TEAMCENTER

Hardware and Software Management

Teamcenter 2412

SIEMENS

About Siemens Digital Industries Software

Siemens Digital Industries Software is a global leader in the growing field of product lifecycle management (PLM), manufacturing operations management (MOM), and electronic design automation (EDA) software, hardware, and services. Siemens works with more than 100,000 customers, leading the digitalization of their planning and manufacturing processes. At Siemens Digital Industries Software, we blur the boundaries between industry domains by integrating the virtual and physical, hardware and software, design and manufacturing worlds. With the rapid pace of innovation, digitalization is no longer tomorrow’s idea. We take what the future promises tomorrow and make it real for our customers today. Where today meets tomorrow. Our culture encourages creativity, welcomes fresh thinking and focuses on growth, so our people, our business, and our customers can achieve their full potential.

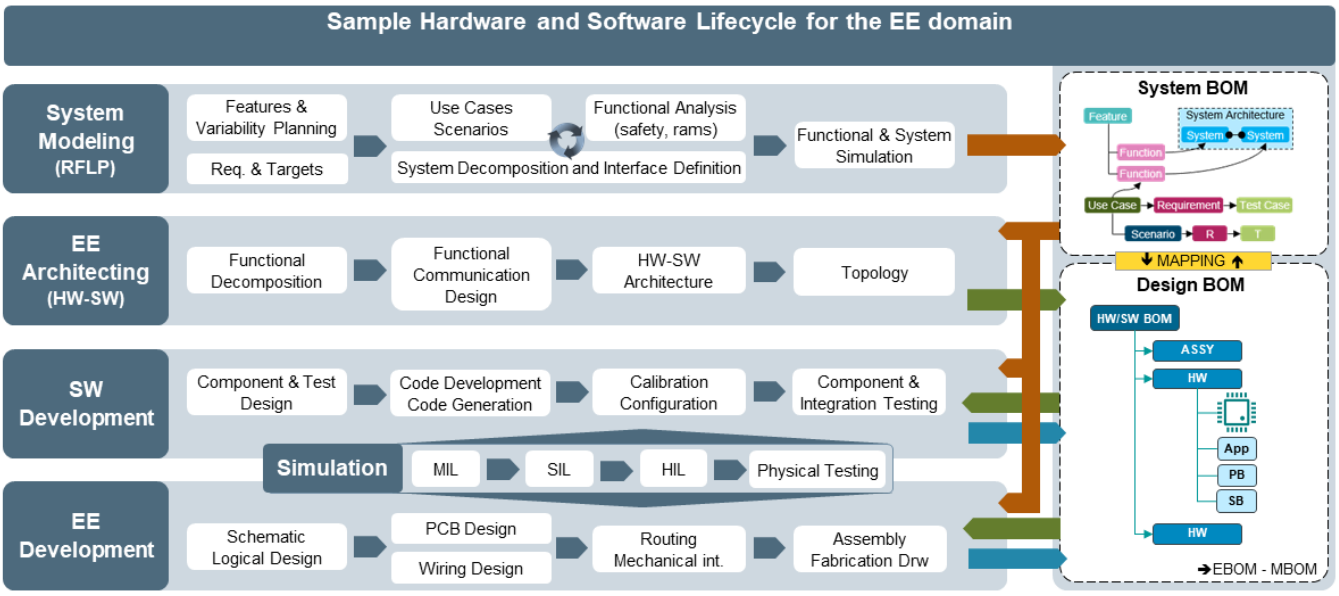
Support Center: **support.sw.siemens.com**

Send Feedback on Documentation: **support.sw.siemens.com/doc_feedback_form**

Contents

1. Understanding hardware and software management

The hardware and software management solution helps you manage the lifecycle of your hardware and software BOM. A typical hardware and software lifecycle in the electrical and electronics (EE) domain is as follows:



- **System modeling phase**

  In this phase, the system model and the system architecture are created.

- **Architecting phase**

  In this phase, the EE architecting tools decompose the system architecture into functions, design, and so on, to finally arrive at the hardware-software architecture and topology. The hardware-software architecture is saved to Teamcenter.

- **Software development phase**

  Once the hardware-software architecture is available in Teamcenter, it is handed off to a software development tool such as Polarion or Jira where code development and testing are done. Once software development is complete, the source code is built in a build tool such as Jenkins and the software deliverables are managed in Teamcenter.

- **EE development phase**

The hardware and software BOM is handed off to EE development tools for further development of the EE system such as PCB design and wiring design.

Where do I go from here?

 Installer	See **Install support for Hardware and Software management**.
 Administrator	
Hand off the software architecture to an ALM tool	See **Handing off software architecture to an ALM tool**.
Publish software deliverables to Teamcenter	**Look at how you can publish a software deliverable to Teamcenter from a publishing tool such as Jenkins**.

2. Install support for Hardware-Software management

Using the MBSE Integration Gateway install kit, you must install server and client features for Software Management.

Install the MBSE Integration Gateway server features for Software Management

You can install MBSE Integration Gateway server features for Hardware-Software management using either Deployment Center or Teamcenter Environment Manager (TEM).

Deployment Center

- In the **Applications** step, select the following component:

  MBSE Integrations > **Software Management**

Teamcenter Environment Manager

- **Extensions** > **MBSE Integrations** > **Server** > **Software Management**

For more information about installing MBSE Integration Gateway server features, see the MBSE Integration Gateway documentation.

Install MBSE Integration Gateway client

You must install MBSE Integration Gateway client on the machine alloted for performing the software management tasks. You can do this either Deployment Center or Teamcenter Environment Manager (TEM).

Deployment Center

- In the **Applications** step, select the following component:

  - **MBSE Integrations** > **Software Management**

Teamcenter Environment Manager

- **Extensions** > **MBSE Integrations** > **Client** > **Software Management**

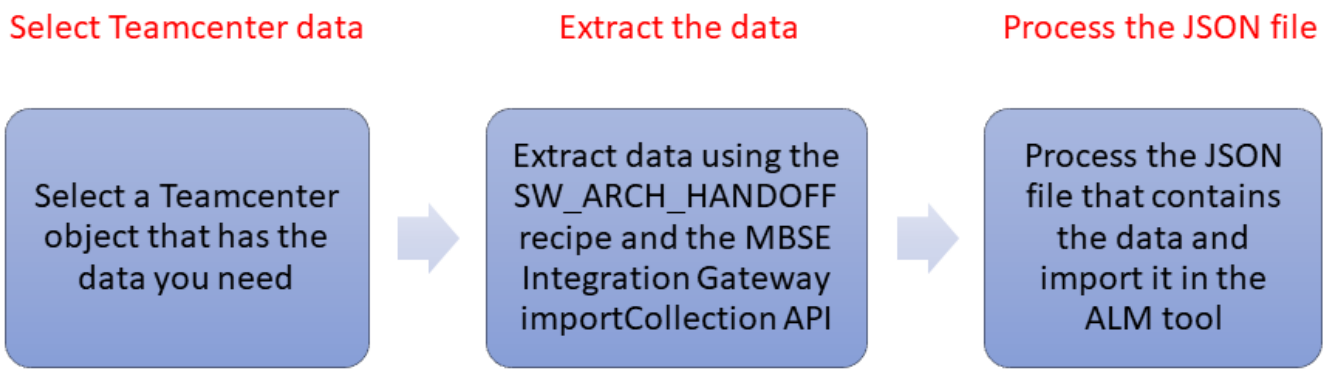
For more information about installing the MBSE Integration Gateway client, see the MBSE Integration Gateway documentation.

3. Preparing a software specification package for handoff to an ALM tool

Overview of preparing a software specification package for handoff to an ALM tool

In the software development phase of the hardware-software lifecycle, the ALM tool needs the data from Teamcenter to carry out software development work such as code development and testing. In Teamcenter, you can prepare a software specification package that consists of the software BOM and other associated data. This package can be used by an ALM tool such as Polarion or Jira.

Preparing a software specification package consists of the following steps:



- Select the data in Teamcenter that you want to handoff to the ALM tool.
- Extract the data from Teamcenter using a recipe. A recipe called **SW_ARCH_HANDOFF** is available by default. You can create your own recipes to customize data extraction.

  You must use the MBSE Integration Gateway **importCollection** API to download the contents of the recipe.
- The **importCollection** API generates a JSON file that contains the Teamcenter data. Your ALM tool must process this JSON file to import the data into the tool.

For data handoff, you can use the following features that are available by default:

- An integration definition file called *POLARION_SWHW_BHMIntegrationDefinition.xml* is available. This file is created for integrating with Polarion. If you have a different ALM tool, you can create a new integration definition file. See the help on *Data mapping using the integration definition file* in the MBSE Integration Gateway documentation for more information.

- A recipe called **SW_ARCH_HANDOFF** is available for generating the data you need. You can also create your own recipes to generate the data you require. See the Model Based Systems Engineering documentation for more information.

- The **importCollection** API is used for extracting the data from Teamcenter. For more information, see the MBSE Integration Gateway documentation.

You can also **perform a test run of the handoff** using the MBSE Integration Gateway SDK.

Prepare a software specification package using the MBSE Integration Gateway SDK

The software specification package from Teamcenter is made available to the ALM tool in the form of a JSON file. You can generate this JSON file by using the sample modeling tool in the MBSE Integration Gateway SDK. To generate the JSON file, ensure that:

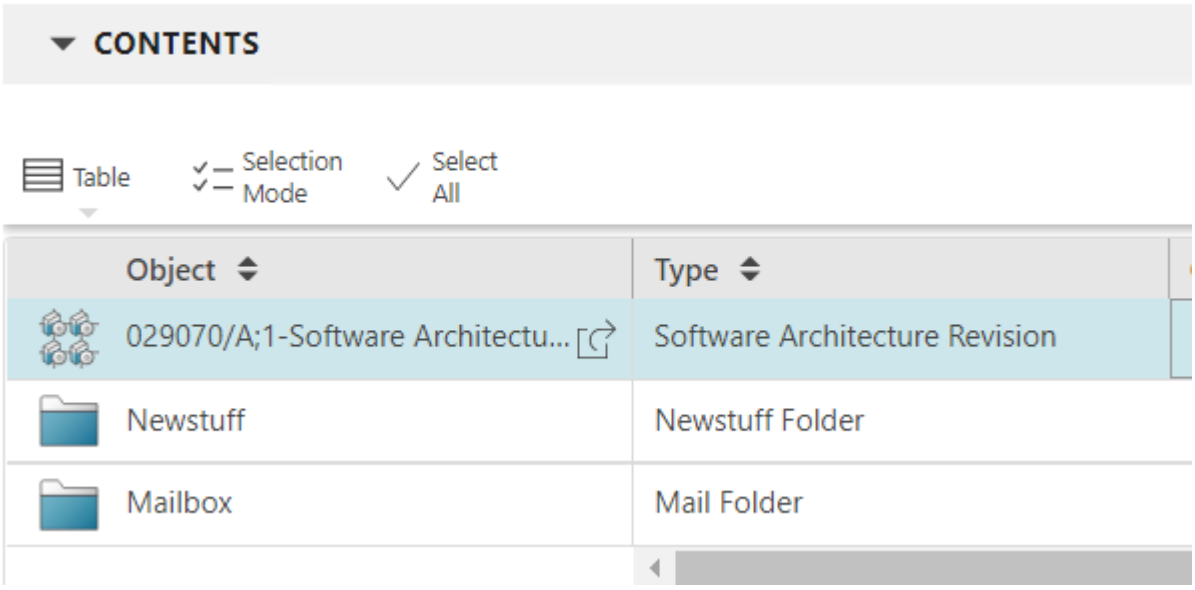
- You have access to an integration definition file.

- You have set up the MBSE Integration Gateway SDK and the sample modeling tool.

- You have data in Teamcenter that meets your test criteria.

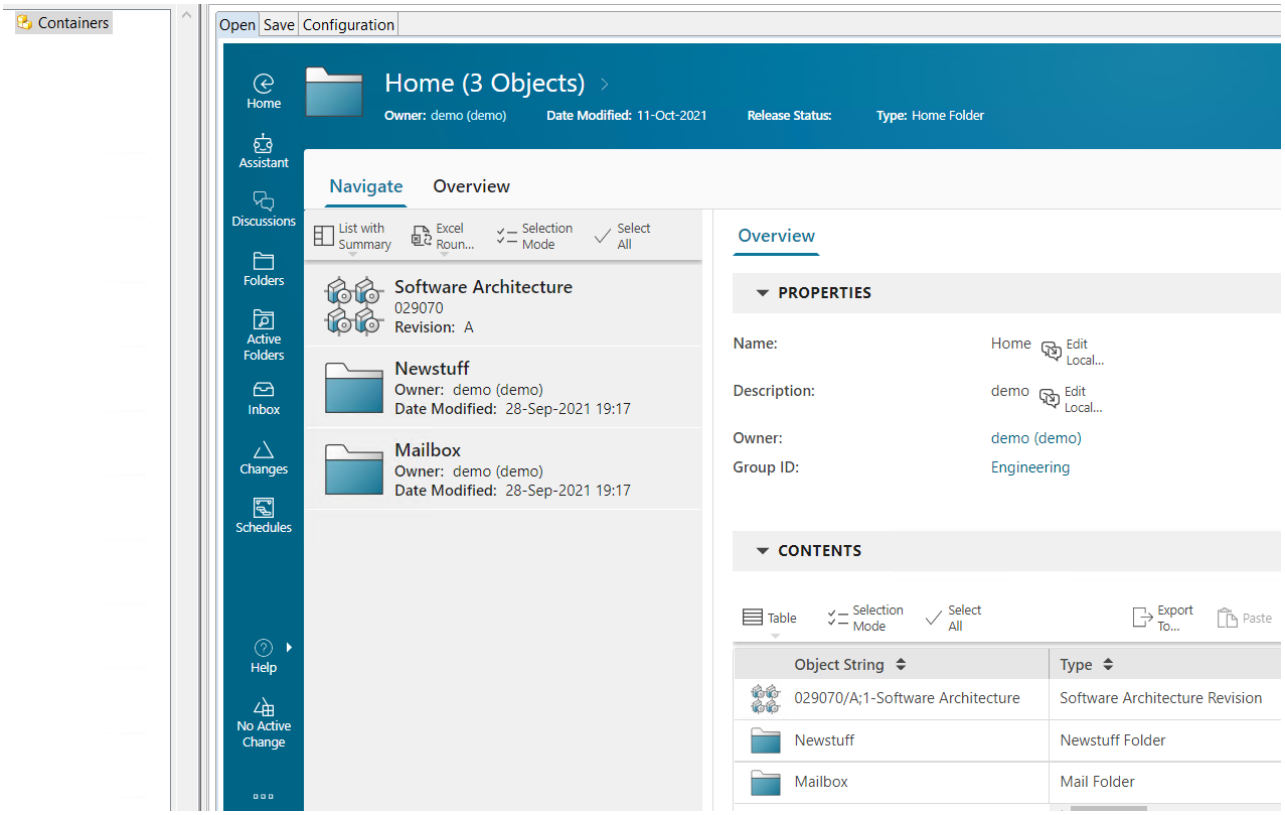
- You have access to the **SW_ARCH_HANDOFF** recipe.

1. Ensure that the data you want to handoff is present in Active Workspace.

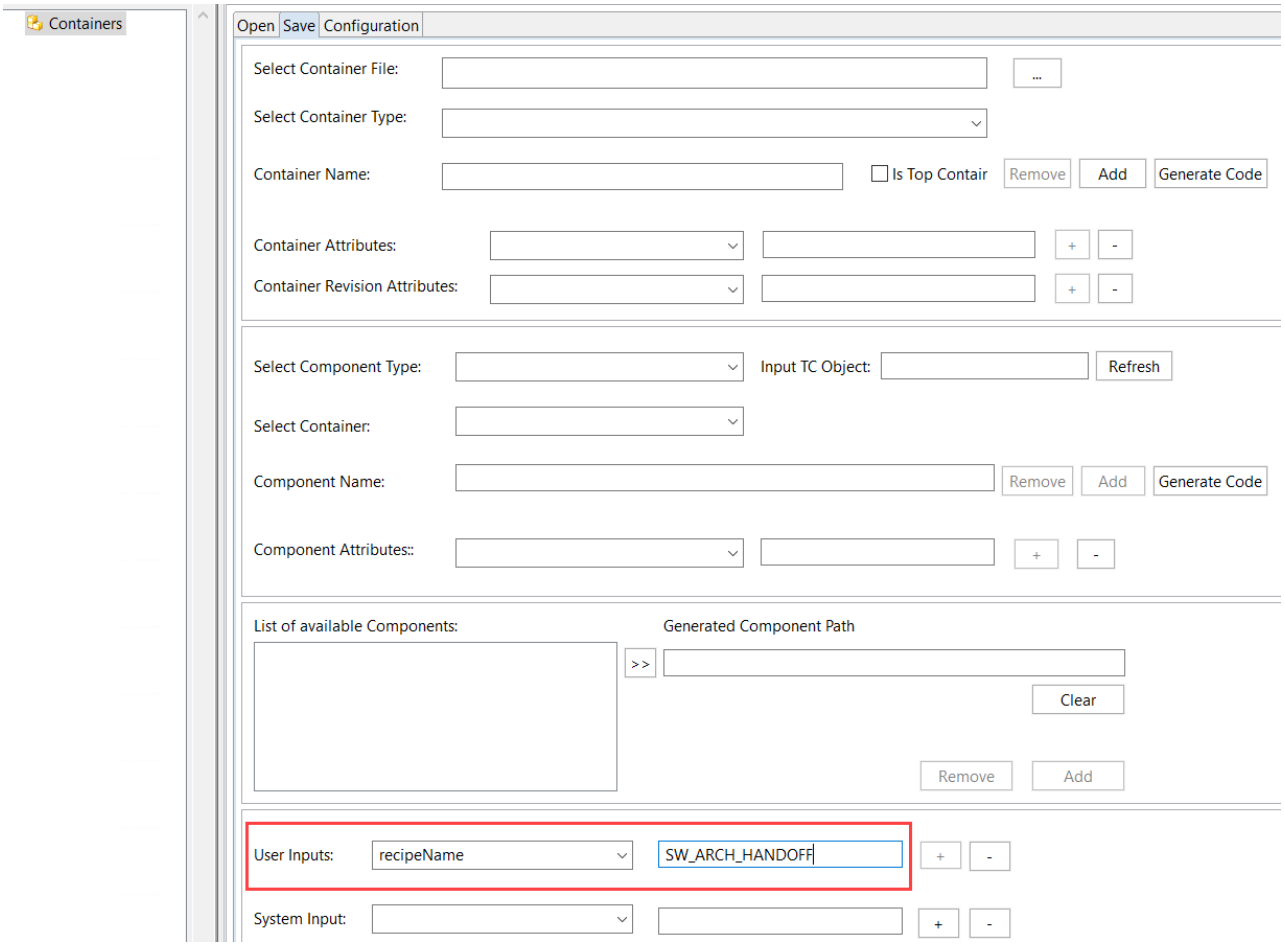


2. Start the sample modeling tool from the MBSE Integration Gateway SDK and load the Teamcenter configurations.

3. In the **Open** tab, select the data you want to hand off.



4. In the **Save** tab, from the **User Input** list, select **recipeName** and type **SW_ARCH_HANDOFF** in the box next to it.



5. Click **Import**.

   The right pane displays the JSON file. This data is generated using the **importCollection** API.

   The JSON file is also saved to the staging directory.

6. Open the JSON file and check if the data is correct.

Once you have successfully generated the current JSON file, ensure that your ALM tool can process this JSON file.

The next step is to automate this process in your ALM tool. You can use contents of the MBSE Integration Gateway SDK within your project to automate the data extraction process.

4. Publishing software deliverables to Teamcenter software part from Jenkins or from the command line

Overview of publishing software deliverables to Teamcenter

You can integrate Continuous Integration and Continuous Deployment systems such as Polarion-Jenkins or Jira-Jenkins to publish software deliverables to Teamcenter.

To publish software deliverables to Teamcenter from Jenkins, ensure that you do the following:

- **Install and set up the MBSE Integration Gateway client on the same machine where a software build tool (such as Jenkins) is installed.**
- Integrate the software build tool (such as Jenkins) with an ALM tool such as Polarion or Jira and ensure that the ALM tool can trigger jobs in Jenkins.
- **Perform a test publication from Jenkins to Teamcenter.**

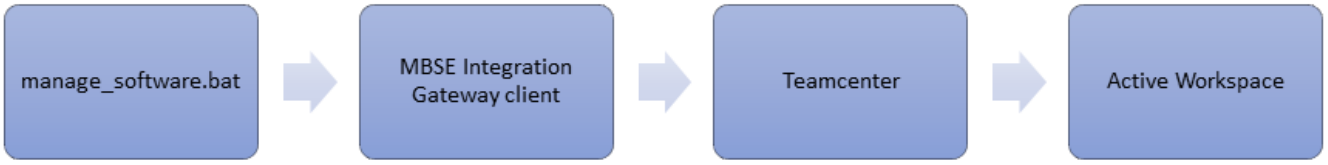
You can also customize the integration with the following configurations:

- **Generate and use an encrypted password file to pass Teamcenter credentials from Jenkins to Teamcenter.**
- **Update the *SCF.json* file to organize the data and set up additional configurations for the data you want to publish to Teamcenter.**
- Update the integration definition file if you want to customize the objects, relations, and file types used in the integration.

  For more information about the integration definition file, see the MBSE Integration Gateway documentation.

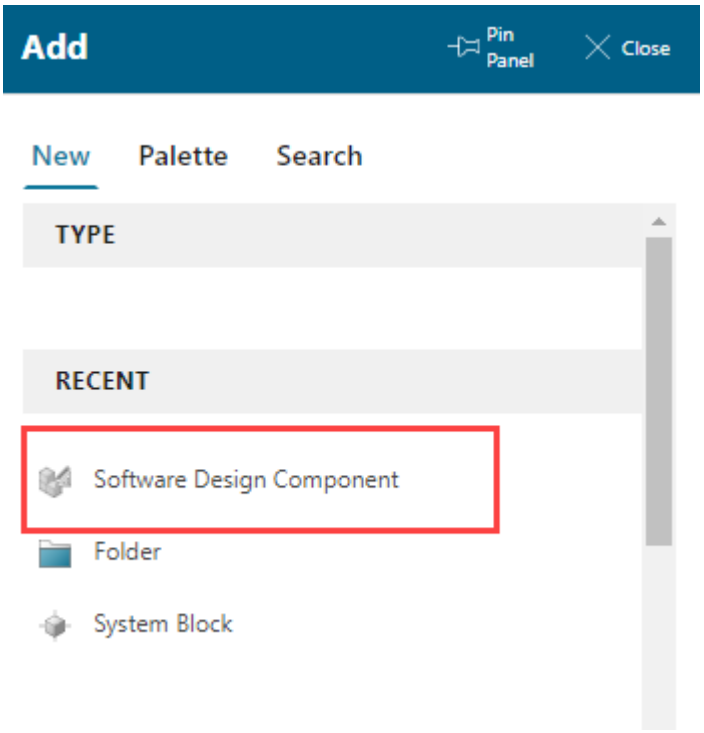
Perform a test publish operation using the manage_software.bat utility

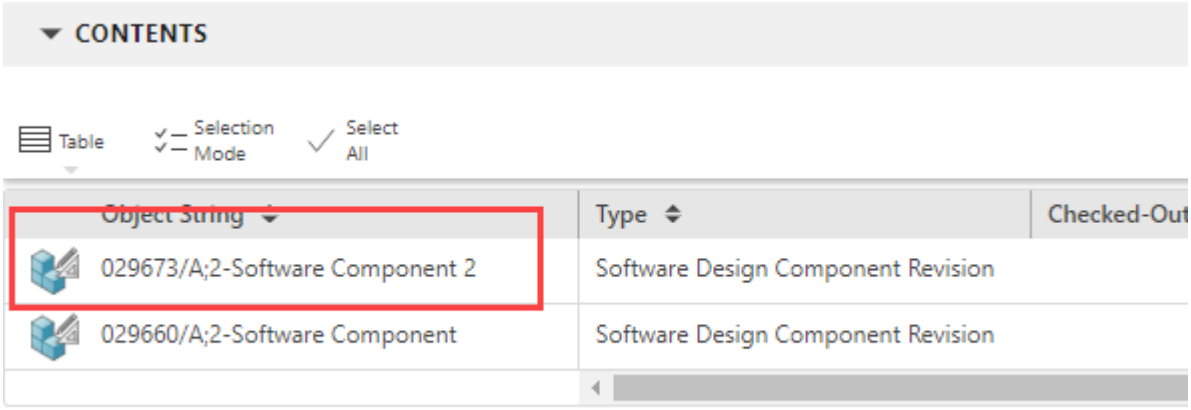
The *manage_software.bat* utility publishes the software deliverable to Teamcenter.



Before you integrate your ALM tool (Polarion, Jira) with your build tool, perform a test publish operation to check if the *manage_software.bat* utility publishes your software deliverable to a software part in Teamcenter.

1. In Active Workspace, create a Software Design Component part to function as the container or parent of the software part that you intend to publish from the utility.





Note down the ID and the revision of the Software Design Component part.

2. Create a folder and add the deliverable that you want to publish to Teamcenter to the folder. An example of such a deliverable is a text file.

3. In the folder that you created, paste the default *SCF.json* file from the *TC-ROOT\bhm\config* folder.

> Note:
>
> Use a copy of the default *SCF.json* file as the JSON file is overwritten after the publish operation.

4. Update the **TC_Object_Info** section of the *SCF.json* file that you copied as follows:

   a. Update the value of the **TC_context_Object_ID** property with the ID of the Software Design Component part that you created in Active Workspace.

   b. Update the value of the **TC_context_Object_revision** property with the revision of the Software Design Component part that you created in Active Workspace.

```
{"TC_Object_Info": {
    "Type": "SoftwareComponent",
    "TC_context_Object_ID": "029673",
    "TC_context_Object_revision": "A",
    "Metadata":     [
                {
```

5. From the command line, run the *manage_software.bat* utility with the following arguments:

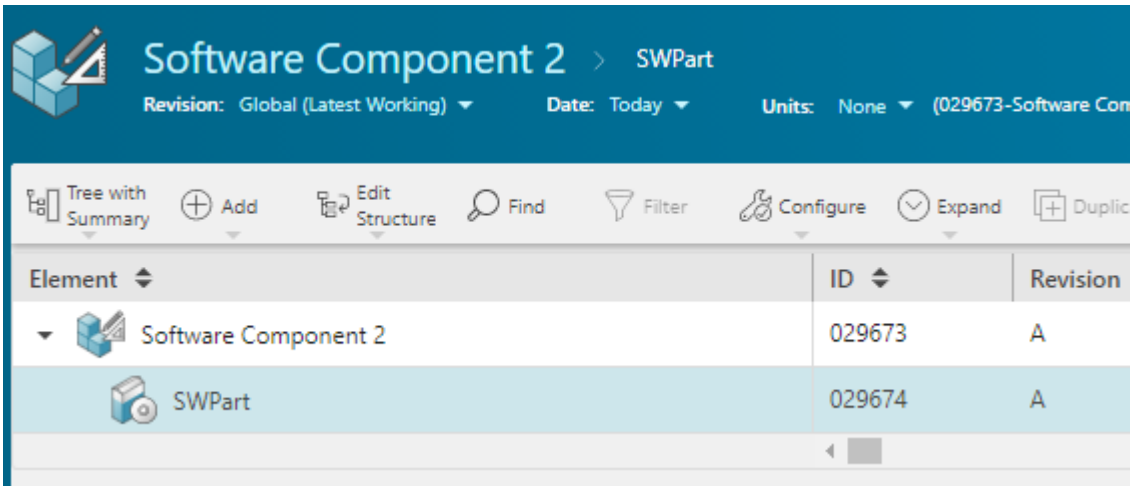
```
manage_software.bat -u=username –p=password -group=group
-pathToConfigFile=scf.json-file-path -inputDataPath=build-output-path
-toolType=SWBOM -useConsole=true
```

Example:

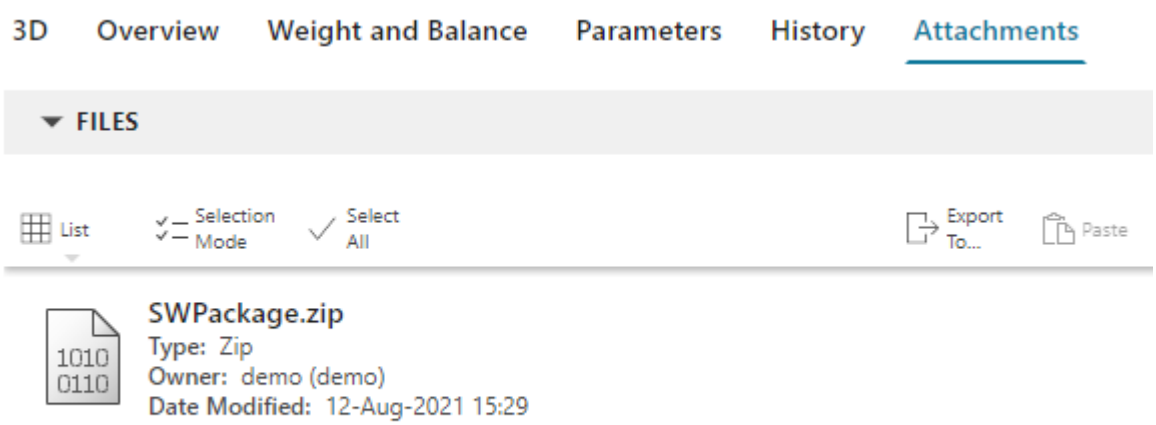
```
manage_software.bat -u=demo –p=demo -group=Engineering
-pathToConfigFile=D:\Test_data\SWHWBOM\Test2\scf.json
-inputDataPath=D:\Test_data\SWHWBOM\Test2 -toolType=SWBOM
-useConsole=true
```

6. Check the Software Design Component part in Active Workspace to ensure that:

   a. An **SWPart** is created under the Software Design Component part.



   b. Select the **SWPart** and then click the **Attachments** tab. The software deliverable must be in a zipped file.



c. Download and extract the contents of the zip file and check if the deliverable that you placed in the folder in step 2 is present.

7. To check if you can update the software deliverable:

   a. In the folder that you created in step 2, add another file, and run the *manage_software.bat* utility again.

   b. In Active Workspace, check the Software Design Component and make sure that the software package in the **Attachments** tab contains the new file.

Publish a software deliverable from your ALM tool and Jenkins to Teamcenter

To publish your software deliverable to Teamcenter from your ALM tool:

1. Integrate your ALM tool with a build tool such as Jenkins.

   For information about integrating Polarion with Jenkins, see the **Polarion documentation**.

   For information about integrating Jira with Jenkins, see the **Atlassian documentation**.

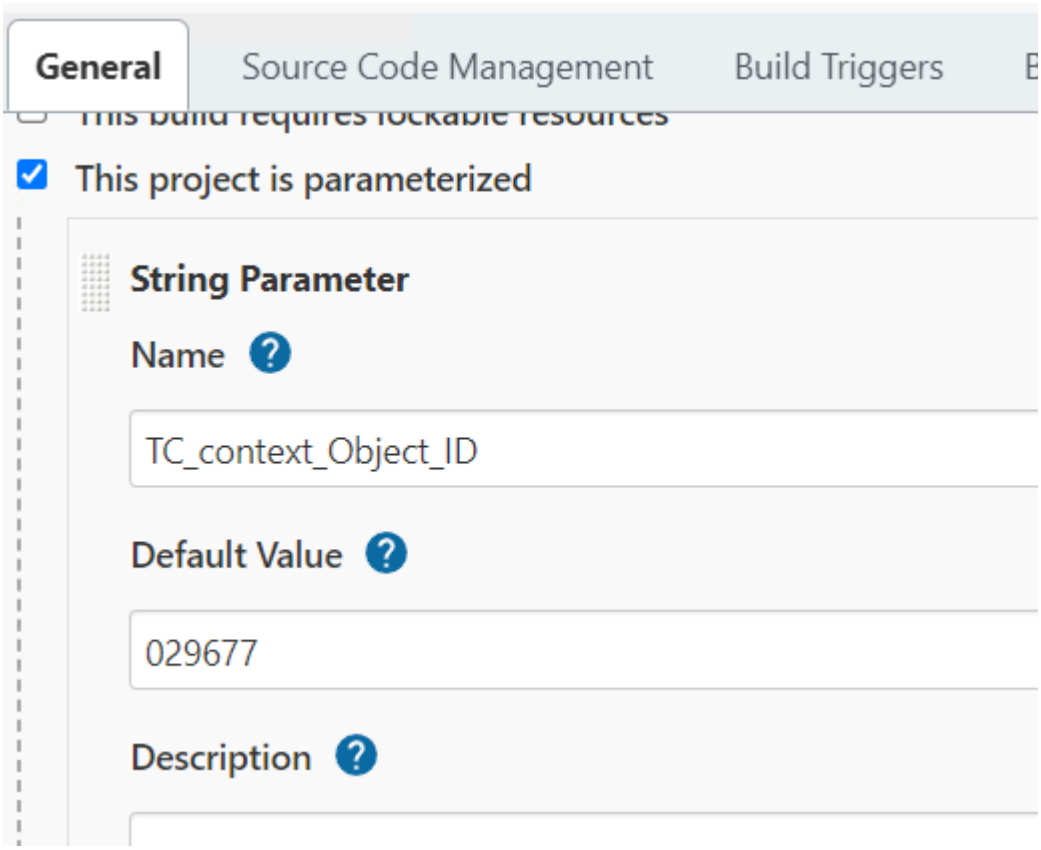
2. Configure Jenkins to trigger the *manage_software.bat* utility.

If you want to specify under what context object to save the deliverable and the revision ID of that context object, in the Jenkins job configuration, add string parameters that correspond to the context ID and context revision in the *SCF.json* file. Note that the parameter name in Jenkins must match the corresponding entry in the *SCF.json* file.

You can add the following string parameters in Jenkins:

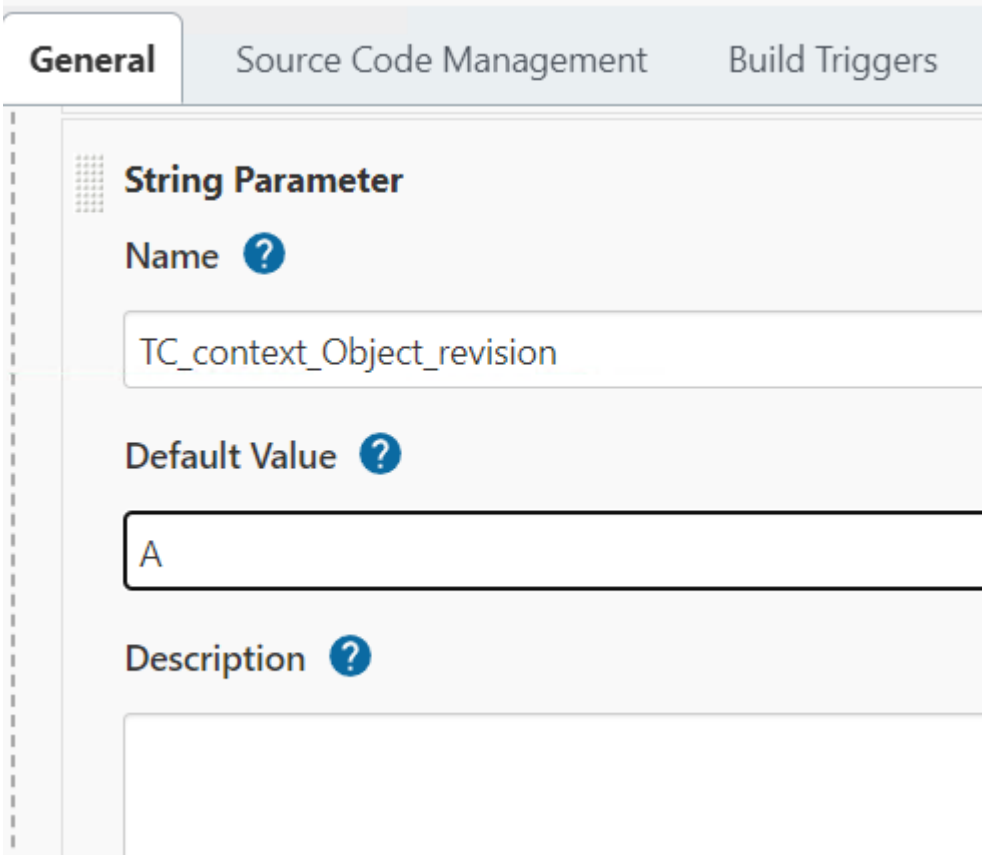
- **TC_context_Object_ID**

  This specifies under which context object the software deliverable is saved.



- **TC_context_Object_revision**

  This specifies the revision for the context object.



3. Verify if the software deliverable is published correctly in Active Workspace.

Using the manage_software.bat utility

The *manage_software.bat* utility publishes the software deliverable to Teamcenter. Before using this utility, ensure that the MBSE Integration Gateway client is installed.

Syntax

manage_software.bat [**-u**=*user-id* {**-p**=*password* | **-pf**=*password-file*} **-g**=*group* **-pathToConfigFile**=*scf.json-file-path* **-inputDataPath**=*build-output-path* **-toolType=SWBOM -useConsole=true -toolName**=*name-of-the-tool*]

Arguments

-u

Specifies the user ID.

This is a user with Teamcenter administration privileges.

> Note:
>
> If Security Services single sign-on (SSO) is enabled for your server, the **-u** and **-p** arguments are authenticated externally through SSO rather than being authenticated against the Teamcenter database. If you do not supply these arguments, the utility attempts to join an existing SSO session. If no session is found, you are prompted to enter a user ID and password.

-p

Specifies the password.

This argument is mutually exclusive with the **-pf** argument.

-pf

Specifies the password file.

This argument is mutually exclusive with the **-p** argument.

-g

Specifies the group associated with the user.

If used without a value, the user's default group is assumed.

-pathToConfigFile

Specifies the path to the *SCF.json* configuration file. An *SCF.json* file is available in the *config* directory of the MBSE Integration Gateway client. If you do not specify a path, the file from the default location is used.

-inputDataPath

Specifies the path to the data that the user wants to publish to Teamcenter.

-toolType

Specifies which way the artifacts will be published in Teamcenter. When you specify **SWBOM** it is published as software part. If you do not specify **SWBOM**, the data will be published as separate line items.

-useConsole

Specifies if this is a UI-based or console-based application. As Software Management is console based, this value must be set to **true**. If you do not specify a value or if the value is empty, you will encounter an error when running the utility.

-toolName

Specifies the tool that runs this batch file, such as Jenkins.

-h

Displays help for this utility.

Example

```
manage_software.bat -u=demo -p=demo -group=Engineering
-pathToConfigFile=D:\G3SCF\SWHW\publish\scf.json file path
-inputDataPath=D:\G3SCF\SWHW\publish -toolType=SWBOM -useConsole=true
-tooName=jenkins
```

Generate and use an encrypted password file to pass Teamcenter credentials from Jenkins to Teamcenter

You can generate an encrypted password file, using the *tcme_encryptpwf.bat* utility. This encrypted password file can be used when running the *manage_software.bat* utility.

`tcme_encryptpwf.bat -p` *password* `-pf` *path-to-password-file*

Example:

```
tcme_encryptpwf.bat -p demo -pf "D:\tc\mbse_client8\pwd.txt"
```

Using the SCF.json file to set up the data to be published to Teamcenter

The *SCF.json* file has the payload to be published in Teamcenter. By default, an *SCF.json* file is available in the *TC-ROOT/bhm/config* folder. Always use a copy of this file when you want to make any changes to it.

This payload is organized in the following sections:

- **TC_Object_Info section**
- **Metadata section**
- **SW_Part section**
- **SW_Package section**

TC_Object_Info section

```
{
    "TC_Object_Info": {
        "Type": "SoftwareComponent",
        "TC_context_Object_ID": "",
        "TC_context_Object_revision": "",
        "Metadata": [
        "SW_Part": [
            {
        ]
    }
}
```

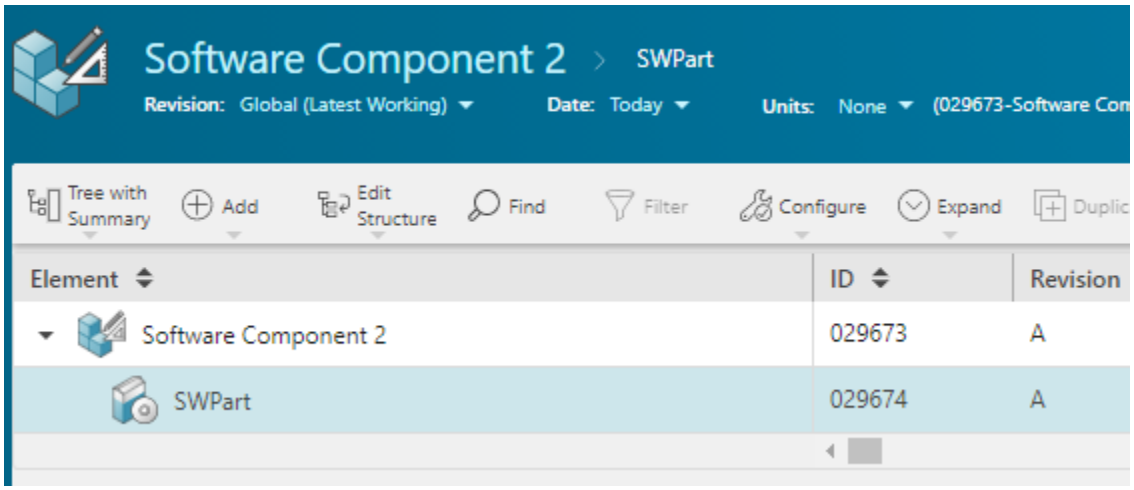
This section specifies where and under what context the software part is published in Teamcenter. This section has the following properties:

- **Type**

  This specifies the type of context object. The value must match the value in the *SWBOM_BHMIntegrationDefinition* file, which is the integration definition file.

  ```
  <ObjectMapping type="SoftwareComponent" behaviorType="MIXED" tctype="SwDesignComp">
      <BHMElement type="RootModel" tctype="SwDesignComp">
  ```

  You can also configure this value by updating the value in the integration definition file. Ideally, this must be a software object, such as **SoftwareComponent**. The software deliverable will be published under this object.



- **TC_context_Object_ID**

Specifies the ID of the context object.

- **TC_context_Object_revision**

  Specifies the revision of the context object.

If you do not specify the value of the **TC_context_Object_ID** and **TC_context_Object_revision**, the software part is directly published to Teamcenter without a context object. It will be an orphan part.

Metadata section

```
"Metadata": [
    {
        "Property": "BUILD_TOOL",
        "Value": ""
    },
    {
        "Property": "BUILD_NUMBER",
        "Value": ""
    },
    {
        "Property": "BRANCH_NAME",
        "Value": ""
    },
    {
        "Property": "BUILD_DISPLAY_NAME",
        "Value": ""
    },
    {
        "Property": "BUILD_URL",
        "Value": ""
    },
    {
        "Property": "JOB_URL",
        "Value": ""
    }
],
```

This section captures the build metadata from Jenkins.

SW_Part section

```
"SW_Part": [
    {
        "Type": "SoftwarePart.BINARY",
        "TC_SW_Part_name": "SWPart",
        "TC_SW_Part_ID": "",
        "TC_SW_Part_Revision": "",
        "Operation": "",
        "needObjectRelease": "No",
        "SW_Package": [
            {
        ]
```

This section specifies information about the software deliverable that is published in Teamcenter. This section has the following properties:

- **Type**

  Specifies the software part under which the software deliverables are published. The value must match the value in the *SWBOM_BHMIntegrationDefinition* file, which is the integration definition file.

```
<ObjectMapping type="SoftwarePart.BINARY" behaviorType="MIXED" tctype="Software">
    <BHMElement type="RootModel" tctype="Software">
```

- **TC_SW_Part_name**

  Specifies a name for the software part that is published. You can modify this name.

- **TC_SW_Part_ID**

  Specifies the ID of the software part that is published. If this field is empty, a new software part is published.

- **TC_SW_Part_Revision**

  Specifies the revision of the software part that is published. If this field and the **TC_SW_Part_ID** field are empty, a new software part is published.

- **Operation**

  Specifies whether to create a new software part or update or revise an existing part.

  - To create a new part, leave this property empty.

- To update an existing part, specify the value of the property as **update**.
- To revise an existing part, specify the value of the property as **revise**.

- **needObjectRelease**

  Specifies whether to release the published software part. Update the value as **Yes** or **No**. If the value is **Yes**, when the software deliverable is saved in Teamcenter, a release status is applied to the software part.

  You can choose which release status to apply by updating the value of the **SWM0_DefaultReleaseStatus** preference.

SW_Package section

```
"SW_Package": [
    {
        "Type": "SoftwarePackage.BINARY",
        "Name": "SWPackage",
        "PreserveFolderStructure": "true",
        "IncludeFiles": [
            "*"
        ],
        "ExcludeFiles": [
            ""
        ],
        "Metadata": []
    }
]
```

This section specifies information about the datasets attached to the software part. This section contains the following properties:

- **Type**

  Specifies where the software packaged is published. This value must match the value in the *SWBOM_BHMIntegrationDefinition* file, which is the integration definition file.

```
<BHMElement type="SoftwarePackage.BINARY" tctype="Zip" behaviorType="GRM" reltype="IMAN_reference" checkforduplicates="false" reftype="SoftwarePackage.BINARY">
    <AttributeMappings >
</BHMElement>
```

- **Name**

  Specifies the name of the software package. You can modify the default name.

- **PreserveFolderStructure**

  Specifies if you want to preserve the original folder structure that was used when publishing to Teamcenter.

- **IncludeFiles**

  Specifies which files to include when publishing the software deliverable to Teamcenter. For more information, see **SCF.json include-exclude pattern reference**.

  The following patterns are valid:

  - ***.ext***
  - *name.ext*
  - *name.**
  - *relative-path-of-folder/name.ext*

    For example, `./jar/abc.jar`
  - *relative-path-of-folder/*.ext*
  - *relative-path-of-folder/name*
  - *
  - ""

    This denotes an empty list.

- **ExcludeFiles**

  Specifies which files to exclude when publishing the software deliverable to Teamcenter. For more information, see **SCF.json include-exclude pattern reference**.

  The following patterns are valid:

  - ***.ext***
  - *name.ext*
  - *name.**
  - *relative-path-of-folder/name.ext*

For example, `./jar/abc.jar`

- *relative-path-of-folder*/*.*ext*
- *relative-path-of-folder*/*name*
- *
- ""

  This denotes an empty list.

- **Metadata**

  Specifies the build metadata. Jenkins metadata definitions are available by default in the JSON file. If you add custom metadata or if you are adding metadata from other build tools, ensure that you also update the integration definition file with the correct mappings.

```
<ObjectMapping type="Metadata" behaviorType="REF" tctype="Ess0SWBuildInformation">
    <BHMElement type="RootModel" tctype="Ess0SWBuildInformation">
       <AttributeMappings>
            <AttributeMapping name="BUILD_TOOL" tcattr="ess0BuildTool" includeinduplicatecheck="false"/>
            <AttributeMapping name="BUILD_NUMBER" tcattr="ess0BuildNumber" includeinduplicatecheck="false"/>
            <AttributeMapping name="BRANCH_NAME" tcattr="ess0BranchName" includeinduplicatecheck="false"/>
            <AttributeMapping name="BUILD_DISPLAY_NAME" tcattr="ess0BuildDisplayName" includeinduplicatecheck="false"/>
            <AttributeMapping name="BUILD_URL" tcattr="ess0BuildURL" includeinduplicatecheck="false"/>
            <AttributeMapping name="JOB_URL" tcattr="ess0JobURL" includeinduplicatecheck="false"/>
            <RevisionAttributeMapping/>
        </AttributeMappings>
    </BHMElement>
</ObjectMapping>
```

SCF.json include-exclude pattern reference

In the *SCF.json* file, you can specify the files to include and exclude when the software deliverable is published.

In the **SW_Package** section of the *SCF.json* file, the **IncludeFiles** property specifies which files to include, and the **ExcludeFiles** property specifies which files to exclude.

IncludeFiles

The following patterns are valid:

- ***.*ext*

  Includes all the files with the specified extension.

Example:

Include all files with the **.txt** extension:

```
"IncludeFiles": ["*.txt"],
```

Include all files ending with the word **_data**:

```
"IncludeFiles": ["*_data"],
```

- *name.ext*

  Includes all the files from the root folder that match the name in the pattern. Only files in the root folder are included. Those in the subfolders are not included.

  To include files from all folders, use the following pattern:

  name.ext

  Example:

  Include files named *common.jar* from the root folder:

  ```
  "IncludeFiles": ["common.jar"],
  ```

  Include files named *common.jar* from all folders:

  ```
  "IncludeFiles": ["*common.jar"],
  ```

- *name.**

  Includes files with the specified name and any extension from the root folder.

  To include files from all folders, use the following pattern:

  .name.*

  Example:

  If you specify the pattern as **data.***, all files named *data*, such as *data.txt* and *data.jar*, are included from the root folder.

  ```
  "IncludeFiles": ["data.*"],
  ```

If you specify the pattern as ***data.***, all files named *data*, such as *data.txt* and *data.jar*, are included from all folders.

```
"IncludeFiles": ["*data.*"],
```

- *relative-path-of-folder**/**name**.**ext*

  Includes files with the specified name and in the specified folder. Files are included only from the specified folder and not from all the folders.

  If you want to include files from the subfolders as well, use the following pattern:

  *relative-path-of-folder**/***name**.**ext*

  Example:

  Include files named *common.jar* from the *jar* folder:

  ```
  "IncludeFiles": ["jars/common.jar"],
  ```

- *relative-path-of-folder**/*.**ext*

  Includes all files with the specified extension in the specified folder and subfolders.

  Example:

  Include jar files from the *jar* folder:

  ```
  "IncludeFiles": ["jars/*.jar"],
  ```

  Include all files that end with the word *_data*:

  ```
  "IncludeFiles": ["./jars/*_data"],
  ```

- *relative-path-of-folder**/**name****

  Includes files starting with a given string in the specified folder.

  Example:

  Include files starting with *libBhm* in the *lib* folder:

  ```
  "IncludeFiles": ["dll/libBhm*"],
  ```

- ***

Includes all files in all the folders.

- **""**

  Denotes an empty list.

ExcludeFiles

The following patterns are valid:

- ***.***ext***

  Excludes all the files with the specified extension.

  Example:

  Exclude all files with the **.txt** extension:

  ```
  "ExcludeFiles": ["*.txt"],
  ```

  Include all files except the files that have the extension **.jar**:

  ```
  "ExcludeFiles": ["*"],

  "ExcludeFiles": ["*.jar"],
  ```

- *name.ext*

  Excludes all the files from the root folder that match the name in the pattern. Note that only files in the root folder are excluded. The files in the subfolders are not excluded.

  To exclude files in all folders, use the following pattern:

  name.ext

  Note:

  Exclude files from the root folder named *data.jar*:

  ```
  "ExcludeFiles": ["data.jar"],
  ```

  Exclude files from all folders named *data.jar*:

  ```
  "ExcludeFiles": ["*data.jar"],
  ```

- *name*.*

  Excludes files with the specified name and any extension from the root folder.

  If you want to exclude files from all folders, use the following pattern:

  .**name*.*

  Example:

  If you specify the pattern as **data.***, all files named *data*, such as *data.txt* and *data.jar*, are excluded from the root folder.

  ```
  "ExcludeFiles": ["data.*"],
  ```

  If you specify the pattern as ***data.***, all files named *data*, such as *data.txt* and *data.jar*, are excluded from all folders.

  ```
  "ExcludeFiles": ["*data.*"],
  ```

- *relative-path-of-folder*/*name*.*ext*

  Excludes files with the specified name and in the specified folder. Files from only the specified folder are excluded. Files from all the folders are not excluded.

  To exclude files from the subfolders as well, use the following pattern:

  relative-path-of-folder/**name*.*ext*

  Example:

  Exclude files named *common.jar* from the *jar* folder:

  ```
  "ExcludeFiles": ["jars/common.jar"],
  ```

  Exclude files named *common.jar* from the *jar* folder and the subfolders of the *jar* folder:

  ```
  "ExcludeFiles": ["jars/*common.jar"],
  ```

  Include the *jars* folder but exclude the file *1.txt* from the *jars* folder:

  ```
  "IncludeFiles": ["jars/"],

  "ExcludeFiles": ["jars/1.txt"],
  ```

- *relative-path-of-folder*/*.*ext*

  Excludes all files with the specified extension in the specified folder and subfolders.

  Example:

  Exclude jar files from the *jar* folder:

  ```
  "ExcludeFiles": ["jars/*.jar"],
  ```

  Include the *jar* folder but exclude all files named *common.jar* from this folder and its subfolders:

  ```
  "ExcludeFiles": ["jars/*common.jar"],
  ```

- *relative-path-of-folder*/*name**

  Excludes files starting with a given string in the specified folder.

  Example:

  Exclude files starting with *libBhm* in the *lib* folder:

  ```
  "ExcludeFiles": ["dll/libBhm*"],
  ```

- *****

  Excludes all files in all the folders.

- **""**

  Denotes an empty list.

Include-exclude cheat sheet

Requirement	Pattern
Include all files	`"IncludeFiles": ["*"]`
Exclude all files	`"ExcludeFiles": ["*"]`
Include a folder named **jars**	`"IncludeFiles": ["jars/"]`
Exclude a folder named **latest jars** inside the **jars** folder	`"ExcludeFiles": ["jars/latest jars"]`
Include all **txt** files from the **demo** folder	`"IncludeFiles": ["demo/*.txt"]` OR

Requirement	Pattern
	`"IncludeFiles": ["demo/*txt"]`
Exclude all files ending with **out** from the **metadata** folder.	`"ExcludeFiles": ["metadata/*out"]`
Include two folders: **data** and **code**	`"IncludeFiles": ["data/", “code/”]`
Exclude the **cache** folder and exclude all the **.pyc** files	`"ExcludeFiles": ["cache/", "*.pyc"]`
Include the files which begin with the string **BOM** from the **api** folder, but not its subfolders	`"IncludeFiles": ["api/BOM*"]`
Exclude the files which begin with the string **BOM** from the **api** folder, but not its subfolders	`"ExcludeFiles": ["api/*BOM*"]`