Binary Repository Manager

White Paper

September 11, 2020

Abstract

A software project of almost any size needs to keep its binary artifacts in a repository, to enable access to them by programmers, tools, and other teams. The quality of the software that manages the repository matters. There are a few categories of such a software, which have their pros and cons, currently on the market. However, none of them fully satisfy the requirements of a large group of software companies. That's why a new product is being created.

1 Introduction

Binary Repository Manager (BRM), according to Wikipedia, is "a software tool designed to optimize the download and storage of binary files used and produced in software development," for example `.jar` or `.zip` archives. BRM is a critical component of most DevOps toolchains (Erich 2018), residing right after the build pipeline, which is why it is sometimes called "build repository", "artifact repository", or "pipeline state repository" (Bass et al. 2015).

A traditional DevOps pipeline, as explained by Humble et al. (2010), expects the source code to be validated, tested, packaged and versioned automatically into an *artifact* (a binary file). Then, the artifact must be stored outside of the source code repository and become available for later stages of the continuous delivery pipeline. The BRM is supposed to host these artifacts, being “a central point for management of binaries and dependencies, and an integrated depot for build promotions of internally developed software,” as noted by Davis et al. (2016).

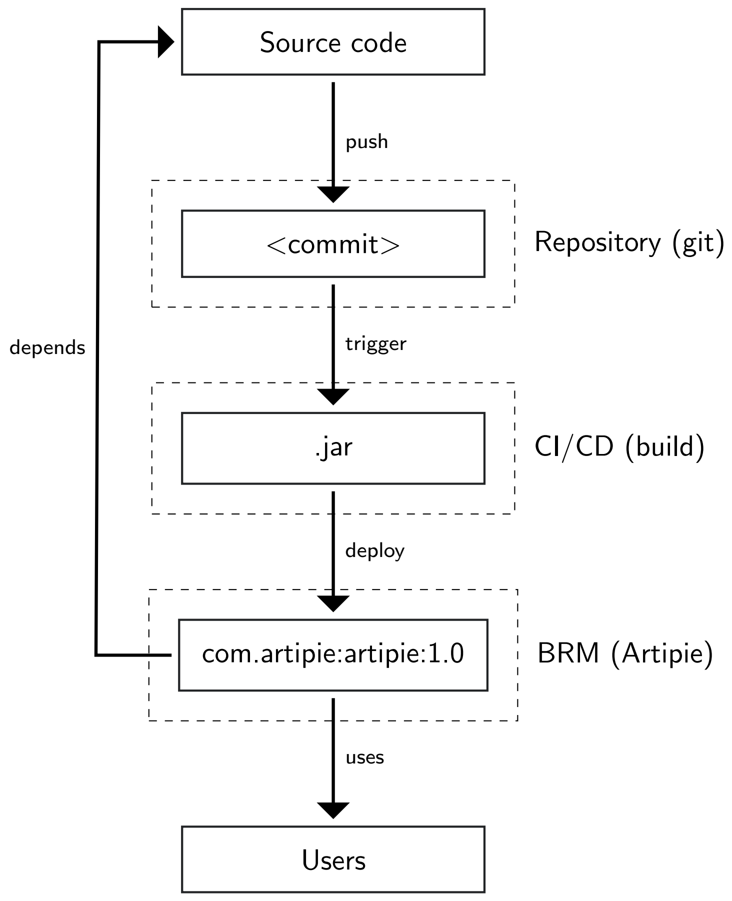


Figure 1: The deliver pipeline of an average software development team.

The Section 2 lists a few categories of existing BRM solutions, analyses requirements their customers may have, and emphasize the most important functional features and non-functional requirements.

2 Requirements

All existing BRM solutions can be categorized as public, commercial, hosted, open source, or surrogate. Even though each of them partially satisfy the needs of a professional software team, neither one is perfect.

Public There are a few hosted BRMs for different programming languages, like Maven Central for Java or Rubygems for Ruby, which are free to use, but do not allow private accounts. This means that all artifacts deployed by some user become available for all other users. This business model is acceptable for open source projects, but is not suitable for software teams that develop proprietary software products.

Commercial There are a few BRMs, like Artifactory/Bintray of JFrog and Nexus of Sonatype, which provide most of the features required by software teams, including fine-grained access control, versioning, seamless integration with build automation software, and many more. However, these tools are pretty expensive[1] and require certain skills to install and manage them. Moreover, their authors (including JFrog and Sonatype) are US-based companies, who may be restricted to sell their products to software teams from certain "sanctioned" countries[2].

Hosted There are a few BRMs, which maintain artifacts on their servers, like CloudRepo for Java or PyDist for Python. Some BRM creators, like JFrog, provide their products in hosted versions too. However, some software teams may not find this option acceptable due to security reasons—eventually the data may be lost, if the company gets out of the market[3] or due to sanctions.

Open source There are also a few entirely free and open source products, like Archiva, which software teams must install, configure and use on

[1]The annual cost of a license for a mid-size team of 50-100 developers is: around $30,000 for Artifactory and around $50,000 for Nexus. There are less expensive products too: ProGet for $10,000,

[2]29th of July, 2019: GitHub, the world's largest host of source code, is preventing users in Iran, Syria, Crimea. 22nd of December, 2018: Slack confirms it will now block all activity in Iran and other sanctioned countries.

[3]13th of March, 2015: Google Code, one of the largest source code repository managers, closed its doors.

their own risk. Even though this may sounds like a good solution for a small team, it may not be acceptable for a larger group of software developers, who expect their artifact repository to be reliable and available.

Surrogate It is possible to organize a BRM without any software, for example, on top of Amazon S3 or a simple FTP server. With the right plugin Maven can deploy to Amazon S3 and then fetch artifacts from there via their built-in HTTP interface. However, such a solution gives very little or no control for a DevOps person and may only work for rather small software teams.

2.1 Features

There are many important qualities and features software developers and DevOps engineers expect a BRM to have, in order to be useful in a continuous delivery pipeline. The most critical non-functional requirements are:

Integrability There are plenty of build automation tools for each programming language, like Maven for Java, Npm for JavaScript, or Rake for Ruby. There are also many continuous integration tools, like Jenkins or Travis. Since automation is the most important aspect of DevOps, as noted by Kerzazi et al. (2016), it is expected to have plugins for each or most of them, to enable seamless intregration with the BRM.

Availability Artifacts are important components of a software development process and they must be available right when they are needed by a programmer or a build tool, without even small delays and delivered at the highest possible speed.

Scalability Most build artifacts are large binary files. Some of them may even be larger than 1Gb, for example Docker images or `.war` production-ready Java archives. The BRM must be able to maintain large data sets, up to almost no limits.

Extensibility It is highly desireable to have full access to the source code of the BRM and to have an ability to extend it with new plugins and modules. Moreover, vendor independence is important.

Reliability An ability to corrupt the data due to software or harware failures must be eliminated, as much as it is possible.

2.2 Non-functional Requirements

The most important functional requirements are:

Versions and Tags New artifacts must not replace previously deployed ones. Instead, older versions must always be accessible. However, it is not expected that the BRM would assign version tags automatically, this is done at the pipeline's side.

Access Control Larger teams may need to control who is allowed to use certain artifacts. Moreover, such teams may need to require the integration of authentication mechanisms of the BRM with the existing enterprise access-control system, via LDAP, for example. On top of regular access control, encryption mechanisms must be in place in order to prevent data leakage in case of software/hardware failures or human mistakes, as noted by Paule (2018).

Analytics Traceability between software artifacts is considered a very important factor in today development process, as noted by Palihawadana et al. (2017). BRM must make it possible to visualize dependencies between artifacts and operate on them.

There are many other essential features required, including authentication and authorization, deployment, publishing, download, removal, usage tracking, email notifications, mirroring, and so on.

2.3 Compare with existing solutions

Feature	Artipie	Artifactory	Nexus
Cloud storage providers	S3 based API clouds	AWS S3, Google Cloud, Azure	No
Supported repository type	Maven, RPM, NPM, Docker, NuGet, PHP-composer, binary files, PyPi, Go, Gem	Bower, Chef, CocoaPods, Conan, Conda, CRAN, Debian, Docker, Git LFS, Go, Helm, Maven, NPM, NuGet, Opkg, P2, PHP-composer, Puppet, PyPi, RPM, Gem, SBT, Vagrant, VCS	Bower, Docker, NPM, PyPI, Raw, RubyGems, Yum*
Installation and maintainance	Easy to install via Docker image or to cluster	Can be installed locally, but require a team of system administrators to support the cluster	Same as Artifactory
Performance	TODO	TODO	TODO

* Nexus can uses external plugins to support more repository types

2.4 Expected Metrics

In a large enterprise it is expected to have the following numbers, in terms of load, size, and speed:

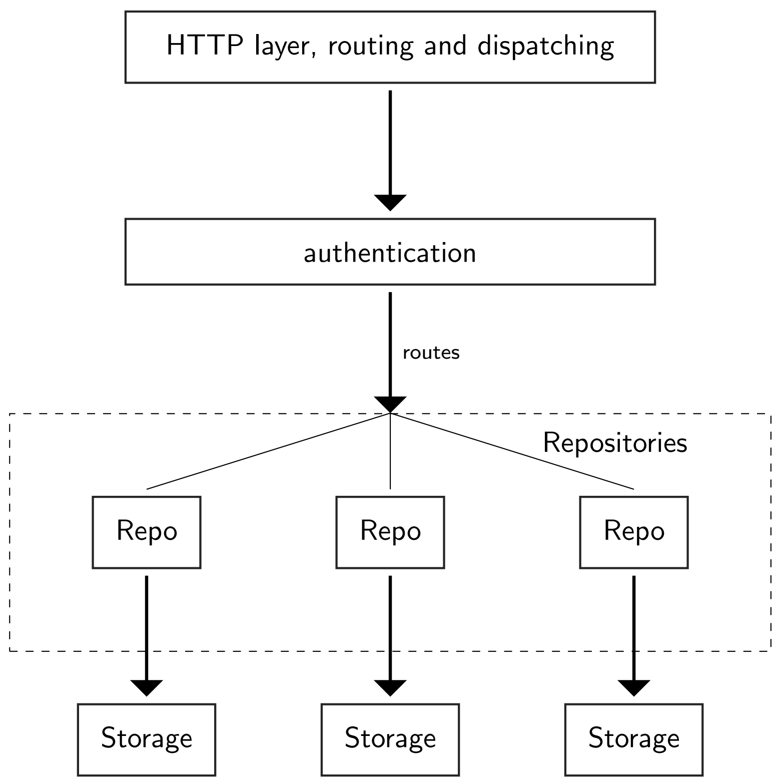
Users, total	80K
Artifacts hosted	100M
New artifacts uploaded, daily	10K
Data hosted	100Tb
Data uploaded, daily	10Gb
Concurrent connections, peek	10K
Upload bandwidth, peek	10M/s
Download bandwidth, peek	100M/s

Smaller companies may have lower expectations.

3 Architecture

Architecure is consisted of 4 essential parts:

1. Artipie HTTP engine
2. Authorization and authentication layer
3. Repositories
4. Storage

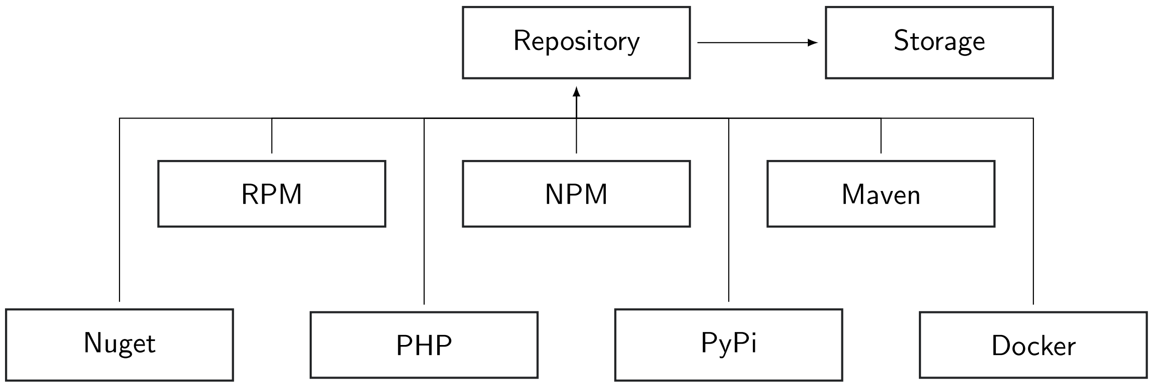


3.1 Design considerations

All of the Artipie components are based on reactive, asynchronous, non-blockng and back-pressured streams and asynchronous, reactive and non-blocking programming principles, allowing Artipie to withstand heavy work-loads with a small amount of kernel threads.

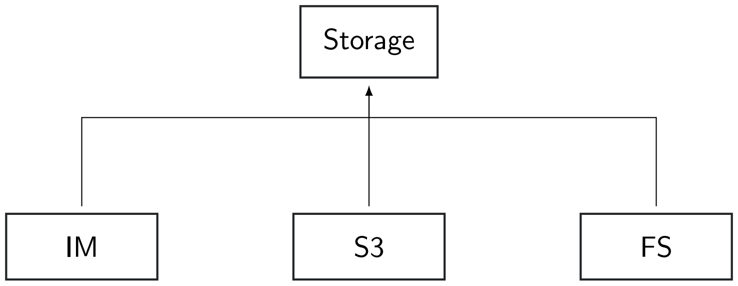
3.2 Artipie engine

Artipie engine is a Java application, which exposes an HTTP endpoint for repository access and management operations. It routes HTTP requests to repositories and provide authentication mechanisms for repositories. Each repository encapsulates storage API to access binary blobs and metadata files.

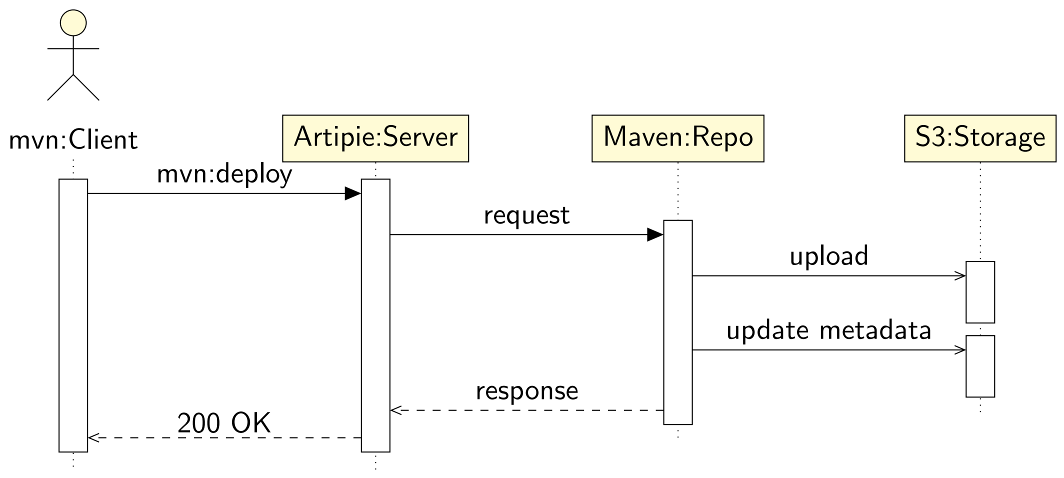


There are various storage implementations, such as:

1. File system storage
2. S3 based storage
3. In-memory storage



The common data flow for Artipie upload: client is sending some binary artifact to the server, server find responsible repository to process the request, repository is saving the stream to the storage, and after complete it updates metadata of the repository (it's most common scenario, some repositories works differently, e.g Docker uses metadata as path).



Engine store repository configuration in a .yml file. An example:

```
repo:
  type: maven
  storage:
    type: s3
    url: s3://acme.com/snapshot
    username: admin
    password: 123qwe
```

This configuration file says that it has a type of maven repository, which automatically enable maven specific metadata generation logic. Diving into storage section, artipie asked to use S3 object storage as a storage for uploaded artifacts and generated metadata.

The ability to chose where to store artifact gave us the flexibility of choice. We can choose any type of storage, whether it is a server file system, an object storage, or a key-value database. The only requirement is: Abstract storage should support it.

Aside from repository configuration, the way artipie stores its settings also can be customized via artipie.yml file:

```
meta:
  storage:
    type: fs
    path: /artipie/storage
```

Artipie will resolve this file, and use local filesystem folder for repository settings management.

3.3 Repository adapters

Repository adapters are independent projects, aimed to implement meta information generation layers for a specific package type(npm, maven, etc...). Artipie engine utilizes adapters in order to provide BRM functionality.

Existed adapters:

- RPM - artipie/rpm-adapter
- NPM - artipie/npm-adapter
- Go - artipie/go-adapter
- Docker - artipie/docker-adapter
- Maven - artipie/maven-adapter
- Gem - artipie/gem-adapter

3.4 Abstract storage

Abstract storage(asto) is an abstraction over physical data storage system. It has a simple interface consisted of two operations: put and get. The simplicity makes it easy to implement the interface of almost any data storage system.

Those design requirements were considered as most important for the asto:

1. High performance
2. Back pressure of data streams on the level of bytes
3. Constant memory pool per data stream

4. Pure java interface, without any external dependencies
5. High operation latency awareness

The following design options has been considered for the interface design implementation:

- `java.io.{In,Out}putStream`'s based option.
- RxJava 3 based option.
- CompletableFuture and Java 9 Flow based option.

The `java.io.{In,Out}putStream`'s approach has a conceptual drawback: it's blocking nature, which affects performance by forcing new thread allocation per user connection. And that is also a reason why any other blocking approach was not considered.

The RxJava 3 option is close to the ideal one, but the negative side is external dependency exposition: clients are getting bounded to the RxJava primitives.

Java non-blocking primitives were counted as the most promising ones, since `CompletableFuture` and `Flow`-based interface can be implemented in a high-performance way and with accordance with all the mentioned requirements.

3.5 Extensions

to be written...

4 Conclusion

To be written...

4.1 Acknowledgements

The document was originally created by Yegor Bugayenko (y00538675).

References

Bass, Len et al. (2015). *DevOps: A software architect's perspective.* Addison-Wesley Professional.

Davis, Jennifer et al. (2016). *Effective DevOps: building a culture of collaboration, affinity, and tooling at scale.* O'Reilly Media, Inc.

Erich, Floris (2018). "DevOps is Simply Interaction Between Development and Operations". *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment.* Springer, pp. 89–99.

Humble, Jez et al. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader).* Pearson Education.

Kerzazi, Noureddine et al. (2016). "Who Needs Release and Devops Engineers, and Why?" *Proceedings of the International Workshop on Continuous Software Evolution and Delivery.* CSED'16. Austin, Texas: ACM, pp. 77–83.

Palihawadana, S et al. (2017). "Tool support for traceability management of software artefacts with DevOps practices". *2017 Moratuwa Engineering Research Conference (MERCon).* IEEE, pp. 129–134.

Paule, Christina (2018). "Securing DevOps: detection of vulnerabilities in CD pipelines". MA thesis.