CodeArts Artifact

Getting Started

Issue 01

Date 2024-10-18

HUAWEI

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions


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

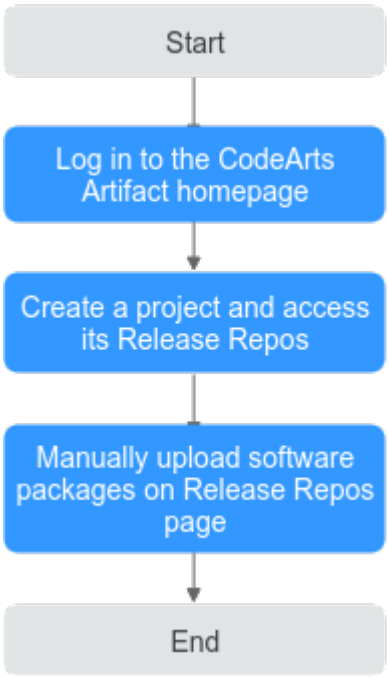
Contents

1 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 1-1** shows the main operation process.

Figure 1-1 Uploading software packages to Release Repos



Preparations

- You have **registered a HUAWEI ID and enabled Huawei Cloud services**.

- You have **subscribed to CodeArts Artifact**.

Logging In to CodeArts Artifact Homepage

Step 1 **Log in to the Huawei Cloud 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 Click **Homepage** in the navigation pane.

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.

NOTE

You do not need to manually create Release Repos. After you create a project, Release Repos with the same name is automatically generated under the project.

----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 **Repository View**. Uploaded software packages are stored in this folder.
- **File**: Select software packages from your local PC to upload.

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

----End

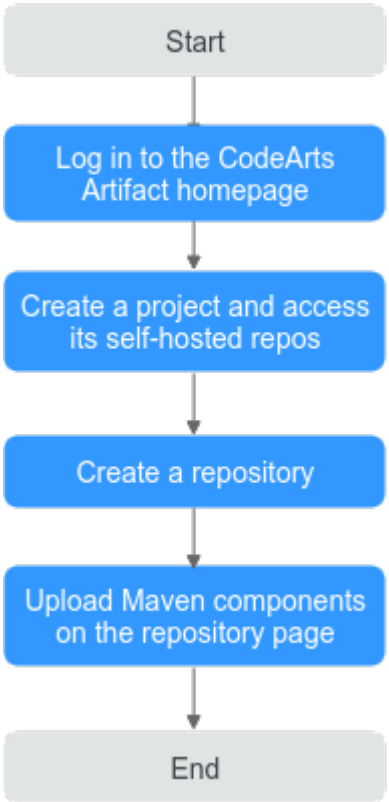
CodeArts Artifact allows you to upload software packages either from the page or through CodeArts Build to Release Repos. For details, see **Uploading Software Packages to Release Repos**.

2 Uploading Components to Maven Repository

Developers often need to share some components with other team members during routine development. Self-hosted repos serve as a shared site where components can be stored and uploaded for sharing. This makes it easy for other team members to obtain components from repositories.

This document describes how to upload components to Maven repository, helping you quickly get started. **Figure 2-1** shows the main operation process.

Figure 2-1 Uploading components to Maven repository



Prerequisites

- You have **registered a HUAWEI ID and enabled Huawei Cloud services**.
- You have **subscribed to CodeArts Artifact**.

Logging In to CodeArts Artifact Homepage

Step 1 **Log in to the Huawei Cloud 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 Click **Homepage** in the navigation pane.

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 Self-Hosted Repo

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 Repository** and **Virtual Repository**. **Local Repository** 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 Components 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 local host.

Step 5 Click **Upload**.

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

----End

CodeArts Artifact allows you to upload components either from the page or through CodeArts Build to self-hosted repos. For details, see **Using Maven for Build**.

3 Releasing/Obtaining a Maven Component via a Build Task

This section describes how to release a Maven component to a self-hosted repo via a build task and obtain the component from the repository for deployment.

Prerequisites

- You already have a project. If no project is available, **create one**.
- You have permissions for the current repository. For details, see **Managing Repository Permissions**
- You have created a Maven repository and **associated it with the project**

Releasing a Maven Component to a Self-Hosted Repo

Step 1 Configure a repository.

1. Log in to CodeArts and go to a created project. Choose **Services** > **Repo** on the top navigation bar.
2. Create a Maven repository. For details, see **Creating a Repository Using a Template**. This procedure uses the **Java Maven Demo** template.
3. Go to the code repository and view the component configuration in the **pom.xml** file.

pom.xml

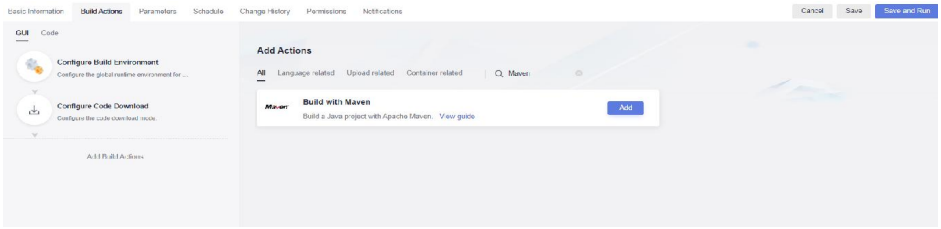
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.huawei.demo</groupId>
  <artifactId>javaMavenDemo</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>
  <name>maven_demo</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
    </dependency>
  </dependencies>
```

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**.

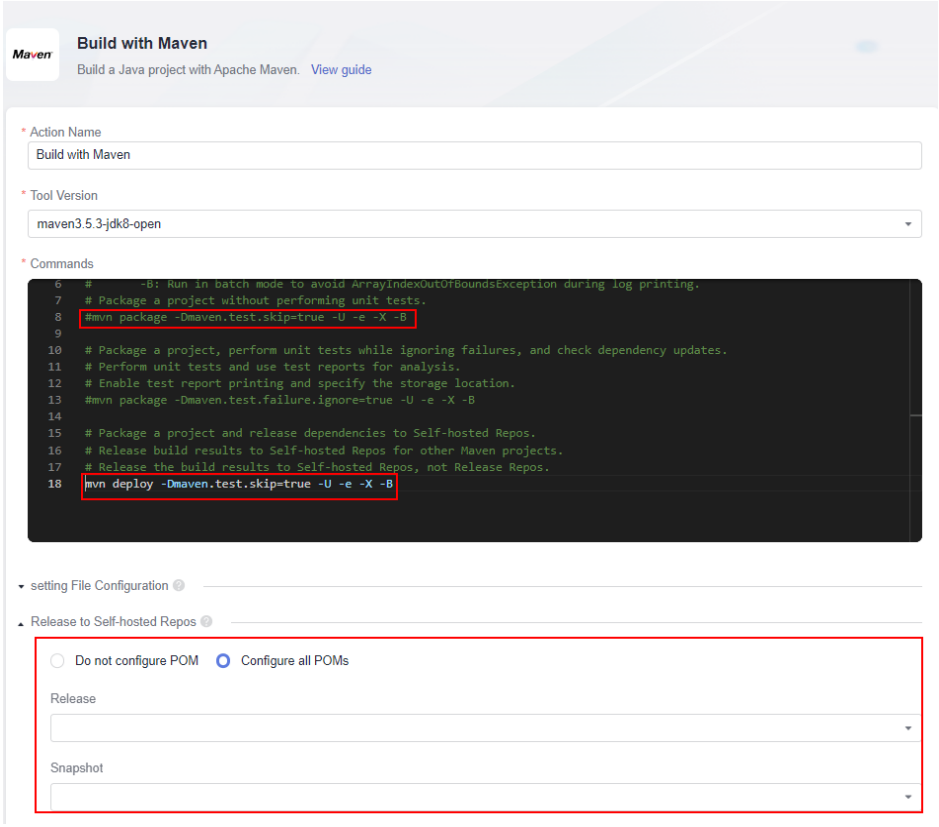
2. Add the **Build with Maven** action.



3. 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.

     NOTE

     If no option is available in the drop-down list, associate the Maven repository with the project of the build task by referring to **Associating Maven Repository with Projects**.



Step 3 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 Maven component.

----End

Obtaining a Maven Component from a Self-Hosted Repo

The following procedure uses the Maven component released in **Releasing a Maven Component to a Self-Hosted Repo** as an example to describe how to obtain the component from a self-hosted repo as a dependency.

Step 1 Configure a repository.

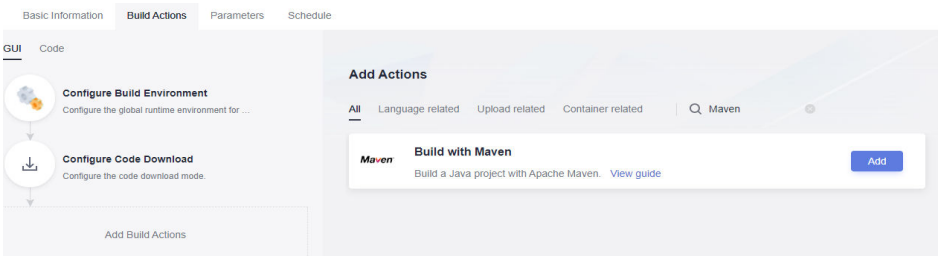
1. Go to the Maven repository and find the Maven component. Click the **.pom** file with the same name as the component and click **Download** on the right.
2. Open the downloaded file and locate the **<groupId>**, **<artifactId>**, and **<version>** lines.

3. Go to Repo. Create a Maven repository. For details, see **Creating a Repository Using a Template**. This procedure uses the **Java Maven Demo** template.
4. Go to the code repository and edit the **pom.xml** file. Copy the dependency code segment to the **dependencies** code segment and modify the version number (for example, **2.0**).

Step 2 Configure and run a build task.

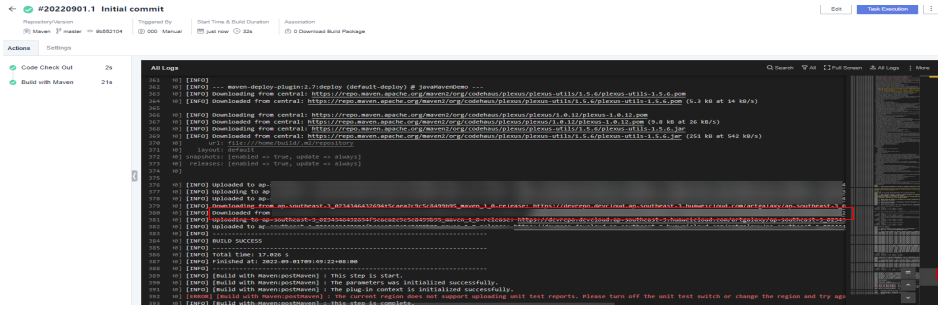
1. On the Repo page, select a repository and click **Create Build Task** in the upper right.

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



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 self-hosted repo.



----End

4 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.

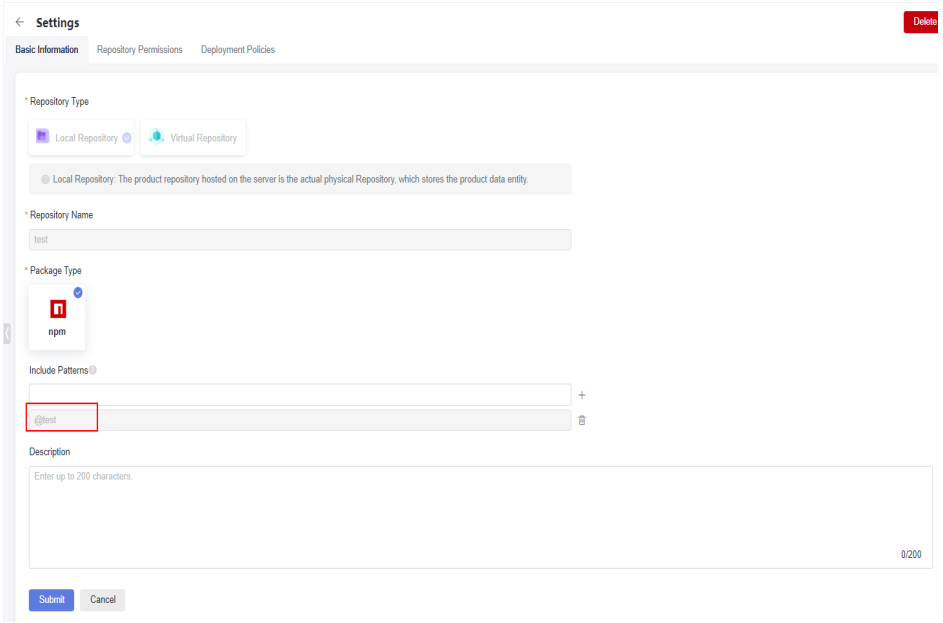
Prerequisites

- 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 **Managing Repository Permissions**

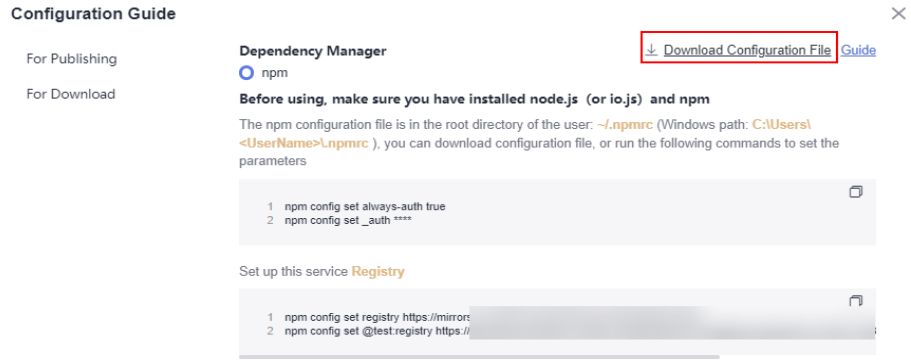
Releasing a Component to an npm Repository

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.



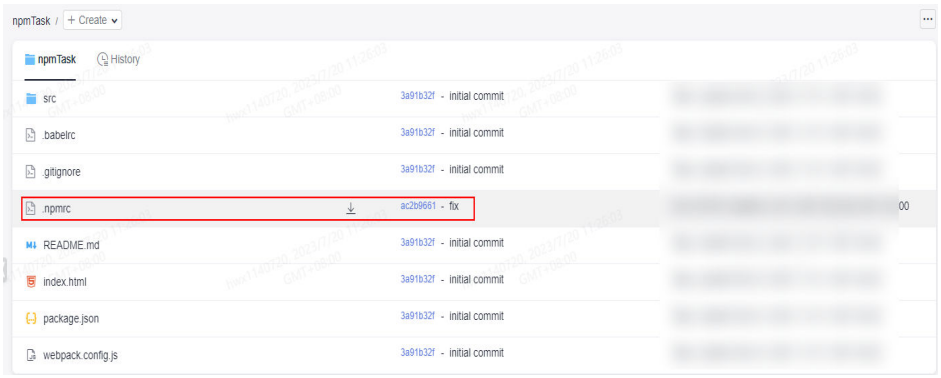
2. Click **Cancel** to return to the npm repository page. Click **Tutorial** on the right of the page.
3. In the displayed dialog box, click **Download Configuration File**.



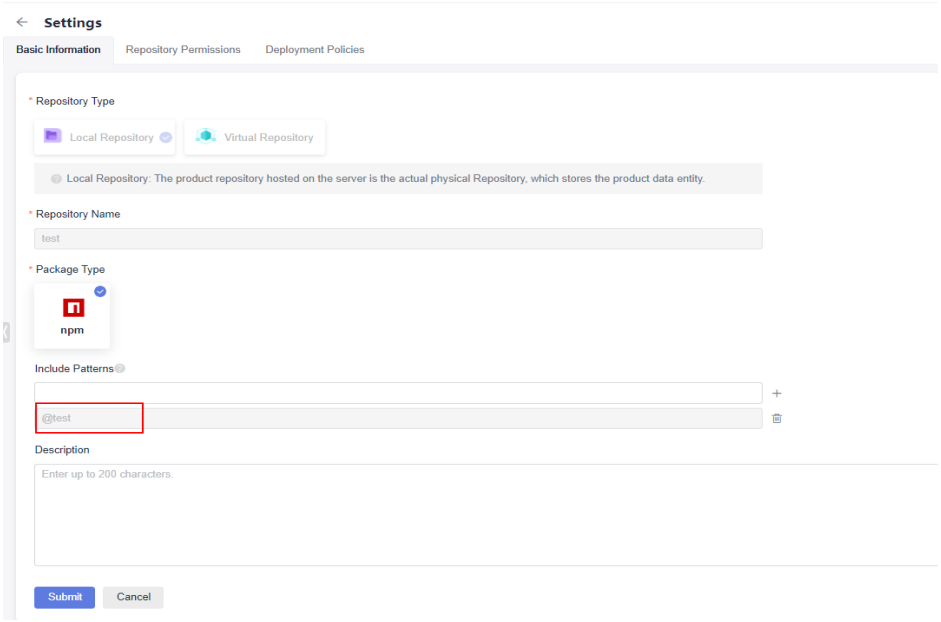
4. Save the downloaded **npmrc** file as an **.npmrc file**.

Step 2 Configure a repository.

1. Go to Repo and create a **Node.js** repository. For details, see **Creating a Repository Using a Template**. 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 .

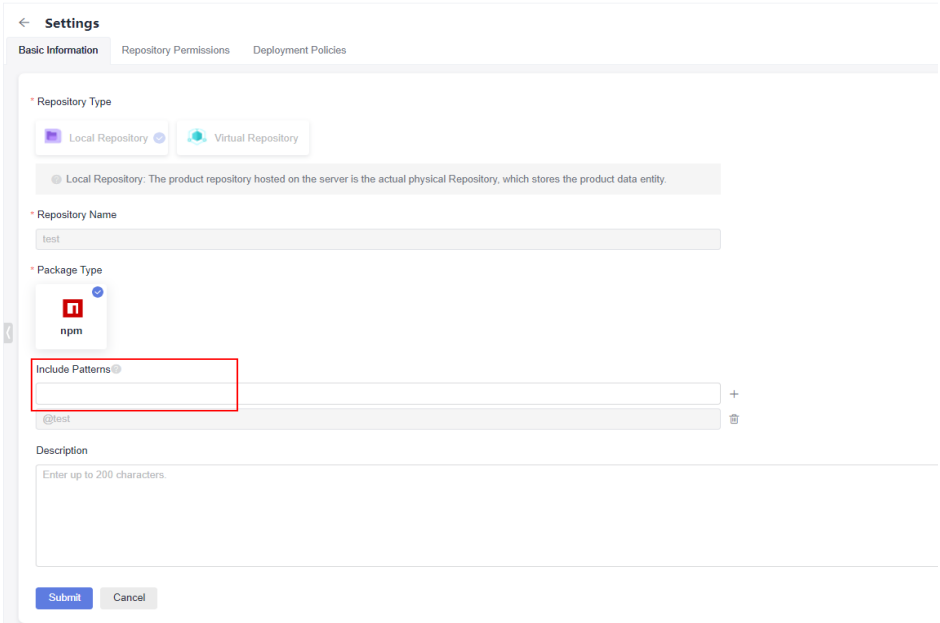


3. Find the **package.json** file in the repository and open it. Add the path recorded on the **Basic Information** under the **Settings** tab page to the **name** field in the file.



NOTE

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

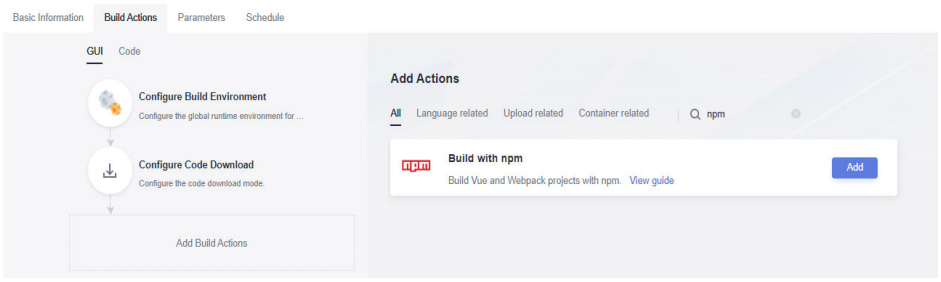


Step 3 Configure and run a build task.

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

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

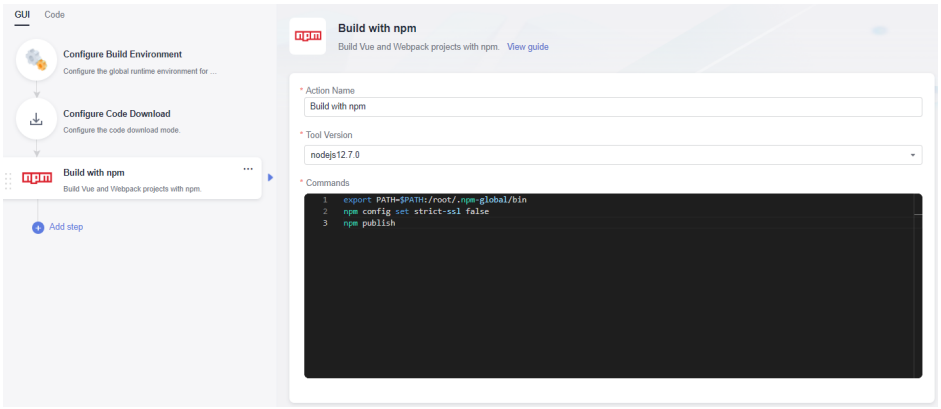
2. Add the **Build with npm** action.



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
     ```



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 Using a Template**. 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:

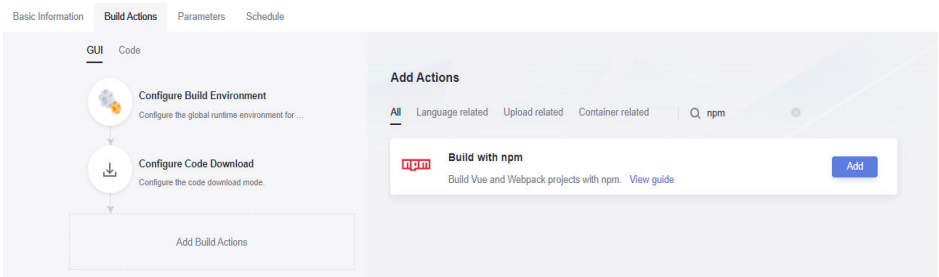
```
"@test/vue-demo": "^1.0.0"
```

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.

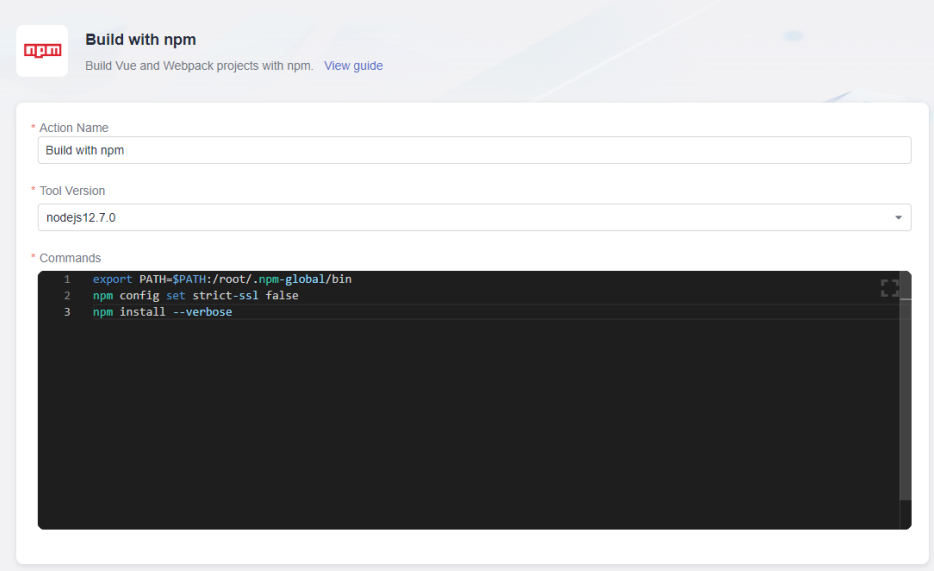
   Select **Blank Template** and click **OK**.

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



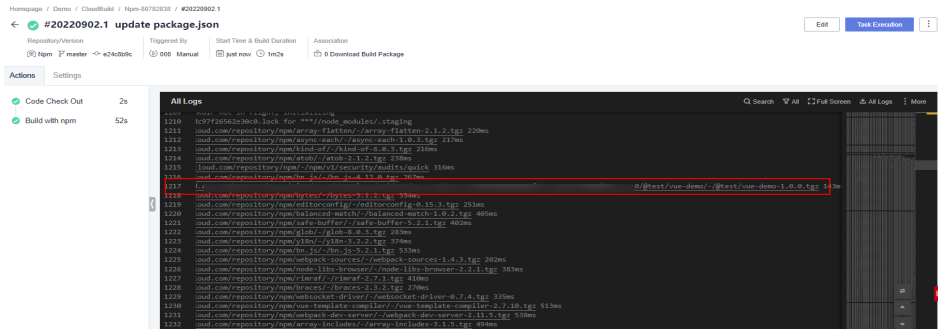
3. Edit the **Build with npm** action.
   - Select the desired tool version. In this example, **nodejs12.7.0** is used.
   - Delete the existing commands and run the following instead:

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

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.



----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 @socpe/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
```


5 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.

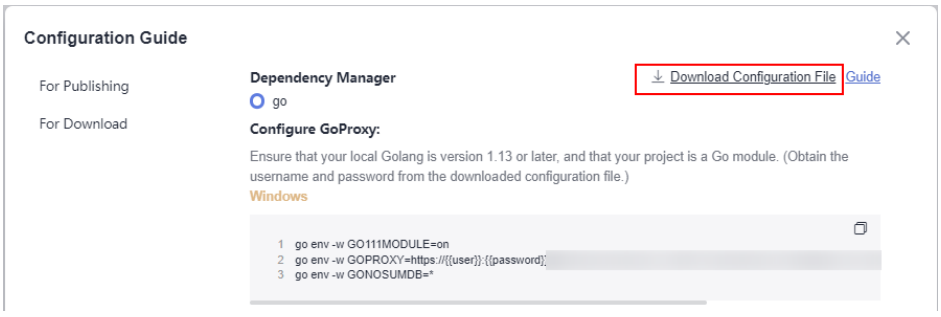
Prerequisites

- 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 **Managing Repository Permissions**

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**.



Step 2 Configure a repository.

1. Go to Repo. Create a Go repository. For details, see **Creating a Repository Using a Template**. 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 . The following figure shows the **go.mod** file used in this example.

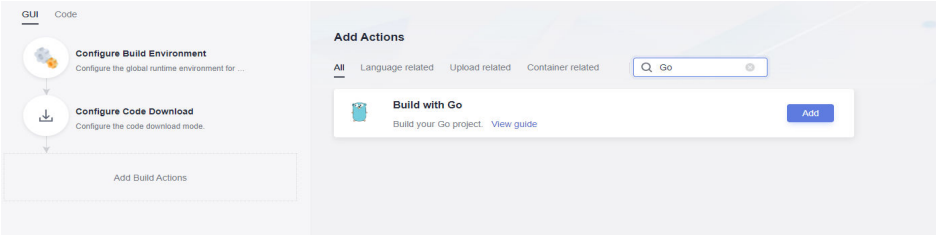

```
go.mod
1    module example.com/demo
```

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 Using a Template**. 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
  ```
  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 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 command is 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

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

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 command is 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

```
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 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 command is 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
```

6 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.

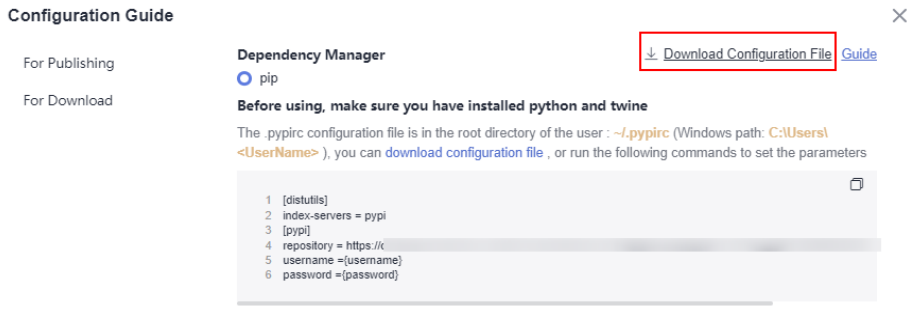
Prerequisites

- 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 **Managing Repository Permissions**

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**.



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

Step 2 Configure a repository.

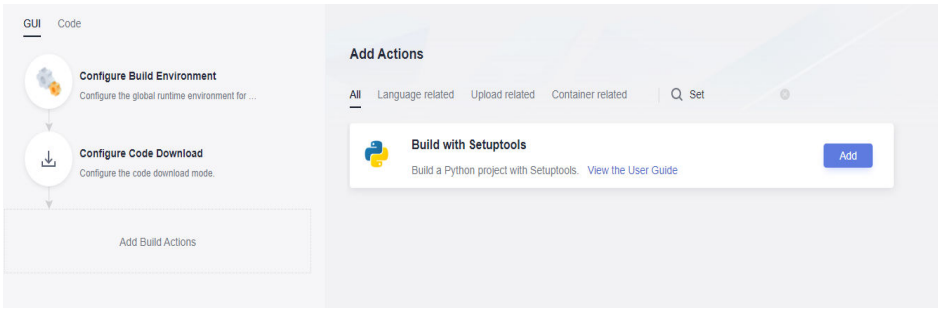
1. Go to Repo and create a Python repository. For details, see **Creating a Repository**. This procedure uses the **Python3 Demo** template.
2. Go to the repository and upload the **.pypirc** file to the root directory of the repository. For details, see .

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/*
```

   NOTE

   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.

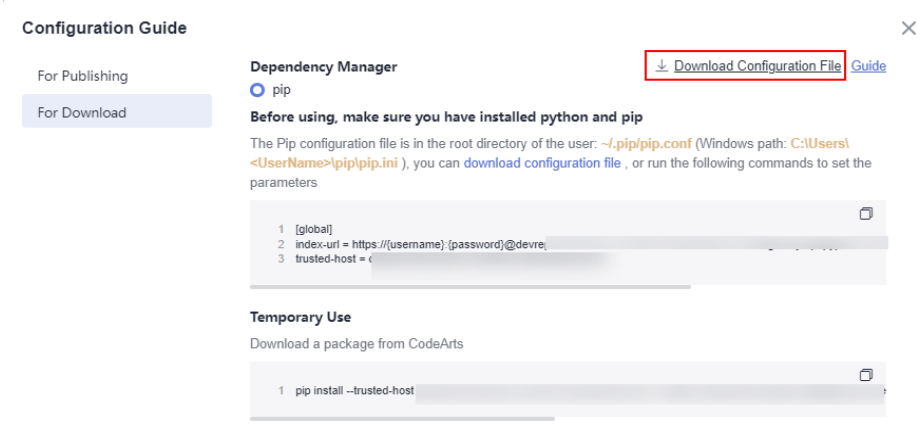
----End

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**.

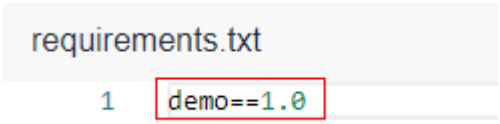


3. Save the downloaded **pip.ini** file as a **pip.conf** file.

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.

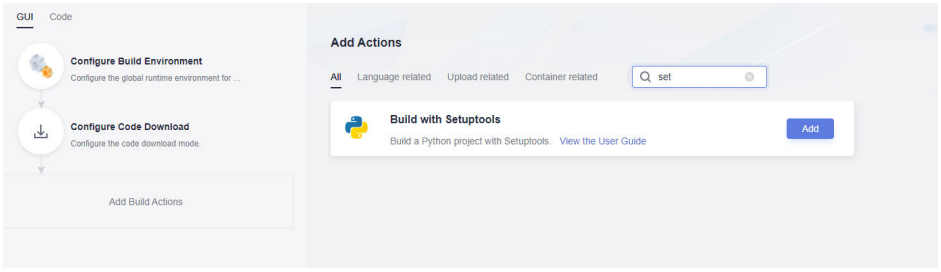
```
demo ==1.0
```



Step 3 Configure and run a build task.

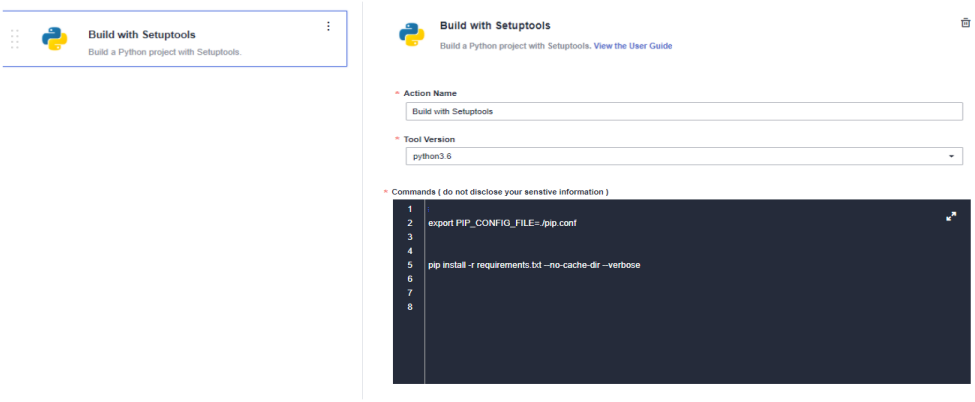
1. On the Repo page, select a repository and click **Create Build Task** in the upper right.

   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
```

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

7 Uploading/Obtaining an RPM Component Using Linux Commands

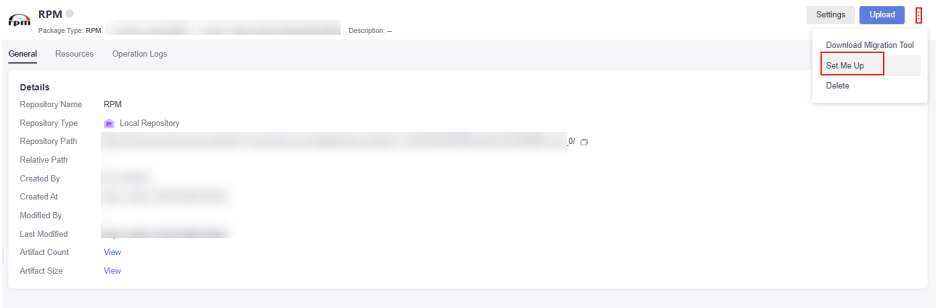
This section describes how to use Linux commands to upload a component to an RPM repository and obtain a dependency from the repository.

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 **Managing Repository Permissions**

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.



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

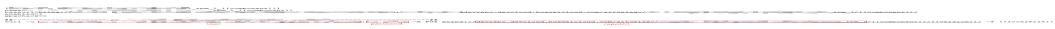
Step 3 On the Linux host, run the following command to upload an RPM component:

```
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.

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

----End

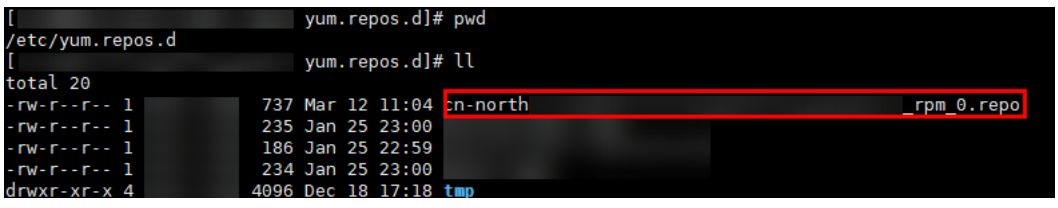
Obtaining a Dependency from an RPM Repository

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.



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

```
yum install hello
```

----End

8 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.

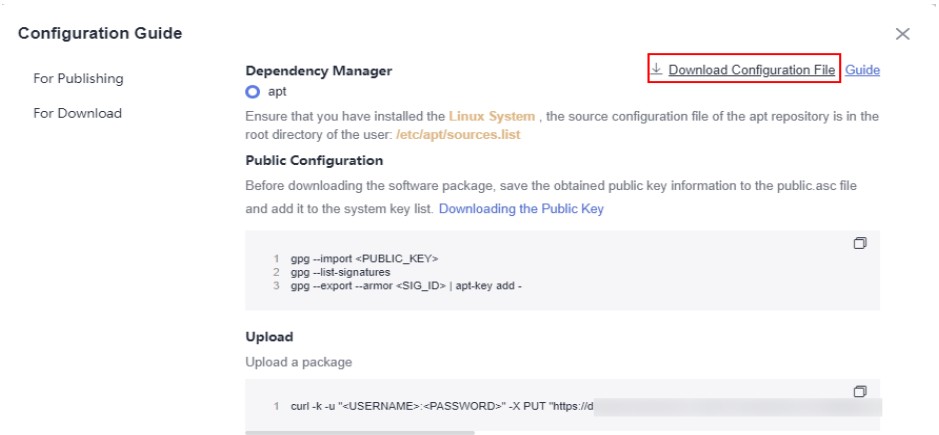
Prerequisites

- 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 **Managing Repository Permissions**

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**.

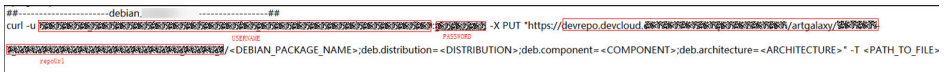


Step 3 On the Linux host, run the following command to upload a Debian component:


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

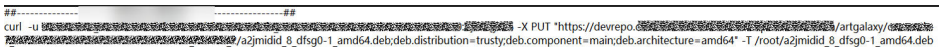


DEBIAN_PACKAGE_NAME, **DISTRIBUTION**, **COMPONENT**, and **ARCHITECTURE** can be obtained from the Debian component.

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.



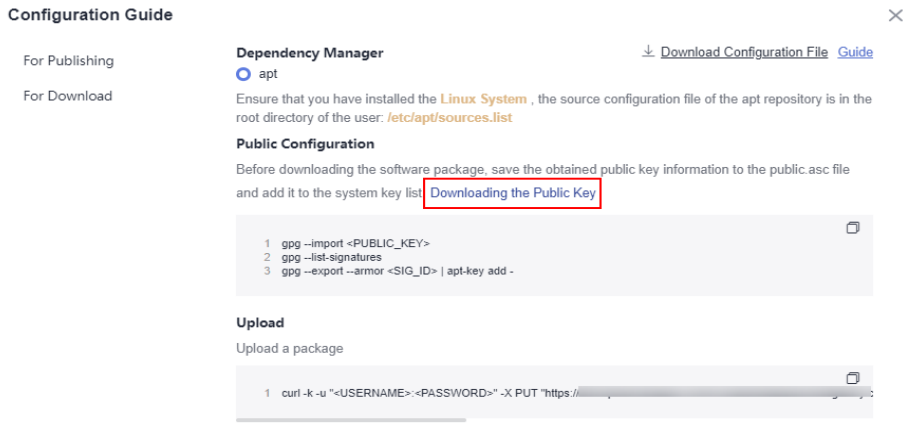
Step 4 After the commands are successfully executed, go to the self-hosted repo and find the uploaded Debian component.

----End

Obtaining a Dependency from a Self-hosted Debian Repo

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.

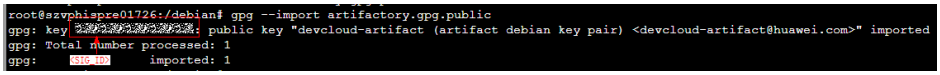
Step 1 Download the **public key** file of the Debian repository by referring to **Releasing a Component to a Debian Repository**.



Step 2 Import the gpg public key.

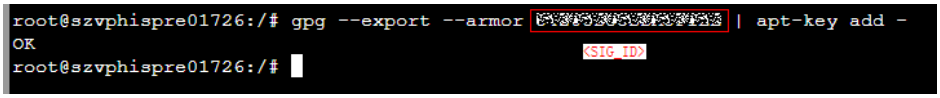
```
gpg --import <PUBLIC_KEY_PATH>
```

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



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 -
```



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
```

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

```
apt download a2jmidid
```


NOTE

Method for obtaining packages:

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

----End

