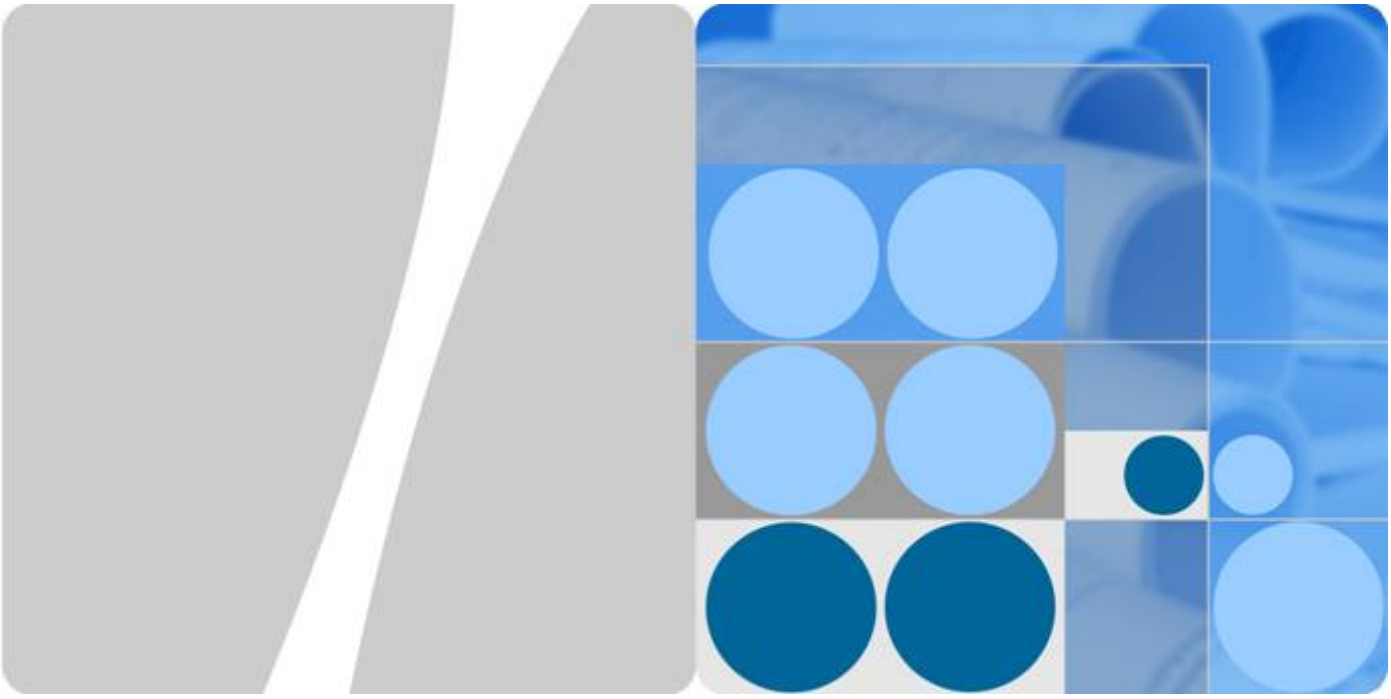


CodeArts Artifact

User Guide

Date **2026-01-12**

Contents

1 Service Overview

1.1 What Is CodeArts Artifact?

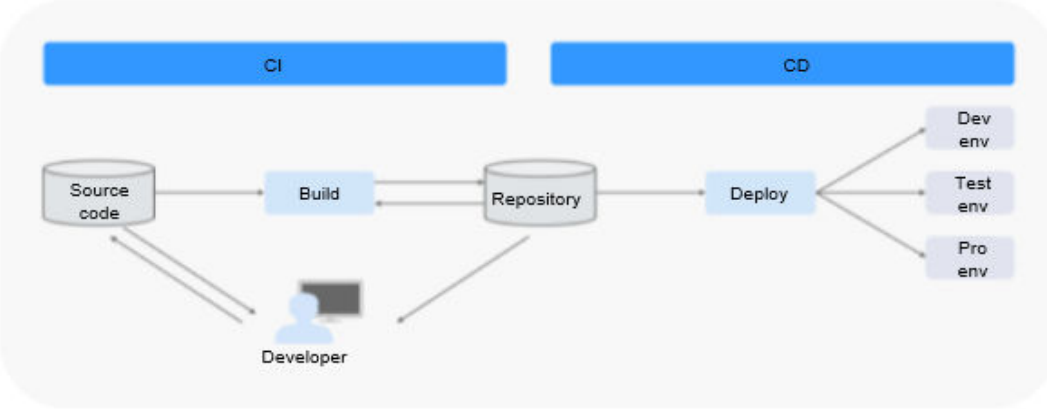
Service Overview

CodeArts Artifact helps software development enterprises manage the software release process in a standardized, visualized, and traceable way.

CodeArts Artifact focuses on and manages the staging **software packages** (usually built from or packed from the **source code**) and their lifecycle metadata. The metadata includes basic properties such as the name and size, repository paths, code branch information, build tasks, creators, and build time.

The management of **software packages** and their properties is the basis of release management. **Figure 1-1** shows the common software development process.

Figure 1-1 Software development process



Repository is a collection of software artifacts and is used to manage packages generated during software development. It is an important link between continuous integration (CI) and delivery (CD). Operations such as release review, tracing, and security control of packages are usually performed in the repository.

CodeArts Artifact provides the following two types of repositories:

- Release repo

  A release repo can store any software packages and tools in any formats.

  Build products can be archived to release repos. You can view and manage the archived software packages and their lifecycle properties. These software packages are used for deployment.

- Self-hosted repo

  A self-hosted repo manages private packages (such as Maven) across various development languages.

  Different development language packages vary in the archive format (for example, the Maven artifact needs to be archived in **GAV** format). CodeArts Artifact manages private packages and shares them with other developers in an enterprise or team.

What Functions Does CodeArts Artifact Provide?

Table 1-1 Release repos

Function	Description
Managing software packages	You can upload, download, search for, and delete software packages. Folders can also be created for better management.
Querying software package properties	You can view the software package lifecycle attributes in release repos. The lifecycle attributes include basic information (such as the name, size, and checksum), and build information (such as the build task, build time, and source code repository).
Uploading software packages to release repos using CodeArts Build	Release repos integrates CodeArts Build. Through configuration, all software packages generated by CodeArts Build can be automatically uploaded to release repos for archiving.
Integration with CodeArts Deploy	Software packages stored in release repos can be used by CodeArts Deploy.
Repo view and version view	You can view a software package in the repo view (storage directory structure) or version view (build task and pipeline).

Table 1-2 Self-hosted repos

Function	Description
Managing private packages	You can upload, download, delete, and search for private packages.

Function	Description
Releasing packages to the self-hosted repo using CodeArts Build	In a build task, you can configure build products to be directly released to a self-hosted repo.
Connecting the local development environment	You can generate a configuration file in one click. After the generated file is configured in the local development tool, you can directly connect the local development environment to the private packages in the self-hosted repo. For example, you can use command lines to upload and download packages in the self-hosted repo.
Managing repository permissions	By setting user roles in repositories, administrator can restrict operation permissions

1.2 Advantages

CodeArts Artifact enriches artifact repository management in common languages by offering artifact lifecycle management, and efficient checking and search. It will keep providing customers with comprehensive, efficient, and trusted artifact management.

In-House, Secure Artifact Repository for Service Continuity

CodeArts Artifact is developed based on the cloud native architecture to resolve service continuity issues caused by external uncontrollable factors. CodeArts Artifact has the following features:

– Security: CodeArts Artifact provides multi-dimensional and fine-grained permission control to meet access control requirements of different roles in an enterprise. It uses the cloud native architecture for physical isolation to reduce the risk of malicious artifact theft.

– Traceability: records user operations.

– Reliability: CodeArts Artifact supports dual-AZ DR and cross-region DR, API traffic limiting and degradation, service dependency and isolation, and automatic service fault detection. These features allow a 99.99% SLA.

– Speed: CodeArts Artifact provides cache acceleration for popular files, incremental upload and download, and full use of cache acceleration advantages for large and small files to improve the build speed, break through the underlying storage bandwidth limit, and implement high-speed concurrent transmission in the same region. Compared with similar open source products, CodeArts Artifact upload performance is improved by 5 times and the download performance by 10 times.

Over 10 Repository Types to Meet Your Needs

CodeArts Artifact supports more than 10 popular package types in Generic, Maven, npm, Go, PyPI, RPM, Debian, Conan, NuGet, and more. It meets the requirements of embedded, web, and mobile application development scenarios. It can seamlessly integrate with on-premises builds, deployment tools, and CI/CD on the cloud. CodeArts Artifact also provides artifact and metadata integrity verification capabilities. It supports fine-grained control and version-based package locking permissions to ensure the integrity of software release tests and comprehensively protect enterprise artifact security.

Search by File Name and Checksum, Locating Hundreds of Millions of Artifacts in Seconds

CodeArts Artifact has powerful search capabilities by using data engine. You can search artifacts quickly from tens of billions of artifact files from multiple dimensions.

It covers multiple package types, such as Maven, npm, Go, PyPI, RPM, Debian, Conan and NuGet. You can search for and locate artifacts in seconds by file name or hash information (MD5, SHA1, SHA256, and SHA512). Based on this, CodeArts Artifact also supports efficient query associating hundreds of millions of metadata and SBOM to quickly trace artifact files. Compared with similar open-source products, CodeArts Artifact improves the search performance by 20 times.

1.3 Constraints

Table 1-3 describes the constraints on CodeArts Artifact.

Table 1-3 Constraints

Category	Item	Limit
Browser	Type	The following browsers are supported: • Chrome: The latest three stable versions are supported and tested. • Firefox: The latest three stable versions are supported and tested. • Edge: Used by default in Windows 10. The latest three stable versions are supported and have been tested. • Internet Explorer is no longer supported and tested. **Chrome and Firefox are recommended.**
Resolution	Resolution	(Recommended) 1280 x 1024 or higher

Category	Item	Limit
Total storage capacity	Release repos and self-hosted repos	Total capacity: 10 GB
Total download capacity	Traffic of release repos and self-hosted repos	Total traffic: 5 GB/month
Constraints on release repos	Maximum size of a file uploaded on the console	1 GB
	Maximum size of a file uploaded by a build task	10 GB
Constraints on self-hosted repos	Maximum size of a file uploaded on the console	Maven/npm/PyPI/Go/RPM/Debian/ Conan: 100 MB NuGet: 20 MB
	Repository quantity	• Max. 100 non-Maven repositories • Max. 50 pairs of Maven repositories
	Maximum size of a file uploaded by a build task	2 GB

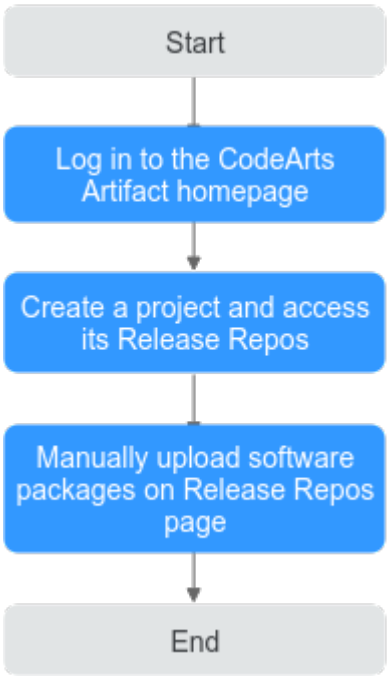
2 Getting Started

2.1 Uploading Software Packages to Release Repos

Software packages are intermediate products generated during compilation and build in software development. They are an indispensable part of continuous integration and continuous delivery. By uploading software packages to release repos for storage and management, you can secure file storage, facilitate software development activities, and provide reliable software package for deployment. Additionally, it provides dependencies for build tasks.

This document describes how to upload software packages to release repos, helping you quickly get started. **Figure 2-1** shows the main operation process.

Figure 2-1 Uploading software packages to release repos



Follow these steps to upload packages to release repos:

1. **Access the CodeArts Artifact homepage**. Before using release repos, go to the CodeArts Artifact homepage.
2. **Create a project and access its release repos**: Create a project on the CodeArts homepage. CodeArts Artifact automatically generates the release repo with the same name. You can then access the repo directly within the project.
3. **Manually uploading software packages to release repos**: You can upload software packages to release repos for secure storage and efficient team collaboration.

Logging In to CodeArts Artifact Homepage

Step 1 Log in to the console.

Step 2 Click  in the upper-left corner of the page and choose **Developer Services** > **CodeArts Artifact** from the service list.

Step 3 Click **Access Service**. The homepage of CodeArts Artifact is displayed.

----End

Creating a Project and Accessing its Release Repos

Step 1 Access the CodeArts homepage.

Step 2 Click **Create Project**.

Step 3 Hover over the **Scrum** card. Click **Select** to use this template to create a project.

Step 4 Set **Project Name** to **Scrum01** and retain the default values for other parameters.

Step 5 Click **OK**. The **Scrum01** project is displayed.

Step 6 Click **Artifact** in the navigation pane to access **Release Repos** of the project. (You do not need to create release repos. After you create a project, release repos with the same project name is automatically generated.)

----End

Manually Uploading Software Packages on the Release Repos Page

Step 1 Go to the release repos named after the project and click **Upload** in the upper-right corner.

Step 2 In the displayed dialog box, configure the following information and click **Upload**.

- **Target Repository**: current release repos. Retain the default setting.
- **Version**: Set the version number for software packages.
- **Upload Mode**: Select **Single file** or **Multiple files**. **Single file** is selected by default here.
- **Path**: After you set the path name, a folder with that name is created in the **Repo View**, where the uploaded package will be stored.
- **File**: Select software packages from the localhost to upload.

Step 3 In the **Repository View**, click the name of the uploaded software package to view its details.

----End

Related Tasks

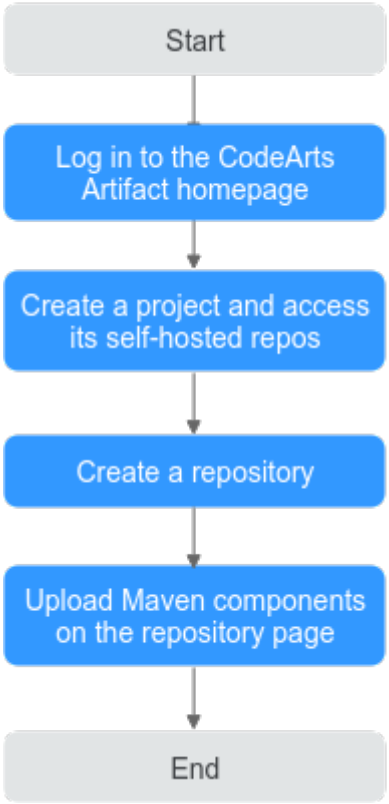
CodeArts Artifact allows you to upload software packages either from the page or through CodeArts Build to release repos. For details, see *User Guide* >"Configuring Build Actions" >"Graphical Build" > "Uploading Software Packages to Release Repos" of CodeArts Build.

2.2 Uploading Artifacts to a Maven Repository

During development, developers often need to share components with teammates. Self-hosted repos provide a shared space to store and upload these components, making it easy for others to access and use them.

This document describes how to upload components to Maven repository, helping you quickly get started. **Figure 2-2** shows the main operation process.

Figure 2-2 Uploading artifacts to a Maven repository



Follow these steps to upload artifacts to a Maven repository:

1. **Access the CodeArts Artifact homepage**. Before using CodeArts Artifact, go to the CodeArts Artifact homepage.
2. **Create a project and access its self-hosted repos**: Create a project on the CodeArts homepage. Then, access the self-hosted repos in the project.

3. **Create a repository**: Create a Maven repository.
4. **Upload Maven artifacts on the self-hosted repo page**: Upload an artifact to a Maven repository on the self-hosted repo page.

Logging In to CodeArts Artifact Homepage

Step 1 Log in to the console.

Step 2 Click  in the upper-left corner of the page and choose **Developer Services** > **CodeArts Artifact** from the service list.

Step 3 Click **Access Service**. The homepage of CodeArts Artifact is displayed.

----End

Creating a Project and Accessing its Self-Hosted Repos

Step 1 Access the CodeArts homepage.

Step 2 Click **Create Project**.

Step 3 Hover over the **Scrum** card. Click **Select** to use this template to create a project.

Step 4 Set **Project Name** to **Scrum01** and retain the default values for other parameters.

Step 5 Click **OK**. The **Scrum01** project is displayed.

Step 6 Click **Artifact** in the navigation pane to access the **Self-hosted Repos** of the project.

----End

Creating a Repository

Step 1 On the Artifact homepage, click the **Repositories** tab.

Step 2 Click **Create Repository**.

Step 3 Configure the basic information and click **Submit**.

- **Repository Type**: **Local Repo** and **Virtual Repo**. **Local Repo** is selected by default.
- **Repository Name**: Enter a repository name.
- **Package Type**: Select **Maven**.
- **Project**: The default value is the current project. You can select another target project from the drop-down list box.
- **Include Patterns**: (Optional) Configure a path whitelist for the repository.
- **Version Policy**: If both of them are selected, the Maven repository generates two types of repositories: Release and Snapshot. Retain the default values.
- **Description**: (Optional) Enter up to 200 characters.

Step 4 The created Maven repository is displayed in the **Repository View**.

----End

Uploading Maven Artifacts on the Self-Hosted Repo Page

Step 1 Go to the **Self-hosted Repos** in the left pane, and click the target repository.

Step 2 Click **Upload**.

Step 3 In the displayed dialog box, set **Upload Mode** to **POM**.

Step 4 In **POM**, click **Select File** and upload components whose name ends with **pom.xml** or **.pom** from the localhost.

Step 5 Click **Upload**.

Step 6 In the **Repository View**, click the name of the uploaded software package to view its details.

----End

Related Tasks

CodeArts Artifact allows you to upload components either from the page or through CodeArts Build to self-hosted repos. For details, see *User Guide* >"Configuring Build Actions" >"Graphical Build" "Using Maven for Build" of CodeArts Build.

3 User Guide

3.1 Release Repo

3.1.1 Overview

CodeArts Artifact is a general, unified repository used to manage software artifacts in different formats. In addition to basic storage functions, it also provides important functions such as build and deployment tool integration, version control, and access permission control. It is a standardized way for enterprises to process all artifact types generated during software development.

Operations pertaining to a release repo include:

- Basic operations: You can upload, download, edit, search for, and delete software packages, as well as create, edit, search for, and delete folders.
- Viewing and editing software package details: Software package details include basic, build metadata, and build packages. The folder name, software package name and status, and release version are editable.
- Managing the recycle bin: After a software package is deleted, it is moved to the recycle bin. You can restore the package to the folder before the deletion or permanently delete the package from the recycle bin.

3.1.2 Accessing Release Repos

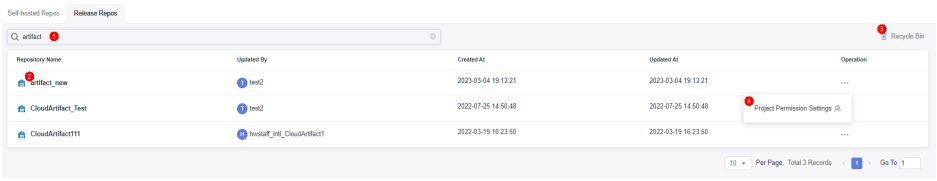
You can access the release repo page in either of the following ways: homepage entry and project entry.

Homepage Entry

Step 1 Log in to CodeArts.

Step 2 Choose **Services** > **Artifact** from the top menu bar.

Step 3 View the project name list of the current tenant. You can perform the following operations as required:



No.	Operation	Description
1	Search for repos	Enter the project name in the search box to find the existing software release repo of the project.
2	View folder details	Click any folder to view the list of archived software packages in the folder or folders. You can upload, download, and edit software packages or folders.
3	Manage recycle bin	Click **Recycle Bin** to go to the recycle bin page. You can delete or restore the software package or folder as required.
4	Set project permissions	Click ··· to go to the **Set Project Permissions** page and edit member permissions. For details, see **Setting Permissions**.

NOTE

On the **Release Repos** page, you will view a project list, but you cannot upload files or create folders there. To perform such operations, click a project name to access the specific project first.

----End

Project Entry

Step 1 Log in to the CodeArts homepage and click a card to access a project.

Step 2 Choose **Artifact** > **Release Repos**.

Step 3 View the list of software packages and folders archived in the current project.

Perform the operations described in the following sections as required.

----End

3.1.3 Performing Basic Operations

Access the release repo homepage by referring to **Project Entry**. You can upload, create, download, edit, search for, and delete software packages.

Creating a Folder

Step 1 On the **Release Repos** tab, click **Create Folder** on the right of the page to create a folder. Folders can be nested.

Click to change a folder name.

Click to delete a folder and software package in the folder. You can choose to permanently delete the folder or move the deleted folder to the recycle bin.

Step 2 Select a folder and click **Create Folder** to create a level-2 folder.

----End

Setting Status

After entering a level-1 folder, you can click next to **Package Status** and select a status from a drop-down list to change the status of a level-2 folder (**Testing** by default).

If the folder status is **Production Package**, the folder cannot be changed or edited (changing the folder name, changing the file name in the folder, uploading the folder, changing the version number, or creating a subfolder). You can only download or delete it.

NOTICE

The folder status can be changed from **Not Released** to **Released**, but such change is irreversible. Exercise caution when performing this operation.

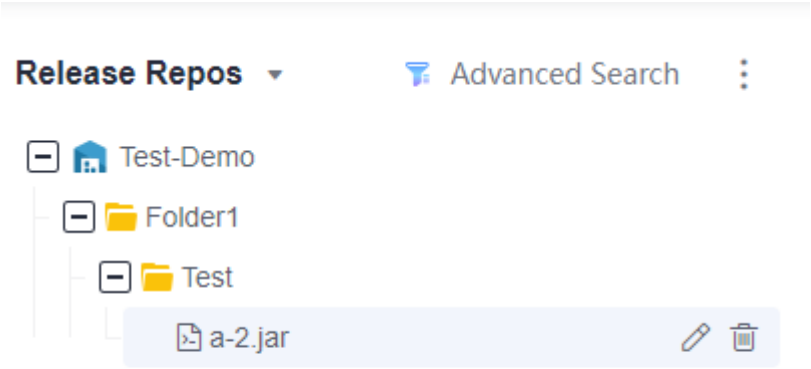
Uploading a Software Package

Step 1 Click **Upload** in the upper right corner of the page to manually upload local software packages to the release repo.

After selecting a folder, click **Upload** to manually upload a local software package to the target folder.

Step 2 Click to change a software package name.

Click to delete the software package permanently or move the deleted folder to the recycle bin.



NOTE

You are advised not to upload files containing sensitive information such as plaintext accounts and passwords to release repos.

----End

Searching

Step 1 Access the release repo by referring to **Project Entry**.

Step 2 Enter a keyword (a folder or file name) in the search box above the list to search for the package whose name contains the keyword.

Step 3 Click a file name to go to the file details page.

----End

Downloading a Software Package

Scenario 1

Step 1 Access the release repo by referring to **Project Entry**, select the software package to be downloaded, and click **Download**.

Step 2 In the displayed dialog box, select a method to download.

- **Local Download**: Download the software package to a local PC.
- **QR Code Download**: Use a mobile phone to scan the QR code to download the file.

----End

Scenario 2

Step 1 Access the release repo through **Project Entry** and hover the mouse pointer over the software package to be downloaded.

Step 2 Click  on the right of the software package.

----End

3.1.4 Viewing/Editing Software Packages

On the release repo page, you can view or edit software package details, including general information, build metadata information, and build packages.

Access the release repo by referring to **Project Entry** and click the software package name. The details about the selected software package are displayed. The software package details are displayed on three pages: **General**, **Build Metadata**, and **Build Packages**.

- **General**: displays the repository name, relative path, download address, released version, creator, creation time, size, and checksum.

  Click  to change the release version of the software package. (The release version archived by CodeArts Build is the build sequence number by default.)

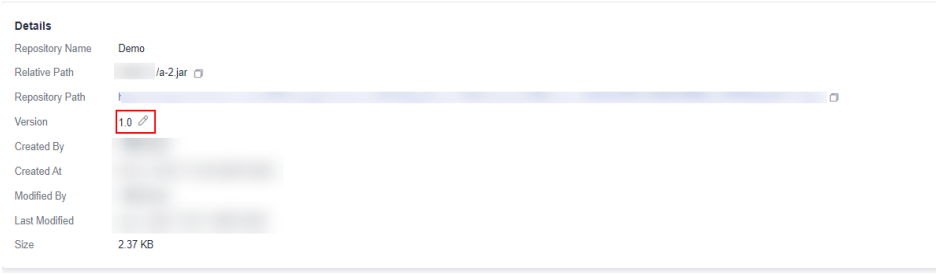
- **Build Metadata**: displays the build task, size, sequence number, builder, code repository, and code branch of the generated software package. Click **Build Task** to link to the task in CodeArts Build.
- **Build Packages**: displays the archiving records of software packages uploaded through build tasks. You can click to download a package.

3.1.5 Version View

CodeArts Artifact displays packages by version. In **Version View** tab, you can filter and check artifacts by name and version number, and sort files by their update time.

Step 1 Access the release repo by referring to **Project Entry**.

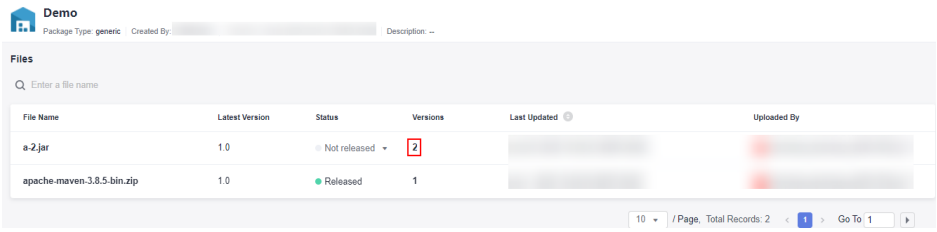
Step 2 Set the version of the uploaded package. By default, CodeArts Build archives the release version as the version number set during the build task.



Step 3 Select **Version View** in the upper left corner of the page. The list of software packages with version numbers is displayed.

Step 4 Release repos store packages of different versions with the same name in the same file. Click a file name in the **File Name** column. The latest version of the package is displayed.

Step 5 Click a number in the **Versions** column. The version list of the package is displayed.



Click a number in the **Version No.** column. The **Overview** and **Files** pages of the package are displayed. In the **Files** tab, click a name in the **File Name** column. The path of the package is displayed.

Step 6 Change the status if needed. By default, the package status is set to **Not released** after the version is configured.

- In **Files**, set the status to **Released**. This will update the package to the **Released** status for the latest version.
- Click a version in the **Versions** column to go to the version list. You can set the status of different versions to **Released**.

NOTE

The status changes from **Not released** to **Released**. The status change is irreversible. Exercise caution when performing this operation. Files in **Released** status cannot be modified or edited (file names or version numbers), but can only be downloaded or deleted.

----End

3.1.6 Managing Recycle Bin

Software packages or folders deleted from the release repo are moved to the recycle bin, where you can manage them.

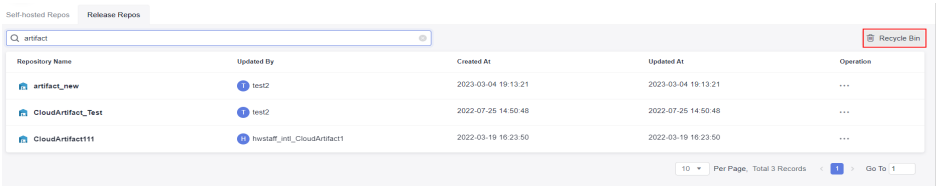
CodeArts Artifact provides a **global recycle bin** and **project-level recycle bins**.

Global Recycle Bin

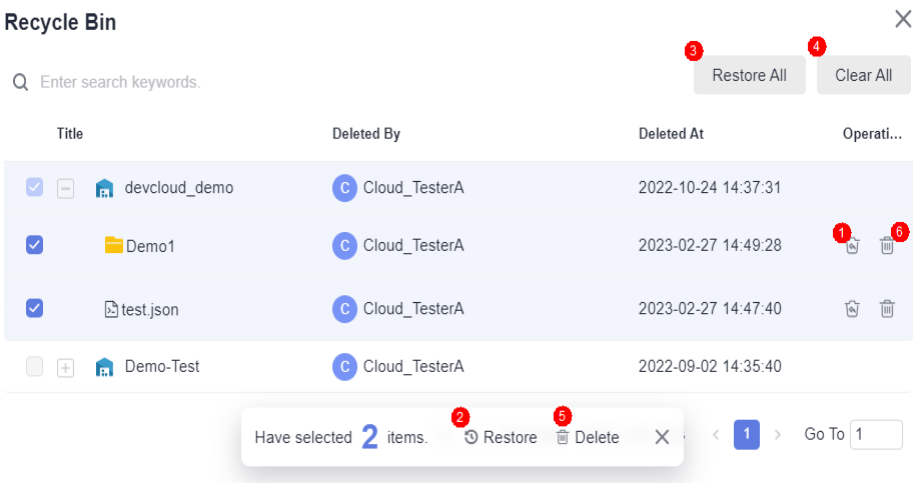
In the global recycle bin, you can manage software packages and folders deleted from all projects.

Step 1 Log in to CodeArts and choose **Services** > **Artifact** from the top menu bar.

Step 2 Click the **Release Repos** tab and click **Recycle Bin**.



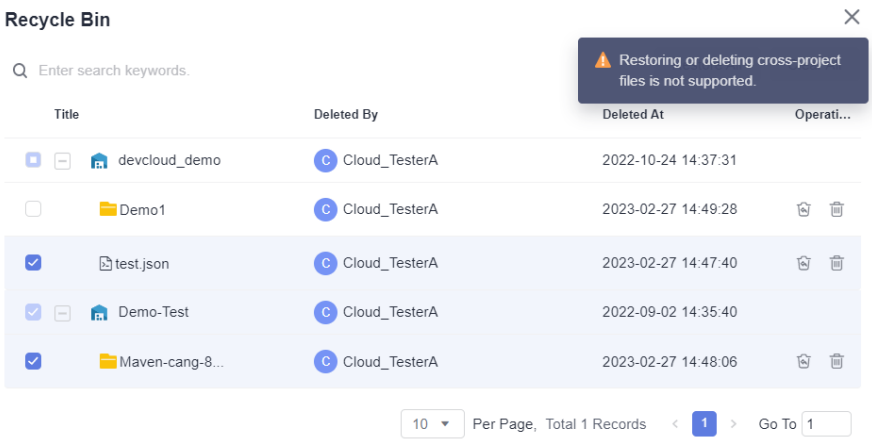
Step 3 Deleted files from different projects are listed on this page. Perform the following operations on a package or folder as required.



No.	Operation	Description
1	Restore	Click in the **Operation** column to restore a package or folder.
2	Batch restore	Select multiple packages or folders and click **Restore** below the list to restore all the selected items.
3	Restore all	Click **Restore All** to restore all packages or folders in the recycle bin by one click.
4	Clear recycle bin	Click **Clear All** to delete all packages or folders from the recycle bin.
5	Batch delete	Select multiple packages or folders and click **Delete** below the list to delete all the selected items.
6	Delete	Click in the **Operation** column to delete a package or folder.

NOTICE

1. Once you delete a package or folder from the recycle bin, it cannot be recovered. Exercise caution when performing this operation.
2. When selecting multiple files to restore or delete in batches, the global recycle bin does not allow you to restore or delete files across projects.



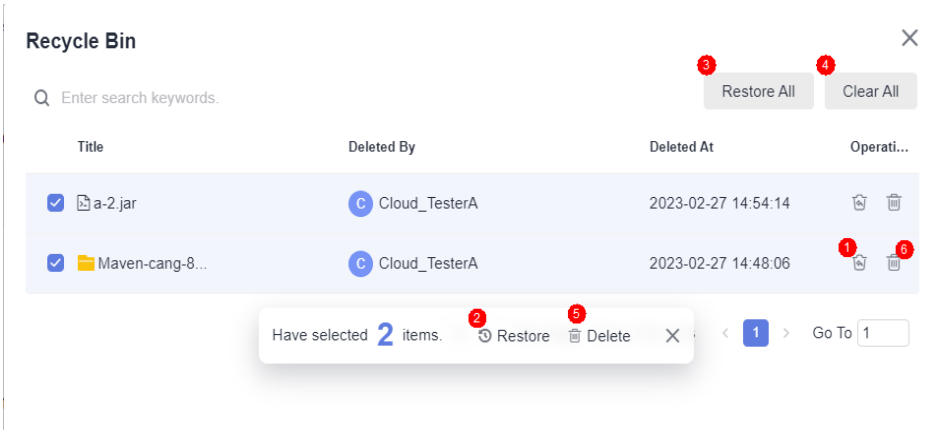
----End

Project Recycle Bin

You can process deleted software packages or folders in a project.

Step 1 Access the **Release Repos** page by referring to **Project Entry** and click **Recycle Bin** in the lower left corner of the page.

Step 2 Deleted files of this project are listed on this page. Perform the following operations on a package or folder as required.



No.	Operation	Description
1	Restore	Click in the **Operation** column to restore a package or folder.
2	Batch restore	Select multiple packages or folders and click **Restore** below the list to restore all the selected items.
3	Restore all	Click **Restore All** to restore all packages or folders in the recycle bin by one click.
4	Clear recycle bin	Click **Clear All** to delete all packages or folders from the recycle bin.
5	Batch delete	Select multiple packages or folders and click **Delete** below the list to delete all the selected items.
6	Delete	Click in the **Operation** column to delete a package or folder.

NOTICE

Once you delete a package or folder from the recycle bin, it cannot be recovered. Exercise caution when performing this operation.

----End

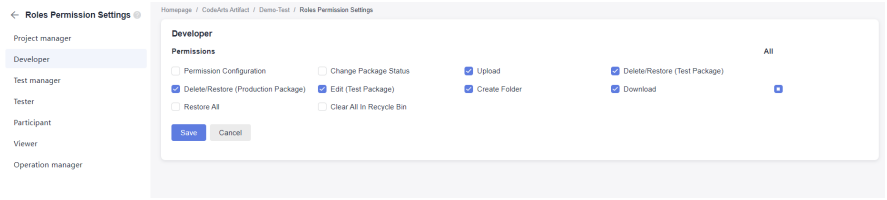
3.1.7 Setting Permissions

In the release repos, project roles have different operation permissions. Members who have the **Permission Settings** permission can edit the permission scope.

Step 1 Access the release repo by referring to **Project Entry**.

Step 2 Click ⋮ in the upper-left corner and select **Set Project Permissions** from the drop-down list.

Step 3 Click the role for which you want to set permissions, select the permissions as required, and click **Save**.



The table below lists the default permission matrix provided by release repos.

Operation/ Role	Project Creator	Project Manager	Developer	Test Manager	Tester	Operation Manager	Participant	Viewer
Set permissions	√	√	-	-	-	-	-	-
Change package status	√	√	-	-	-	√	-	-
Upload	√	√	√	√	√	√	-	-
Delete/ Restore (test package)	√	√	√	√	-	√	-	-
Delete/ Restore (production package)	√	-	-	-	-	√	-	-
Edit (test package)	√	√	√	√	-	√	-	-
Create a folder	√	√	√	√	√	√	-	-
Download	√	√	√	√	√	√	√	√
Restore all	√	√	-	√	-	-	-	-
Clear recycle bin	√	√	-	√	-	-	-	-

NOTE

- By default, the project creator has all permissions and is not shown on the page. Their permission scope cannot be changed.
- Viewers only have permission to download, and their permission scope cannot be changed.

----End

3.1.8 Clearing Policies

Release repos automatically clear files on a scheduled basis. You can set a clearing policy to move expired files from the repository to the recycle bin or delete them permanently from the recycle bin based on the specified retention periods.

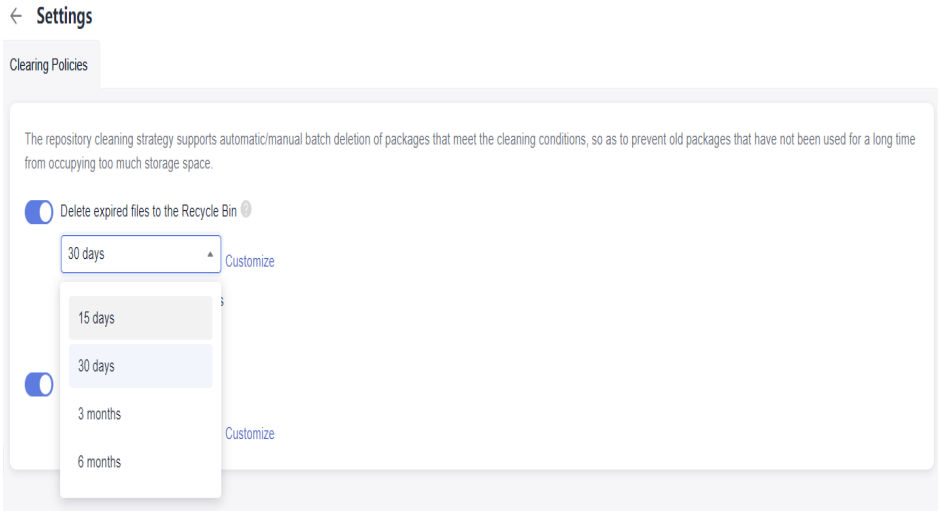
Step 1 Access the release repo by referring to **Project Entry**.

Step 2 Click **Settings** in the upper right corner of the page and select **Clearing Policies**.

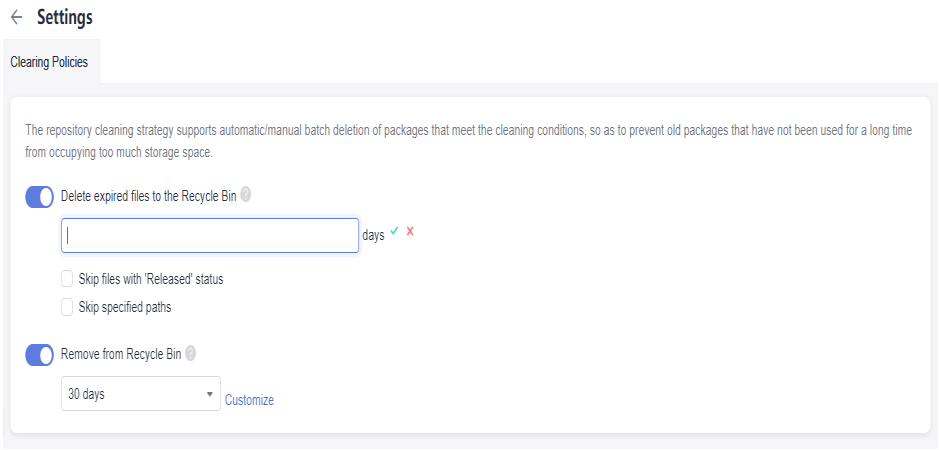
Step 3 Enable **Move expired files to the Recycle Bin** or **Clear from Recycle Bin** as required, and select a retention period from the drop-down list.

Default retention periods:

- **Move expired files to the Recycle Bin**: 30 days
- **Clear from Recycle Bin**: 30 days



You can also set a period. Click **Customize**, enter a number, and click √ to save the setting.



NOTE

Parameters below are optional.

- **Skip released files**: The system retains files in the production package state when deleting files. For details, see **Setting Status**.
- **Skip specified paths**: The system retains packages that match the file paths you set when clearing files. You can set multiple file paths, each starting with a slash (/) and separated by semicolons (;).

----End

3.2 Self-Hosted Repo

3.2.1 Introduction

A self-hosted repo manages private packages, supporting Maven, npm, Go, PyPI, RPM, Debian, Conan, NuGet, and OHPM.

A self-hosted repo provides the following functions:

- Repository management includes creating repositories, editing basic repository information, managing repository permissions, and connecting repositories to your local development environment.
- Private package management includes uploading, downloading, searching, and deleting private packages, as well as managing items in the recycle bin.

3.2.2 Accessing Self-Hosted Repos

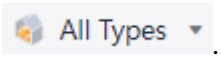
You can access the self-hosted repo page in either of the following ways: homepage entry and project entry.

Homepage Entry

Step 1 Log in to CodeArts.

Step 2 Choose **Services** > **Artifact** from the top menu bar.

Step 3 Click the **Self-hosted Repos** tab. All self-hosted repos created for the service are displayed.

Click  . In the drop-down list, you can view repositories by type.

Step 4 Click a repository name to go to the self-hosted repo page for the project.

----End

Project Entry

Step 1 Log in to the CodeArts homepage and click a card to access a project.

Step 2 Choose **Artifact** > **Self-hosted Repos** from the navigation pane.

Step 3 View the list of different types of repositories archived in the current project.

Perform the operations described in the following sections as required.

----End

3.2.3 Creating a Repository

If you use this service for the first time, you need to create a repository. Only tenant administrators have the permission to create self-hosted repos.

- Follow instructions in **Homepage Entry** to access the self-hosted repo. You can click **Create Self-hosted Repo** in the upper left corner of the page.
- Access the self-hosted repo by referring to **Project Entry**. You can click  in the upper left corner of the page.

Step 1 The **Create Repository** page is displayed.

Step 2 Configure basic information and click **OK**.

Configuration Item	Required	Description
Name	Yes	Enter up to 20 characters. Only letters, digits, underscores (_), hyphens (-), and periods (.) are supported. **NOTE** After a repository is created, the repository name cannot be changed.
Package Type	Yes	support Maven, npm, Go, PyPI, RPM, Debian, Conan, Docker, CocoaPods, OHPM, and NuGet artifacts. Complete the configuration for the selected format by referring to **Configuring Repository Items**.
Project	Yes	Choose a project to associate with the new repository. After the settings are complete, the project to which the user belongs cannot be changed.

Configuration Item	Required	Description
Description	No	Enter up to 200 characters.

Step 3 The created repositories are displayed on the repo view. You can click a repository name to view its details page. The repository details are displayed on the **General**, **Repository Permissions**, **Resources**, and **Activity** tabs.

- **General**: displays the repository name, repository type, repository URL, relative path, creator, creation time, modifier, modification time, artifact count, and artifact size.
- **Repository Permissions**: displays the operation permissions allowed for each role on the current repository.
- **Resources**: collect statistics on artifacts uploaded to the repository by **File Counts** and **Storage Capacity (Byte)**.
- **Operation Logs**: displays the operation history of uploading, deleting, and restoring data from the recycle bin in the repository.

----End

Configuring Repository Items

The following table describes the configuration items specific to each type of repository.

Type	Configuration Item	Required	Description
Maven	Version Policy	Yes	The options are **Release** and **Snapshot**. You are advised to select both. If so, two repositories will be generated: **Release** and **Snapshot**. If you select one, a **Release** or **Snapshot** repository will be generated.
	Include patterns	No	Enter rules to specify paths allowed for artifact uploads. You can click **+** to add multiple paths. During builds, only Maven files whose paths start with the preceding path prefix can be uploaded to the self-hosted repo.
npm	Include patterns	No	Enter rules to specify paths allowed for artifact uploads. You can click **+** to add multiple paths. During builds, only npm files whose paths start with the preceding path prefix can be uploaded to the self-hosted repo.

Type	Configuration Item	Required	Description
Go	Include patterns	No	Enter rules to specify paths allowed for artifact uploads. You can click **+** to add multiple paths. During builds, only Go files whose paths start with the preceding path prefix can be uploaded to the self-hosted repo.
PyPI	Include patterns	No	Enter rules to specify paths allowed for artifact uploads. You can click **+** to add multiple paths. During builds, only PyPI dependency packages in which the **name** value in the **setup.py** file matches the preceding path prefix can be uploaded to the self-hosted repo.
RPM	Include patterns	No	Enter rules to specify paths allowed for artifact uploads. You can click **+** to add multiple paths. During builds, only RPM binary files whose paths start with the preceding path prefix can be uploaded to the self-hosted repo.
Conan	Include patterns	No	Enter rules to specify paths allowed for artifact uploads. You can click **+** to add multiple paths. Only Conan files whose paths start with the preceding path prefix can be uploaded from a local client to the self-hosted repo.
Docker	Include patterns	No	Enter rules to specify paths allowed for artifact uploads. You can click **+** to add multiple paths. Only image files whose paths start with the preceding path prefix can be pushed to the self-hosted repo.
CocoaPods	Include patterns	No	Enter rules to specify paths allowed for artifact uploads. You can click **+** to add multiple paths. During builds, only CocoaPods files whose paths start with the preceding path prefix can be uploaded to the self-hosted repo.
OHPM	Include Patterns	No	Enter rules to specify paths allowed for artifact uploads. You can click **+** to add multiple paths. For example, if you add **demo**, that is, **demo/**/***, all paths starting with **demo/** are allowed. During builds, only OHPM files whose paths start with the preceding path prefix can be pushed to the self-hosted repo.

3.2.4 Managing Repositories

You can edit repository descriptions, add paths, delete repositories, and manage user permissions.

Editing Repository Descriptions and Paths

Step 1 Go to the self-hosted repo page. In the left pane, click the name of the target repository to be edited.

Step 2 Click **Settings** on the right of the page to view the basic information about the repository.

Step 3 Edit the repository description as required and click **Submit**.

NOTE

On the basic information page, the repository name, package type, home project, and permission scope cannot be modified.

On the **Basic Information** page of the repository, set **Include Patterns** and click + to add paths for Maven, npm, Go, PyPI, RPM, and Conan repositories.

To delete path rules, click .

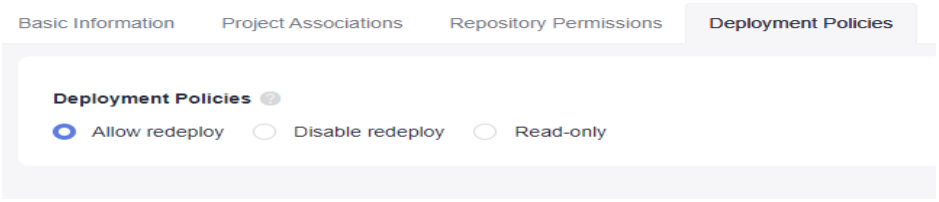
----End

Configuring Deployment Policies

The self-hosted repo supports three version policies: **Allow redeploy**, **Disable redeploy**, and **Read-only**. You can set whether to allow artifacts in the same path to be uploaded and overwrite the original package.

Step 1 Access the self-hosted repo homepage. In the left pane, click the target repository name.

Step 2 Click **Settings** on the right of the page and click the **Deployment Policies** tab.



- **Allow redeploy** (selected by default): Artifacts in the same path can be uploaded, replacing the original package.
- **Disable redeploy**: Artifacts in the same path cannot be uploaded.
- **Read-only**: Artifacts cannot be uploaded, updated, or deleted. You can download an uploaded artifact.

Step 3 Click **OK**.

----End

Deleting a Repository

You can delete a self-hosted repo. Deleted repositories are moved to the recycle bin.

Step 1 Go to the self-hosted repo page. In the left pane, click the name of the target repository.

Step 2 Click **Settings** on the right of the page to view the basic information about the repository.

Step 3 Click **Delete** on the right of the page. Check that the deleted repository is no longer displayed in the repository list in the left pane.

----End

Tutorial

You can connect the self-hosted repo to a local development environment so that private packages in the self-hosted repo can be used during local development.

Step 1 Go to the self-hosted repo page. In the left pane, click the name of the repository to be connected to your local development environment.

Step 2 Click **Tutorial** on the right of the page.

Step 3 In the displayed dialog box, click **Download Configuration File** to download the configuration file to your local directory.

Step 4 Copy the downloaded file to the target directory based on the instructions in the **Information** dialog box.

----End

Configuring the Encryption Mode of a Maven Repository

Dependency management tool: Maven

- Ensure that you have installed JDK and Maven.
- Download the configuration file, or modify the Maven **settings.xml** file in the **conf** or **.m2** directory as follows.

To encrypt a password, perform the following operations:

Step 1 Create a master password.

```
mvn --encrypt-master-password <password>
```

An encrypted password is generated, for example, **{jSMOWnoPFgsHVpMvz5VrIt5kRbzGpI8u+9EF1iFQyJQ=}**.

Save it to the **${user.home}/.m2/settings-security.xml** file. For example:

```
<settingsSecurity>
<master>{jSMOWnoPFgsHVpMvz5VrIt5kRbzGpI8u+9EF1iFQyJQ=}</master>
</settingsSecurity>
```

Step 2 Encrypt the password.

```
mvn --encrypt-password <password>
```

An encrypted password is generated, for example: **{COQLCE6DU6GtcS5P=}**.

Save it to the **settings.xml** file. For example:

```
<settingsSecurity>
<master>{jSMOWnoPFgsHVpMvz5VrIt5kRbzGpI8u+9EF1iFQyJQ=}</master>
</settingsSecurity>
```

----End

Dependency management tool: Gradle

- Ensure that you have installed JDK and Gradle.

Step 1 Add the **nu.studer.credentials** plug-in to the **build.gradle** file in the Gradle project.

```
plugins{
 id 'nu.studer.credentials' version '2.1'
}
```

Step 2 Create an encryption password.

```
gradle addCredentials -PcredentialsKey=accountPassword -PcredentialsValue="**"
```

The following encrypted password is generated: **cVe******kg\=\=**.

By default, the data is stored in the **GRADLE_USER_HOME/gradle.encrypted.properties** file. For example:

```
accountPassword=cVe******kg\=\=
```

Step 3 Configure the repository with an encrypted password in the **build.gradle** file.

```
def password = credentials.accountPassword
            repositories {
              maven {
                credentials {
                  username
'{region}-1_f9e40463c23845438ca9efd3a7ec854e_67902fb51ded48c2868310a07e1569e7'
                  password password
                }
                url 'https://{url}/artgalaxy/{region}-1_f9e40463c23845438ca9efd3a7ec854e_maven_1_7/'
              }
            }
```

----End

Resetting Repository Passwords

You can reset the password in the self-hosted repo configuration file. After the password is reset, download the configuration file again to replace the original file.

Step 1 Access the self-hosted repo homepage. Click ⋮ above the repository list on the left and choose **Reset Repository Password**.

Step 2 In the displayed dialog box, click **OK**. Check that a message is displayed indicating the password has been reset.

----End

Obtaining Repository URLs

The URL of the self-hosted repo will be used when you connect the repository to the local development environment. You can perform the following operations to obtain the URL:

Step 1 Access the self-hosted repo homepage. In the left pane, click the target repository name.

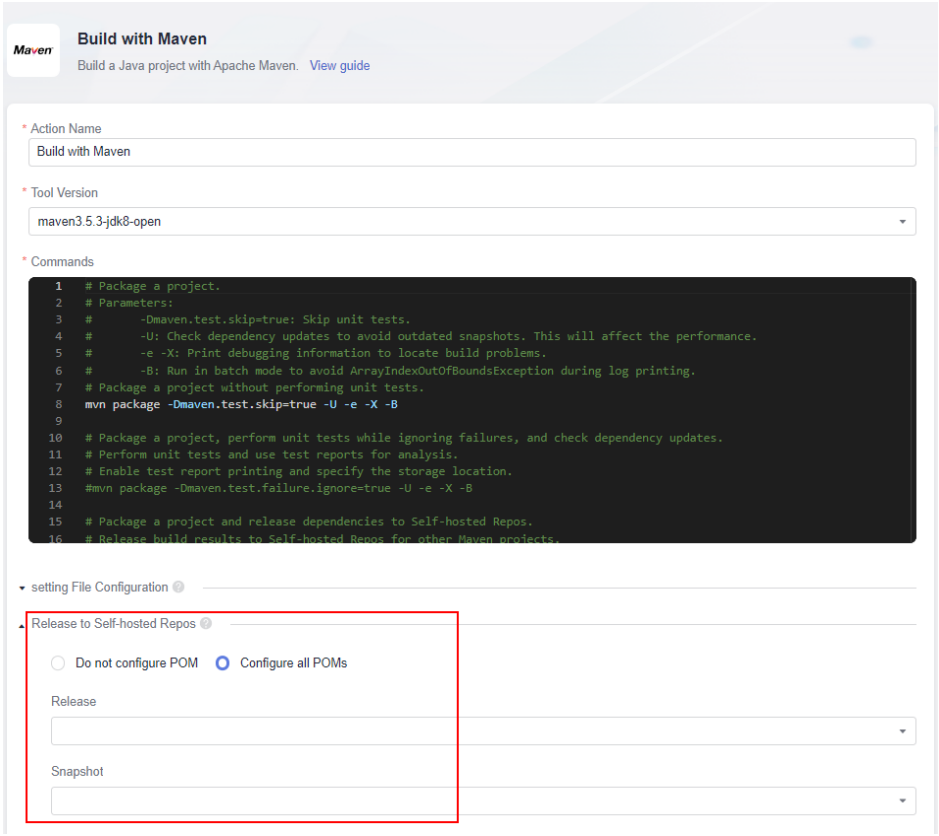
Step 2 The repository URL is displayed on the **General** page. Click to obtain the URL.

----End

Obtaining Associations Between Maven Repository and Project

When uploading a Maven artifact to the self-hosted repo through a build task, specify the repository path in the **Build with Maven** step.

- **Do not configure POM**: The dependency package is not released to the self-hosted repo.
- **Configure all POMs**: If you run the **mvn deploy** command, the dependency is released to the specified **Release** and **Snapshot** repositories.



After the Maven repository is associated with a project, you can select the repository in the build step of the build task in the project.

Step 1 Access the self-hosted repo homepage. In the left pane, click the name of a Maven repository.

Step 2 Click **Settings** on the right of the page, and select **Project Associations**.

Step 3 In the **Operation** column of the target project in the Maven repository, click .

Step 4 In the displayed dialog box, select the repository name, and click **OK**.

After the "Operation Successful" message is displayed, the value of **Associated Repositories** for the project will be updated according to the number of selected repositories.

----End

3.2.5 Configuring Clearing Policies

You can automatically and manually delete artifacts that meet deletion conditions in batches. When you create a Maven repository, the storage repositories include **Release** and **Snapshot**.

The snapshot of a Maven artifact is a special version that specifies a copy of the current development progress, and is different from a common version. Maven checks a new snapshot in the remote repository during each build and provides the maximum number of snapshot versions that can be retained and the function of automatically clearing expired snapshot versions.

The artifact cleanup strategy reduces the waste of storage space, makes artifacts in the repository clear, and ensures that artifacts are transferred in order during development, testing, deployment, and release.

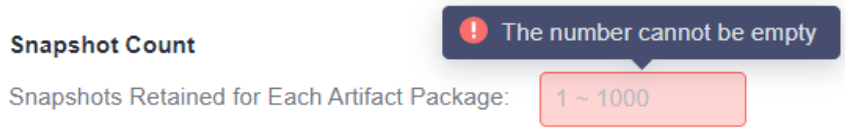
Procedure

Step 1 Access the self-hosted repo by following instructions in **Project Entry**.

Step 2 Select the corresponding **Snapshot** Maven repository from the repository list on the left and click **Settings** in the upper-right corner of the page.

Step 3 Click the **Clearing Policies** tab.

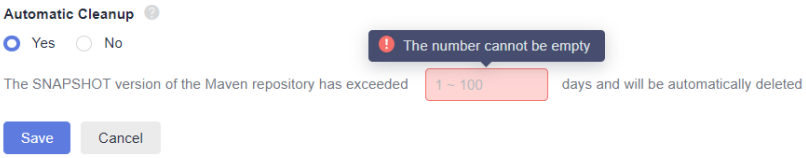
Step 4 Set the maximum number of **Snapshot Count**. The value ranges from 1 to 1000.



When the version of an artifact package exceeds this value, the package of the earliest version is overwritten by the package of the latest version.

Step 5 Enable automatic cleanup (**No** by default). Click **Yes** and enter the number of days. Snapshot versions that have been stored for more than the specified number of days will be automatically cleaned up.

The automatic cleanup time cannot be less than 1 day or greater than 100 days.



Step 6 Click **Save** to complete configuration.

----End

3.2.6 Uploading a Private Package

Only repository administrators and developers can upload private packages. You can set repository roles on the **Repository Permissions** page.

Procedure

Step 1 Go to the self-hosted repo page. In the left pane, click the target repository to which the private package is to be uploaded.

Step 2 Click **Upload** on the right of the page.

Step 3 Set the package parameters, select the file, and click **Upload**. The detailed configuration for each type of package is described below.

NOTE

1. The maximum file size for uploads is 100 MB for Maven, npm, PyPI, RPM, and Debian types, and 20 MB for NuGet.
2. You are advised not to upload files containing sensitive information such as plaintext accounts and passwords to self-hosted repos.

----End

Maven Artifacts

- Project Object Model (POM) is the basic working unit of a Maven project. It is an XML file that contains basic project information to describe how to build a project and declare project dependencies. When a build task is run, Maven searches for the POM in the current directory, reads its content, obtains the required configuration information, and builds the target package.
- Maven coordinates: X, Y, and Z are used to uniquely identify a point in the three-dimensional space. In Maven, GAV is used to identify a unique package. It stands for **groupId**, **artifactId**, and **version**. **Group ID** indicates a company or organization. For example, Maven core components are in the **org.apache.maven** organization. **artifactId** indicates the name of a package. **version** indicates the version of the package.
- Maven dependencies are crucial for POM files. The building and running of most projects depend on the dependency on other components. Add the dependency list to your POM file. If the **App** component depends on the **App-Core** and **App-Data** components, the configuration is as follows:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.companyname.groupname</groupId>
    <artifactId>App</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>
    <dependencies>
      <dependency>
        <groupId>com.companyname.groupname</groupId>
        <artifactId>App-Core</artifactId>
        <version>1.0</version>
      </dependency>
    </dependencies>
    <dependencies>
      <dependency>
```

```
        <groupId>com.companyname.groupname</groupId>
        <artifactId>App-Data</artifactId>
        <version>1.0</version>
      </dependency>
    </dependencies>
</project>
```

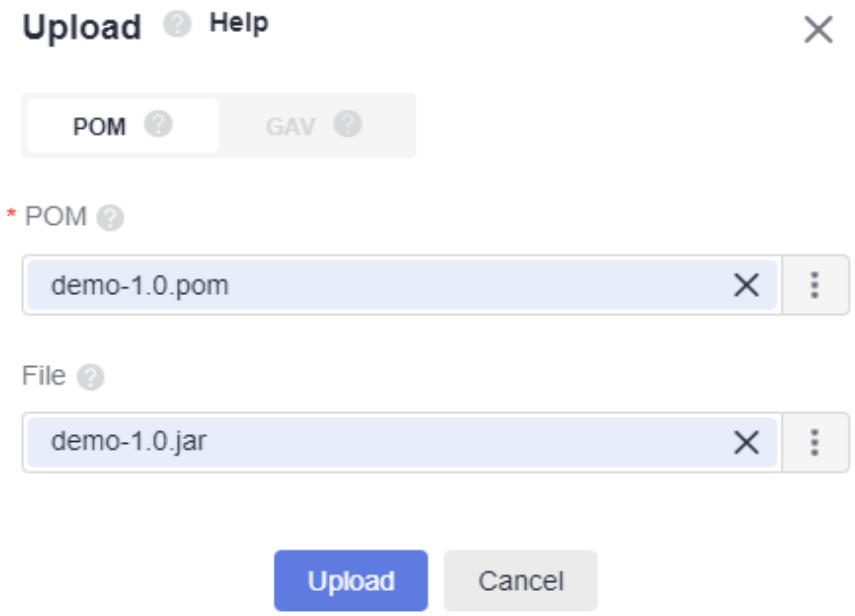
Uploading a Maven Artifact

POM and GAV upload modes are supported.

Upload Mode	Description
POM	GAV parameters are obtained from the POM file. The system retains transitive dependencies of components.
GAV	GAV, short for **Group ID**, **Artifact ID**, and **Version**, is the unique identifier of a JAR package. In this mode, GAV parameters are manually specified. The system automatically generates a POM file without any transitive dependency.

- POM

  In POM mode, you can either upload just the POM file or both the POM file and its related components. The uploaded file must match the **artifactId** and **version** values specified in the POM. As shown in the following figure, if the **artifactId** value is **demo** and the **version** value is **1.0** in the POM, then the uploaded file must be named **demo-1.0.jar**.

  The **POM** file structure is as follows:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>demo</groupId>
 <artifactId>demo</artifactId>
```

```
  <version>1.0</version>
</project>
```

NOTE

The **modelVersion** tag must exist and the value must be **4.0.0**, indicating that **Maven2** is used.

If you upload files in both the **POM** and **File** area, the **artifactId** and **version** values in **POM** must match the file name in **File**. For example, if the **artifactId** value is **demo** and the **version** value is **1.0** in **POM**, then the uploaded file must be named **demo-1.0** in **File**.

- GAV

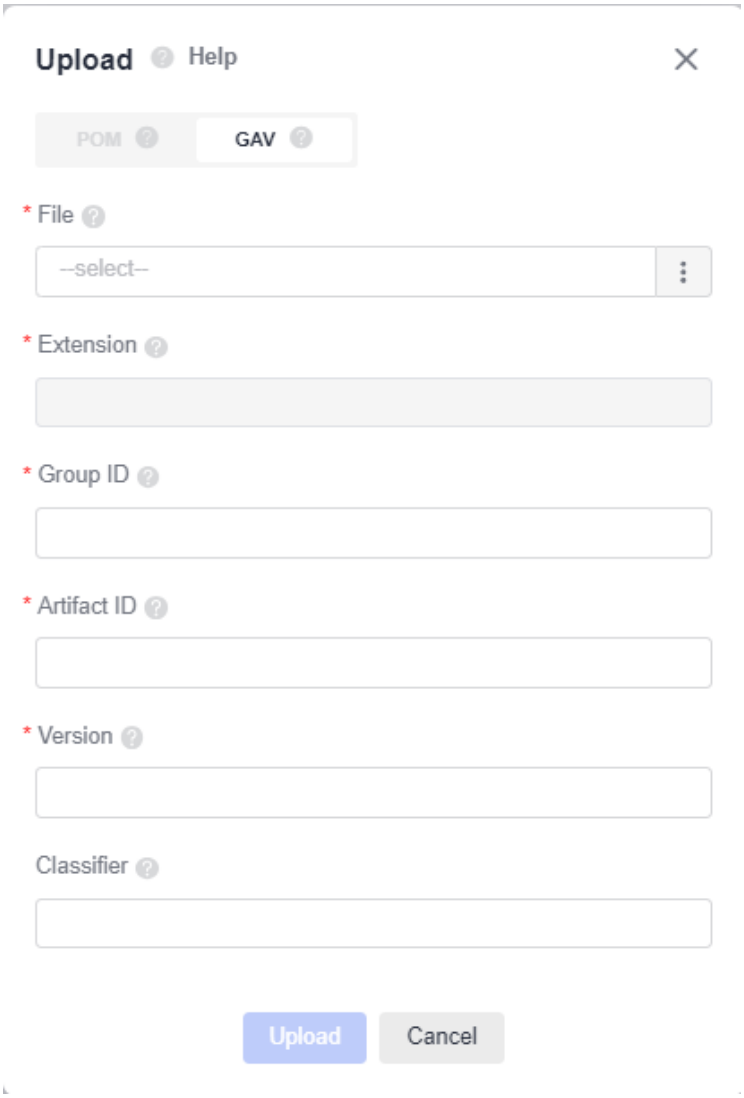
  In the GAV mode, the **Group ID**, **Artifact ID**, and **Version** parameters must be manually specified and they determine the name of the file to upload. **Extension** indicates the packaging type and determines the file type to be uploaded.

  Classifier is used to differentiate artifacts that are built from the same POM but contain different contents. This field is optional. It can contain letters, digits, underscores (_), hyphens (-), and dots (.). If you specify this field, it will be appended to the file name.

  Common Usage Scenario

  – Differentiate versions by name, such as **demo-1.0-jdk13.jar** and **demo-1.0-jdk15.jar**.

  – Differentiate usage by name, such as **demo-1.0-javadoc.jar** and **demo-1.0-sources.jar**.



npm Packages

Node Package Manager (npm) is a JavaScript package management tool. An npm package is the item managed by npm, and the npm registry is used to store and manage these packages.

The npm package consists of a structure and file description.

- Package structure: organizes various files in a package, such as source code files and resource files.
- Description file: describes package information. Example: **package.json**, **bin**, and **lib** files.

The **package.json** file in the package is a description file of a project or module package. It contains information such as the name, description, version, and author. The **npm install** command downloads all dependent modules based on this file.

Example: **package.json**

```
{
  "name": "third_use",          //Package name
```

```
  "version": "0.0.1",           //Version number
"description": "this is a test project", // Description
  "main": "index.js",           //Entry file
  "scripts": {                  //Script commands
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [                 //Keyword
    "show"
  ],
  "author": "f",                //Developer name
  "license": "ISC",             //License agreement
  "dependencies": {             //Project production dependencies
    "jquery": "^3.6.0",
    "mysql": "^2.18.1"
  },
  "devDependencies": {          //Project development dependencies
    "less": "^4.1.2",
    "sass": "^1.45.0"
  }
}
```

The **name** and **version** are the most important fields and must be included. Otherwise, the current package cannot be installed. The two attributes together form the unique identifier of an npm package.

name indicates the name of the package. The first part of the name, such as **@scope**, is used as the namespace. The other part is **name**. Generally, you can search with the **name** field to install and use the required package.

```
{
  "name": "@scope/name"
}
```

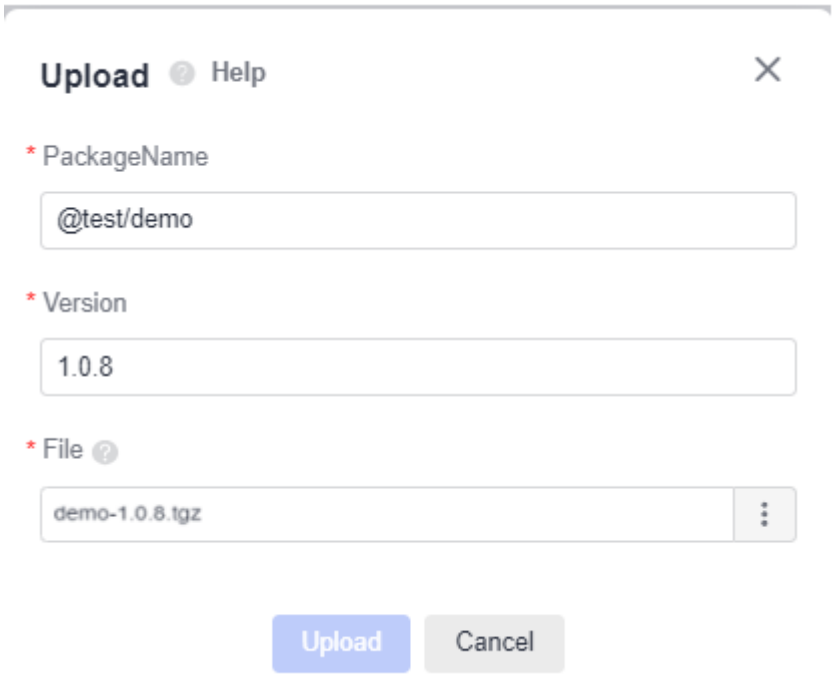
version indicates the version of the package, which is in the x.y.z format.

```
{
  "version": "1.0.0"
}
```

Uploading an npm Package

npm packages in .tgz format can be uploaded to self-hosted repos. When uploading packages, you need to set the following parameters.

Parameter	Description
PackageName	The value must be the same as that of **name** in the **package.json** file.
Version	The value must be the same as that of **version** in the **package.json** file.



NOTE

When uploading a package, ensure that the package name starts with a path in the path list added during repository creation. For details, see **Configuring Repository Items** in the help guide.

Example:

The path **@test** is added when you create an npm registry.

When uploading the package to the registry, make sure that the **PackageName** starts with **@test**. If a path outside the path list is used, for example, **@npm**, the upload will fail.

After the upload is successful, you can view the package in .tgz format in the repository package list and the corresponding metadata is generated in the **.npm** directory.

Uploading a Go Package

Go, also known as Golang, is a programming language developed by Google. Golang 1.11 and later support modular package management. A module is a unit for source code exchange and versioning in Go. A MOD file identifies and manages a module. A ZIP file is a source code package. There are two types of Go modules: v2.0 and later and v2.0 and earlier. The management of the Go module is different between these two versions.

To upload a Go package, you need to upload both a ZIP file and a MOD file and set the following parameters.

Parameter	Description
Zip Path	Complete path of the ZIP file. Valid path formats are: ● Versions earlier than v2.0: *{moduleName}*/@v/*{version}*.zip ● Versions later than v2.0: – If the ZIP file contains **go.mod** and the path ends with **/vN**, the file path format is: *{moduleName}*/vX/@v/vX.X.X.zip – If the ZIP file does not contain **go.mod** or the first line in **go.mod** does not end with **/vN**, the file path format is: *{moduleName}*/@v/vX.X.X+incompatible.zip
Zip File	Directory structure of the ZIP file. Valid directory structure formats are: ● Versions earlier than v2.0: *{moduleName}*@*{version}* ● Versions later than v2.0: – If the ZIP file contains **go.mod** and the path ends with **/vN**, the directory structure format is: *{moduleName}*/vX@*{version}* – If the ZIP file does not contain **go.mod** or the first line in **go.mod** does not end with **/vN**, the directory structure format is: *{moduleName}*@*{version}*+incompatible
Mod Path	Complete path of the MOD file. Valid path formats are: ● Versions earlier than v2.0: *{moduleName}*/@v/*{version}*.mod ● Versions later than v2.0: – If the ZIP file contains **go.mod** and the path ends with **/vN**, the file path format is: *{moduleName}*/vX/@v/vX.X.X.mod – If the ZIP file does not contain **go.mod** or the first line in **go.mod** does not end with **/vN**, the file path format is: *{moduleName}*/@v/vX.X.X+incompatible.mod
Mod File	MOD file content. Valid content formats are: ● Versions earlier than v2.0: module *{moduleName}* ● Versions later than v2.0: – If the ZIP file contains **go.mod** and the path ends with **/vN**, the content format is: module *{moduleName}*/vX – If the ZIP file does not contain **go.mod** or the first line in **go.mod** does not end with **/vN**, the content format is: module *{moduleName}*

Uploading a PyPI Package

You are advised to go to the project directory (which must contain the **setup.py** configuration file) and run the following command to compress the packages to

be uploaded into a wheel (.whl) installation package. By default, the installation package is generated in the **dist** directory of the project directory. The Python software package management tool pip supports only wheel installation packages.

```
python setup.py sdist bdist_wheel
```

You need to set the following parameters.

Parameter	Description
PackageName	The value must be the same as that of **name** in the **setup.py** file.
Version	The value must be the same as that of **version** in the **setup.py** file.

After the upload is successful, you can view the installation package in **.whl** format in the repository package list. In addition, the corresponding metadata is generated in the **.pypi** directory, which can be used for pip installation.

Uploading an RPM Package

Introduction to RPM

- Red Hat Package Manager (RPM), developed by Red Hat, is used by many Linux distributions. It is a software management system that installs software on Linux in database recording mode.
- You are advised to package and name the RPM binary file according to the following rules:

Software name-*Main version number of the software*.*Minor version number of the software*.*Software revision number*-*Number of software compilation times*.*Hardware platform suitable for the software*.**rpm**

For example, **hello-0.17.2-54.x86_64.rpm**. **hello** is the software name, **0** is the major version number of the software, **17** is the minor version number, **2** is the revision number, **54** is the number of times that the software is compiled, and **x86_64** is the hardware platform suitable for the software.

Software Name	Major Version	Minor Version	Revision No.	Compilation Times	Applicable Hardware Platform
hello	0	17	2	54	x86_64

Note: You need to set the following parameters when uploading packages.

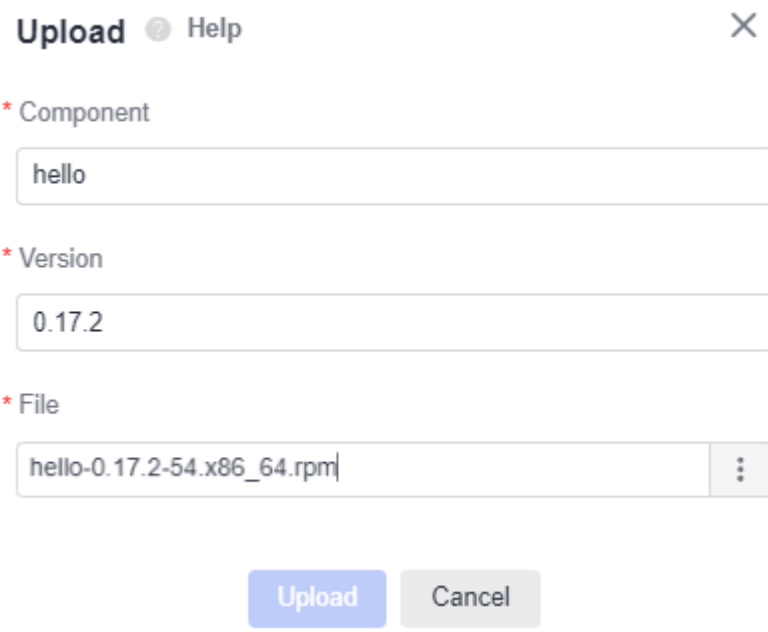
Parameter	Description
Component	Component name

Parameter	Description
Version	Version of the RPM binary package

Step 1 Go to the self-hosted repo page. In the left pane, click the target repository to which the private package is to be uploaded.

Step 2 Click **Upload** on the right of the page.

Step 3 Set the package parameters, select the file, and click **Upload**.



----End

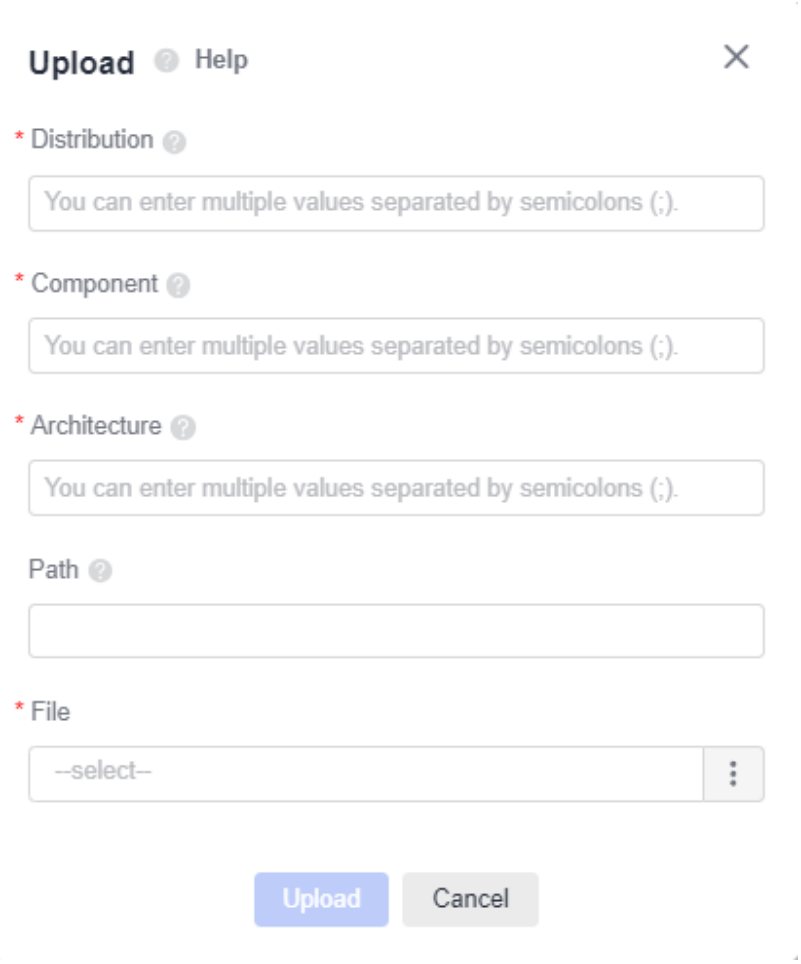
After the upload is successful, you can find the RPM binary package in the repository list and the corresponding metadata **repodata** directory is generated in the package name directory. You can use Yum to install the package.

Uploading a Debian Package

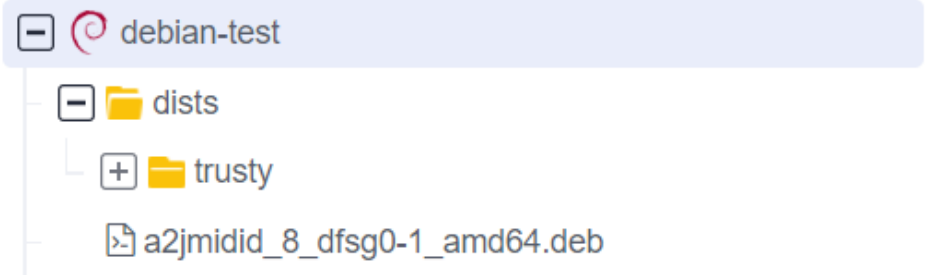
When uploading a Debian package, you need to set the following parameters:

Parameter	Description
Distribution	Release version of the package
Component	Name of the package
Architecture	Package architecture

Parameter	Description
Path	Path for storing the package. By default, the package is uploaded to the root path.
File	Local storage path of the package to be uploaded



After the upload is successful, you can view the installation package in **.deb** format in the repository package list. In addition, the corresponding metadata is generated in the **dists** directory, which can be used for Debian installation.



Uploading a NuGet Package

A NuGet package is a single ZIP file with a **.nupkg** extension. As a shareable unit of code, developers can publish it to a dedicated server to share it with other team members.

CodeArts Artifact creates a NuGet repository to host the NuGet package.

- You are advised to package and name the NuGet file according to the following rules:

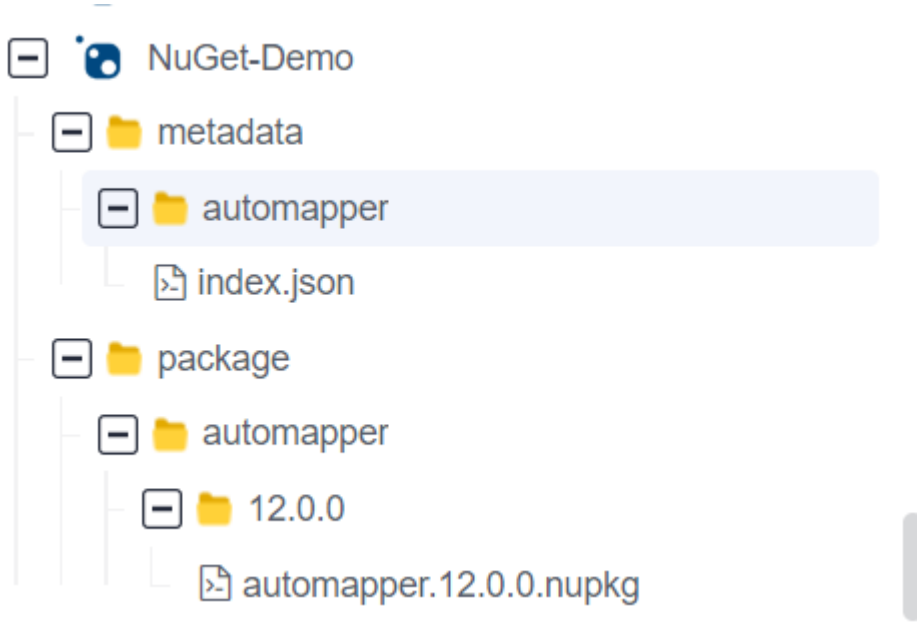
  Software name-*Major version number of the software*.**nupkg**

  Example: **automapper.12.0.0.nupkg**

Step 1 Go to the self-hosted repo page. In the left pane, click the target NuGet repository to which the private package is to be uploaded.

Step 2 Click **Upload** on the right of the page, select the NuGet file to be uploaded from the localhost, and click **Upload**.

Step 3 View packages that are successfully uploaded in the repository list.



metadata stores metadata and is named after the package name. **metadata** cannot be deleted manually. It will be deleted or added automatically when the corresponding package is deleted or restored.

package stores components.

----End

Uploading a Docker Image

Step 1 Go to the self-hosted repo page. In the left pane, click the target Docker registry to which the private image is to be uploaded.

Step 2 Click **Upload** and configure page information (for details, see the following table).

Parameter	Description
Upload Mode	The default upload mode is **Single file**.
PackageName	Enter an image name.
File	Select the Docker file to be uploaded from the localhost.

Step 3 Click **Upload**.

----End

Uploading a CocoaPods Package

Step 1 Access the self-hosted repo homepage. In the left pane, click the target CocoaPods repository to which the private package is to be uploaded.

Step 2 Click **Upload** and configure page information (for details, see the following table).

Parameter	Description
Upload Mode	The default upload mode is **Single file**.
PackageName	Enter an artifact package name.
File	Select the CocoaPods file to be uploaded from the localhost.

Step 3 Click **Upload**.

----End

Uploading an OHPM Package

OHPM packages

OpenHarmony Package Manager (OHPM) is an ArkTS package management tool. OHPM manages packages, which are stored and organized in the OHPM repository.

The OHPM package contains the Harmony Archive (HAR) package and Harmony Shared Package (HSP) package.

- HAR: Static shared package, which is reused in the build phase. It is mainly used as a second-party or third-party library for other applications to depend on. It includes resource files, *.so files, and metadata files.
- HSP: Dynamic shared package, which is reused in the running phase. It provides shared code and resources to improve code reusability and maintainability.

The metadata file, **oh-package.json5**, in the HAR package describes the component, including its name, version, and dependency information. The **ohpm install** command automatically reads the dependencies in **oh-package.json5** and downloads them.

Example: **oh-package.json5**

```
{
  "name": "@scope/demo",                  // Component name
  "version": "0.0.1",                  // Version
  "author": "artifact",                     // Author
  "license": "Apache-2.0",                  // License agreement
  "main": "index.ts",                     // Entry file
  "description": "basic information.",      // Description
  "dependencies": {                         // Dependency
    "@scope/demo1": "1.0.0",
    "@scope/demo2": "file:./demo2.har"
  },
  "devDependencies": {                      // Development dependency
    "@scope/demo3": "1.0.0",
    "@scope/demo4": "1.0.0"
  },
  "metadata": {                           // Metadata information
    "sourceRoots": ["./src/main"]
  }
}
```

The **name** and **version** are the most important fields and must be included. Otherwise, the current package cannot be installed. The two attributes together form the unique identifier of an OHPM package.

name indicates the name of the package. The first part of the name, such as **@scope**, is used as the namespace. The other part is **name**. Generally, you can search with the **name** field to install and use the required package.

```
{
  "name": "@scope/name"
}
```

version indicates the version of the package, which is in the x.y.z format.

```
{
  "version": "1.0.0"
}
```

Uploading an OHPM package

You can upload OHPM packages in .har or .hsp format to the self-hosted repo. The procedure is as follows:

Step 1 Access the self-hosted repo page. In the left pane, click the target OHPM repository to which the private package is to be uploaded.

Step 2 Click **Upload** and configure page information (for details, see the following table).

Parameter	Description
Upload Mode	The default upload mode is **Single file**.
PackageName	Enter an artifact package name. The value must be the same as that of **name** in the **oh-package.json5** file. **NOTE** Ensure that the **PackageName** starts with a path in the path list added during repository creation. For details, see **Configuring Repository Items** in the help guide. Example: The path **@test** is added when you create an OHPM repository. When uploading the package to the repository, make sure that the **PackageName** starts with **@test**. If a path outside the path list is used, for example, **@ohos**, the upload will fail.

Parameter	Description
Version	Enter the version number. The value must be the same as that of **version** in the **oh-package.json5** file.
File	Select the OHPM file to be uploaded from the localhost.

Step 3 Click **Upload**.

When the progress bar in the lower-right corner of the page shows 100%, it indicates the upload is complete.

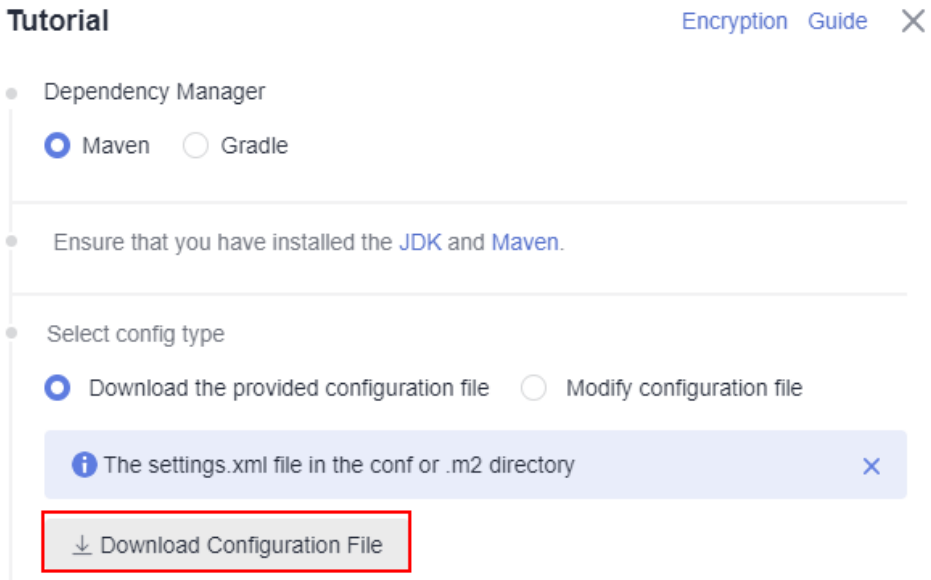
You can find the package in the repository list and the corresponding metadata is generated in the **.ohpm** directory.

----End

3.2.7 Uploading and Downloading Packages on the Client

Uploading a Maven Artifact from the Client

- The client tool is Maven. Ensure that the JDK and Maven have been installed.
  a. Download the **settings.xml** file from the self-hosted repo page and replace the downloaded configuration file with the new one or modify the **settings.xml** file of Maven as prompted.



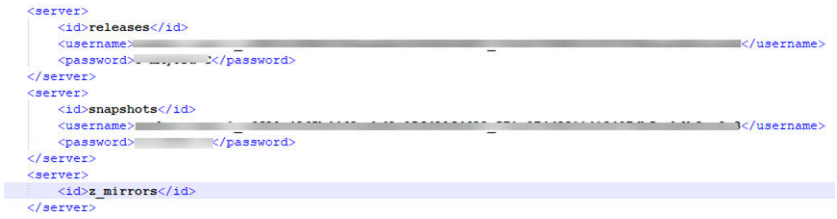
  b. Run the following command to upload a package from the client.

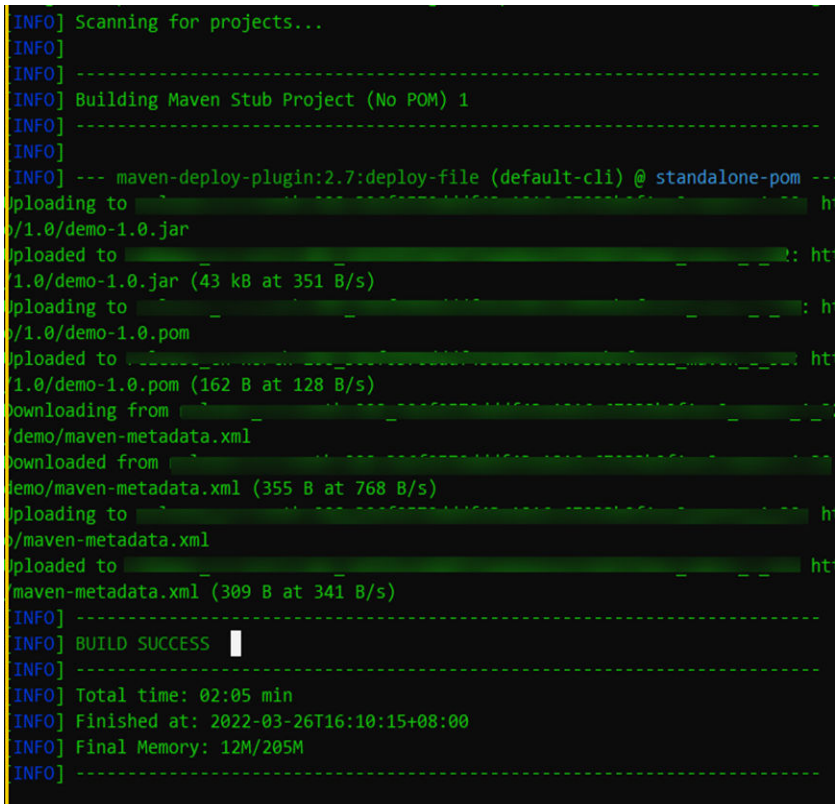
  NOTE

  Run this command in the directory where the uploaded POM file is located.

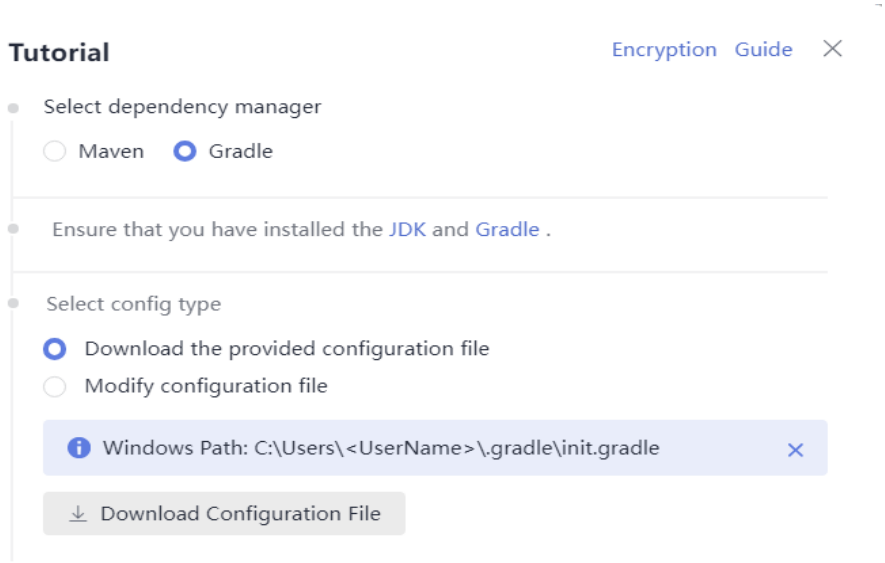
```
mvn deploy:deploy-file -DgroupId={groupId} -DartifactId={artifactId} -Dversion={version} -Dpackaging=jar -Dfile={file_path} -DpomFile={pom_path} -Durl={url} -DrepositoryId={repositoryId} -s {settings_path} -Dmaven.wagon.http.ssl.insecure=true -Dmaven.wagon.http.ssl.allowall=true -Dmaven.wagon.http.ssl.ignore.validity.dates=true
```

- **Description**
  - *DgroupId*: uploaded group ID
  - *DartifactId*: uploaded artifact ID
  - *Dversion*: version of the uploaded file
  - *Dpackaging*: type of the package to be uploaded, such as JAR, ZIP, and WAR
  - *Dfile*: path of the uploaded entity file
  - *DpomFile*: path of the entity POM file to be uploaded. (For a release version, if this parameter does not exist, the system automatically generates a POM file. If there are special requirements for the POM file, specify this parameter.)
  - The values of *DgroupId*, *DartifactId*, and *Dversion* in the POM file must be the same as those outside the POM file. Otherwise, error 409 is reported.
  - Select either *DpomFile* or one of *DgroupId*, *DartifactId*, and *Dversion*.
  - *Durl*: path for uploading files to the repository.
  - *DrepositoryId*: ID corresponding to the username and password configured in settings, as shown in the following figure.





- The client tool is Gradle. Ensure that JDK and Gradle have been installed.
  a. Download the **inti.gradle** file from the self-hosted repo page.



  b. Find the **build.gradle** file in the local project and add the following commands to the **gradle** file:

```
uploadArchives {
    repositories {
        mavenDeployer {repository(url:"***") {
                authentication(userName: "{repo_name}", password: "{repo_password}")
            }
            // Pom file of the build project
            pom.project {
                name = project.name
                packaging = 'jar'
                description = 'description'
            }
        }
    }
}
```

     - *url*: path for uploading files to the repository. You can click on the Maven repository page to obtain the path.
     - *{repo_name}*: username obtained from the **inti.gradle** file downloaded from the Maven repository page.
     - *{repo_password}*: password obtained from the **inti.gradle** file downloaded from the Maven repository page.

  c. Run the following command in the directory where the local project is located:

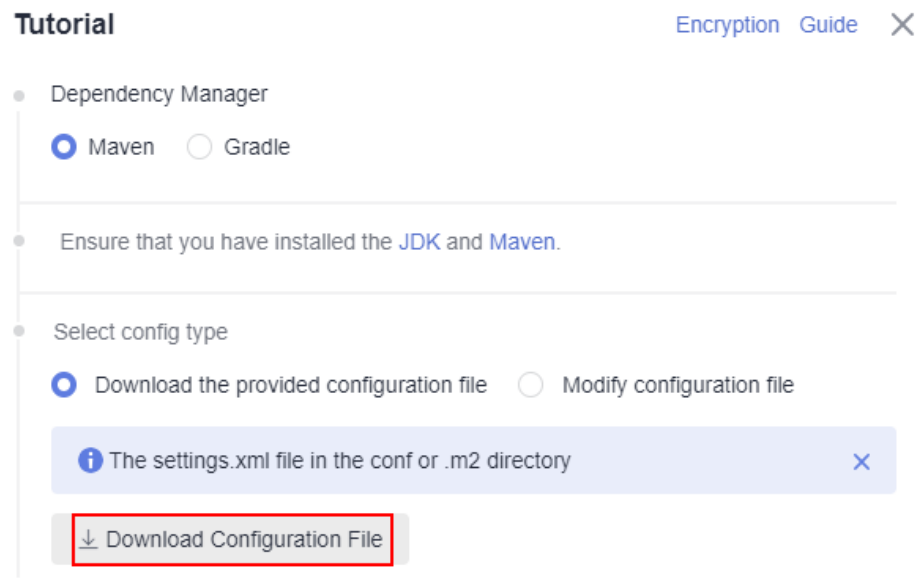
```
gradle uploadArchives
```

  d. Return to the target Maven repository to view the uploaded artifacts.

Downloading a Maven Artifact to the Client

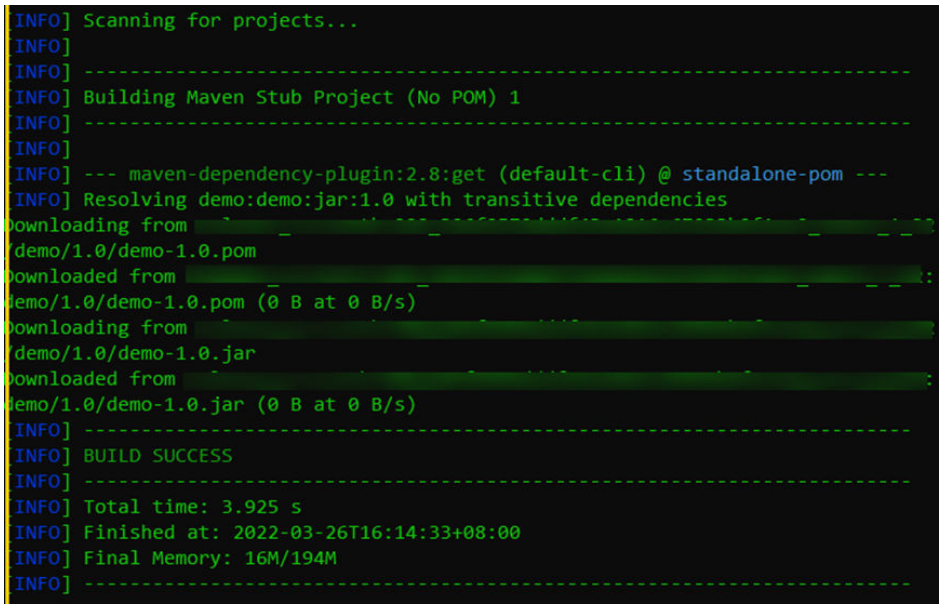
- The client tool is Maven. Ensure that the JDK and Maven have been installed.

  1. Download the **settings.xml** file from the self-hosted repo page and replace the downloaded configuration file with the new one or modify the **settings.xml** file of Maven as prompted.



2. Download the client.

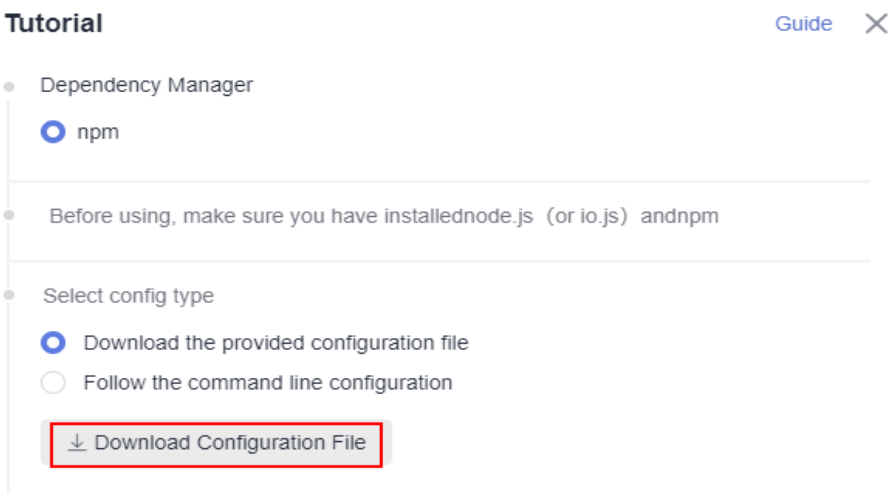
```
mvn dependency:get -DremoteRepositories={repo_url} -DgroupId={groupId} -DartifactId={artifactId} -Dversion={version} -Dmaven.wagon.http.ssl.insecure=true -Dmaven.wagon.http.ssl.allowall=true -Dmaven.wagon.http.ssl.ignore.validity.dates=true
```



Uploading an npm Package from the Client

- The client tool is npm. Ensure that **Node.js** (or **io.js**) and npm have been installed.

  1. Download the NPMRC file from the self-hosted repo page and save the downloaded NPMRC file as an **.npmrc** file.



2. Copy the file to the user directory. In Linux, the path is **~/.npmrc** (C:\Users\<*UserName*>\.npmrc).

3. Go to the npm project directory (where the **package.json** file is stored), open the **package.json** file, and add the path information entered during repository creation to the value of the name field.

```
{
  "name": "@test/demo",
  "version": "1.0.0",
  "description": "demo",
  "main": "index.js",
  "engines": {
    "node": ">= 8.0.0",
    "npm": ">= 5.0.0"
  },
```

4. Upload the npm package to the repository:

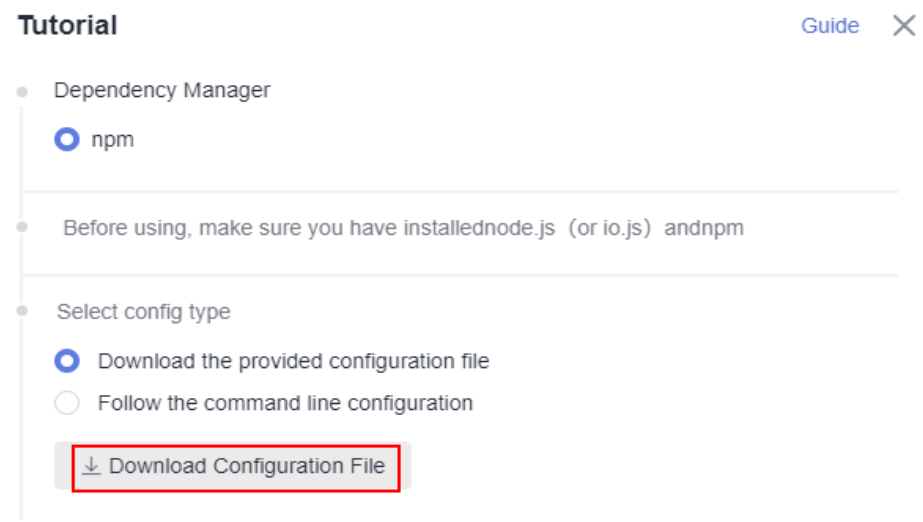
```
npm config set strict-ssl false
npm publish
```

```
npm notice === Tarball Details ===
npm notice name:          @test/demo
npm notice version:       1.0.0
npm notice package size:  8.7 MB
npm notice unpacked size: 10.6 MB
npm notice shasum:
npm notice integrity:
npm notice total files:   102
npm notice
+ @test/demo@1.0.0
```

Downloading an npm Package to the Client

- The client tool is npm. Ensure that **Node.js** (or **io.js**) and npm have been installed.

1. Download the NPMRC file from the self-hosted repo page and save the downloaded NPMRC file as an **.npmrc** file.



2. Copy the file to the user directory. In Linux, the path is **~/.npmrc**. In Windows, the path is **C:\Users\<*UserName*>\.npmrc**.

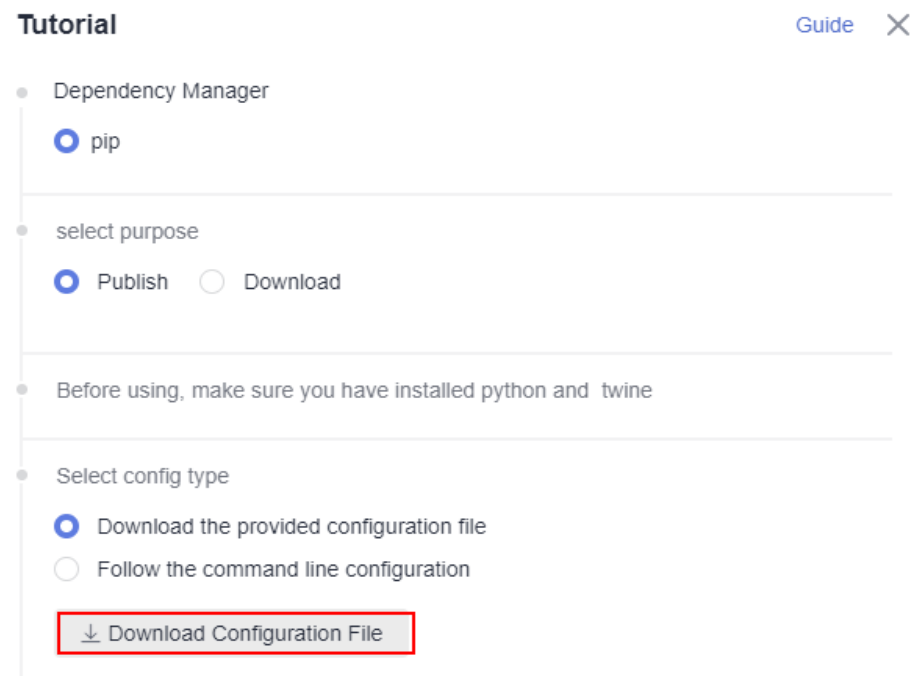
3. Go to the npm project directory (where the **package.json** file is stored) and run the following commands to download the npm dependency:

```
npm config set strict-ssl false
npm install ––verbose
```

Uploading a PyPI Package from the Client

- The client tools are python and twine. Ensure that python and twine have been installed.

  1. Download the PYPIRC file from the self-hosted repo page and save the downloaded PYPIRC file as a **.pypirc** file.



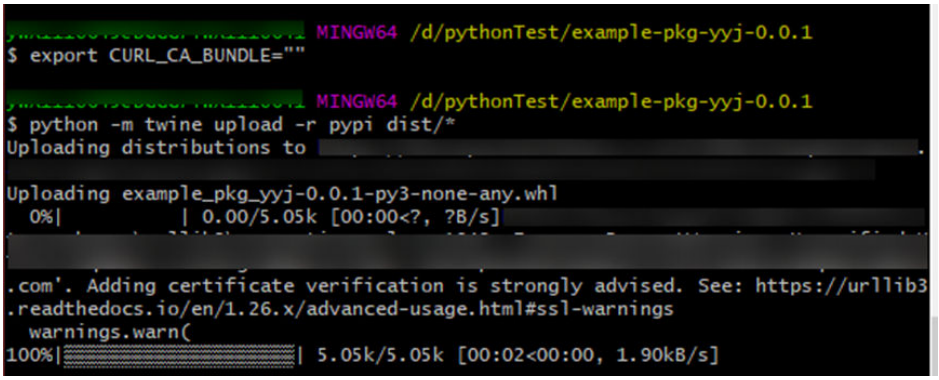
2. Copy the file to the user directory. In Linux, the path is **~/.pypirc**. In Windows, the path is **C:\Users**<*UserName*>**\.pypirc**.

3. Go to the Python project directory and run the following command to compress the Python project into a **.whl** package:

```
python setup.py bdist_wheel
```

4. Upload the file to the repository.

```
python -m twine upload -r pypi dist/*
```



If a certificate error is reported during the upload, run the following command (use git bash in Windows) to set environment variables to skip certificate verification.

```
export CURL_CA_BUNDLE=""
```

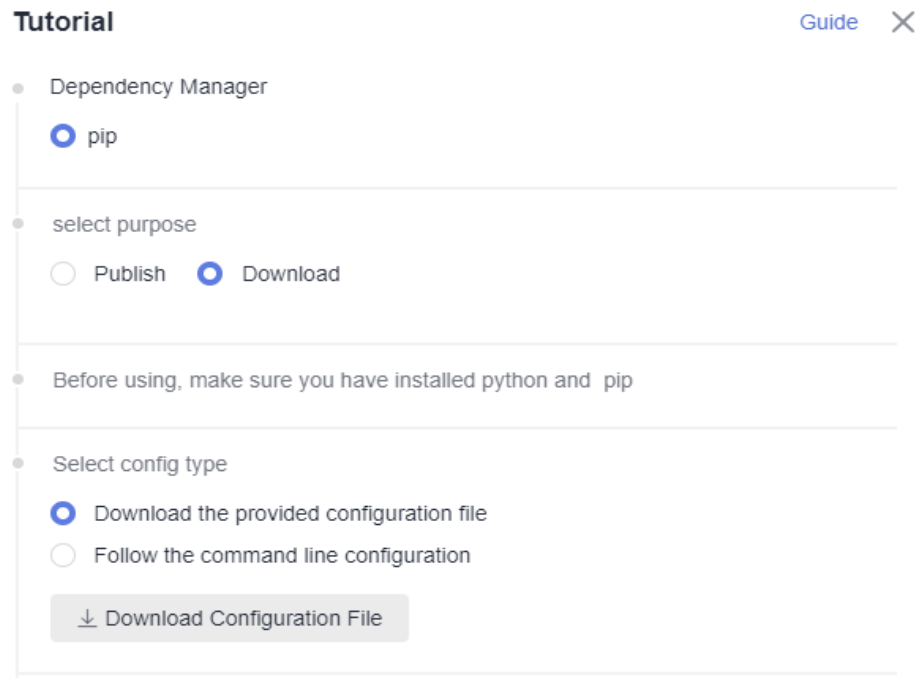
NOTE

The environment variables will be reset if you log in to the server again, switch users, or open a new bash window. Be sure to set the environment variables before each upload.

Downloading a PyPI Package to the Client

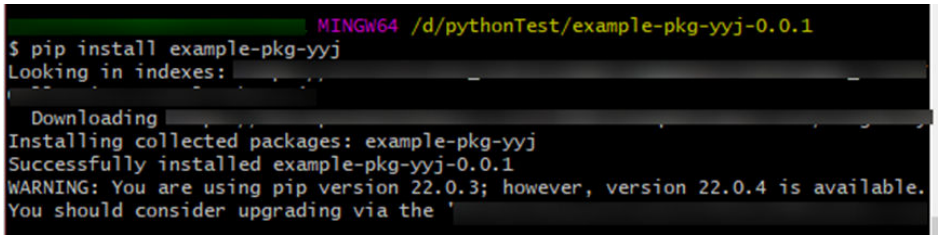
- The client tools are Python and PIP. Ensure that Python and PIP have been installed.

1. Download the **pip.ini** file from the self-hosted repo page and copy the file to the user directory. In Linux, the path is **~/.pip/pip.conf** (**C:\Users\<*UserName*>\pip\pip.ini** on Windows)



2. Install the Python package:

```
pip install {package name}
```



Uploading and Downloading a Go Package on the Client

The client tool is Go. Ensure that V1.13 or a later version has been installed and the project is a Go module project.

- Go Modules packaging mode and package upload

  This section describes how to build and upload Go packages through Go modules. The username and password used in the following steps can be obtained from the downloaded configuration file for the Go repository.

  Perform the following steps:

  a. Create a source folder in the working directory.

  ```
  mkdir -p {module}@{version}
  ```

  b. Copy the code source to the source folder.

  ```
  cp -rf . {module}@{version}
  ```

  c. Compress the package into a ZIP package.

  ```
  zip -D -r [package name] [package root directory]
  ```

  d. Upload the ZIP package and the **go.mod** file to the self-hosted repo.

  ```
  curl -k -u {{username}}:{{password}} -X PUT {{repoUrl}}/{filePath} -T {{localFile}}
  ```

The package directory varies according to the package version. The version can be:

- Versions earlier than v2.0: The directory is the same as the path of the **go.mod** file. No special directory is required.
- v2.0 or later:
  - If the first line in the **go.mod** file ends with **/vX**, the directory must contain **/vX**. For example, if the version is v2.0.1, the directory must contain **v2**.
  - If the first line in the **go.mod** file does not end with **/vN**, the directory remains unchanged and the name of the file to be uploaded must contain **+incompatible**.

  The following are examples of package directories for different versions:

Versions earlier than v2.0

The **go.mod** file is used as an example.

go.mod

```
1  module example.com/demo
```

a. Create a source folder in the working directory.

   The value of *module* is **example.com/demo** and that of *version* is **1.0.0**. The command is as follows:

   ```
   mkdir -p ~/example.com/demo@v1.0.0
   ```

b. Copy the code source to the source folder.

   The command is as follows (with the same parameter values as the previous command):

   ```
   cp -rf . ~/example.com/demo@v1.0.0/
   ```

c. Compress the package into a ZIP package.

   Run the following command to go to the upper-level directory of the root directory where the ZIP package is located:

   ```
   cd ~
   ```

   Then, use the **zip** command to compress the code into a package. In this command, the package root directory is **example.com** and the package name is **v1.0.0.zip**. The command is as follows:

   ```
   zip -D -r v1.0.0.zip  example.com/
   ```

d. Upload the ZIP package and the **go.mod** file to the self-hosted repo.

   Parameters *username*, *password*, and *repoUrl* can be obtained from the configuration file of the self-hosted repo.

   - For the ZIP package, the value of *filePath* is **example.com/demo/@v/v1.0.0.zip** and that of *localFile* is **v1.0.0.zip**.
   - For the **go.mod** file, the value of *filePath* is **example.com/demo/@v/v1.0.0.mod** and that of *localFile* is **example.com/demo@v1.0.0/go.mod**.

   The commands are as follows (replace *username*, password, and *repoUrl* with the actual values):

```
curl -k -u {{username}}:{{password}} -X PUT {{repoUrl}}/example.com/demo/@v/v1.0.0.zip -T v1.0.0.zip
curl -k -u {{username}}:{{password}} -X PUT {{repoUrl}}/example.com/demo/@v/v1.0.0.mod -T example.com/demo@v1.0.0/go.mod
```

- **v2.0 and later, with the first line in the go.mod file ends with /vX**.

  The **go.mod** file is used as an example.

  go.mod

  ```
  1  module example.com/demo/v2
  ```

  a. Create a source folder in the working directory.

     The value of *module* is **example.com/demo/v2** and that of *version* is **2.0.0**. The command is as follows:

     ```
     mkdir -p ~/example.com/demo/v2@v2.0.0
     ```

  b. Copy the code source to the source folder.

     The command is as follows (with the same parameter values as the previous command):

     ```
     cp -rf . ~/example.com/demo/v2@v2.0.0/
     ```

  c. Compress the package into a ZIP package.

     Run the following command to go to the upper-level directory of the root directory where the ZIP package is located:

     ```
     cd ~
     ```

     Then, use the **zip** command to compress the code into a package. In this command, the package root directory is **example.com** and the package name is **v2.0.0.zip**. The command is as follows:

     ```
     zip -D -r v2.0.0.zip  example.com/
     ```

  d. Upload the ZIP package and the **go.mod** file to the self-hosted repo.

     Parameters *username*, *password*, and *repoUrl* can be obtained from the configuration file of the self-hosted repo.

     - For the ZIP package, the value of *filePath* is **example.com/demo/v2/@v/v2.0.0.zip** and that of *localFile* is **v2.0.0.zip**.
     - For the **go.mod** file, the value of *filePath* is **example.com/demo/v2/@v/v2.0.0.mod** and that of *localFile* is **example.com/demo/v2@v2.0.0/go.mod**.

     The commands are as follows (replace *username*, password, and *repoUrl* with the actual values):

     ```
     curl -u {{username}}:{{password}} -X PUT {{repoUrl}}/example.com/demo/v2/@v/v2.0.0.zip -T v2.0.0.zip
     curl -u {{username}}:{{password}} -X PUT {{repoUrl}}/example.com/demo/v2/@v/v2.0.0.mod -T example.com/demo/v2@v2.0.0/go.mod
     ```

- **v2.0 and later, with the first line in the go.mod file does not end with /vX**.

  The **go.mod** file is used as an example.

  go.mod

  ```
  1  module example.com/demo
  ```

  a. Create a source folder in the working directory.

The value of *module* is **example.com/demo** and that of *version* is **3.0.0**. The command is as follows:

```
mkdir -p ~/example.com/demo@v3.0.0+incompatible
```

b. Copy the code source to the source folder.

The command is as follows (with the same parameter values as the previous command):

```
cp -rf . ~/example.com/demo@v3.0.0+incompatible/
```

c. Compress the package into a ZIP package.

Run the following command to go to the upper-level directory of the root directory where the ZIP package is located:

```
cd ~
```

Then, use the **zip** command to compress the code into a package. In this command, the package root directory is **example.com** and the package name is **v3.0.0.zip**. The command is as follows:

```
zip -D -r v3.0.0.zip  example.com/
```

d. Upload the ZIP package and the **go.mod** file to the self-hosted repo.

Parameters *username*, *password*, and *repoUrl* can be obtained from the configuration file of the self-hosted repo.

- For the ZIP package, the value of *filePath* is **example.com/demo/@v/v3.0.0+incompatible.zip** and that of *localFile* is **v3.0.0.zip**.
- For the **go.mod** file, the value of *filePath* is **example.com/demo/@v/v3.0.0+incompatible.mod** and that of *localFile* is **example.com/demo@v3.0.0+incompatible/go.mod**.

The commands are as follows (replace *username*, password, and *repoUrl* with the actual values):

```
curl -k -u {{username}}:{{password}} -X PUT {{repoUrl}}/example.com/demo/@v/
v3.0.0+incompatible.zip -T v3.0.0.zip
curl -k -u {{username}}:{{password}} -X PUT {{repoUrl}}/example.com/demo/@v/
v3.0.0+incompatible.mod -T example.com/demo@v3.0.0+incompatible/go.mod
```

- **Downloading a Go package using the Go client**

  Certificate verification cannot be bypassed on the Go client. You need to add the domain name certificate corresponding to the self-hosted repo to the local certificate trust list and perform the following steps to add the trust certificate:

  1) Export the certificates.

  ```
  openssl s_client -connect {host}:443 -showcerts </dev/null 2>/dev/null | sed -ne '/-
  BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' |openssl x509 -outform PEM
  >mycertfile.pem
  openssl x509 -outform der -in mycertfile.pem -out mycertfile.crt
  ```

  mycertfile.pem and **mycertfile.crt** are the downloaded certificates.

  2) Add the certificates to the root certificate trust list.

  3) Run the go commands to download the dependency:

  ```
  ##1. Packages of versions earlier than v2.0
  go get -v <moudlename>
  ##2. v2.0 and later versions
  ##a. The ZIP package contains go.mod and the path ends with /vN.
  go get -v {{moduleName}}/vN@{{version}}
  ##b. The ZIP package does not contain go.mod or the first line in go.mod does not
  ```

```
end with /vN.
go get -v {moduleName}}@{{version}}+incompatible
```

Uploading and Downloading an RPM Package on the Client

The client tool is Linux OS with Yum. Ensure that the Linux OS is used and Yum has been installed.

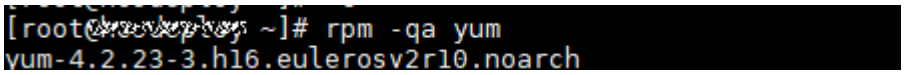
- **Releasing a package to an RPM repository**

Step 1 Check whether the yum tool is installed in Linux.

On the Linux host, run the following command:

```
rpm -qa yum
```

If the following information is displayed, yum has been installed on the server:



Step 2 Log in to the CodeArts homepage and access the self-hosted repo for RPM. Click **Tutorial** on the right of the page.

Step 3 In the displayed dialog box, click **Download Configuration File**.

Step 4 On the Linux host, upload an RPM package:

```
curl -k -u {{user}}:{{password}} -X PUT https://{{repoUrl}}/{{component}}/{{version}}/ -T {{localFile}}
```

In this command, *user*, *password*, and *repoUrl* can be obtained from the **RPM upload command** in the configuration file downloaded in the **previous step**.

- *user*: character string before the colon (:) between **curl -u** and **-X**
- *password*: character string after the colon (:) between **curl -u** and **-X**
- *repoUrl*: character string between **https://** and **/{{component}}**

  component, *version*, and *localFile* can be obtained from the RPM package to be uploaded. The **hello-0.17.2-54.x86_64.rpm** package is used as an example.
  - *component*: software name, for example, **hello**.
  - *version*: software version, for example, **0.17.2**.
  - *localFile*: RPM package, for example, **hello-0.17.2-54.x86_64.rpm**.

    The following figure shows the complete command.

    After the command is successfully executed, go to the self-hosted repo and find the uploaded RPM package.

----End

Uploading and Downloading a Debian Package on the Client

Step 1 Before downloading the package, save the obtained public key information to the **public.asc** file and add the information to the system key list.

```
gpg --import <PUBLIC_KEY>
gpg --list-signatures
gpg --export --armor <SIG_ID> | apt-key add -
```

Step 2 Upload the package.

```
curl -k -u "<USERNAME>:<PASSWORD>" -X PUT "https://<Debian repository URL>/<DEBIAN_PACKAGE_NAME>;deb.distribution=<DISTRIBUTION>;deb.component=<COMPONENT>;deb.architecture=<ARCHITECTURE>" -T <PATH_TO_FILE>
```

Step 3 Download the package.

```
apt-get install <Package name> -reinstall -d
```

----End

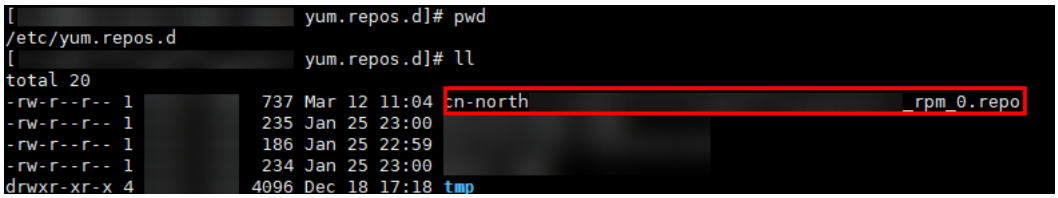
Obtaining a Dependency from an RPM Repository

The following section uses the RPM private package released in **#li18941713182313** as an example to describe how to obtain dependencies from the RPM repository.

Step 1 Download the configuration file of the RPM repository by referring to **Step 2** and **Step 3** of the released RPM package.

Step 2 Open the configuration file, replace all *{{component}}* in the file with the value of *{{component}}* (**hello** in this file) used for uploading the RPM file, delete the **RPM upload command**, and save the file.

Step 3 Save the modified configuration file to the **/etc/yum.repos.d/** directory on the Linux host.



Step 4 Run the following command to download the RPM package. Replace **hello** with the actual value of **component**.

```
yum install hello
```

----End

Uploading a Conan Package from the Client

Conan is a package manager for C and C++ developers. It applies to all operating systems, such as Windows, Linux, OSX, FreeBSD, and Solaris.

Prerequisites

- You have installed the Conan client.
- You have created a Conan repository in the self-hosted repo.

Step 1 Select the target Conan repository from the self-hosted repo page and click **Tutorial** to download the configuration file.

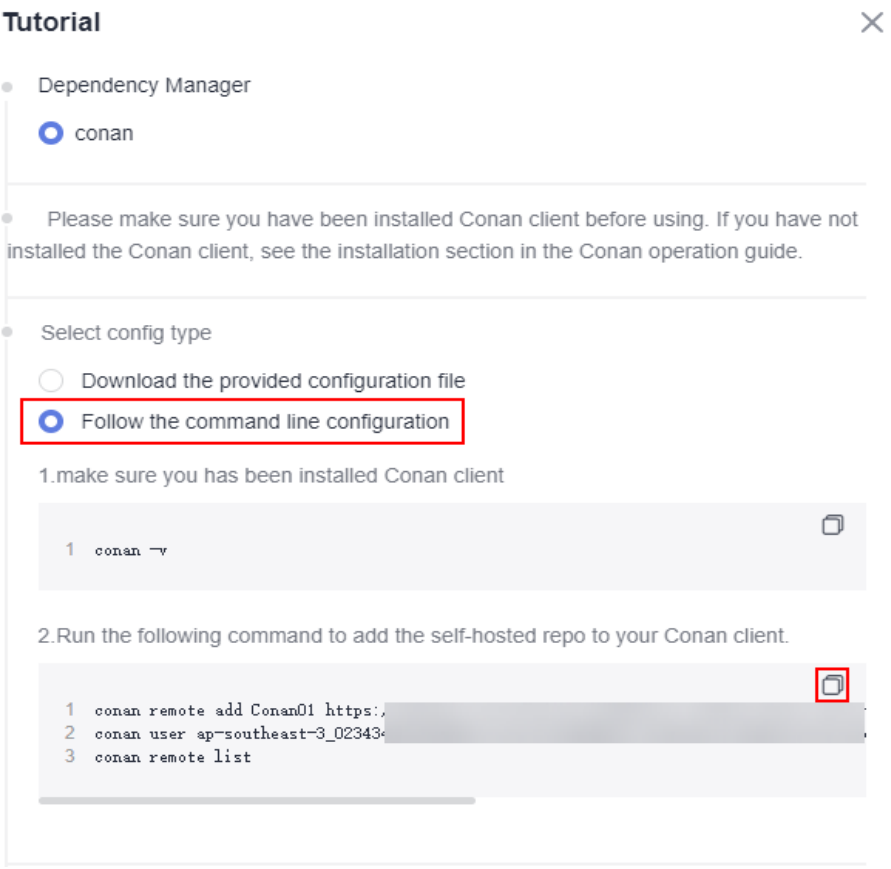
You can replace local Conan configurations with the obtained configuration file (the path is **~/.conan/remotes.json** in Linux or **C:\Users\<UserName>\.conan\remotes.json** in Windows).

Step 2 Copy and run the following commands on the configuration page to add the self-hosted repo to the local Conan client:

```
conan remote add Conan {repository_url}
conan user {user_name} -p={repo_password} -r=Conan
```

Check whether the remote repository has been configured on the Conan client.

```
conan remote list
```



Step 3 Upload all software packages to the remote repository. In the example, **my_local_server** is the remote repository. You can replace it with your own repository.

```
$ conan upload hello/0.1@demo/testing --all -r=my_local_server
```

Step 4 Check the software packages that have been uploaded to the remote repository.

```
$ conan search hello/0.1@demo/testing -r=my_local_server
```

----End

Downloading a Conan Package to the Client

Step 1 Select the target Conan repository from the self-hosted repo page and click **Tutorial** to download the configuration file.

You can replace local Conan configurations with the obtained configuration file (the path is **~/.conan/remotes.json** in Linux or **C:\Users\<UserName>\.conan\remotes.json** in Windows).

Step 2 Download the Conan dependency from the remote repository.

```
$ conan install ${package_name}/${package_version}@${package_username}/${channel} -r=cloud_artifact
```

Step 3 Check the downloaded Conan package.

```
$ conan search "*"
```

Step 4 Remove the package from the local cache.

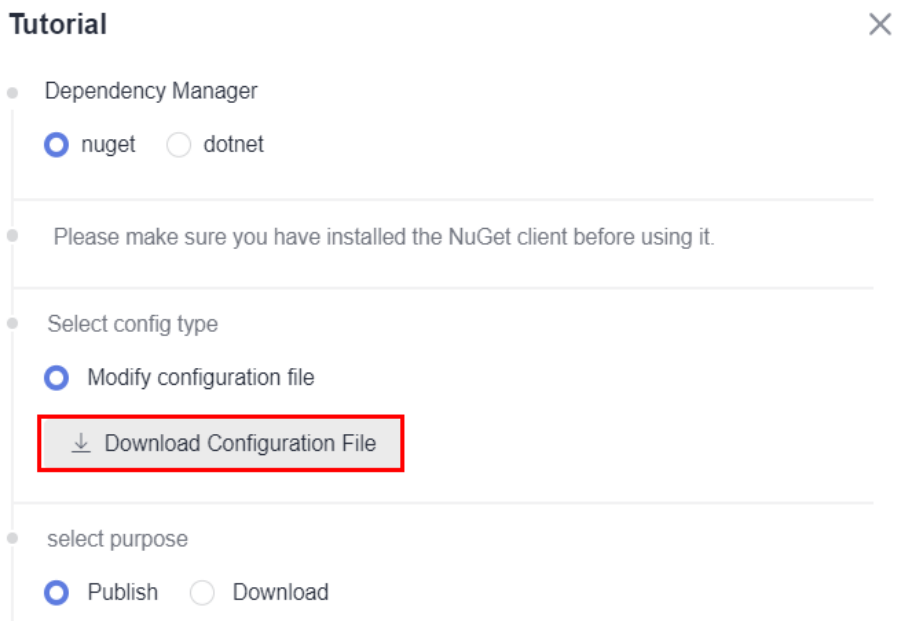
```
$ conan remove ${package_name}/${package_version}@${package_username}/${channel}
```

----End

Uploading a NuGet Package from the Client

Ensure that you have installed the NuGet.

Step 1 Select the target NuGet repository from the self-hosted repo page and click **Tutorial** to download the configuration file **NuGet.txt**.



Step 2 Open the downloaded file and run the following commands to add the source.

```
##----------------------NuGet add source----------------------##
nuget sources add -name {repo_name} -source {repo_url} -username {user_name} -password
{repo_password}
```

Step 3 Run the following commands to upload the package. Replace **<PATH_TO_FILE>** with the path of the file to be uploaded and run the upload statement. (If a configuration source exists, use the configured source name as the parameter following **-source**.)

```
##----------------------NuGet Download----------------------##
nuget push <PATH_TO_FILE> -source <SOURCE_NAME>
```

----End

Uploading a .NET Package from the Client

Ensure that you have installed the .NET.

NOTE

You can use the .NET client only after you have added the trusted server certificate.

- To obtain the Windows trust certificate, perform the following steps:

  1. Export the server certificate.

  ```
  openssl s_client -connect {host}:443 -showcerts </dev/null 2>/dev/null | sed -ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' |openssl x509 -outform PEM >mycertfile.pem
  openssl x509 -outform der -in mycertfile.pem -out mycertfile.crt
  ```

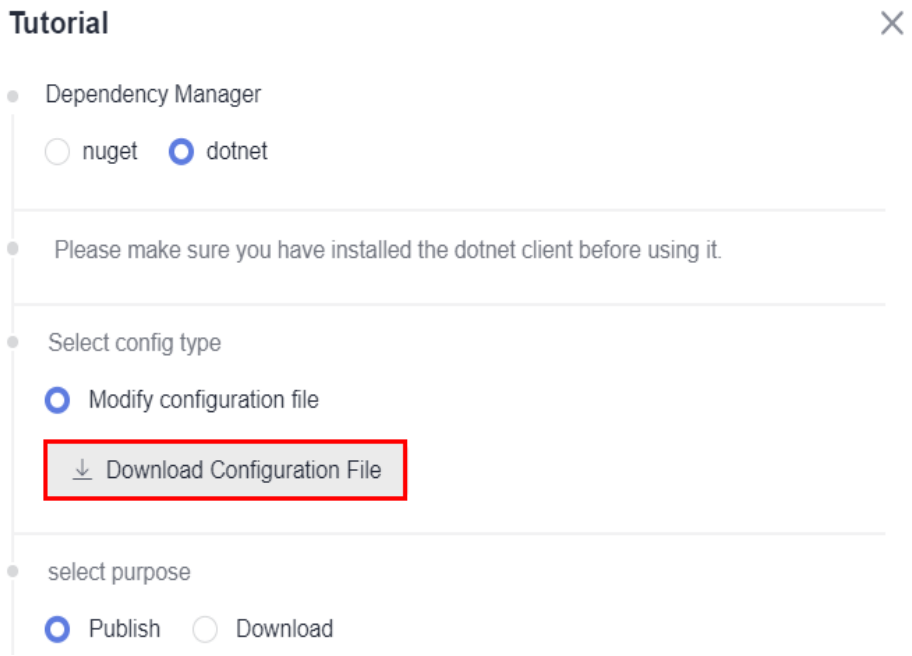
  mycertfile.pem and **mycertfile.crt** are the downloaded certificates.

  2. In Windows, you need to use PowerShell to add the certificate trust.

  Adding a certificate

  ```
  Import-Certificate -FilePath "mycertfile.crt" -CertStoreLocation cert:\CurrentUser\Root
  ```

Step 1 Select the target NuGet repository from the self-hosted repo page and click **Tutorial** to download the configuration file **dotnet.txt**.



Step 2 Open the configuration file, find the command under **dotnet add source**, and add the source.

```
##----------------------dotnet add source----------------------##
dotnet nuget add source {repo_url} add -n {repo_name} -u {user_name} -p {repo_password}
```

Step 3 Find the statement under **dotnet upload**, replace **<PATH_TO_FILE>** with the path of the file to be uploaded, and run the upload statement.

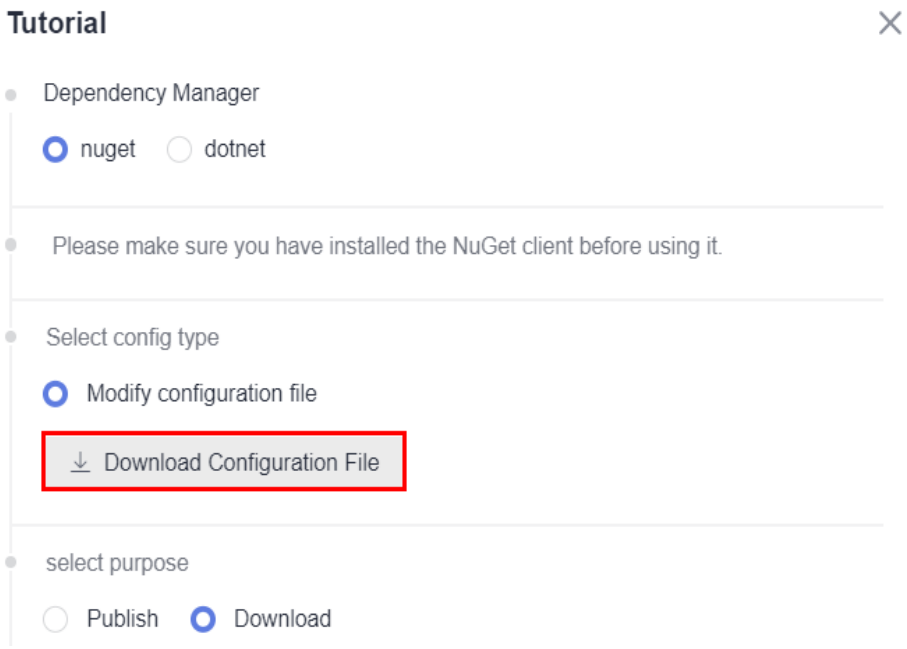
```
##----------------------dotnet upload----------------------##
dotnet nuget push <PATH_TO_FILE> -s {repo_name}
```

----End

Downloading a NuGet Package to the Client

Ensure that you have installed the NuGet.

Step 1 Select the target NuGet repository from the self-hosted repo page and click **Tutorial** to download the configuration file **NuGet.txt**.



Step 2 Open the file, find the command under **NuGet add source**, and add the source.

```
##----------------------NuGet add source----------------------##
nuget sources add -name {repo_name} -source{repo_url} -username {user_name} -password
{repo_password}
```

Step 3 Open the file, find the statement under **NuGet Download**, replace **<PACKAGE>** with the name of the package to be downloaded, and run the download statement. (If a configuration source exists, use the configured source name as the parameter following **-source**.)

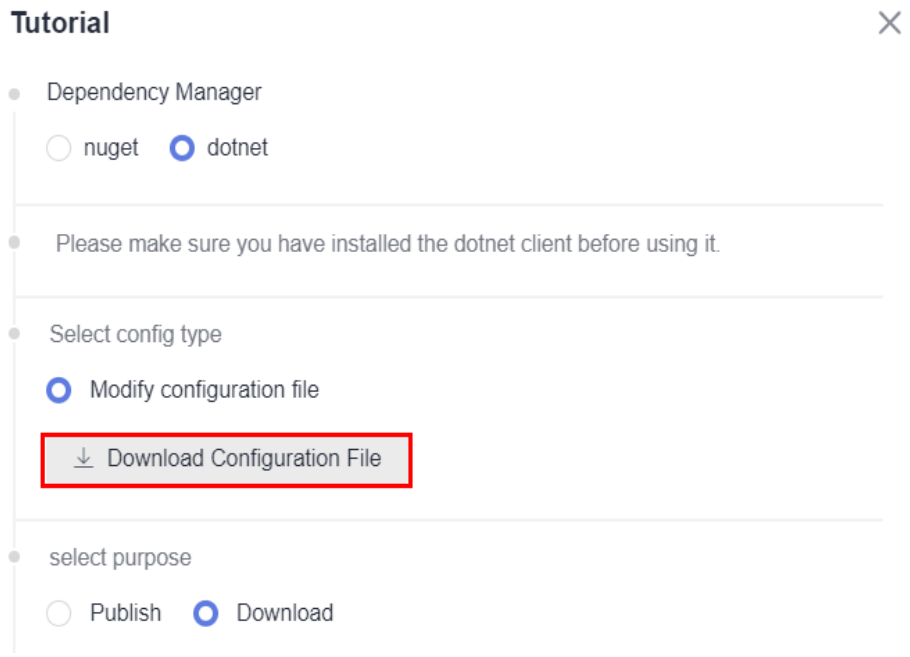
```
##----------------------NuGet Download----------------------##
nuget install <PACKAGE>
```

----End

Downloading a .NET Package to the Client

Ensure that you have installed the .NET.

Step 1 Select the target NuGet repository from the self-hosted repo page and click **Tutorial** to download the configuration file **dotnet.txt**.



Step 2 Open the configuration file, find the command under **dotnet add source**, and add the source.

```
##----------------------dotnet add source----------------------##
dotnet nuget add source {repo_url} add -n {repo_name} -u {user_name} -p {repo_password}
```

Step 3 Find the statement under **dotnet download**, replace **< PACKAGE >** with the name of the package to be downloaded, and run the download statement.

```
##----------------------dotnet download----------------------##
dotnet add package <PACKAGE>
```

----End

Uploading a Docker Image from the Client

Prerequisites

- You have installed the Docker client.
- You have created a Docker registry in the self-hosted repo.

Procedure

Step 1 Select the target Docker registry from the self-hosted repo page and click **Tutorial** on the right of the page.

Step 2 Click **Download Configuration File** to download the **config.json** configuration file.

Step 3 Obtain *{username}* and *{password}* from the downloaded file.

Step 4 Run the following command on the local client to log in to the Docker registry:

```
docker login {url} -u ${username} -p ${password}
```

url: repository address.

username: *{username}* obtained in **Step 3**.

password: *{password}* obtained in **Step 3**.

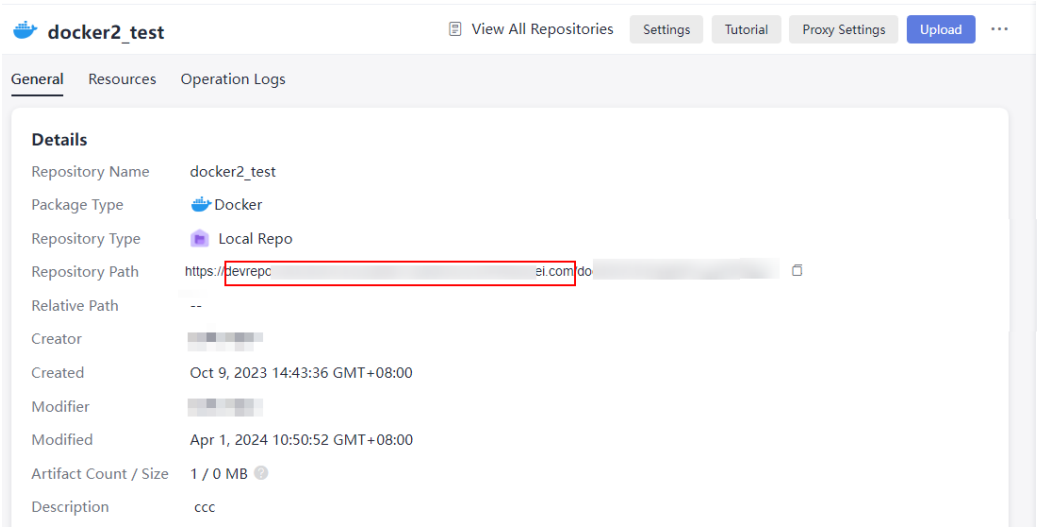
Step 5 Run the following command on the local client to package the image:

```
docker tag ${image_name1}:${image_version1} {url}/${image_name2}:${image_version2}
```

image_name1: local image name.

image_version1: local image version.

url: repository address, as shown in the following figure.



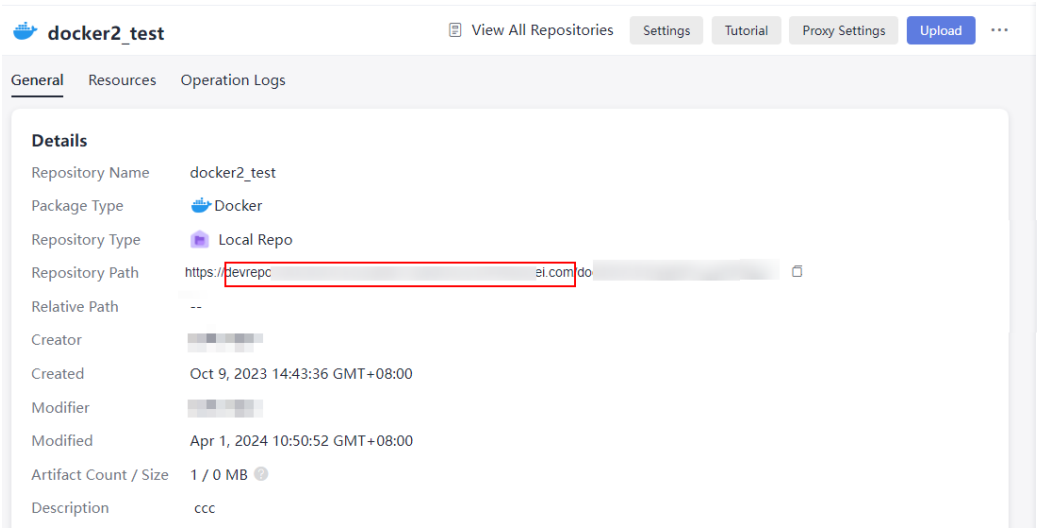
image_name2: You can name the uploaded image. The image name is displayed in the image list of the Docker registry.

image_version2: You can set the version of the uploaded image.

Step 6 Run the following command on the local client to upload the Docker image to the self-hosted repo:

```
docker push {url}/${image_name}:${image_version}
```

url: repository address, as shown in the following figure.



image_name: Enter **image_name2** in **Step 5**.

image_version: Enter **image_version2** in **Step 5**.

Step 7 Check the uploaded images in the Docker registry.

----End

Downloading a Docker Image to the Client

Prerequisites

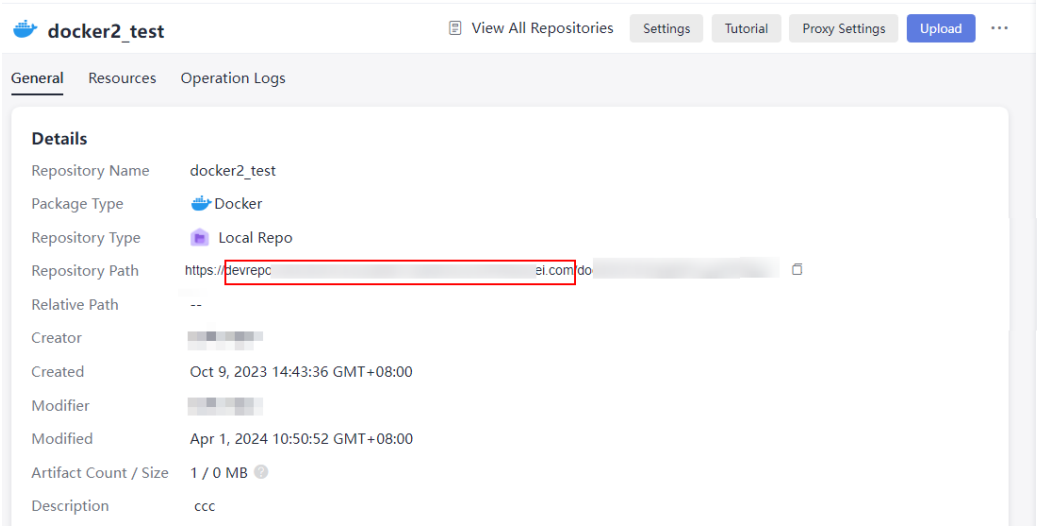
- You have installed the Docker client.
- You have created a Docker registry in the self-hosted repo.

Downloading a Docker image to the client

Run the following command on the local client to download the Docker image:

```
docker pull {url}/${image_name}:${image_version}
```

url: repository address, as shown in the following figure.



image_name: component name.

image_version: component version.

Uploading and Downloading a CocoaPods Package on the Client

Prerequisites

- You have installed the Ruby client and cocoapods-art plug-in.
- You have created a CocoaPods repository in the self-hosted repo.

Downloading configuration files and uploading CocoaPods packages to self-hosted repos

Step 1 Select the target CocoaPods repository from the self-hosted repo page and click **Tutorial** on the right of the page.

Step 2 Select **Download the provided configuration file** and click **Download Configuration File** to download the **cocoapods.txt** file.

Step 3 Obtain *{username}* and *{password}* from the downloaded file.

Step 4 Run the following command on the local client to upload the local CocoaPods package to the target self-hosted repo:

```
curl -k -u "<USERNAME>:<PASSWORD>" -XPUT "{url}/<TARGET_FILE_PATH>/" -T <PATH_TO_FILE>/<FILE_NAME>
```

USERNAME: *{username}* obtained in **Step 3**.

PASSWORD: *{password}* obtained in **Step 3**.

url: repository address of the CocoaPods repository.

TARGET_FILE_PATH: name of the self-hosted repo folder to be stored.

PATH_TO_FILE: local path of the package to be uploaded.

FILE_NAME: name of the self-hosted repo to which the package is uploaded.

Step 5 Check the uploaded package in the CocoaPods repository.

----End

Downloading the provided configuration file to download CocoaPods packages

Step 1 Select the target CocoaPods repository from the self-hosted repo page and click **Tutorial** on the right of the page.

Step 2 Select **Download the provided configuration file**.

Step 3 In the **Select purpose** area, click **Download**.

Step 4 Run the following commands to download files to the local client.

Download the package from the remote repository.

```
pod repo-art add {package_name} {url}
```

package_name: name of the CocoaPods dependency to be downloaded.

url: repository address of the self-hosted repo.

Change the path of the CocoaPods repository.

```
pod repo-art update {package_name} {url}
```

package_name: The CocoaPods dependency cannot be modified.

url: enter the repository address of the target self-hosted repo.

Query the downloaded package.

```
pod repo-art list
```

Remove the local dependencies.

```
pod repo-art remove <repo-name>//repo_name: name of the CocoaPods dependency
```

----End

Following the command line configuration to upload CocoaPods packages to self-hosted repos

Step 1 Select the target CocoaPods repository from the self-hosted repo page and click **Tutorial** on the right of the page.

Step 2 Select **Follow the command line configuration**.

Step 3 Check whether the Ruby client has been installed.

```
ruby -v
```

Step 4 Install the cocoapods-art plug-in.

```
sudo gem install cocoapods-art
```

Step 5 Add the self-hosted repo to your CocoaPods client.

```
pod repo-art add <repo_name> "{url}"
```

repo_name: name of the folder for storing packages of self-hosted repos on the local client.

url: repository address of the CocoaPods repository.

Step 6 In the **Select purpose** area, click **Publish**.

Step 7 Upload the local package to the target CocoaPods repository.

```
curl -k -u "<USERNAME>:<PASSWORD>" -XPUT "{url}/<TARGET_FILE_PATH>/" -T <PATH_TO_FILE>/<FILE_NAME>
```

USERNAME: *{username}* obtained in **Step 3**.

PASSWORD: *{password}* obtained in **Step 3**.

url: repository address of the CocoaPods repository.

TARGET_FILE_PATH: name of the self-hosted repo folder to be stored.

PATH_TO_FILE: local path of the package to be uploaded.

FILE_NAME: name of the self-hosted repo to which the package is uploaded.

Step 8 Check the uploaded package in the CocoaPods repository.

----End

Following the command line configuration to download the CocoaPods package

Step 1 Select the target CocoaPods repository from the self-hosted repo page and click **Tutorial** on the right of the page.

Step 2 Select **Follow the command line configuration**.

Step 3 Check whether the Ruby client has been installed.

```
ruby -v
```

Step 4 Install the cocoapods-art plug-in.

```
sudo gem install cocoapods-art
```

Step 5 Add the self-hosted repo to your CocoaPods client.

```
pod repo-art add <repo_name> "{url}"
```

repo_name: name of the folder for storing packages of self-hosted repos on the local client.

url: repository address of the CocoaPods repository.

Step 6 In the **Select purpose** area, click **Download**.

Step 7 Run the following commands to download files to the local client.

Download the package from the remote repository.

```
pod repo-art update {package_name}//package_name: package name.
```

Query the downloaded package.

```
pod repo-art list
```

Remove the local dependencies.

```
pod repo-art remove <repo-name>//repo-name: name of the CocoaPods dependency
```

----End

Uploading an OHPM Package from the Client

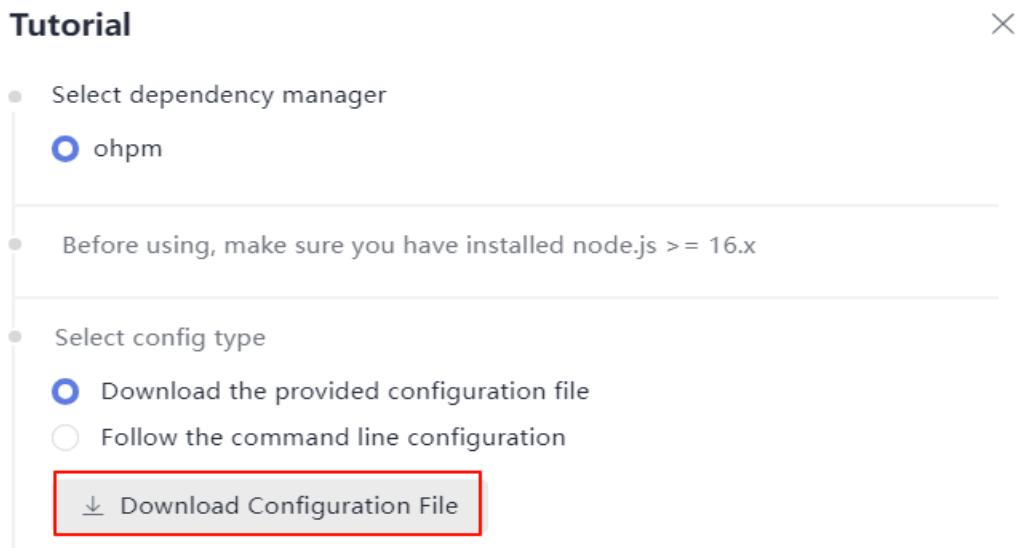
Prerequisites

- You have logged in to the CodeArts homepage.
- The client tool is npm, and the dependency management tool is OHPM. Ensure that node.js 16.x or later has been installed.

Procedure

Step 1 Select the target OHPM repository from the self-hosted repo page and click **Tutorial** on the right of the page.

Step 2 In the displayed dialog box, click **Download Configuration File** to download the **ohpmrc** file and rename it **.ohpmrc**.



Step 3 Copy the file to the user directory. The path is as follows:

Linux: **~/.ohpmrc**

Windows: **C:\Users**<*UserName*>**\.ohpm\.ohpmrc**

Step 4 Upload the OHPM package to the repository.

```
ohpm publish {file path}\{package name}
```

file path: path of the local OHPM package.

package name: name of the local OHPM package.

----End

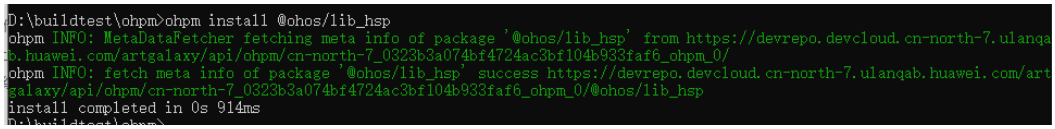
Downloading an OHPM Package to the Client

Download the client.

```
ohpm install {package name}
```

package name: **name** parameter in the return value in **Uploading an OHPM Package from the Client**, as shown in the following figure.

The following figure indicates the download success.



3.2.8 Managing Packages

Searching for a Package

Step 1 Go to the **Self-hosted Repos** page and click **Advanced Search** in the upper left corner of the page.

Step 2 In the upper part of the page, select the repository where the package to be queried is located. By default, all repositories are selected.

Step 3 Enter the keyword of the file name in the search box, and click to search for the package.

Step 4 Click the **File Name** to view its details.

----End

- Searching for artifacts by checksums

1. Click the drop-down list on the left of the search box and select **Checksums** (The default value is the file name).
2. Enter the **MD5/SHA-1/SHA-256/SHA-512 checksum** and click to find the desired package.

Downloading a Package

Step 1 Go to the self-hosted repo page. In the left pane, locate the package to be downloaded, and click its name.

If there are too many repositories or packages, you can search for the desired one by referring to **Searching for a Package**.

Step 2 Click **Download**.

----End

Deleting a Package

Step 1 Go to the self-hosted repo page. In the left pane, locate the package to be downloaded, and click its name.

If there are too many repositories or packages, you can search for the desired one by referring to **Searching for a Package**.

Step 2 Click **Delete**.

Step 3 In the displayed dialog box, click **Yes**.

----End

Favoriting or Unfavoriting

Step 1 Go to the self-hosted repo page. In the left pane, locate the package to be downloaded, and click its name.

If there are too many repositories or packages, you can search for the desired one by referring to **Searching for a Package**.

Step 2 Click **Favorite** on the right of the page.

When the icon changes to ⭐, click **My Favorites** in the lower left corner of the page to view the list of favorited items. Click a value in the **Path** column to go to the package details page.

----End

3.2.9 Managing Recycle Bin

Repositories and packages deleted from a self-hosted repo are moved to the recycle bin, where you can manage them.

The self-hosted repo provides the recycle bin entry both from the homepage and project pages.

Homepage Recycle Bin

You can manage all deleted packages in the recycle bin on the homepage.

Step 1 Access the self-hosted repo by following instructions in **Homepage Entry**.

Step 2 Click **Recycle Bin** in the upper right corner of the page. The **Recycle Bin** page is displayed on the right.

Step 3 Delete or restore the repositories and packages in the list as required.

If the icons and are both displayed in the **Operation** column, the repository shown in the corresponding row has been deleted. Otherwise, the repository shown in the corresponding row has not been deleted but the packages in it have been deleted. You can click the repository name to view the deleted packages in the repository.

Available operations are as follows.

Operation Type	Operation	Description
Restore	Restore a repository	Click in the **Operation** column to restore the repository.
	Restore a package	Go to the repository where the package to restore is located, and click in the **Operation** column to restore the package.
	Batch restore packages	Go to the repository where the packages to restore are located, select them, and click **Restore** below the list to restore them in batches.
	Restore all	Click **Restore All** in the upper right corner of the page to restore all selected items in the recycle bin.
Delete	Delete a repository	Click in the **Operation** column to delete a repository.

Operation Type	Operation	Description
	Delete a package	Go to the repository where the package to delete is located, and click in the **Operation** column to delete the package.
	Batch delete packages	Go to the repository where the packages to delete are located, select them, and click **Clear** below the list to permanently delete them in batches.
	Clear recycle bin	Click **Clear All** in the upper right corner of the page to delete all selected items in the recycle bin.

----End

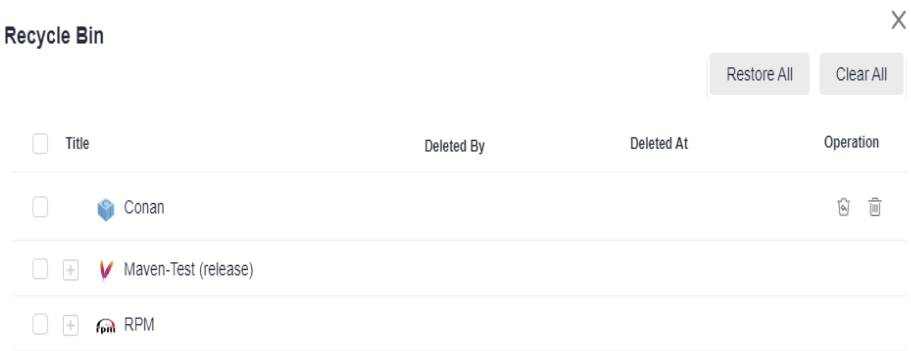
Project Recycle Bin

Step 1 Access the self-hosted repo by following instructions in **Project Entry**.

Step 2 In the lower left corner of the page, click **Recycle Bin**. The **Recycle Bin** page is displayed on the right.

Step 3 Delete or restore the repositories and packages in the list as required.

If the icons and are both displayed in the **Operation** column, the repository shown in the corresponding row has been deleted. Otherwise, the repository shown in the corresponding row has not been deleted but the packages in it have been deleted. You can click the repository name to view the deleted packages in the repository.



Available operations are as follows.

Operation Type	Operation	Description
Restore	Restore a repository	Click  in the **Operation** column to restore the repository.
	Restore a package	Go to the repository where the package to restore is located, and click  in the **Operation** column to restore the package.
	Batch restore packages	Go to the repository where the packages to restore are located, select them, and click **Restore** below the list to restore them in batches.
	Restore all	Click **Restore All** in the upper right corner of the page to restore all selected items in the recycle bin.
Delete	Delete a repository	Click  in the **Operation** column to delete a repository.
	Delete a package	Go to the repository where the package to delete is located, and click  in the **Operation** column to delete the package.
	Batch delete packages	Go to the repository where the packages to delete are located, select them, and click **Clear** below the list to permanently delete them in batches.
	Clear recycle bin	Click **Clear All** in the upper right corner of the page to delete all selected items in the recycle bin.

----End

4 FAQ

4.1 Release Repo

4.1.1 Why Can't I Upload Files or Create Directories on the Release Repo Homepage Under CodeArts Artifact?

On the release repo homepage, the top-level directory names map to your project names. They are used to distinguish the project to which a software package belongs.

You can only browse files and directories in this directory and cannot upload files or create directories.

However, you can click a project name to access the project folder and upload files and create directories there.

4.1.2 Can I Change the Dependency ID in pom.xml to Invoke a Specified JAR Package in Release Repo?

No.

Packages in Release Repo are used for deployment, not as dependencies during build.

You can upload dependency packages to be referenced to Self-Hosted Repo.

4.2 Self-Hosted Repo

4.2.1 How Do I Upload Snapshots to a Maven Repository?

Background

Snapshots can be uploaded in the following modes:

- Uploading Snapshots on the Self-Hosted Repos Page
- Uploading Snapshots Using the Maven CLI
- Releasing Snapshots to a Maven Repository Through CodeArts Build

Uploading Snapshots on the Self-Hosted Repos Page

Step 1 Log in to CodeArts.

Step 2 Choose **Services** > **Artifact** from the top navigation bar, click the **Self-hosted Repos** tab, and find the corresponding repository.

Step 3 Click the snapshot repository in the repository list on the left. Click **Upload**. In the **Upload** dialog box displayed, select **GAV** as required.

There are two GAV definition modes.

GAV Definition Mode	Description
POM	GAV information is extracted from POM files.
GAV	GAV information is manually specified.

Step 4 Set related parameters as prompted and upload the corresponding package.

----End

Uploading Snapshots Using the Maven CLI

Step 1 Select Maven as the package type, and choose the **Snapshot** repository in the repository list.

Step 2 Click **Tutorial** in the upper right corner. The **Configuration File** dialog box is displayed.

Step 3 Configure the local Maven tool by following the configuration guide.

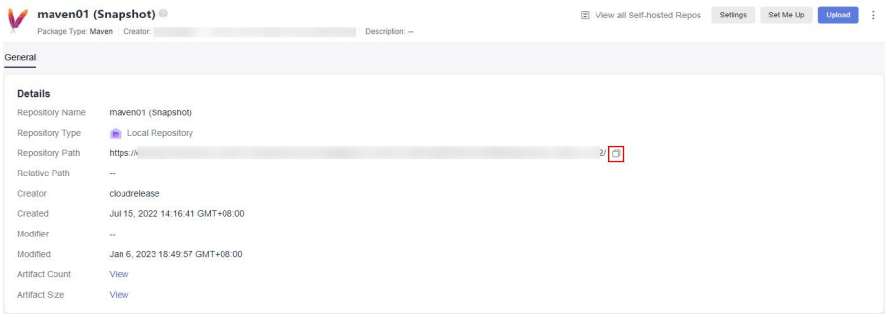
Step 4 Run **mvn deploy** to upload the Maven project.

In the Maven CLI, access the directory where the **pom.xml** file of the Maven project is stored, then run the following command to upload a local JAR package:

```
mvn deploy:deploy-file -DgroupId=com. -DartifactId=aopalliance -Dversion=1.0-SNAPSHOT -Dpackaging=jar -Dfile=D:\aopalliance-1.0-SNAPSHOT.jar -Durl={Maven Snapshot address} -DrepositoryId=snapshots
```

NOTE

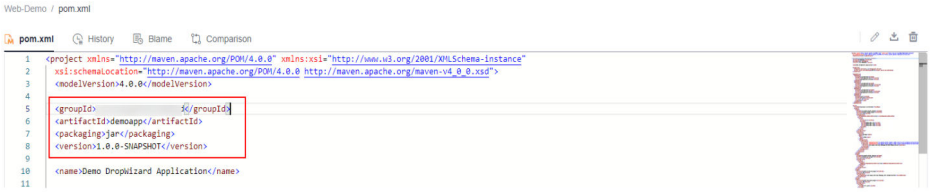
- Set **DgroupId**, **DartifactId**, **Dversion**, and **Dpackaging** as required.
- Set **Dfile** to the absolute path of the local JAR package.
- Set **Durl** to the Maven snapshot address, which can be obtained by clicking in the following figure.



----End

Releasing Snapshots to a Maven Repository Through CodeArts Build

Step 1 Access the code repository, open the **pom.xml** file, and define the GAV information of the component to upload.



NOTE

- When a build task is run, CodeArts Build identifies the component properties uploaded to the Maven repository based on the definition.
- version: Releases are uploaded by default. To upload a snapshot, add the suffix **-SNAPSHOT** to the value of **version**, for example, **1.0-SNAPSHOT**.

Step 2 Edit a build task. Specifically, in the build action **Build with Maven**:

- In the command box, comment out the **mvn package** command (add **#** before the command) and uncomment the **mvn deploy** command (delete **#** before the command).
- Click **Release to Self-hosted Repos**, and select **Configure all POMs**.

Step 3 Run the build task. After the build task is executed, you can find the generated Maven component in the Maven repository.

----End

4.2.2 How Do I Call a Private Component from a Self-hosted Maven Repo?

Step 1 Access the self-hosted repo, and click the name of a private component to open the file attribute page.

Step 2 Obtain the dependency download address, copy it, and add it to the **pom.xml** file.

----End

4.2.3 Can I Call Software Packages from Self-Hosted Repos During Local Builds?

Yes.

Go to the repository where the packages are stored and click **Tutorial** in the upper right corner. Download the configuration file and modify it by following the guide.

4.2.4 Why Can't the Repository Receive Requests?

Symptoms

Local build task fails, **Connection reset** is displayed, and the log information similar to the following is displayed.

```
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-deploy-plugin:2.7:deploy (default-deploy) on project
base-parent: Failed to retrieve remote metadata [redacted].framework:base-parent:4.1.200-SNAPSHOT/maven-
metadata.xml: Could not transfer metadata [redacted].framework:base-parent:4.1.200-SNAPSHOT/maven-
metadata.xml from/to snapshots [redacted]
[redacted]/framework/bas
e-parent/4.1.200-SNAPSHOT/maven-metadata.xml: Connection reset -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
```

Cause Analysis

The Java version is too early and does not support TLS 1.2.

Solution

- If Java 6 is used, upgrade it to Java 8 or later.
- If Java 7 is used, TLS 1.2 is supported. However, TLS 1.2 is not supported in versions earlier than 1.7.0_131-b31. You can run the following command to enable TLS 1.2:

  ```
  mvn -Dhttps.protocols= TLSv1.2 <goals>
  ```

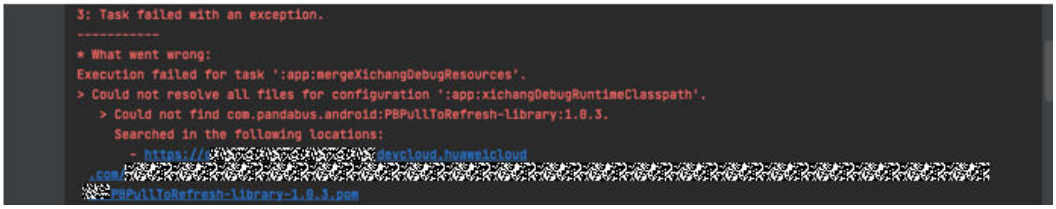
  You can also add the following command to your environment or build script.

  ```
  export MAVEN_OPTS=-Dhttps.protocols= TLSv1.2
  ```

4.2.5 Why Did the Dependency WAR or JAR Files Fail to Be Downloaded?

Symptom

The local tools cannot download components in the self-hosted repo. A message is displayed indicating that the POM file cannot be found. The log information similar to the following is recorded.



Root Cause

The POM file is missing in the dependency.

When downloading dependency packages using Gradle or Maven, you need to download a POM file first, and then a JAR or WAR file. Otherwise, the download will fail.

Solution

Re-upload the components that cannot be downloaded according to the components uploading standard.

4.2.6 Why Is Error 401 Returned When Uploading Maven Artifacts to Self-Hosted Repos?

Symptom

Failed to upload Maven artifacts to self-hosted repos from the local IDE, and **401-Insufficient Permission** is displayed.

Cause Analysis

The self-hosted repo information configured in the code repository file **pom.xml** does not match the **settings.xml** file.

Solution

When uploading artifacts, replace the **repository_id** value in the **distributionManagement** element of the **pom.xml** file with the **repository_id** value in the **server** element of the **settings.xml** file.

The uploading process is as follows:

Step 1 Access the self-hosted repo homepage, and choose Maven from the repository list.

Step 2 Click **Tutorial** in the upper right corner. The **Configuration Guide** dialog box is displayed.

Step 3 Configure the local Maven tool by following the configuration guide.

Step 4 Run **mvn deploy** to upload the Maven project.

- In the Maven CLI, access the directory where the **pom.xml** file of the Maven project is stored, check whether the **repository_id** value in the **distributionManagement** element of the **pom.xml** file matches the **repository_id** value in the **server** element of the **settings.xml** file.

  setting.xml

  ```
  <server>
      <id>release_cn-north-[redacted]</id>
      <username>cn-north-[redacted]</username>
      <password>[redacted]</password>
  </server>
  <server>
      <id>snapshot_cn-north-[redacted]</id>
      <username>cn-north-[redacted]</username>
      <password>[redacted]</password>
  </server>
  <server>
      <id>z_mirrors</id>
  </server>
  ```

  pom.xml

  ```
  <dependencies>
    <dependency>
      <groupId>a</groupId>
      <artifactId>a</artifactId>
      <version>a</version>
    </dependency>
  </dependencies>
  <distributionManagement>
      <repository>
          <id>release_cn-north-[redacted]</id>
          <url>[redacted]</url>
      </repository>
      <snapshotRepository>
          <id>[redacted]</id>
          <url>[redacted]</url>
      </snapshotRepository>
  </distributionManagement>
  </project>
  ```

- Upload the local JAR package:

  ```
  mvn deploy:deploy-file -DgroupId=com. -DartifactId=aopalliance -Dversion=1.0 -Dpackagi=jar
  ```

----End