BEST PRACTICES FOR STRUCTURING AND NAMING ARTIFACTORY REPOSITORIES

March 2018 | www.jfrog.com

TABLE OF CONTENTS

EXECUTIVE SUMMARY 3

INTRODUCTION 5

1. Using a Repository Manager 5
2. Creating Naming Conventions 6

NAMING STRUCTURE BASICS 7

1. Team or Product 7
2. Technology 8
3. Maturity 8
4. Locator 9

REPOSITORY TYPES 10

1. Local Repositories 11
2. Remote Repositories 12
3. Virtual Repositories 13
4. Distribution Repositories 14

REPOSITORY ORGANIZATION AND MANAGEMENT 15

1. Security 15
2. Performance 15
3. Operability 16

RECOMMENDED CONFIGURATIONS 17

CONCLUSION 19

EXECUTIVE SUMMARY

"There are 2 hard problems in computer science:
cache invalidation, naming things, and off-by-1 errors."

Leon Bambrick, Computer Programmer

Devising the right repository naming conventions for your organization is essential. Creating the right repository structures, for any product development, plays a vital role in promoting a coherent product scaling strategy. It not only reduces overhead of random multiple repository creations, but helps teams discern the purpose of using a repository manager.

Using Artifactory as your repository manager, combines the power of a robust universal binary repository that hosts all your different kind of binaries in one place, with enterprise-grade features that fully integrate into your software development lifecycle.

Software development involves open-ended and evolving processes. And with the various teams that are involved in product development, maintaining a repository structure with utmost precision becomes one of the imperative tasks of the process. The challenge is that there are no hardcoded guidelines to follow for naming conventions or creating a repository structure.

JFrog recommends a four-part naming structure that includes:

1. A **product or team name** as the primary identifier of the project
2. The **technology**, tool or package type being used.
3. The package **maturity** level, such as the development, staging and release stages.
4. The **locator**, the physical topology of your artifacts.

This structure produces the following JFrog recommended repository naming structure that should be used throughout your organization: `<team>-<technology>-<maturity>-<locator>`.

Additional guidelines apply to the four different Artifactory repository types, that include: local, remote, virtual and distribution. **Local repository** naming conventions are composed of two use cases. The first is where the stored artifacts are your own, and the second is when they are third party. **Remote repositories** are either part of an Artifactory topology and their naming conventions should align with those defined for your local repositories, or they are central repositories making them external and giving them slightly different naming conventions. **Virtual repositories** are topology agnostic so they lack locators. And last but not least, **distribution repositories** support multiple technology types and generally end with "-dist".

When setting up your naming conventions for your repositories, the three main categories to consider are: **security, performance** and **operability.**

When organizing your repositories in Artifactory, it is best practice to manage security permissions at the repository level. This security factor will determine the different repositories you should manage, depending on the different teams working in your organization.

Performance concerns vary according to technology, and cleanup policies should be implemented in order to ensure the highest repository efficiency. Additionally, operability considerations should be applied, both at the repository structure, according to business value that depends on the way your organization is using Artifactory, and the structure of your teams. Although fewer repositories are preferred by administrators, sometimes it is better to create separate repositories, with different read/write/delete permissions, in order to prevent teams from interfering with each other's work.

All of these considerations, covered in this white paper will enable you to scale your Artifactory across global topologies and provide the DevOps support needed for large-scale enterprise JFrog Artifactory installations.

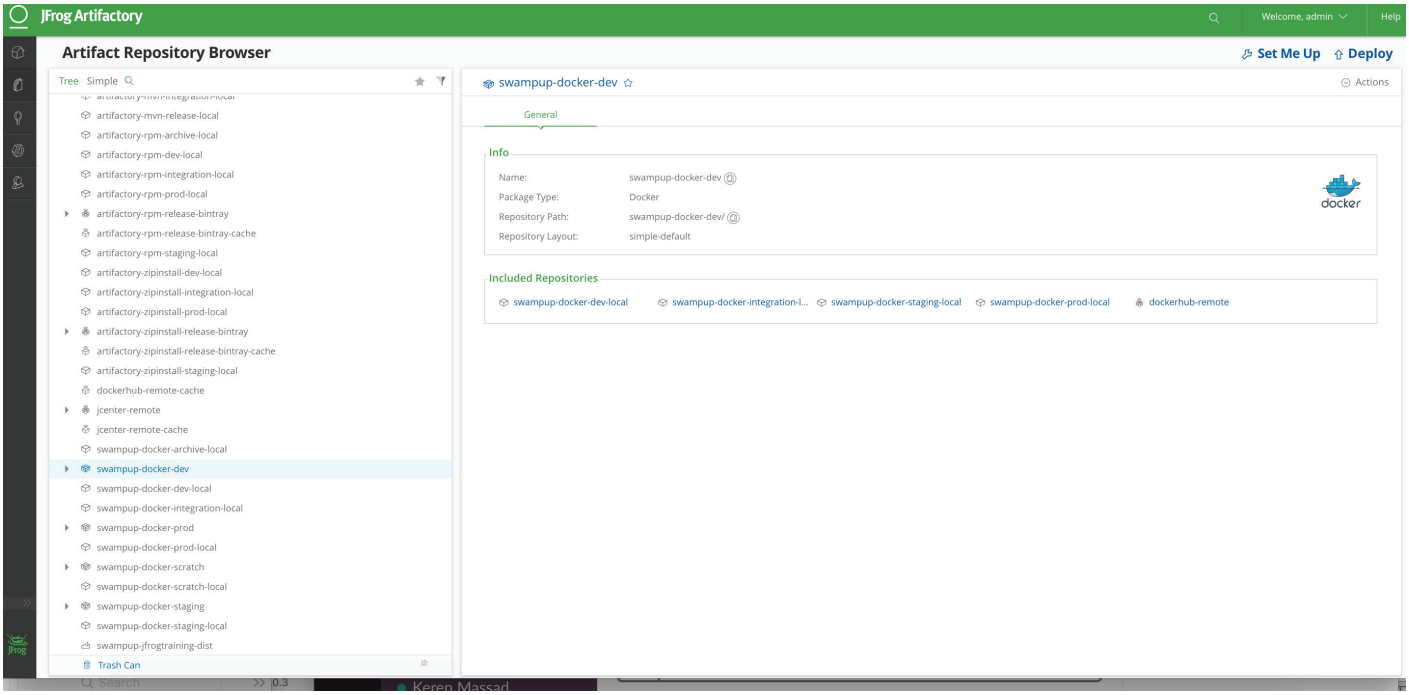
INTRODUCTION

USING A REPOSITORY MANAGER

JFrog Artifactory is a *Universal Binary Repository Manager* that was created to speed up development cycles. This means that it's not only **a repository**, but also a highly capable **manager** that aids in organizing multiple repositories to ease the distributed software development process.

When defining guidelines and conventions for your repositories, flexibility is preferred over rigid rules. Creating elastic guidelines offers Artifactory administrators enough room to tailor rules on a need basis.

Naming conventions and repository structures go hand in hand. It is always a tough call to choose an appropriate name and decide if you need a single repository or multiple repositories. It is important that the organization structure you pick be one that works with how your development, test, deployment and distribution flow works in your organization. The naming convention and organization structure represented here is based largely on a number of fairly common flows, but may not be suitable for all organizations. Hopefully, however, you can use the considerations in organization and naming laid out here to adapt it to your own naming convention.



CREATING NAMING CONVENTIONS

Organizations often deal with multiple **technologies, life cycles**, and **products**, that yield in multiple repositories. And whenever you have more than one of something, you need to name it. As developers, over the past several decades we have learned that a name can either clarify what you are doing or confuse it.

This white paper talks about repository naming conventions and management. For additional information on **artifact** naming conventions, refer to Repository Layouts.

Before we dive into the details, lets review three overarching concerns:

The first is defining repository names that generate **usable URLs**. For example, since Artifactory is case sensitive, it's a good idea to use lower case letters. More importantly, avoid using characters that require URL encoding in your environments, for example the '_' character. This will make things easier for end-consumers of your Artifactory instance by simplifying their URLs, as well as on administrators who have to manage reverse proxies and load balancers.

A second concern should be familiar to all coders: **self documenting code!** Ensure that your repository names are self-documenting wherever possible. Although there is a description field, it makes things much easier when the repository name is clear.

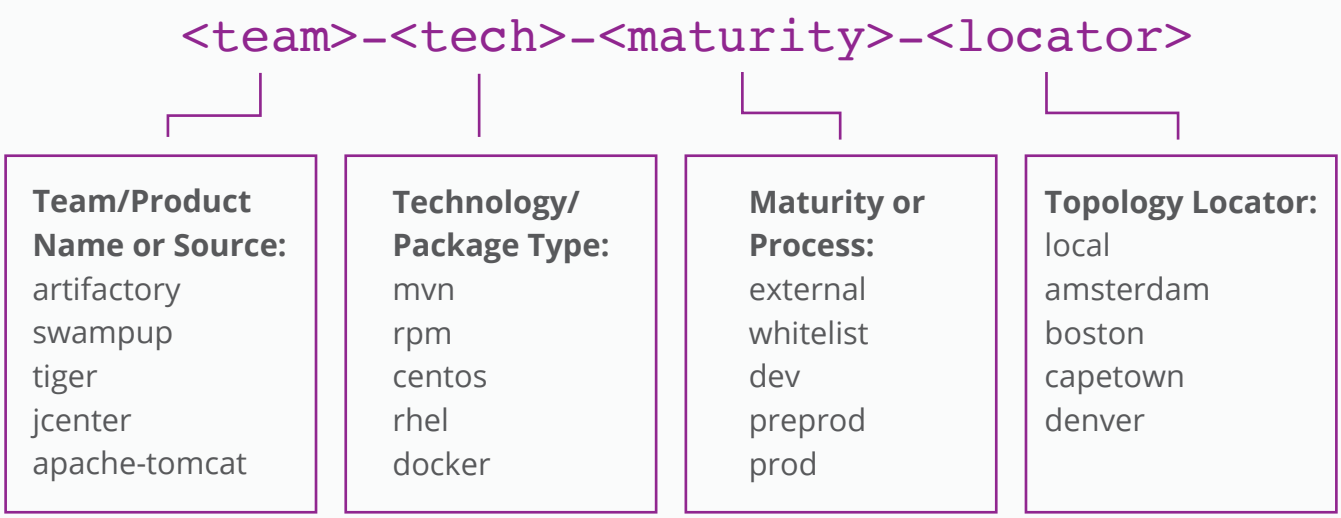
A third concern is based on the **Artifactory UI**. The most relevant piece of information that identifies your repositories should be first. By doing this, after filter options are applied, the alphabetization will place similar repositories next to one another in the Artifactory tree browser based on the significance of the components of the name.

A fourth concern is based on certain restrictions that are implied regardless of how you devise your conventions. For example, there are a some special characters ('/', '\\', ':', '|', '?', '*', '"', '”, '<', '>', '+', space) that are outright forbidden. The name can be up to 64 characters, and 58 for remote repositories. There are also some reserved and not recommended names, such as 'repo' and 'trash'. Appending the word '-cache' is also considered reserved because it is largely used for automatically created cache for remote repositories.

NAMING STRUCTURE BASICS

JFrog recommends a **four-part naming structure**, preferably in the following order.

FOUR PART NAMING CONVENTION



1. TEAM OR PRODUCT

A product or team name is the **primary identifier of the project.** You can choose to tailor the abbreviation based on your corporate naming conventions. For example, some organizations prefer to use a product keycode instead of using the entire product name. On the other hand, some prefer to use the team name along with the product name. The main idea is to **choose a name that is relevant and easily understood by your team.**

For example: `tiger`

Choosing the level of granularity for the team/product name part of the naming convention is one of the most difficult parts of developing a naming convention. This will be further discussed later on in this white paper, in the repository organization section. However, due to virtual repositories, this is also something that can be changed fairly easily later on if need be, so don't worry too much, instead pick something easily understood and consistent and see whether it works for you.

2. TECHNOLOGY

Technology largely refers to the **type of tool or package**. Artifactory is a universal binary repository manager, and its core capability enables it to store various types of packages that cover technologies such as Maven, NuGet, and Docker. Each repository should hold one type of binary files.

Building on our example: `tiger-docker`

Advanced Users
In general, any repository type may have any desired binary as far as the repository type responds to a single tool or in some cases family of tools through its APIs and indices. Therefore, when you select a repository type, you are reflecting what tool you plan to use to retrieve the artifact. This impacts the type of index Artifactory will calculate.
If you are not using any tool beyond get/put rest API commands, you may want to consider generic repositories and avoid the overhead of index calculation altogether.

Including the type of tool or package name in the naming convention helps developers identify artifacts, making it easier to browse them based on their type. In most cases this will exactly reflect the package type selected at repository creation, but you can choose to be more specific. For example, if your generic repository stores videos, you may choose the word “video” as the technology type. Other examples are: using ‘centos’ instead of ‘rpm’ or ‘rhel’, and ‘ubuntu’ instead of ‘deb’.

3. MATURITY

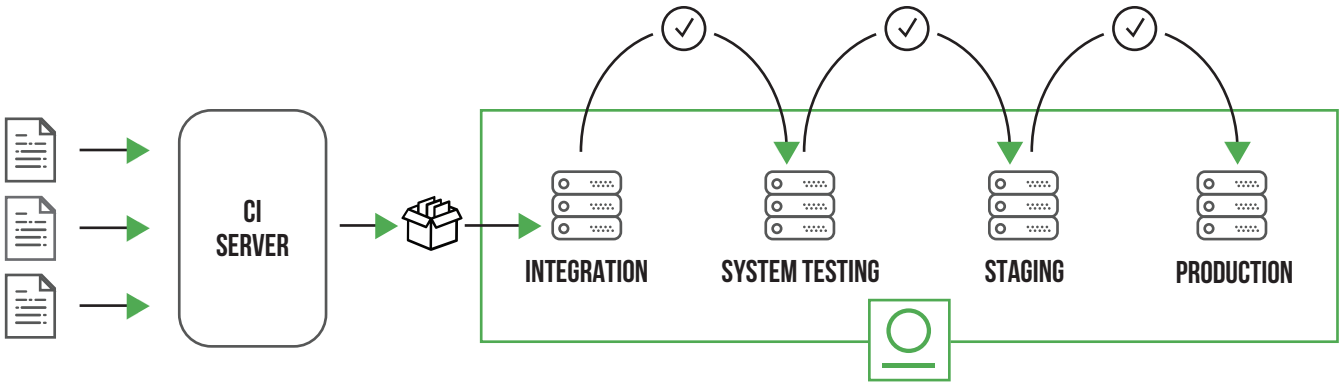
Maturity refers to the **package maturity level**, such as the development, staging and release stages. Artifact promotion can be done in many different ways within Artifactory. From simple property tagging for lesser events (e.g. “passed test X”), to larger quality gates the artifact has passed through. For the purposes of this discussion we are interested in **promotion**, where an artifact is moved or copied from one repository to another.

So why do we do this? Typically this is done when the artifact changes its **control state**. In traditional development models this may represent actual teams who own the software in different stages of its life cycle. You may have a **"sandbox"**, while the artifact is being tested by developers at their desks, and **"dev"** or **"snapshot"** for builds that are occurring out of the CI system in the initial build-on-commit. The artifact will then move to a **"qa", "preprod"** or **"staging"** repository, and finally to a **"release"** or **"prod"** repository. When an artifact retires, or when it triggers certain regulatory requirements for retention, the artifact and possibly all its dependencies can move to **"archive"**.

What about in DevOps? According to DevOps principles artifacts should not be passed off to new teams, rather they should be owned by the same team throughout their lifecycle. From an automation perspective, the control state is not about the teams within the company, rather based on the different **environments** which have different permission models to ensure artifacts are not deployed prematurely.

Continuing to build on our example: `tiger-docker-release`

While much of this white paper is focused on naming conventions, it's really about the organizationof your artifacts. There is no greater consideration in this than the concept of artifact maturity. The following diagram illustrates a typical promotion concept. The artifact progressesfrom one DevOps stage to another if quality requirements are met:

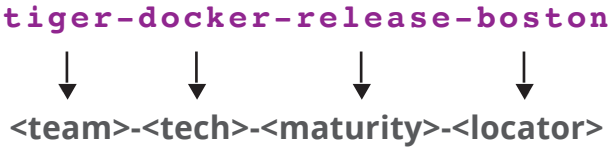


4. LOCATOR

Locator essentially refers to the **physical** topology **of your artifacts**. Each repository in a topology must be unique. **Local repositories** that are truly local, meaning their content is managed/uploaded locally, should end in **"-local"**. Local and remote repositories that are the targets of replication activity for content managed elsewhere should end in a designator for the other service.

For example, **"boston"** can be used for artifacts managed in **a datacenter** in **Boston**. For conformity, remote repositories that access external locations should end in "-remote". This is often omitted, particularly for the main central repositories, on the assumption that users are familiar with "jcenter" and "npmjs" as central repositories by name, but such assumptions can cause confusion.

Completing our example with the following repository name:



REPOSITORY TYPES

Artifactory hosts four repository types: **Local, Remote, Virtual** and **Distribution**. Local and remote repositories are true physical repositories, while a virtual repository is actually an aggregation of them used to create controlled domains for search and resolution of artifacts. Distribution repositories are a special-case for exporting data from Artifactory to JFrog Bintray.

This section provides guidelines on how to apply the naming structure outlined above, specifically for each repository type.

Any part of the naming convention can be optional when it is not relevant, and the general concept of the four-part naming convention can be adapted for additional circumstances not addressed in the initial conventions.

1. LOCAL REPOSITORIES

Using the four-part naming structure described in the previous section, we can address all required considerations for a local repository naming convention, including: **Team/Organization** (business unit or product), **Technology, Maturity**, and **Locator**. As discussed, the order represents the significance. The JFrog recommendation is: **`<team>-<tech>-<maturity>-<locator>`**, although other orders may apply in some use cases.

There are two basic use cases for local repositories:

The **first use case** is when you are referring to artifacts that relate to your own organization artifacts. In this case, **`locator`** is purely based on topological considerations, and is also fairly self-explanatory. On the other hand, **`team`** and **`maturity`** get a little more complex, and basically depend on the number of repositories needed. Team depends on business logic and permissions. Maturity depends on the gates and artifact ownership/disposition. If an Artifactory instance is focused on deployment, rather than generation, there is merit in considering that maturity is actually more important than technology. However, conforming to a uniform naming convention takes precedence.

Local Repositories
Local repositories are physical, locally-managed repositories into which you can deploy artifacts. Typically, these are used to deploy internal and external releases as well as development builds, but they can also be used to store binaries that are not widely available on public repositories such as 3rd party commercial components. Using local repositories, all of your internal resources can be made available from a single access point across your organization from one common URL.
Learn more >

A critical **secondary use case** for local repositories is when they are used to store third party artifacts. This usually covers either a scenario where for whatever reason you can't remote the source of the third party artifact (either because of an air-gap or just because it doesn't have http access), or you're implementing a white-list approach. In both of these cases, in general, **`technology`** remains the same, but the **`team name`** should be something that indicates its source location; for example, tomcat or centos. Because typically there is still a topology for these, locator also works the same as it is for other local repositories. **`Maturity`**, however, is now not something like release/dev, but instead reflects the trust level of the artifact. So it might be "upload" or "whitelist". For example, "tomcat-mvn-upload-local". If you are using local repositories to snapshot a remote in a state, this might be a date. For example,"centos7-rpm-oct2017-local".

2. REMOTE REPOSITORIES

Remote repositories fit into two categories:

Those that are part of an **Artifactory topology,** in which case their naming convention should align with that of local repositories and the four relevant parts, with the locator indicating the source repository being remoted.

Remote Repositories
A remote repository serves as a caching proxy for a repository managed at a remote site such as JCenter or Conan-Center. Artifacts are stored and updated in remote repositories according to various configuration parameters that control the caching and proxying behavior. Learn more >

Those that are **central repositories.** These are the external repositories your artifacts are being pulled from, and can be referred to by their source id, such as JCenter. For strict conformance, you could consider the following model, `<central_name>-<technology>-remote`, where the default Artifactory naming behavior uses the source. Generally, this helps to easily identify artifacts.

JCenter and Central Repositories
JCenter is a one-stop-shop for Maven artifacts in Bintray, JFrog's software distribution platform.

It is a large, popular, Maven repository that is managed by Bintray and is publicly available to all Bintray users to publish, download and share binaries with the developer community.

Conan-Center is an additional central repository available on Bintray, for C/C++ Conan packages.

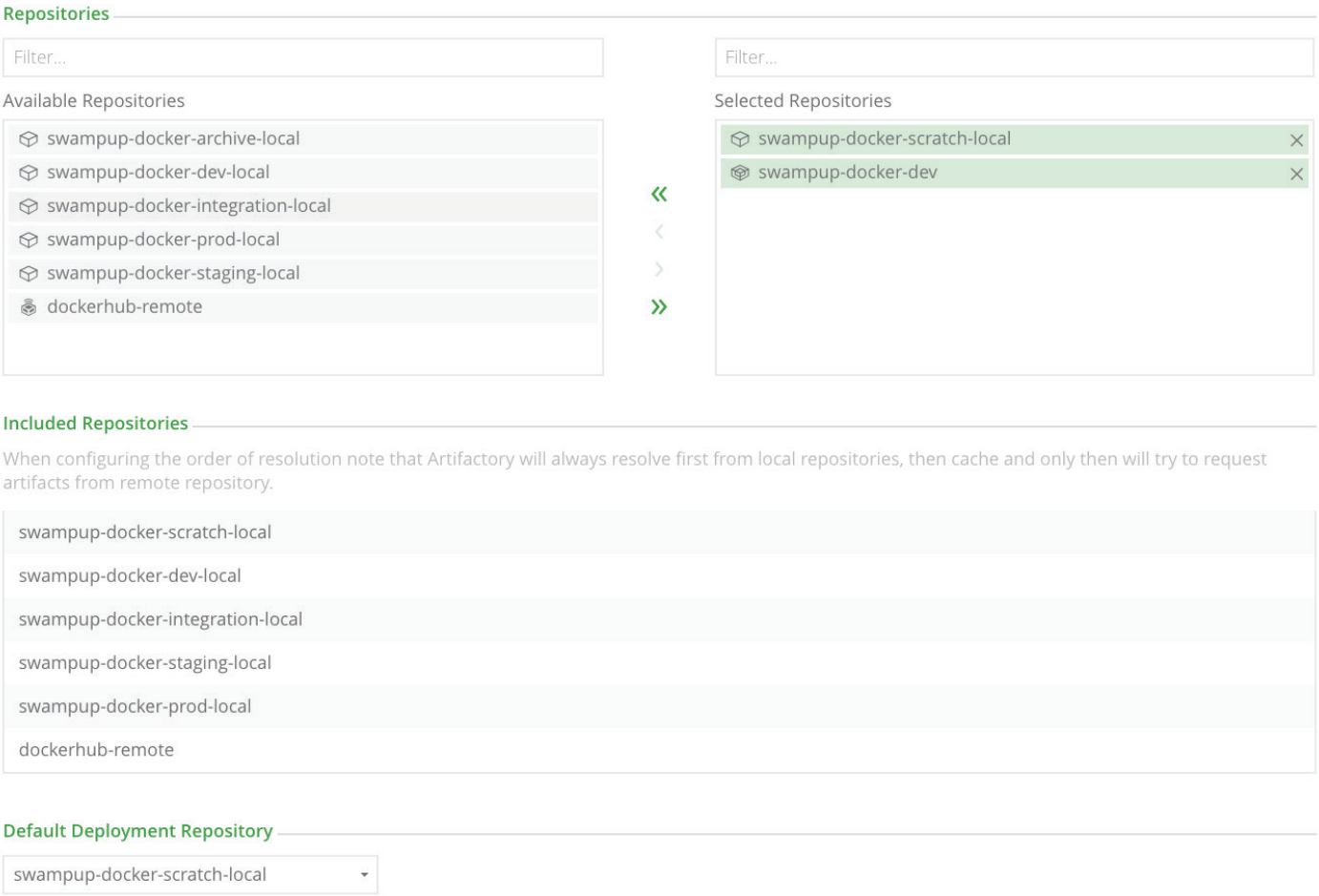
3. VIRTUAL REPOSITORIES

There are two types of virtual repository names.

Most virtual repositories do not contain a **`<locator>`**, and are made up of **`<team>-<tech>-<maturity>`**. In many cases, users do not need to know about topological implementation details. In general, its best practice that all consumption and writes are done through virtual repositories, as opposed to local/remote repositories. This is so that as many implementation details as possible can be omitted, letting the users work with a single, well-known URL. Additionally, while for local repositories maturity is strictly about artifact stages, for virtual repositories you may consider the audience more. For example, virtual repositories containing "-dev" in their name indicate the virtual repositories that the developers should be using. Finally, a common use case is for an entire company to use a virtual repository that aggregates all repositories of a specific technology, such as Docker, for both resolution and read permissions. While strict conformance with the naming convention would require the team name to be 'all' or something similar (e.g. all-mvn-release), it is more common to simply omit the team name and have repository name such as docker-stage.

The other major type of virtual repository name is aliasing for conformance, for example, with the requirements of an external tool or legacy automation. Virtual repositories allow you to make an alias of a single or multiple repositories. This may be a conformant name, but can also be highly useful if you need to accommodate a legacy build process or a particular tool to use a specific name. For example, for homebrew, it is useful to have a virtual repository called "bottles". In general these names are not subject to conformance with a standard practice, although where possible try to avoid outright violations where a virtual repository seems to conform but does not. An example would be calling a virtual repository "ci-files-local" due to requirements of automation needing this repository name; this is distinctly not recommended if it can be avoided.

Virtual Repositories
A virtual repository encapsulates any number of local and remote repositories, and represents them as a unified repository accessed from a single URL. It gives you a way to manage which repositories are accessed by developers since you have the freedom to mix, match and modify the actual repositories included within the virtual repository. You can also optimize artifact resolution by defining the underlying repository order so that Artifactory will first look through local repositories, then remote repository caches, and only then Artifactory will go through the network and request the artifact directly from the remote resource. For the developer it's simple. Just request the package, and Artifactory will safely and optimally access it according to your organization policies.
Learn more >



4. DISTRIBUTION REPOSITORIES

Distribution repositories are a bit of an outlier in this convention, because they can support multiple technology types, and generally don't have multiple maturity levels, as you should only be distributing mature artifacts. In general they should end in "-dist" as the **`locator.`** Distribution repositories that distribute products should use the following convention: **`<productname>-product-<orgname>-dist`**. Distribution repositories that are more generic should have some name designator of the ruleset they're configured for with a convention like **`<ruleset>-<orgname>-dist`**. Orgname can be omitted if your organization does not have multiple Bintray organizations, which is relatively common. However, it is mandatory to have different distribution repositories to distribute to different organizations, and additional organizations (for example to support open-source projects) are not unheard of either. A bit of forward planning doesn't hurt anything, which is why its included in the recommended convention.

Distribution Repositories
Distribution repositories provide an easy way to move artifacts from Artifactory to Bintray, for distribution to end users. As opposed to other repositories in Artifactory, distribution repositories are not typed to a particular package format, but rather, are governed by a set of rules that specify how an artifact that gets to the distribution repository should be routed to its corresponding repository in Bintray.
Learn more >

REPOSITORY ORGANIZATION AND MANAGEMENT

Now that we've established the basic repository naming structure, let's review the different considerations you need to take when organizing your repositories in JFrog Artifactory.
In essence, repository organization boils down to three things: **security, performance** and **operability**. And mostly, these considerations will determine what granularity you set **"team"** at, and to a lesser extent what granularity you calculate **maturity** levels.

1. SECURITY

Artifactory permission targets allow for managing permissions via include/exclude patterns at an individual folder or even file level. In general, the best practice here is to manage permissions at the repository level. For repositories with highly structured organization, like Maven and RPM, it is possible to achieve a great deal of granularity at the folder level. However, this can still be too complex for administrators to keep track of (although effective permissions analysis can help). This is particularly true of READ permissions, although the finer granularity for those technologies where it works may be used for write permissions.

At a minimum, you should have separate repositories within the same technology and maturity level whenever you have teams that are not collaborating or sharing data, and thus do not have/need read permissions on each other's software. You may also choose to provide different repositories based on write permissions, and assume they are aggregated in virtual repositories for read. This choice of write-based repositories is especially crucial in repository types which aren't well divided by namespacing, such as the default NuGet behavior or an npm repository that isn't scoped.

2. PERFORMANCE

Another major concern is **performance.** This varies a bit by technology, but for any given technology there tends to be a maximum number of packages that make sense in that repository. In Maven this tends to be hundreds of thousands and driven more by UI considerations. Whereas in Yum/Debian this tends to be more in the tens of thousands, and driven more by the overall approach to calculating indexes and the size of the resulting index files, and their impact on client performance.

The other side of this are cleanup policies. An artifactory server with absolutely no cleanup policies in place will grow in storage usage very fast, and in general most of it will not be things you actually need to store. Mechanisms for implementing cleanup policies are a different discussion. Some can be found here. Based on the business requirements of the organization, different

projects may have different policies. A primary driver for this tends to be maturity, discussed above. For example, a dev-sandbox docker registry may have a policy which states that any Docker tag which hasn't been downloaded in the last two weeks should be deleted. On the other hand, a regulated industry may have a regulatory requirement that any object which has been in the regulated production environment must be retained for ten years. A solid promotion model between these stages of the lifecycle to different repositories is critical. But these policies are also probably not the same for all applications being developed. While an application for processing stock trades in production will fall under regulation, that same company's tool for managing what to order for lunch can probably be discarded shortly after its "production" life cycle is complete, but does need to be maintained while it is actually being used.

3. OPERABILITY

When it gets to administering artifact repositories for specific teams in specific environments, other basic operability considerations apply. In general, these policies will want to be handled at the repository level, and so this will be a driving determination in choosing your repository structure.

The first is a fairly simple one: determining business value. If you are managing an Artifactory that spans multiple large projects and business units within the company, in addition to the considerations above, you will want to be able to determine how these different projects/units are using the Artifactory service. This may be for explicit chargebacks, or merely to track what units are resulting in what sorts of costs. As soon as you want to track usage for a given unit of organization in the company separately from other organizations, it should have its own repositories, and be broken down in the naming conventions accordingly for ease of identification.

Additionally, at a minimum, you must have separate repositories once you go beyond the bounds where the business can successfully coordinate naming conventions and directory structure organizations. That is to say if a team is too large to successfully manage something like group ids/naming conventions for artifacts without a horribly bureaucratic process, it is better to just give them separate repositories, and there is always a scale where this limitation exists. In general write permissions, and even more so delete permissions, should be reasonably specific to prevent teams from interfering with each other's work. Delete permissions in general should only be provided to a very small group, outside of policy-based reapers (see the discussion on cleanup policies in the performance section above)

Taking all this into consideration, typically administrators prefer fewer repositories. Even though the more heavily automated your repository management process is, the less it really matters. For example, in a strong DevOps environment you could end up in a situation where every single test could be viewed as a promotion. While it might make sense to use the promotion API for each test, it probably does not make sense to have a repository for each one of dozens of tests, but rather to track this via properties, and reserve separate repositories for major control points.

RECOMMENDED CONVENTIONS

The following tables summarize the best practice naming convention with examples for each repository type.

1. LOCAL REPOSITORIES

`team-tech-maturity-locator` →

Examples:

- tigerteam-docker-dev-local
- tigerteam-docker-release-local

Name Part	Recommendation
<team>	This is the hardest part of the naming convention. It is based on the granularity you want to manage permissions/performance/operability concerns. It may also be a product name, or refer to a source for third party libraries.
<team>	The content type. This is typically the package type, such as: mvn, rpm, docker. It may also be more specific, such as centos or ubuntu.
<maturity>	The maturity level within a process, either the SDLC process or a whitelisting/approval process for third party artifacts. ● For example, a series such as: ○ scratch (For developers sharing from their systems e.g. docker) ○ dev (For CI builds) ○ qa (Promoted builds) ○ preprod (Promoted builds) ○ prod (Promoted builds for use) ○ archive (Builds retained for regulatory purposes) ● For third party libraries it might be values such as: ○ upload ○ whitelist ○ jan2018 (typically used when snapshotting a remote repository)
<locator>	Based on the physical location/artifactory service ID. The default is 'local' for a repository that is actually written to, but in case of multi-push replication it may be the site of the source of pushed events. In general you should not write to a repository that doesn't have the 'local' designator except through replication.

2. REMOTE REPOSITORIES

Examples:
- tiger-mvn-release-boston
- jcenter-remote

Repository Use Case	Recommendation
Part of an Artifactory topology	For remoting another artifactory server, go with the same naming convention as local repositories, based on the repository it is remoting. ● <locator> in this case should be the identifier of the remote artifactory.
Central Repository	**<centralname>-remote** "-remote" is optional, but helpful to avoid confusion with virtual repository naming conventions. ● i.e. jcenter-remote or just 'jcenter'

3. VIRTUAL REPOSITORIES

<team>-<tech>-<maturity> →

Examples:
- tiger-docker-dev
- tiger-docker-prod

Name Part	Recommendation
<team>	A group that shares read permissions. If you are using virtual write to control writes, then you may control this at the write permission level.
<tech>	The package type.
<maturity>	This part may be omitted. However, it is often used as part of the write-control feature and/or specifically for production. Unlike <maturity> in local repositories, it is much more likely to be controlled from a deployment model perspective than a CI perspective.

4. DISTRIBUTION REPOSITORIES

Examples:

- artifactorypro-product-jfrog-dist
- examples-jfrogtraining-dist

Repository Use Case	Recommendation
Distribution names with a product	**`<productname>-product-<orgname>-dist`** <productname> is the name of the product in bintray
Other distribution repositories	**`<rule designator>-<orgname>-dist`** <rule designator> is a selector for a unique rule set indicating which teams/products should use it for distribution repositories not associated with a product
<orgname>	Designates the bintray organization being distributed to. If you are certain you will only ever have one (which is not uncommon) it can be omitted.

CONCLUSION

Organizing repositories and picking a naming convention is one of the first and most significant decisions a JFrog Artifactory administrator needs to make. While good use of virtual repositories can allow changes later, it is best to pick a naming convention up front.

This white paper has presented various considerations for a repository organization and naming convention that should help you answer the following question: "how many repositories do I need?". It provided a four-part convention, **`<team>-<tech>-<maturity>-<locator>`**, which can be used as a basic best-practice guideline for your naming and organization structure. Using this suggested convention, most organizational questions become fairly clear.

Although team granularity can be a bit of a challenge, this granularity is usually decided according to **security**, **performance** and **operability** concerns. While you may have to adjust granularity over time, a good naming convention combined with using virtual repositories can make this a relatively painless process for your team.. Additionally, you can use virtual repository aliases to avoid breaking builds as you move forward.

The conventions described in this white paper will allow you to scale your Artifactory across global topologies. It will provide DevOps support large-scale enterprise installations that serve thousands of developers across many different teams and projects.