DZone Refcardz

Brought to you by:
JFrog

CONTENTS INCLUDE:

Binary Repository Management

Patterns for Performance, Security, and Traceability

By: Carlos Sanchez

INTRODUCTION

Software development produces two distinct kinds of artifacts: (1) source code, and (2) binary artifacts. This Refcard assumes basic familiarity with source repository management, and is intended to help you design and configure a binary repository, optimize it for various workflows, and fit it smoothly into your software development lifecycle.

INTRODUCTION: REPOSITORY REQUIREMENTS

An artifact is the output of any step in the development process. Many artifacts result from builds, but other types are crucial as well. Common artifact types include:

- **ZIP or tarball files**
- **RPM or DEB packages (Linux)**
- **JAR, WAR, and EAR packages (Java)**
- **Gems (Ruby)**
- **Python packages**
- **DLLs (Windows)**
- **Source packages**
- **Documentation packages**

The two artifact super-types

The various types listed above cluster into two groups: (1) source and (2) binary. And while it is possible to use a source repository to store binary artifacts, some crucial differences between these two artifact super-types make this solution non-ideal.

What source repositories are for

Source repositories are designed simply to manage source code. A well-built source repository therefore boasts a feature-set tailored to source code management, e.g.: diffing versions, tracking deleted or overwritten files, branching, and tagging.

Source repositories deal with relatively small files. Large files (like binaries) degrade performance of the entire repository.

DVCS (Distributed Version Control Systems), like Git, streamline distributed development by cloning the full source repository to each developer's machine. Developers don't usually mess with the binaries directly, so cloning binaries stored in a source repository could waste tremendous bandwidth.

What binary repositories are for

Binary Repositories are to binaries what source repositories or VCS (Version Control Systems) are to sources. Where source repositories deal with relatively small code files that change constantly and are often cloned with abandon, binary repositories manage a completely different workflow.

Binary artifacts are often orders of magnitude larger than source files.

From the point of view of the developer (though not the designer), binary packages don't need to be diffed.

Except very rarely (e.g. snapshots and nightly builds), binary artifacts are not deleted or overwritten.

Binary artifacts usually need to store lots of metadata (package name, version, license, etc.).

When to use binary dependencies

Many static dependencies could be stored in source form, but this practice begets three problems: (1) it complicates the build process unnecessarily; (2) it encourages project-specific branching, which in many cases (e.g. shared libraries) might affect totally separate projects; (3) reproducing a build from source requires building all the dependencies, which may or may not reproduce the original build. Therefore, when you use cross-project dependencies that don't need to change quickly, it's better to store the dependencies as binaries, in a distinct binary repo.

Briefly, then, the basic pragmatic differences between the needs of source management and the needs of binary artifact management are:

Source code management	Binary artifact management
Diffing, branching, tagging	None of these (within the development process proper)
Frequent deletes/overwrites	Rare deletes/overwrites
Small files	Large files
Minimal file-specific metadata	Lots of file-specific metadata
Changing, project-specific dependencies	Relatively static, cross-project dependencies

Table 1: Needs of source vs. binary repositories

ELEMENT 1: REPOSITORY DESIGN

By itself, the term 'binary repository' doesn't say much about actual implementation. Strictly speaking, a 'binary repository' can be anything from a simple directory of files served via HTTP to a full-fledged, feature-rich repository management server that offers scads of features specifically tailored to the needs of binary repositories in particular. Therefore, you'll need to make quite a few design decisions up front

Segments and permissions

Divide each repository into groups (or software providers), each of which (a) manages a specific part of the repository namespace and (b) has appropriate permissions for its particular part. The goal is to avoid clashes between projects – an especially tricky problem when some of the shared artifacts are binary dependencies. Groups can have multiple projects, and each project can have multiple versions.

Binary Repository Management

Best practice: repository grouping
Repositories can multiply quickly, because multiple teams are using the repository manager with different permissions and usage patterns, or even because many third party repositories are being used.

In such cases, repository setup becomes extremely complex, with a possibly long list of repositories to configure. To pare down the repository list, the repository manager can define virtual repositories (or groups of repositories) where every request to that group is served from any of the repositories in the group:

	all	QA	production
releases	✓	✓	✓
third-party artifacts	✓	✓	✓
release candidates	✓	✓	x
snapshots	✓	x	x

Build and release schedule: nightly, continuous, snapshot

Distinguish which binaries require nightly, continuous, or snapshot builds and releases. (Some tools will do this for you – see feature matrix at the end of this Refcard.) Because snapshot repositories are more resource-intensive, use a snapshot binary repository only when binaries:

- **are non-durable**
- **have slightly different semantics**
- **can be deleted**
- **are implemented at build time in the simplest way possible (e.g., without source tagging and any other requirements of a formal release)**

Common build tools by platform

Some build tools will prove particularly helpful for binary repository management. Apache Maven is the most popular Java build tool that uses binary repositories (although it wasn't the first – e.g. Debian packages). Binary repositories are still quite common in the JVM world, especially with Maven, but plenty of other tools are available for various platforms:

Platform	Build tool w/binary repos
JVM	Maven, Ivy, Maven Ant Tasks, Gradle
.NET	NuGet
OSGi	P2
Linux	apt-get, Yum

ELEMENT 2: HOSTING AND MANAGEMENT

A binary repository is a hub for development teams across the whole organization, centralizing the management of all the binary artifacts generated and used by the organization. The inevitable resulting diversity of binary artifact types, and their differing positions in the overall workflow, is one major reason to use a dedicated binary repository manager, rather than just a simple file server. But more focused functionality means more decisions.

What a binary repository needs to store

Binary repositories store (1) files and (2) metadata; plus, for each of these, both (a) releases and (b) nightly builds (based on retain policies). The 'files' set has a complicating subset: third-party artifacts need to be handled differently (for combined legal and technical reasons).

In most cases it's fair to assume that files will not change and deletions will happen only for snapshots.

Common types of metadata

Selecting appropriate binary metadata can be tricky because many data-points that source files already include (because these both code and metadata are text) need to be specified separately from the binary files themselves. Here are some common metadata types and their uses:

Metadata type	Used for
Versions available	Upgrading and downgrading automatically
Dependencies	Other artifacts the current artifact depends on
Downstream dependencies	Other artifacts that depend on the current artifact
License	Legal compliance
Build date and time	Traceability
Documentation	Contextual documentation in IDEs; offline availability
Approval information	Traceability
Metrics	code coverage, rules compliance, test results
User-created metadata	Custom reports and processes

Managing multiple repositories

Even smaller organizations will probably need to host several binary repositories, each designated by project, department, and permissions. In this case a full-fledged repository manager, with permissions separate from the file system, will prove especially useful.

Typically each project should include separate snapshot and release repositories, with their desired set of permissions, and repositories for other external projects used.

Metadata and artifact search

Source code is inherently searchable; binaries obviously are not, so you'll need to put a little effort into enabling search in your binary repository. Here's what and why:

Metadata search is absolutely essential. Consider the types of metadata (and uses) listed above, and think of how useful a search feature would be – for quickly locating dependencies, isolating problematic artifacts, navigating a tangled web of cross-enterprise projects.

Metadata searches are especially crucial for **collaborative development across teams** – since one team may very easily be using a binary built by another team, and won't be able to peek through the veil of the other team's code.

Artifact search abilities for tools like IDEs, can save hundreds of hours over a period of several months. Most developers want to be able to find an artifact, and immediately add it as a dependency, without leaving the IDE. Some repositories expose search APIs (as any other modern web search), others offer downloadable precalculated indexes.

Exposing binaries with APIs

Binary repositories should allow users to upload and download artifacts, typically through HTTP/WebDav.

Users should also be able to query the metadata with REST APIs. A solid metadata query API will let you do two things: (a) automate configuration, and (b) integrate the repository with other tools, like continuous integration servers. For example, a CI server will need to query the repository metadata to see whether a new version of an artifact is available.

Caching external binaries

At some point, in some (perhaps most) projects, you'll probably use third party artifacts that are hosted in a repository external to your organization. Network latency and bandwidth will affect development speed directly – especially when your external artifacts are (gigantic) binaries – even if your team is fully on-premise. Now imagine you need to work every day with the latest build of several dependencies and each takes several minutes to download – possibly several times a day. Now consider a long chain of dependencies, and you're immediately (and with no payoff on the development side) in binary download dependency hell.

To skirt this time-sink, cache these files in your repository manager. The cached binaries can be served rapidly to other machines on the same network, after the initial request – either to human coders or directly to CI servers themselves.

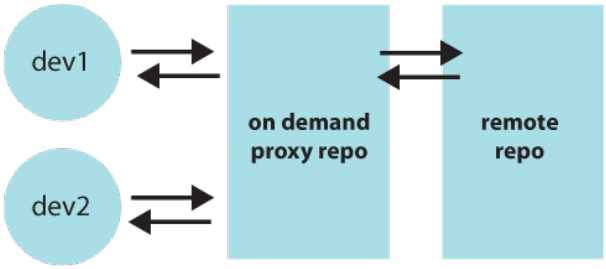
Proxying external binaries

Further, external dependencies introduce an element of unnecessary risk – simply because you can't control access to them. To remove this risk, configure your repo manager to proxy these files. Keep a copy in your private repository; then dependency availability will be up to you. You can also apply your own backup and availability policies, guaranteeing access to the artifacts even if they disappear on the upstream repository.

The external repositories can be proxied (a) on demand or (b) mirrored.

Option 1: on-demand repository

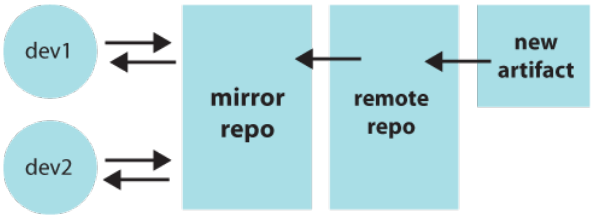
In an on-demand proxy scenario, requests to the remote repositories happen only the first time any developer requests an artifact that is not yet cached in the proxy repository. Any further requests from other developers will use the copy in the proxy repository:



Besides typical HTTP proxy features, a repository manager adds features specific to binary repositories (backed by all the metadata and information stored in the repositories). Consider the advantages of, for example, filtering by group or artifact IDs or expiring unused artifacts to reduce the space needed.

Option 2: mirrored repository

In a mirrored repository scenario, all changes are automatically synchronized to the mirror. So even the first request for an artifact is always resolved from the repository that is closest to you:



The simplest way to implement a mirrored repository is simply to use rsync from the filesystem backing the repository.

Some repository managers offer more sophisticated, checksum-based mirroring, implemented by using repository REST APIs.

To choose between on-demand and mirror repository proxying, use this table:

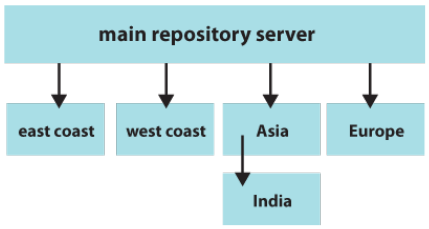
Feature	On-demand	Mirror
First request for an artifact is resolved from	Remote repository	Proxy repository
Space requirements	Low (only used artifacts are cached)	High (all artifacts are cached)
Bandwidth requirements (between proxy and remote)	Lower in principle (under low demand) but potentially higher, if many developers fetch artifacts at once	Higher in principle (all artifacts are transferred) but potentially lower in practice, if many developers fetch artifacts at once

Supporting distributed teams

When teams that access the repositories are located in different locations or distributed across the globe, it is also important to mirror the internal repositories as well as the third party repos.

To do this, setup repository manager servers hierarchically. Run a server in each location to (a) serve as mirror of the remote server and (b) synchronize the repositories' contents either on demand or (preferably) mirrored.

Use a master server as a write-only instance. Let all the other servers (distributed in different locations) proxy the master for local caching.



Artifact promotion

When an artifact is pushed to a repository it may not be the final place for it. Imagine a workflow where a release candidate artifact needs to go through integration testing and QA processes. Only artifacts that go through this process should be available for other teams or clients.

A repository manager can enforce this workflow by setting different permissions for each repository while letting only authorized users promote or move the artifacts between repositories.

For instance, when releases are pushed to the release candidate repository, set permissions to allow only the QA team to move artifacts from there to the releases repository when their tests are done (either manually or automatically). The production systems can then be configured to pull only artifacts from the final releases repository, thus enforcing the completion of the QA process.

ELEMENT 3: SECURITY & MAINTENANCE

Authentication & Authorization

Since the binary repository stores project-related binaries, the same permissions enforced for the projects themselves (such as the source code access permissions) should be used for protecting the resulting binaries. In some cases, access to the binaries may be granted without granting access to the source – and this can be managed at the repository level.

To simplify and centralize user management, configure your repository manager to integrate with other organization systems such as LDAP or single sign on servers.

Hot Tip

As with source traceability, so with binary traceability. Trace changes in the repository (such as which user uploaded an artifact and when, or who is downloading artifacts) for audit purposes.

Purging policies

Although artifacts usually must be kept for a long time (the same as any other product or distribution), there are some cases when we can benefit from purging repository contents.

Snapshot repositories need to be purged from time to time to ensure a reasonable disk usage—especially when using continuous integration heavily, since CI can easily generate several builds per day. Usually, snapshots can be purged when a new version is released, but that may be changed to just keep the last n snapshots.

Some binary repository managers offer automatic purging procedures based on defined policies (e.g. number of snapshots to retain).

Proxied repositories for third party artifacts can also be purged when the artifacts are not being used by any release — for instance, for artifacts used during a proof of concept that is discarded. In these cases it is a good practice to separate the artifacts being used in production from the artifacts used during development for trials or proof of concept.
(This can also be done during promotion: promote not only the built artifacts, but also the dependencies.) This will considerably simplify management downstream.

Managing Third Party Artifacts

Some organizations may have a policy about third party dependencies because of licensing or approval processes. Enforce these policies in your binary repository by preventing unauthorized publication to certain repositories, while still not obstructing development.

Here's a common example: third party artifacts need to be requested by a developer and approved by a legal department. But development should not be waiting on legal approval, since legal approval is required for only for release binaries. Configure your repository manager to allow use of any dependency during development.

To simplify the process, some repository managers (e.g. Artifactory) also include automatic license discovery and management and integration with license management software (e..g, BlackDuck Code Center)

High Availability

Using a repository manager to hold all your development dependencies also means that your repository is a central piece to your infrastructure: any downtime means halting development, with all the consequences.

Using a repository mirror, we can have a copy of the repository ready to be swapped if the main one fails, or we can use a more complex strategy depending on the specific repository manager used. A Network Attached Storage (NAS) can be used for the artifacts' backend and database master-slave configuration for repository managers that use a RDBMS as metadata backend.

Disaster Recovery

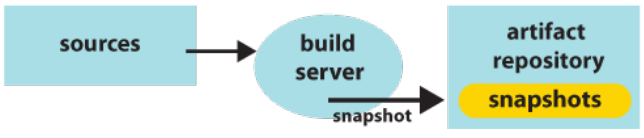
For the same reasons, the repository manager needs to be backed up and the adequate recovery measures put in place, such as offsite backups of configuration, artifacts, and metadata. Consider that the binary repository is as important as your source code repository: it holds all your releases and dependencies that may no longer be available in other locations.

ELEMENT 4: BINARY RELEASES

Continuous Integration and Binary Repositories

Continuous Builds

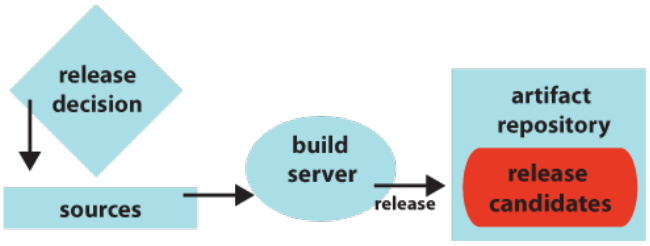
As part of the development lifecycle, source code is continuously being built into binary artifacts. As part of your continuous integration process, those artifacts should get pushed to the repository from the CI server. Don't push from developer's boxes, as all sorts of factors may affect the build (e.g. software used in the developer machine, specific configuration options, environment variables, etc.). Push like this:



Releases

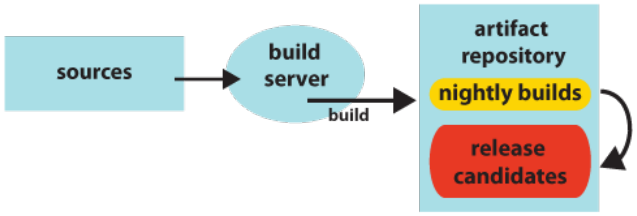
Within releases, there are two alternatives, depending on what you consider a release: (a) a release can be a special build; or (b) a release be just a promotion of a nightly build.

Release as a special build by (1) changing the version number to a release (i.e. 1.0), (2) tagging the sources, and (3) doing a new build. Maven offers the release plug-in to assist on these steps, ensuring none of them are forgotten and that the process can be automated (but functionality is limited). CI servers may have some support too, like Artifactory's Jenkins release integration (which is more useful for special build releases, since promoted snapshots are in the binary repository). Here's how a special-build release looks:



Release as a promotion of a nightly build involves (1) tagging the sources for each build, (2) picking one of those builds, and (3) deciding that this build should be the released one. This option involves a bit more discipline (as each build should be ready to be released), but reduces the implications and troubles of a scheduled release. It is also more in line with agile methodologies.

That build can be promoted in the repository manager from one repository to another (e.g., from snapshots to release candidates repositories), while the artifact itself does not change.

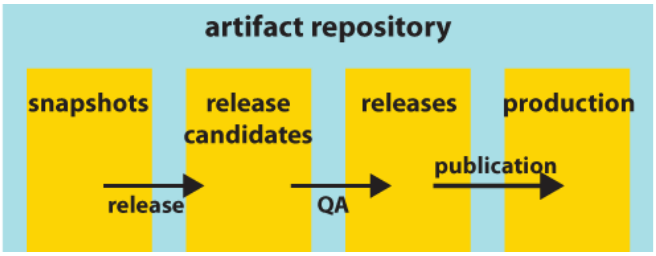


Either way you will end up with a release candidate artifact in the repository. This doesn't need to be the final one, but just one step in the process.

Use this table to decide how to treat releases:

	Special build	Promotion
Initiated by:	Source control	REST API
Performed by:	Build server	Binary repository
Process:	Change version number to release (e.g. 1.0) and build again	Tag each build and decide which will become a release after the fact (file doesn't change)
Tool support:	Maven release plugin, Artifactory's Jenkins plugin'	Repository manager servers
Trouble at release time:	Increases with the amount of time between releases	Minimal, in line with agile methodologies
Repository usage:	Release is uploaded to a different repository than snapshots	Build is promoted from one repository to another, i.e., from nightly builds to release candidates

By having separate repositories with their own semantics and permissions (based on the step within the lifecycle and target audience), we can model and enforce a healthy development lifecycle:



Tighter Build Server Integration

Continuous integration servers can interact with the repositories as any developer would, by getting artifacts from the repositories and pushing builds there. But there are also advantages to a tighter integration:

CI action	Reason
Delete local caches of the repository	Dependencies must already be present in the repository
Create new processes for each build	Enforce environment variables
Prevent users from installing packages by hand	Ensure that all dependencies are defined

Since the CI server is a 'single source of truth' for builds, it can also be used to generate the Bill of Materials to be stored with the artifacts in the repository. Include the following metadata:

- **User that triggered the build (manually or by committing to SCM)**
- **Modules built**
- **Sources used (commit id, revision, branch)**
- **Dependencies used**
- **Environment variables**
- **Packages installed**

All this information can prove useful later – for artifact scans or reports, artifact audit and security checks, and, most importantly, build traceability.

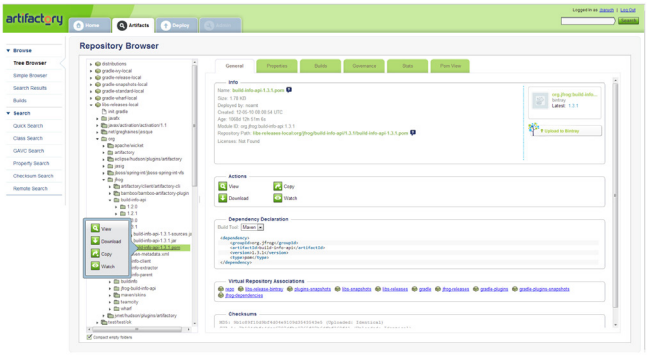
REPOSITORY MANAGERS

Although, strictly speaking, a repository can be hosted simply by serving a file directory over HTTP, specialized repository manager servers were created to implement the practices proper to the best binary repository management (discussed above).

An early project called Maven-Proxy implemented caching from Maven central repositories—an in-demand feature at the time, when transfers from the central Maven repository at iBiblio would take a long time. Since that initial tool, there are now three major repository managers contending in the space.

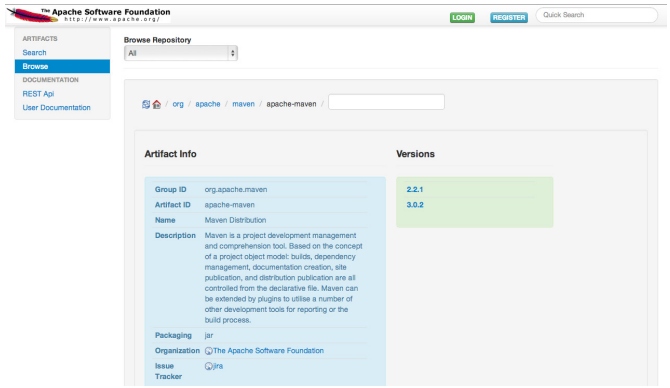
Artifactory

First released in mid-2006, JFrog Artifactory offered indexed searches, security controls and web 2.0 UI. Artifactory's development is user-needs driven, and is primarily focused on enterprise features. JFrog offers an Artifactory open source version, a commercial Artifactory Pro with extra features and Artifactory Cloud, a SaaS solution.



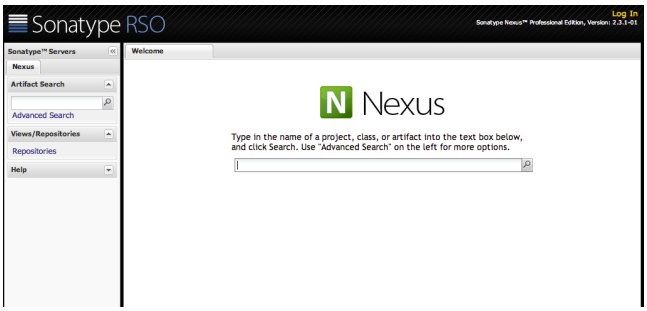
Apache Archiva

Apache Archiva launched in November 2005. It was a simple framework on top of some existing repository conversion tools within the Maven project. Initial development focused on repository conversion, error reporting, and indexing. In January of 2006, a web application was released that visualized the information and incorporated the functionality from the unmaintained Maven-Proxy project. Archiva became an Apache "top level project" in March 2008.



Nexus

Sonatype Nexus is the continuation of the Proximity repository manager, released in 2005. In 2007, Sonatype continued development as Nexus, offering both an open-source version (Nexus OSS) and a commercial one (Nexus PRO) with extra features.



Repository Manager Feature Matrix

Apache Archiva, Artifactory and Nexus share a good number of typical features expected in a repository manager. They differ mostly on enterprise-grade features (integration with other enterprise systems, security management options, etc.),support of different repository types for non-Maven artifacts, integration with CI servers, and licensing model. Most of these features are included in the commercial versions of Artifactory and Nexus.

This is a non-comprehensive list of features showing mainly the differences between the three tools. For more details there is a wiki page in continuous update at http://docs.codehaus.org/display/MAVENUSER/Maven+Repository+Manager+Feature+Matrix

Feature	Apache Archiva	JFrog Artifactory	Sonatype Nexus
License	ASL	LGPL 3 / Commercial	EPL / Commercial
Last release	1.4-M3 October 2012	3.0.0 April 2013	2.3.1 February 2013
Repositories			
Maven 1 / 2	√	√	√
Ivy	√ (Maven layout only)	√	√ (Maven layout only)
Gradle	√ (Maven layout only)	√	√(Maven layout only)
NuGet	X	√(pro only)	√(pro only)
Yum	X	√(pro only)	X
P2	X	√(pro only)	√(pro only)
Repository Storage	File system	Checksum-based, filesystem or DB	File system

Feature	Apache Archiva	JFrog Artifactory	Sonatype Nexus
Metadata			
User attached custom metadata	✓	✓	✓
Searchable custom metadata	X	✓(pro only)	✓(pro only)
Schema-less metadata	X	✓	X
Attach metadata as part of deployment	X	✓(pro only)	✓(pro only)
Build-related metadata	X	✓(extended in pro)	X
Continuous Integration server metadata	X	✓(pro only)	X
Artifact management			
Snapshots purge	✓	✓	✓
Unused proxy artifacts purge	X	✓	✓
Staging releases	✓	✓	✓(pro only

Feature	Apache Archiva	JFrog Artifactory	Sonatype Nexus
CI Server plugins			
Jenkins-CI	X	✓	X
Hudson-CI	X	✓	X
JetBrains TeamCity	X	✓	X
Atlassian Bamboo	X	✓	X
Security			
LDAP authentication	✓	✓	✓
LDAP authorization	X	✓(pro only)	✓(pro only)
Single Sign On	X	Atlassian Crowd, SAML, user plugins (pro only)	Atlassian Crowd (pro only)
License vulnerabilities governance	X	BlackDuck, lightweight (pro only)	X
Custom user plugins			
Deploy plug-ins without recompilation	✓	✓	✓
Plugin dynamic DSL	X	Groovy DSL	X

ABOUT THE AUTHOR

Carlos Sanchez @csanchez is Co-Founder & Architect of MaestroDev (http://maestrodev.com), a company building a DevOps Orchestration engine for Continuous Delivery, Agile development, DevOps, and Cloud Federation. Highly committed to open source, he is a member of the Apache Software Foundation among other groups, has contributed to a variety of projects, like Apache Maven, Continuum, Archiva, Spring Security, or Fog, and regularly speaks at conferences around the world.

RECOMMENDED BOOK

Agile ALM is a guide for Java developers who want to integrate flexible agile practices and lightweight tooling along all phases of the software development process. The book introduces a new vision for managing change in requirements and process more efficiently and flexibly. It synthesizes technical and functional elements to provide a comprehensive approach to software development.

Buy Here

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. "'DZone is a developer's dream',' says PC Magazine.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome

refcardz@dzone.com

Sponsorship Opportunities

sales@dzone.com

ISBN-13: 978-1-936502-80-6
ISBN-10: 1-936502-80-1

50795

9 781936 502806

$7.95

Version 1.0