CodeArts Build

Best Practices

Issue	01
Date	2023-11-15

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 Graphical Build

1.1 Using Maven to Create a Docker Image

1.1.1 Background

CodeArts Build provides a large number of build actions and templates, and implements out-of-the-box build experience through cache, the self-hosted repo, and Huawei Mirrors. If you are using CodeArts Build for the first time, it may be difficult to get started. Therefore, CodeArts Build provides best practices for you to cope with common complex build scenarios.

This section describes how to use CodeArts Build to build Maven projects, how to use the build package to create a Docker image and push it to SWR, and how to use Huawei Mirrors, the self-hosted repo, and cache in the build process.

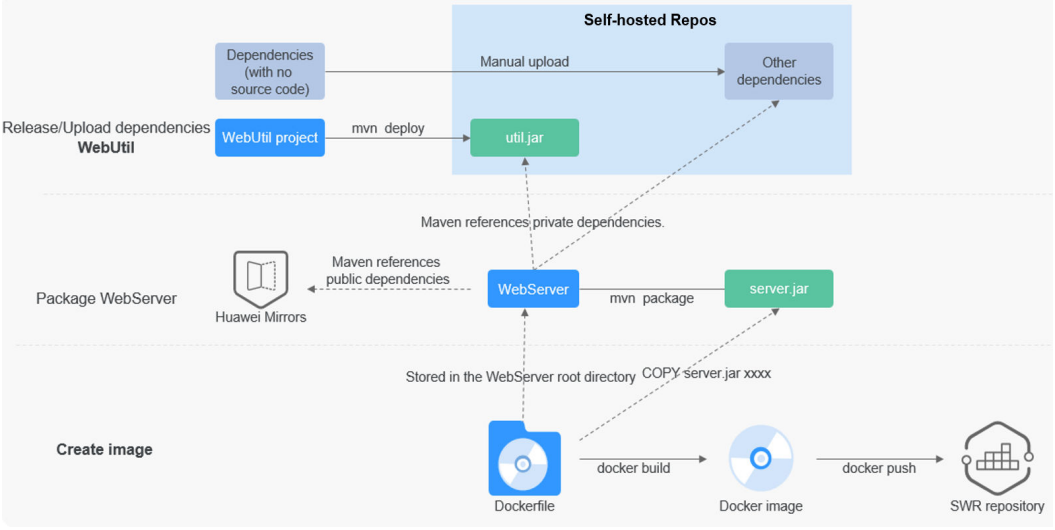
SoftWare Repository for Container (SWR) stores Docker images uploaded by users. These images can be used in build, deployment, and other scenarios.

1.1.2 Description

This section involves the building of two Maven projects, one base Docker image, and one Dockerfile.

- WebServer project: project to be built. A build package will be obtained and used to create a Docker image. The Dockerfile used for creating the image will be stored in the root directory of this project.
- WebUtil project: a self-developed tool package required by WebServer. It is introduced in the pom file of the WebServer project and is used to demonstrate the applicable scenario of the self-hosted repo.
- Base image: You can create a Docker image by adding the WebServer build package to the base image.
- Dockerfile: used to create images.

The following figure shows the build process.



This section describes the entire process from preparing the code repository to creating and pushing the image to SWR. The procedure is as follows:

- **Preparations**
- **Releasing Dependencies to the Self-Hosted Repo**
- **Packaging, Creating, and Pushing an Image**
- **Viewing the Build Results**

1.1.3 Preparations

If you are using CodeArts Build for the first time, **create a project** before starting this example.

Preparing a Repository for the WebServer Project

Step 1 Create the **WebServer** directory for storing code and enter the directory.

Step 2 Create the **pom.xml** file in the **WebServer** directory. The file content is as follows:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.xx.demo</groupId>
  <artifactId>server</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>
  <name>server</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>


  <build>
    <pluginManagement>
      <plugins>
        <plugin>
```



```
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.6</version>
        <configuration>
          <archive>
            <manifest>
              <addClasspath>true</addClasspath>
            </manifest>
            <manifestEntries>
              <Main-Class>
                HelloWorld
              </Main-Class>
            </manifestEntries>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
 </build>
</project>
```

Step 3 Create the **src\main\java** directory.

Step 4 Create the **HelloWorld.java** file in the directory created in **Step 3**. The file content is as follows:

```
public class HelloWorld {
 public static void main(String[] args) {
  System.out.println("Hello World!");
 }
}
```

Step 5 On the CodeArts Build homepage, click  in the upper right corner and select **Custom Build Environments**.

Step 6 On the **Custom Build Environments** page, click **CentOS 7-based x86 Base Image** to obtain the Dockerfile corresponding to the base image.

In this example, CentOS is used as the base image.

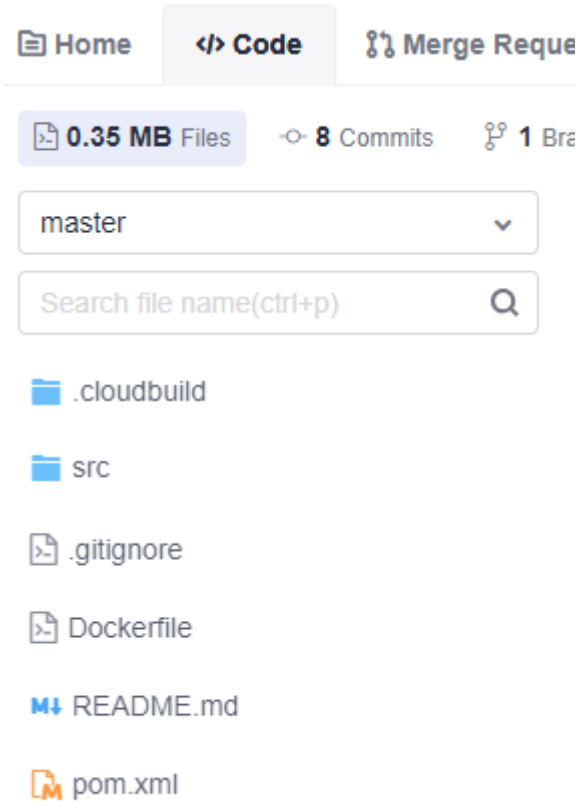
Step 7 View the coordinate definition of the build package in the **pom.xml** file of the WebServer project in **Step 2**.

The Maven build package name is <*artifactId*>-<*version*>.packaging. By default, the build package is generated in the **./target** directory. The final build package path is **./target/server-1.0.jar**.

Step 8 Use the build package path obtained in **Step 6** to compile the Dockerfile. The content is as follows:

```
FROM centos
MAINTAINER <devcloud@demo.com>
USER root
RUN mkdir /demo
COPY ./target/server-1.0.jar /demo/app.jar
```

Step 9 On the navigation bar, choose **Services** > **Repo**. Create a repository named **WebServer** by referring to **Creating a Repository**, and upload the file created in **Step 2**, **Step 3**, and **Step 7** to the repository by referring to **Uploading Code to a Repository**. After the code is uploaded, you can go to the repository to view the code.



----End

Preparing a Repository for the WebUtil Project

Step 1 Create the **WebUtil** directory for storing code.

Step 2 Create the **pom.xml** file in the directory created in **Step 1**. The file content is as follows:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.xx.demo</groupId>
  <artifactId>util</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>
  <name>util</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>


  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
```



```
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.6</version>
        <configuration>
          <archive>
            <manifest>
              <addClasspath>true</addClasspath>
            </manifest>
            <manifestEntries>
              <Main-Class>
                HelloWorld
              </Main-Class>
            </manifestEntries>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
 </build>
</project>
```

Step 3 Create the **src\main\java** directory.

Step 4 Create the **HelloWorld.java** file in the directory created in **Step 3**. The file content is as follows:

```
public class HelloWorld {
 public static void main(String[] args) {
  System.out.println("Hello World!");
 }
}
```

Step 5 On the navigation bar, choose **Services** > **Repo**. Create a repository named **WebUtil** by referring to **Creating a Repository**, and upload the file created in **Step 2** and **Step 4** to the repository by referring to **Uploading Code to a Repository**. After the code is uploaded, you can go to the repository to view the code.

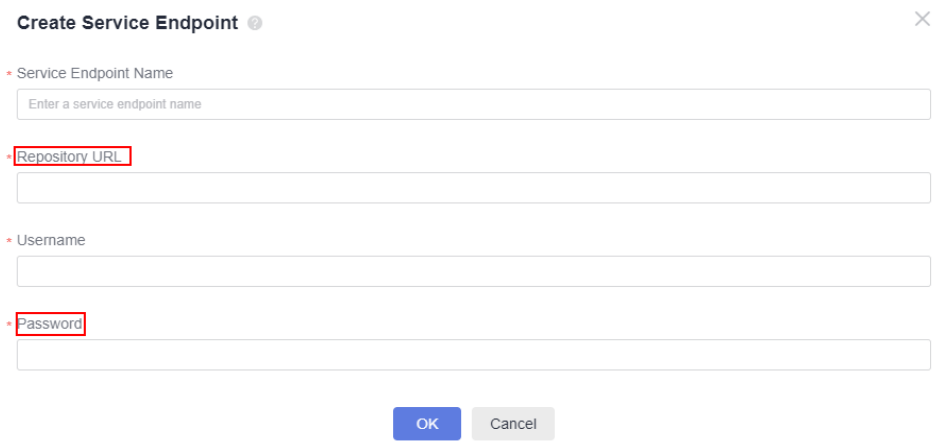
----End

Preparing a Self-hosted Repo Endpoint

Step 1 In the navigation pane, choose **Settings** > **General** > **Service Endpoints**.

Step 2 Click **Create Endpoint** and select **nexus repository**.

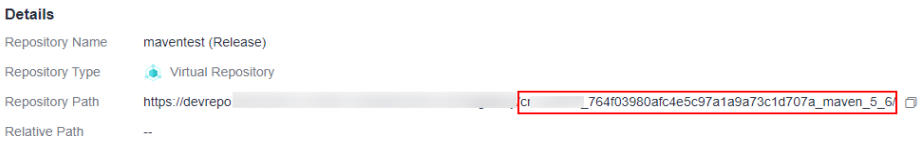
Step 3 In the displayed dialog box, enter the required parameters.



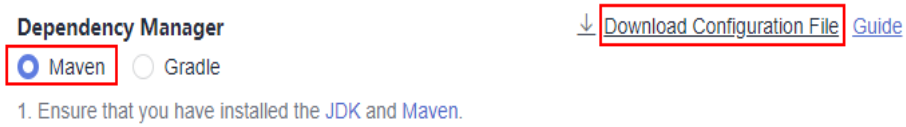
- **Service Endpoint Name**: Use the displayed self-hosted repo endpoint name.
- **Repository URL**: the address of the self-hosted repo.
- **Username**: the username in the configuration file downloaded from the self-hosted repo.
- **Password**: the password in the configuration file downloaded from the self-hosted repo.

The following describes how to obtain the parameters.

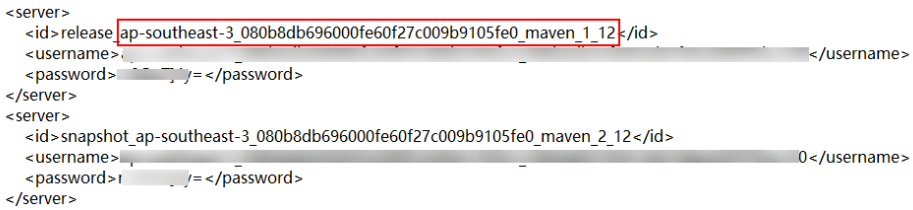
1. In the navigation pane, choose **Artifact** > **Self-hosted Repos** and click next to the repository path to obtain the address of the self-hosted repo.



2. Click **Tutorial**.
3. Select **Maven** as the **Dependency Manager** and click **Download Configuration File**.



4. Obtain the username and password for the file.



Use the username and password under a specific ID. The ID refers to **the last parameter of the repository address**.

Step 4 When creating an image and pushing it to the SWR repository, go to **the SWR console**, **create an organization**, and specify the SWR organization name.

----End

1.1.4 Releasing Dependencies to Self-Hosted Repos

This section describes how to release required dependencies to a self-hosted repo. Before using CodeArts Build, read the following precautions. Incorrect uses of dependencies may cause build errors.

> **NOTE**
>
> The self-hosted repo and release repo are two different services. Pay attention to the differences and avoid releasing dependencies to the release repo. If you upload a dependency to the release repo, it cannot be downloaded during a build.
>
> - The release repo is used to archive software packages for deployment or other purposes.
> - The self-hosted repo is used to store tool packages for other projects, for example, WebUtil.jar.

In this example, the WebServer uses three dependencies:

- WebUtil: a self-developed common component that is released to the self-hosted repo through CodeArts Build.
- CommonUtil: a partner-provided component that contains JAR packages and **pom** files (the **pom** file of the CommonUtil project cannot be used as the **pom** file of the WebServer project). You can manually upload the package in POM mode.
- MessageSDK: a third-party component that contains only JAR packages. You need to consider whether the **pom** file can be obtained in other ways or whether the file can be uploaded in GAV mode.

Preparations

If you are using CodeArts for the first time, access the self-hosted repo homepage to initialize the self-hosted repos. For details, see **Creating a Self-Hosted Repo**.

Releasing WebUtil

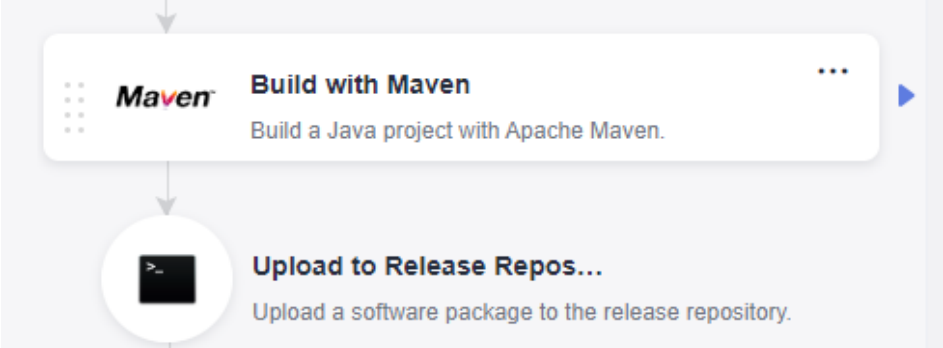
For self-developed tool packages, dependencies need to be compiled and released at a certain frequency. You are advised to use **Build with Maven** provided by CodeArts Build to build and release dependencies to the self-hosted repo. This solution has the following advantages:

- You can release tool packages in one-click during version iteration and avoid repetitive manual operations.
- Automatic continuous integration can be implemented based on functions such as scheduled building and code merging triggering in CodeArts Build and CodeArts Pipeline.
- Maven automatically generates content, which avoids missing or damaged files caused by manual operations and ensures the integrity and validity of the uploaded dependencies.

The configuration procedure is as follows:

Step 1 **Create a build task**. Select the repository **WebUtil** created in **preparations** as the code source, select **Maven** as the build template, and set the task name to **Release WebUtil to the Self-hosted Repo**.

The Maven template is preconfigured with the actions **Build with Maven** and **Upload to Release Repos** and default build commands. In most scenarios, you can directly use the preset actions to build and release the generated software package to the release repos.



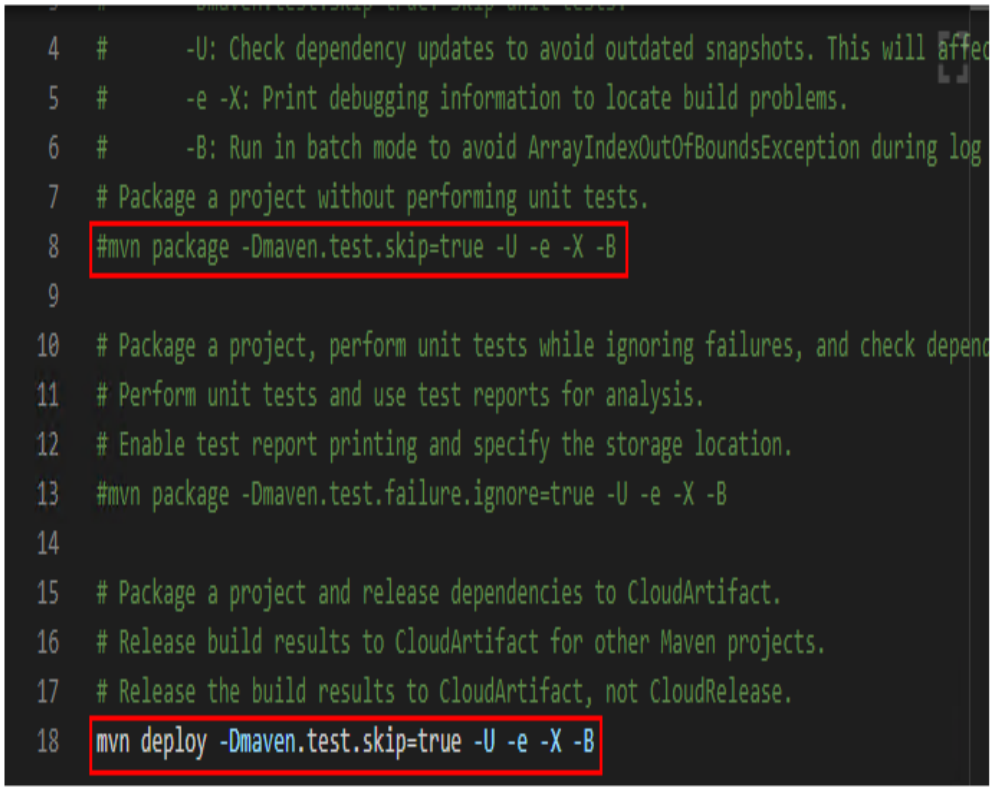
Step 2 Delete the action **Upload to Release Repos**.

NOTE

This section describes how to release the tool package which the project requires to the self-hosted repo. Therefore, the action **Upload to Release Repos** is not needed. If you need to archive the software package to the release repo, retain this step. The build package is automatically generated in the **./target** directory by default.

Step 3 Configure the action **Build with Maven**.

1. Comment out the default **mvn package** command and enable the **mvn deploy** command that has been commented out.



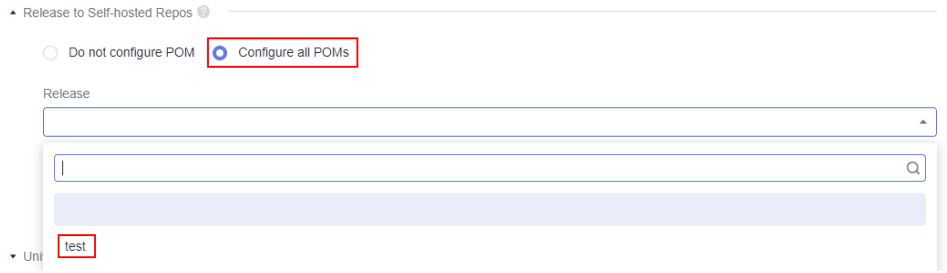
2. Check the build commands. The template provides default parameter settings. Only verify that the parameters are correct.
   - By default, the command reads the **pom** file from the root directory. In this example, the **pom** file of the WebUtil project is in the root directory and requires no changes.

- WebUtil project requires JDK 1.8 for compilation and running. Ensure that the tool version is **maven3.5.3-jdk8-open**.
- This build aims to release a dependency. The default command is **mvn deploy**, which has been enabled.

  NOTE

  - **mvn deploy** command is used to package the project and release it to the specified self-hosted repo for other projects to reference. If the project does not need to be released to the self-hosted repo or the self-hosted repo is faulty, you can run the **mvn install** to package the project and cache it. The project can also be directly referenced by other projects. (The cache is required during the build. The cache content does not ensure data durability. If the cache content is lost, the project needs to be rebuilt.)
  - For details about the default command parameters, see **What Do Default Commands in Build with Maven Indicate?**

3. Choose **Release to Self-hosted Repos** > **Configure all POMs**, and select the self-hosted repo to be released to.



Check cache setting:

- CodeArts Build supports caches to accelerate your build. You can determine whether to use the cache by **Cache Setting**.
- Network jitter, concurrent builds, or other extreme conditions may result in abnormal cache. Consequently, build errors may occur and you need to clear the cache by referring to the **Cache Clearing**.

Step 4 After the task is executed successfully, go to the **Artifact** > **Self-hosted Repos** to view the uploaded dependency.

----End

Manually Releasing CommonUtil in POM Mode

Some dependencies are released through systemPath or manually released to a self-built self-hosted repo using the JAR package provided by a third party (It is the CommonUtil package in this example). These dependencies cannot be downloaded from public repositories. You need to manually upload these dependencies to self-hosted repos in POM mode. The procedure is as follows:

Step 1 Choose **Artifact** > **Self-hosted Repos** and select a repository type according to the type of the dependency to be uploaded.

Step 2 Obtain the **pom** file.

- Method 1: Download the **pom** file from the repository.

  Some dependencies may come from a third-party repository that CodeArts cannot access. You can download the dependencies from the repository. In

this scenario, the private Maven repository provides both JAR packages and **pom** files. You can download the **pom** file from the repository.

- Method 2: Obtain the **pom** file from the JAR package.

  If only the JAR package, not the repository, source code, or **pom** file, can be found for some dependencies, perform the following steps to obtain the **pom** file (the WebUtil package is used as an example):

  a. Decompress the **util-1.0.jar** package. If the package cannot be decompressed, change the file name extension to a supported format.

  b. Enter the directory generated after the decompression, open the **META-INF/maven/{groupid}/{artifactid}** directory (**META-INF/maven/com.xx.demo/util** in this example), open the **pom** file, and verify that the file is correct.

  c. If the **pom** file cannot be found, check whether the file can be uploaded in **GAV mode**.

Step 3 Click **Upload** in the upper right corner, select **POM**, and select the **pom** and **jar** files to upload.

NOTE

Pay attention to the following when manually uploading WebUtil:

The WebServer project depends on the WebUtil project. When uploading the WebUtil project, you must use the **pom** file of the WebUtil project. If the **pom** file of the WebServer project and the JAR package of the WebUtil project are uploaded by mistake, the uploaded dependency coordinates are inconsistent with the expected ones. As a result, the dependency download fails.

----End

Uploading MessageSDK in GAV Mode

You are advised to manually upload the dependency in POM mode. If the **pom** file cannot be found, upload the dependency in GAV mode. However, this mode has potential risks. Evaluate the risks before using this mode.

The following describes the application scenarios and risks of GAV mode:

- When the GAV mode is used, the self-hosted repo automatically generates a **pom** file based on the entered coordinate information. The file content contains only the coordinate definition of the dependency.
- Take WebUtil as an example. If the WebUtil project uses the tool package lib.jar, after WebUtil is uploaded in GAV mode, the lib.jar package cannot be downloaded for the build of WebServer project. As a result, the build package is not as expected.
- If the WebUtil project does not require any dependency (the node of the **pom** file is empty), you can upload the file in this mode.

If you have carefully read the preceding risk description and ensure that the to-be-uploaded dependency does not have the preceding risks or you accept the risks, perform the following steps:

Step 1 Choose **Artifact** > **Self-hosted Repos** and select a repository type according to the type of the dependency to be uploaded.

Step 2 Click **Upload** in the upper right corner, select **GAV**, edit the coordinate information as prompted, select the JAR package, and upload it.

----End

1.1.5 Packaging, Creating, and Pushing an Image

Step 1 **Create a build task**. Select the repository **WebServer** created in **preparations** as the code source, select **Maven** as the build template, and set the task name to **Create a Docker Image Using WebServer**.

The Maven template is preconfigured with the actions **Build with Maven** and **Upload to Release Repos** and default build commands. In most scenarios, you can directly use the preset actions to build and release the generated software package to the release repos.

Step 2 Delete the action **Upload to Release Repos**.

NOTE

This section describes how to pack, create, and push the image on which the project requires to SWR. Therefore, the action **Upload to Release Repos** is not needed. If you need to archive the software package to the release repo, retain this step. The build package is automatically generated in the **./target** directory by default.

Step 3 Configure the action **Build with Maven** and ensure that the build commands and cache configuration are correct.

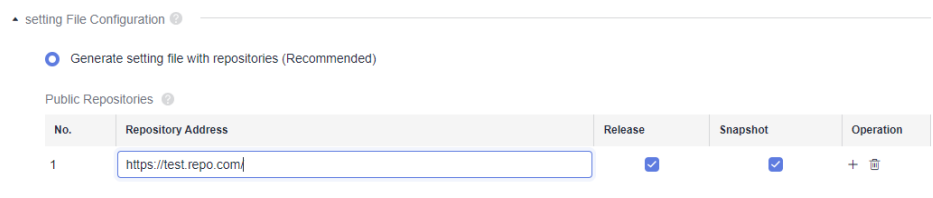
1. Check the build commands: The template provides default parameter settings. Only verify that the parameters are correct.
   - By default, the command reads the **pom** file from the root directory. In this example, the **pom** file of the WebServer project is in the root directory and requires no changes.
   - WebServer requires JDK 1.8 for compilation and running. Ensure that the tool version is **maven3.5.3-jdk8-open**.
   - The target of this build is packaging. The default command is **mvn package**. For details about the default parameters, see **the description of the default commands for Build with Maven**.
2. Check the cache setting.
   - CodeArts Build supports caches to accelerate your build. You can determine whether to use the cache by **Cache Setting**.
   - Network jitter, concurrent builds, or other extreme conditions may result in abnormal cache. Consequently, build errors may occur and you need to clear the cache by referring to the **Cache Clearing**.

> **NOTE**
>
> Huawei Mirrors is automatically configured as the open-source dependency source. Dependencies can be automatically downloaded into CodeArts Build without any additional configuration. The following image sources have been proxied or synchronized on Huawei Mirrors:
>
> - Maven2: https://repo1.maven.org/maven2/
> - Jboss: https://repository.jboss.org/nexus/content/repositories/releases/
> - Jcenter: https://mvnrepository.com/repos/jcenter
> - Grails-core: https://repo.grails.org/grails/core/
> - Grails-plugins: https://repo.grails.org/grails/plugins/
> - Spring-release: https://repo.spring.io/libs-release/
> - Spring-plugins: https://repo.spring.io/plugins-release/

Step 4 Configure the public repository that is not provided by CodeArts.

In this example, the WebServer uses the **lib.jar** file from the third-party repository **https://test.repo.com/**. You need to configure the **lib.jar** file in **Build with Maven**.

The configuration is shown in the following figure.



> **NOTE**
>
> The repository of this type must meet the following requirements:
>
> - The repository address can be directly accessed from the public network (Chinese Mainland).
> - No identity authentication information is required for dependency download.

Step 5 Configure a self-hosted repo.

The **util-1.0.jar** package of the WebUtil project has been released to the self-hosted repo. This section uses this dependency as an example to describe how to use the self-hosted repo in the **Creating a Docker Image Using WebServer** task.

1. Edit the **pom.xml** file in the local WebServer code directory and add the **util-1.0.jar** dependency to the **<dependencies></dependencies>** file.

```
<dependency>
   <groupId>com.xx.demo</groupId>
   <artifactId>util</artifactId>
   <version>1.0</version>
</dependency>
```

2. Save the **pom.xml** file and upload it to the **WebServer** code repository again.
3. In the action **Build with Maven**, select the self-hosted repo endpoint created in the **preparations**.

Step 6 Add the action **Build Image and Push to SWR** after the action **Build with Maven** and enter the required image information.

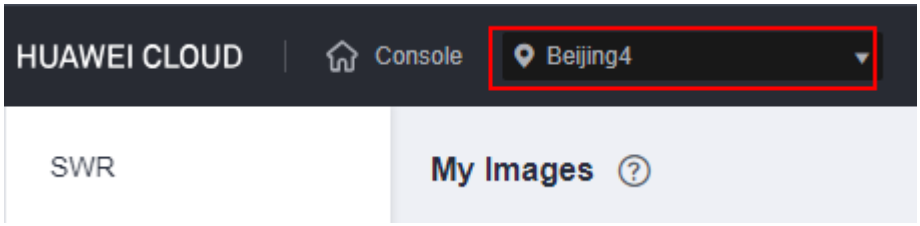
- **Image Repository**: Retain the default value.
- **Organization**: Enter the name of the organization created in **preparations**.
- **Image Name**: Set this parameter to **webserver**.
- **Image Tag**: Set this parameter to **v1.1**.
- **Work Directory**: Retain the default value.
- **Dockerfile Path**: In **preparations**, **Dockerfile** has been stored in the root directory of the WebServer project. The current build directory is the root directory of the project. The default value is **./Dockerfile**.

Step 7 Save and execute the task. After the task is successfully executed, you can **view the build result**.

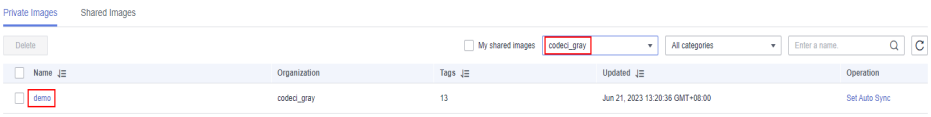
----End

1.1.6 Viewing Build Results

Step 1 Go to the **SWR** console and select the target region.



Step 2 Click **My Images** in the navigation pane, select the organization specified in the **Build Image and Push to SWR** action, and view the built and uploaded image.



----End

1.1.7 Q&A

What Is Cache? How Do I Clear Abnormal Cache?

CodeArts Build allows you to cache the dependencies in your private storage space. Once cached, the dependencies will not have to be pulled for future builds. This greatly accelerates builds.

- Configuring cache setting

  Cache is enabled by default when a build task is created. You can change this setting in **Build with Maven** > **Cache**.

- Clearing the cache

  Network jitter, concurrent builds, or other extreme conditions may result in abnormal cache. Consequently, build errors may occur. The following describes how to clear the abnormal cache.

Before clearing cache, make sure you are well aware of the following precautions:

- The cache directory is shared by multiple users of the same tenant. Frequent cache clearing may cause exceptions (file loss, often the case) in other users' builds. Therefore, clear cache only when the cache is abnormal. After the cache returns to normal, edit the task to delete the clearing command.
- Use a detailed file path when clearing the cache. For example, to clear demo 1.0.0, run **rm -rf /path/com/xx/demo/1.0.0**. If an upper-level directory is entered in the cache clearing command, the next build will be slow or the dependency will be abnormal due to network problems.
- For security purposes, the cache clearing command can be executed only in the **Build with Maven** action. If this command is executed in other actions, the clearing operation may not succeed or an error will be displayed indicating that **the directory does not exist**.

If you have read the risk description carefully and understand and accept the risks, perform the following steps to clear the cache:

a. Prepare the clearing command.

   - The format of the cache clearing command is **rm -rf /repository/local/maven/{*group_ID*}/{*artifact_ID*}/{*version*}**.
   - {*Group_ID*}, {*Artifact_ID*}, and {*Version*} are the **groupId**, **artifactId**, and **version** of the dependency.

     For example, if the dependency is as follows:

     ```
     <dependency>
        <groupId>com.xx.devcloud</groupId>
        <artifactId>demo</artifactId>
        <version>1.0.9-SNAPSHOT</version>
     </dependency>
     ```

     The command for clearing the dependency is **rm -rf /repository/local/maven/com/xx/devcloud/demo/1.0.9-SNAPSHOT**.

b. Edit the build task and configure the action **Build with Maven**.

c. Find the **mvn xxxx** command, add a line before the command, enter the prepared clearing command, and save the task.

d. Run the build task again.

e. Edit the task again and remove the cache clearing command.

What Do Default Commands in Build with Maven Indicate?

The built-in default build commands of the build service are as follows:

```
# Function: packaging
Parameter description:
#           -Dmaven.test.skip=true: Skip unit test.
#           -U: Check for dependency update for each build so that the snapshot version dependency in the cache is always updated. However, this will lower performance.
#           -e -X: Print the debugging information. Use this parameter to locate difficult build problems.
#           -B: Run in batch mode to avoid the ArrayIndexOutOfBoundsException exception during log printing.
# Scenario: Used for packaging projects when unit tests are not required.
mvn package -Dmaven.test.skip=true -U -e -X -B
```

The meaning of each command/parameter is as follows:

- **mvn package**: Use Maven to perform packaging. After this command is executed, a software package is generated in the target directory of the project. You can change the directory as required.
- **-Dmaven.test.skip=true**: Skip the unit test. You are advised to retain this parameter.
- **-U**: Check for dependency update for each build so that the snapshot version dependency in the cache is always updated. However, this will lower performance. You are advised to retain this parameter.
- **-e -X**: Print the debugging information. Use this parameter to locate difficult build problems.
- **-B**: Run in batch mode to avoid the ArrayIndexOutOfBoundsException exception during log printing.

1.2 Using Node.js to Create a Docker Image

Objective

This section helps you use a Node.js build task to create a Docker image on CodeArts Build.

Preparations

You are familiar with the basic concepts, environment installation, and basic operations of Docker. For details on how to install Docker on each OS platform, see **Docker Documentation**.

If you are using CodeArts Build for the first time, **create a project** before starting this example.

Step 1 Create the **nodesource** directory for storing code and enter the directory.

Step 2 Create the **package.json** file in the **nodesource** directory. The content is as follows:

```
{
"name": "docker_web_app",
"version": "1.0.0",
"description": "Node.js on Docker",
"author": "First Last <first.last@example.com>",
"main": "server.js",
"scripts": {
"start": "node server.js"
},
"dependencies": {
"express": "^4.16.1"
}
}
```

Step 3 Create the **server.js** file. The file content is as follows:

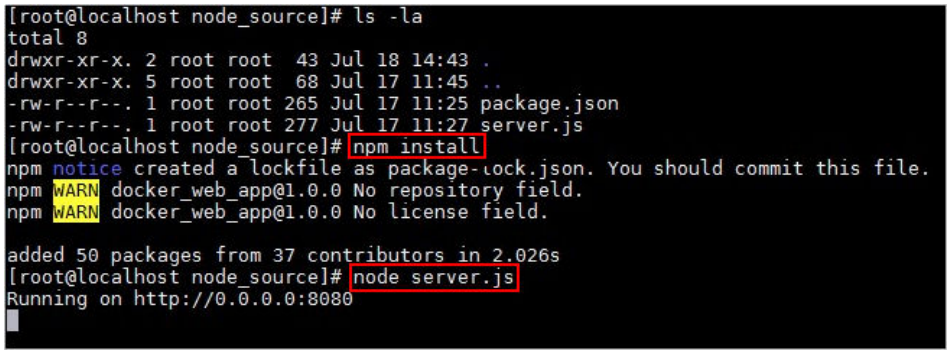
```
'use strict';
const express =require('express');
// Constants
const PORT=8080;
const HOST='0.0.0.0';
// App
const app =express();
app.get('/',(req, res)=>{
res.send('Hello world\n');
```



```
});
app.listen(PORT,HOST);
console.log(`Running on http://${HOST}:${PORT}`);
```

Step 4 Verify the code. After the code is prepared, compile the code locally and run the project to check whether the result is normal.

1. Open a command line tool, run the **cd** command to enter the directory where **nodesource** is located, run the **npm install** command to complete dependency installation, and then run the **node server.js** command to run the project.



2. Open the browser and enter <*local_IP_address*>**:8080** in the address box. If the following information is displayed, the service is running properly.

Step 5 Create a **Dockerfile** in the **nodesource** directory. The content is as follows:

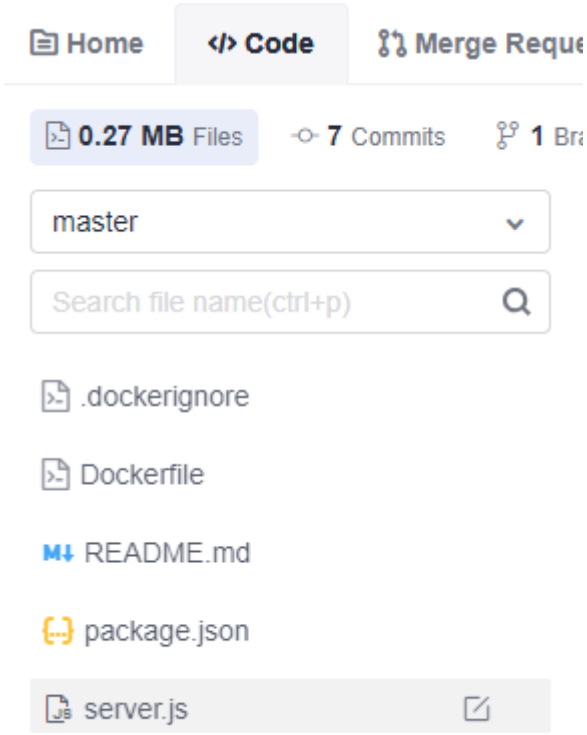
```
#use the latest LTS (long term support) version 12 of node available from the Docker Hub
FROM node:12
# Create app directory
WORKDIR /usr/src/app
# Install app dependencies
# A wildcard is used to ensure both package.json AND package-lock.json are copied
# where available (npm@6+)
COPY package*.json ./
RUN npm install
# If you are building your code for production
# RUN npm ci --only=production
# Bundle app source
COPY . .
EXPOSE 8080
CMD ["node","server.js"]
```

Step 6 Create a **.dockerignore** file with the following content:

```
node_modules
npm-debug.log
```

Step 7 Upload the code to CodeArts Repo.

On the navigation bar, choose **Services** > **Repo**. Create a repository named **nodesource** by referring to **Creating a Repository**, and upload code to the repository by referring to **Uploading Code to a Repository**. After the code is uploaded, you can go to the repository to view the code.



----End

Packaging, Creating, and Pushing an Image

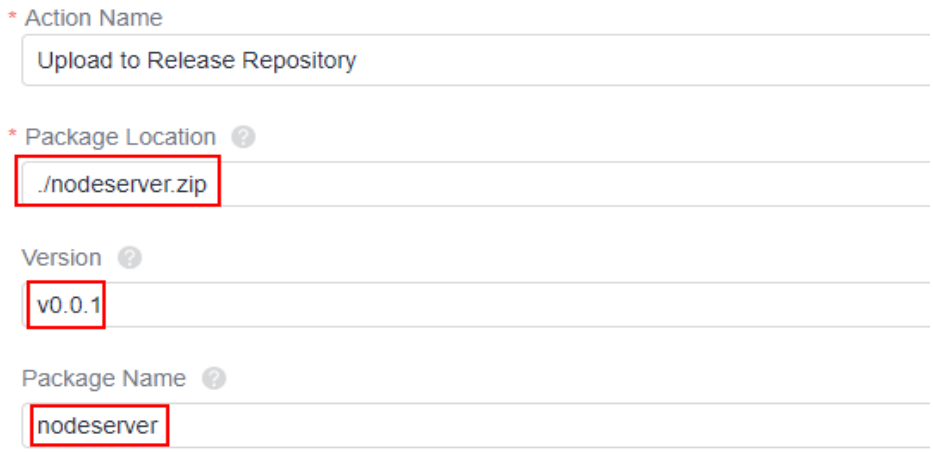
Step 1 Create a build task. Select the repository **nodesource** created in Preparations as the code source, select **npm** as the build template, and set the task name to **nodesource-build**.

Step 2 Configure build actions.

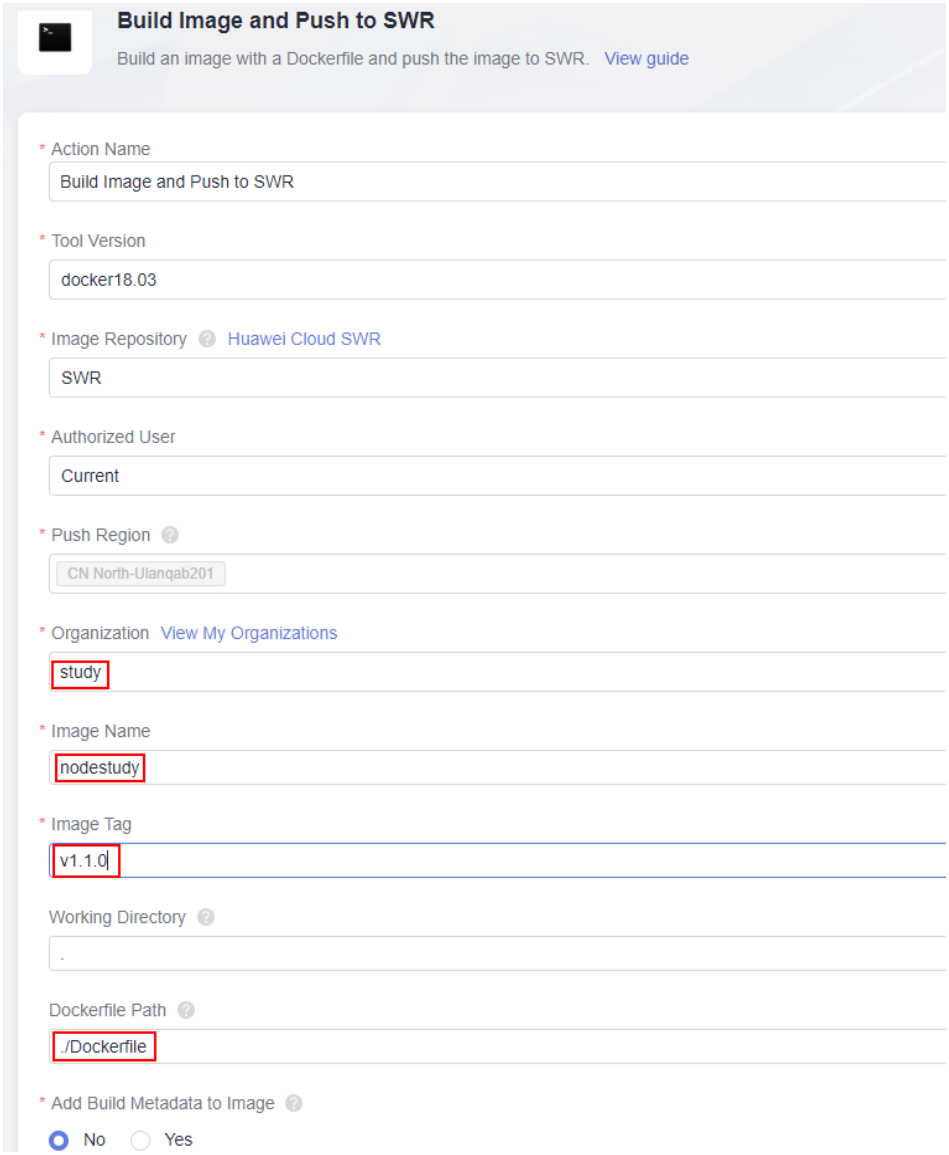
1. Configure the action **Build with npm**. For **Commands**, comment out **npm run build** and enter **zip -r ./nodeserver.zip ./** to pack the code into **nodeserver.zip**.


```
export PATH=$PATH:/root/.npm-global/bin
# Set the cache directory.
npm config set cache /npmcache
npm config set prefix '~/.npm-global'
# To install node-sass, run this command:
#npm install node-sass --verbose
# Load dependencies.
npm install --verbose
# Build projects with default settings.
#npm run build
zip -r ./nodeserver.zip ./
#tar -zcvf demo.tar.gz ./*
```

2. Configure the action **Upload to Release Repos**, with the parameters specified as shown in the figure below.
   - **Package Location**: directory of the software package to be uploaded.
   - **Version**: version of the software package.
   - **Package Name**: name of the software package.

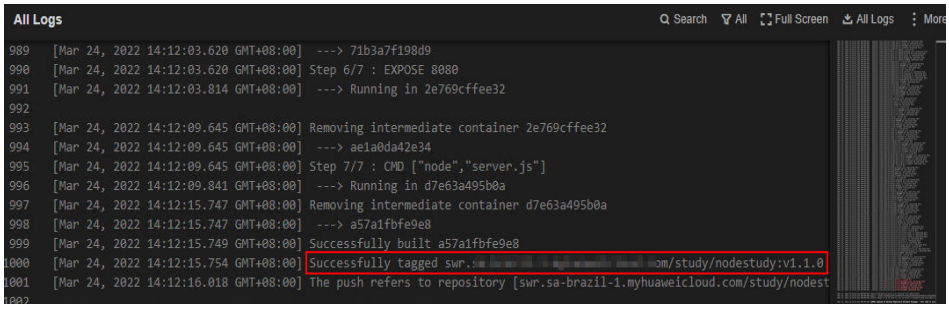


3. Add **Build Image and Push to SWR** action next to the action **Upload to Release Repos**. Specify the parameters as shown in the figure below.



Step 3 After configuring all build actions, click **Create and Run** to run build task.

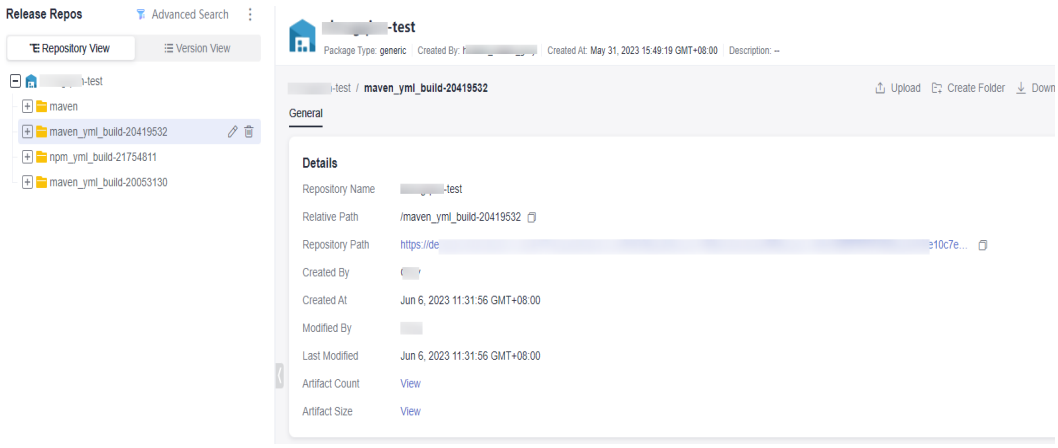
After the task is successfully run, you can view the new image address in the log.



----End

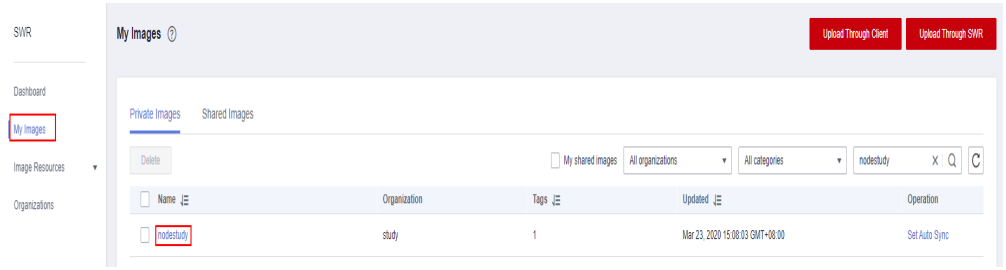
Viewing and Verifying the Build Result

Step 1 In the navigation pane, choose **Artifact** > **Release Repos** to view the uploaded software package whose name is the same as the build task name.

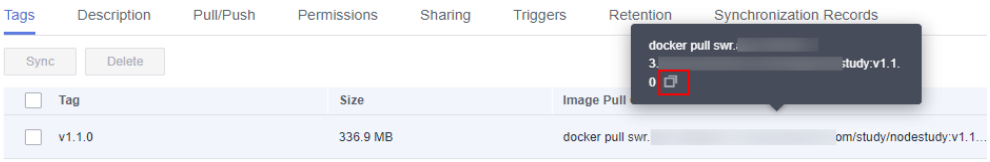


Step 2 Obtain an image pull command and set the image as a public image.

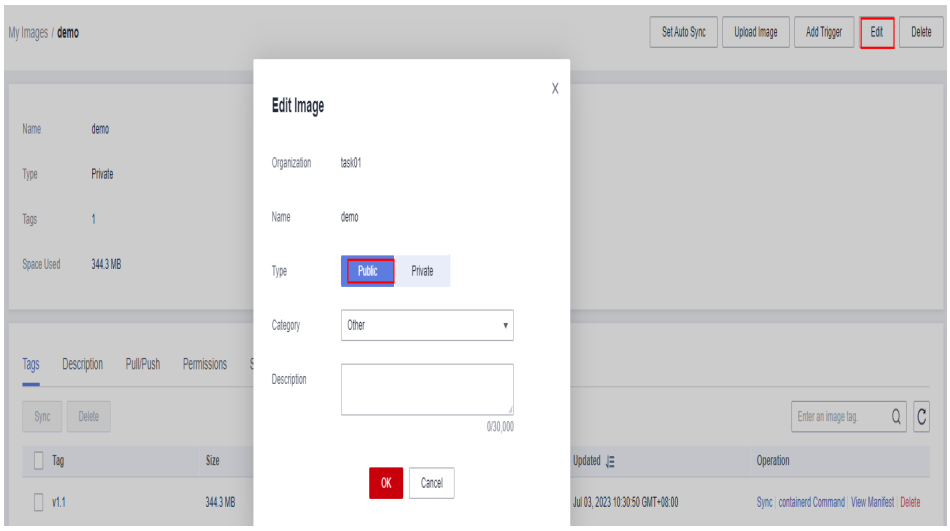
1. On the **SWR** console, click **My Images** in the navigation pane on the left, and click the image **nodestudy**.



2. On the image details page, click in the **Image Pull Command** column to obtain the image pull command (*docker pull* + *image address*).



3. Select image tag **v1.1.0** and click **Edit** in the upper right corner. In the displayed dialog box, select **Public** and click **OK**.



Step 3 Verify the image.

1. Find a host where Docker is installed and enter the image pull command obtained in the previous step in the CLI.

```
[root@localhost nodejs]# docker pull swr.                    m/study/nodestudy:v1.1.0
v1.1.0: Pulling from study/nodestudy
a4d8138d0f6b: Already exists
dbdc36973392: Already exists
f59d6d019dd5: Already exists
aaef3e026258: Already exists
6e454d3b6c28: Already exists
c717a7c205aa: Already exists
e8566b4564fe: Already exists
04239395be03: Already exists
27ace7d95321: Already exists
ad27193056cd: Pull complete
98878eca7158: Pull complete
6341743603fc: Pull complete
423da2eb0660: Pull complete
Digest: sha256:355cb860206f1489587a1061967f5e1a371d76a227b67e11bef10526ce52bfc1
Status: Downloaded newer image for swr.c                     m/study/nodestudy:v1.1.0
```

2. Run **docker run -p 28080:8080 -d** *image_address* to run the image.

   In the preceding information, **-p 28080:8080** indicates that port 8080 in the image is mapped to port 28080 on the local host.

```
[root@localhost nodejs]# docker run -p 28080:8080 -d swr.c                    m/study/nodestudy:v1.1.0
d01dfc340d036841205fa1779160154168f8b457f699538e56ba4cd212adbaf8
[root@localhost nodejs]#
```

3. Enter **http://ip:port** in the address box of the browser. If the following page is displayed, the image is created successfully.

   ip indicates the IP address of the host.

----End

1.3 Using Dockerfile to Create a Docker Image

Background

CodeArts Build provides a large number of build actions and templates. If required dependency packages and tools are not included, you can customize Docker images, manually compile the Dockerfile to add dependencies and tools.

- Dockerfile is a script consisting of instructions and arguments. It begins with a **FROM** instruction and is followed by various methods, instructions, and arguments. You can apply these instructions to the base image and create an image. These instructions simplify the E2E process, including deployment. For more information, visit the Docker official website.
- CodeArts Build provides x86 and Arm base images based on CentOS 7 (containing various common tools) and Ubuntu 18 (containing various common tools). You can create a Dockerfile based on a base image.

This section uses a Maven build as an example to describe how to use Dockerfile to customize a simple container image and push it to SoftWare Repository for Container (SWR).

Preparations

- **Organization**

  When creating an image and pushing it to SWR, specify the SWR organization name. **Create an organization**. For details about organization restrictions, see **Notes and Constraints**.

- **Project code**

  Use the system template Java Maven Demo to create a code repository. For details, see **Creating a Repository Using a Template**.

- **Dockerfile**

  Create a Dockerfile based on the base image. In this example, CentOS 7 is used as the base image.

  a. On the CodeArts Build homepage, click  in the upper right corner and select **Custom Build Environments**.

  b. On the **Custom Build Environments** page, click **CentOS 7-based x86 Base Image** to obtain the Dockerfile corresponding to the base image.

  c. Obtain the build package directory.

     The Maven build package name is <*artifactId*>-<*version*>.packaging. By default, the build package is generated in the **./target** directory.

     i. In the navigation pane, choose **Code** > **Repo**.

     ii. Click the code repository name. The **Code** tab page is displayed.

     iii. Check the coordinate definition in the **pom.xml** file of the code repository. As shown in the following figure, the final build package path is **./target/javaMavenDemo-1.0.jar**.


```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  <modelVersion>4.0.0</modelVersion>
  <groupId>com       .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>
```

d. Use the build package path obtained in **step 3** to compile the Dockerfile downloaded in **step 2**. Add the Maven build package to the base image by running the following command:

```
COPY ./target/javaMavenDemo-1.0.jar /demo/app.jar
```

This command is used to copy the build package to the **demo** directory of the image and name the build package **app.jar**.

e. After completing the Dockerfile, upload the Dockerfile and other files required for image creation to the root directory of the code repository. For details, see **Uploading Local Code to CodeArts Repo**.

Procedure

Step 1 Choose **CICD** > **Build**.

Step 2 Click **Create Task**. On the displayed page, configure the build task information.

Table 1-1 Information setting

Parameter	Description
Task Name	Enter the name of the task.
Project	Select or create a home project for the task.
Code Source	**Repo**: CodeArts Build pulls code from CodeArts Repo. Select a source code repository and branch created in **Preparations**.
Description	Describe the task.

Step 3 Click **Next**. The **Build Template** page is displayed.

Step 4 Select template **Maven** and click **Next**.

Step 5 Add the action **Build Image and Push to SWR**.

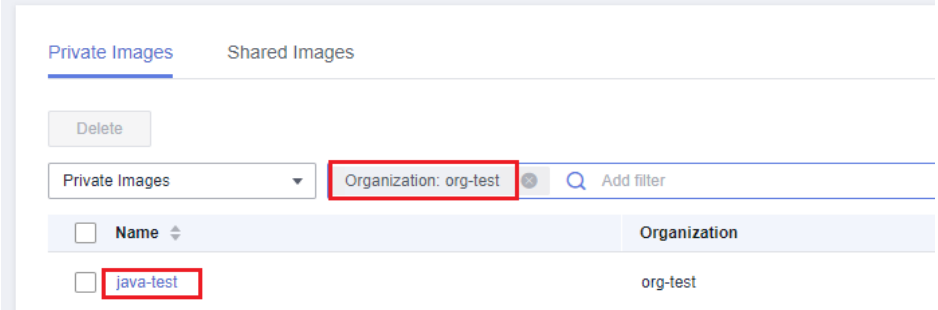
In the **Build with Maven** action, retain default values of the parameters. In the **Build Image and Push to SWR** action, set the parameters as described in the following table.

Parameter	Description
Action Name	Name of a build action. It can be customized.
Tool Version	Select a tool version or use the default one.
Image Repository	By default, CodeArts Build provides the SWR repository address of each Huawei Cloud region. You do not need to change the address. **NOTE** Images can be pushed to custom image repositories.
Authorized User	Current user. Ensure that you have the permissions to edit or manage all images in the organization. For details, see **User Permissions**.
Organization	Enter the SWR organization name created in **Preparations**.
Image Name	Name of the created image. It can be customized.
Image Tag	Specifies the image tag, which can be customized. You can use *Image name*:*Tag* to specify a unique image.
Working Directory	The context path parameter in the docker build command is the relative path to the root directory of the CodeArts Repo repository. When Docker builds an image, the **docker build** command packs all content under the context path and sends it to the container engine to help build the image.
Dockerfile Path	Path of the Dockerfile. Set this parameter to a path relative to the working directory. For example, if the working directory is a root directory and the Dockerfile is in the root directory, set this parameter to **./Dockerfile**.
Add Build Metadata to Image	Add the build information to the image. After the image is created, run the **docker inspect** command to view the image metadata.

Step 6 After configuring the build steps, click **Create** to start the build task.

Step 7 After the command is executed successfully, go to **SWR**.

Step 8 Click **My Images** in the navigation pane, select the organization specified in the **Build Image and Push to SWR** action, and view the built and uploaded image.



----End

2 Code-based Build

2.1 Uploading Software Packages with CMake

Background

CodeArts Build allows you to configure build scripts using YAML files. You can use YAML syntax to compile the build environment, parameters, commands, and steps into a **build.yml** file, which can be stored in a code repository with the built code. The system uses the **build.yml** file as the build script to execute the build task, making the build process traceable, recoverable, secure, and reliable. The build with CMake is used as an example in this section.

Prerequisites

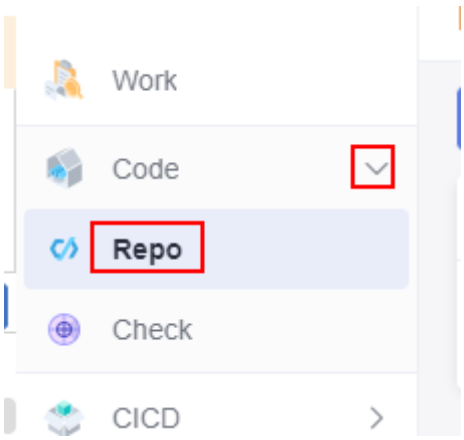
A project is available. If no project is available, **create one**.

Creating a Code Repository

Step 1 Log in to the CodeArts homepage using the Huawei Cloud account.

Step 2 Click the name of the project to create a repository for it.

Step 3 In the navigation pane, choose **Code** > **Repo**.



Step 4 On the CodeArts Repo homepage, click **Create Repository** and select **Common**.

Step 5 Set parameters based on **Table 2-1** and click **OK**.

Table 2-1 Creating a code repository

Parameter	Description
Repository Name	Customize the name for the code repository, for example, **CppDemo_build**. • The name starts with a digit, letter, or underscore (_). • The value can contain periods (.) and hyphens (-). • The value cannot end with **.git**, **.atom**, or a period (.).
Description	Describe the code repository.
.gitignore Programming Language	Select **.gitignore** based on the programming language, for example, Java.
Permissions	Select all. • **Allow project members to access the repository**: A project manager is automatically set as the repository administrator, and a developer is set as a common repository member. When the two roles are added to the project, they will be automatically synchronized to existing repositories. • **Generate README**: You can edit the README file to record information such as the project architecture and compilation purpose, which is similar to a comment on the entire repository. • **Automatically create Check task (free of charge)**: After the repository is created, you can view the code check task of the repository in the check task list.
Visibility	Set this parameter to **Private**. • **Private**: Only repository members can access and commit code. • **Public**: The repository is open and read-only to all guests, but is not displayed in their repository list or search results. You can select an open-source license as the remarks.

----End

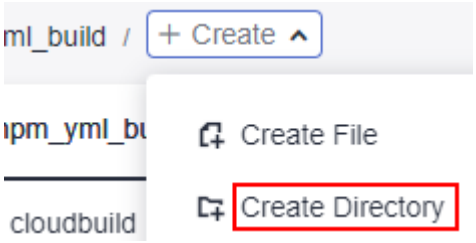
Creating a build.yml File

Step 1 In the navigation pane, choose **Code** > **Repo**.

Step 2 Click the name of the code repository you created (see **Creating a Code Repository**).

Step 3 Choose **Create** > **Create Directory**, as shown in **Figure 2-1**.

Figure 2-1 Creating a directory



Step 4 Set parameters based on **Table 2-2** and click **OK**.

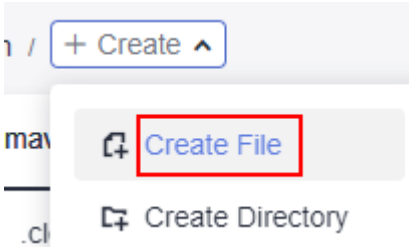
Table 2-2 Creating a directory

Parameter	Description
Directory Name	You can customize the value, for example, **.cloudbuild**.
Commit Message	Enter remarks of the directory. The message is used to record the description of the files in the folder.

Step 5 Click the name of the directory created in **Step 4**.

Step 6 Choose **Create** > **Create File**, as shown in **Figure 2-2**.

Figure 2-2 Creating a file



Step 7 Name the file **build.yml** and copy the following code to the file:

```
# This YAML is the default template and can be modified based on this
---
version: 2.0
steps:
  PRE_BUILD:
  - checkout:
      name: "checkout"
      inputs:
        scm: "codehub"
        url: "git@codehub.devcloud.cn-north-4.huaweicloud.com:cszl00001/cppDemo.git"
        branch: "master"
        lfs: false
        submodule: false
  BUILD:
  - cmake:
      name: "CMake Build"
      inputs:
        command: |
          # Create the build directory and switch to the build directory.
          mkdir build && cd build
          # Generate makefiles for the Unix platform and perform the build.
```



```
        cmake -G 'Unix Makefiles' ../ && make -j
- upload_artifact:
    inputs:
      path: "build/*"
      version: 2.1
      name: packageName
```

Step 8 Click **OK**.

----End

Creating a CMakeLists.txt File

Step 1 Create a file named **CMakeLists.txt** in the root directory by referring to **Step 6** and **Step 7**. The code in the file is as follows:

```
cmake_minimum_required (VERSION 2.5)

project (HolleWorld)
AUX_SOURCE_DIRECTORY(. DIR_SRCS)

add_executable(bin ${DIR_SRCS})
```

Step 2 Click **OK**.

----End

Creating a helloworld.cpp File

Step 1 Create a file named **helloworld.cpp** in the root directory by referring to **Step 6** and **Step 7**. The code in the file is as follows:

```
#include <iostream>
int main()
{
    std::cout << "Hello World !"  << std::endl;
    return 0;
}
```

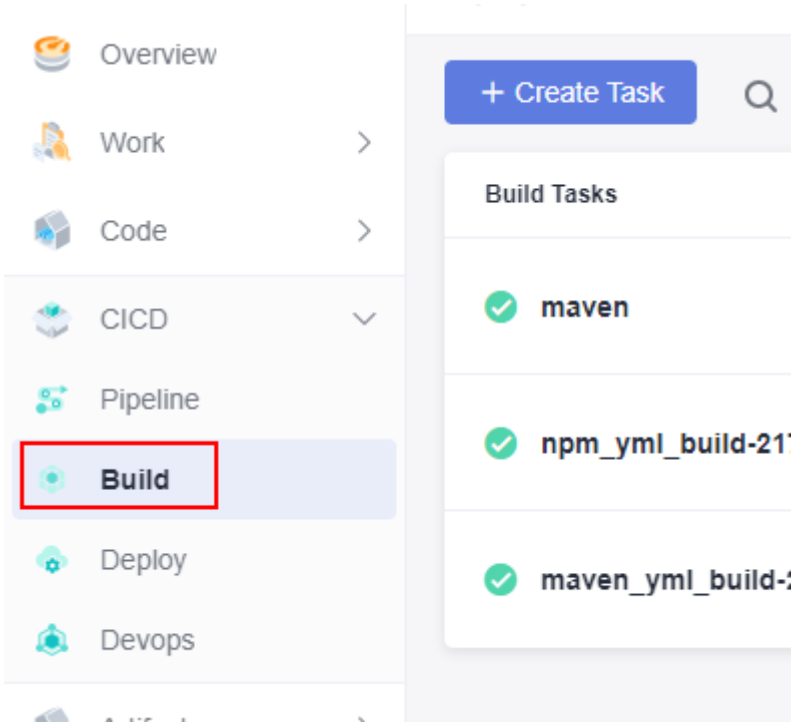
Step 2 Click **OK**.

----End

Creating a Build Task

Step 1 In the navigation pane, choose **CICD** > **Build**, as shown in **Figure 2-3**.

Figure 2-3 Accessing the CodeArts Build homepage



Step 2 Click **Create Task**.

Step 3 Set parameters based on **Table 2-3**.

Table 2-3 Basic information

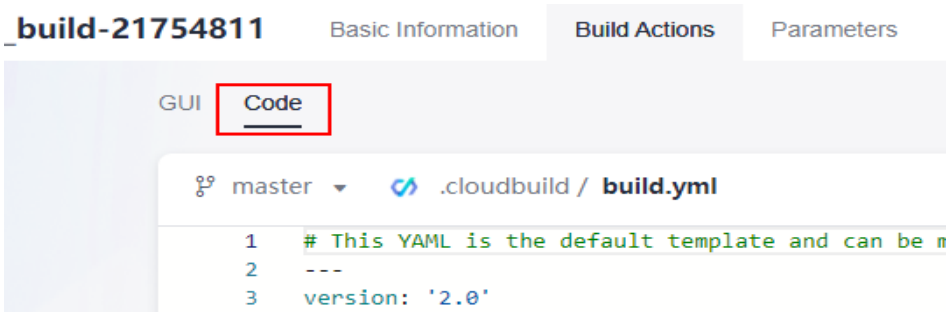
Parameter	Description
Task Name	Enter a custom task name, for example, **CppDemo_build**.
Code Source	Select **Repo**.
Source Code Repository	Select the code repository you created (see **Creating a Code Repository**).
Branch	Select the branch created when you create the repository in **Creating a Code Repository**. If no branch is available, select the default **master**.
Description	Describe the build task.

Step 4 Click **Next**.

Step 5 Select **Blank Template** and click **Next**.

Step 6 Click the **Code** tab to view the imported build script, as shown in **Figure 2-4**.

Figure 2-4 Code tab



Step 7 Click **Create and Run** in the upper right corner.

----End

Viewing and Verifying the Build Result

Step 1 In the navigation pane, choose **Artifact** > **Release Repos**.

Step 2 Go to the release repos to view the released software package. The software package name is the same as the task name when you **create a build task**.

----End

2.2 Uploading Software Packages with Maven

When to Use

CodeArts Build allows you to configure build scripts using YAML files. You can use YAML syntax to compile the build environment, parameters, commands, and steps into a **build.yml** file, which can be stored in a code repository with the built code. The system uses the **build.yml** file as the build script to execute the build task, making the build process traceable, recoverable, secure, and reliable. The build with Maven is used as an example in this section.

Prerequisites

A project is available. If no project is available, **create one**.

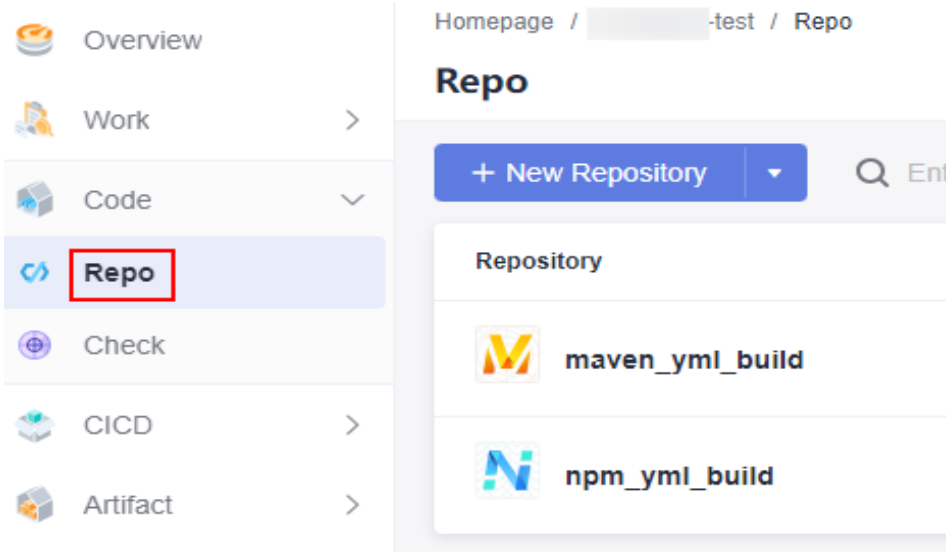
Creating a Code Repository

Step 1 Log in to the CodeArts homepage using the Huawei Cloud account.

Step 2 Click the name of the project to create a repository for it.

Step 3 In the navigation pane, choose **Code** > **Repo**, as shown in **Figure 2-5**.

Figure 2-5 CodeArts Repo



Step 4 Click **New Repository**.

Step 5 Set parameters based on **Table 2-4** and click **OK**.

Table 2-4 Creating a code repository

Parameter	Description
Repository Name	Customize the name for the code repository, for example, **maven_yml_build**. • The name starts with a digit, letter, or underscore (_). • The value can contain periods (.) and hyphens (-). • The value cannot end with **.git**, **.atom**, or a period (.).
Description	Describe the code repository.
.gitignore Programming Language	Select **.gitignore** based on the programming language, for example, Java.
Permissions	Select all. • **Allow project members to access the repository**: A project manager is automatically set as the repository administrator, and a developer is set as a common repository member. When the two roles are added to the project, they will be automatically synchronized to existing repositories. • **Generate README**: You can edit the README file to record information such as the project architecture and compilation purpose, which is similar to a comment on the entire repository. • **Automatically create Check task (free of charge)**: After the repository is created, you can view the code check task of the repository in the check task list.

Parameter	Description
Visibility	Set this parameter to **Private**. • **Private**: Only repository members can access and commit code. • **Public**: The repository is open and read-only to all guests, but is not displayed in their repository list or search results. You can select an open-source license as the remarks.

----End

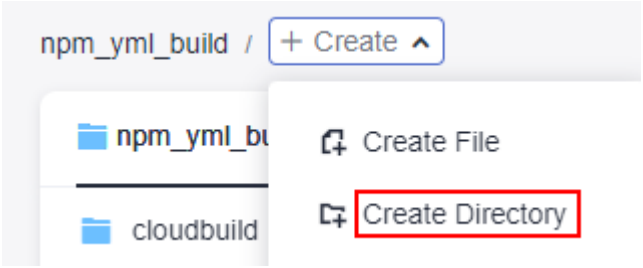
Creating a build.yml File

Step 1 In the navigation pane, choose **Code** > **Repo**.

Step 2 Click the name of the code repository you created (see **Creating a Code Repository**).

Step 3 Choose **Create** > **Create Directory**, as shown in **Figure 2-6**.

Figure 2-6 Creating a directory



Step 4 Set parameters based on **Table 2-5** and click **OK**.

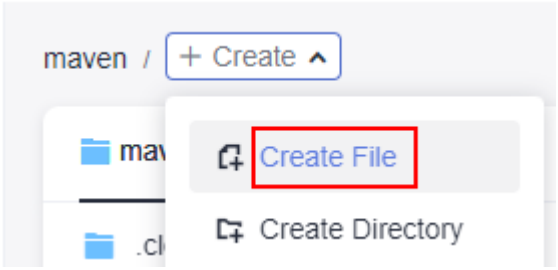
Table 2-5 Creating a directory

Parameter	Description
Directory Name	You can customize the value, for example, **.cloudbuild**.
Commit Message	Enter remarks of the directory. The message is used to record the description of the files in the folder.

Step 5 Click the name of the directory created in **Step 4**.

Step 6 Choose **Create** > **Create File**, as shown in **Figure 2-7**.

Figure 2-7 Creating a file



Step 7 Name the file **build.yml** and copy the following code to the file:

```
# This YAML is the default template and can be modified based on this
---
version: 2.0
steps:
  BUILD:
  - maven:
      image: cloudbuild@maven3.5.3-jdk8-open # You can customize the image path.
      inputs:
        settings:
          public_repos:
            - https://mirrors.huawei.com/maven
        cache: true # Indicates whether to enable the cache.
        command: mvn package -Dmaven.test.failure.ignore=true -U -e -X -B
  - upload_artifact:
      inputs:
        path: "**/target/*.?ar"
  - build_image:
      inputs:
        organization: codeci_gray # Organization name
        image_name: maven_demo # Image name
        image_tag: 1.0 # Image tag
        dockerfile_path: ./Dockerfile
```

Step 8 Click **OK**.

----End

Creating a Java File

Step 1 Create a directory named **src/main/java** by referring to **Step 4**.

Step 2 Create the **HelloWorld.java** file in the **src/main/java** directory by referring to **Step 6** and **Step 7**. The code in the file is as follows:

```
/**
 * Hello world
 *
 */

public class HelloWorld {

  public static void main(String[] args) {
    System.out.println("Hello World!");
  }

}
```

Step 3 Click **OK**.

----End

Creating a Dockerfile

Step 1 Create a file named **Dockerfile** in the root directory by referring to **Step 6** and **Step 7**. The code in the file is as follows:

```
FROM swr.cn-north-5.myhuaweicloud.com/codeci/special_base_image:centos7-base-1.0.2-in
MAINTAINER <devcloud@demo.com>
USER root
RUN mkdir /demo
COPY ./target/server-1.0.jar /demo/app.jar
```

server-1.0.jar combines the values of **artifactId**, **packaging**, and **version** in the **pom.xml** file.

Step 2 Click **OK**.

----End

Creating a pom.xml File

Step 1 Create a file named **pom.xml** in the root directory by referring to **Step 6** and **Step 7**. The code in the file is as follows:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.huawei.demo</groupId>
  <artifactId>server</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>
  <name>server</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>


  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-jar-plugin</artifactId>
          <version>2.6</version>
          <configuration>
            <archive>
              <manifest>
                <addClasspath>true</addClasspath>
              </manifest>
              <manifestEntries>
                <Main-Class>
                  HelloWorld
                </Main-Class>
              </manifestEntries>
            </archive>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

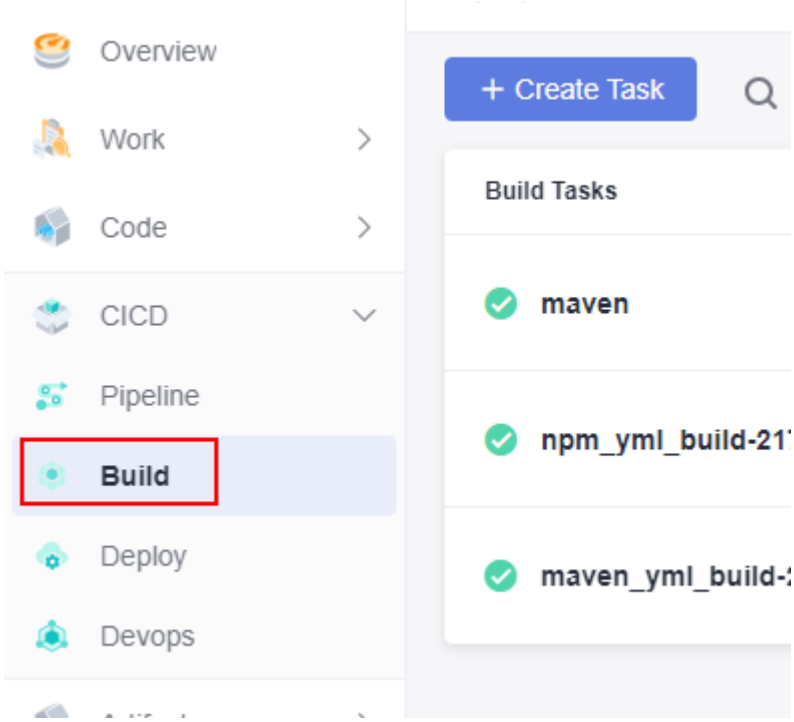

Step 2 Click **OK**.

----End

Creating a Build Task

Step 1 In the navigation pane, choose **CICD** > **Build**, as shown in **Figure 2-8**.

Figure 2-8 CodeArts Build homepage



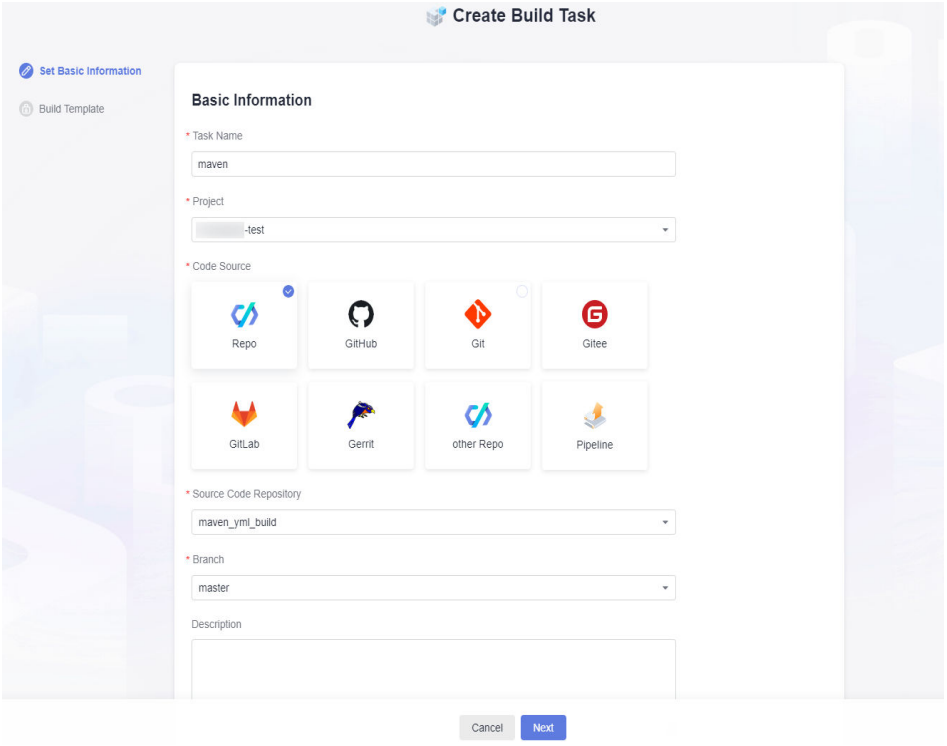
Step 2 Click **Create Task**.

Step 3 Set the parameters based on the **Table 2-6**, as shown in **Figure 2-9**.

Table 2-6 Basic information

Parameter	Description
Task Name	Enter a custom task name, for example, **maven_yml_build**.
Code Source	Select **Repo**.
Source Code Repository	Select the code repository you created (see **Creating a Code Repository**).
Branch	Select the branch created when you create the repository in **Creating a Code Repository**. If no branch is available, select the default **master**.
Description	Describe the build task.

Figure 2-9 Creating a Build Task

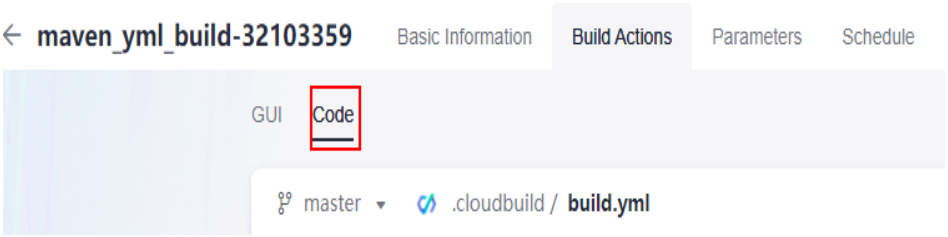


Step 4 Click **Next**.

Step 5 Select **Blank Template** and click **Next**.

Step 6 Click the **Code** tab to view the imported build script, as shown in **Figure 2-10**.

Figure 2-10 Code tab



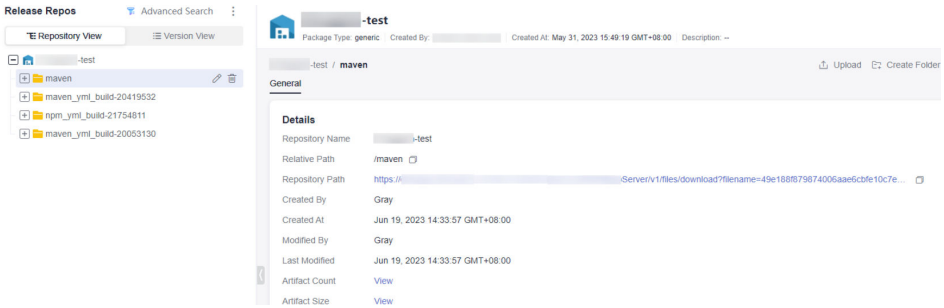
Step 7 Click **Create and Run** in the upper right corner.

----End

Viewing and Verifying the Build Result

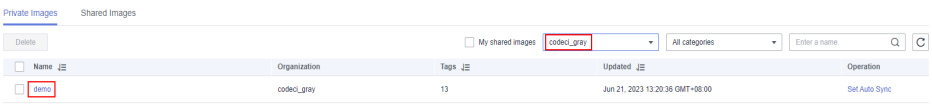
- Verifying the uploaded the software package
  a. In the navigation pane, choose **Artifact** > **Release Repos**.
  b. Go to the release repos to view the released software package. The name of the software package is the same as the task name you use when **Creating a Build Task**, as shown in **Figure 2-11**.

Figure 2-11 Viewing the software package



- Viewing the pushed image
  a. Go to the **SWR** console.
  b. In the navigation pane, choose **My Images**. In the organization filter box, search for the organization name you enter in the code when you **create a build.yml file**, for example, **codeci_gray**.
  c. In the filtering results, click the image name you enter in the code when you **create a build.yml file**, for example, **maven_demo**, as shown in **Figure 2-12**.

Figure 2-12 Filtering images



2.3 Uploading Software Packages with npm

When to Use

CodeArts Build allows you to configure build scripts using YAML files. You can use YAML syntax to compile the build environment, parameters, commands, and steps into a **build.yml** file, which can be stored in a code repository with the built code. The system uses the **build.yml** file as the build script to execute the build task, making the build process traceable, recoverable, secure, and reliable. The build with npm is used as an example in this section.

Prerequisites

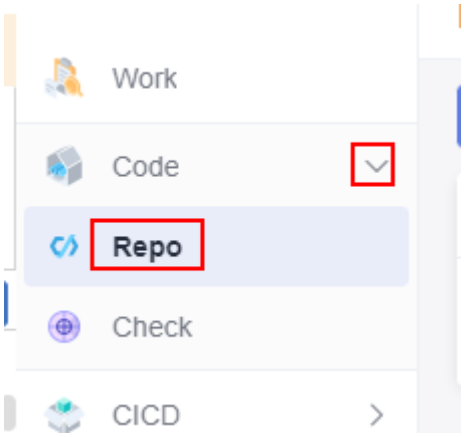
A project is available. If no project is available, **create one**.

Creating a Code Repository

Step 1 Log in to the CodeArts homepage using the Huawei Cloud account.

Step 2 Click the name of the project to create a repository for it.

Step 3 In the navigation pane, choose **Code** > **Repo**.



Step 4 On the CodeArts Repo homepage, click **Create Repository** and select **Common**.

Step 5 Set parameters based on **Table 2-7** and click **OK**.

Table 2-7 Creating a code repository

Parameter	Description
Repository Name	Customize the name for the code repository, for example, **npm_yml_build**. • The name starts with a digit, letter, or underscore (_). • The value can contain periods (.) and hyphens (-). • The value cannot end with **.git**, **.atom**, or a period (.).
Description	Describe the code repository.
.gitignore Programming Language	Select **.gitignore** based on the programming language, for example, Java.
Permissions	Select all. • **Allow project members to access the repository**: A project manager is automatically set as the repository administrator, and a developer is set as a common repository member. When the two roles are added to the project, they will be automatically synchronized to existing repositories. • **Generate README**: You can edit the README file to record information such as the project architecture and compilation purpose, which is similar to a comment on the entire repository. • **Automatically create Check task (free of charge)**: After the repository is created, you can view the code check task of the repository in the check task list.

Parameter	Description
Visibility	Set this parameter to **Private**. • **Private**: Only repository members can access and commit code. • **Public**: The repository is open and read-only to all guests, but is not displayed in their repository list or search results. You can select an open-source license as the remarks.

----End

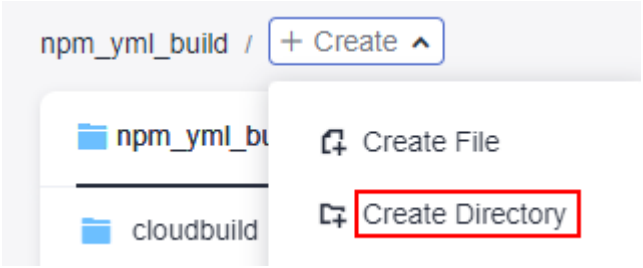
Creating a build.yml File

Step 1 In the navigation pane, choose **Code** > **Repo**.

Step 2 Click the name of the code repository you created (see **Creating a Code Repository**).

Step 3 Choose **Create** > **Create Directory**, as shown in **Figure 2-13**.

Figure 2-13 Creating a directory



Step 4 Set parameters based on **Table 2-8** and click **OK**.

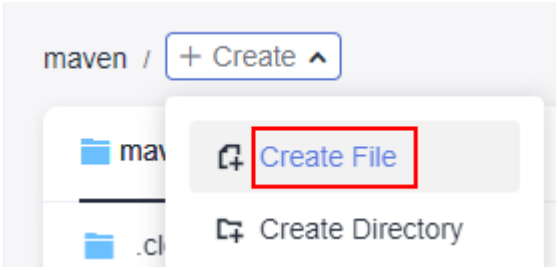
Table 2-8 Creating a directory

Parameter	Description
Directory Name	You can customize the value, for example, **.cloudbuild**.
Commit Message	Enter remarks of the directory. The message is used to record the description of the files in the folder.

Step 5 Click the name of the directory created in **Step 4**.

Step 6 Choose **Create** > **Create File**, as shown in **Figure 2-14**.

Figure 2-14 Creating a file



Step 7 Name the file **build.yml** and copy the following code to the file:

```
# This YAML is the default template and can be modified based on this
---
version: '2.0'
steps:
  BUILD:
    - npm:
        inputs:
          #check:
            #project_dir: .
          command: |
            export PATH=$PATH:~/.npm-global/bin
            #Set the cache directory.
            npm config set cache /npmcache
            npm config set registry http://mirrors.tools.huawei.com/npm/
            npm config set disturl http://mirrors.tools.huawei.com/nodejs
            npm config set sass_binary_site http://mirrors.tools.huawei.com/node-sass/
            npm config set phantomjs_cdnurl http://mirrors.tools.huawei.com/phantomjs
            npm config set chromedriver_cdnurl http://mirrors.tools.huawei.com/chromedriver
            npm config set operadriver_cdnurl http://mirrors.tools.huawei.com/operadriver
            npm config set electron_mirror http://mirrors.tools.huawei.com/electron/
            npm config set python_mirror http://mirrors.tools.huawei.com/python
            npm config set prefix '~/.npm-global'
            npm install --verbose
            zip -r ./nodeserver.zip ./
    - upload_artifact:
        inputs:
          path: "./nodeserver.zip"
```

Step 8 Click **OK**.

----End

Creating a package.json File

Step 1 Create a file named **package.json** in the root directory by referring to **Step 6** and **Step 7**. The code in the file is as follows:

```
{
"name": "docker_web_app",
"version": "1.0.0",
"description": "Node.js on Docker",
"author": "First Last <first.last@example.com>",
"main": "server.js",
"scripts": {
"start": "node server.js"
},
"dependencies": {
"express": "^4.16.1"
}
}
```

Step 2 Click **OK**.

----End

Creating a server.js File

Step 1 Create a file named **server.js** in the root directory by referring to **Step 6** and **Step 7**. The code in the file is as follows:

```
'use strict';
const express =require('express');
// Constants
const PORT=8080;
const HOST='127.0.0.1';
// App
const app =express();
app.get('/',(req, res)=>{
res.send('Hello world\n');
});
app.listen(PORT,HOST);
console.log(`Running on http://${HOST}:${PORT}`);
```

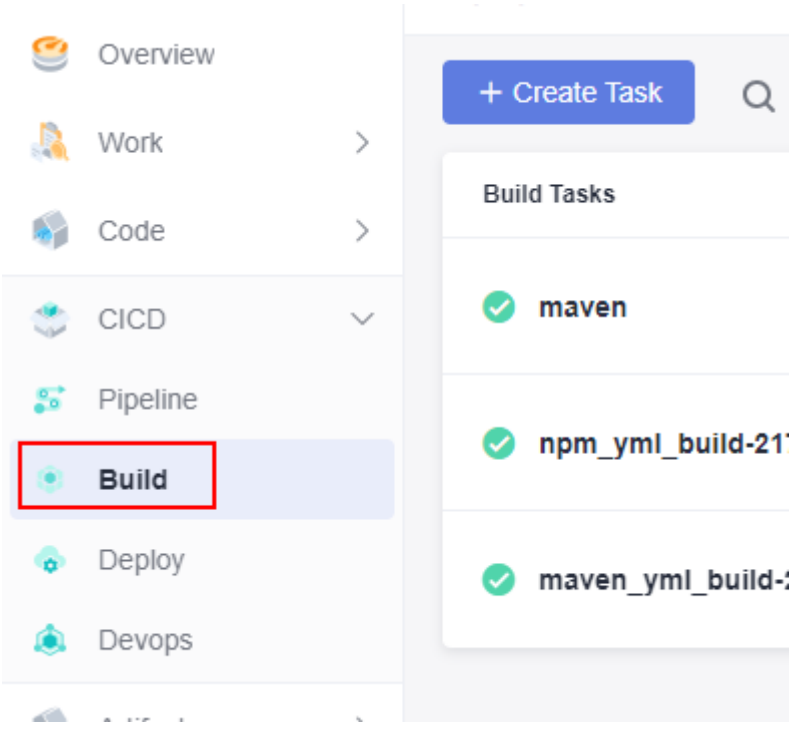
Step 2 Click **OK**.

----End

Creating a Build Task

Step 1 In the navigation pane, choose **CICD** > **Build**, as shown in **Figure 2-15**.

Figure 2-15 Accessing the CodeArts Build homepage



Step 2 Click **Create Task**.

Step 3 Set parameters based on **Table 2-9**.

Table 2-9 Basic information

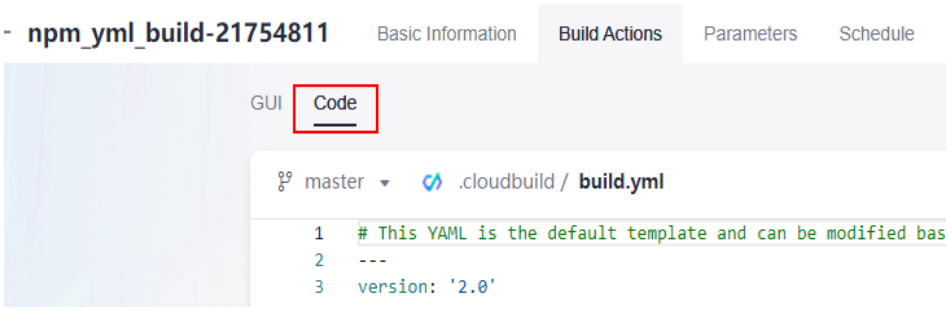
Parameter	Description
Task Name	Enter a custom task name, for example, **npm_yml_build**.
Code Source	Select **Repo**.
Source Code Repository	Select the code repository you created (see **Creating a Code Repository**).
Branch	Select the branch created when you create the repository in **Creating a Code Repository**. If no branch is available, select the default **master**.
Description	Describe the build task.

Step 4 Click **Next**.

Step 5 Select **Blank Template** and click **Next**.

Step 6 Click the **Code** tab to view the imported build script, as shown in **Figure 2-16**.

Figure 2-16 Code tab



Step 7 Click **Create and Run** in the upper right corner.

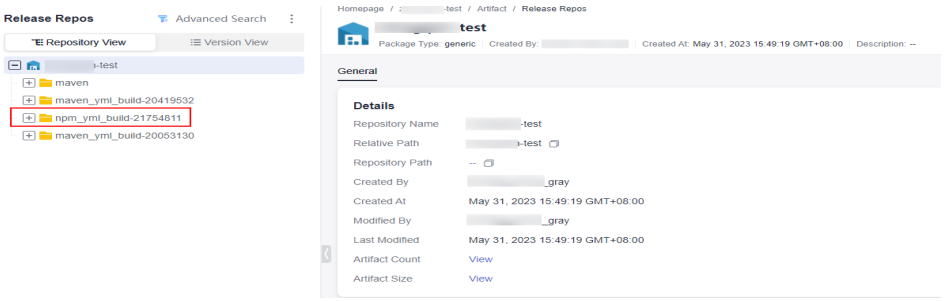
----End

Viewing and Verifying the Build Result

Step 1 In the navigation pane, choose **Artifact** > **Release Repos**.

Step 2 Go to the release repos to view the released software package. The name of the software package is the same as the task name you use when **Creating a Build Task**, as shown in **Figure 2-17**.

Figure 2-17 Viewing the software package



----End

