve as "cache loaders" for ESI cache servers, it might be good to have multiple Jahia browsing servers in order to sufficiently serve the requests coming from the cache server.

Edition-specific nodes are also quite interesting because they clearly avoid loading the browsing nodes with edition operations, which makes sure they stay fully available for the larger number of users only browsing the site. Usually the number of editors is significantly lower than the number of "simple" users, so this type of configuration is quite interesting. It is of course also recommended to limit the maximum number of editors per node, and therefore to add editing nodes if the number of content modifiers grows.

PROCESSING SERVER

In Jahia, long-running tasks such as workflow validation operations, copy & pasting, content import and indexing are executed as background tasks. This way while these long operations are executed, the server is still able to process content browsing and editing requests. Despite this, these background tasks may eat up significant memory and CPU resources, so this is rarely ideal for medium to large Jahia installations.

In order to off-load Jahia browsing and editing nodes of long-running operations such as workflow operations, copy & pasting, import of content, such tasks could be directed to a Jahia server configured as a stand-alone processing server. Usually this server will not be made available to serve Jahia pages, although it is capable of doing so. In most cluster installations the processing server is a node dedicated to this task, which is a good solution, both in terms of user-experience and system load. It should also be noted that it is mandatory to have one and only one processing node.

INDEXING SERVER

File and content indexation can be a very expensive operation both in terms of CPU usage and memory load. For example, in order to index PDF files, the server must actually execute the script instructions that are contained within the file in order to extract the text content. This requires the execution of a full Postscript-like language, including its memory management and instruction processing, just to extract text content. In the case of complex Microsoft Office files, extracting the text can require a lot of memory and CPU. Indexing large files also requires memory that cannot be freed until the actual extraction has been completed.

Indexation is usually considered to be an operation that is not needed in real-time, and therefore can run as a background task. In order to reduce the load on the Jahia servers doing the actual content editing or processing, it is strongly recommended to install a separate content indexing server, especially if the content will contain a lot of typical office files.

It is also possible to install multiple indexing nodes, so that there is no single point of failure for this type of operation. In this case the filesystem between the nodes must be shared, as indexes are stored on a shared filesystem.

HOW TO SIZE THE CLUSTER DEPLOYMENT

Now that we have introduced all the different parts of a typical Jahia installation, it is quite common to ask what is really needed for a specific customer site deployment. As much as this question is quite easy to ask, unfortunately it is not as easy to answer, as it depends on lots of different parameters, and some of them might be hard to guess in advance.

Basically, the main criteria for selecting the size of the cluster installation will be the user load. By this we mean to say the different types of users that will

use the site, the number of these types of users and the level of personalization as well as dynamic content (such as portlets). All this will help determine the projected CPU and memory load on the servers, which in turn will give a clearer idea of what type of install to perform.

Other important criteria are the content size and frequency of modification, as well as the number of files that will be managed by the Jahia installation. More importantly, the type of file is quite significant, especially in the case of PDF files. It is quite common for large sites to have needs to store large number of PDF files, and therefore this will introduce specific requirements on the deployment planning, especially the indexing configuration.

Rules of thumb to size the cluster deployment

Number of "editing nodes" : this is determined by the projected expected number of content editors that will be working on the site simultaneously. Anonymous and logged users can be important if their numbers are quite significant, but content editors usually generate a higher CPU and memory load than non content editors. Also if the browsing nodes are separated from editing nodes, the browsing users need to be quantified in order to know how many browsing nodes will be necessary.

Processing server : usually recommended as soon as the number of content editors reaches 5-10 simultaneous users of if you are using lots of copy/paste, cron or validation operations on large sites. Although this figure is not precise, it is in general a good idea to separate long-running operations to a separate server.

Indexing server : this requirement is mostly determined by the number of files that will be introduced into the system, and the frequency of these introductions.

Cluster configuration

Please read the « Howto cluster » guide, that is referenced from the Jahia readme to find the procedures that detail how to configure Jahia in cluster.

TESTING PERFORMANCE

Along with the initial planning of deployment, it is also a very good time to plan for performance testing. In order to avoid last minute surprises, especially when going into production, it is highly recommended to have executed minimal performance testing on the Jahia servers. Actually this is not even specific to Jahia, and is in general a good idea for any web system deployment.

Testing should be developed to simulate each user type, in a manner as close as possible to the real thing, but automated so that it is possible to stress-test too. Stress testing will possibly illustrate flaws in the installations that will hopefully never occur, but give a good idea of the limitations of the system, which in turn will help planning for scaling the system even further. For example, stress testing might show that the database back-end is getting overloaded, and will help plan for looking at clustering solutions on the database side.

CONFIGURATION

In order to setup Jahia for performance testing, a few configuration parameters must be modified. Basically first you must determine your user load, which will be used as the basis for configuration of all the rest of the Jahia installation. In this example we will assume that you will be used 200 simultaneous users.

On Jahia

In the jahia.properties file, located in the tomcat/webapps/jahia/WEB-INF/etc/config directory, you must adjust the following parameters to the following values :

```
maxParallelProcessings = 200
```

The above setting makes sure that Jahia allows at least 200 simultaneous worker threads to serve the content (excluding pages served from the HTML cache). Usually in production, in order to save memory it would be best to put this as close to the "usual" load of simultaneous connections, but not much higher, so that when load starts peaking, the server correctly limits connections.

```
pageGenerationWaitTime = 1
```

The above setting controls the wait time (in milliseconds) that threads above the maxParallelProcessings value will have to wait. In our tests we set this value to 1 ms, in order to immediately reject the extra requests. Normally we should never have this case if the parallel thread limit has been properly setup, and by setting this delay very low, if an extra thread request is received, we will receive "503 - server overloaded" error messages, making us aware that we should augment the maxParallelProcessings value.

On Tomcat

In the tomcat/conf/server.xml file, modify the "maxThreads" parameter on the "connector" tag to at least 400 threads, as Tomcat is responsible to serve not only the HTML, but also all associated resources (images, CSS, Javascript). It should look like this :

```
    <Connector port="8080" maxHttpHeaderSize="8192"
               maxThreads="400" minSpareThreads="25" maxSpareThreads="75"
               enableLookups="false" redirectPort="8443" acceptCount="100"
               connectionTimeout="20000" disableUploadTimeout="true"
               emptySessionPath="true"/>
```

In the tomcat/conf/Catalina/localhost/jahia.xml file, the database connection pool settings must be raised to allow at least 200 simultaneous connections. This is done by changing the "maxActive" parameter on the "Resource" tag, as shown below :

```
<Resource name="jdbc/jetspeed" auth="Container"
            factory="org.apache.commons.dbcp.BasicDataSourceFactory"
            type="javax.sql.DataSource" username="jahia" password="jahia"
            driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost/jahia_502?useUnicode=true&amp;characterEncoding=U
TF-8" defaultAutoCommit="true"
            maxActive="200" maxIdle="30" maxWait="10000"/>
```

On the database

The database configuration must allow for at least the same number of connections that were allocated to the database pool configuration in the "On Tomcat" section. Please refer to your database documentation to figure out how to change the maximum number of simultaneous connections allowed.

BEFORE EACH TEST

Restart each computer running Jahia

As long-running and stressed instances of Jahia can consume a lot of resources, especially TCP/IP connections, it is a good idea to restart the computer (not only the Jahia or web application server), before each performance test. This ensures reproduceability of test results too.

Warm up pass

Make sure that before you run your performance tests, you do once full warm up pass, as new releases of Jahia introduce a lot of precalculation that must be done only once. This way you can ensure that your performance will also be as close as possible to real-world performance.

TOOLS

Performance testing can be performed with many various tools, and we will present here a few of the most-used ones :

- Apache JMeter (http://jakarta.apache.org/jmeter/) : A very well known open-source tool written in Java to test performance of web applications. Apache JMeter may be used to test performance both on static and dynamic resources (files, Servlets, Perl scripts, Java Objects, Data Bases and Queries, FTP Servers and more). It can be used to simulate a heavy load on a server, network or object to test its strength or to analyze overall performance under different load types. You can use it to make a graphical analysis of performance or to test your server/script/object behavior under heavy concurrent load. JMeter requires users to build the testing scripts pretty much by hand, which can be a little tedious at first, but the interface for building the scripts requires almost no programming.

- OpenSTA (http://www.opensta.org/) : OpenSTA is a distributed software testing architecture designed around CORBA. The current toolset has the capability of performing scripted HTTP and HTTPS heavy load tests with performance measurements from Win32 platforms. However, the architectural design means it could be capable of much more. OpenSTA uses a proxy server to record requests and then generates scripts that can be edited to make them more configurable. The scripting language can be quite cryptic at times and the generated scripts are harder to maintain than JMeter handcoded scripts.

- Mercury LoadRunner (http://www.mercury.com/us/products/performance-center/loadrunner/) : the leading commercial performance testing tool.

You can find more tools that will help you test performance in our "Tools" section at the end of this document. Mostly what we have not included in this section are profiling tools that can be used on the server to diagnose the cause of performance bottlenecks.

EXTRACTING TEST INPUT DATA

In order to build the datasets for testing, usually you will need a list of valid page IDs that you can use to test browsing the site for example. You can retrieve the list of accessible page ID with the following SQL request :

```
SELECT id_jahia_pages_data FROM jahia_pages_data WHERE
pagetype_jahia_pages_data=0 AND workflow_state=1 ORDER BY
id_jahia_pages_data;
```

The above request returns the LIVE mode pages, i.e. pages that have already been validated. If the objective is to load test in EDIT mode, the request must be slightly modified to include the non-validated pages :

```
SELECT DISTINCT id_jahia_pages_data FROM jahia_pages_data WHERE
pagetype_jahia_pages_data=0 AND workflow_state>=1 ORDER BY
id_jahia_pages_data;
```

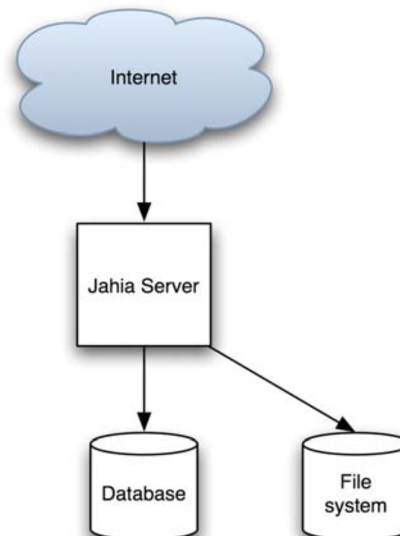
CASE STUDIES

INTRODUCTION

In this section we will present three case studies that illustrate common Jahia deployment scenarios. We will talk about the typical use of such systems, the strength and limitations of each configuration, as well as the possibility to customize the deployment depending on the specific needs.

SMALL JAHIA SITE

Jahia is quite able to provide a lot of functionality on a single server, and this configuration is aimed at smaller installations such as personal websites, small intranets, etc. In this configuration the Jahia installation doesn't have any requirements for high-availability or high performance, and background tasks such as XML imports or indexing will directly influence the overall performance. The configuration will look something like the following diagram :

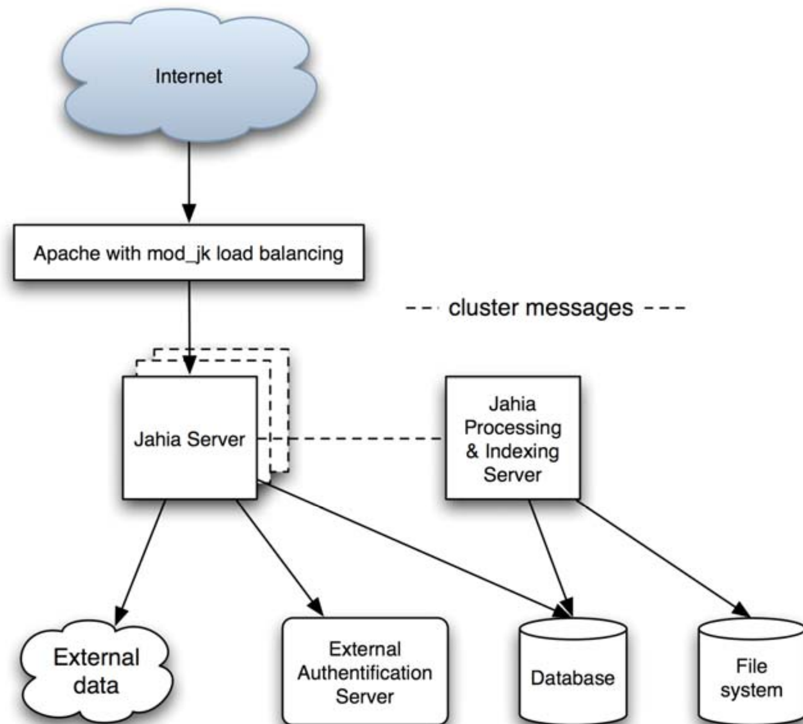


This means that the single Jahia server will handle all the caching, the browsing, the editing, the workflow processing, XML import, indexing and many other functions in an all-in-one server. User database is also completely handled by Jahia. Everything is stored in the database and the file-system (used for search indexes).

MEDIUM JAHIA SITE

The medium site case study is targeted towards high-volume mostly static sites. This doesn't mean that they cannot contain elements of personalization or external data, but these will be built on top of Jahia internal output cache system, rather than by-passing it.

In order to achieve both high-performance and dynamic rendering of content and personalization, tasks are delegated to both the server and the client. Basically the server will serve cached HTML for the pages, and answer small HTTP requests coming from the client. The client will generate AJAX requests to the server, in order to render the dynamic part over the static cached pages. This allows Jahia to be able to serve pages really fast without having to regenerate full pages so often.



In the above illustration we can see the following elements :

- Apache Web server with mod_jk connector or any other hardware load balancer
- Jahia server, that may be installed on different cluster nodes
- A separate Jahia server dedicated to processing and indexing
- External data sources, such as syndicated content

- External custom authentification server (in this example non-LDAP)
- The database containing all the data managed by Jahia
- The file-system used to store search indexes

This case study includes the following features :

- Medium amount of pages (in the thousands of pages range)
- Large number of browsing users, very few editors, not a lot of personalization
- Integration of external data through client-side data aggregation
- Integration of an external authentification server that is implemented as a Jahia user provider

The Apache Web Server is installed on top of the complete stack so that it may perform both URL rewriting and load-balancing functions. The URL rewriting configuration is useful for exposing different URLs than the standard Jahia ones, and also if there is a desire to separate browsing and editing loads on separate servers. The load-balancing is based on a sticky-session mechanism because Jahia sessions must stay on the same server.

The integration of external data through client-side aggregation is important because it allows the page served by Jahia to come directly from the output (HTML) cache, and the external data is then included through Javascript that loads it from another server and modifies the page's DOM (Document Object Model) to add the data as soon as it is loaded. Another possibility of doing aggregation on the client is to use IFrames to aggregate content. IFrames size is usually fixed, but it could be made flexible by using Javascript to automatically resize the IFrame to the content size (see http://www.dyn-web.com/dhtml/iframes/) .

User personalization is done the same way, basically upon login a cookie is inserted that is used by Javascript code to access external personalization data, or using the AJAX aggregation requests.

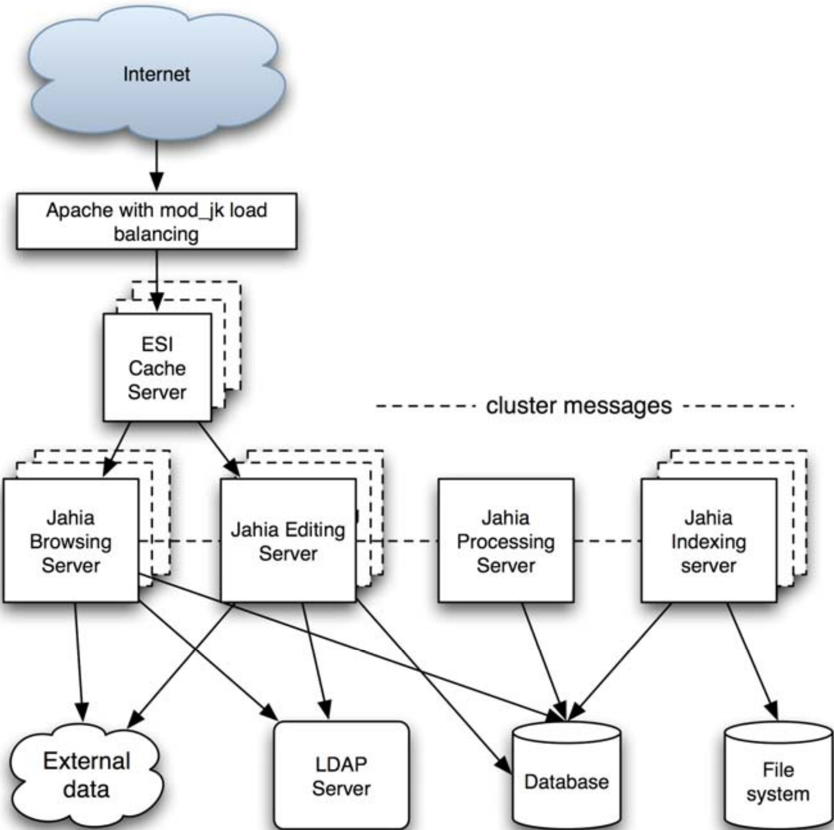
Another interesting part of this case study is that it integrates with an external authentification system that is non-LDAP. This is done through the custom development of Jahia user (and possibly group) providers, which is a pluggable interface. This of course is not a requirement and in most cases the LDAP provider will suffice as an authentification system, but we just wanted to illustrate the possibility to integrate with custom systems, as was done in this case study.

The clustering section of this case study allows to instantiate multiple times the Jahia server nodes, in order to handle more load. It would even be possible to extend on this setup to separate browsing and editing nodes,

much in the same way as we will present it in the “large Jahia site” case study. A separate Jahia server node is dedicated for both processing and indexing workloads. This avoids loading the browsing and editing nodes with time-consuming operations such workflows, XML importing and indexing. It also allows the “simple” nodes to keep all available memory to support as many concurrent users as possible. In the above illustration we have illustrated the horizontal traffic that flows between the various Jahia server nodes, which is different from the request processing traffic that is presented vertically.

LARGE JAHIA SITE

The large Jahia site case study is similar to an “all-you-can-eat” configuration of Jahia, and is possibly the “higher-end” Jahia installation. Such an environment is geared towards high-traffic and highly dynamic web site usage. It includes personalization, portlet usage, filters, sorters and searchers, separate browsing and editing servers, possibility to fail-over, LDAP server connection for user authentification, etc. The configuration we will present here can be summarized in the following diagram.



The above illustration is basically an expanded version of the medium site configuration. What is new is the introduction of the ESI front-end cache proxy server as well as the separation of Jahia servers into more granular specialized servers.

So basically the installation is composed of :

- A load-balancing Apache Web Server with a mod_rewrite and mod_jk configuration, or any other hardware load balancer. This server distributes the load over the various ESI cache servers, and is in charge of separating browsing and editing mode load. It is also possible to extend the configuration of the Apache configuration to support fail-over. More information about this type of deployment can be found in the below fail-over section.

- The ESI cache server is responsible of caching fragment of pages as well as the complete generated page. The idea here is to take advantage of fragment re-use across pages as well as sharing of fragments among similar groups of users, and to make it possible to mix both "static" and dynamic elements on a page, reducing performance impact to a minimum. So in this configuration the ESI server is also responsible of dispatching to the below Jahia servers, and balancing the load on them if desired.

- Jahia browsing nodes : these are basically Jahia nodes reserved for navigating the web site. The simplest way of doing this is to use Apache Web Server rewrite rules to choose which servers are dedicated to browsing, and which are dedicated to editing. This allows for fail-over configurations too, as we will see in more detail below.

- Jahia editing nodes : these nodes are dedicated to the workload generated by content editors.

- The Jahia processing server is in charge of handling all the long running jobs, except for indexing. This server has a requirement of being the only instance of this type, because of the potential corruption that could occur if two long-running jobs were running concurrently. In order to reliably perform jobs, they are serialized in the database, and should the server ever be shutdown and restarted, it will restart the jobs. Internally it uses the open-source Quartz enterprise scheduler (http://www.opensymphony.com/quartz/) that is a very reliable and flexible implementation of a job processing system.

- The Jahia indexing server is in charge of both the text extraction and the indexing of the content inserted into Jahia. The extraction of text from Adobe PDF or Microsoft Office is a heavy operation that uses up both CPU and memory, and therefore it is very important to separate this load on high-end installations that will contain large number of office files. This server can be replicated to scale out the configuration

to better distribute load or to provide fail-over using mirrored file-systems.

- External data : the integration of external data in this case study is different from the medium site configuration, and can be performed with portlets for example. This is another reason why it is important to distribute browsing load on multiple machines, as they will also process the logic operations that the portlets contain. It is very important therefore to also make sure the portlets are very efficiently implemented, because in performance as in security the weakest link can bring the whole system down, so a portlet accessing a remote data server, if not down properly, could seriously hinder performance. Another way of integrating external data could be through the use of custom tag libraries integrated in Jahia templates. In this latter example, an integration with ESI fragment caching is also strongly recommended in order to benefit from the fragment optimizations that this server offers, in order to reduce the load on the Jahia server as well as the external data server.

- LDAP server : quite common in enterprise deployments is the integration of Jahia with a directory service. Jahia works well with various LDAP implementations such as Active Directory, OpenLDAP, Novell LDAP.

- Database : in this type of high-end configuration, the database choice and configuration is critical. As the number of requests can become very important, the proper dimensioning of the database server is critical. The chosen database should also be installed on powerful hardware, including lots of memory, and possibly also be clustered.

Scaling out the configuration

In the above illustration, you will notice that the ESI cache server, the Jahia browsing and editing nodes as well as the indexing server all have dashed boxes behind them. These represent the possibility to extend the load balancing by replicating these nodes multiple times in order to better distribute load as well as offer better availability.

Other important parts of the clustering configuration include of course the Apache Web Server that must be properly configured to distribute load, as well as the back-end servers. The LDAP server must be able to handle properly the larger amount of requests that will be generated by a growing number of nodes, and the external data server must also be able to scale out properly.

The database is really critical in terms of scaling out, and must be dimensioned properly. It is not the purpose of this document to explain how to achieve this, as each database implementation usually has good documentation on how to improve performance and scale out, but it should

be remembered that when growing the installation for higher performance, the database is an extremely critical part of the equation.

For more information on clustering we refer you to the previously presented clustering description in this document.

Fail-over configurations

In the proposed configuration, it is possible to introduce fail-over installations, mostly by using the Apache web server with a powerful URL rewriting configuration that will test first the availability of the Jahia server before dispatching to them.

The really powerful mod_rewrite module of the Apache Web Server can be extended to do all sort of things using the RewriteMap configuration parameter that allows external programs to provide mapping. Here on the custom deployment you could develop (or integrate) a high-availability system that would check the status of the servers below, and provide a mapping function for the rewriting module. As of the time of writing though we do not provide such a script as part of our Jahia configuration, but it should just be noted that it is technically possible to achieve this type of install.

At the ESI cache server level, the cache server is capable of being connected to multiple Jahia server, and will modify its dispatching rules dynamically if a Jahia server becomes unavailable, and bring it back into the server pool once it is available again.

Jahia browsing nodes are configured actually as fully capable nodes, that include the possibility to perform edit loads, because of fail-over requirements. Basically the Apache Web Server is in charge of limiting the edit load to a subset of servers, checking if they are indeed available for editing, and if not the script should be capable of detecting the failure, and redirecting the editing load to the browsing nodes temporarily.

The fail-over configuration is more limited for the processing server, as it has inherent limitations because the long-running processes risk interfering with each other. Fortunately all the jobs are also stored in the database, so even if the server goes down and comes back up, the jobs will restart as expected. Therefore it is critical to have availability monitoring systems installed so that the downtime of the processing server is quickly detected and so that it may be restarted rapidly.

The indexing servers may be replicated to provider fail-over, but they also work the same way as the processing server, so fail-over is really dependent on quick failure detection and restart of the server, in order to guarantee system integrity. Also as they store their indexes on a file-system, it should also be setup so that a mirror file-system can takeover in case of failure. The configuration of such a setup is not in the scope of this document though, but

let it be known that most modern operating systems offer solutions for problems like these.

The last important part of the fail-over configuration is the back-end system. The external data server must be also made reliably available, or at the minimum the front-end system accessing it must tolerate failure and possibly display a message asking to try again later. The LDAP server must be configured in a fault-tolerant configuration. Portlets and any back-end they may access should properly handle failure conditions, and possibly simply mark unavailability to the end-user if something goes wrong, until the system can be made available again. The most important one is the database, that must be really highly available, as any downtime could lead to serious data corruption. There are here fortunately many solutions specific to each database vendor, and it is usually not too much a problem of function but rather an issue of configuration and licensing cost.

TEMPLATE FINE-TUNING

DURING DEVELOPMENT

1) Use the Jahia API with care

If you plan to develop advanced Jahia features in addition to basic templates (for example if you plan to automatically create a default set of pages for each new virtual site or if you plan to automatically import external content into the Jahia content repository), please take care to fully understand the Jahia model and to fully and extensively test your custom developments before putting them in production and in the hand of the end-users. Certain Jahia operations are quite complex and tricky (e.g. the page creation operation). Having full access to the source code and to all the underlying Jahia classes, doesn't imply that you can use them as-is without fully understanding the Jahia object model. You risk corrupting the integrity of the database by forgetting to integrate certain mandatory checks or certain tests in your code.

Moreover the Jahia developers does not guarantee that the Jahia API will not change from a point release to another. So your workarounds or your advanced features may not work anymore in the next releases. Consequently, think twice before implementing such advanced features. Always think if it isn't better to contribute your changes to the community or to sponsor a longer term development rather than implementing a short term custom workaround which risks becoming rapidly unmaintainable and may corrupt your whole content model if badly implemented.

Also, if you are trying to import content, please always refer to the built-in features, such as the XML import mechanism that can cover a lot of needs.

2) Use OSCache or ESI to cache fragments

OSCache can perfectly be used for fragments of the page, which take much time to render and where the result is always the same, especially if it is the same on every page (header, footer, menu, last 5 news,...). However you might want to use it only in LIVE mode, as in EDIT mode you might want to always see the latest items, even if the page render time is a little slower.

You could do it like this:

```
<%
ContainerBean boxCont = (ContainerBean)
pageContext.getAttribute("boxContainerBean");
int boxContainerId = boxCont.getId();
```

```
boolean isCacheActive = jData.gui().isNormalMode();
%>
<cache:cache time='<%= isCacheActive ? 120 : 0 %>'
    key='<%="box" + (new Integer(boxContainerId)).toString()%>'>

    [... the display of the box...]

</cache:cache>
```

In this example the box will be cached in LIVE mode for 120 seconds. You can decide whether you want to cache a fragment for minutes or hours. It depends how often it changes. For instance the header and footer will not change often. For the navigation, it might happen more frequent that new pages are coming or old pages are being deleted. In order to prevent 404 errors, if a fragment is still showing a page, which has been deleted, you could also call

```
<cache:flush key='<%="box" + (new Integer(boxContainerId)).toString()%>' />
```

in EDIT mode on the JSP, where the container gets changed. If you are using a cluster, then you should also consider the configuration, described here: http://www.opensymphony.com/oscache/wiki/Clustering.html.

For the key, it also depends whether you are using access rights on the container list, so each user might have a different view. For this case you might want to add the user-id to the key, then the fragment will be cached per user. You should surely limit the time on these fragments.

To use the OSCache you need to deploy the JAR to the lib directory of the Jahia webapp, you need to add the oscache.tld to the web.xml and then you can use the taglib in your templates.

As mentioned in the title of this tip, it is also possible to ESI to cache fragments, and this behaves much in a similar way than the OSCache solution. For more information on how to use ESI to cache page fragments, please refer to the ESI integration documentation.

3) Avoid doing unnecessary container and field definitions

You can re-use field definitions between multiple container definitions (if using the Jahia API calls). Also, container defs and fields defs are shared for all pages using the same template, no need to instantiate them more than that.

4) Avoid calculating default values for field definition dynamically

Avoid calculating default values for field definition dynamically if possible. If a default value is calculated for example using the current user name, the definition will be modified each time a new user comes on the template. Updating container definitions is an expensive operation and might lead to

coherency problems if a template is loading content from a definition while it is being updated

5) Avoid making definitions on pages that will only retrieve the container list content

This is especially true for absolute container list usages. We do not need the definition to be executed just to retrieve the content of the container list.

6) Avoid building interfaces with lots of hierarchies

Avoid building interfaces with lots of hierarchies, as retrieving sub-pages and content can become expensive operations as the sites grow larger. An example of this is the classic DHTML scroll-down menu, which, if displayed on every page of the website, can become a real performance problem because Jahia will have to track it's usage on all pages, to know which pages to flush when content is added / modified (when the built-in full page HTML cache is used).

7) Think about total page loading time

When designing Jahia templates, remember that the total page loading time not only includes the HTML generation, but also all the surrounding resources that will be loaded from the HTML links such as: Javascript source files, CSS files and images. You can use a tool like Firebug (http://www.getfirebug.com/) for Firefox to analyze total page loading time. A slow loading resource can also give the impression that the whole system is slow because some browsers "wait" on a resource before displaying the page. Also check the configuration of your application server so that it has enough free connections to serve all the requests **including** the resources. Also, remember that access to external resources such as external images, RSS feeds, external Javascript files or other resources can cause problems if the connection with the external site is slow, or if the Internet connection is not always available.

IN PRODUCTION

8) Precompile your set of templates

If you do not plan to change your set of templates often, you may precompile your templates with the jspc command or another similar tool. By default, the Jahia engines (i.e. the edition pop-ups) are already pre-compiled.

In Jahia 5 we have also introduced a new servlet, which you can trigger after deployment of new templates or template changes. Simply call

http://<yourservername:port>/jahia/precompileServlet and then select an option by clicking on the link.

JAHIA TUNING

DURING DEVELOPMENT

9) Avoid loading the webapps/portlets if they are not necessary

By default and for demo purposes, each bundled web application is an independent servlet which launches its own embedded Hypersonic database. Each web application is then loaded in memory with its own embedded database at startup. This behavior can rapidly cause "Out of Memory" problems and/or some performance issues.

Certain web applications can be configured to use another data source than the embedded one (e.g. the database used for your Jahia installation). So instead of launching a separate database for each web application, it is recommend to centralize all your web application data into one centralized storage system. This will also simplify database backups.

Warning: Not all the web applications provided by default with Jahia have been developed to be able to use another data source. Modifications to the web application may be required.

IN PRODUCTION

10) 64-bit installations

As Jahia is quite demanding in terms of operating memory, it is strongly recommended to install Jahia on a 64-bit operating system, so that the JVM can allocate more than 1.5GB of RAM. The more available memory a Jahia installations has, the less it will reach potential memory saturation, which can be both bad for performance and stability.

11) Prefer UNIX-based OS

Prefer a UNIX-based install, as they are much faster on I/O than Microsoft Windows systems. This is especially true if you are using open-source databases such as MySQL or PostgreSQL that are highly optimized under Linux, but a lot less on Windows.

12) Reduce logs verbosity

Do not forget to reduce log verbosity on the production server. Jahia makes use of the Log4J library for all debugging info. Log4J defines some logging levels as follows (from the more to the less verbose):

```
ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF
←  More Verbose                    Less Verbose →
```

By default, point releases of Jahia (as opposed to SVN builds) have the log levels set to INFO. If you want to increase the log level to trace a problem, you will need to modify the log4j.xml file located in the following directory:

```
%TOMCAT_HOME%/webapps/jahia/WEB-INF/etc/config/
(e.g. C:\jahia405 \tomcat\webapps\jahia\WEB-INF\etc\config)
```

At the bottom of the file, you have the <root>... </root> part. Change the <priority value="info"/> to <priority value="debug"/> for example to have more debugging information in the console.

Setting Debug-level logging in log4j.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM
"log4j.dtd">

<log4j:configuration
xmlns:log4j="http://jakarta.apache.org/log4j/
">

….

   <root>
       <priority value="debug"/>
       <appender-ref ref="STDOUT"/>
       <!--
       <appender-ref ref="Chainsaw"/>
       -->
   </root>
</log4j:configuration>
```

Setting Info-level logging in log4j.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM
"log4j.dtd">

<log4j:configuration
xmlns:log4j="http://jakarta.apache.org/log4j/
">

….

    <root>
       <priority value="info"/>
       <appender-ref ref="STDOUT"/>
       <!--
       <appender-ref ref="Chainsaw"/>
       -->
    </root>
</log4j:configuration>
```

Don't forget to change back the values to INFO, as DEBUG log level has a pretty important impact on performance. If you encounter a problem afterwards, you can perfectly temporarily switch the production server to DEBUG mode (for example to have the time to get the full stack trace exceptions). This can be done without rebooting since Jahia automatically detects all the changes in log4j.xml every 30 seconds.

13) Modify your default JVM settings

We recommend to use the following JVM settings (for Tomcat set in catalina.bat or catalina.sh) when using Sun JVMs:

```
set CATALINA_OPTS=%CATALINA_OPTS% -server -Xms128m -Xmx1536m -
XX:MaxPermSize=128m -XX:+UseParNewGC -XX:NewRatio=4 –
Dsun.io.useCanonCaches=false
```

Here are the details of each option and it's significance :

```
-server
```

Use Server Hotspot VM. Must be the first option.

Starting with Sun JDK 5.0 this option is set automatically for Solaris or Linux, whenever the machine has at least 2 CPUs and at least 2GB of physical memory. On Microsoft Windows platforms the parameter is never set automatically. So we recommend to set it manually.

```
-Xms128m -Xmx1536m
```

Jahia is quite memory consuming, so you may typically want to increase the -Xmx maximum memory value to the highest possible one supported by your system or configure the JVM settings for your specific usage. More information about the SUN JVM configurations can is available here:

http://java.sun.com/docs/hotspot/VMOptions.html

As the maximum heap size is limited on 32-bit systems, we are recommending to use a 64-bit operating system.

```
-XX:MaxPermSize=128m
```

There are reported issues with Jahia, when the JVM ran out of space in the permanent generation heap. Therefore we recommend to increase the value to 128 MB.

```
-XX:+UseParNewGC -XX:NewRatio=4
```

If you are using a Sun JVM 1.4.1_x or 1.4.2_x on a machine with 2 or more CPUs, then we strongly recommend using the parallel copying collector and to use a ratio of young generation to old generation of 4 (or even 3), what means that the space for young objects is one fifth of the heap size (ratio 1:4). The default is in most set ups 8, thus the young generation is too small and the garbage collection will slow down your system. Starting with Sun JDK 5.0 the default settings regarding garbage collection are better, but especially on Windows systems with more than 2 CPUs we are also recommending you to try those two settings and see whether you get a better performance.

```
-Dsun.io.useCanonCaches=false
```

This parameter is required to support Windows file names.

You may also install and use other JVMs (IBM, SUN, etc…). Be warned though that although these JVMs may run faster, they haven't necessarily been tested with Jahia. Small differences may then occur and lead to unexpected behaviors especially under heavy loads. If you change the default SUN JVM, we suggest carefully monitoring your site over a period of time in order to verify that everything is running smoothly. Some of the settings we described are not available in JVMs of the other vendors

14) Fine-tune your connection pool

This point is one of the most important. Jahia relies heavily on a fast and efficient database. So it is critical not to forget to allocate enough connections to the database.

Jahia relies on the Apache Database Connection Pool. You can consult a detailed description of all the available configuration options here: http://jakarta.apache.org/commons/dbcp/configuration.html. By default Jahia is already installed with the maxActive value increased to 100 (instead of 8).

If your database supports query caching, please enable it. For example MySQL 4.0.1+ integrates an internal cache. By default this cache is not turned on. Using it will increase a lot the overall Jahia performance (more information available on the MySQL cache here: http://dev.mysql.com/doc/mysql/en/query-cache.html)

Jahia also can use the client side prepared statement pool, while in the versions before we were using a custom implementation. Now you can set this setting in %TOMCAT_HOME%/conf/Catalina/localhost/jahia.xml by adding the following parameter to the <Resource> tags in the XML file :

```
poolPreparedStatements="true"
```

So your XML file should look something like this (default HSQLDB configuration shown here, yours may have different values, but don't modify them) :

```
<Resource name="jdbc/jetspeed" auth="Container"
            factory="org.apache.commons.dbcp.BasicDataSourceFactory"
            type="javax.sql.DataSource" username="sa" password=""
            driverClassName="org.hsqldb.jdbcDriver"
url="jdbc:hsqldb:hsql://localhost" defaultAutoCommit="true"
            maxActive="100" maxIdle="30" maxWait="10000"
poolPreparedStatements="true" />

<Resource name="jdbc/jetspeedNonTx" auth="Container"
            factory="org.apache.commons.dbcp.BasicDataSourceFactory"
            type="javax.sql.DataSource" username="sa" password=""
            driverClassName="org.hsqldb.jdbcDriver"
url="jdbc:hsqldb:hsql://localhost" defaultAutoCommit="true"
            maxActive="20" maxIdle="30" maxWait="10000"
poolPreparedStatements="true" />
```

15) Install your Tomcat as a service

Once in production, you may want to run your Tomcat application server as a service. Please refer to the Jahia technical FAQ to get more information about how to run Tomcat as a service:
http://www.jahia.org/jahia/page454.html#3 or to the Tomcat documentation.

16) Fine-tune Tomcat configuration

For your production system, please ensure that the following Tomcat parameters are set in %TOMCAT_HOME%/conf/web.xml:

<table>
<tr><th>Settings for DefaultServlet</th><th>Description</th></tr>
<tr><td><pre><servlet>
  <servlet-name>default</servlet-name>
  ...
    <init-param>
      <param-name>listings</param-name>
      <param-value>false</param-value>
    </init-param>
  ...
  </servlet></pre></td><td>listings should be set to false, because of security and because the robots of the search machines might use the output of directories to call JSPs directly, which will lead to exceptions.</td></tr>
</table>

<table>
<tr><th>Settings for JSP servlet</th><th>Description</th></tr>
<tr><td><pre><servlet>
  <servlet-name>jsp</servlet-name>
  ...
    <init-param>
      <param-name>fork</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>development</param-name>
      <param-value>false</param-value>
    </init-param>

From Tomcat 5 onwards:

    <init-param>
      <param-name>trimSpaces</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>genStrAsCharArray</param-
name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>checkInterval</param-
name>
      <param-value>300</param-value>
    </init-param>
  ...
  </servlet></pre></td><td>fork should be set to true, as the compiler leaks memory and should be called in a separate process. This way it also does not block the JVM.

development should be set to false, as this way Tomcat will not check, whether the JSP changed on every request, but will cache JSPs for a while (checkInterval parameter).

When using Tomcat 5 or newer, you should also consider:

trimSpaces should be set to true, what will compact the HTML results and delete all empty lines
genStrAsCharArray should be set to true as it increases performance on some JSPs
checkInterval should be set to 300, as with Tomcat 5 the default is 0, while with Tomcat 4 the default is 300. With 0 changes in JSPs are not considered , while the system is running.</td></tr>
</table>

17) Periodically backup your Jahia environment

Do not forget to customize automatic backup procedures of your Jahia environment. You may want to read the following Jahia FAQ to get more information about how to backup Jahia:
http://www.jahia.net/jahia/Jahia/site/jahia_net/pid/589#14 or read the Jahia Administration Guide. Basically to be sure not to forget anything, backup everything in your Tomcat directory and your entire database.

18) Monitor your Jahia Server

Once in production and running, you may also want to monitor your Jahia server in order to react promptly in case of downtime. Please use monitoring tools such as IPSentry (http://www.ipsentry.com/) or others to monitor your Jahia server.

19) Do not forget to upgrade to the latest Jahia release

Before sending email in the Jahia mailing lists or calling a Jahia expert, please upgrade to the latest available Jahia release. Hundreds of bugs are corrected from a service pack to another. It is possible that the bug or the performance issue you are currently experiencing is already corrected in the latest release. In any case, it is highly recommended, for security and stability reasons, to be up to date with your Jahia release.

20) Do not forget to remove or modify the password of the default Jahia users

Do not forget to remove in production the default Jahia users. Usually, Jahia is installed with the default administrator "siteadmin". So please do not forget to modify the default password of this user or to delete him.

21) Fine-tune some Jahia properties

In Jahia there are some parameters for fine tuning your Jahia server. You should edit:

```
%TOMCAT_HOME%/webapps/jahia/WEB-INF/etc/config/jahia.properties
```

Here is the list of the settings you should consider :

```
maxParallelProcessings = 50
```

With this parameter you can control how many threads in Jahia should be used to do page render processing. Requests, which are served from the HTML cache are not affected. This way you can prevent for instance OutOfMemory exceptions, when the load is high, and also the performance

will be more stable. The default it is set to 10 threads, but you can raise this value depending of the power of your production server. By simulating a heavy load with parallel requests from different users or for different pages, you could determine, how many processing threads are still improving your performance.

```
pageGenerationWaitTime = 20000
```

If all processing threads are working, then this setting is used for the waiting threads, which need to wait until the first processing thread finishes its work. In order to not wait endlessly, there is a timeout in milliseconds, which is set to 1800000 on default (30 minutes). Set this to a short value, because users should be rejected quickly. A good value would be 20-30 seconds.

```
org.jahia.acl.preload_active=false
```

For large volumes of content, this will deactivate ACL preloading upon Jahia startup. Large sites can have huge numbers of ACLs in the database (it is not uncommon to see 100'000 or more ACLs in the database), so preloading them can cause large startup times of the Jahia application. In this case it becomes more interesting to startup fast (high availability) even if this means that initial page generation time will be a little slower. This is compensated quickly over time.

```
org.jahia.workflow.preload_active=false
```

Much in the same way that Jahia preloads ACLs on startup, it also pre-calculates Workflow status for a certain number of page levels in each site. In large sites, this can again slow down Jahia startup. We recommend deactivating preloading, as it is best to have the system up and running quickly, in order to avoid down time.

```
editModeSessionTimeout=7200
```

This setting will set the session duration for editing users to 2 hours instead of the default session expiration. In tomcat/conf/web.xml the following setting should be lowered from 30 minutes to something like 10 minutes to save memory :

```
<session-timeout>10</session-timeout>
```

Warning : users using online forms will also be expired using the general timeout, so please take this into account when setting the global session expiration. Another possibility would be to expand the session length by adding a call to the session.setMaxInactiveInterval in the template that needs the extended timeout. This method is documented here : http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpSession.html#setMaxInactiveInterval(int)

If you are using the HTML output cache with sites that have larger number of pages (more than 5000), it must also be updated to use the new expiration-only mode for live content (EDIT mode content will no longer be cached !)

```
outputCacheExpirationOnly=true
outputCacheDefaultExpirationDelay=3600000
```

The last value is in milliseconds and represents an expiration delay of an hour. You might have to adjust this setting to your own needs. Also you might want to prefer the container HTML cache that is the new default cache in Jahia and performs better on large sites.

22) Adjust the "permission to enter EDIT mode" role

Jahia has introduced in 5.0 Service Pack 2 a new "Permission to enter EDIT mode" setting, that lets you specify the groups of users that are allowed to actually switch to EDIT mode. This was done because the precise resolution of entering EDIT mode requires the resolution of all write permissions on the content of the whole page. As this is an expensive operation, especially for users that will never have write permissions, we introduced the possibility to control the switching to EDIT. For compatibility reasons, the default is that this permission is given to all users, which makes for a transparent migration and no change in functionality. But in order to fully benefit from the performance gain this new function can bring you, you should for example define a "content editors" groups that contains all the people that will be given write access on content, and set the EDIT mode permission to include only this group of users.

23) Preload your site in memory

You may want to preload pages in the Jahia HTML cache after a reboot or during low activity period (e.g. during the night). You can use any kind of offline site browser utility such as wget or httrack (http://www.httrack.com) to preload Jahia pages in memory.

Please do not forget to limit the maximum number of connections per second if you want such a preloading to be as transparent for the end-users as possible.

You should then customize your utility in order to avoid browsing the sitemap or to avoid to include parameters beginning with "?".

24) Use JAMon to diagnose performance problems

In order to more finely diagnose performance problems in production, Jahia already pre-integrates a performance monitoring tool called JAMon (http://jamonapi.sourceforge.net/). This tool enables you to see in real time the performance of the internal Jahia services and figure out which back-end system is causing any bottlenecks. This might be a good way to for example illustrate a performance problem of a remote LDAP server, or the database server that might not be answering requests fast enough. It should be noted that JAMon can be activated "on-the-fly" while the Jahia server is running,

without requiring a restart, and that it has about a 10-20% impact on performance while monitoring is active. So it is an ideal way to start examining internal performance on production system without the need to install any complex tool.

For more information on using JAMon to monitor Jahia performance, you can check out our Monitoring howto guide that is included in each Jahia install, and also available online here :
http://www.jahia.net/download/jahia5.0/stable/howto_monitoring.html

25) Fine-tune Lucene parameters

The Lucene indexation default parameters have been updated to speed up lucene indexation. You should make the following modifications in the file WEB-INF/etc/spring/applicationcontext-basejahiaconfig.xml :

```
<prop key="org.apache.lucene.mergeFactor">30</prop> <!-- previously 10 -->
<prop key="org.apache.lucene.maxBufferedDocs">1000</prop><!-- previously 50 -
->
<prop key="indexingJobBatchSize">500</prop><!-- previously 100 -->
```

To reduce memory usage if you encounter too much OutOfMemoryException, you can adjust the org.apache.lucene.mergeFactor property with smaller value, 20, 10, 5. But smaller value means slower indexation.

26) Delayed indexation

In the SP3 release we added new options in order to delay (re)indexation of the jahia content objects to a given hour (e.g midnight). We implemented 3 rules conditions such as:

By content type: All content typesor BlogEntries container etc…

By file field extension: File field with PDf, doc, xls or ppt extension

By page sub tree : Sub pages child of a given page node

To activate them you need to uncomment the bean you want in the configuration file : WEB-INF/etc/spring/applicationcontext-indexationpolicy.xml

The list of rules for Jahia Content Indexation and File Field indexation are defined using the beans “contentIndexationRulesList” and “fileFieldIndexationRulesList”:

```
<bean id="contentIndexationRulesList"
class="org.springframework.beans.factory.config.ListFactoryBean">
<property name="sourceList">
<list>
<!-- uncomment each rule bean you want to add to the list -->
<!--ref bean="delayedContentIndexationRule"/-->
```

```
<!--ref bean="blogContentIndexationRule"/-->
<!--ref bean="excludedSubTreeIndexationRule"/-->
</list>
</property>
</bean>

<bean id="fileFieldIndexationRulesList"
class="org.springframework.beans.factory.config.ListFactoryBean">
<property name="sourceList">
<list>
<!-- uncomment each rule bean you want to add to the list -->
<!--ref bean="delayedFileFieldIndexationRule"/-->
</list>
</property>
</bean>
```

These two list beans are used to set up the JahiaSearchIndexationService in the file :WEB-INF/etc/spring/applicationcontext-services.xml

```
<bean id="JahiaSearchIndexationService" parent="proxyTemplate">
<property name="target">
<bean
class="org.jahia.services.search.indexingscheduler.JahiaSearchIndexationServi
ce" parent="jahiaServiceTemplate" factory-method="getInstance">
<property name="contentIndexationRules">
<ref bean="contentIndexationRulesList"/>
</property>
<property name="fileFieldIndexationRules">
<ref bean="fileFieldIndexationRulesList"/>
</property>
</bean>
</property>
</bean>
```

These two lists are empty by default, so that, this is the normal indexation behaviour that will be used.

The applicationcontext-indexationpolicy.xml configuration file comes with some sample declaration of indexation rules:

1) Example of rule that can be used to delay the indexation for all Jahia Content:

```
<bean id="delayedContentIndexationRule"
class="org.jahia.services.search.indexingscheduler.impl.rule.BaseIndexationRu
le">
        <property name="id" value="1"/><!-- a unique id must be assigned to
each rule -->
        <property name="namedID" value="delayedContentIndexationRule"/><!--
optional unique name used for declarative rule. -->
        <property name="name" value="Delayed Content Indexation Rule"/><!--
human readable name or description -->
        <property name="indexationMode" value="2"/><!-- Indexation mode :
                                                        0 = don't index,
                                                        1 = index immediately
( as soon as possible ),
```

```
                                                        2 = scheduled at
specified time -->
        <property name="conditions"><!-- a rule returns true only if all its
conditions evaluate to true -->
            <list>
                <bean
class="org.jahia.services.search.indexingscheduler.impl.condition.ContentType
RuleCondition">
                    <property name="allowAll" value="true"/>
                    <!-- In case allowAll is set to false, more precise
content type pattern can be defined
                         "ContentPage" : match all pages
                         "ContentContainer" : match all containers
                         "ContentContainer|name_BlogContainer" : match all
containers with the definition name "BlogContainer"
                         "ContentContainer|id_43" : match all containers
using the definition with the given ID=43
                    -->
                    <property name="allowedContentTypes">
                        <list>
                            <value>ContentPage</value>
                            <value>ContentContainer</value>
                            <!--
value>ContentContainer|name_BlogContainer</value-->
                            <!--value>ContentContainer|id_43</value-->
                        </list>
                    </property>
                </bean>
            </list>
        </property>
        <property name="dailyIndexationTimes"><!-- the range of allowed
indexation time. Only used when the indexationMode == 2 -->
            <list>
                <bean
class="org.jahia.services.search.indexingscheduler.TimeRange">
                    <property name="startHour" value="23"/>
                    <property name="startMinute" value="00"/>
                    <property name="endHour" value="3"/>
                    <property name="endMinute" value="00"/>
                </bean>
            </list>
        </property>
    </bean>
```

2) Example of rule that can be used to delay the indexation of all File Field with a PDF or Office file:

```
<bean id="delayedFileFieldIndexationRule"
class="org.jahia.services.search.indexingscheduler.impl.rule.BaseIndexationRu
le">
        <property name="id" value="2"/><!-- a unique id must be assigned to
each rule -->
        <property name="namedID" value="delayedFileFieldIndexationRule"/><!--
optional unique name used for declarative rule. -->
        <property name="name" value="Delayed File Field Indexation Rule"/><!-
- human readable name or description -->
        <property name="indexationMode" value="2"/><!-- Indexation mode :
                                                        0 = don't index,
                                                        1 = index immediately
( as soon as possible ),
                                                        2 = scheduled at
specified time -->
        <property name="conditions"><!-- a rule returns true only if all its
conditions evaluate to true -->
```

```
            <list>
                <bean
class="org.jahia.services.search.indexingscheduler.impl.condition.FileFieldRu
leCondition">
                    <property name="fileExtensions">
                        <list>
                            <value>.pdf</value>
                            <value>.doc</value>
                            <value>.xls</value>
                            <value>.ppt</value>
                        </list>
                    </property>
                </bean>
            </list>
        </property>
        <property name="dailyIndexationTimes"><!-- the range of allowed
indexation time. Only used when the indexationMode == 2 -->
            <list>
                <bean
class="org.jahia.services.search.indexingscheduler.TimeRange">
                    <property name="startHour" value="23"/>
                    <property name="startMinute" value="00"/>
                    <property name="endHour" value="3"/>
                    <property name="endMinute" value="00"/>
                </bean>
            </list>
        </property>
    </bean>
```

3) Example of rule that can be used to schedule Blog Content for immediate indexation:

```
<bean id="blogContentIndexationRule"
class="org.jahia.services.search.indexingscheduler.impl.rule.BaseIndexationRu
le">
        <property name="id" value="3"/><!-- a unique id must be assigned to
each rule -->
        <property name="namedID" value="blogContentIndexationRule"/><!--
optional unique name used for declarative rule. -->
        <property name="name" value="Blog Content Indexation Rule"/><!--
human readable name or description -->
        <property name="indexationMode" value="1"/><!-- Indexation mode :
                                                        0 = don't index,
                                                        1 = index immediately
( as soon as possible ),
                                                        2 = scheduled at
specified time -->
        <property name="conditions"><!-- a rule returns true only if all its
conditions evaluate to true -->
            <list>
                <bean
class="org.jahia.services.search.indexingscheduler.impl.condition.ContentType
RuleCondition">
                    <property name="allowAll" value="false"/>
                    <!-- In case allowAll is set to false, more precise
content type pattern can be defined
                         "ContentPage" : match all pages
                         "ContentContainer" : match all containers
                         "ContentContainer|name_BlogContainer" : match all
containers with the definition name "BlogContainer"
                         "ContentContainer|id_43" : match all containers
using the definition with the given ID=43
                    -->
                    <property name="allowedContentTypes">
```

```
                        <list>
                            <value>ContentPage|name_blog</value><!—match Page
of template blog -->
                            <value>ContentPage|name_blog_listing</value><!—
match Page of template blog_listing

<value>ContentContainer|name_blogEntries</value><!—match container of type
blogEntries
                            <value>ContentContainer|name_comments</value><!—
match container of type comments
                        </list>
                    </property>
                </bean>
            </list>
        </property>
    </bean>
```

4) Example of rule that can be used to exclude a whole sub tree from indexation:

```
<bean id="excludedSubTreeIndexationRule"
class="org.jahia.services.search.indexingscheduler.impl.rule.BaseIndexationRu
le">
        <property name="id" value="4"/>
        <property name="namedID" value="excludedSubTreeIndexationRule"/>
        <property name="name" value="excluded SubTree Indexation Rule"/>
        <property name="indexationMode" value="0"/><!-- Indexation mode :
                                                        0 = don't index,
                                                        1 = index immediately
( as soon as possible ),
                                                        2 = scheduled at
specified time -->
        <property name="conditions">
            <list>
                <bean
class="org.jahia.services.search.indexingscheduler.impl.condition.ContentPage
PathRuleCondition">
                    <property name="parentNodePages">
                        <list>
                            <value>3</value><!-- will match all content that
are child or sub child of the parent page node 3 -->
                        </list>
                    </property>
                </bean>
            </list>
        </property>
    </bean>
```

PORTLETS FINE-TUNING

DURING DEVELOPMENT

27) Render must be really fast

When developing portlets, focus all your efforts on making sure that whatever your portlets render will be really fast, as this is where your application will be most stressed, as each time a page is requested, potentially the render (or do* methods) will be called.

So for example if your portlets are accessing remote data (database or other external systems), you might want to consider caching the default views to offer better performance to the end-user. Also using systems such as Hibernate or EJBs that offer built-in caching can be a good idea.

28) Use renderURLs instead of actionURL

Similar to optimizing rendering, try to avoid as much processAction calls as possible, especially if no data is being processed, only a default view is refreshed.

29) External portlet performance guide

You can find more information about portlet performance tuning here : http://edocs.bea.com/wlp/docs92/portlets/performance.html

It should be noted that some of these points are specific to the application server and not available on Jahia.

IN PRODUCTION

30) Limit the number of deployed portlets

As each portlet is actually contained in a full-fledged JEE application, it is a good idea to limit to a minimum the number of deployed portlets, as they each consume memory not only for their logic operations, but also for all the classes loaded in memory. Also large application will consume more memory just for their codebase, so plan your memory sizing accordingly.

DATABASE TUNING

DURING DEVELOPMENT

31) NEVER use your development database in production

While developing or customizing your new set of templates or portlets, you will most likely generate some Java exceptions. This will cause application server crashes or require Java processes to be terminated quite abruptly. All these actions may corrupt unfinished Jahia transactions and indirectly also corrupt your underlying database. Therefore never use a development database in production. Always migrate your set of templates, once fully tested, on a fresh or on a stable production server. If you want to test your templates with some real content, always copy the database and files from the production server to your development workstation. Never do the opposite.

IN PRODUCTION

32) Do not use HSQLDB in production.

Jahia is by default prepackaged with Hypersonic SQL. This is a small Java database embedded in Jahia for demo or development purposes only. We do not suggest using this database in production. If you want to use a free and open source database, please consider using MySQL or PostgreSQL.

33) Database profiling tools

Use database profiling tools to figure out if some tables are being accessed in sub-optimal way, adding indexes if necessary (for example SQL Profiler, for more information see the “Tools” section)

34) MySQL Query cache

Make sure database is properly configured for performance : activate query cache on MySQL. More information about the MySQL query cache can be found here : http://dev.mysql.com/doc/refman/5.0/en/query-cache.html

35) Vacuum the database

On PostgreSQL : schedule VACUUM FULL operations daily, and configure auto-vacuum so that it doesn't interfere with high-load times. You can find more information here :
http://www.postgresql.org/docs/8.2/interactive/sql-vacuum.html as well as here :
http://www.postgresql.org/docs/8.2/interactive/maintenance.html

Equivalent index reconstruction and table cleanup operations may exist on other databases, and are strongly recommended if they improve performance. Here an experience database administration (DBA) is highly recommended to make sure the database is always optimally configured.

36) Update Jahia database indexes

In order to update the indexes when deploying a new Jahia build, you must first delete all the standard indexes from the database, and then execute the script jahia-schema-index.sql that can be found in the directory :
tomcat\webapps\jahia\WEB-INF\var\db\sql\schema\DATABASE_TYPE
where DATABASE_TYPE is the type of database you are using.

OTHER USEFUL TOOLS

In this section we present tools that help integrators diagnostic possible errors and performance bottlenecks.

- YourKit Profiler (http://www.yourkit.com/) : a powerful JEE and Java profiler, with low overhead. YourKit introduces about 20-30% slowdown, but it is still acceptable to run in production environment. It allows to diagnose both CPU and memory usage, as well as analyze JEE traffic to the database, servlets, etc. It is a very good tool to get an idea of where the problem might come from. An alternative to YourKit is JProfiler (http://www.ej-technologies.com/products/jprofiler/overview.html), is generally more precise, but has more overhead and therefore is not suited for production analysis.

- JAMon, already integrated with Jahia, is a great tool to monitor performance on production servers. You can find more information about JAMon here : http://jamonapi.sourceforge.net/. This tool enables you to see in real time the performance of the internal Jahia services and figure out which back-end system is causing any bottlenecks. This might be a good way to for example illustrate a performance problem of a remote LDAP server, or the database server that might not be answering requests fast enough. It should be noted that JAMon can be activated "on-the-fly" while the Jahia server is running, without requiring a restart, and that it has about a 10-20% impact on performance while monitoring is active. So it is an ideal way to start examining internal performance on production system without the need to install any complex tool.

- SAP Memory Analyzer (https://www.sdn.sap.com/irj/sdn/wiki?path=/display/Java/Java+Memory+Analysis&) is a great tool to analyze JVM memory dumps, especially to diagnose OutOfMemory exceptions. This tool will give a detailed inside view of all the objects in memory, as well as their parents. Information on generating heap dumps can be found here : https://www.sdn.sap.com/irj/sdn/wiki?path=/pages/viewpage.action&pageId=33456

- MySQL Advisors (http://www.mysql.com/products/enterprise/advisors.html) : If you are deploying using MySQL, the advisors are a part of MySQL Enterprise version that allows to fine-tune your MySQL installation to your custom usage of the database.

- Java 6 Monitoring tools (http://java.sun.com/javase/6/docs/technotes/guides/management/index.html) : Java 6 introduces new monitoring tools such as JConsole, which are extremely useful to monitor basic system performance until real load scenarios.

- BEA JRockit JVM (http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic/jrockit/) : a solid alternative to Sun's JVM is BEA's JVM, which provides good performance and solid monitoring tools.

- Jahia Doctor (http://cvspub.jahia.org/cgi-bin/cvsweb.cgi/jahiadoctor/): The Jahia team developed a convenient utility to carry out integrity tests on your Jahia database. This utility attempts to automatically fix certain problems. Warning: This tool is a low-level development tool. Please use only against a fully backed-up copy of your database and, if automatic-fixes are enabled, please make in-depth tests on the resulting database before migrating it back into production.

- SQL Profiler (http://sqlprofiler.jahia.org/) : the Jahia team has also developed a small utility to automatically regroup and identify the most frequently used SQL queries and, if necessary, to automatically generate new database indexes. This tool may be useful if you have developed new SQL-intensive Jahia features. It should be noted that YourKit is now also able to profile SQL requests and may be more easy to use than this tool.

Finally if you still cannot determine the source of your problem, please contact a Jahia expert. You could first submit your problem to the free Jahia mailing lists. Please do not forget to give as much information as possible (OS used, Jahia build used, database used; stack trace of the exception (if any), etc…) so that the community can help you identify and solve the issue. If this is not possible by email, you can buy some commercial support tickets (www.jahia.com/support). Jahia experts will then spend some time re-installing your full Jahia environment and carry out extensive performance tests and diagnostics.