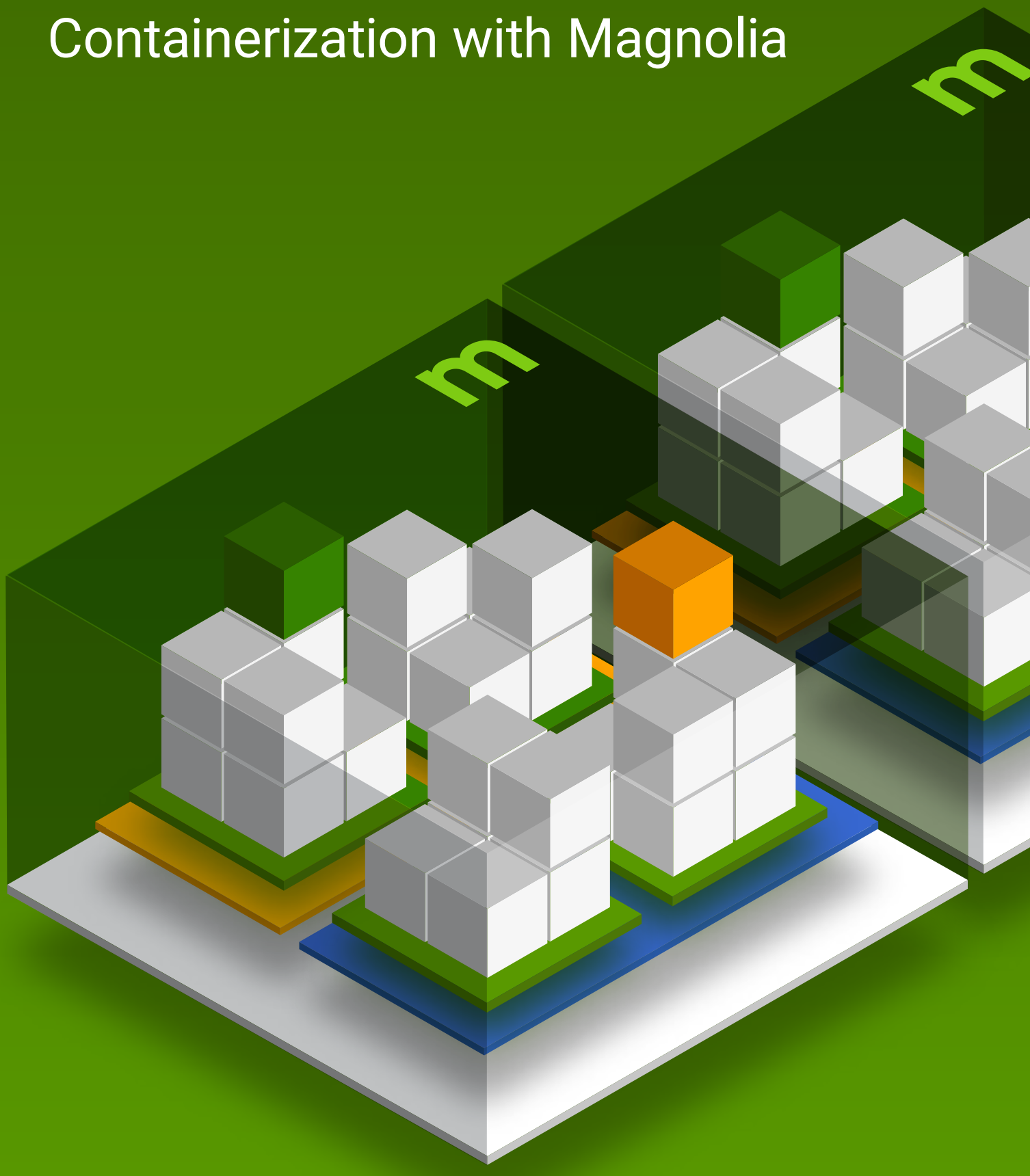
magnolia®

WHITE PAPER

Magnolia in a Can

Containerization with Magnolia



Contents

Magnolia in a Can

Scalability, standardization of deployment, operational environments and ease of management are all excellent reasons to containerize Magnolia.

Magnolia can be containerized and run in tools like Docker, but there is no “one size fits all” way of going about it. Much depends on how you would like to deploy and run Magnolia and what your CI/CD pipeline looks like.

Since there’s no “one size fits all” solution that could cover all or even most situations, we don’t provide a public Docker image for Magnolia to base your container on.

But don’t despair, we’ve outlined the possibilities and choices for containerizing Magnolia, their pros and cons and what we consider to be “best practices” in building a Docker container for Magnolia.

You’ll learn how you can containerize Magnolia to work in your environment, with your requirements and constraints, and hopefully avoid some common pitfalls when operating Magnolia.

This guide isn’t an introduction to Docker and containers, but we don’t assume you know everything about Magnolia. Features useful to containerizing and operating Magnolia are highlighted and explained here.

Image Ingredients

Magnolia needs the following to run in a Docker container:

- Java
- Tomcat or another web application container
- A relational database for the JCR repository (or an embedded database)
- Environment setup

These ingredients will go into your Docker image but can be combined in different ways, depending on your needs and preferences.

So, let's dive in and start to pick the ingredients for our Docker image.

Decision #1: Choose a Docker base image

There are several options when picking a base image:

1. Go for an image already containing one of your basic Magnolia ingredients, such as Tomcat or a relational database to build up your Magnolia image
2. Go distroless: use a Java based distroless image such as gcr.io/distroless/java:11 and add your other Magnolia container ingredients
3. Pick an OS image for your base image and add the needed Magnolia ingredients (Java, web application container and database).

Picking a Tomcat base image saves you the trouble of adding Java and Tomcat to your image and may save you some work down the road with updates to the image. If you want to store your JCR repository in a relational database, you'll have to add it into your image but you can also use an embedded database like H2 or Derby to run Magnolia.

Picking a database image gets you an underlying OS image to start, but you have to add Java and Tomcat to your image to run Magnolia.

On the other hand, picking an OS image gives you full control over what you add into your Magnolia image but also leaves you with the work of maintaining and updating what you added.

Going distroless forecloses one option for your Magnolia container: running both a separate relational database and web app container (with Magnolia inside of course) in the same Docker container: launching both a relational database and the web container with one CMD or ENTRYPOINT would mean adding more ingredients into your distroless base image, a shell, a relational database and other utilities.

RECOMMENDATION

Look over the Magnolia certified stack before picking your base image.

Pick a well-known image, one that aligns with your skill set and you are familiar with operating.

When picking a distro image, pick a distro that is part of the Magnolia certified stack.

We test Magnolia releases against a range of operating systems:

- Ubuntu - all currently supported LTS releases
- SuSE Linux Enterprise Server - all releases with existing (SuSE) general support
- Fedora - latest two releases
- Red Hat Enterprise Linux Server
  - For Magnolia 5.x: All releases with full support or maintenance support
  - For Magnolia 6.x: RHEL 7 (and later) with full support or maintenance support
- CentOS 6 and 7
- Debian - all currently supported LTS releases
- Windows Server 2012 R2
- Windows 2019 Standard or Datacenter
- Windows 10

Magnolia may run on distro images that are not in the certified stack but the additional testing that goes into the certified stack may help you avoid problems down the road.

GOTCHA!

Alpine, a popular compact image, won't run Magnolia out of the box. The system libraries are incompatible with some of the Java libraries used in Magnolia 6.

Alpine can be used as a base image for a Magnolia container but it takes a bit of tweaking, see the "Hacking Docker: making Magnolia work on Alpine" section below.

Decision #2: Choose a database for the JCR repository

This is a big decision as the database you choose will likely have a big impact on how you operate Magnolia in your container.

Again, it's a good idea to take a look at the Magnolia certified stack before picking your database.

Here are the databases in the Magnolia certified stack:

Embedded databases:

- H2 1.4.200 and later
- Derby 10.3.1.4 (included)

External databases:

- MySQL 5.5 and later
- Oracle 10g Enterprise Edition and later
- PostgreSQL 9 and later

The embedded databases - H2 and Derby - don't require a separate database service to be run in your container. They store their data in files in the file system, which makes it possible to copy the JCR repository for Magnolia by simply copying files to a new container (more on this later). Derby and H2 don't have the sophisticated caching that external databases do and are less performant than external relational databases with large JCR repositories.

External databases require a separate database service (obviously) and they are another thing to add into your image or a separate container and manage when operating.

Here's a summary of the pros and cons of each:

Embedded databases (H2 and Derby):
Pro: No additional database service to run Magnolia
Con: Less performant on large JCR repositories
Con: Must shut down Magnolia instance to back up JCR repository

External databases (MySQL, Oracle, PostgreSQL):
Pro: Better performance, especially on large repositories
Pro: Better tools for monitoring and management
Pro: Can use a single database service for multiple Magnolia instances
Con: Another service to containerize and manage

Decision #2.1: Determine whether to run Magnolia and its database in the same container or in separate containers

Docker's golden rule - one service per container - makes sense in many situations. It helps break down complex applications into their constituent services. It makes setting up an ENTRYPOINT or CMD in your image easier when managing only one service.

But as a whole, Magnolia is a single service. The underlying JCR repository that Magnolia needs isn't a separate thing and can't really be operated on its own even if the JCR repository is stored in a different database service.

There are different ways to resolve this quandary:

Use an embedded database - H2 or Derby - for Magnolia's JCR repository. No database service is needed, and only one container is required to run Magnolia.

The downsides: H2 and Derby may not be as performant with large JCR repositories and you won't have an extensive toolbox for monitoring and managing the database.

RECOMMENDATION

Be aware of your use case. Operating the Magnolia author instance versus the Magnolia public instance(s).

Operating a Magnolia author instance will be different from operating Magnolia public instances. The JCR repository for a Magnolia author instance is valuable: it is the master copy of all content, especially content under development and not yet published. That content should be protected to prevent its loss: the JCR repository of your Magnolia author instance should be bullet-proofed, backed up and monitored.

While the author instance will probably be on a private network, the public instances will probably be on a public network. Usually there will be a single author instance, but often there will be several, maybe many, public instances.

The JCR repository of a Magnolia public instance isn't unique: Magnolia public instances will each have a copy of all published content in their respective JCR repositories. If the repository of a Magnolia public instance running in a container becomes corrupted, you can shut it down and start a new container to replace it.

RECOMMENDATION

The database service and the web container for the Magnolia author instance should be in separate containers.

The database service and the web container for Magnolia public instances should be in the same container.

The public instances are disposable and you may want to add or remove new public instances to meet changing traffic. Managing public instances is easier if everything is in a single container, especially if you are automating the management.

One container for Magnolia public instances and two containers for author instances probably means you will need separate images for author instances and public instances. Crafting and maintaining two images instead of one is more work, but trying to build an image for both may be complex as well.

Decision #3: Choose an application server

Magnolia runs inside an application server so your image must set up and launch the server.

The certified stack offers several application servers to choose from:

- Apache Tomcat
- Wildfly
- JBoss EAP
- IBM WebSphere Application Server
- IBM WebSphere Liberty

RECOMMENDATION

Choose an application server that you are familiar with operating.

Again, it is best to pick an application server you are familiar with; if that's Tomcat (the most commonly used application server), use Tomcat. If it is another application server, use that.

Other application servers supporting Java web applications may be capable of running Magnolia, but application servers in the certified stack are tested against Magnolia releases.

Decision #4: Choose a Java version

This is probably the easiest decision of all.

Magnolia runs on Java 8, 9, 10, 11, 12 and 13. Magnolia can be run with a JDK or a JRE, though if you want to use your image in a development environment, you probably will want to use a JDK.

Magnolia runs on both Oracle and OpenJDK Java.

Your choice of application server may determine your choice of Java version. Tomcat versions may require you use a certain Java version.

RECOMMENDATION

Use Java 11 or later for better support running in a container: JVM flags like `AlwaysActAsServerClassMachine` can improve JVM performance when running Magnolia in a container.

You will probably want to run Magnolia in a container using the least amount of resources possible. JVM flags such as AlwaysActAsServerClassMachine are a good starting point for tuning garbage collection.

Containerizing Your Magnolia Web App

Now that you have made all the decisions about the ingredients for your image, it's time to containerize your Magnolia web app.

BEST PRACTICE **Make your Magnolia web application self-starting.**

By default, your Magnolia web app requires your input when starting up. You will be prompted to:

- Approve the installation of Magnolia
- Enter your Magnolia license

To start Magnolia without your intervention you can enable auto updates and configure your license.

Enable Auto Update

When Magnolia starts up for the first time or a Magnolia module has been updated, Magnolia will prompt you before proceeding with the installation or update.

You can avoid this prompt by setting a Magnolia property:

```
magnolia.update.auto=true
```

When set to true, Magnolia will proceed with installing or updating itself without prompting.

Magnolia properties are a powerful way to control and run Magnolia and come in handy when containerizing your Magnolia web application. We will discuss other Magnolia properties later on.

Configure the Magnolia License

Magnolia expects to find its license in the JCR repository. You can see your Magnolia license by opening the Configuration app and navigating to `/modules/enterprise/license`.

If Magnolia does not find its license there, Magnolia will prompt you to enter the license during start-up.

There are several options for adding your Magnolia license:

- **Bundling the license in the Magnolia web app**: create a Maven module that sets the license when it is loaded.
- **Setting the license from the container environment**: use the Configuration Injection module to add your license at runtime.
- **Building a custom Magnolia Java module**: a custom module can retrieve and set the license when Magnolia starts.

Each of the above options has pluses and minuses:

Bundling the license in the Magnolia web app

Bundling your license into your Magnolia web app puts an expiration date on your web app. When the Magnolia license expires, the Magnolia web app will start but some features will be disabled. Including the Magnolia web app in your Docker image means your Docker image has an expiration date as well.

This is typically not an issue, because chances are that you will update the Magnolia web app before the license expires, so including an updated license won't be a lot of extra work.

However, bundling the license in a module and including it in your Magnolia web app could be considered a security risk. Your license could be recovered from the Magnolia WAR file, your source code, or Docker image.

To protect your Magnolia license, make sure that your artifact and source code repositories as well as your Docker artifacts are secure.

Setting the license from the container environment

You can't set the Magnolia license through Magnolia properties out-of-the-box, but you can use the Configuration Injection module from Magnolia's Incubator. This module allows you to set the Magnolia license from a Magnolia property when Magnolia starts up. This approach allows you to set the license using a Docker ENV variable when starting your container decoupling the license from the Magnolia web app.

Setting the Magnolia license from the container environment could be considered a security risk as well, but it minimizes the number of places the Magnolia license can be recovered from. To eliminate this risk, you could use your container platforms's security features, for example by passing your license as a `Secret`.

Building a custom Magnolia Java module

Don't like either bundling the license in the Magnolia web app or setting the license from the container environment? Or do you have additional requirements, for example, retrieving your Magnolia license from a secure vault? Then building a custom Magnolia Java module may be the best option for you.

Magnolia Java modules have lots of useful tools like startup tasks and dependency management. This option gives you total freedom to implement a solution that meets your needs.

Building a custom Magnolia Java module requires some knowledge of Java, Java tools like Maven, and possibly the Content Repository API for Java (a.k.a. JCR), as well as an understanding of how Magnolia is put together.

Of the options above, this is what we recommend:

BEST PRACTICE

Set the license from the container environment using the Configuration Injection module.

The Configuration Injection module introduces the system property **`magnolia.inject.config`**. It can be used to create a startup task that sets the Magnolia license at **`/modules/enterprise/license`**:

```
magnolia.inject.config=createPath:/modules/
enterprise/license;setProperty:/modules/enterprise/
license,owner,<email for license>;setProperty:/modules/
enterprise/license,key,<your Magnolia license key>
```

For more information on the Configuration Injection module see https://wiki.magnolia-cms.com/display/SERVICES/Configuration+Injection.

If you prefer to bundle the license in the Magnolia web app you can create a Magnolia Java module that automatically loads the license on startup.

Magnolia Java modules are one of the basic ways to customize and extend Magnolia. They require some Java coding to produce a Java jar file, but in our case the amount of coding is limited. You can also find a template for a license bundle module here: https://git.magnolia-cms.com/projects/SERVICES/repos/autolicense/browse.

For more information on developing Magnolia Java modules, check the following links:

https://documentation.magnolia-cms.com/display/DOCS62/How+to+create+and+use+a+custom+Magnolia+Maven+module+for+custom+Java+components

https://documentation.magnolia-cms.com/display/DOCS62/Bootstrapping+in+Maven+modules

Magnolia properties and containerization

Magnolia properties control key aspects of Magnolia when it is running.

Above we mentioned the Magnolia property **`magnolia.update.auto`**, but there are many other Magnolia properties you may want to use in your Docker image.

Magnolia properties can control:

- Whether Magnolia runs as an author instance or a public instance
- The configuration of the JCR repositories
- Database connection
- File system locations for:
  - Resources
  - Light modules
  - Magnolia publication keys
  - Temporary files

Magnolia properties can be set in property files included in your Magnolia web application but you can override any Magnolia property by setting a Java property with the same name to a new value.

For more on Magnolia property files, see https://documentation.magnolia-cms.com/display/DOCS62/WAR+file+with+multiple+configurations#WARfilewithmultipleconfigurations-magnolia.propertiesfile.

Here's a quick tour of some of the more interesting Magnolia properties:

`magnolia.home`: the granddaddy of all Magnolia properties, **`magnolia.home`** is used to set several other Magnolia properties specifying locations like **`magnolia.resources.dir`** (location of Magnolia resources and light content), **`magnolia.cache.startdir`** (location of persisted Magnolia cache files), **`magnolia.upload.tmpdir`** (destination of files uploaded to Magnolia), **`magnolia.repositories.home`** (location of Magnolia's JCR repository), **`magnolia.logs.dir`** (location of Magnolia log files) and **`magnolia.author.key.location`** (location of the Magnolia publication key). **`magnolia.home`** is used to specify other file system destinations used by Magnolia under a parent directory.

Don't forget: you can still override individual Magnolia properties defining locations.

magnolia.update.auto: if true, Magnolia doesn't wait for user input to install or update Magnolia.

magnolia.resources.dir: the location of Magnolia light modules. Your Magnolia web application will probably be a combination of Magnolia Java modules and Magnolia light modules consisting of files read from the file system.

If you want to use light modules in your Magnolia container, you may want to set **magnolia.resources.dir** to a Docker volume where you add files to and share among your Magnolia containers.

magnolia.repositories.home: the directory where Magnolia stores JCR repository files and Lucene indices. You may want to persist, add or modify these files for your Magnolia container. **magnolia.repositories.home** lets you set its location.

magnolia.repositories.jackrabbit.config: the location of the Jackrabbit JCR configuration file. This file contains database configuration and file locations.

magnolia.logs.dir: the directory where Magnolia log files will be stored.

magnolia.bootstrap.dir: the directories where bootstrap content will be loaded from.

magnolia.bootstrap.authorInstance: one of the most fundamental Magnolia properties. It controls whether Magnolia will run as an author instance (true) or as a public instance (false).

magnolia.develop: improves Javascript generation performance when developing, should be set to false for production deployments. You may want to change this based on how the container is used (development versus production containers).

magnolia.author.key.location: the location of the private and public key used for publication of content from author to public instances.

You can also define your own properties and use them in Magnolia configuration files. This includes the Jackrabbit JCR configuration file.

Here's an example:

```
<DataSources>
 <DataSource name="magnolia">
   <param name="driver" value="com.mysql.jdbc.Driver" />
   <param name="url" value="jdbc:mysql://localhost:3306/magnolia" />
   <param name="user" value="root" />
   <param name="password" value="password" />
   <param name="databaseType" value="mysql"/>
   <param name="validationQuery" value="select 1"/>
 </DataSource>
</DataSources>
```

The properties **magnolia.database.url**, **magnolia.database.user** and **magnolia.database.password** now specify the database connection used by the JCR repository and can be set when you build your Docker image or run the container.

BEST PRACTICE

Set key Magnolia properties as Java properties, e.g. `-Dmagnolia.update.auto=true` when running the JVM containing Magnolia.

There are a couple of ways to do this:

- In your ENTRYPOINT script
- If you are using Tomcat, in the CATALINA_OPTS environment variable or a **setenv.sh** script

Here's a simple example of a **setenv.sh** script that initializes CATALINA_OPTS:

```
#!/usr/bin/env bash

# Container settings - Adjust these default settings according to your needs

#
# JVM settings
#
```

```
export CATALINA_OPTS="$CATALINA_OPTS \
    -server \
    -Djava.security.egd=file:/dev/./urandom \
    -Djava.awt.headless=true"

#
# JVM memory settings
#
export CATALINA_OPTS="$CATALINA_OPTS \
    -Xms$JVM_XMS \
    -Xmx$JVM_XMX"
```

This script defines the Java properties **`java.security.egd`** and **`java.awt.headless`** as well as the starting and maximum heap size for the JVM. It also enables the Java HotSpot Server VM.

The starting and maximum heap sizes are set from environment variables when the JVM is started and could be set from ENV variables you define for your Docker image.

BEST PRACTICE

Use ENV and ARG parameters to set important Magnolia properties.

With so many Magnolia properties, what properties should you set through ENV and ARG parameters?

It's hard to give absolute recommendations without qualifications for every situation, but we recommend thinking about what you want to end up with: one image that can be run in any situation (possible, but you'll probably end up with a lot of ENV parameters and possibly encounter problems setting several related Magnolia properties) or several images with some key Magnolia properties set as ARGs and other Magnolia properties set as ENV properties.

Here's a breakdown of how commonly used Magnolia properties might be considered as ENV or ARG parameters:

ENV parameters

Magnolia display properties like **`magnolia.ui.sticker.environment`**, **`magnolia.ui.sticker.color`**, **`magnolia.ui.sticker.color.background`** and **`magnolia.webapp`**.

These properties can be used to customize the appearance of Magnolia Admincentral to help users identify what Magnolia instance they are using, such as a production, test or dev instance; and the Magnolia instance, such as author or public instance.

ARG parameters

Magnolia path properties like **`magnolia.resources.dir`**, **`magnolia.repositories.home`**, **`magnolia.bootstrap.dir`**, **`magnolia.logs.dir`** and **`magnolia.author.key.location`**.

These locations could be set with ENV parameters, but you will probably want a standard layout for your Magnolia related files and directories for all your Magnolia containers. You can implement a standard file structure for Magnolia by setting these as ARG parameters.

ENV or ARG parameters

Magnolia runtime properties like **`magnolia.bootstrap.authorInstance`** and **`magnolia.develop`**.

Magnolia Jackrabbit properties like **`magnolia.repositories.jackrabbit.config`** or custom Java properties used in Jackrabbit configuration or the magnolia.properties file.

These are Magnolia properties that could be set as ARG parameters if you want Docker images with fixed key Magnolia properties or ENV parameters if you want to defer the setting of Magnolia properties to deployment time.

JVM settings

While you are setting your Magnolia properties, don't forget to set important JVM parameters, such as:

- Starting heap size
- Maximum heap size
- Garbage collection settings

BEST PRACTICE

Set a starting heap size of at least 1 GB, e.g. `-Xms1g`.

Magnolia does a lot of work on starting up; a larger starting heap size will minimize start up duration by reducing time for collecting garbage.

BEST PRACTICE

Monitor memory usage in your Magnolia web app under realistic conditions to determine the optimal maximum heap size.

We can't give a blanket recommendation for what your maximum heap size should be. It really depends on your Magnolia web app, but maximum heap sizes from 2 to 4 GB are common. Keep in mind that bigger is not necessarily better: a very large heap might reduce the number of garbage collections but make them longer, interrupting request handling.

BEST PRACTICE

Use the `-XX:+AlwaysActAsServerClassMachine` flag if your JVM supports it.

This sets better garbage collection defaults for a Docker container and helps you avoid less efficient serial garbage collection.

We have touched on the complicated subject of JVM garbage collection configuration.

Given different JVMs, different servlet containers, different Magnolia web apps, it's hard to go beyond the general recommendations we have made (starting and max heap settings, the AlwaysActAsServerClassMachine flag).

There can be performance gains by tuning the garbage collection of your JVM, but there are a couple of caveats: you should test your changes in realistic conditions (as we mentioned earlier) and avoid tunnel vision.

'Realistic conditions' means reproducing the requests and traffic volume you expect or want to serve with a Magnolia instance. 'Tunnel vision' means not ignoring other ways to gain performance. In general, other means like implementing a good caching strategy, tuning the Tomcat connection or looking for performance bottlenecks in your Freemarker templates will yield bigger performance wins than tuning your JVM garbage collection. And don't forget, if you containerized your Magnolia public instance, you could always spin up a new container to handle more traffic.

Docker container runtime settings

One important container setting often overlooked when running Magnolia is the open files limit.

GOTCHA!

Magnolia needs a generous open files limit to run. An open files limit of 1024 in a container may not be sufficient, especially if you use an embedded database.

You have hit the open files limit if you see errors like this:

```
SEVERE: Endpoint ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=80] ignored
exception: java.net.SocketException: Too many open files
java.net.SocketException: Too many open files
```

Or

```
ERROR  org.apache.jackrabbit.core.SearchManager SearchManager.java(onEvent:431)
17.01.2008 12:52:00  Error indexing node.
java.io.FileNotFoundException: /usr/local/tomcat/webapps/myApp/repositories/
magnolia/workspaces/config/index/redo.log (Too many open files)
```

Magnolia usually needs more than 1024 open files to run, so set the open file limit to something generous when running the Magnolia container:

```
docker run -it --ulimit nofile <soft limit>:<hard limit> <your image>
```

See: https://docs.docker.com/engine/reference/commandline/run/#set-ulimits-in-container---ulimit

Containerizing Your Content

Once you have created a container including the Magnolia web application and all its ingredients, you may want to think about what is going to happen when you launch a container based on your image.

Suppose your Magnolia container is going to be used as a Magnolia public instance. In that case your instance will have defined web pages, images, resources and other web content.

Some of that content - page, area and component templates, customizations of Magnolia apps and configuration and more - may be defined as 'light' content.

Some of the content may be bootstrapped from Magnolia modules in your Magnolia web application but you may want to load other content when starting Magnolia in a new container.

Suppose you have this scenario: a Magnolia author instance and two Magnolia public instances running in Docker containers built from your Magnolia image. You want to start a third Magnolia public instance to handle increased traffic.

That new Magnolia public instance must have the same content (web pages, images, resources, etc.) as the other public instances.

There are a few ways of initializing content on a new Magnolia public instance:

- Restore the JCR repository on the new instance from a backup
- Synchronize the content from the Magnolia author instance
- Import an export the JCR repository
- Add bootstrapped content to a Magnolia module included in the Magnolia web app

You can even use a mixture of the above techniques to initialize the content on a new Magnolia instance.

The above techniques may affect how you build your image. For example, you could:

- Add logic to your CMD or ENTRYPOINT script to retrieve a Magnolia backup and restore it before starting the web container and Magnolia.
- Chain together separate scripts in your CMD or ENTRYPOINT, one to retrieve a backup and restore it, another one to launch the web container and Magnolia.
- Add logic to your CMD or ENTRYPOINT script to launch content synchronization between your new Magnolia public instance and the Magnolia author instance.

Capturing content: backup and restore

If you have a running Magnolia instance with up-to-date content, you can make a copy of the content and restore it on a new Magnolia instance.

Let's take a closer look at your options for backing up and restoring Magnolia content.

Magnolia Backup module

Magnolia has its own backup and restore tool: https://documentation.magnolia-cms.com/display/DOCS62/Backup+module

The Backup module can make a copy of your entire JCR repository and restore it on another Magnolia instance, even if the new Magnolia instance uses a different JCR repository configuration. This is a useful feature if you are transferring data between different environments, for example, between a Magnolia production instance that uses MySQL for its JCR repository and a Magnolia development instance using a different database, like Derby, to store its repository.

The Backup module can read and write zipped backups to reduce the size of large repositories.

A backup can be kicked off via Magnolia's **`command`** REST API (cf. https://documentation.magnolia-cms.com/display/DOCS62/Commands+endpoint+API), or by a scheduled job (https://documentation.magnolia-cms.com/display/DOCS62/Scheduler+module).

There are a few things to be aware of when using the Backup module:

- A backup can fail if Magnolia is writing to the JCR repository while the backup is taken. To overcome this issue you can specify retry options. The Backup module will then wait and try to take the backup again.
- Magnolia doesn't have to be running to make a backup. The Backup module can be run from the command line. This prevents failed backups due to changes to the JCR repository.
- You can't restore a backup while Magnolia is running. When restoring a backup, Magnolia must be stopped.
- The Backup module can only backup and restore the entire JCR repository. It cannot back up and restore individual workspaces or parts of workspaces.
- The Backup module may not be suitable for very large JCR repositories. It may be slower than a native database backup and restore.

Magnolia import and export

You can export JCR content from Magnolia in XML or YAML and import it in a different Magnolia instance. To export or import content Magnolia must be running .

Like with the Backup module, exporting and importing JCR content doesn't depend on your JCR configuration. Unlike with the Magnolia Backup module, you can choose what content to restore - an entire JCR workspace or part of a JCR workspace.

However, that flexibility comes at a price. You can't export the entire JCR repository in a single job, so you will need to manage a collection of files on import. You will probably need to automate exporting and importing using a script that handles the resulting files.

Exporting and importing will be slower than backing up and restoring with the Backup module, especially if your JCR repository is large.

Native Database Backup and Restore

You don't have to use Magnolia to backup and restore your JCR content.

Most of your content will be stored in a database and you can use database tools to back up and restore your JCR content.

GOTCHA!

Your JCR content may not be entirely stored in the database. Some of it may be stored in files.

What gets stored in the database and what gets stored in files depends on the JCR configuration used by Magnolia.

If your JCR configuration uses a file data store like this:

```
<DataStore class="org.apache.jackrabbit.core.data.FileDataStore">
 <param name="path" value="${rep.home}/repository/datastore"/>
 <param name="minRecordLength" value="1024"/>
</DataStore>
```

Or if your JCR configuration has a workspace that uses FilePersistenceManager (e.g. **`org.apache.jackrabbit.core.persistence.bundle.BundleFsPersistenceManager`** or **`org.apache.jackrabbit.core.persistence.xml.XMLPersistenceManager`**), you will also have to make copies of any files Jackrabbit is using to store your JCR content.

You can configure Jackrabbit to use in-memory data stores or database-based data stores and persistence managers to avoid storing any JCR content in files, of course, but there may be performance tradeoffs.

Using a native database backup and restore tool will probably be faster than using the Magnolia Backup module or Magnolia export and import. But, remember that you need to back up the content stored in the database and the content stored in the file system.

Synchronizing content

Another option to synchronize your JCR content to a new Magnolia instance is Magnolia's Synchronization module. It uses the same mechanisms that are used to publish content to transfer data from the Magnolia author instance to the new public instance.

The Synchronization module also has a REST API, the Synchronization REST module, that can help manage the transfer of content.

More information on these modules can be found here:

https://documentation.magnolia-cms.com/display/DOCS62/Synchronization+module

https://documentation.magnolia-cms.com/display/DOCS62/Synchronization+REST+module

Both the Magnolia author instance and the new Magnolia public instance have to be running to synchronize.

There are a few considerations you should be aware of when using the Synchronization module:

- Synchronization will be slower than backup and restore.
- Synchronization will put some load on the Magnolia Author instance to select, prepare and send the content to the public instance.
- Synchronization is done per JCR workspace and you may need to synchronize several workspaces in the JCR repository to your new public instance with as many synchronization requests to the Magnolia author instance.

Given these caveats, we recommend:

BEST PRACTICE

Use the Backup module for most of your content and use the Synchronization module to update only the content that has changed since the backup was made.

You can specify a **`fromDate`** parameter when synchronizing using the REST API or synchronization commands. Only content that has been modified since the **`fromDate`** will be sent to the target public instance.

BEST PRACTICE

To avoid overloading the Magnolia author instance, use the Synchronization REST module to check for running synchronizations before launching a new synchronization.

Containerizing Your Light Content

The Magnolia web app isn't the only place to create your web project. You can also use Magnolia 'light development' to define:

- Web page, area and component templates
- Content types
- Content apps
- REST endpoints
- Customizations of Magnolia apps
- Light modules

Light development uses YAML files to build 'definitions' - a new page template, for example - and make it available in Magnolia without rebuilding and redeploying your Magnolia web app.

Unlike the Magnolia web app, if you want to change your light content, you change the definition file directly and Magnolia will reload the definition as soon as it is saved.

Think of your Magnolia web project as being contained in:

- The Magnolia web app bundle (usually built with Maven and always present)
- Magnolia light content in definition files (optional)

Your Docker image might contain both the Magnolia web app and light content and definitions.

Light content is made to be changed and extended. The light content in your Magnolia project will probably be updated much more frequently than the Magnolia web app itself (which is a good thing, saving you from rebuilding and redeploying the Magnolia web app), so your Magnolia Docker image should make deploying light content easy.

Which brings us to our first recommendation when using light content in your Magnolia web project:

BEST PRACTICE **Don't bundle light content into your Magnolia web app.**

Bundling your light content into your Magnolia web app would mean you would have to change your web app when your light content changes, possibly update your Docker image, and if so, spin up new containers from the changed image.

BEST PRACTICE

Add your light content when you start your Magnolia container.

If your Magnolia web app and light content are decoupled, you don't need to include your light content in your Docker image and can easily add the light content when you launch a container.

BEST PRACTICE

Define a volume in your Docker image for light content.

Light content is just a collection of YAML files, so it is best to provide a volume for them in your Docker image.

Reminder: Magnolia needs to know where its light content is stored, specified by the Magnolia property **`magnolia.resources.dir`**. Your image should set **`magnolia.resources.dir`** to point to the volume where light content will be stored.

Because light content will be deployed and updated outside of your Docker image, your image should accommodate different requirements.

Your web project will probably be run in different environments:

- A production environment to serve your live content
- A test environment to test new features and content
- A development environment(s) to build new features and content

Each of these environments will use different versions of light content and your Docker image should be able to run in any of these environments.

One way to achieve this is to use your source code management tool. You probably will have different versions of your light content files in your SCM repository: the current development versions, the current candidates for release under test before deployment and the current versions deployed in your production environment.

BEST PRACTICE

Automate the deployment of your light content with your source code management tool.

Many SCM tools use tags or branches to designate specific groups of files. Many also provide hooks to automatically execute scripts when a branch or tag is changed.

You could automate the updates to your different environments as follows:

- Production environments: When the “production” branch of your light content files is changed, check out the changed light content files into the volumes used to store light content in your production containers.
- Test environments: When the “test” branch of your light content files is changed, check out the changed light content files into the volumes used to store light content in your test containers.
- Development environments: Send out a notification that the main development branch has been changed so developers can check out or merge new changes when they are ready.

Taking the automated management of light content a step farther is possible: rather than having separate volumes for Docker containers in different environments, consider sharing volumes among running containers:

- Production environment: Shared volume for light content used by production containers
- Test environment: Shared volume for light content used by test containers
- Development environments: Private volumes for light content for developers to avoid breaking or overwriting light content files

Hacking Magnolia

Making Magnolia work on Alpine

GOTCHA!

Alpine has not been certified by Magnolia, because it does not meet the requirements to run all Magnolia features. Tread carefully.

If you choose an Alpine or an Alpine-derived image like openjdk:8-alpine as the base image for your Magnolia container, you may see errors like these in the Magnolia log when starting Magnolia or logging in:

```
ERROR info.magnolia.admincentral.findbar.search.ResultCollector 14.08.2019 14:29:33
-- An error occurred during the search process, therefore an empty collection
will be returned.
java.util.concurrent.CompletionException: com.google.common.util.
concurrent.UncheckedExecutionException: info.magnolia.objectfactory.
MgnlInstantiationException: Failed to create instance of [interface info.magnolia.
periscope.rank.ResultRanker]
      at java.util.concurrent.CompletableFuture.encodeThrowable(CompletableFuture.
java:314) ~[?:?]
      at java.util.concurrent.CompletableFuture.completeThrowable(CompletableFutu
re.java:319) ~[?:?]
      at java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.
java:1702) ~[?:?]
      at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.
java:1128) ~[?:?]
      at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.
java:628) ~[?:?]
      at java.lang.Thread.run(Thread.java:834) [?:?]
```

The stack trace goes a long way down, but here's the root cause:

```
Caused by: java.lang.NoClassDefFoundError: Could not initialize class org.nd4j.
linalg.factory.Nd4j
```

Or the JVM crashes when you try to log in, like this:

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00000000000021c6, pid=1, tid=0x00007fef9c12fb10
#
# JRE version: OpenJDK Runtime Environment (8.0_212-b04) (build 1.8.0_212-b04)
# Java VM: OpenJDK 64-Bit Server VM (25.212-b04 mixed mode linux-amd64 compressed
oops)
# Derivative: IcedTea 3.12.0
# Distribution: Custom build (Sat May  4 17:33:35 UTC 2019)
# Problematic frame:
# C  0x00000000000021c6
#
# Failed to write core dump. Core dumps have been disabled. To enable core
dumping, try "ulimit -c unlimited" before starting Java again
#
# An error report file with more information is saved as:
# /usr/local/tomcat/hs_err_pid1.log
#
# If you would like to submit a bug report, please include
# instructions on how to reproduce the bug and visit:
#   https://icedtea.classpath.org/bugzilla
#
```

Here's the problem: the nd4j library is a native Java library depending on system libraries not present on Alpine and the JVM itself.

Magnolia's Periscope Result Ranker module (https://documentation.magnolia-cms.com/display/DOCS62/Periscope+Result+Ranker+module) uses the nd4j library. This module uses neural networks to store Find Bar search results and rank new search results by relevance for an improved user experience.

There is no fix for the native nd4j libraries for Alpine itself. However, you can choose to not use the Periscope Result Ranker to avoid the library incompatibilities.

You can:

- Exclude the module from your Magnolia web app
- Disable the Periscope result ranker in your Magnolia web app

Excluding the Periscope result ranker module will reduce the size of your Magnolia WAR file by about 207 megabytes worth of various jar files used by the Periscope result ranker module and its dependencies.

```
<dependency>
 <groupId>info.magnolia.dx</groupId>
 <artifactId>magnolia-dx-core-webapp</artifactId>
 <type>war</type>
 <exclusions>
   <exclusion>
     <groupId>info.magnolia.periscope</groupId>
     <artifactId>magnolia-periscope-result-ranker</artifactId>
   </exclusion>
 </exclusions>
</dependency>
<dependency>
 <groupId>info.magnolia.dx</groupId>
 <artifactId>magnolia-dx-core-webapp</artifactId>
 <type>pom</type>
 <exclusions>
   <exclusion>
     <groupId>info.magnolia.periscope</groupId>
     <artifactId>magnolia-periscope-result-ranker</artifactId>
   </exclusion>
 </exclusions>
</dependency>
```

If you still want to include the Periscope result ranker module in your Magnolia web app, because you want to run the same Magnolia web app in both Alpine containers and non-Alpine containers, you can disable the result ranker:

- Through a Magnolia Incubator module that allows you to control the result ranker with a Magnolia property
- Through a light module that turns off the result ranker

The Periscope Control module allows you control whether the Periscope result ranker is enabled or disabled through the Magnolia property **`magnolia.periscope.resultRanking`**.

You can set **`magnolia.periscope.resultRanking`** in a Magnolia properties file bundled in your Magnolia web app or you can set it as Java property just like other Magnolia properties we have already discussed.

To use the Periscope Control module, add the following Maven dependency to your Magnolia web app POM:

```
<dependency>
 <groupId>info.magnolia.services</groupId>
 <artifactId>periscope-control</artifactId>
 <version>1.0</version>
</dependency>
```

Once you added the Periscope Control module to your web app, you can turn the result ranker on or off by setting **`magnolia.periscope.resultRanking`** through ENV or ARG variables passed to Magnolia through your container environment.

If you don't want to use the Periscope Control module, you can use definition decoration (see https://documentation.magnolia-cms.com/display/DOCS62/Definition+decoration) to disable the result ranker.

Here's how to disable the Periscope result ranker by decorating its configuration:

1. Create a directory at **`<magnolia.resources.dir>/<your light module name>`**. This is the starting point for a new light module that Magnolia will load.

2. Create a file <**`magnolia.resources.dir>/<your light module name>/module.yaml`** containing:

```
version: 1.0
dependencies:
  core:
      version: 6.1/*
  periscope-core:
      version: 1.2.2/*
```

This is your light module declaration. It has two dependencies, one for the Magnolia core module (must be v6.1 and later) and one for the Periscope core module (must be v1.2.2 and later).

3. Create a decoration file for periscope-core at **`<magnolia.resources.dir>/<your light module name>/decorations/periscope-core/core.yaml`** containing:

```
resultRankerConfiguration:
  disabled: true
```

This decoration will disable the Periscope result ranker when Magnolia loads your light module.

Of course, if you already have light modules, you could just add the definition decoration to an existing light module.

Finally, there's one last thing to be aware of if you build your own light module for controlling the Periscope result ranker:

GOTCHA!

Don't forget to include the dependencies on core and periscope-core in your light module declaration! The dependencies ensure that your updated configuration for Periscope is correctly loaded.

Running Magnolia in memory-limited containers

If you are using Magnolia 6 or later and want to make it work in a container with limited memory, you should limit the memory used by the Periscope Result Ranker module.

You have several options for putting Magnolia on a memory diet:

- Turn off result ranking
- Set Java properties to control the memory used by the Periscope Result Ranker module

You may consider doing both, depending on how you use Magnolia:

- Magnolia public instances
- Magnolia author instance

Periscope result ranking is useful to content authors working in Admincentral, usually on a Magnolia author instance. Your content authors probably would not be working in Admincentral on your Magnolia public instances, so you may want to turn off Periscope result ranking on those.

If you want to use Periscope result ranking, we recommend that you set Java properties that limit the memory used by the result ranker:

- **`org.bytedeco.javacpp.maxbytes:`** limits off-heap memory used by the result ranker
- **`org.bytedeco.javacpp.maxphysicalbytes:`** limits on-heap memory used by the result ranker

Both **`org.bytedeco.javacpp.maxbytes`** and **`org.bytedeco.javacpp.maxphysicalbytes`** can be absolute sizes like the JVM starting and maximum heap sizes. They can also be specified as relative percentages of the JVM (**`org.bytedeco.javacpp.maxphysicalbytes`**) and the physical memory of the container (**`org.bytedeco.javacpp.maxbytes`**).

GOTCHA!

If `org.bytedeco.javacpp.maxbytes` isn't specified, a limit of off-heap memory equal to the maximum JVM heap size will be used. Make sure your JVM heap and memory settings for the result ranker add up!

If you are running Magnolia in a memory-limited container, make sure your memory is big enough to fit:

- The maximum JVM heap
- The maximum off-heap memory for the result ranker
- Any other memory used by your container (OS, other processes)

For more on tuning the memory used by result ranker, see: https://deeplearning4j.konduit.ai/config/config-memory

magnolia®

Magnolia is a leading digital experience software company. We help brands outsmart their competition through better customer experiences and faster DX projects. Get full headless flexibility and seamless workflows across best-of-breed digital experience stacks. Global leaders such as New York Times, JetBlue, Avis and Atlassian all rely on Magnolia for maximum reliability, high speed project implementation and exceptional omnichannel experiences.

Get in touch

info@magnolia-cms.com
www.magnolia-cms.com

Switzerland - Headquarters
Oslo-Strasse 2
4142 Münchenstein (Basel)
Switzerland

Office +41 61 228 90 00

United States
311 W 43rd
New York, NY 10036
United States of America

Office (305) 267-3033

Czech Republic
Chobot 1578
767 01 Kroměříž
Česká republika

Office +420 571 118 715

Spain
Paseo de la Castellana 194
28046 Madrid
Spain

Office +34 91 579 85 82

United Kingdom
16 Upper Woburn Place
London WC1H 0AF
United Kingdom

Office + 44 203 741 8083

Singapore
7 Temasek Boulevard
Suntec Tower One, Level 44-01
038987 Singapore

Office +65 64 30 67 78

Vietnam
Etown 1 Building
Unit 7.10
364 Cong Hoa Street
Tan Binh District
Ho Chi Minh City, Vietnam

Office +84 28-3810-6465

China
上海市闵行区申长路998号龙湖虹桥条街E栋5F
智筹工场 532室

Office +86 2133 280 628