Hewlett Packard Enterprise

Operations Orchestration

Software Version: 10.70

Windows and Linux Operating Systems

Action Developers Guide

Document Release Date: November 2016

Software Release Date: November 2016

Legal Notices

Warranty

Restricted Rights Legend

Copyright Notice

Trademark Notices

Documentation Updates

To check for recent updates or to verify that you are using the most recent edition of a document, go to: https://softwaresupport.hpe.com/.

This site requires that you register for an HP Passport and to sign in. To register for an HP Passport ID, click **Register** on the HPE Software Support site or click **Create an Account** on the HP Passport login page.

You will also receive updated or new editions if you subscribe to the appropriate product support service. Contact your HPE sales representative for details.
(missing or bad snippet)

Contents

Introduction

This document provides content developers with guidelines for developing new operations for the Operations Orchestration platform. Operations are used to create new flows that can be executed inside HPE Operations Orchestration flow execution engine.

Introducing new content requires building an extension and deploying it to HPE OO Central or Studio. An extension is an artifact that contains actions written in Java or.NET.

Actions are code blocks used to create new operations for HPE Operations Orchestration. You can also use actions to create new operations for the CloudSlang platform. HPE Operations Orchestration can also execute operations and flows written in CloudSlang language.

System Requirements

- Java 1.7
- NET Framework 4.0 (required only if you want to create .NET actions)

- Maven 3.3.9 (if you want to build operations for CloudSlang)
- Maven 3.2.1
- Maven 3.0.5 (if you want to build an operation for Operations Orchestration 10.20 or lower versions)
- Java IDE (optionally, you may want to install a Java IDE with Maven support, like Eclipse or IntelliJ)
- Python IDE (optionally, you may want to install a Python IDE like PyCharm or Eclipse)

Prerequisites

> Java or .NET knowledge is required for developing new actions.

The action development process relies on Maven for resolving dependencies and building the project.

In order to properly comprehend all the topics discussed in this document, it is recommended that you read the “Maven Getting Started Guide” reference at the end of this document before continuing.

To develop CloudSlang content, it is recommended that you first read the “CloudSlang Tutorial” reference at the end of this document.

HPE OO Content Development

HPE OO Plugins

To create new content for HPE Operations Orchestration 10.x, you need to develop an extension called a plugin.

An OO plugin is a JAR file, packaged as a Maven plugin that contains one or multiple actions.

> Each plugin defines its own isolated classpath. Classpath isolation ensures that different plugins can use conflicting dependencies. For example, plugin A can use dependency X version 1.0 and plugin B can use the same X dependency, but with version 2.0. Classpath isolation ensures that you can use both plugins in the same flow regardless of the conflicting classpath issue.

HPE Operations Orchestration 10.x provides a simple way for creating Java actions by introducing the @Action annotation. See "Developing Actions" for more information.

HPE Operation Orchestration can also execute .NET actions. Actions written in .NET are referenced by a wrapping Java plugin. See ".NET Extensions" for more information.

> In previous versions of HPE Operations Orchestration, extensions were called IActions, They implemented the IAction interface. The IAction interface is now deprecated. Users writing new content should refrain from implementing the IAction interface and instead use the @Action annotation.

Developing Plugins for Java Actions

This section describes how to develop plugins containing only java actions, marked with the @Action annotation.

Using the Maven archetype **com.hp.oo.sdk:oo-plugin-archetype** you can create a skeleton for a plugin and a Studio project.

Preparing to Create a Plugin Using a Maven Archetype

Install Maven

Install Maven on your computer, adding the bin folder to the "Path" system variable. This enables you to run the `mvn` command from anywhere in the file system.

For more information on the Maven Installation process, check the “Installing Apache Maven” reference in the References section.

HPE Operations Orchestration SDK 10.6x depends on Maven 3.2.1. This version of Maven has a defect concerning the fact that it tries to resolve dependencies using external repositories even though we specify our internal Nexus artifact management system in the **settings.xml** file. The workaround is to force Maven to use a single repository – the internal repository. For more information, see the “Mirror Remote Maven Repositories” section.

Mirror Remote Maven Repositories

You can force Maven to use a single remote repository by mirroring all requests to remote repositories through an internal (company) repository. The internal repository must either contain all the desired artifacts, or it should be able to proxy the requests to other repositories. This setting is most useful when using an internal company repository with the Maven Repository Manager to proxy external requests.

To achieve this, configure a mirror for everything (*) in your **settings.xml** file. For complete guidelines, check the “Using Mirrors for Repositories” reference at the end of this document.

```
<settings>
  ...
  <mirrors>
    <mirror>
       <id>internal-repository</id>
       <name>Maven Repository Manager running on repo.mycompany.com</name>
       <url>http://repo.mycompany.com/proxy</url>
       <mirrorOf>*</mirrorOf>
    </mirror>
  </mirrors>
  ...
 </settings>
```

Create a local Maven repository

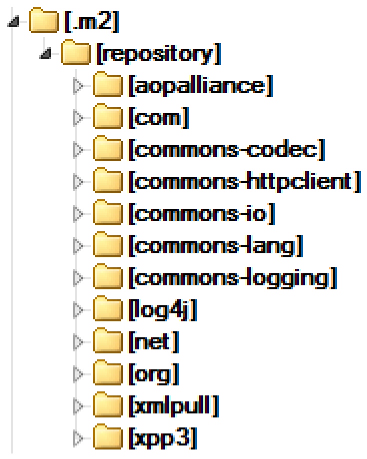
- Expand `sdk-dotnet-<version>.zip and sdk-java-<version>`.zip to:

  Windows: %HOMEPATH%\.m2\repository.

  Linux: $HOME/.m2/repository.

  > These files are located on the HPE OO ZIP file in the SDK folder.

  Following is an example of a directory structure, if the files were correctly extracted:



```
[.m2]
  [repository]
    [aopalliance]
    [com]
    [commons-codec]
    [commons-httpclient]
    [commons-io]
    [commons-lang]
    [commons-logging]
    [log4j]
    [net]
    [org]
    [xmlpull]
    [xpp3]
```

Register the plugin archetype

- Open the command prompt and enter the following command:

  ```
  mvn archetype:crawl
  ```

  This updates the Maven archetype catalog under $HOME/.m2/repository.

Creating a Plugin Using a Maven Archetype

Create a sample project

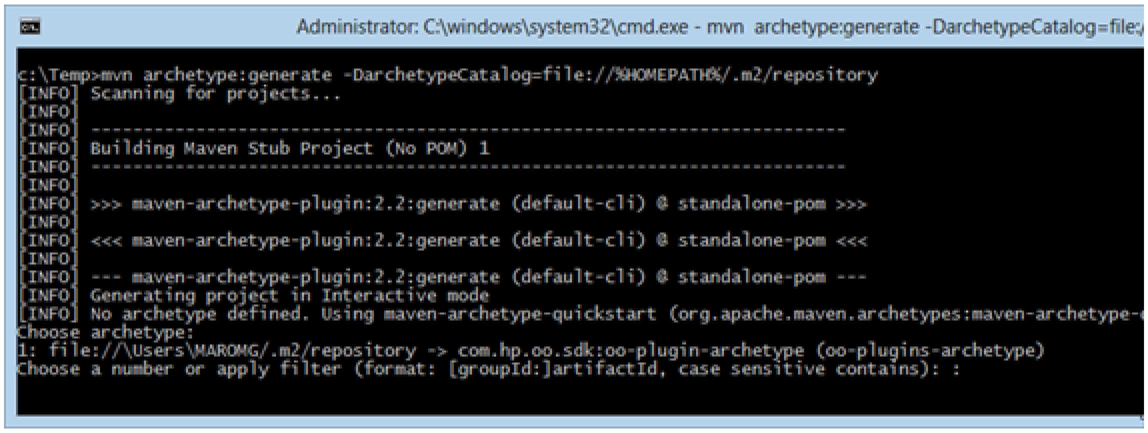
1. Go to the path where you want to create a sample plugin project, and enter the following command in the command line:

   ```
   mvn archetype:generate -DarchetypeCatalog=file://$HOME/.m2/repository
   ```

   Note: For Windows, use %HOMEPATH%.

   This initiates the project creation. A list of archetypes found in the catalog appears. Select the number representing the **archetype com.hp.oo.sdk:oo-plugin-archetype**.

   In the example below, select 1.

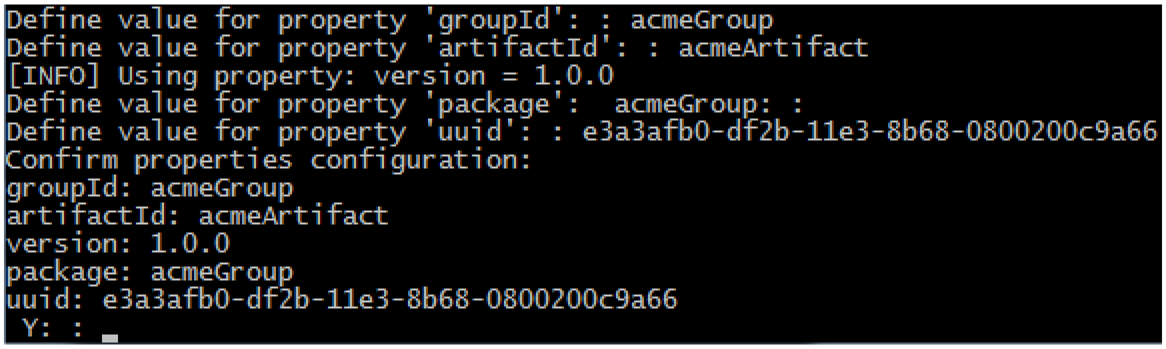
   

   Administrator: C:\windows\system32\cmd.exe - mvn archetype:generate -DarchetypeCatalog=file:

   ```
   c:\Temp>mvn archetype:generate -DarchetypeCatalog=file://%HOMEPATH%/.m2/repository
   [INFO] Scanning for projects...
   [INFO]
   [INFO] ------------------------------------------------------------------------
   [INFO] Building Maven Stub Project (No POM) 1
   [INFO] ------------------------------------------------------------------------
   [INFO]
   [INFO] >>> maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom >>>
   [INFO]
   [INFO] <<< maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom <<<
   [INFO]
   [INFO] --- maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom ---
   [INFO] Generating project in Interactive mode
   [INFO] No archetype defined. Using maven-archetype-quickstart (org.apache.maven.archetypes:maven-archetype-
   Choose archetype:
   1: file://\Users\MAROMG/.m2/repository -> com.hp.oo.sdk:oo-plugin-archetype (oo-plugins-archetype)
   Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): :
   ```

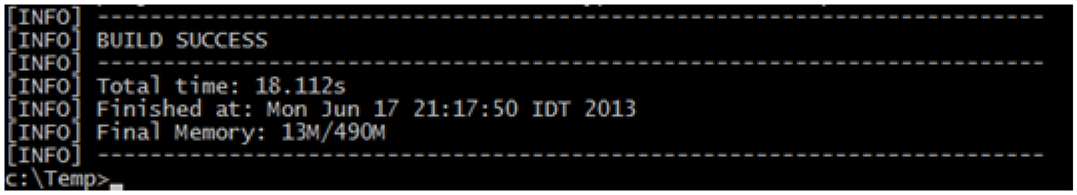
2. During the archetype creation, enter the following details:

   - `groupId`: The group id for the resulting Maven project. `acmeGroup` is used in the example below.
   - `artifactId`: The artifact id for the resulting Maven project. `acmeArtifact` is used in the example below.
   - `package`: The package for the files in the project. The default for this option is the same as the `groupId`.
   - `uuid`: The UUID of the generated project. A randomly generated UUID is used in the example below.



```
Define value for property 'groupId': : acmeGroup
Define value for property 'artifactId': : acmeArtifact
[INFO] Using property: version = 1.0.0
Define value for property 'package':  acmeGroup: :
Define value for property 'uuid': : e3a3afb0-df2b-11e3-8b68-0800200c9a66
Confirm properties configuration:
groupId: acmeGroup
artifactId: acmeArtifact
version: 1.0.0
package: acmeGroup
uuid: e3a3afb0-df2b-11e3-8b68-0800200c9a66
 Y: : _
```

The build finishes and a project is created.



```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 18.112s
[INFO] Finished at: Mon Jun 17 21:17:50 IDT 2013
[INFO] Final Memory: 13M/490M
[INFO] ------------------------------------------------------------------------
c:\Temp>_
```

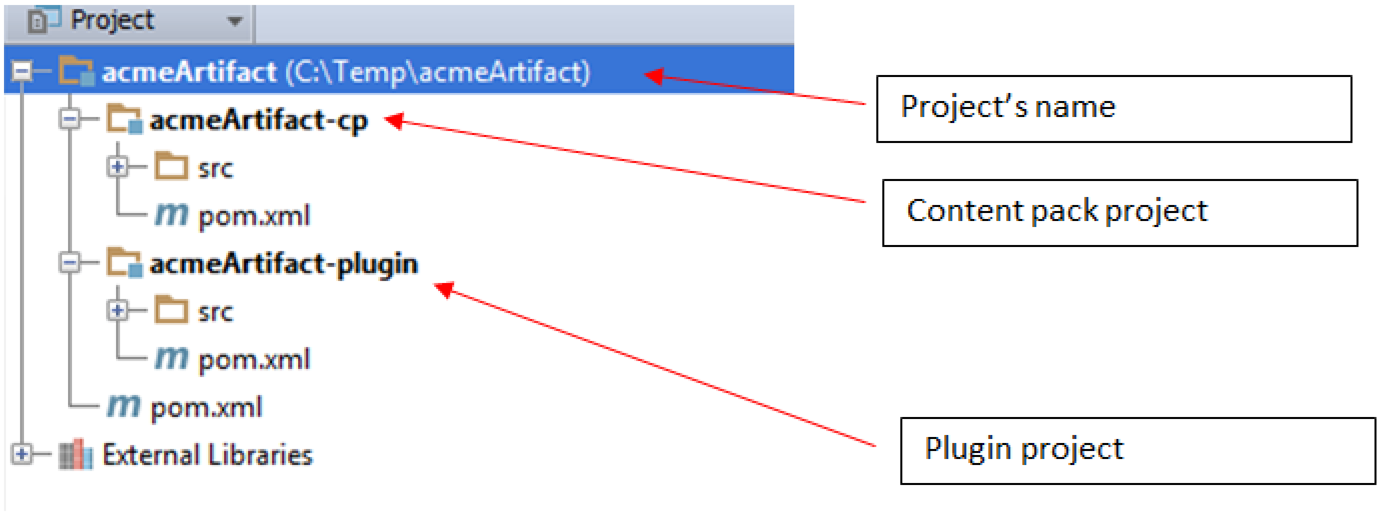
Open the Project in a Java IDE

The previous step created a new Java project with a Maven-based model.

Open this project in a Java IDE application.

The project contains two modules that have the same prefix as the provided artifact ID. One of the projects is a content pack project and the other is a plugin project.

For example:



Parent project

In the illustrated example, the parent project is called **acmeArtifact**.

By default it contains two modules—one a content pack and the other a plugin. This project groups @Actions and their relevant operations and flows into a single content pack.

> In large projects you can group actions in plugins based on functionality, protocols or technologies such as: ssh-plugin, http-plugin etc. When you create operations from the actions of a project, place them all in a single content pack project, disregarding the plugin in which it is contained.

Example: If you were developing a Microsoft Office integration, you might create several plugin projects—one for each Office version. But there would be a single content pack project containing the operations and flows.

Plugin project

In the example, the plugin project is called **acmeArtifact-plugin**.

This module contains the @Actions. When you are building this project (with Maven), the code inside is compiled and the resulting JAR file can be opened in Studio, and operations can be created from the @Actions inside.

Inside this module, a sample @Action can be found. You can delete it and write your own.

Content pack project

In the example, the content pack project is called **acmeArtifact-cp**.

This module represents the content pack. It includes any plugin modules upon which it is dependent, for example, **acmeArtifact-plugin** and its dependencies), as well as any flows, operations, and configuration items defined within it.

Creating a New Action

A Java action is a method that conforms to the signature `public Map<String, String> doSomething(parameters)`, annotated with the **@Action** annotation.

You can find the complete specification of the **@Action** annotation in the Developing Actions section.

Build Maven Plugin

To build your maven project, navigate to the location of the parent project, for example, c:\temp\acmeArtifact), open the command line and `run mvn clean install`.

You can see the artifacts resulted from the build process under the **target** folder of each module.

These artifacts are also installed in the local maven repository located at **%HOMEPATH%\.m2\repository or $HOME/.m2/repository**.

For example, you can find the resulted artifact from building **acmeArtifact-plugin** loacted in: **%HOMEPATH%\.m2\repository\acmeGroup\acmeArtifact-plugin\1.0.0\acmeArtifact-plugin-1.0.0.jar**.

Developing Plugins for .NET Actions

In order to create content using .NET actions, you need to:

1. Create a DLL file containing the implementation of the desired actions. The action class should implement an IAction interface.
2. Deploy the created DLL, including referenced libraries, to the local Maven repository, using `mvn install:install-file`. For more information on installing artifacts that were not built by Maven, see http://maven.apache.org/plugins/maven-install-plugin/usage.html.
3. Generate an HPE OO Maven plugin, wrapping the .NET action. To do this, you need to:
    a. Create a **pom.xml** file. For POM references, see http://maven.apache.org/pom.html.
    b. Under the <dependencies> tag, add a list containing all the required DLLs. Define all DLL artifacts using <type>dll</type>.
    c. Run the mvn install command from the folder containing the **pom.xml** file. This is assuming that the Maven bin folder is contained in the system path.

The result is the Maven plugin, placed in the target folder and installed to the local Maven repository. The target folder location is relative to the current folder.

The template of the **pom.xml** is:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
                 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

     <groupId>[my plugin groupId]</groupId>
     <artifactId>[my plugin artifactId]</artifactId>
     <version>[my plugin version]</version>

     <packaging>maven-plugin</packaging>

     <properties>
<oo-sdk.version>[THE LATEST HPE OO_SDK VERSION]</oo-sdk.version>
<oo-dotnet.version>[THE LATEST HPE OO_DOTNET VERSION]</oo-dotnet.version>
     </properties>

     <dependencies>
        <!-- required dependencies -->
        <dependency>
              <groupId>com.hp.oo.sdk</groupId>
              <artifactId>oo-dotnet-action-plugin</artifactId>
              <version>${oo-sdk.version}</version>
        </dependency>

        <dependency>
              <groupId>com.hp.oo.dotnet</groupId>
              <artifactId>oo-dotnet-legacy-plugin</artifactId>
              <version>${oo-dotnet.version}</version>
              <type>dll</type>
         </dependency>

         <dependency>
              <groupId>${project.groupId}</groupId>
              <artifactId>IAction</artifactId>
              <version>9.0</version>
              <type>dll</type>
         </dependency>
         <!-- end of required dependencies -->

         <dependency>
              <groupId>[groupId-1]</groupId>
              <artifactId>[artifactId-1]</artifactId>
              <version>[version-1]</version>
              <type>dll</type>
         </dependency>

         <dependency>
              <groupId>[groupId-2]</groupId>
              <artifactId>[artifactId-2]</artifactId>
              <version>[version-2]</version>
              <type>dll</type>
         </dependency>

        ...

         <dependency>
              <groupId>[groupId-n]</groupId>
              <artifactId>[artifactId-n]</artifactId>
              <version>[version-n]</version>
              <type>dll</type>
         </dependency>
    </dependencies>

    <build>
       <plugins>
          <plugin>
                <groupId>com.hp.oo.sdk</groupId>
                <artifactId>oo-action-plugin-maven-plugin</artifactId>
                <version>${oo-sdk.version}</version>
                <executions>
                     <execution>
                         <id>generate plugin</id>
                         <phase>process-sources</phase>
                         <goals>
                            <goal>generate-dotnet-plugin</goal>
                         </goals>
                     </execution>
                 </executions>
           </plugin>
         </plugins>
     </build>
 </project>
```

Example: In the following example:

- The POM file is named **example.pom.xml**.
- The **my-dotnet-actions.dll** contains the desired actions.
- The generated Maven plugin is **com.example:my-dotnet-plugin:1.0**.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                       http://maven.apache.org/xsd/maven-4.0.0.xsd">

     <modelVersion>4.0.0</modelVersion>
```

```
 <groupId>com.example</groupId>
 <artifactId>my-dotnet-plugin</artifactId>
 <version>1.0</version>

 <packaging>maven-plugin</packaging>

 <properties>
    <oo-sdk.version>[THE LATEST HPE OO_SDK VERSION]</oo-sdk.version>
    <oo-dotnet.version>[THE LATEST HPE OO_DOTNET VERSION]</oo-dotnet.version>
 </properties>

 <dependencies>
     <!-- required dependencies -->
     <dependency>
          <groupId>com.hp.oo.sdk</groupId>
          <artifactId>oo-dotnet-action-plugin</artifactId>
          <version>${oo-sdk.version}</version>
     </dependency>
     <dependency>
          <groupId>com.hp.oo.dotnet</groupId>
          <artifactId>oo-dotnet-legacy-plugin</artifactId>
          <version>${oo-dotnet.version}</version>
          <type>dll</type>
     </dependency>
     <dependency>
          <groupId>${project.groupId}</groupId>
          <artifactId>IAction</artifactId>
          <version>9.0</version>
          <type>dll</type>
     </dependency>
     <!-- end of required dependencies -->

     <dependency>
          <groupId>com.example</groupId>
          <artifactId>my-dotnet-actions</artifactId>
          <version>1.0</version>
          <type>dll</type>
     </dependency>
</dependencies>

<build>
    <plugins>
         <plugin>
             <groupId>com.hp.oo.sdk</groupId>
             <artifactId>oo-action-plugin-maven-plugin</artifactId>
             <version>${oo-sdk.version}</version>
             <executions>
                <execution>
                   <id>generate plugin</id>
```

```
                        <phase>process-sources</phase>
                        <goals>
                           <goal>generate-dotnet-plugin</goal>
                        </goals>
                     </execution>
                 </executions>
              </plugin>
         </plugins>
     </build>
</project>
```

Developing Plugins for Legacy Actions

> This is a deprecated way of writing actions. It is recommended to use the @Action annotation for developing new Java actions.

In order to create content using legacy actions, you need to:

1. Verify that you have a JAR containing the implementation of the desired actions, just like in version 9.x. The action class should implement an IAction interface.
2. Deploy the JAR, including referenced libraries, to the local Maven repository, using mvn install:install-file. For more information on installing artifacts that were not built by Maven, see http://maven.apache.org/plugins/maven-install-plugin/usage.html.
3. Generate an HPE OO Maven plugin, wrapping the legacy actions library. To do this, you need to:
    a. Create a **pom.xml** file. For POM references, see http://maven.apache.org/pom.html.
    b. Run the `mvn install` command from the folder containing the **pom.xml** file. This is considering that the Maven bin folder is contained in the system path.

The result is the Maven plugin, placed in the target folder and installed to the local Maven repository. The target folder location is relative to the current folder.

The content of the **pom.xml** is:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
```

```
<groupId>[my plugin groupId]</groupId>
<artifactId>[my plugin artifactId]</artifactId>
<version>[my plugin version]</version>

<packaging>maven-plugin</packaging>

<properties>
    <oo-sdk.version>[THE LATEST HPE OO_SDK VERSION]</oo-sdk.version>
    <oo-dotnet.version>[THE LATEST HPE OO_DOTNET VERSION]</oo-dotnet.version>
</properties>

<dependencies>
     <!-- required dependencies -->
     <dependency>
        <groupId>com.hp.oo.sdk</groupId>
        <artifactId>oo-legacy-action-plugin</artifactId>
        <version>${oo-sdk.version}</version>
     </dependency>
     <!-- end of required dependencies -->

     <dependency>
        <groupId>[groupId-1]</groupId>
        <artifactId>[artifactId-1]</artifactId>
        <version>[version-1]</version>
     </dependency>

     <dependency>
        <groupId>[groupId-2]</groupId>
        <artifactId>[artifactId-2]</artifactId>
        <version>[version-2]</version>
     </dependency>

     ...

    <dependency>
        <groupId>[groupId-n]</groupId>
        <artifactId>[artifactId-n]</artifactId>
        <version>[version-n]</version>
    </dependency>
</dependencies>


<build>
     <plugins>
        <plugin>
            <groupId>com.hp.oo.sdk</groupId>
            <artifactId>oo-action-plugin-maven-plugin</artifactId>
            <version>${oo-sdk.version}</version>
            <executions>
```

```
                   <execution>
                    <id>generate plugin</id>
                    <phase>process-sources</phase>
                       <goals>
                            <goal>generate-legacy-plugin</goal>
                       </goals>
                    </execution>
                </executions>
             </plugin>
         </plugins>
    </build>
</project>
```

Example: In the following example:

- The POM file is named **example.pom.xml**.
- The my-legacy-actions.jar contains the desired actions.
- The generated Maven plugin is `com.example:my-legacy-actions:1.0`.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>my-legacy-actions-plugin</artifactId>
    <version>1.0</version>

    <packaging>maven-plugin</packaging>

    <properties>
         <oo-sdk.version>[THE LATEST HPE OO_SDK VERSION]</oo-sdk.version>
         <oo-dotnet.version>[THE LATEST HPE OO_DOTNET VERSION]</oo-dotnet.version>
    </properties>

    <dependencies>
         <!-- required dependencies -->
         <dependency>
            <groupId>com.hp.oo.sdk</groupId>
            <artifactId>oo-legacy-action-plugin</artifactId>
            <version>${oo-sdk.version}</version>
         </dependency>
         <!-- end of required dependencies -->

         <dependency>
```

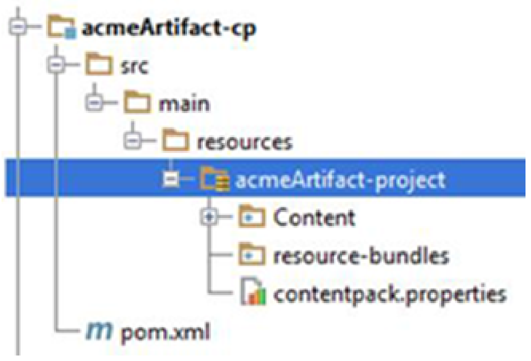
```
            <groupId>com.example</groupId>
            <artifactId>my-legacy-actions</artifactId>
            <version>1.0</version>
         </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>com.hp.oo.sdk</groupId>
                <artifactId>oo-action-plugin-maven-plugin</artifactId>
                <version>${oo-sdk.version}</version>
                <executions>
                    <execution>
                        <id>generate plugin</id>
                        <phase>process-sources</phase>
                        <goals>
                            <goal>generate-legacy-plugin</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
 </project>
```

Creating Operations from @Actions

To create an operation from the @Action that you developed in the Developing Plugins for Java Actions section, you have to complete the following steps:

1. Find the Studio project inside the content pack module, for example, **acmeArtifact-cp/src/main/resources/acmeArtifact-project**.



2. Import this project into Studio.
3. Build the **acmeArtifact** parent module using Maven.
4. Import **acmeArtifact-plugin** to Studio.

   The location of the plugin that you have to import is: **%HOMEPATH%\.m2\repository\acmeGroup\acmeArtifact-plugin\1.0.0\acmeArtifact-plugin-1.0.0.jar**.
5. In Studio, create a new operation, and select your plugin in the **Create Operations** dialog box.

   Note: If you encounter errors when creating a new operation due to some missing dependencies, perform the following steps:

   a. Import the content pack from location: **%HOMEPATH%\.m2\repository\acmeGroup\acmeArtifact-cp\1.0.0\acmeArtifact-cp-1.0.0.jar**.
   b. Close the content pack in Studio.
   c. Retry to create the operation.

> These steps also apply for .NET and legacy Java actions.

Create a Content Pack

After creating an operation, you must create a content pack from your project, in order to deploy and use it in Central. You have three possible approaches to create the content pack:

- Create a content pack with Studio: see Studio Guide for details.
- Create a content pack with Maven:
  - Navigate to the location of the parent project, in the example in section Open the Project in a Java IDE, **c:\temp\acmeArtifact**, open the command line and `run mvn clean install`.
  - The content pack will be created inside the target folder from your content pack project.Open the Project in a Java IDE, **c:\temp\acmeArtifact\acmeArtifact-cp\target\acmeArtifact-cp-1.0.0.jar**)
- Create a content pack with OOSHA: see OOSHA Guide for details.

For details on how to deploy the content pack in Central, see Central Guide.

CloudSlang Content Development

Maven Artifacts

To create new CloudSlang content, you need to develop Java actions, grouped under a Maven Artifact.

In CloudSlang, actions are methods annotated with the @Action annotation.

> CloudSlang does not support .NET or legacy Java actions.

Developing Maven Artifacts for Java Actions

Preparing to Create a Maven Artifact

Install Maven

See the Preparing to Create a Plugin Using a Maven Archetype section from for a complete guideline on how to install and configure Maven.

> Install Maven 3.3.9 to develop actions for CloudSlang

Creating a Maven Project

1. Navigate to the location on the filesystem where your project should reside and start a shell in that directory.
2. On your command line, execute the following Maven command:
   ```
   mvn archetype:generate -DarchetypeArtifactId=maven-archetype-quickstart
   ```
3. During the project creation, enter the following details and press **Enter** after each one:

- `groupId`: The group id for the resulting Maven project. `acmeGroup` is used in the example below.
- `artifactId`: The artifact id for the resulting Maven project. `acmeArtifact` is used in the example below.
- `version`: The version of the resulting Maven project. `1.0-SNAPSHOT` is used in the example below.
- `package`: The package for the files in the project. The default for this option is the same as the `groupId`.

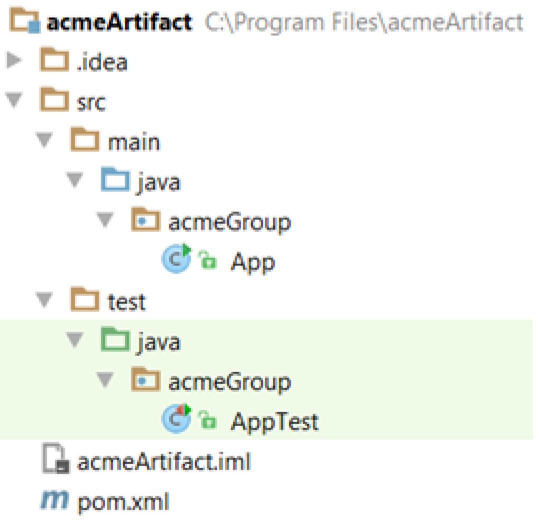
4. Locate the **pom.xml** file inside your project. Open it, and add the dependency below inside the `<dependencies>` tag.

```
<dependency>
    <groupId>com.hp.score.sdk</groupId>
    <artifactId>score-content-sdk</artifactId>
    <version>1.10.6</version>
</dependency>
```

Open the Project in a Java IDE

Open this project in a Java IDE application. It is recommended to use a Java IDE with Maven support.

The project contains the following directory structure:


```
acmeArtifact C:\Program Files\acmeArtifact
▶ .idea
▼ src
  ▼ main
    ▼ java
      ▼ acmeGroup
          App
  ▼ test
    ▼ java
      ▼ acmeGroup
          AppTest
  acmeArtifact.iml
  pom.xml
```

Creating a New Action

A Java action is a method that conforms to the signature `public Map<String, String> doSomething(parameters)`, annotated with the **@Action** annotation.

You can find the complete specification of the **@Action** annotation in the Developing Actions section.

Build Maven Artifact

To build your maven project, navigate to the location of the project, for example, **c:\Program Files\acmeArtifact**, open the command line and run `mvn clean install`.

You can see the artifacts resulted from the build process under the target folder generated inside your project.

These artifacts are also installed in the local maven repository located at **%HOMEPATH%\.m2\repository or $HOME/.m2/repository**.

For example, you can find the resulted artifact from building `acmeArtifact` located in: **%HOMEPATH%\.m2\repository\acmeGroup\acmeArtifact\1.0-SNAPSHOT\acmeArtifact- 1.0-SNAPSHOT.jar**.

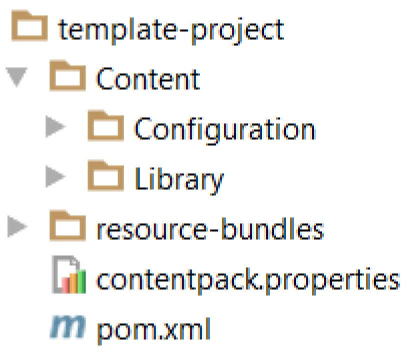
Creating CloudSlang Operation from Java Action

To create a CloudSlang operation, you must first create a project for placing the operation.

Creating a new Project from a predefined template

To create a new project:

1. Download **template-project.zip** from HPE LN.
2. Unzip **template-project.zip** to the desired location.
3. Find the directory **template-project** with the following structure.



```
template-project
▼ Content
  ▶ Configuration
  ▶ Library
▶ resource-bundles
  contentpack.properties
  pom.xml
```

4. Rename **template-project** directory to the name you want to provide for the new project

   For a **CloudSlang** project, the project naming convention is: `cs-[<vendor>]-<product|technology>`

   For example: `cs-vmware-vcenter`

5. Locate **pom.xml** file. Open it and replace the value inside the `artifactId` tag with the name of the project.

   For example: `<artifactId>cs-vmware-vcenter </artifactId>`

6. Locate the contentpack.properties file. Open it and provide values for the properties specified in that file.

   - **content.pack.name**: It is recommended to specify a value for this property. The value should be the name of the project.
   - **content.pack.version**: It is recommended to specify a value for this property. The value must be the same as the value from the version tag inside the **pom.xml** file. If you want to update the version of the project, you must update it inside both files.
   - **content.pack.description**
   - **content.pack**
   - **publisher**

Create CloudSlang Operation

Locate the Library folder inside your project, at `<project_path>/Content/Library`.

Inside the Library folder create a directory structure, representing the **namespace** of the operation.

> The **namespace** for a CloudSlang operation has a similar role to a Java package.

Example: Create the **namespace acme/operations**

Inside your **namespace**, create a new file with the **.sl** extension.

Example: The path of the new file will be **<project_path>/Content/Library/acme/operations/my_op.sl**.

Inside the **.sl** file, create a CloudSlang operation that calls the Java action. See the CloudSlang tutorial on how to build an operation for details.

Developing Maven Artifacts for Python Scripts

Create Maven Project

1. Create a Python project having the following tree structure:

   ```
   my-python-module
   |--__init__.py
   |--another-module
   |--__init__.py
   |--script1.py
   |--script2.py
   ```

2. Place the two template files, **pom.xml** and **assembly.xml** inside **my-python-module** folder.
3. Edit the **pom.xml** file and change the GAV of the module.

   ```
   <groupId>mycompany</groupId>
   <artifactId>my_first_artifact</artifactId>
   <version>87.0</version>
   ```

Open Project in Python IDE

Open the project in a Python IDE, to create Python functions that will be called from Python operations.

Build Maven Artifact

To build your maven project, navigate to the location of the Python project (my-python-module path), open the command line and run `mvn clean install`.

You can see the artifacts resulted from the build process under the target folder generated inside your project.

These artifacts are also installed in the local maven repository located at **%HOMEPATH%\.m2\repository or $HOME/.m2/repository**.

For example, you can find the resulted artifact from building **my-python-module** located in: **%HOMEPATH%\.m2\repository\ mycompany \my_first_artifact\87.0\ my_first_artifact-87.0.zip**.

Creating CloudSlang Python Operation

To create a CloudSlang operation, you must first create a project for placing the operation.

You can create the project from a predefined template as described in section Creating CloudSlang Operation from Action.

Create CloudSlang Operation

Locate the Library folder inside your project, at **<project_path>/Content/Library**.

Inside the Library folder create a directory structure, representing the namespace of the operation.

> The namespace for a CloudSlang operation has a similar role to a Java package.

Example: Create the **namespace acme/operations**

Inside your **namespace**, create a new file with the **.sl** extension.

Example: The path of the new file will be **<project_path>/Content/Library/acme/operations/my_op.sl**.

Inside the **.sl** file, create a CloudSlang operation that calls a Python script. In the Python script, you can invoke the Python functions that you developed. See the CloudSlang tutorial on how to build a Python operation for details.

Create a Content Pack

After creating an operation, you must create a content pack from your project, in order to deploy and use it in Central. You can create a content pack from a CloudSlang project using OOSHA. See OOSHA Guide for details.

For details on how to deploy the content pack in Central, see Central Guide.

> Python libraries will not be included in the content pack built by OOSHA. In order to include them, you must copy the zip resulted from building your project inside the content pack Lib folder.

For example, you will have to copy the zip from **%HOMEPATH%\.m2\repository\ mycompany \my_first_artifact\87.0\ my_first_artifact-87.0.zip** at the following location inside the content pack created by OOSHA: **Lib/mycompany/my_first_artifact/87.0/my_first_artifact-87.0.zip**.

Developing @Actions

An action is a method that has the `@com.hp.oo.sdk.content.annotations.Action` annotation.

@Action must be applied on Java methods that conforms to the signature:

```
@Action
public Map<String, String> doSomething(parameters)
```

The @Action annotation is invoked during flow execution, and specifies the following action information:

- **name**: name of the action
- **outputs**: action outputs
- **responses**: action responses

"Hello World!" Example

Following is a simple **"Hello World!"** action example:

```
public class MyActions {
     @Action
     public void sayHello() {
          System.out.println("Hello World!");
     }
 }
```

By default, the created @Action is named after the method that defines it. In the "Hello World!" example, the @Action name is **sayHello**. The @Action name is used in the operation's definition. The operation is the means to expose an @Action to Studio and to flow authors. Each operation points to a specific groupId, artifactId, version, and @Action name (GAV+@Action name).

You can customize the @Action name and provide a name that is different from the method name. You can do this using the @Action annotation value parameter. The following code defines the same "Hello World!" @Action, but names it my-hello-action:

```
public class MyActions {
    @Action(“my-hello-action”)
    public void sayHello() {
        System.out.println("Hello World!");
    }
 }
```

Passing Arguments to @Actions

An @Action is exposed to the flow context and can request parameters from it. The flow context holds the state of the flow. For example, consider the following @Action, which adds two numbers and prints the result to the console:

```
@Action
 public void sum(int x, int y){
   System.out.println(x+y);
 }
```

Parameters are taken from the context by name. The sum method requests two integer parameters x and y from the context. When invoking the @Action, HPE Operations Orchestration assigns the value of x and y from the context to the method arguments with the same name.

Just like with @Action, it is possible to customize parameter names and request that HPE Operations Orchestration resolves the value while using a custom name. In the following example, the sum method requests that the context op1 parameter is assigned to the x argument and op2 to the y argument:

```
@Action
 public void sum(@Param("op1") int x, @Param("op2") int y){
   System.out.println(x+y);
 }
```

The classes `ResponseNames`, `ReturnCodes`, `InputNames`, and `OutputNames`, under the `com.hp.oo.sdk.content.constants` package, include commonly used constants, which you can use in the @Action.

For example, input names such as HOST, USERNAME, PASSWORD, PORT, and so on, or response names such as SUCCESS, FAILURE, NO_MORE, and so on.

Return Values

An @Action, like any Java method, can also return a single value. The returned value is considered the return result of the @Action and is used as return result in the operation. It is also possible for an @Action to return multiple results to the operation. This is done by returning a Map<String, String>, where the Map key is the name of the result, and the associated value is the result value. Returning a Map<String, String> is a way for an @Action to pass multiple outputs to the operation at runtime.

Adding @Action Annotations

@Action annotations are used tso generate new operations in the Studio. When generating an @Action based operation, the new operation's initial attributes (description, inputs, outputs, responses) are taken from the @Action annotations definitions.

When developing plugins, you must correctly annotate the actions that return only a single value. The annotation has to declare an output with the special name `singleResultKey`. There is a constant `ActionExecutionGoal.SINGLE_RESULT_KEY` that assists you, for example:

```
@Action(name = "modulo-ten",
   description = "returns the last digit",
   outputs = @Output(ActionExecutionGoal.SINGLE_RESULT_KEY),
   responses = @Response(text = ResponseNames.SUCCESS,
   field = OutputNames.RETURN_RESULT,
   value = "0", matchType = MatchType.ALWAYS_MATCH,
   responseType = ResponseType.RESOLVED)
 )
 public int moduloTen(@Param("number") int number) {
   return number % 10;
 }
```

> It is important that you use @Action annotations; otherwise, operations created from these @Actions are harder to use.

Annotations

Adding metadata means adding or setting the relevant annotations and their attributes. The following table describes the @Action, @Output, @Response and @Param annotations:

Action

The `@com.hp.oo.sdk.content.annotations.Action` annotation specifies information on an action.

Attributes:

- value (optional): the name of the @Action
- description (optional)
- Output[] (optional): array of outputs (see below)
- Response[] (optional): array of responses (see below)

Comments:

You have two options for setting the name of the @Action:

1. The value attribute:

   ```
   @Action("aflPing") public void ping(…)
   ```

   or

   ```
   @Action(value="aflPing") public void ping(…)
   ```

2. The method name:

   ```
   @Action
    public void ping(…)
   ```

The names are checked in the above order. The first one checked is the value attribute. If it doesn’t exist, the method name is selected.

Param

The `@com.hp.oo.sdk.content.annotations.Param` annotation specifies information on a parameter of an action.

Attributes:

- value: the name of the input
- required (optional): by default is false
- encrypted (optional): by default is false
- description (optional)

Comments:

This is important not only for the @Action data, but also for execution.

Inputs give an operation or flow the data needed to act upon. Each input is mapped to a variable. You can create an input for a flow, operation, or step.

In Studio, inputs can be:

- Set to a specific value
- Obtained from information gathered by another step
- Entered by the person running the flow, at the start of the flow

See the HPE OO 10 Studio Authoring Guide for more information and see "Passing Arguments to @Actions" for details on the execution functionality.

Output

The `@com.hp.oo.sdk.content.annotations.Output` specifies an output for an action.

Attributes:

- value: the name of the output
- description (optional)

Comments:

In order for the operation in Studio to have multiple outputs, the @Action itself has to declare them. Assigning values to multiple outputs can be achieved by creating an @Action whose return value is a Map<String, String>.

In order for the operation in studio to have only one output, the @Action itself has to declare it in the return value, and use the `SINGLE_RESULT_KEY` for binding.

The output is the data produced by an operation or flow. For example, success code, output string, error string, or failure message.

In Studio, the different kinds of operation outputs include:

- Raw result: the entire returned data (return code, data output, and error string).
- The primary and other outputs, which are portions of the raw result.

Response

The `@com.hp.oo.sdk.content.annotations.Action` annotation specifies a possible response of an action.

Attributes:

- `text`: the text displayed by each response transition
- `field`: the field to evaluate
- `value`: the expected value in the field
- `description`: (optional)
- `isDefault`: Indicates whether this is the default response. The default value is false. Only one response in a @Action can have this set to true.
- `mathType` : The type of matcher to activate against the value. For example if we defined (field = fieldName, value = 0, matchType = COMPARE_GREATER) this means that this response will be chosen if the field fieldName will have a value greater than 0.
- `responseType`: The type of the response (Success, Failure, Diagnosed, No_Action or Resolve_ By_Name).
- `isOnFail`: Indicates whether this is the On-Fail response. The default value is false. Only one response in a @Action can have this set to true.
- `ruleDefined`: Indicates whether or not this response has a rule defined. Responses that have no rules defined can be used as the default response. There should be only one response without a rule defined in a single @Action.

Comments:

A response is the possible outcome of an operation or flow. The response contains a single rule: field matches value. See the HPE OO 10 Studio Authoring Guide for more information.

@Action Data Definition Example

```
@Action(value = "aflPing",
        description = "perform a dummy ping",
        outputs = {@Output(value = RETURN_RESULT, description ="returnResult
description"),
                   @Output(RETURN_CODE),
                   @Output("packetsSent"),
                   @Output("packetsReceived"),
                   @Output("percentagePacketsLost"),
                   @Output("transmissionTimeMin"),
                   @Output("transmissionTimeMax"),
                   @Output("transmissionTimeAvg")},
        responses = {@Response(text = "success", field = RETURN_CODE, value =
PASSED),
                     @Response(text = "failure", field = RETURN_CODE, value =
FAILED)})

public Map<String, String> doPing(
   @Param(value = "targetHost",
         required = true,
         encrypted = false,
         description = "the host to ping") String targetHost,
   @Param("packetCount") String packetCount,
   @Param("packetSize") String packetSize) { …
}
```

Testing Extensions

As an @Action is a simple Java method, it is possible to test it using standard Java test tools such as JUnit, leveraging the normal lifecycle phases of a Maven project.

As the @Action itself is a regular method, it does not require invoking any HPE Operations Orchestration components. The invocation can be a direct Java method invocation in the test case.

Testing Extensions as Part of the Project Build

Once they are packaged into a plugin, you can invoke extensions from the command line for test purposes. The following is an @Action example:

```
public class TestActions {
    @Action
    public int sum(@Param("op1") int x, @Param("op2") int y){
        return x+y;
    }
}
```

Suppose the TestActions class is in a plugin with the following groupId, artifactId and version (GAV): com.mycompany:my-actions:1.0

You can invoke the sum @Action from the command line as follows:

```
mvn com.mycompany:my-actions:1.0:execute -Daction=sum -Dop1=1 -Dop2=3 -X
```

The result of this command is a long trace. The -X option is required to see log messages. Towards the end of the trace you can see:

[DEBUG] Configuring mojo 'com.mycompany:my-actions:1.0::execute' with basic configurator --> [DEBUG] (f) actionName = sum [DEBUG] (f) session = org.apache.maven.execution.MavenSession@21cfa61c [DEBUG] -- end configuration -- [DEBUG] Action result: action result = 4

References

- Maven Getting Started Guide: https://maven.apache.org/guides/getting-started/index.html
- Installing Apache Maven: https://maven.apache.org/install.html
- Using Mirrors for Repositories: https://maven.apache.org/guides/mini/guide-mirror-settings.html
- CloudSlang Tutorial: http://cloudslang-docs.readthedocs.io/en/v1.0/section_tutorial.html