CodeArts Artifact

Best Practices

Issue 01

Date 2025-02-07

HUAWEI TECHNOLOGIES CO., LTD.

Copyright © Huawei Technologies Co., Ltd. 2025. All rights reserved.

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Technologies Co., Ltd.

Trademarks and Permissions

HUAWEI and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Security Declaration

Vulnerability

Huawei's regulations on product vulnerability management are subject to the *Vul. Response Process.* For details about this process, visit the following web page:
https://www.huawei.com/en/psirt/vul-response-process
For vulnerability information, enterprise customers can visit the following web page:
https://securitybulletin.huawei.com/enterprise/en/security-advisory

Contents

1 CodeArts Artifact Best Practices

This document summarizes the operation practices of CodeArts Artifact in common application scenarios. It provides detailed solutions for each practice, helping users easily use CodeArts Artifact in different application scenarios.

Table 1-1 CodeArts Artifact best practices

Best Practice	Description
Releasing Maven Components and Archiving to a Self-Hosted Repo	CodeArts Artifact focuses on and manages the staging software packages (usually built by 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. Throughout the development process, the software package is continuously refined across different versions. The management of software packages and their properties is the basis of release management. Therefore, developers need to regularly review the version history of software packages. This practice describes how to archive a Maven component by version to a self-hosted repo via a build task.
Releasing/Obtaining an npm Component via a Build Task	A self-hosted repo manages private component packages (such as Maven) corresponding to various development languages. Different development language components vary in the archive format. A self-hosted repo manages private development language components and shares them with other developers in the same enterprise or team. This practice describes how to release a component to an npm repository via a build task and obtain a dependency from the repository for deployment.
Releasing/Obtaining a Go Component via a Build Task	This practice describes how to release a component to a Go repository via a build task and obtain a dependency from the repository for deployment.

Best Practice	Description
Releasing/Obtaining a PyPI Component via a Build Task	This practice describes how to release a component to a PyPI repository via a build task and obtain a dependency from the repository for deployment.
Uploading/Obtaining an RPM Component Using Linux Commands	This practice describes how to use Linux commands to upload a component to an RPM repository and obtain a dependency from the repository.
Uploading/Obtaining a Debian Component Using Linux Commands	This practice describes how to use Linux commands to upload a component to a Debian repository and obtain a dependency from the repository.
Batch Migrating Maven/npm/PyPI Components to a Self-Hosted Repo	Self-hosted repos allow you to manually upload and download components. They can also interconnect with your local development environment to upload and download components. Uploading numerous packages one at a time can be cumbersome. You can use the migration tool provided by self-hosted repos to upload components in batches from Nexus or other repositories.

2 Releasing Maven Components and Archiving to a Self-Hosted Repo

Background

CodeArts Artifact focuses on and manages the staging software packages (usually built by 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. Throughout the development process, the software package is continuously refined across different versions.

The management of software packages and their properties is the basis of release management. Therefore, developers need to regularly review the version history of software packages.

Preparations

- You already have a project. If no project is available, **create one**.
- You have permissions for the current repository. For details, see **Configuring Repository Permissions 2.0**.

Creating a Maven Repository and Associating It with a Project

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 Click + Create , select **Local Repository** as the repository type, enter the repository name, and select **Maven** as the package type.

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

Step 5 In the **Repository View**, click the name of the target repository and click **Settings**.

Step 6 Click the **Project Associations** tab, click in the **Operation** column of the target project, and select target self-hosted repos in the displayed dialog box.

Step 7 Click **OK**.

----End

Configuring Component Versions in Repo

Step 1 Log in to CodeArts and go to a created project.

Step 2 Choose **Services** > **Repo** on the top navigation bar.

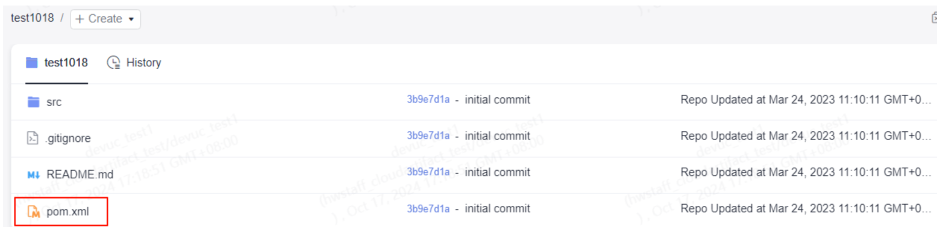
Step 3 Click **New Repository**.

Step 4 Select a project from the **Project** drop-down list, select **Template**, and click **Next**.

Step 5 Search for the **Java Maven Demo** template, and click **Next**.

Step 6 Enter the repository name and click **OK**.

Step 7 Go back to Repo and click **pom.xml** to view the component configuration.



test1018 / + Create

test1018 History

src	3b9e7d1a - initial commit	Repo Updated at Mar 24, 2023 11:10:11 GMT+0...
.gitignore	3b9e7d1a - initial commit	Repo Updated at Mar 24, 2023 11:10:11 GMT+0...
README.md	3b9e7d1a - initial commit	Repo Updated at Mar 24, 2023 11:10:11 GMT+0...
pom.xml	3b9e7d1a - initial commit	Repo Updated at Mar 24, 2023 11:10:11 GMT+0...

Step 8 On the component configuration page, the **<version>** field displays the version number of the current component. The default version number is **1.0**.

Click in the upper right corner of the page to change the version number. Then, click **OK** to save the changes.

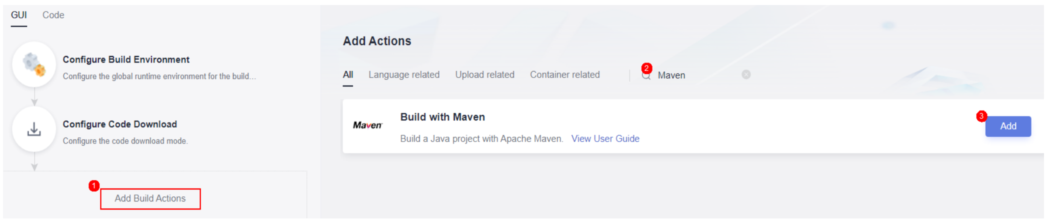
----End

Releasing Maven Components to a Self-Hosted Repo Through CodeArts Build

Step 1 After configuring the component version in Repo, click **Create Build Task** in the upper right corner of the page.

Step 2 Select **Blank Template** and click **OK**.

Step 3 Click **Add Build Actions**. Search for and add the **Build with Maven** action.


GUI
Code
Configure Build Environment
Configure the global runtime environment for the build...
Configure Code Download
Configure the code download mode.
1
Add Build Actions
Add Actions
All
Language related
Upload related
Container related
2
Maven
Build with Maven
Build a Java project with Apache Maven. View User Guide
3
Add

Step 4 Edit the **Build with Maven** action.

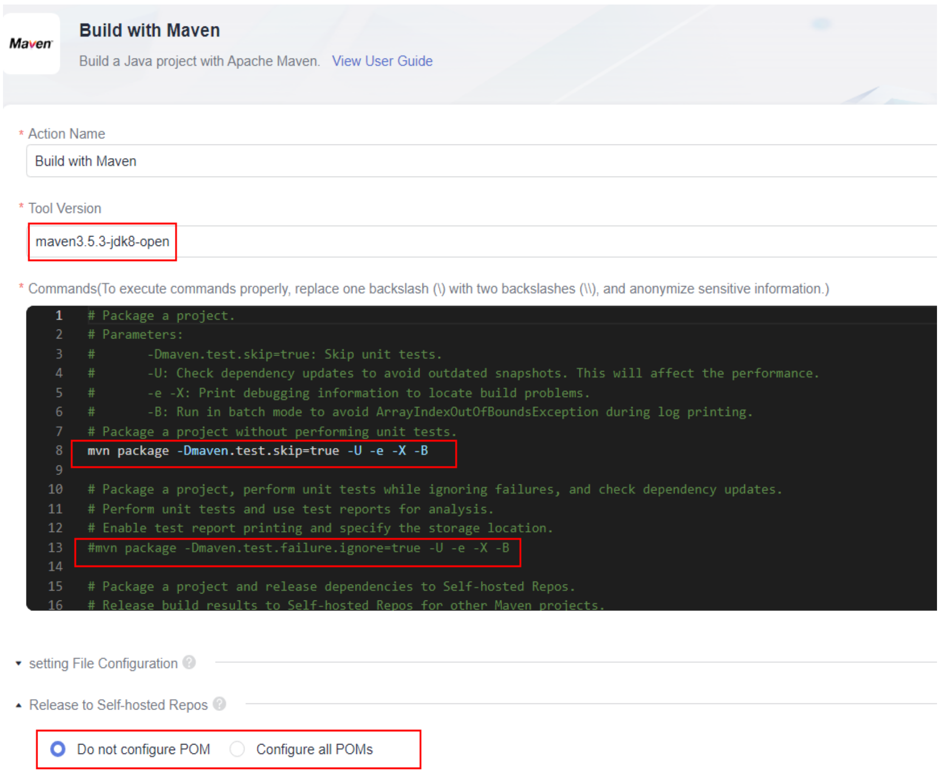
- Select the desired tool version. In this example, **maven3.5.3-jdk8-open** is used.
- Find the following command and delete **#** in front of this command:

```
#mvn deploy -Dmaven.test.skip=true -U -e -X -B
```

  Find the following command and add **#** in front of this command:

```
mvn package -Dmaven.test.skip=true -U -e -X -B
```

- Select **Configure all POMs** under **Release to Self-hosted Repos**, and select the Maven repository associated with the project.



Build with Maven

Build a Java project with Apache Maven. View User Guide

* Action Name: Build with Maven

* Tool Version: maven3.5.3-jdk8-open

* Commands(To execute commands properly, replace one backslash (\) with two backslashes (\\), and anonymize sensitive information.)

```
# Package a project.
# Parameters:
#       -Dmaven.test.skip=true: Skip unit tests.
#       -U: Check dependency updates to avoid outdated snapshots. This will affect the performance.
#       -e -X: Print debugging information to locate build problems.
#       -B: Run in batch mode to avoid ArrayIndexOutOfBoundsException during log printing.
# Package a project without performing unit tests.
mvn package -Dmaven.test.skip=true -U -e -X -B

# Package a project, perform unit tests while ignoring failures, and check dependency updates.
# Perform unit tests and use test reports for analysis.
# Enable test report printing and specify the storage location.
#mvn package -Dmaven.test.failure.ignore=true -U -e -X -B

# Package a project and release dependencies to Self-hosted Repos.
# Release build results to Self-hosted Repos for other Maven projects.
```

setting File Configuration

Release to Self-hosted Repos

[✓ Do not configure POM] Configure all POMs

Step 5 Click **Save and Run** on the right of the page to start the build task.

----End

Viewing Archived Components in the Version View of the Maven Repository

Step 1 Go to the self-hosted repo page, locate the target maven repository, and find the Maven component uploaded by build task.

Repeat the preceding steps to **set the component version** in Repo and archive components of multiple versions to self-hosted repos.

Step 2 Click the **Version View** tab.

In the package list, view the number of versions and the latest version of the package obtained from build tasks.

Step 3 Click a name in the **Package Name** column. The **Overview** page for the latest version of the package is displayed.

Step 4 Click the **Files** tab, click ⤓ in the **Operation** column of the target component to download it to your local host.

Step 5 After modifying a component and setting a new version number, click **Upload** in the right of the target self-hosted repo to upload the latest version of the component.

The package list in the version view displays the latest uploaded version of each component and the number of archived versions.

----End

Copyright © Huawei Technologies Co., Ltd.

3 Releasing/Obtaining an npm Component via a Build Task

This section describes how to release a component to an npm repository via a build task and obtain a dependency from the repository for deployment.

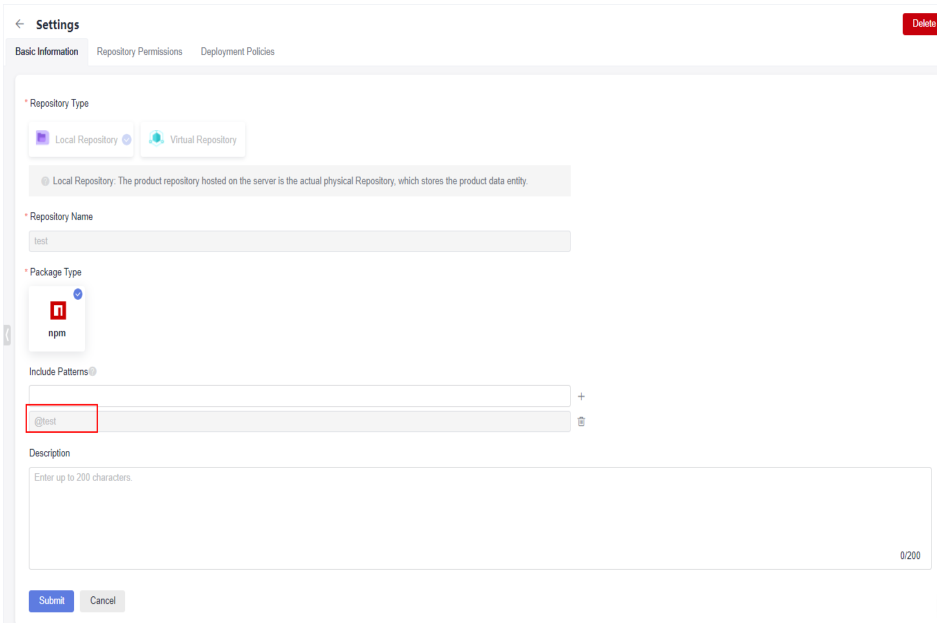
Prerequisites

- You already have a project. If no project is available, **create one**.
- You have created an npm repository.
- You have permissions for the current repository. For details, see **Configuring Repository Permissions 2.0**

Releasing a Component to an npm Repository

Step 1 Download the configuration file.

1. Log in to CodeArts Artifact and access the npm repository. Click **Settings** in the upper right corner and record the repository path.



Settings

Delete

Basic Information | Repository Permissions | Deployment Policies

* Repository Type

Local Repository | Virtual Repository

Local Repository: The product repository hosted on the server is the actual physical Repository, which stores the product data entity.

* Repository Name

test

* Package Type

npm

Include Patterns

@test

Description

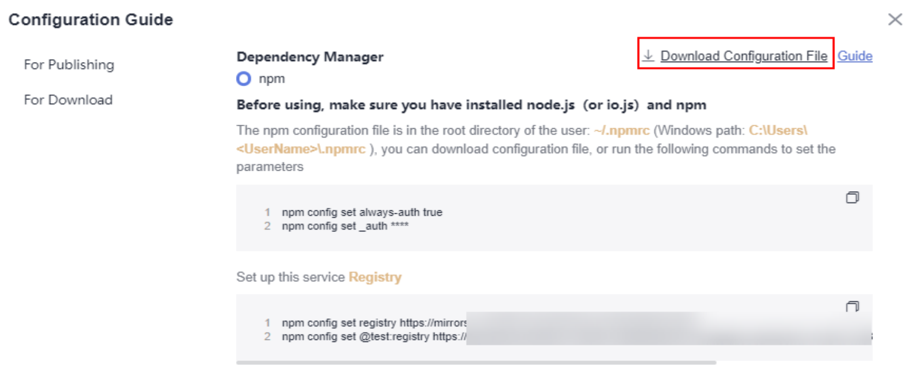
Enter up to 200 characters.

0/200

Submit | Cancel

Copyright © Huawei Technologies Co., Ltd.

2. Click **Cancel** to return to the npm repository page. Click **Tutorial** on the right of the page.
3. In the displayed dialog box, click **Download Configuration File**.



Dependency Manager

npm

Before using, make sure you have installed node.js (or io.js) and npm

The npm configuration file is in the root directory of the user: ~/.npmrc (Windows path: C:\Users\<UserName>\.npmrc), you can download configuration file, or run the following commands to set the parameters

```
npm config set always-auth true
npm config set _auth ****
```

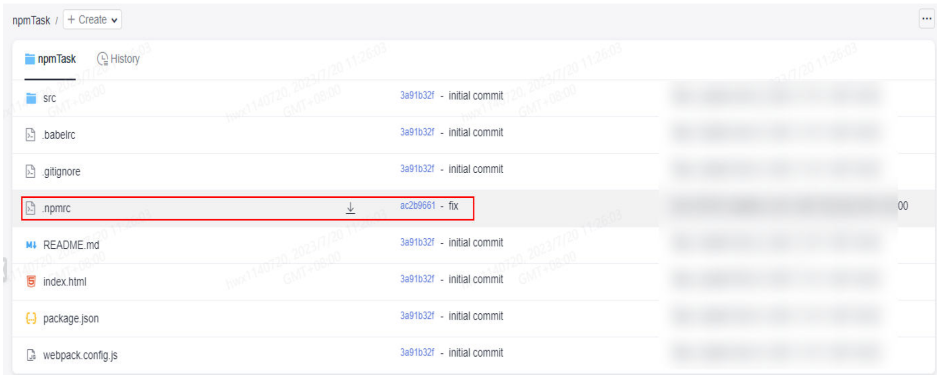
Set up this service Registry

```
npm config set registry https://mirrors
npm config set @test:registry https://
```

4. Save the downloaded **npmrc** file as an **.npmrc file**.

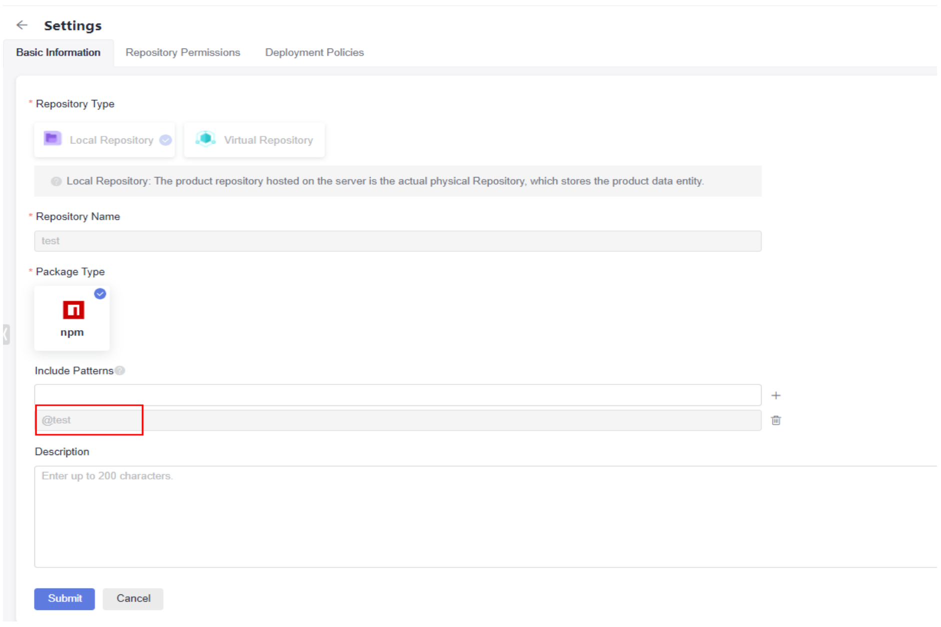
Step 2 Configure a repository.

1. Go to Repo and create a **Node.js** repository. For details, see **Creating a Repository**. This procedure uses the **Nodejs Webpack Demo** template.
2. Go to the repository and upload the **.npmrc** file to the root directory of the repository. For details, see **Uploading Code Files to CodeArts Repo**.

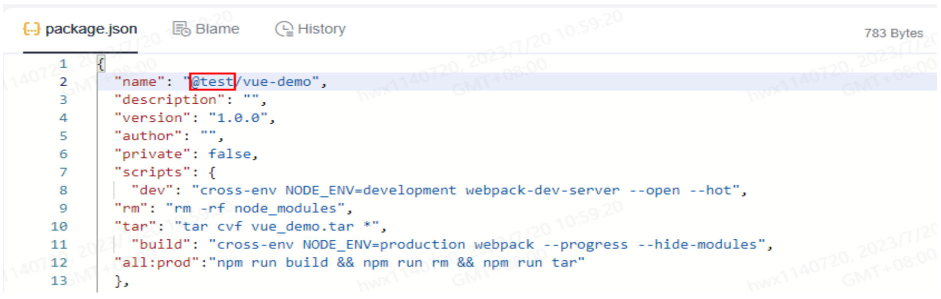


src	3a91b32f - initial commit
.babelrc	3a91b32f - initial commit
.gitignore	3a91b32f - initial commit
.npmrc	ac2b9661 - fix
README.md	3a91b32f - initial commit
index.html	3a91b32f - initial commit
package.json	3a91b32f - initial commit
webpack.config.js	3a91b32f - initial commit

3. Find the **package.json** file in the repository and open it. Add the path recorded on the **Basic Information** under the **Settings** tab page to the **name** field in the file.

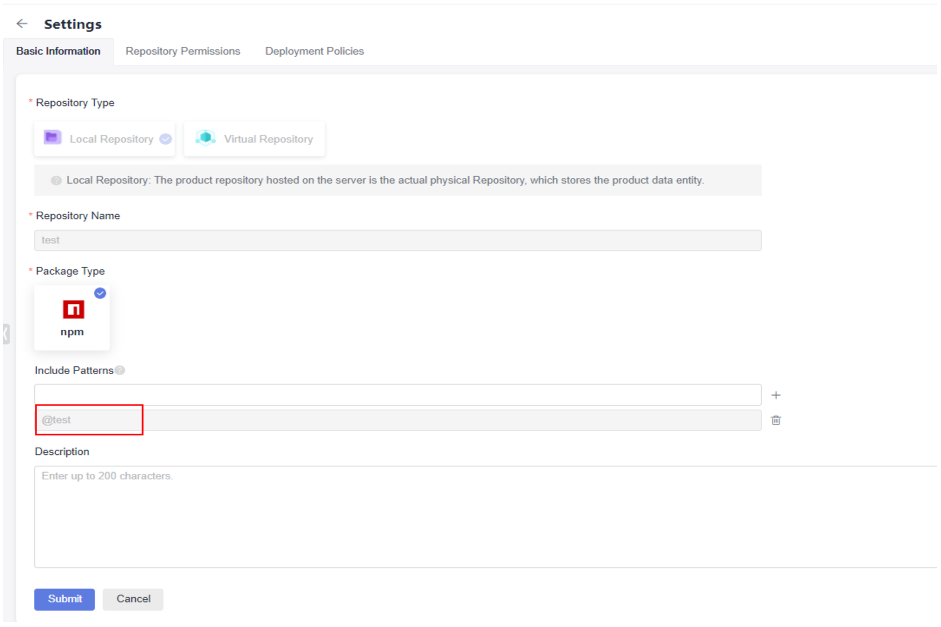


Local Repository: The product repository hosted on the server is the actual physical Repository, which stores the product data entity.



```
{
  "name": "@test/vue-demo",
  "description": "",
  "version": "1.0.0",
  "author": "",
  "private": false,
  "scripts": {
    "dev": "cross-env NODE_ENV=development webpack-dev-server --open --hot",
  "rm": "rm -rf node_modules",
  "tar": "tar cvf vue_demo.tar *",
    "build": "cross-env NODE_ENV=production webpack --progress --hide-modules",
  "all:prod":"npm run build && npm run rm && npm run tar"
  },
```

If the **name** field cannot be modified, add the path to the **Include Patterns** field on the **Basic Information** under the **Settings** tab page.



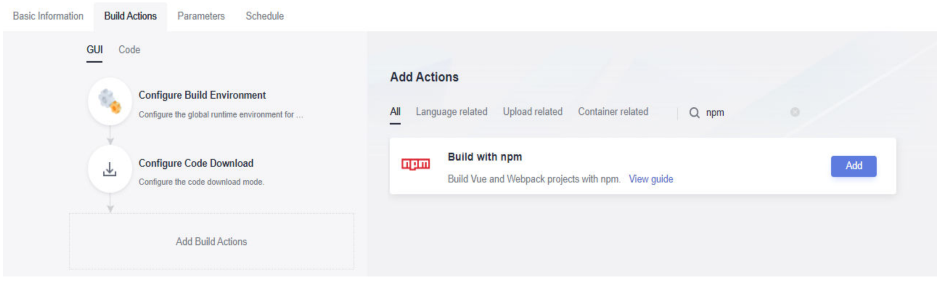
Local Repository: The product repository hosted on the server is the actual physical Repository, which stores the product data entity.

Step 3 Configure and run a build task.

1. On the Repo page, select a repository and click **Create Build Task** in the upper right.

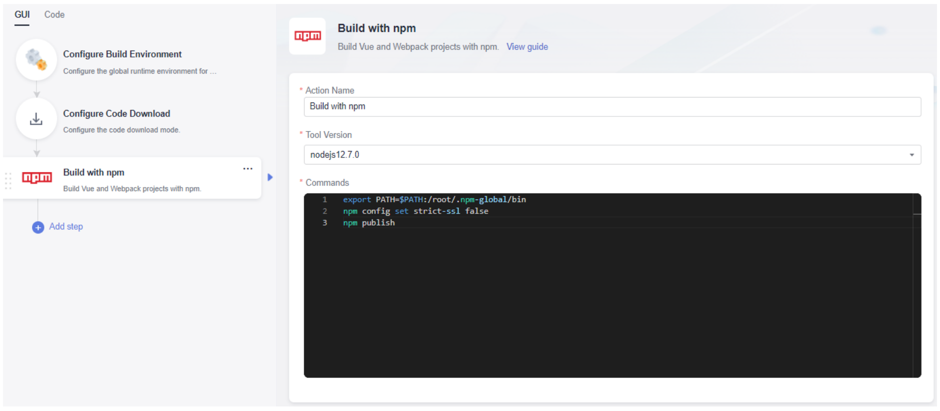
Select **Blank Template** and click **OK**.

2. Add the **Build with npm** action.



3. Edit the **Build with npm** action.
   – Select the desired tool version. In this example, **nodejs12.7.0** is used.
   – Delete the existing commands and run the following instead:

```
export PATH=$PATH:/root/.npm-global/bin
npm config set strict-ssl false
npm publish
```


GUI
Code
Configure Build Environment
Configure the global runtime environment for ...
Configure Code Download
Configure the code download mode.
Build with npm
Build Vue and Webpack projects with npm.
Add step
Build with npm
Build Vue and Webpack projects with npm.
View guide
Commands

* Action Name: Build with npm

* Tool Version: nodejs12.7.0

```
export PATH=$PATH:/root/.npm-global/bin
npm config set strict-ssl false
npm publish
```

4. Click **Save and Run** on the right of the page to start the build task.

   After the task is successfully executed, go to the self-hosted repo page and find the uploaded npm component.

----End

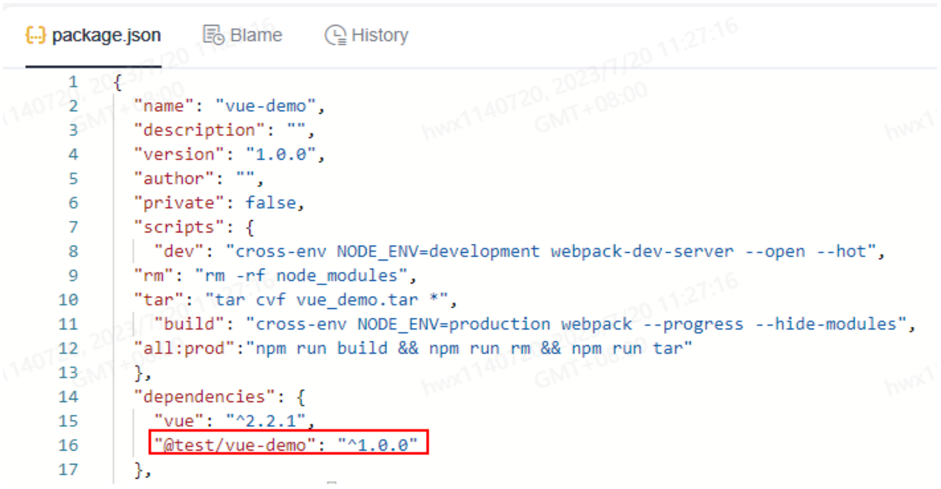
Obtaining a Dependency from an npm Repository

The following procedure uses the npm component released in **Releasing a Component to an npm Repository** as an example to describe how to obtain a dependency from an npm repository.

Step 1 Configure a repository.

1. Go to Repo and create a **Node.js** repository. For details, see **Creating a Repository**. This procedure uses the **Nodejs Webpack Demo** template.
2. Obtain the **.npmrc** file (see **Releasing a Component to an npm Repository**) and upload it to the root directory of the repository where the npm dependency is to be used.
3. Find and open the **package.json** file in the repository, and configure the dependency to the **dependencies** field. In this document, the value is as follows:

```
"@test/vue-demo": "^1.0.0"
```


package.json Blame History

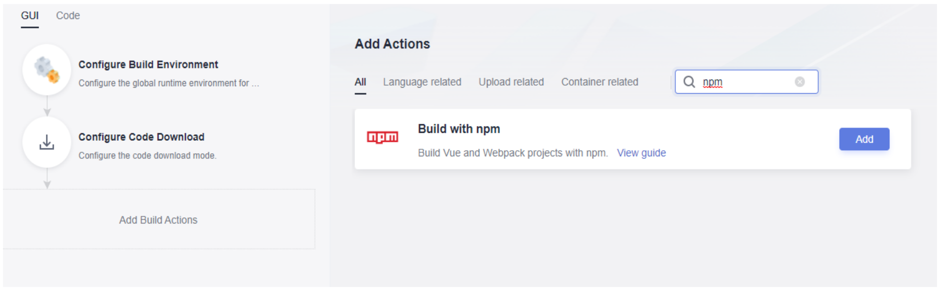
```
{
  "name": "vue-demo",
  "description": "",
  "version": "1.0.0",
  "author": "",
  "private": false,
  "scripts": {
    "dev": "cross-env NODE_ENV=development webpack-dev-server --open --hot",
  "rm": "rm -rf node_modules",
  "tar": "tar cvf vue_demo.tar *",
    "build": "cross-env NODE_ENV=production webpack --progress --hide-modules",
  "all:prod":"npm run build && npm run rm && npm run tar"
  },
  "dependencies": {
    "vue": "^2.2.1",
    "@test/vue-demo": "^1.0.0"
  },
```

Step 2 Configure and run a build task.

1. On the Repo page, select a repository and click **Create Build Task** in the upper right.

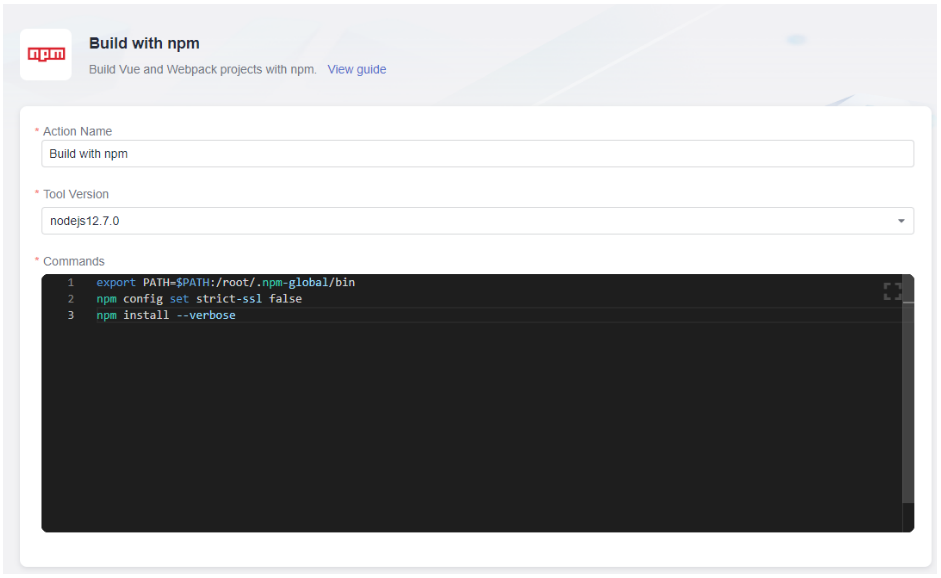
   Select **Blank Template** and click **OK**.

2. Add the **Build with npm** action.


GUI
Code
Configure Build Environment
Configure the global runtime environment for ...
Configure Code Download
Configure the code download mode.
Add Build Actions
Add Actions
All
Language related
Upload related
Container related
npm
Build with npm
Build Vue and Webpack projects with npm.
View guide
Add

3. Edit the **Build with npm** action.
   – Select the desired tool version. In this example, **nodejs12.7.0** is used.
   – Delete the existing commands and run the following instead:

```
export PATH=$PATH:/root/.npm-global/bin
npm config set strict-ssl false
npm install --verbose
```


Build with npm

Build Vue and Webpack projects with npm. View guide

* Action Name: Build with npm

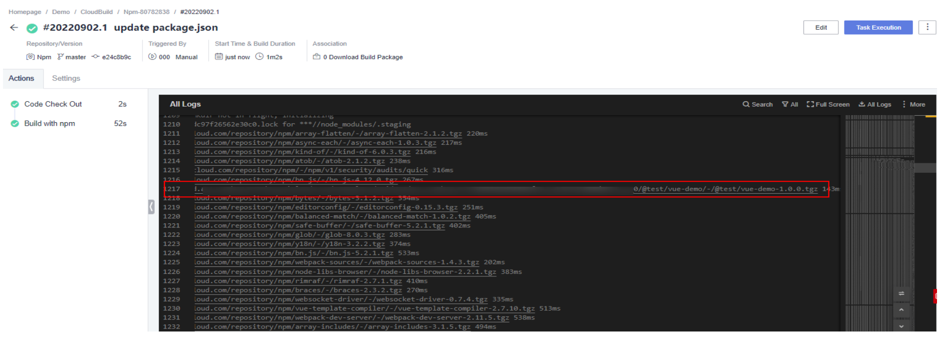
* Tool Version: nodejs12.7.0

* Commands

```
export PATH=$PATH:/root/.npm-global/bin
npm config set strict-ssl false
npm install --verbose
```

Step 3 Click **Save and Run** on the right of the page to start the build task.

After the task is successfully executed, view the task details. If information similar to the following is found in the log, the dependency has been downloaded from the npm repository.



```
:c97f26562e30c0.lock for ***//node_modules/.staging
loud.com/repository/npm/array-flatten/-/array-flatten-2.1.2.tgz 220ms
loud.com/repository/npm/async-each/-/async-each-1.0.3.tgz 217ms
loud.com/repository/npm/kind-of/-/kind-of-6.0.3.tgz 216ms
loud.com/repository/npm/atob/-/atob-2.1.2.tgz 238ms
:loud.com/repository/npm/-/npm/v1/security/audits/quick 316ms
loud.com/repository/npm/bn.js/-/bn.js-4.12.0.tgz 267ms
d.a[redacted]0/@test/vue-demo/-/@test/vue-demo-1.0.0.tgz 143ms
loud.com/repository/npm/bytes/-/bytes-3.1.2.tgz 354ms
loud.com/repository/npm/editorconfig/-/editorconfig-0.15.3.tgz 251ms
loud.com/repository/npm/balanced-match/-/balanced-match-1.0.2.tgz 405ms
loud.com/repository/npm/safe-buffer/-/safe-buffer-5.2.1.tgz 402ms
loud.com/repository/npm/glob/-/glob-8.0.3.tgz 283ms
loud.com/repository/npm/y18n/-/y18n-3.2.2.tgz 374ms
loud.com/repository/npm/bn.js/-/bn.js-5.2.1.tgz 533ms
loud.com/repository/npm/webpack-sources/-/webpack-sources-1.4.3.tgz 202ms
loud.com/repository/npm/node-libs-browser/-/node-libs-browser-2.2.1.tgz 383ms
loud.com/repository/npm/rimraf/-/rimraf-2.7.1.tgz 410ms
loud.com/repository/npm/braces/-/braces-2.3.2.tgz 270ms
loud.com/repository/npm/websocket-driver/-/websocket-driver-0.7.4.tgz 335ms
loud.com/repository/npm/vue-template-compiler/-/vue-template-compiler-2.7.10.tgz 513ms
loud.com/repository/npm/webpack-dev-server/-/webpack-dev-server-2.11.5.tgz 538ms
loud.com/repository/npm/array-includes/-/array-includes-3.1.5.tgz 494ms
```

----End

npm Commands

When configuring build tasks, you can also run the following npm commands as required:

- Delete an existing component from the npm repository.
  ```
  npm unpublish @scope/packageName@version
  ```
- Obtain tags.
  ```
  npm dist-tag list @scope/packageName
  ```
- Add a tag.
  ```
  npm dist-tag add @scope/packageName@version tagName --registry registryUrl --verbose
  ```
- Delete a tag.
  ```
  npm dist-tag rm @scope/packageName@version tagName --registry registryUrl --verbose
  ```

Command parameter description:

- **scope**: path of a self-hosted repo. For details about how to obtain the path, see **Releasing a Component to an npm Repository**.
- **packageName**: the part following **scope** in the **name** field of the **package.json** file.
- **version**: value of the **version** field in the **package.json** file.
- **registryUrl**: URL of the self-hosted repo referenced by **scope** in the configuration file.
- **tagName**: tag name.

The following uses the component released in **Releasing a Component to an npm Repository** as an example:

- **scope**: **test**
- **packageName**: **vue-demo**
- **version**: **1.0.0**

The command for deleting this component is as follows:

```
npm unpublish @test/vue-demo@1.0.0
```

4 Releasing/Obtaining a Go Component via a Build Task

This section describes how to release a component to a Go repository via a build task and obtain a dependency from the repository for deployment.

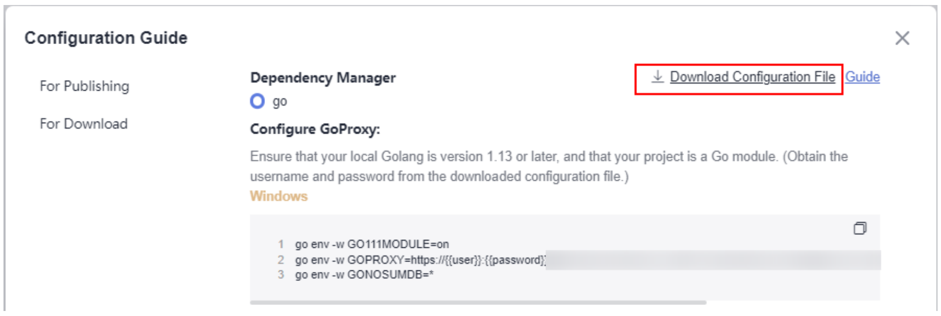
Prerequisites

- You already have a project. If no project is available, **create one**.
- You have created a Go repository.
- You have permissions for the current repository. For details, see **Configuring Repository Permissions 2.0**

Releasing a Component to a Go Repository

Step 1 Download the configuration file.

1. Log in to CodeArts Artifact and access the Go repository. Click **Tutorial** on the right of the page.
2. In the displayed dialog box, click **Download Configuration File**.



Ensure that your local Golang is version 1.13 or later, and that your project is a Go module. (Obtain the username and password from the downloaded configuration file.)

Windows

```
go env -w GO111MODULE=on
go env -w GOPROXY=https://{{user}}:{{password}}[redacted]
go env -w GONOSUMDB=*
```

Step 2 Configure a repository.

1. Go to Repo. Create a Go repository. For details, see **Creating a Repository** This procedure uses the **Go web Demo** template.
2. Prepare the **go.mod** and upload it to the root directory of the repository. For details, see **Uploading Code Files to CodeArts Repo** The following figure shows the **go.mod** file used in this example.

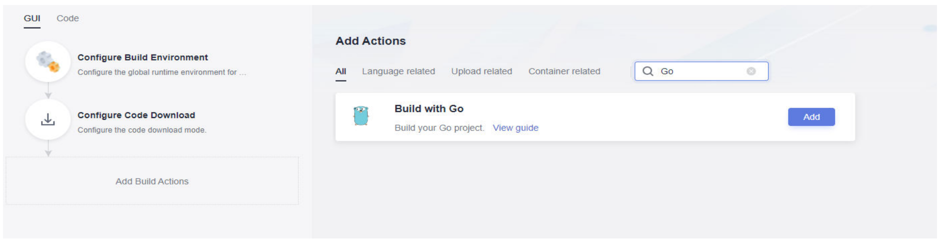
```
go.mod
module example.com/demo
```

Step 3 Configure and run a build task.

1. On the Repo page, select a repository and click **Create Build Task** in the upper right.

   Select **Blank Template** and click **OK**.

2. Add the **Build with Go** action.



3. Edit the **Build with Go** action.
   - Select the desired tool version. In this example, **go-1.13.1** is used.
   - Delete the existing commands, open the configuration file downloaded in **Step 1**, and copy the commands for configuring Go environment variables in Linux to the command box.
   - Copy the Go upload command segment in the configuration file to the command box, and replace the parameters in the commands by referring to **Go Module Packaging**. (In this example, the package version is **v1.0.0**.)

4. Click **Save and Run** on the right of the page to start the build task.

   When the message **build successful** is displayed, go to the self-hosted repo page and find the uploaded Go component.

----End

Obtaining a Dependency from a Go Repository

The following procedure uses the Go component released in **Releasing a Component to a Go Repository** as an example to describe how to obtain a dependency from a Go repository.

Step 1 Download the configuration file by referring to **Releasing a Component to a Go Repository**.

Step 2 Go to Repo and create a Go repository. For details, see **Creating a Repository**. This procedure uses the **Go web Demo** template.

Step 3 Configure and run a build task.

1. On the Repo page, select a repository and click **Create Build Task** in the upper right.

   Select **Blank Template** and click **OK**.

2. Add the **Build with Go** action.

3. Edit the **Build with Go** action.
   - Select the desired tool version. In this example, **go-1.13.1** is used.
   - Delete the existing commands, open the downloaded configuration file, and copy the commands for configuring Go environment variables in Linux to the command box.
   - Copy the Go download commands in the configuration file to the command box and replace the **<modulename>** parameter with the actual value. (In this example, the parameter is set to **example.com/demo**).

Step 4 Click **Save and Run** on the right of the page to start the build task.

When a message **build successful** is displayed, view the task details. If information similar to the following is found in the log, the dependency has been downloaded from the self-hosted repo.

----End

Go Module Packaging

This section describes how to build and upload Go components through Go module packaging.

Perform the following steps:

1. Create a source folder in the working directory.
```
mkdir -p {module}@{version}
```
2. Copy the code source to the source folder.
```
cp -rf . {module}@{version}
```
3. Compress the component into a ZIP package.
```
zip -D -r [package name] [package root directory]
```
4. Upload the component ZIP package and the **go.mod** file to the self-hosted repo.
```
curl -u {{username}}:{{password}} -X PUT {{repoUrl}}/{filePath} -T {{localFile}}
```

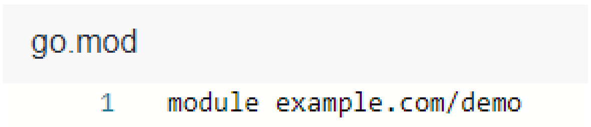
The component directory varies according to the package version. The version can be:

- Versions earlier than v2.0: The directory is the same as the path of the **go.mod** file. No special directory structure 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 component directories for different versions:

- **Versions earlier than v2.0**

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



go.mod
```
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 component 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 component 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 component 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.

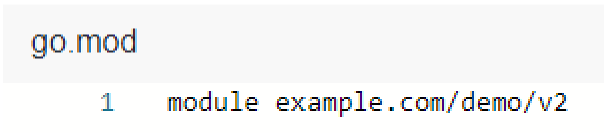
   - For the ZIP package, the value of **filePath** is **example.com/demo/@v/v1.0.0.zip** and that of **localFile** is **v1.0.0.zip**.
   - For the **go.mod** file, the value of **filePath** is **example.com/demo/@v/v1.0.0.mod** and that of **localFile** is **example.com/demo@v1.0.0/go.mod**.

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

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

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



go.mod

```
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 component 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 component 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 component 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 go.mod not ending with /vX**

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

go.mod

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

5 Releasing/Obtaining a PyPI Component via a Build Task

This section describes how to release a component to a PyPI repository via a build task and obtain a dependency from the repository for deployment.

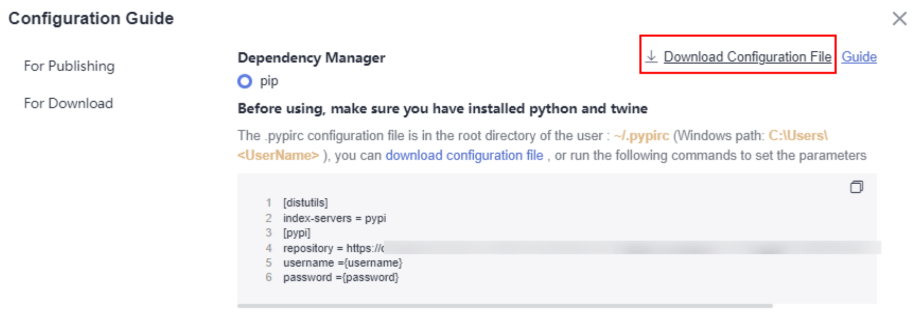
Prerequisites

- You already have a project. If no project is available, **create one**.
- You have created a PyPI repository.
- You have permissions for the current repository. For details, see **Configuring Repository Permissions 2.0**

Releasing a Component to a PyPI Repository

Step 1 Download the configuration file.

1. Log in to CodeArts Artifact and access the PyPI repository. Click **Tutorial** on the right of the page.
2. In the displayed dialog box, find the **For Publishing** and click **Download Configuration File**.



Before using, make sure you have installed python and twine

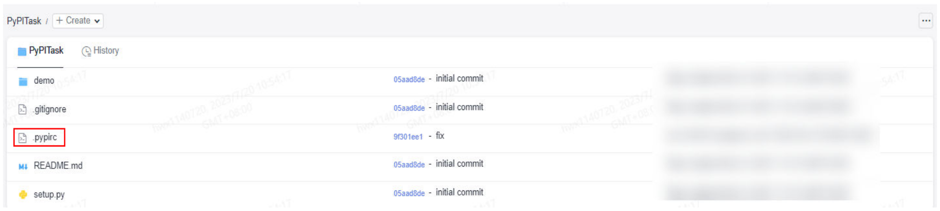
The .pypirc configuration file is in the root directory of the user : ~/.pypirc (Windows path: C:\Users\<UserName>), you can download configuration file , or run the following commands to set the parameters

```
[distutils]
index-servers = pypi
[pypi]
repository = https://c[redacted]
username ={username}
password ={password}
```

3. Save the downloaded **PYPIRC** file as a **.pypirc** file.

Step 2 Configure a repository.

1. Go to Repo and create a Python repository. For details, see **Creating a Repository**. This procedure uses the **Python3 Demo** template.
2. Go to the repository and upload the **.pypirc** file to the root directory of the repository. For details, see .

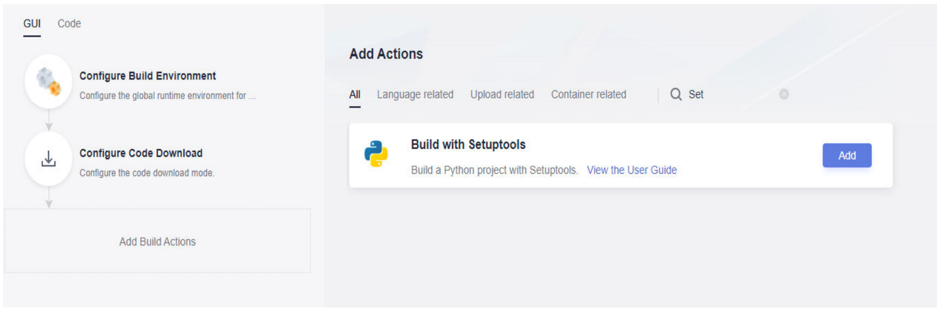

PyPITask / + Create
PyPITask
History
demo
05aad8de · initial commit
.gitignore
05aad8de · initial commit
.pypirc
9f301ee1 · fix
README.md
05aad8de · initial commit
setup.py
05aad8de · initial commit

Step 3 Configure and run a build task.

1. On the Repo page, select a repository and click **Create Build Task** in the upper right.

   Select **Blank Template** and click **OK**.

2. Add the **Build with Setuptools** action.



3. Edit the **Build with Setuptools** action.
   - Select the desired tool version. In this example, **python3.6** is used.
   - Delete the existing commands and run the following instead:

```
# Ensure that the setup.py file exists in the root directory of the code, and run the following command to pack the project into a WHL package.
python setup.py bdist_wheel
# Set the .pypirc file in the root directory of the current project as the configuration file.
cp -rf .pypirc ~/
# Upload the component to the PyPI repository.
twine upload -r pypi dist/*
```

     If certificate verification fails during the upload, add the following command to the first line of the preceding command to skip certificate verification:

```
export CURL_CA_BUNDLE=""
```

4. Click **Save and Run** on the right of the page to start the build task.

   After the task is successfully executed, go to the self-hosted repo page and find the uploaded PyPI component.

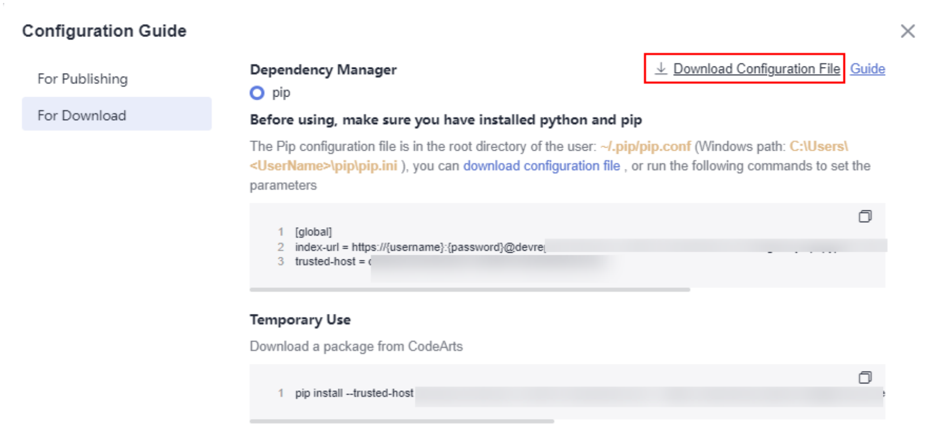
----End

Obtaining a Dependency from a PyPI Repository

The following procedure uses the PyPI component released in **Releasing a Component to a PyPI Repository** as an example to describe how to obtain a dependency from a PyPI repository.

Step 1 Download the configuration file.

1. Go to the PyPI repository and click **Tutorial** on the right of the page.
2. In the displayed dialog box, find the **For Download** and click **Download Configuration File**.



Dependency Manager

pip

Before using, make sure you have installed python and pip

The Pip configuration file is in the root directory of the user: ~/.pip/pip.conf (Windows path: C:\Users\<UserName>\pip\pip.ini), you can download configuration file , or run the following commands to set the parameters

```
[global]
index-url = https://{username}:{password}@devre[redacted]
trusted-host = [redacted]
```

Temporary Use

Download a package from CodeArts

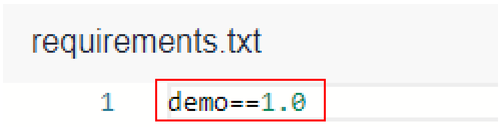
```
pip install --trusted-host [redacted]
```

3. Save the downloaded **pip.ini** file as a **pip.conf** file.

Step 2 Configure a repository.

1. Go to Repo and create a Python repository. For details, see **Creating a Repository**. This procedure uses the **Python3 Demo** template.
2. Go to Repo, and upload the **pip.conf** file to the root directory of the repository where the PyPI dependency is to be used.
3. Find the **requirements.txt** file in the repository and open it. If the file is not found, create it by referring to **Managing Files**. Add the dependency configuration to this file, as shown in the following figure.

```
demo ==1.0
```



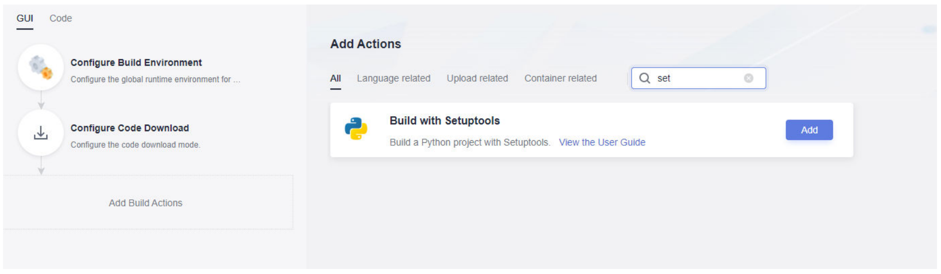
requirements.txt

```
demo==1.0
```

Step 3 Configure and run a build task.

1. On the Repo page, select a repository and click **Create Build Task** in the upper right.

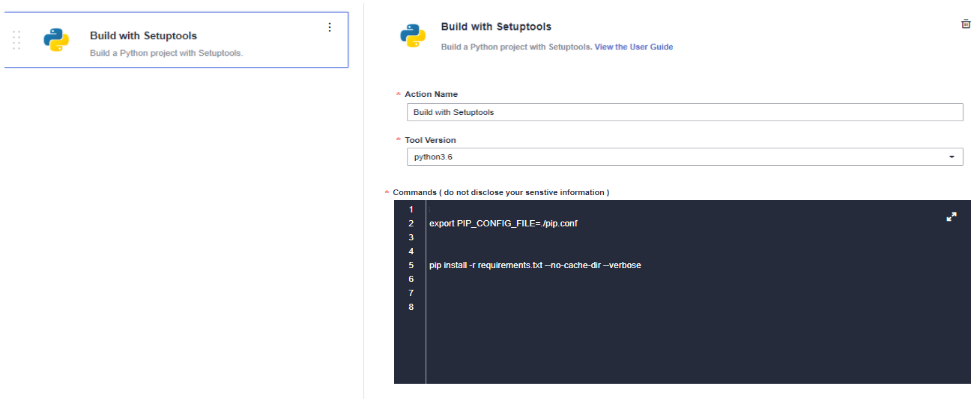
   Select **Blank Template** and click **OK**.
2. Add the **Build with Setuptools** action.



3. Edit the **Build with Setuptools** action.
   - Select the desired tool version. In this example, **python3.6** is used.
   - Delete the existing commands and run the following instead:

```
# Set the pip.conf file in the root directory of the current project as the configuration file.
 export PIP_CONFIG_FILE=./pip.conf
# Download the PyPI component.
 pip install -r requirements.txt --no-cache-dir
```

Copyright © Huawei Technologies Co., Ltd.


Build with Setuptools
Build a Python project with Setuptools.
Build with Setuptools
Build a Python project with Setuptools. View the User Guide

* Action Name
Build with Setuptools

* Tool Version
python3.6

* Commands (do not disclose your senstive information)

```
export PIP_CONFIG_FILE=./pip.conf


pip install -r requirements.txt --no-cache-dir --verbose
```

Step 4 Click **Save and Run** on the right of the page to start the build task.

After the task is successfully executed, view the task details. If information similar to the following is found in the log, the dependency has been downloaded from the self-hosted repo.

----End

6 Uploading/Obtaining an RPM Component Using Linux Commands

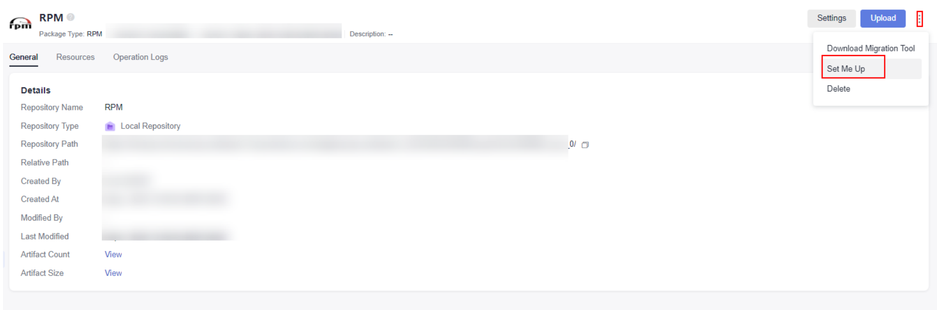
This section describes how to use Linux commands to upload a component to an RPM repository and obtain a dependency from the repository.

Prerequisites

- An RPM component is available.
- A Linux host that can connect to the public network is available.
- You have created an RPM repository.
- You have permissions for the current repository. For details, see **Configuring Repository Permissions 2.0**

Releasing a Component to an RPM Repository

Step 1 Log in to CodeArts Artifact and access the RPM repository. Click **Tutorial** on the right of the page.



RPM
Package Type: RPM Description: --

Details

Repository Name	RPM
Repository Type	Local Repository
Repository Path	0/
Relative Path	
Created By	
Created At	
Modified By	
Last Modified	
Artifact Count	View
Artifact Size	View

Step 2 In the displayed dialog box, click **Download Configuration File**.

Step 3 On the Linux host, run the following command to upload an RPM component:

```
curl -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**.

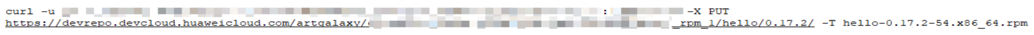
- *user*: character string before the colon (:) between **curl -u** and **-X**

- *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 component. The **hello-0.17.2-54.x86_64.rpm** component is used as an example.

- *component*: software name, for example, **hello**.
- *version*: software version, for example, **0.17.2**.
- *localFile*: RPM component, for example, **hello-0.17.2-54.x86_64.rpm**.

The following figure shows the complete command.



```
curl -u [redacted]:[redacted] -X PUT
https://devrepo.devcloud.huaweicloud.com/artgalaxy/[redacted]_rpm_1/hello/0.17.2/ -T hello-0.17.2-54.x86_64.rpm
```

Step 4 After the commands are successfully executed, go to the self-hosted repo and find the uploaded RPM component.

----End

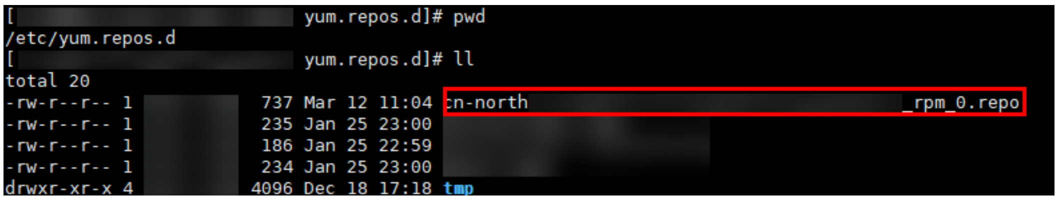
Obtaining a Dependency from an RPM Repository

The following procedure uses the RPM component released in **Releasing a Component to an RPM Repository** as an example to describe how to obtain a dependency from an RPM repository.

Step 1 Download the configuration file by referring to **Releasing a Component to an RPM Repository**.

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.



```
[[redacted] yum.repos.d]# pwd
/etc/yum.repos.d
[[redacted] yum.repos.d]# ll
total 20
-rw-r--r-- 1 [redacted]  737 Mar 12 11:04 cn-north[redacted]_rpm_0.repo
-rw-r--r-- 1             235 Jan 25 23:00 [redacted]
-rw-r--r-- 1             186 Jan 25 22:59
-rw-r--r-- 1             234 Jan 25 23:00
drwxr-xr-x 4            4096 Dec 18 17:18 tmp
```

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

```
yum install hello
```

----End

7 Uploading/Obtaining a Debian Component Using Linux Commands

This section describes how to use Linux commands to upload a component to a Debian repository and obtain a dependency from the repository.

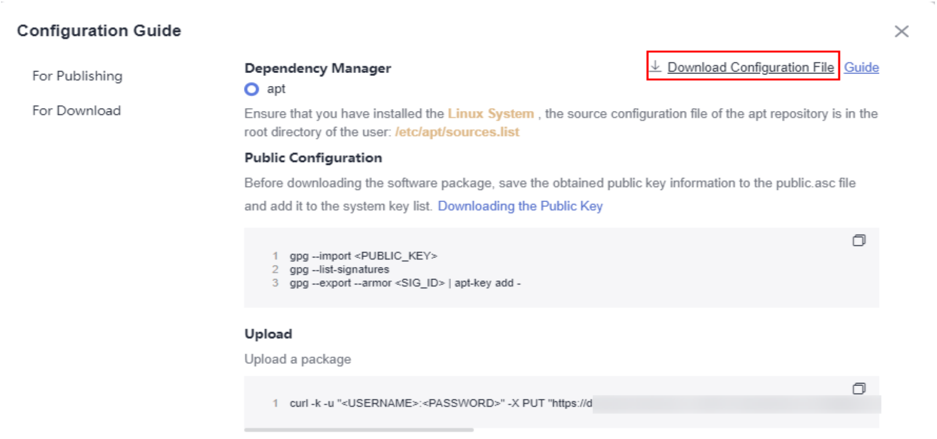
Prerequisites

- A Debian component is available.
- A Linux host that can connect to the public network is available.
- You have created a Debian repository.
- You have permissions for the current repository. For details, see **Configuring Repository Permissions 2.0**

Releasing a Component to a Debian Repository

Step 1 Log in to CodeArts Artifact and access the Debian repository. Click **Tutorial** on the right of the page.

Step 2 In the displayed dialog box, click **Download Configuration File**.



Dependency Manager

apt

Ensure that you have installed the Linux System , the source configuration file of the apt repository is in the root directory of the user: /etc/apt/sources.list

Public Configuration

Before downloading the software package, save the obtained public key information to the public.asc file and add it to the system key list. Downloading the Public Key

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

Upload

Upload a package

```
curl -k -u "<USERNAME>:<PASSWORD>" -X PUT "https://d[redacted]
```

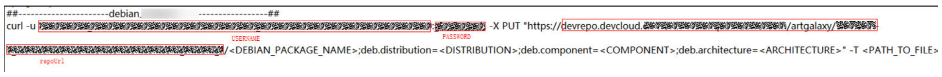
Step 3 On the Linux host, run the following command to upload a Debian component:

Copyright © Huawei Technologies Co., Ltd.

```
curl -u <USERNAME>:<PASSWORD> -X PUT "https:// <repoUrl>/
<DEBIAN_PACKAGE_NAME>;deb.distribution=<DISTRIBUTION>;deb.component=<COMPONENT>;deb.archite
cture=<ARCHITECTURE>" -T <PATH_TO_FILE>
```

In this command, **USERNAME**, **PASSWORD**, and **repoUrl** can be obtained from the **Debian upload command** in the configuration file downloaded in the **previous step**.

- **USERNAME**: username used for uploading files, which can be obtained from the Debian configuration file. For details, see the example figure.
- **PASSWORD**: password used for uploading files, which can be obtained from the Debian configuration file. For details, see the example figure.
- **repoUrl**: URL used for uploading files, which can be obtained from the Debian configuration file. For details, see the example figure.



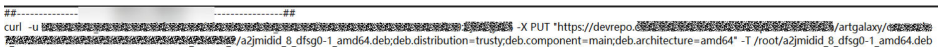
```
##-----------------------debian_[redacted]-----------------##
curl -u [redacted]:[redacted] -X PUT "https://devrepo.devcloud.[redacted]/artgalaxy/[redacted]
[redacted]/<DEBIAN_PACKAGE_NAME>;deb.distribution=<DISTRIBUTION>;deb.component=<COMPONENT>;deb.architecture=<ARCHITECTURE>" -T <PATH_TO_FILE>
```

DEBIAN_PACKAGE_NAME, **DISTRIBUTION**, **COMPONENT**, and **ARCHITECTURE** can be obtained from the Debian component.

The **a2jmidid_8_dfsg0-1_amd64.deb** component is used as an example.

- **DEBIAN_PACKAGE_NAME**: software package name, for example, **a2jmidid_8_dfsg0-1_amd64.deb**.
- **DISTRIBUTION**: release version, for example, **trusty**.
- **COMPONENT**: component name, for example, **main**.
- **ARCHITECTURE**: system architecture, for example, **amd64**.
- **PATH_TO_FILE**: local storage path of the Debian component, for example, **/root/a2jmidid_8_dfsg0-1_amd64.deb**.

  The following figure shows the complete commands.



```
##--------------[redacted]----------------##
curl  -u [redacted]:[redacted] -X PUT "https://devrepo.[redacted]/artgalaxy/[redacted]
[redacted]/a2jmidid_8_dfsg0-1_amd64.deb;deb.distribution=trusty;deb.component=main;deb.architecture=amd64" -T /root/a2jmidid_8_dfsg0-1_amd64.deb
```

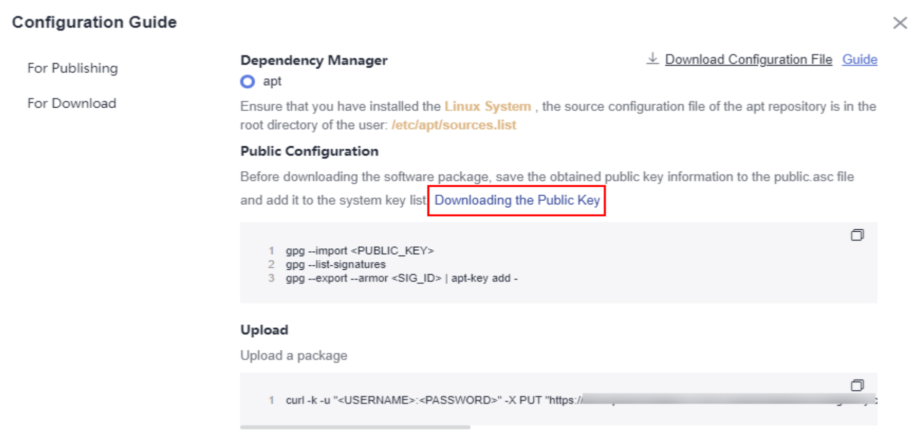
Step 4 After the commands are successfully executed, go to the self-hosted repo and find the uploaded Debian component.

----End

Obtaining a Dependency from a Debian Repository

The following procedure uses the Debian component released in **Releasing a Component to a Debian Repository** as an example to describe how to obtain a dependency from a Debian repository.

Step 1 Download the **public key** file of the Debian repository by referring to **Releasing a Component to a Debian Repository**.



Configuration Guide

For Publishing

For Download

Dependency Manager

Download Configuration File Guide

● apt

Ensure that you have installed the Linux System , the source configuration file of the apt repository is in the root directory of the user: /etc/apt/sources.list

Public Configuration

Before downloading the software package, save the obtained public key information to the public.asc file and add it to the system key list Downloading the Public Key

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

Upload

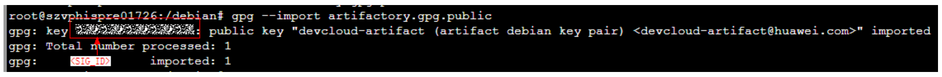
Upload a package

```
curl -k -u "<USERNAME>:<PASSWORD>" -X PUT "https://[redacted]
```

Step 2 Import the gpg public key.

```
gpg --import <PUBLIC_KEY_PATH>
```

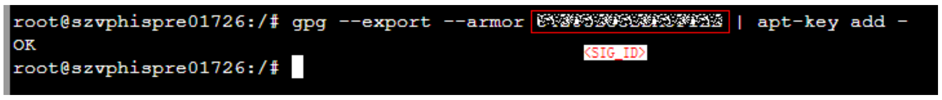
PUBLIC_KEY_PATH: local path for storing the Debian public key, for example, **artifactory.gpg.public**.



```
root@szvphispre01726:/debian# gpg --import artifactory.gpg.public
gpg: key [redacted]: public key "devcloud-artifact (artifact debian key pair) <devcloud-artifact@huawei.com>" imported
gpg: Total number processed: 1
gpg:     <SIG_ID>      imported: 1
```

Step 3 Add the public key to the list of keys used by apt to authenticate packages.

```
gpg --export --armor <SIG_ID> | apt-key add -
```



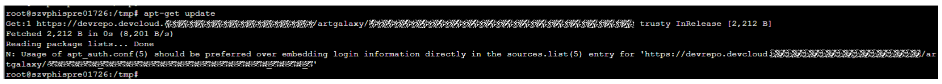
```
root@szvphispre01726:/# gpg --export --armor [redacted] | apt-key add -
OK                                           <SIG_ID>
root@szvphispre01726:/#
```

Step 4 Add the apt repository source.

Open the configuration file (for details about how to obtain the file, see **Releasing a Component to a Debian Repository**), replace all **DISTRIBUTION** fields with the value of **COMPONENT** (for example, **main**) used for uploading the Debian file, and add the repository source based on the downloaded configuration file **sources.list**.

Step 5 After the repository source is added, run the following command to update the repository source:

```
apt-get update
```



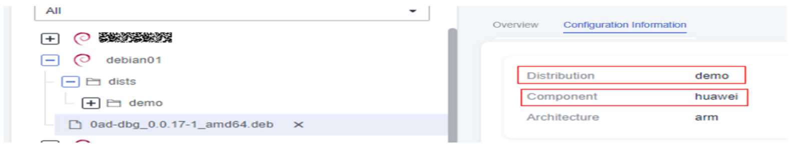
```
root@szvphispre01726:/tmp# apt-get update
Get:1 https://devrepo.devcloud.[redacted]/artgalaxy/[redacted] trusty InRelease [2,212 B]
Fetched 2,212 B in 0s (8,201 B/s)
Reading package lists... Done
N: Usage of apt_auth.conf(5) should be preferred over embedding login information directly in the sources.list(5) entry for 'https://devrepo.devcloud[redacted]/ar
tgalaxy/[redacted]'
root@szvphispre01726:/tmp#
```

Step 6 Run the following command to download the Debian package: Replace **a2jmidid** with the actual value of **PACKAGE**.

```
apt download a2jmidid
```

To obtain *<PACKAGE>*, perform the following steps:

Download the Packages source data of the Debian component. The following uses the **a2jmidid** package as an example.



Distribution	demo
Component	huawei
Architecture	arm

----End

8 Batch Migrating Maven/npm/PyPI Components to a Self-Hosted Repo

Background

Self-hosted repos allow you to manually upload and download components. They can also interconnect with your local development environment to upload and download components. For details, see **Uploading/Downloading Components on the Self-Hosted Repo Page**.

Uploading numerous packages one at a time can be cumbersome. You can use the migration tool provided by self-hosted repos to upload components in batches from Nexus or other repositories.

Preparations

- You have created a self-hosted repo of the corresponding type.
- You have the Python3 environment available.
- If Nexus is used, the migration tool and Nexus must run on the same Linux host and be connected to the CodeArts service network. Python3 must be installed on the host.

Migrating a Maven Component

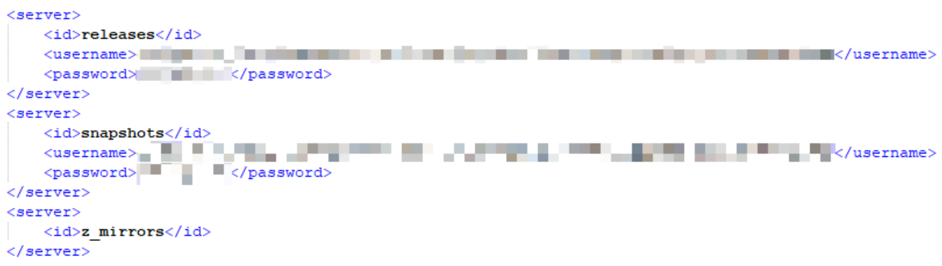
Step 1 Find the components to be migrated from the local Maven repository (for example, **C:\Users\***xxxxx***\\.m2\repository**) and copy them to a specified directory.

Step 2 Go to the self-hosted repo page and select the target Maven repository in the left pane.

Step 3 Click the repository name. The **Repository Path** is displayed on the right. Click ⧉ to copy it.

Step 4 Click **Tutorial** in the upper right corner of the page. In the displayed dialog box, click **Download Configuration File** to download the **settings.xml** file to your local host.

Open the file on the local host and find the username and password.



```
<server>
    <id>releases</id>
    <username>[redacted]</username>
    <password>[redacted]</password>
</server>
<server>
    <id>snapshots</id>
    <username>[redacted]</username>
    <password>[redacted]</password>
</server>
<server>
    <id>z_mirrors</id>
</server>
```

Step 5 Click ··· in the upper right corner of the page and click **Download Migration Tool** (**uploadArtifact2.py** script and **artifact.conf** configuration file) to the local host.

Step 6 Configure **artifact.conf**.

```
[artifact]
packageType = Component type. Set it to Maven.
userInfo = username:password (username and password obtained in step 4)
repoRelease = Repository path (Release) (repository path obtained in step 3)
repoSnapshot = Repository path (Snapshot) (repository path obtained in step 3)
srcDir = Component directory (specified by users), for example, the target directory for storing the
downloaded component in step 1.

[nexus]
nexusAddr = Nexus address
nexusPort = Nexus port
repoName = Name of the Nexus repository to be migrated
userName = Nexus username
passwd = Nexus password
```

Step 7 Run the migration script **python uploadArtifact2.py**.

Step 8 Go to the target self-hosted repo and check whether the component package is uploaded.

----End

Migrating an npm Component

Step 1 Go to the self-hosted repo page and select the target npm repository in the left pane.

Step 2 Click the repository name. The **Repository Path** is displayed on the right. Click ⧉ to copy it.

Step 3 Click **Tutorial** in the upper right corner of the page. In the displayed dialog box, click **Download Configuration File** to download the **npmrc** configuration file to your local host.

Open the configuration file on the local host, find the value of the **_auth** field, and decode the value using base64.

Step 4 Click ··· in the upper right corner of the page and click **Download Migration Tool** (**uploadArtifact2.py** script and **artifact.conf** configuration file) to the local host.

Step 5 Configure **artifact.conf**.

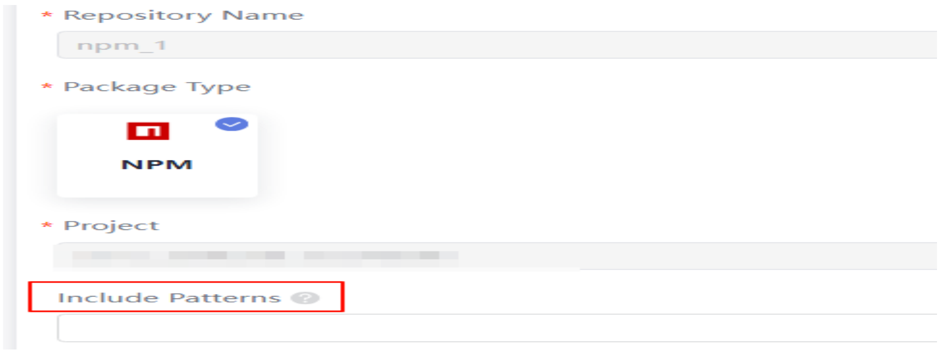
```
[artifact]
packageType = Component type. Set it to npm.
userInfo = Value of the _auth field decoded using base64 in the npmrc configuration file of the npm
repository. (For details, see step 3.)
```

```
repoRelease = Repository path (repository path obtained in step 2)
repoSnapshot = Left empty
srcDir = Component directory, for example, C:|Users|xxxxxx|repository. You can specify a directory.


[nexus]
nexusAddr = Nexus address
nexusPort = Nexus port
repoName = Name of the Nexus repository to be migrated
userName = Nexus username
passwd = Nexus password
```

Step 6 Check whether the **Include Patterns** is configured for the target npm repository.

Check whether the private binary packages in the **package.json** file are in the whitelist. Only packages on the whitelist can be uploaded successfully. If no whitelist is configured, all private binary packages in the **package.json** file can be uploaded successfully.



Step 7 Run the migration script **python uploadArtifact2.py**.

Step 8 Go to the target self-hosted repo and check whether the component package is uploaded.

----End

Migrating a PyPI Component

Step 1 Go to the self-hosted repo page and select the target PyPI repository in the left pane.

Step 2 Click the repository name. The **Repository Path** is displayed on the right. Click ⧉ to copy it.

Step 3 Click **Tutorial** in the upper right corner of the page. In the displayed dialog box, click **Download Configuration File** to download the **pypirc** file to your local host.

Open the file on the local host and find the username and password.

Step 4 Click ··· in the upper right corner of the page and click **Download Migration Tool** (**uploadArtifact2.py** script and **artifact.conf** configuration file) to the local host.

Step 5 Configure **artifact.conf**.

```
[artifact]
packageType = Component type. Set it to PyPI.
userInfo = username:password (username and password obtained in step 3)
repoRelease = Repository path (repository path obtained in step 2)
repoSnapshot = Left empty
```

Copyright © Huawei Technologies Co., Ltd.

```
srcDir = Component directory, for example, C:|Users|xxxxxx|repository. You can specify a directory.

[nexus]
nexusAddr = Nexus address
nexusPort = Nexus port
repoName = Name of the Nexus repository to be migrated
userName = Nexus username
passwd = Nexus password
```

Step 6 Run the migration script **python uploadArtifact2.py**.

Step 7 Go to the self-hosted repo page and check whether the binary package is successfully uploaded.

----End