Using Repository Managers

The best way to organize, store, and distribute software components.

UPDATED BY **BRIAN FOX**
ORIGINAL BY **CARLOS SANCHEZ**

CONTENTS

INTRODUCTION

Software development depends upon two distinct kinds of components: (1) source code, and (2) binary components. This Refcard assumes basic familiarity with source control management and is intended to help you design and configure a Repository Manager to store, manage, and share binaries, optimize it for various workflows, and fit it smoothly into your DevOps pipeline.

Repository Managers are typically available as free OSS and paid professional versions. Most organizations will want to implement a paid professional version as they grow and mature their DevOps practices.

BASIC FEATURES (FREE)	PRO FEATURES (PAID)
Proxy public repositories	Enterprise support
Host local binaries or components	High availability
Group repositories for access rights and security	Component intelligence

INTRODUCTION: REPOSITORY REQUIREMENTS

A binary component is the output of any step in the development process. Many components result from builds, but other types are crucial as well. Common component types include:

- ZIP or tarball files
- RPM or DEB packages (Linux)
- JAR, WAR, and EAR packages (Java)
- npm (JavaScript)
- NuGet packages (.NET)
- RubyGems (Ruby)
- PyPI packages (Python)
- Docker Images
- DLLs (Windows)
- Source packages
- Documentation packages

THE TWO TYPES OF PACKAGES

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

WHY SOURCE REPOSITORIES?

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.: differing versions, tracking deleted or overwritten files, branching, and tagging).

WHY (BINARY) REPOSITORY MANAGERS?

Repository Managers 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. Repository managers provide one source of truth for the binaries used in waterfall, agile, and CI/CD processes.

Binary components 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 in rare situations (e.g. snapshots and nightly builds), binary components are not deleted or overwritten.


88%
are concerned about
container security.
You should be too.
SEE WHY
Sonatype

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

WHEN TO USE A REPOSITORY MANAGER

To begin, let's clarify that a Repository Manager is different than a repository. The Repository Manager is in charge of managing multiple repositories that it hosts, each with a set of specific functions and permissions. The design of your Repository Manager will help specify the role each repository serves for the development, QA/test, and operations tool and teams that access them.

The choice of how to best interact with dependencies, either in source or binary form, is usually predicated on what build system you're using. In fact, the ability to reuse binary dependencies without having to recompile them is usually a key criterion in selection of the build tool in the first place.

PLATFORM	BUILD TOOL WITH REPOS
JVM	Maven, Ivy, Maven Ant Tasks, Gradle
.NET	NuGet
OSGi	P2
Yum	Linux
Docker	Docker

While each different build tool or component format may have its own purpose built repository or Repository Manager, nearly all organizations are heterogeneous in terms of languages/ build tools and component formats. It's not uncommon to find multiple tools at play even within a single application these days. Take for example a Java application built with Maven, with a JavaScript UI with dependencies fetched from an npm repository, ultimately distributed to testing and production as a Docker container. That's three tools, formats, and repositories in a single application. Do you want to manage three separate servers, with their own idiosyncrasies, requiring backup, permissions, etc.?

A Repository Manager is a hub for development teams across the whole organization, centralizing the management of all the components generated and used by the organization. The inevitable resulting diversity of component types, and their differing positions in the overall workflow, is one major reason to use a dedicated Repository Manager, rather than just a simple file server.

Thus, the decision to use a Repository Manager generally revolves around how many repositories you need, what types of component formats are dictated by those repositories and build tools, and what higher level functionality you need above the bare minimum repository. The common factors to consider are discussed below.

FACTOR 1: THIRD-PARTY BINARY MANAGEMENT

PROXYING BINARIES

At some point, in most projects, you'll use third party components that are hosted in a repository external to your organization. Network latency and bandwidth will affect development speed directly – especially when your external components are (in some cases, gigantic) binaries – even if your team is fully on-premises.

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 component download dependency hell.

Further, external dependencies introduce an element of unnecessary risk simply because you can't control access to them. To remove this risk, configure your Repository 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 components even if the upstream repository goes down, or they disappear on the upstream repository. For those of you who remember "npm gate" in 2016, this became a real-world issue for thousands of development teams.

At its most basic essence, a Repository Manager is a caching proxy of these remote repositories. The cached components can be served rapidly to other machines on the same network after the initial request – either to human coders or directly to Continuous Integration (CI) servers themselves. The ability to cache the things you need locally isolates you from the inevitable network latency, internet connectivity issues, or components randomly disappearing from improperly managed remote repository ala "npm gate".

ADVANCED MANAGEMENT OF THIRD-PARTY COMPONENTS

Some organizations may have a policy about what third-party dependencies may be used because of licensing or security concerns.

Here's a common example: third-party components need to be requested by a developer and approved by a legal department. Frequently we see attempts to manage this process by a simple whitelist / blacklist approach. This is bound to fail for several reasons:

- The breadth and volume of third-party components is staggering. A typical enterprise can easily consume many hundreds of thousands of components that each release

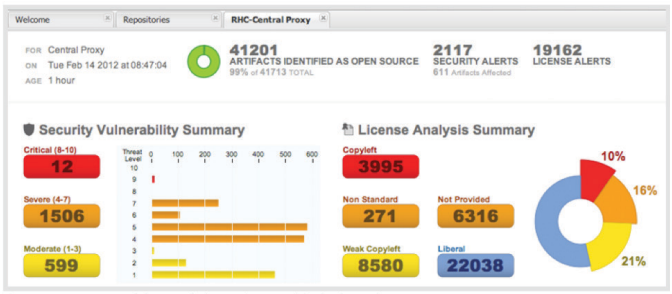
new versions four times a year on average. Having humans review each one is simply impossible.

- Delays in human reviews of the list cause developers to do one of two things: 1) stop updating dependencies because the friction is too high, leading to increased exposure to vulnerabilities over time and making it harder to upgrade later, or 2) work around the system to get their job done, leading to decreased visibility of what is actually in use and ultimately defeating the entire purpose of the process.

Fortunately, there are ways to deal with this that don't have unintended side effects.

COMPONENT INTELLIGENCE

Some professional versions of Repository Managers include health checks to provide instant insight into potential component security, license, and quality risks so that development teams can take corrective action early and quickly. This intelligence can help organizations identify known security, license, and architectural issues for each component. Health check capabilities can be used as an automated audit tool for build managers, architects, open source governance, security, and legal professionals.



COMPONENT FIREWALL

Building upon the component intelligence, some Repository Managers can provide a form of Firewall capability. It becomes possible to automate the decisions of what components to allow into the organization by using the intelligence and combining it with a rules engine.

This allows you to stop bad components (e.g. ones with already existing known vulnerabilities, or ones with licenses incompatible with your business model) from being proxied and integrated only to be ripped out later. This model is the only one that can scale without requiring an army of human reviewers.

Further, components that are known to be good when they are first used become bad later when new vulnerabilities are discovered. In a manual review process, almost no organization can keep up with new requests and thus no one ever goes back to check the things previously requested and approved.

The automated rules engine and continuously updated Component Intelligence can alert you when these components go bad so that your developers can immediately triage and remediate the problem.

FACTOR 2: INTERNAL COMPONENTS

In addition to consuming third-party components, most modern build tools also need a location to push the artifacts of the build to a repository. This is done because the internal artifacts themselves are often sub-assemblies or otherwise dependencies of yet another build. This makes hosting internally developed components an equally important capability of a Repository Manager.

There are several factors to consider when structuring your internal hosted repositories. You'll want to partition things into different repositories. Doing this effectively requires you to balance ease of administration by not having too many repositories vs. challenges like security (covered later) as well as partitioning by use case.

A typical use case worth partitioning is for temporal components. In Maven, these are formalized as "snapshots" and are required to be separated from "releases". In other tools, the separation isn't baked in. Administration and cleanup will be easier if you keep the components that are constantly churning out short-lived versions separate from the ones you intend to keep for a long time or forever. This allows easier purge policies (discussed later), as well as optimizations in how you store the components on disk that allow easier backups of the permanent things, less block level fragmentation, and other I/O-related concerns.

COMPONENT STAGING

Another common use case for internal components is to manage them through a staging and promotion lifecycle.

When a component is pushed to a repository, that repository may not be its final destination. Imagine a workflow where a release candidate component needs to go through integration testing and QA processes. Only components that go through this process should be available for other teams or clients.

A Repository Manager can enable this workflow by providing mechanisms to associate components and promote them through various phases, where each phase may result in them being available to different users on different known URLs.

This type of functionality is often done in conjunction with an automated CI/CD pipeline that is discussed in more depth later.

FACTOR 3: SECURITY AND MAINTENANCE

AUTHENTICATION AND AUTHORIZATION

Since the Repository Manager 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, Active Directory, or single sign on servers (SSO).

As with source traceability, binary traceability is equally important. Track changes in the repository (such as which user uploaded a component and when, or who is downloading components) for audit purposes.

PURGING POLICIES

Although most components are usually 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 reasonable disk usage—especially when using Continuous Integration heavily, since CI can easily generate dozens of new builds per day. Usually, snapshots can be purged when a new version is released, but that may be changed to just keep the most recent snapshots.

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

SUPPORTING DISTRIBUTED TEAMS

When teams that access the repositories are located in different locations or distributed across the globe, it is also important to provide access to all the components, both internal and third-party. Recall that the basic essence of a Repository Manager is a caching proxy and therefore to reap the benefits, you really want to have one located in each physical location where you have more than a few developers. Otherwise those developers may suffer slow and unreliable build times because they are fetching components from the internet and/or across the WAN.

In some cases, you will want to pre-emptively replicate some content to another location, either to prime the repo for developers, or to achieve some level of HA or DR. Some Repository Managers will force the complexity of replication onto the admins by requiring that you configure point-to-point mirroring, often for each repository individually. Other solutions separate the notion of what components need to be exposed in what logical repositories from the notion of where and how they are located geographically so that the tool can do the work intelligently and dynamically.

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. In a CI/CD environment, when a Repository Manager is not available, a build cannot execute nor deploy to production, which could be disastrous to the business or organization.

Today, advanced Repository Managers use a private binary cloud storage and backend for all components. This component fabric intentionally decouples the physical node topology from the logical component topology. Any component deployed or proxied on one Repository Manager is immediately available to all others since the component fabric shares the knowledge about new components and their metadata - no custom setup is required to replicate components between repositories.

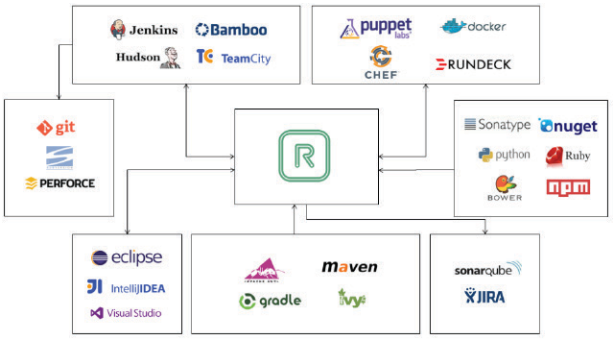
DISASTER RECOVERY

The component fabric is also a critical piece in disaster recovery. In the case with a loss of network connectivity to a data center, the component fabric stores all data in separate nodes in other datacenters, so Repository Manager requests can be directed to other Repository Managers. When the network issue is fixed and the data center comes back online, any new components and data in the fabric are automatically synced with the datacenter that is now available again.

FACTOR 4: REPOSITORY MANAGERS IN THE DEVOPS TOOLCHAIN

Repository Managers have become integral to the DevOps pipeline and are included in almost every reference architecture found in organizations around the world.

Any component or build artifact that is produced or needed in the CI/CD process is stored in a Repository Manager. Repositories are integrated to Jenkins, Maven, Gradle, Puppet, Chef, and almost every other tool in the DevOps toolchain. Rundeck, for example, orchestrates the deployment of applications to production and relies on a Repository Manager to get the components it needs for deployment. Repository Managers are central and critical to implementing modern DevOps environments.



CI/CD PIPELINES

WHAT IS CONTINUOUS INTEGRATION?

With the advent of lean, agile, and more recently, Continuous Delivery and DevOps, projects no longer incubate for months in a waterfall-like development process. Instead, they undergo constant changes and releases. In many cases, these projects are distributed throughout an organization at different stages of the development lifecycle.

This always-on development means the volume of changes alone necessitates an automated approach to building applications. To support developers at scale, several tools have been developed to automate much of what goes into building and releasing applications. This includes the ability to package applications or components generated during build, and do so continuously as changes are made. Colloquially this is referred to as Continuous Integration, or CI.

PIPELINE

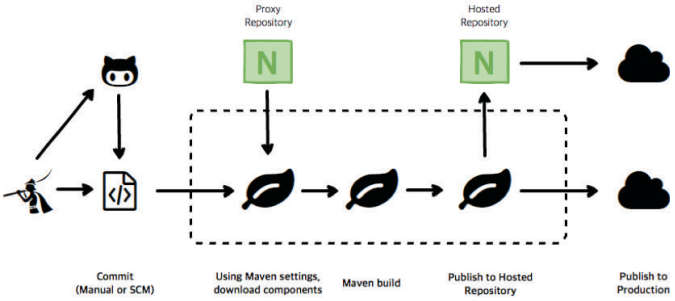
Building on the automation benefits from CI servers, teams are now able to completely customize what goes where, when, and how. In other words, teams add in various checkpoints throughout the process to ensure applications are free of major defects. This isn't merely quality assurance anymore, but rather governance to avoid vulnerability, license, and architectural issues in the applications and components a team produces. The result is a pipeline-like approach.

REPOSITORY MANAGERS AND CI TOOLS

Repository managers provide several interaction points with CI tools like Jenkins and Bamboo. This can range from simply requesting and storing a proxy of components as part of the build process, to publishing internally developed components for distribution across a development organization.

BUILDING PROJECTS

When an application is built using Maven (or similar build tools), it gathers the components from a specified location (configured via Maven settings) and compiles them. The end result can be an application, or even another binary or component. Jenkins provides automation for this by allowing the inclusion of a Maven build step for freestyle and multi-configuration projects. When the Build step is called, the Maven project will build, and if Maven has been configured to do so, it can request components from a Repository Manager. In a similar fashion, using Maven's Deploy goal (Maven has goals such as clean, compile, test, package, install, deploy all of which are managed by plugins) components can be published as a build step to a desired location. It's important to note that this is Maven functionality and not something unique to Jenkins or any other system.

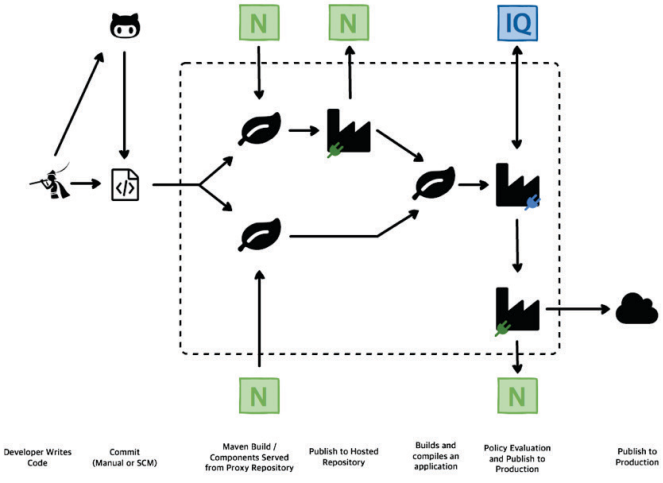


PUBLISHING PROJECTS

For teams moving into more advanced, continuous delivery and DevOps (Pipeline) models, Jenkins provides support for pipeline-style projects. This becomes less about simply building or compiling projects (à la Maven or similar tool), and more about automating tooling to take necessary action and/or get things where they need to be throughout the development lifecycle.

For example, it could be building a Maven package and making sure it's published to a repository for testing, then removed once testing is complete. In more complex environments it likely means multiple builds are taking place simultaneously, and in parallel, then assembling each build together in a single package that is passed on through to staging and eventually into production. In some instances, the product of those builds is moved forward. In others, once it's no longer needed it's automatically removed.

The impact here is that it decouples the publishing process from the compilation tool, allowing greater customization and for components to be passed to the Repository Manager at any point in the development lifecycle, regardless of the development ecosystem.



CONTAINER CONSIDERATIONS

Docker containers and their usage have revolutionized the way applications and the underlying operating system are packaged and deployed to development, testing, and production systems. Docker Hub is the public registry for Docker container images and it is being joined by more and more other publicly available registries such as the Google Container Registry.

In many ways, a container is just like other components, with a few key differences:

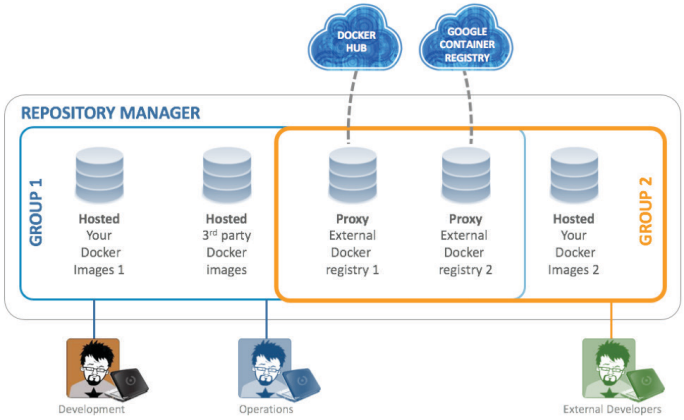
1. They can be huge. A Java component might be a few hundred kilobytes, but a container image can be many gigabytes or larger. This can put a strain on the underlying disk infrastructure if size and volume is not anticipated.

2. They are comprised of many layers. Each layer is effectively a file system delta of changes upon the layers below it. This makes it interesting because all but the lowest level layer are unable to stand alone.

Repository Managers also offer support for Docker containers. The Repository Manager acts as a private Docker registry that is capable of hosting proprietary containers as well as proxying the public registries when non-proprietary containers need to be downloaded. You can expose these Docker repositories to the client-side tools directly or as a repository group, which is a repository that merges and exposes the contents of multiple repositories in one convenient URL.

This allows you to reduce time and bandwidth usage for accessing Docker images in a private registry as well as share proprietary images within your organization in a hosted repository. Users can then launch containers based on those images, resulting in a completely private Docker registry with all the features available in the Repository Manager.

Think of the Repository Manager as a heterogeneous location to store and manage your Docker images, open source software components, and other build artifacts. By comparison, other private container registry solutions act as homogeneous repositories.



ABOUT THE AUTHOR

BRIAN FOX is the co-founder and CTO of Sonatype, and is also a member of the Apache Software Foundation and former Chair of the Apache Maven project. As a direct contributor to the Maven ecosystem, including the maven-dependency-plugin and maven-enforcer-plugin, he has over 20 years of experience driving the vision behind, as well as developing and leading the development of software for organizations ranging from startups to large enterprises. Brian is a frequent speaker at national and regional events including Java User Groups and other development related conferences.

BROUGHT TO YOU IN PARTNERSHIP WITH Sonatype

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com