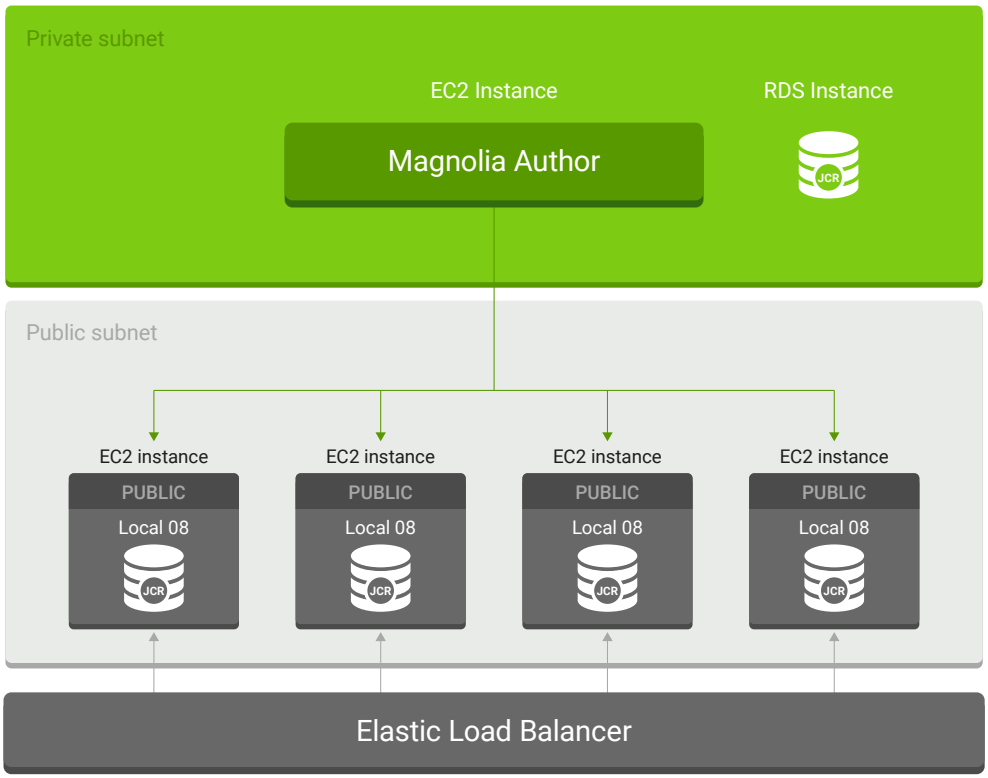
WHITE PAPER

magnolia®

Architectural Blueprints: Moving your content management into the cloud

Contents

Architectural Blueprint 1: Standard Magnolia deployment



The first of the four blueprints, looks at the general principles of deploying a CMS into the cloud environment. While using AWS and Magnolia as a case study, the lessons learnt can be applied to other enterprise-level CMS deployments in the cloud.

This blueprint shows the typical way of deploying Magnolia, whether in AWS or in another cloud service.

The blueprint shows the standard Magnolia architecture: Magnolia public instances are registered with Magnolia author instance. The Magnolia author instance uses transactional publication to push new content from the author to its public instances.

AWS services used

- **AWS Cloudwatch events** (for autoscaling notifications)
- **AWS Lambda functions** (for coordination and content synchronization)
- **AWS SNS notifications** (optionally for invoking Lambda functions)
- **AWS SSM agent** (for executing commands on remote EC2 instances, optionally for executing backups and restoring backups)
- **AWS S3** (for storing backups)

Advantages

Minimizes the number of Magnolia servers needed

No standby Magnolia instances (and licenses) are used.

No major dependencies on AWS services

While some kind of orchestration capability is needed to bring up a new Magnolia public instance, the architecture does not use many AWS services to handle autoscaling scenarios.

Disadvantages

May not be suitable for scaling for load

If a new Magnolia public instance is being started to handle increased demand, making a backup on an existing Magnolia instance will impose a certain load on the public instance being backed up. If the load on the public instance is high, running a backup on the instance may cause it to become unresponsive and AWS autoscaling may launch another EC2 instance to take its place, thus causing a cascading series of failures.

AWS limits during autoscaling

Long-running AWS Lambda functions may run into the maximum execution time limit for Lambda functions: 300 seconds. Backing up and restoring a large JCR repository can take some time depending on how large the JCR repository is. Waiting for Magnolia to start may take some time, depending on what AWS machine type is used, depending on whether the database used by the JCR repository is a local database or an RDS database. All that time to perform possibly long-running tasks could exceed the maximum execution time for Lambda functions.

Backing up Magnolia from the command line or as a scheduled job could fail

If a write-lock in the JCR repository is encountered during the backup, the backup will stop and report an error. You can run a backup with additional parameters—to retry a set number of times and wait a specified time between retries—but a backup could still fail, complicating autoscaling scenarios.

Recommendations

Autoscaling

This blueprint is recommended for small to medium-sized content repositories: synchronizing Magnolia content depends on how much content there is to synchronize and will affect how quickly a Magnolia instance will be available in an autoscaling scenario.

Recommended for five Magnolia public instances or less

More than five public instances subscribed to a single Magnolia author instance may suffer from slow publishing and performance load affecting Admin Central on the Magnolia author.

Achieving autoscaling on AWS

When bringing up a new Magnolia public instance during autoscaling, the new public instance has to get the same content as the other public instances. There are several ways to synchronize the content on the new public instance:

- Use the Magnolia Backup module to make a backup of the JCR repository on an existing Magnolia public instance and restore the backup on the new public instance.
- Export the JCR content from an existing Magnolia public instance and import it on the new public instance.
- Copy the JCR content from the database and file system used by an existing Magnolia instance to the database and file system of the new public instance.
- Use the Magnolia Synchronization module to synchronize all published content from the Magnolia author instance to the new public instance.
- Restore most of the content using a previously made backup and use the Magnolia Synchronization module to restore any content changed since the backup.

Each of the above techniques have pluses and minuses:

Backup and restore using the Magnolia Backup module

Plus

- Copies and restores both JCR content stored on the file system and in a database.
- JCR repository configuration of the backup target can be different from the JCR repository configuration of the new public instance; the Backup module can handle different JCR repository configurations.
- Backup and restore can be done while Magnolia is not running.

Minus

- All JCR content is backed up or restored; can't backup and restore individual JCR workspaces.
- Backup may fail if simultaneously reading or writing to the JCR repository; however, backups may be set up to automatically retry on failure.

Export/Import of JCR content

Plus

- Can export and import individual JCR workspaces.
- Can handle differing JCR repository configuration between the backup target and the new public instance.

Minus

- Magnolia must be running when exporting JCR content or when importing JCR content.
- Content is copied exactly, including IDs of each copied item.

Copying JCR content from database and file system

Plus

- Faster than other methods (backup/restore and import/export).
- Can be done when Magnolia is not running.

Minus

- JCR repository configuration must be identical between Magnolia public instances.
- Is a two-step process that must handle failure at either step (database copy fails, file system copy fails).

Synchronize JCR content from author instance to new public instance

Plus

- Synchronization can be targeted to selected JCR workspaces.

Minus

- Magnolia author has to send all content to the new public instance, possibly a significant performance load.

Synchronize JCR content with backup and synchronization

Plus

- Content synchronization can be split between restoring a backup and using synchronization of content changes made since the backup.
- Minimizes amount of time spent synchronizing content during Lambda function execution.
- Minimizes load on Magnolia author when synchronizing content.

Minus

- Is a two-step process that must handle failure at either step (backup restore fails, synchronization fails).

RECOMMENDED

Autoscaling recipe using backup and synchronization

In this recipe, the step of making a backup (and the time to make a backup) is avoided. The most recent backup is used and any content changed since the backup is synchronized using the Magnolia Synchronization module.

Step 1

AWS Lambda function restores most recent backup on new public instance:

- Select most recent backup available.
- Restore selected backup on new public instance.
- Register new public instance as a subscriber on Magnolia author instance.
- Launch Magnolia on new public instance.

Step 2

New public instance synchronizes modified content with Magnolia author:

- When Magnolia is running and ready, request the Magnolia author to synchronize selected content with the new public instance.

Step 3

Once the new Magnolia public has been synchronized, add it to the load balancer.

Note: the backup selection and restore in Step 1 could be performed by a user data script when the new public instance is started up.

Basic autoscaling recipe using backup/restore

In this recipe, most of the coordination and work is done by a Lambda function. The Lambda function receives autoscaling events from an AWS SNS queue.

The function app is responsible for making and restoring the backup on the new Magnolia public instance, registering new public instances with the Magnolia author instance and ensuring that no publications are done on the author instance while the new public instance is starting up.

Step 1

AWS Lambda function sets up the new public instance:

- Select an existing public instance for backing up.
- Start publication freeze on the Magnolia author.
- Make a backup on the selected public instance and upload it to a S3 bucket.
- Download designated new backup from S3 on the new public instance.
- Restore new backup on the new public instance.
- Register new public instance as a subscriber on Magnolia author instance.
- Stop publication freeze on the Magnolia author.

Step 2

New public instance starts Magnolia:

- Launch Magnolia.

Step 3

Once the new Magnolia public instance is up and running, add it to the load balancer.

The final step of adding the new public instance could be triggered by either an AWS Lambda function (triggered by posting a notification to an SNS topic).

Variation: autoscaling recipe using export/import

This recipe uses export/import of JCR content to synchronize content to a new Magnolia public instance.

Step 1

AWS Lambda function sets up the new public instance:

- Select an existing public instance for export.
- Start publication freeze on the Magnolia author.
- Export content from selected workspaces and upload resulting export files to a S3 bucket.

Step 2

New public instance starts Magnolia:

- Launch Magnolia.
- Download export files from S3 on the new public instance.
- Import export files into Magnolia.

Step 3

- Register new public instance as a subscriber on Magnolia author instance.
- Stop publication freeze on the Magnolia author.
- Add new public instance to the load balancer.

The final step of adding the new public instance could be triggered by either an AWS Lambda function (triggered by posting a notification to an SNS topic) or the new instance itself.

Step 4

AWS Lambda function restores most recent backup on new public instance:

- Launch Magnolia on new public instance.

Step 5

New public instance synchronizes modified content with Magnolia author:

- When Magnolia is running and ready, request the Magnolia author to synchronize selected content with the new public instance.

Once the new Magnolia public has been synchronized, add it to the load balancer.

Step 6

While this scenario seems to be the simplest, relying on synchronization alone puts the most load on the Magnolia author instance and on the new Magnolia public instance. It also makes the step of adding the new Magnolia public instance to the public load balancer more difficult: Magnolia must be up and running on the new instance and the synchronization between the Magnolia author instance and public must be finished before the instance is ready to be added to the load balancer.

Tooling

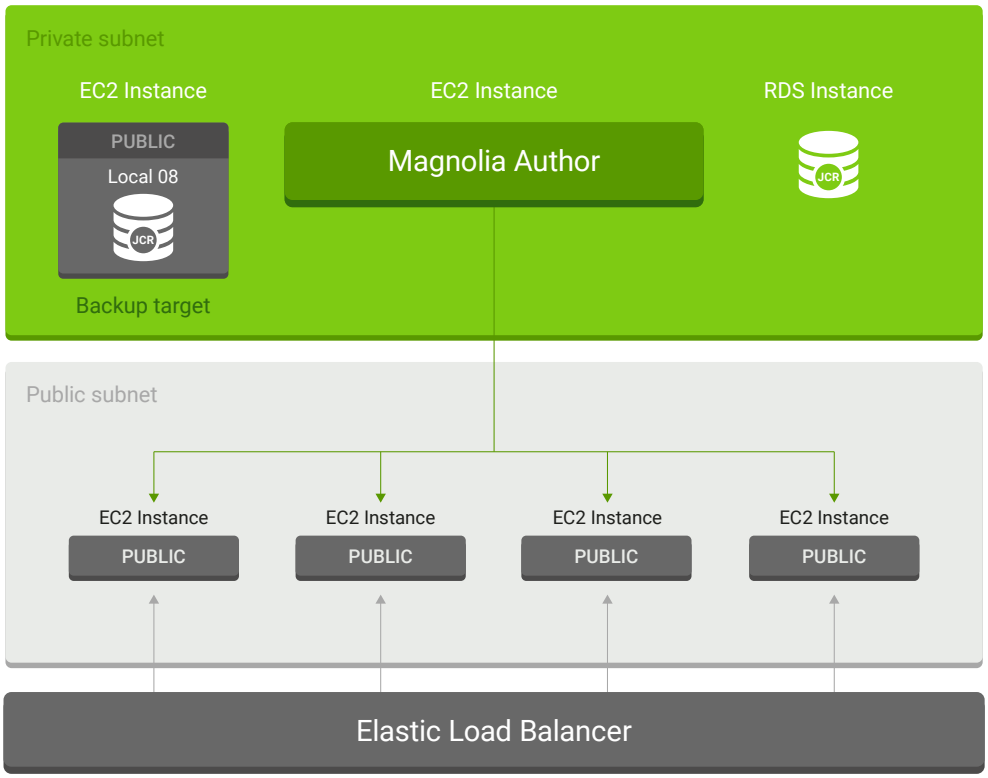
The following Magnolia features and extensions will help in building this blueprint:

- **Magnolia Backup module:** back up and restore a JCR repository
- **Magnolia Synchronization module:** synchronize content between a Magnolia author and public instance
- **Magnolia Services auto-license module:** installs your Magnolia license and avoids the license prompt on first start-up
- **Magnolia Services subscription tools modules:** create and delete Magnolia subscribers
- **Magnolia Services publication freeze modules:** allow and disallow publication on a Magnolia author

More about Magnolia modules:
https://documentation.magnolia-cms.com/display/DOCS56/Modules

More about Magnolia Extensions:
https://wiki.magnolia-cms.com/display/EX/Magnolia+Extensions

Architectural Blueprint 2: Magnolia + content source target



The second of the four blueprints, looks at the general principles of deploying a CMS with content source target. While using AWS and Magnolia as a case study, the lessons learnt can be applied to other enterprise-level CMS deployments in the cloud.

This blueprint is a twist on the standard Magnolia deployment blueprint. In that blueprint, we recommend using the Magnolia Backup module or the Magnolia Synchronization module separately or in combination to synchronize the content on a new Magnolia public instance. Content synchronization of a new public instance is an important step in your autoscaling recipe.

Running a backup using the Magnolia Backup module on a live Magnolia instance has a risk: the backup may fail. To produce a consistent backup, the Backup will stop if it encounters a lock in the JCR repository.

You can run a backup to retry if a lock is encountered (see the retryTimeout and maxRetries parameters to the backup command here: https://documentation.magnolia-cms.com/ display/DOCS56/Backup+module#Backupmodule-Backupcommand) but you can avoid or minimize the chance a backup will fail by using this blueprint.

There is another consideration when running a backup on a live Magnolia public instance: making a backup on a live Magnolia instance adds some load, perhaps at an inopportune moment if a new Magnolia is being launched to meet increased demand.

The content source target—a Magnolia public instance, registered as a subscriber with the Magnolia author instance—is not publicly accessible and is not serving content to requests outside of the private subnet where it resides. The content source target is not in the autoscaling group controlling the public instances in the load balancer.

This blueprint avoids the problems with making a backup on a Magnolia public that is also serving content. Having a content source Magnolia instance gives you two options:

- Shutting down the content source instance when backing up to guarantee a successful backup
- Running a backup on an unloaded content source

A standby Magnolia instance—the content source above—is not serving content, thus increasing the chance that a backup will not fail because the instance is busy, and will not cause a cascading failure by backing up a Magnolia public instance that is itself a member of the same autoscaling group.

The autoscaling recipes are much the same, slightly simplified as there is only the content source target used for making backups.

AWS services used

- **AWS Cloudwatch events** (for autoscaling notifications)
- **AWS Lambda functions** (for coordination and backup)
- **AWS SNS notifications** (for invoking Lambda functions)
- **AWS SSM agent** (for executing commands on remote EC2 instances, executing backups and restoring backups)
- **AWS S3** (for storing backups)

Advantages

Reduced time to synchronize content of a new Magnolia instance

Separation of concerns

Decreases or avoids the chance that a backup will fail on the backup target, causing a failure during autoscaling. Backup of content is a separate task from starting up a new Magnolia instance.

Robustness

Prevents a cascading failure since the backup target is outside the autoscaling group for Magnolia public instances. Magnolia public instances serving content do not have to be set up to make backups.

Disadvantages

A single point of failure for autoscaling

If the backup target is not available for any reason, an autoscaling recipe that relies on making a backup on the backup target would fail or have to handle that contingency (say, by using synchronization alone to initialize a new Magnolia public instance).

AWS limits during autoscaling

Restoring a large JCR repository can take some time, depending on how large the JCR repository is. Restoring the content repository from a backup may exceed the maximum execution time for a AWS Lambda function if the content repository is too large.

Recommendations

Autoscaling

This blueprint is recommended for small to medium-sized content repositories: synchronizing Magnolia content depends on how much content there is to synchronize and will affect how quickly a Magnolia instance will be available in an autoscaling scenario.

Recommended for five Magnolia public instances or less

More than five public instances subscribed to a single Magnolia author instance may suffer from slow publishing and performance load affecting Admin Central on the Magnolia author.

RECOMMENDED

Autoscaling recipe using backup and synchronization

In this recipe, the step of making a backup (and the time to make a backup) is avoided. The most recent backup is used and any content changed since the backup is synchronized using the Magnolia Synchronization module.

This recipe assumes the content source Magnolia instance has been set up to make regularly scheduled backups, either as scheduled tasks within Magnolia or as backups triggered outside of Magnolia, for example as a cron job or a scheduled AWS Lambda execution.

Step 1

AWS Lambda function restores most recent backup on new public instance:

- Select most recent backup available (from a S3 bucket or shared volume).
- Restore selected backup on new public instance.
- Register new public instance as a subscriber on Magnolia author instance.
- Launch Magnolia on new public instance.

Step 2

New public instance synchronizesmodified content with Magnolia author:

- When Magnolia is running and ready, request the Magnolia author to synchronize selected content with the new public instance.

Step 3

Once the new Magnolia public has been synchronized, add it to the load balancer.

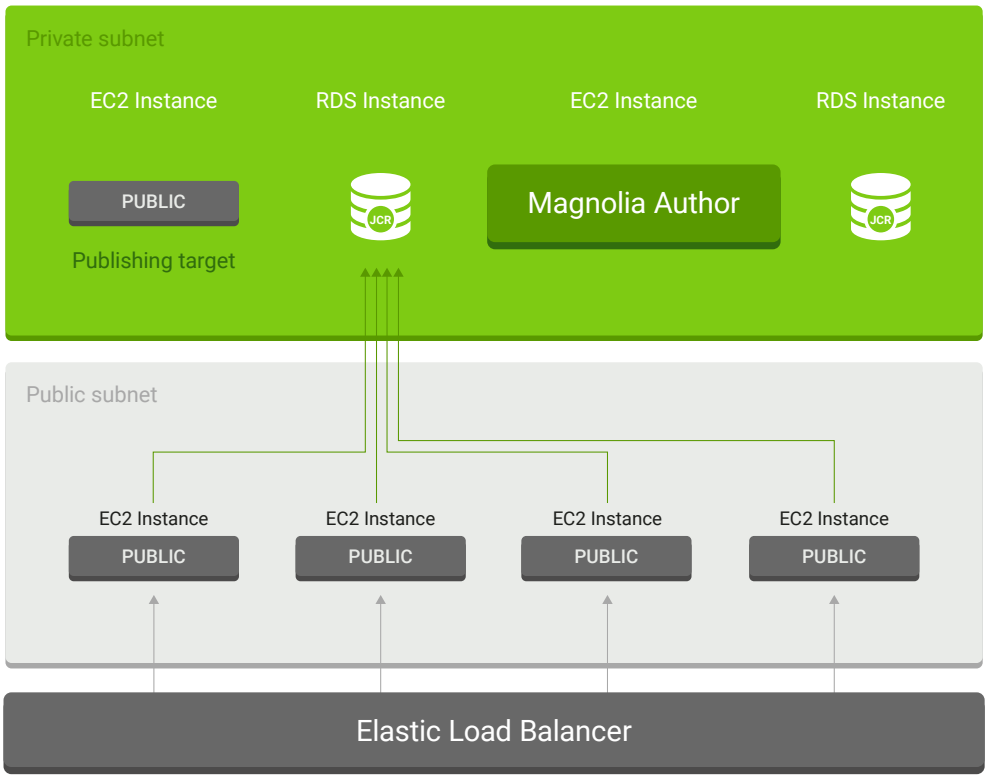
Note: the backup selection and restore in Step 1 could be performed by a user data script when the new public instance is started up instead of a Lambda function.

Tooling

The following Magnolia features and extensions will help in building this blueprint:

- **Magnolia Backup module:** back up and restore a JCR repository
- **Magnolia Synchronization module:** synchronize content between a Magnolia author and public instance
- **Magnolia Services auto-license module:** installs your Magnolia license and avoids the license prompt on first start-up
- **Magnolia Services subscription tools modules:** create and delete Magnolia subscribers
- **Magnolia Services publication freeze modules:** allow and disallow publication on a Magnolia author

Architectural Blueprint 3: Magnolia with JCR clustering



The third of the four blueprints, looks at the general principles of deploying a CMS with JCR clustering. While using AWS and Magnolia as a case study, the lessons learnt can be applied to other enterprise-level CMS deployments in the cloud.

This blueprint uses JCR clustering to avoid the tricky business of synchronizing the content on a new Magnolia public instance: sharing a clustered JCR repository among Magnolia instances means all content is already there in the shared JCR repository.

A Magnolia deployment using JCR clustering has a single “slave” Magnolia public instance that receives all publications from the Magnolia author and is responsible for updating the JCR repository. All other Magnolia public instances share the JCR repository maintained by the slave public instance.

JCR clustering caveats

JCR clustering doesn't work well in every situation. You should use JCR clustering only if:

- The database hosting the JCR repository is running on a separate VM
- The amount of publications from the Magnolia author to the public instance are infrequent
- Any autoscaling of public instances (creating new public instances or shutting running public instances) must be done with care
- You do not need to update Magnolia too frequently

The database hosting the JCR repository is running on a separate VM.

The database for the shared JCR repository should not run on the VM running a Magnolia public instance; the database may be lost if the VM is shut down or destroyed.

The shared JCR repository should run on an AWS SQL database - Magnolia supports a variety of SQL databases - including provisions for backing up and maintaining the database.

The amount of publications from the Magnolia author to the public instance are infrequent.

Every change to JCR nodes creates entries in a journal maintained by JCR repository. The journal must be periodically cleaned or the JCR repository may slow down or even become unresponsive affecting all Magnolia public instances sharing the repository.

Any autoscaling of public instances (creating new public instances or shutting running public instances) must be done with care.

The application container running a Magnolia public instance must be shut down gracefully; the JCR repository updates its underlying database during shutdown. If interrupted during shutdown (or killed), the JCR repository may be corrupted and other running Magnolia instances sharing the JCR will be affected.

You do not need to update Magnolia too frequently

Updating Magnolia usually entails writing changes to the JCR repository, module versions and updates to Magnolia configuration.

Getting the order right is important: changes to Magnolia configuration should only be made once, so all Magnolia public instances sharing the JCR repository can't be updated individually, this makes the deployment of a new Magnolia version complicated and the update steps must be done in correct order:

1. Shut down all Magnolia public instances.
2. Update the Magnolia WAR on the publishing target and launch the publishing target.
3. Wait for the publishing target to make any changes and updates.
4. When the publishing is ready, launch the slave Magnolia public instances sharing the JCR repository. (Note that the public instances will not update or install any changes, the update of the Magnolia publishing target in charge of the shared JCR repository has already taken care of that.)

This complex orchestration prevents incremental updates of individual Magnolia public instances sharing a JCR repository. You can still achieve high availability with a blue-green deployment with two separate clusters of Magnolia public instances each with a separate clustered JCR repository.

Advantages

There are other advantages when using JCR clustering:

- Registering a new Magnolia public instance as a subscriber on the Magnolia author instance is unnecessary: only the slaved public instance is registered as a receiver. Other Magnolia instances sharing the JCR repository are not (and shouldn't be) registered as receivers.
- The time to publish content from the Magnolia author is constant: only the slave Magnolia public instance is the recipient of publications; all other public instances do not receive publications.

Disadvantages

However, there are significant downsides to JCR clustering:

- The shared JCR repository is a single point of failure. If a single Magnolia instance corrupts the JCR repository, all other Magnolia instances sharing the repository will be affected.
- Any problem with the clustered JCR repository will affect all the Magnolia instances sharing the repository. JCR repositories can be corrupted—for example, by improperly suiting down a Magnolia instance—and can affect other Magnolia instances

- JCR clustering is not scalable to large numbers (more than six to ten) Magnolia public instances.
- There may be performance effects as more and more Magnolia instances are added: more Magnolia instances means more database connections are added. There may be increased competition between Magnolia instances for access to the JCR repository.
- Startup of a new Magnolia instance will be slower than a Magnolia instance using a local database for its non-clustered JCR repository.
- The JCR repository should be hosted on an AWS SQL database that is accessible to all the Magnolia instances using the clustered JCR repository. The more Magnolia instances sharing the JCR repository, the bigger the virtual machine needed to run your SQL database.
- The performance of the database hosting a clustered JCR repository will affect the performance of all Magnolia instances sharing the JCR repository.
- JCR clustering goes against the strategy of making your Magnolia public instances disposable: while a Magnolia instance itself is disposable, the JCR repository is not. Care must be taken when tearing down a Magnolia instance using a clustered JCR repository; improperly shutting down Magnolia can corrupt the JCR repository.

And last, but far from least: you will need a recovery plan or procedure for a clustered JCR repository if it becomes corrupted. Magnolia provides tools that can be used in this scenario, such as the Magnolia Backup module and the Magnolia Synchronization module.

Recommendations

Use an appropriately sized RDS instance to host the shared JCR repository.

Minimize the use of file storage for JCR workspaces. Locking in Jackrabbit for file
storage is broader and slower than locking for a JCR repository stored in a database.

Recommended for large content repositories

No content is synchronized when starting up a new Magnolia public instance.

Recommended for autoscaling scenarios where fast startup of a Magnolia instance is critical

Since no content synchronization is done on startup, this is the quickest way to get a Magnolia instance up and running.

Recommended for five Magnolia public instances or less

More than five public instances may overburden the database used by the JCR repository. There is a theoretical and practical limit to how many Magnolia public instances can share a clustered JCR repository. Each Magnolia instance will use database connections and the database hosting the clustered JCR repository will be unable to efficiently serve all connections.

Recommended for applications needing less responsiveness

A Magnolia public instance sharing a JCR repository hosted on a remote RDS database will be slower than a Magnolia public instance using a private JCR repository hosted on a database running on the local machine. Scaling up to a large RDS instance will not necessarily affect transmission times of JCR data to and from the database.

AWS services used

- **AWS Cloudwatch events** (for autoscaling notifications)
- **AWS Lambda functions** (for coordination and backup)
- **AWS SNS notifications** (for invoking Lambda functions)
- **AWS SSM agent** (for executing commands on remote EC2 instances, executing backups and restoring backups)
- **AWS EFS** (shared journaling for JCR clustering)

Autoscaling recipe when using a clustered JCR repository

Unlike previous recipes, this recipe contains no content synchronization step because it assumes there is a clustered JCR repository already available.

JCR clustering in Magnolia is set up through the Jackrabbit configuration used by Magnolia.

There are several key pieces of configuration set in the Jackrabbit configuration:

- The JDBC connection details for connecting to a shared database
- A unique cluster ID (unique among Magnolia instances sharing the JCR cluster)
- A shared file system for cluster journals

The Jackrabbit configuration is usually part of your deployable Magnolia WAR file, but can contain configuration specific to the Magnolia instance being started. There are several ways to go about this:

- Modify the Jackrabbit configuration for the new public instance, build a new WAR and deploy **(not recommended)**
- Modify the Jackrabbit configuration within your WAR and deploy it **(not recommended)**
- Set Jackrabbit configuration through Java system properties on the Magnolia instance **(recommended)**

Property placeholders in the Jackrabbit configuration file will be replaced with their system property values when Magnolia is starting up. You can use this to include a generic Jackrabbit configuration in your Magnolia WAR that will take their values on starting up.

For example, when using JCR clustering, your Jackrabbit configuration must define a Cluster element:

```
<Cluster id="${magnolia.cluster.id}" syncDelay="${magnolia.cluster.syncDelay}">
        <Journal class="org.apache.jackrabbit.core.journal.DatabaseJournal">
                <param name="revision" value="${magnolia.cluster.revisionHome}/revision.log" />
                <param name="driver" value="com.mysql.jdbc.Driver" />
                <param name="url" value="${magnolia.jdbc.url}" />
                <param name="user" value="${magnolia.jdbc.user}" />
                <param name="password" value="${magnolia.jdbc.password}" />
                <param name="schema" value="mysql" />
                <param name="schemaObjectPrefix" value="journal_" />
        </Journal>
</Cluster>
```

The cluster definition uses the following system properties to customize the Magnolia installation:

- **`magnolia.cluster.id`**, a unique ID for the Magnolia instance
- **`magnolia.cluster.syncDelay`**, the synchronization delay for the Magnolia instance
- **`magnolia.cluster.revisionHome`**, the shared home directory for the revision log
- **`magnolia.jdbc.url`**, the JDBC URL for the database
- **`magnolia.jdbc.user`**, the database user
- **`magnolia.jdbc.password`**, the database user's password

The values for all these properties may change depending on your Magnolia deployment.

If you have installed Tomcat as Linux service, youcan also modify the service configuration and add system property definitions that will be used by Magnolia. As with CATALINA_OPTS, your service definition can define system properties for the JVM running Magnolia; Magnolia will substitute the values of system properties used in Magnolia configuration files.

Autoscaling recipe with JCR clustering

The autoscaling recipe when using JCR clustering is simpler than other blueprints because no content synchronization and publication freeze and thaw is needed

Step 1

AWS Lambda function sets up the new public instance:

- Define system properties for Jackrabbit configuration (including generating a unique cluster ID for the new instance) and save these properties on the new instance.
- Mount shared file systems for journals defined for the shared JCR clustering.
- Launch Magnolia.

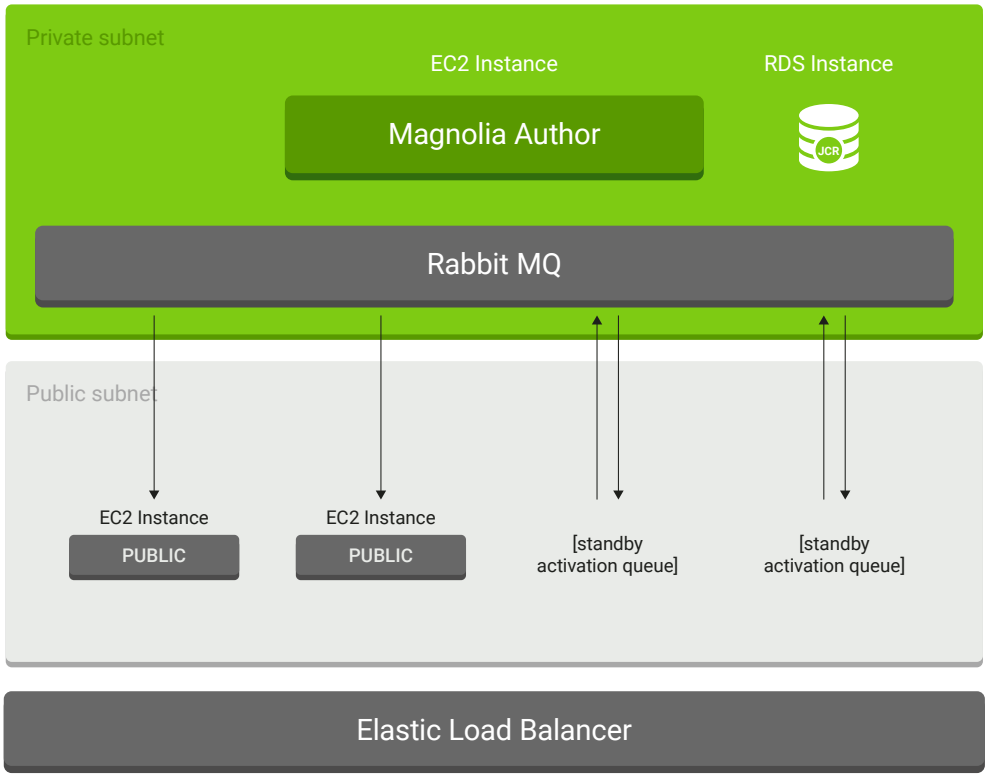
Step 2

Once the new Magnolia public instance is up and running, add it to the load balancer.

Tooling

No special tooling or extensions are needed to use JCR clustering with Magnolia. However, you should take precautionary measures like regularly backing up your JCR repository (we recommend using the Magnolia Backup module) to protect your JCR repository.

Architectural Blueprint 4: Magnolia with RabbitMQ publishing



The last of the four blueprints looks at the general principles of deploying a CMS with RabbitMQ. While using AWS and Magnolia as a case study, the lessons learnt can be applied to other enterprise-level CMS deployments in the cloud.

You may have noticed that there are certain common problems in the architectural blueprints we have presented: content synchronization on a new Magnolia instance and registering a subscription for a new Magnolia public instance, for example.

Publishing content in Magnolia uses “transactional activation”. The Magnolia author instance ensures that all subscribed public instances receive and save the published content. If one of the public subscribers fails to publish the content, the content publication is rolled back across all public subscribers.Transactional publication guarantees that all public subscribers are kept in sync. Transactional publication comes at a cost: the time to publish content is proportional to the number of public subscribers; with more public instances, more time will be taken to ensure a successful publication.

Publishing content with RabbitMQ publication is an alternative to transactional publication. It uses RabbitMQ, an open-source messaging broker, to deliver publication messages from the Magnolia author to Magnolia public subscribers simultaneously.

RabbitMQ allows a looser coupling between the Magnolia author and public instances: publication with RabbitMQ publication will not be transactional, but the time to publish content will not depend on how many public instances are running since publication will be done in parallel. With RabbitMQ publishing, public instances could get out of sync, but also makes it easier to start and stop Magnolia public instances if there are problems publishing content.

RabbitMQ publishing allows many Magnolia public instances to be connected to a single Magnolia author without increasing the time to publish content.

There are other benefits to using RabbitMQ publication. Since the Magnolia author and public instances are decoupled, there is no need for a publication freeze on the Magnolia author to prevent publications while a new Magnolia public is starting. There is no need to register the new public instance with the Magnolia author either; the public instance is registered with RabbitMQ, not the Magnolia author.

To provide a robust, scalable delivery mechanism, RabbitMQ can be set up to provide high availability queues with federation and clustering to ensure that the Magnolia author can always send publication messages for distribution to Magnolia public instances.

Content synchronization and RabbitMQ activation

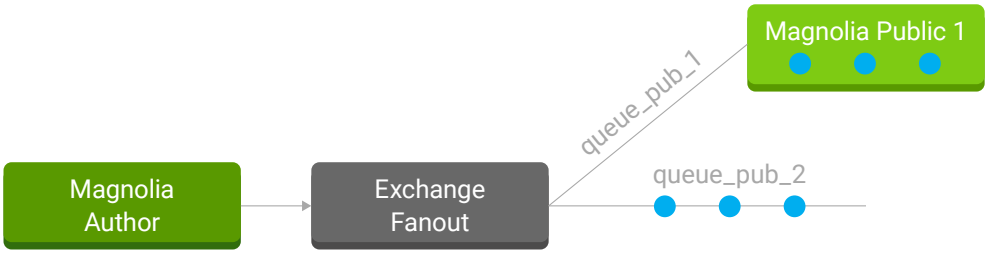
In other blueprints, we recommended a hybrid approach to content synchronization: use the Magnolia Backup module to restore most of your JCR content, followed up by the Magnolia Synchronization module to synchronize any content changed since the last backup.

You can still use the Synchronization module with RabbitMQ activation, but RabbitMQ activation gives you another way to update recently changed content.

RabbitMQ is a message broker, distributing messages to clients through queues. Messages stay in RabbitMQ queues until they are delivered to clients. RabbitMQ queues that don’t have a client will save messages until they are delivered to a client.

RabbitMQ activation uses RabbitMQ to deliver activation messages from a Magnolia author to Magnolia public instances. Each Magnolia public instance is attached to a queue, waiting for activation messages. With RabbitMQ, you can create “standby” activation queues without a Magnolia public instance client. Activation messages will be saved in the standby queue until a public instance claims the standby queue and RabbitMQ will deliver the activation messages to the public instance.

Activation messages are relatively small and RabbitMQ is capable of managing many queues and messages; standby activation queues could store many hundreds or thousands of messages in a RabbitMQ broker using minimal system resources.



Content synchronization coordination

There is some practical limit to how many messages can be stored in RabbitMQ queues. At some point, the standby activation queues must be flushed.

Standby activation queues can be used instead of the Magnolia Synchronization module. You will need to coordinate the backup with the standby activation queues.

When a backup is made, the standby activation queues should be flushed, since any publications waiting in the standby queues will be contained in the backup.

There are two ways you can make a scheduled backup of a Magnolia instance:
• By setting up a scheduled job within Magnolia (note that Magnolia must be running to make the backup)
• By setting an AWS Lambda function to run at a scheduled time (note that Magnolia doesn’t have to be running when the backup is made)

Setting up a scheduled job in Magnolia to launch a backup is easy; see the Magnolia Scheduler module: https://documentation.magnolia-cms.com/ display/DOCS56/Scheduler+module

A scheduled AWS Lambda function could launch a Magnolia backup via the Magnolia REST API and flush the RabbitMQ standby activation queues. The coordination between backup and flushing queues doesn't even have to be precise: with RabbitMQ activation, a Magnolia public instance will discard any activation messages it has already received.

Handling out-of-sync public instances

When using transactional activation, publishing content can fail, but Magnolia guarantees all the public instances will be in sync. Transactional activation can fail for many different reasons: a subscribed Magnolia instance might not be actually running or is unable to process the publication within a set time; the Magnolia instance's JCR repository is corrupted; there is a time difference between the Magnolia author and Magnolia public instance; and many other reasons. Transactional activation prevents the public instances from getting out sync, but may also prevent any publications until a failing Magnolia public instance is repaired or taken out of service and its subscription is deactivated. Unfortunately, transactional activation doesn't provide a convenient hook for AWS services for detecting ailing Magnolia subscribers and correcting them.

RabbitMQ activation provides a feedback channel for activations: Magnolia subscribers can report whether a publication succeeded or failed on RabbitMQ acknowledgement. That acknowledgement can be monitored by the RabbitMQ monitoring app in the Magnolia author. The monitoring app shows what activations succeeded and failed for each Magnolia public instance, how long an activation message stayed in queue and how long it took for the public instance took to process it.

The RabbitMQ monitoring app can help you see whether Magnolia instances are in sync or not, but the underlying notification mechanism—the activation acknowledgement queue—can be an integration hook for managing Magnolia public instances with AWS services.

Here's how: activation notifications from RabbitMQ activations are also stored with Magnolia in a separate JCR workspace. Those notifications note whether an activation succeeded or failed, how long it waited for delivery in RabbitMQ and how long it took the Magnolia public instance to process the activation once RabbitMQ delivered it. All this information could be used to identify out-of-sync or ailing Magnolia public instances.

There are at least two ways you could tie in activation acknowledgements to AWS services:

Within Magnolia: use the Magnolia Observation module to watch the RabbitMQ activation notification workspace. When the workspace is updated with notification that shows a problem with a Magnolia public instance (a failed activation or a slow activation, for example), post a notification on an AWS SNS topic noting a problem with the Magnolia instance. AWS Lambda functions could then do further notifications to notify Magnolia sysadmins or terminate the Magnolia instance and replace it.

Outside Magnolia: build an external client to monitor the activation acknowledgement queue and post a notification on an AWS SNS topic noting a problem with the Magnolia instance, letting other AWS services (like a Lambda function) kick in and handle the problem.

Recommendations

Recommended when running five or more Magnolia public subscribers

RabbitMQ publication allows publications to many public instances.

Recommended for high frequency of publication

Publishing is an expensive operation for the Magnolia author. Using RabbitMQ greatly reduces the load on the Magnolia author in publishing to many public instances.

Recommended for large numbers of Magnolia Admin Central users

Magnolia can support up to 30 to 50 simultaneous users of Admin Central. RabbitMQ publication reduces the performance load when those authors publish content.

AWS services used

- **AWS Cloudwatch events** (for autoscaling notifications)
- **AWS Lambda functions** (for autoscaling coordination)
- **AWS SNS notifications** (for invoking Lambda functions)
- **AWS SSM agent** (for executing commands on remote EC2 instances and restoring backups)

Autoscaling recipe with RabbitMQ activation

In this recipe, the logic of selection of an available standby activation queue resides in a Lambda function.

Step 1

AWS Lambda function sets up the new public instance:

- Select an available standby activation queue.
- Launch Magnolia.
- Wait for Magnolia to become available; configure the instance's RabbitMQ client configuration to use the selected standby activation queue.

Step 2

Once the new Magnolia public instance is up and running, add it to the load balancer.

Variation: autoscaling recipe without AWS Lambda

The logic of selecting an available standby activation queue could be included in a Magnolia module, so no Lambda function would be
needed.

The module could query RabbitMQ, find an unused standby activation queue, and update its RabbitMQ client configuration directly.

This variation avoids waiting for Magnolia to start up before changing the RabbitMQ configuration.

Step 1

On Magnolia starting up:

- Select an available standby activation queue and update its RabbitMQ client configuration directly.

Step 2

In a Lambda function handling the autoscaling notification:

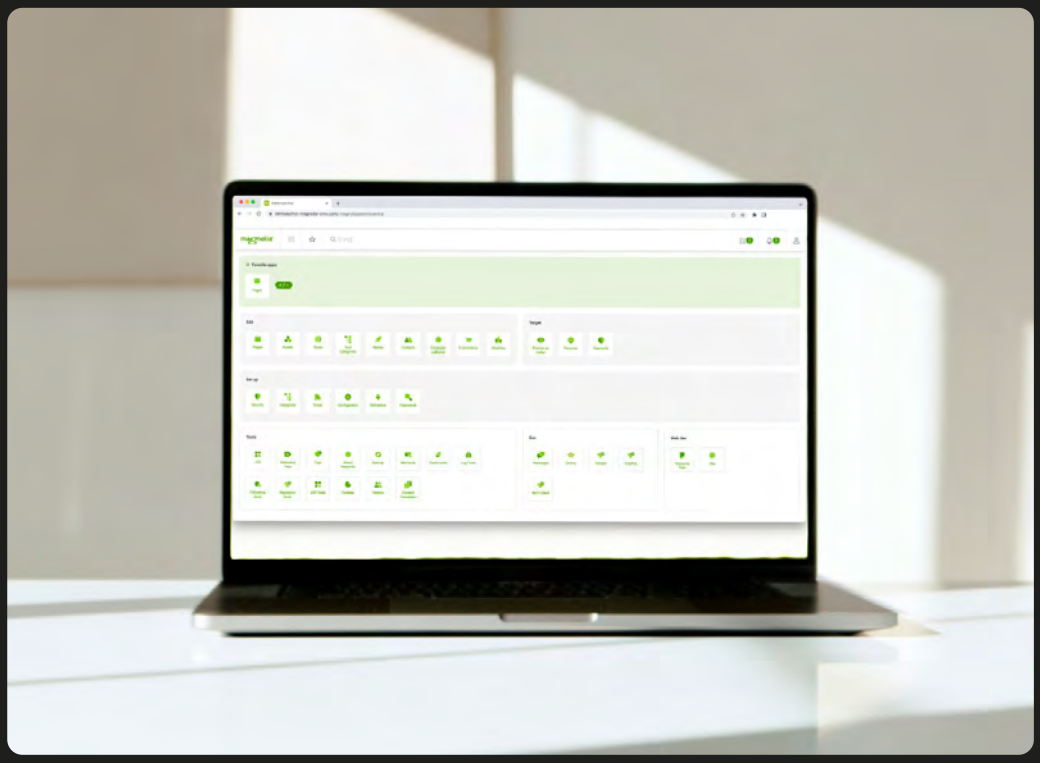
- Add the new Magnolia public instance to the load balancer.

Tooling

The following Magnolia features and extensions will help in building this blueprint:

- **Magnolia RabbitMQ modules:** publish content with RabbitMQ
- **Magnolia Property REST API:** for adjusting the RabbitMQ client configuration
- **Magnolia Node REST API:** for adjusting the RabbitMQ client configuration
- **Magnolia Backup module:** back up and restore a JCR repository
- **Magnolia Synchronization module:** synchronize content between a Magnolia author and public instance
- **Magnolia Observation module:** get notifications when the contents of a JCR workspace is changed
- **Magnolia Services auto-license module:** installs your Magnolia license and avoids the license prompt on first start-up
- **Magnolia Scheduler module:** execute commands at specified times

magnolia®



Get in touch

To learn how Magnolia can help you launch great digital experiences faster, contact us at:

Magnolia HQ Switzerland

Office +41 61 228 90 00

contact@magnolia-cms.com

Or see our offices worldwide

www.magnolia-cms.com