



CUSTOM CLEANUP STRATEGIES 101

By: David Robin,
Senior Solution Engineer, JFrog

Copyright © 2023 JFrog Ltd.
Published: November 2023



Table of contents

- What is a Cleanup Strategy? 3
- JFrog Platform Cleanup Challenges 3
- Reminder Repository Management Best Practices 4
- Setting Up a Fine-Grained Cleanup Strategy 5
 - Overview 5
 - Example of Find Repositories 6
 - Example of Get Repositories to Clean 6
 - Example of a Cleaning Query 7
 - Example of Handling Docker Images 8
 - Community Alternatives 9
- Conclusion 10

What is a Cleanup Strategy?

A "cleanup strategy" in software refers to a set of actions or processes that are implemented to ensure the proper and efficient management of resources and data within a software system. It involves identifying and removing unnecessary or redundant resources, freeing up memory, and maintaining the overall health and performance of the system.

Cleanup strategies are commonly used in various areas of software development, such as file and database management, network connections, and temporary data handling. They help prevent resource leaks, optimize system performance, and ensure the smooth operation of the software.

Cleanup strategies may include:

- Database cleanup
- Network cleanup
- Temporary data cleanup

Implementing an effective cleanup strategy is crucial for ensuring the stability, reliability, and efficiency of software systems, especially in long-running applications or systems with limited resources

JFrog Platform Cleanup Challenges

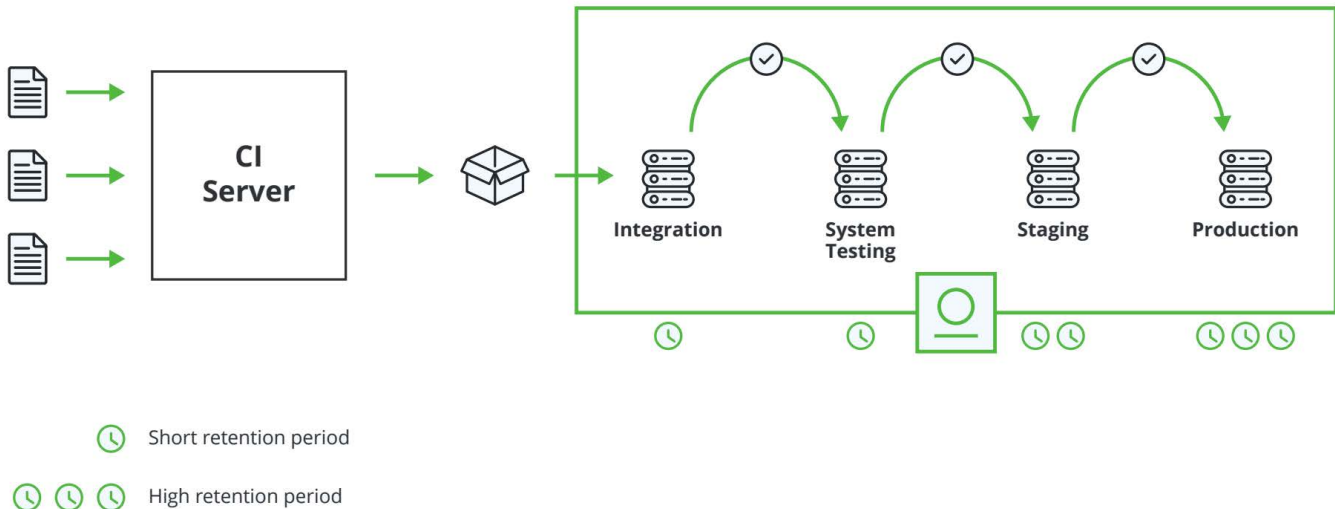
When it comes to the specific challenges of the JFrog Platform, let's focus on disk space usage and data cleanup. Out-of-the-box tools are provided for implementing a classic cleanup strategy, to optimize efficiency and gain control of the filestore's size.

In some cases, however, that may not be enough, as organizations often have specific needs regarding artifact retention, or limitations due to legacy repository systems. In such cases, the default controls that we provide through the administration UI, or the Artifactory cleanup plugin may not be sufficient.

This document goes beyond the basics, by presenting several tools that the JFrog Platform provides that help to implement a customized solution according to each organization's cleanup strategy.

Reminder Repository Management Best Practices

Prior to creating a repository, it is recommended to use JFrog Project which is available as an option of the JFrog Platform subscription. This enables you to decentralize the management of artifact cleanup by establishing disk space quotas specific to each team, which will require each team to take responsibility and implement retention policies that are suitable for their usage.



Mixing Promotion and Retention Policy Concepts

It's also recommended to combine the promotion principle with retention settings at the repository level to ensure that teams only keep builds that are truly stable or have been deployed at least once in a given environment.

Edit Project Davidro

* Project Name * Project Key ?

Description

Storage Quota

☐ Unlimited

☒ Notify When Reaching GB ?

Project Admins Privileges

☒ Manage Resources ☒ Manage Members

☒ Index Resources

JFrog Project's "Storage Quota" Feature

For more information, see the JFrog [Build Promotion](#) and [Project](#) documentation.

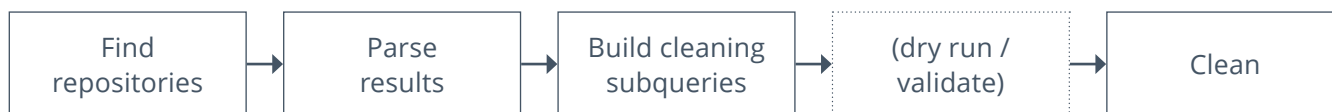
Setting Up a Fine-Grained Cleanup Strategy

In some cases it is not possible to implement a "global" cleanup strategy similar to the plan suggested in our cleanup user plugin example. Whether that is due to limitations of legacy systems or some other reason, the result may be that some of the repositories are in a state where global rules cannot be leveraged to regularly clean artifacts using JFrog's out-of-the-box tools. In such cases, it is recommended to implement an automated custom cleanup mechanism, based on controlling the purging behavior of artifacts at the repository level.

The main idea is to control behavior through a set of properties placed on the repositories to be cleaned, and develop a cleaning pipeline that runs regularly based on those settings.

Overview

In this screenshot, the "vacuum" property is used to identify the repositories that need to be cleaned.



Example of a scheduled cleaning pipeline

Two other properties, "vacuum-pattern" and "vacuum-retention," can be used to customize the cleaning strategy for that repository by specifying:

- A pattern to follow for the name of the artifacts to be deleted
- A delay in days to remove artifacts created earlier

The screenshot shows the 'Properties' tab for a repository named 'dro-cleanup-demo-local'. It features a search bar for 'Property name' and a 'Recursive' checkbox. Below, a table lists three properties: 'vacuum' (value: 'on'), 'vacuum-pattern' (value: '*.png'), and 'vacuum-retention' (value: '60d').

Property	Value(s)
<code>vacuum</code>	on
<code>vacuum-pattern</code>	*.png
<code>vacuum-retention</code>	60d

The "vacuum" property identifies which repositories to clean

Example of Find Repositories

Let's use the JFrog CLI and its `"rt search"` method to demonstrate how the first step of "Find Repositories" can be implemented.

This command can take a `FileSpec` containing an AQL query as a parameter. In this example, the AQL query selects all repositories (level 0 folders) that have the 'vacuum' property with the value 'on'.

```
{
  "files": [
    {
      "aql": {
        "items.find": {
          "type": {"$eq": "folder"},
          "depth": {"$eq": "0"},
          "@vacuum": {"$match": "on"}
        }
      }
    }
  ]
}
```

get-repositories-to-clean.json

This allows identifying the repositories to be purged and retrieving the configuration parameters specific to the current repository. In this particular example, it is based on the value of the vacuum-pattern and vacuum-retention properties.

Example of Get Repositories to Clean

Using the `get-repositories-to-clean.json` file with the JFrog CLI requires the following input:

```
`jf rt s --spec ./get-repositories-to-
clean.json | jq '. |
map({"path":.path,
"props": .props})`
```

This will produce the following output

```
[
  {
    "path": "my-backend-maven-local/",
    "props": {
      "vacuum": [
        "on"
      ],
      "vacuum-retention": [
        "3m"
      ]
    }
  },
  {
    "path": "my-cleanup-demo-local/",
    "props": {
      "vacuum": [
        "on"
      ],
      "vacuum-pattern": [
        "*.png"
      ],
      "vacuum-retention": [
        "60d"
      ]
    }
  }
]
```

Using the results of this query, it is then possible to perform cleaning operations repository by repository with a dynamic configuration.

Example of a Cleaning Query

By iterating over the results obtained in the previous step and using information from the properties, we can now construct a cleaning query like the one below:

```
{
  "files": [
    {
      "aql": {
        "items.find": {
          "repo": "my-backend-maven-local",
          "created": {
            "$before": "6mo"
          }
        }
      }
    }
  ]
}
```

clean-repository-query.json

And then execute it:

```
`jf rt delete --spec ./clean-repository-query.json --quiet=true`
```

clean-repository-query.json

Note: This command has a dry-run option to visualize the files before they are actually deleted.

Example of Handling Docker Images

Docker images are common in most repositories and require special treatment. Given the "composite" nature of a Docker image, which contains one or more manifests with their associated layers, the approach is slightly different for the previous query.

First, it is necessary to locate the manifests that need to be deleted. For multi-architecture Docker images, this can be done by locating all files named "manifest.json" or "list.manifest.json", and then deleting all the files, including the manifests and layers, located in the parent directory of the manifest.

From this query, retrieve the parent directories by searching for the attribute "path" to create a subset of the results.

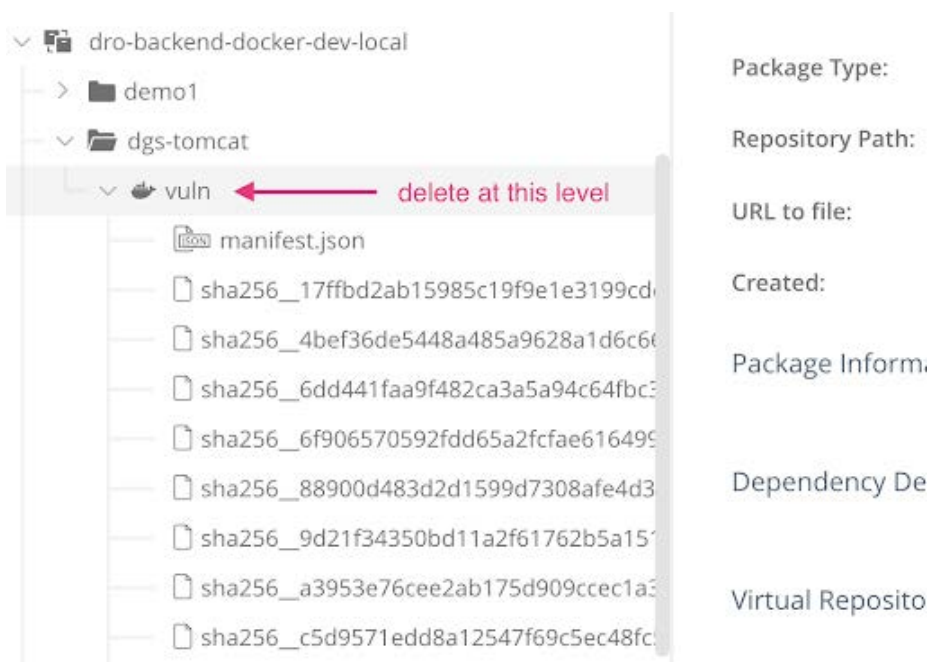
Here is an example of a query that identifies the parent directories of manifest.json / list.manifest.json associated with images to be deleted.

```
items.find({
  "repo": "dro-backend-docker-dev-local",
  "$or": [
    {"name": "manifest.json"},
    {"name": "list.manifest.json"}
  ],
  "created": {"$before": "3mo"}
}).include("path")
```

find-docker.aql

You can execute this query with:

```
jf rt curl -XPOST -T ./aql/find-docker.
aql "api/search/aql"
```



Note: The example folder above corresponds to the "tag" level in the Docker hierarchy.

Sample output should appear as below:

```
{
  "results" : [ {
    "path" : "multiarch/dgs-
graphql/1.40.0"
  }, {
    "path" : "multiarch/dgs-
graphql/1.41.0"
  }, {
    "path" : "multiarch/dgs-graphql/sha
256:6824845c0ed8c8e312dd09cac30
f65480e9ce5652770d8ad2f272da7cf
a4e033"
  }, {
    "path" : "multiarch/dgs-graphql/sha
256:687364eeb74d4462d13c154429
e2aa4d8eda9ffe59297defcdc08582c
11aa1c7"
  }, {
    "path" : "multiarch/dgs-graphql/sha
256:a0dc1980c741bea95f72e2779dc
36da1dbcc40055accb6ae864d39298
5e6cf85"
  }, {
    "path" : "multiarch/dgs-graphql/sha
256:a2624d8535fbff9015d15eef991
59140e70f97c9c7355914ed300311fc
5cd98d"
  }, {
    "path" : "multiarch/dgs-graphql/sha
256:b69d0479a8426f7345f7249c1de
0212b30334ae11d79319616cfa333e
f118c85"
  }, {
    "path" : "multiarch/dgs-graphql/sha
```

```
256:c31d2ef3f0d2de7be55bbe092d3
7971dfa80be1f08fefee4b2ee34e511
29affd"
  } ],
  "range" : {
    "start_pos" : 0,
    "end_pos" : 8,
    "total" : 8
  }
}
```

These resulting paths can then be deleted using the Docker V2 REST API through the JFrog CLI.

```
jf rt curl -XDELETE "/dro-backend-
docker-dev-local/dgs-tomcat/
vuln"
```

For more information, see the JFrog [Docker Registry](#) documentation.

Community Alternatives

Even though there are open source tools which are not developed or officially supported by JFrog, they can still be leveraged using our standard open source connectivity. Many of these tools are actively maintained and offer a rich set of features for implementing fine-grained

cleanup strategies using the REST API and JFrog CLI interfaces. Two good examples include the [DevOpSHQ](#) and [Crazy Max](#) packages.

Conclusion

The best way to make sure that software systems are running at optimal efficiency is to have a cohesive Cleanup Strategy. This offers several benefits that contribute to the overall quality and performance of the system, including key benefits such as:

- Resource optimization
- System stability
- Performance optimization
- Maintenance and troubleshooting
- Scalability and resource planning

A comprehensive cleanup strategy is crucial for maintaining the health, stability, and performance of software systems. Regarding the JFrog Platform, leveraging our out-of-the-box tools enables implementation of a well-designed cleanup strategy, to ensure all aspects of the JFrog Platform continue to run smoothly, efficiently, and reliably.

For more information, [request a demo](#) by the JFrog Solutions Team to see how our tools make it easier than ever to implement a comprehensive Cleanup Strategy for the JFrog Platform.

