HP Operations Orchestration

Best Practices for Action Development

Document Release Date: October 2015
Software Release Date: October 2015

Legal Notices

Warranty

The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

The information contained herein is subject to change without notice.

Restricted Rights Legend

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notice

Trademark Notices

Adobe™ is a trademark of Adobe Systems Incorporated.

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation. UNIX® is a registered trademark of The Open Group.

This product includes an interface of the 'zlib' general purpose compression library, which is Copyright © 1995-2002 Jean-loup Gailly and Mark Adler.

Documentation Updates

The title page of this document contains the following identifying information:

- Software Version number, which indicates the software version.
- Document Release Date, which changes each time the document is updated.
- Software Release Date, which indicates the release date of this version of the software.

To check for recent updates or to verify that you are using the most recent edition of a document, go to: **http://h20230.www2.hp.com/selfsolve/manuals**

This site requires that you register for an HP Passport and sign in. To register for an HP Passport ID, go to: **http://h20229.www2.hp.com/passport-registration.html**

Or click the **New users - please register** link 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 HP sales representative for details.

Support

Visit the HP Software Support Online web site at: https://softwaresupport.hp.com/
This web site provides contact information and details about the products, services, and support that HP Software offers.
HP Software online support provides customer self-solve capabilities. It provides a fast and efficient way to access interactive technical support tools needed to manage your business. As a valued support customer, you can benefit by using the support web site to:

- Search for knowledge documents of interest
- Submit and track support cases and enhancement requests
- Download software patches
- Manage support contracts
- Look up HP support contacts
- Review information about available services
- Enter into discussions with other software customers
- Research and register for software training

Most of the support areas require that you register as an HP Passport user and sign in. Many also require a support contract. To register for an HP Passport ID, go to:
http://h20229.www2.hp.com/passport-registration.html
To find more information about access levels, go to:
http://h20230.www2.hp.com/new_access_levels.jsp
HP Software Solutions Now accesses the HPSW Solution and Integration Portal Web site. This site enables you to explore HP Product Solutions to meet your business needs, includes a full list of Integrations between HP Products, as well as a listing of ITIL Processes. The URL for this Web site is
http://h20230.www2.hp.com/sc/solutions/index.jsp

Contents

Introduction

This document provides best practices and guidelines for OO Action Development. OO offers an SDK with interfaces and classes which help achieve many of the practices described throughout the document.

Prerequisites

In order to properly comprehend all the topics discussed in this document, it is recommended that you read the following documents:

- Action Developers Guide
- Concepts Guide
- Best Practices for Content Authoring
- Introduction to Dependency Mechanism

Links to these documents are listed in the References section at the end of this document.

Audience

This document is suitable for OO Action Developers. You can find more information about the persona of this user in the *Concepts Guide*. This document is designed to help OO Action Developers follow a standard methodology of developing, testing, packaging and delivering OO Actions.

This document covers best practices that apply to OO Action Development, including practices for structuring Maven projects and modules, developing Actions, handling exceptions, unit testing and working with 3rd party products APIs. These practices are necessary in order to have modular and maintainable components which transpose in robust and cohesive plugins.

Best Practices for structuring Maven Projects and Modules

Each OO 10.x Content Pack is based on one or more Maven projects containing Maven Plugins. The recommended version of Maven to use is 3.2.1.

The best practice used when building Content Packs is called *Maven Bill of Materials* (BOM), where there is a root Maven project (pom.xml) that defines common dependencies, versions, properties, etc. and the other Maven projects depend on the root Maven Project. A BOM dependency model keeps track of version numbers and ensures that all dependencies (both direct and transitive) are at the same version.

Get familiar with the Maven concepts explained in the "Introduction to dependency mechanism" reference at the end of this document before continuing. Also, it is strongly suggested you follow the steps in the *Action Developers Guide* section "Developing Plugins" in order to create a plugin using the OO archetype and then the Maven project itself. This will help you better understand the following sections of this document, and you will also have an example of a Maven project as reference.

The Maven project generated using the OO archetype will contain the following Maven items:

- Parent Maven Project - A Maven module that contains all the necessary Maven sub-modules and defines properties and dependencies used in the sub-modules.
- Content Pack Maven Module - A Maven module containing resource bundles and XMLs of flows and operations. All these resources are packaged inside the final Content Pack.
- Plugin Maven Module - A Maven module that contains the Actions. When this project is built with Maven, the code inside is compiled and the resulting JAR file can be used for creating operations from the Actions inside.

Parent Project

The parent project **pom.xml** should list the child modules and should contain the common set of third party dependencies all the plugins use, common Maven properties, Source Control Management System configurations and the common Maven plugins definitions used for building the Maven modules.

Naming conventions

For the naming conventions listed throughout this document, it is recommended to use lowercase characters (unless otherwise specified). Furthermore, all the *ArtifactId* values must represent valid Maven id patterns.

1. *GroupId*: com.<company>[.department>].oo.content.<project|product>
   `Example: com.acme.oo.content.f5`
2. *ArtifactId*: oo-<project | product | use case>
   `Example: oo-f5`

3. *Version*: **Major.Minor.Micro**
   Example: 1.0.2-SNAPSHOT - Development version
            1.0.2   - Release version

Try to keep the parent *pom.xml* as small as possible, with only the common properties used in all your sub modules of the content pack, as seen in the skeleton below:

```
Example:

<properties>
      <scm_user>${username}</scm_user>
      <scm_passwd>${password}</scm_passwd>
<!-- Platform dependencies -->
      <oo-sdk.group>com.hp.oo.sdk</oo-sdk.group>
      <oo-sdk.version>1.0</oo-sdk.version>
<!-- Content dependencies -->
       <oo-thirdparty.version>1.0.1</oo-thirdparty.version>
      <content.groupId>com.hp.oo.content</content.groupId>
<!-- Maven properties -->
      <compiler.source.version>1.7</compiler.source.version>
      <compiler.target.version>1.7</compiler.target.version>
      <project.build.sourceEncoding>UTF-8
      </project.build.sourceEncoding>
</properties>
```

Using the *Introduction to Dependency Mechanism* reference at the end of this document, familiarize yourself with the way Maven uses dependency scopes to reduce the transitivity of a dependency.

Use the *compilerVersion* property to specify the JDK version of the compiler that the plugin will use.

A complete list of Maven properties which can be added to the pom.xml can be found in the *Apache Maven Site Plugin*.

Content Pack Maven Module

Naming conventions

1. *GroupId*: same as parent project
2. *ArtifactId*: *oo-<projectName>-cp*
3. *Version*: same as parent project

The *ArtifactId* value represents the name of the jar file that will be created based on the Maven Module. Because of this, in addition to representing a valid Maven id pattern, the ArtifactId must also represent a valid name for an OO CP.

```
Example:
       <parent>
             <artifactId>oo-base</artifactId>
             <groupId>com.hp.oo</groupId>
             <version>1.2.0-SNAPSHOT</version>
       </parent>
       <artifactId>oo-base-cp</artifactId>
```

This module represents a valid OO project that contains the following items:

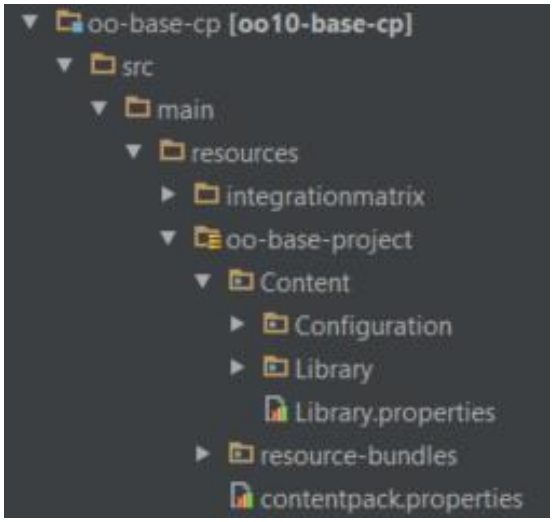
- *Content* folder with 2 subfolders:
  - Configuration folder containing all configuration items XMLs
  - Library folder containing the operations and flows XMLs.

- *resource-bundles* - containing the translation properties files*: cp.properties, cp_en_US.properties, etc.*

- *contentpack.properties* - containing the following properties for the content pack :

  - *name* (mandatory)
  - *uuid* (mandatory)
  - *version* (mandatory)
  - publisher (recommended to be supplied)
  - description
  - date of creation

  Example:



Plugin Maven Module

Each operation in the plugin module points to a specific *groupId*, *artifactId*, *version*, and @Action name.

Naming conventions

1. *GroupId*: same as parent project
2. *ArtifactId*: *oo-<projectName>[-<component|module>]-plugin*
3. *Version*: same as parent project

The best practice is to inherit (in the module) the *Version* directly from parent, see the example below.

Having sub-modules versions match the parent version is a Maven practice that helps improve the consistency of the Maven project and increases the ability and easiness to keep track of what version of each plugin is included in which project.

Example:

```
<parent>
        <artifactId>oo-acme</artifactId>
        <groupId>com.acme.oo</groupId>
        <version>1.2.0-SNAPSHOT</version>
</parent>
<artifactId>oo-databases-acme-sql-plugin</artifactId>
<packaging>maven-plugin</packaging>
```

The plugin *oo-action-plugin-maven-plugin* (from the OO SDK) *needs* to be added to the build of each plugin module in order to generate the actual plugin during the build.

```
<plugin>
    <groupId>${sdk.group}</groupId>
    <artifactId>oo-action-plugin-maven-plugin</artifactId>
    <version>${oo-sdk.version}</version>
    <executions>
        …
    </executions>
</plugin>
```

Make sure that all Content Pack Maven projects are independent of each other. For every Content Pack that is going to be released, each plugin module should not be dependent of another plugin from a different content pack. If such a scenario is unavoidable and the dependency cannot be removed, create a new, independent maven module and extract the common code, then reference it from both plugins (such a plugin is referred to as a Commons Maven module).

Action Name

Should be the same as the name of the operation that will be created based on it.
Use the following naming conventions related to Action names:

- Naming must be consistent between multiple content entities.
- Use the *<Verb> <Noun>* name format.
- Where possible, use *Create, Read, Update*, and *Delete, Get, or Set* verbs for *<Verb>*.

Examples: Create Snapshot, Get Image Details

3rd party dependency management

Make sure that your build components are stable (all unit and integration tests pass).
Use the *dependencyManagement* section in the project pom in order to control the dependency versions.

All the plugin dependencies should be defined in the plugin module pom.xml. The dependencies listed in the plugin pom.xml together with their own dependencies (transitive dependencies) form the plugin *classpath*.

If a dependency is defined in the parent Content Pack Maven project, it should be defined *without* a version in the Maven plugin module, unless it *needs to override the version* used in the parent.

Example:

Parent pom example:

```
<dependencyManagement>
    <dependencies>
      <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>1.7.1</version>
        </dependency>
        <dependency>
            <groupId>com.googlecode.json-simple</groupId>
            <artifactId>json-simple</artifactId>
            <version>1.1</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Plugin pom example:

```
<dependency>
      <groupId>com.google.code.gson</groupId>
      <artifactId>gson</artifactId>
</dependency>
<dependency>
      <groupId>com.googlecode.json-simple</groupId>
      <artifactId>json-simple</artifactId>
      <version>1.2</version>
</dependency>
```

Following is the order of how the Maven modules should be built, and should include inside the parent project's pom:

- Commons Maven module (in case there is one)
- Plugin Maven modules
- Content Pack Maven module

Complete example of parent pom.xml:

```
<parent>
    <artifactId>acme-parent</artifactId>
    <groupId>com.acme.oo</groupId>
    <version>1.0.21</version>
</parent>
<artifactId>oo-acme</artifactId>
<version>1.1.0-SNAPSHOT</version>
<packaging>pom</packaging>
<modules>
    <module>oo-acme-plugin</module>
    <module>oo-acme-cp</module>
</modules>
<scm>
  <connection>
  scm:git:ssh://git@127.0.0.1:7999/project/oo-acme.git
 </connection>
  <developerConnection>
  scm:git:ssh://git@127.0.0.1:7999/project/oo-acme.git
  </developerConnection>
 <url>
  http://127.0.0.1:7990/projects/CSTD/repos/oo-hp-solutions
  </url>
  <tag>master</tag>
</scm>
<properties>
   <content.version>1.0.1</content.version>
    <action-plugin.goal>generate-plugin</action-plugin.goal>
    <oo-sdk.group>com.hp.oo.sdk</oo-sdk.group>
    <oo-sdk.version>10.20.6</oo-sdk.version>
  <argLine>
    -XX:-UseSplitVerifier -Xmx2024m -XX:MaxPermSize=512m
   </argLine>
</properties>
```

```
<dependencyManagement>
    <dependencies>
       <dependency>
          <groupId>${oo-sdk.group}</groupId>
          <artifactId>oo-action-plugin</artifactId>
          <version>${oo-sdk.version}</version>
       </dependency>
       <dependency>
          <groupId>${oo-sdk.group}</groupId>
          <artifactId>oo-sdk</artifactId>
          <version>${oo-sdk.version}</version>
       </dependency>
       <dependency>
          <groupId>com.google.code.gson</groupId>
          <artifactId>gson</artifactId>
          <version>1.7.1</version>
       </dependency>
 <!-- testing dependencies -->
       <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>4.10</version>
          <scope>test</scope>
       </dependency>
    </dependencies>
</dependencyManagement>
<build>
  <pluginManagement>
    <plugins>
      <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-release-plugin</artifactId>
       <version>2.5.2</version>
      </plugin>
       <plugin>
          <groupId>${oo-sdk.group}</groupId>
          <artifactId>oo-action-plugin-maven-plugin</artifactId>
          <version>${oo-sdk.version}</version>
```

```
          <executions>
             <execution>
               <id>generate action plugin</id>
               <phase>process-sources</phase>
                 <goals>
                    <goal>${action-plugin.goal}</goal>
                 </goals>
              </execution>
          </executions>
        </plugin>
        <plugin>
           <groupId>${oo-sdk.group}</groupId>
           <artifactId>oo-contentpack-maven-plugin</artifactId>
           <version>${oo-sdk.version}</version>
           <executions>
             <execution>
               <id>generate lib folder</id>
               <phase>generate-resources</phase>
               <goals>
                 <goal>generate-contentpack</goal>
               </goals>
             </execution>
           </executions>
         </plugin>
      </plugins>
    </pluginManagement>
  </build>
```

Plugin pom example:

```
<parent>
    <artifactId>oo-acme</artifactId>
    <groupId>com.acme.oo</groupId>
    <version>1.1.0-SNAPSHOT</version>
</parent>

<artifactId>oo-databases-acme-sql-plugin</artifactId>
<packaging>maven-plugin</packaging>

<properties>
    <action-plugin.goal>generate-action-plugin
    </action-plugin.goal>
    <maven.deploy.skip>true</maven.deploy.skip>
</properties>
<dependencies>
    <dependency>
        <groupId>com.googlecode.json-simple</groupId>
        <artifactId>json-simple</artifactId>
    </dependency>

    <!-- SDK dependencies -->
    <dependency>
        <groupId>${oo-sdk.group}</groupId>
        <artifactId>oo-sdk</artifactId>
    </dependency>
    <dependency>
        <groupId>${oo-sdk.group}</groupId>
        <artifactId>oo-action-plugin</artifactId>
    </dependency>

    <!-- testing dependencies -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <scope>test</scope>
    </dependency>
```

```
    </dependencies>

    <build>
        <plugins>
          <plugin>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven-compiler-plugin</artifactId>
          </plugin>
          <plugin>
            <groupId>${oo-sdk.group}</groupId>
            <artifactId>oo-action-plugin-maven-plugin</artifactId>
             <version>${oo-sdk.version}</version>
          </plugin>
        </plugins>
    </build>
```

Content Pack Maven Module Pom example:

```
<parent>
<artifactId>oo-acme</artifactId>
  <groupId>com.acme.oo</groupId>
  <version>1.1.0-SNAPSHOT</version>
</parent>
<artifactId>oo10-acme-cp</artifactId>
<properties>
  <cpname>Acme CP</cpname>
  <transform.xmls.phase>process-sources</transform.xmls.phase>
  <sign.cp.phase>verify</sign.cp.phase>
  <current.release.version>1.1.0</current.release.version>
</properties>
<build>
  <!--  build plugins and resources -->
</build>
```

General Best Practices for Actions

HP Operations Orchestration 10.x can be extended programmatically by creating Actions and introducing them as custom content. This requires using the Operations Orchestration 10.x SDK and its components and means using the Java programming language to develop new operations in OO.

OO 10.x Actions are packaged in Maven Plugins Modules, described in the previous section. A Maven Plugin contains one or more Actions and references to all required dependencies. Actions are defined using the @Action annotation (an interface in the OO 10.x SDK), and this annotation can only be applied to methods.

The recommended practice is to have at most one @Action annotated method in any Java class. This enables the single responsibility principle with regards to Java classes and makes the code more organized and maintainable.

The following best practices apply to Action inputs, outputs and responses and the concepts used are described in the *OO Concepts Guide* ("HP OO Entities" section).

General Best practices for @Action interface fields

Every @Action annotation should define the following fields:

- *name* - represents the actual name of the Action. It should represent the name of the operation that will be created based on it. Using this field allows defining inputs and responses fields, and increases code readability

- *outputs* - represents the outputs (results) of the Action. It is specified through a curly braced enclosed, comma-separated list of *@Output* annotations. The *@Output* annotations must include at least the *value* field (described in a section below)

- *responses* - is specified through a curly braced enclosed, comma-separated list of *@Response* annotations that must include at least the *text, field and value* fields. These fields represent the name of the response, the name of the result to be checked and the value of the result for which the particular response will be active (described below).

Best practices for @Action parameters

@Action parameters correspond to operation inputs. The recommendation is that each @Action parameter and its corresponding input in the operation have the same name.

Every @Action annotated method should define the Action Parameters using the *@Param* annotation:

- Use the constants provided in the SDK: *com.hp.oo.sdk.content.constants.InputNames* as much as possible.
- The *@Param* annotation must include at least the *value*, *required* and *encrypted* fields.
- The *value* field represents the name of the Action input, *required* is a boolean flag that represents whether or not the input is required, and *encrypted* is a boolean flag that represents whether or not the input should be encrypted.

- The *required* and *encrypted* fields should be set in the @Action code instead of relying on editing the operations, because the behavior of the Action should be encapsulated in the Action itself.

For example, if a certain input is required for an Action to work, then the operation created based on that Action must already contain this condition after creation, and not need to explicitly specify it. Also, Action parameters that are specified as encrypted should also be marked as encrypted in the operation (this happens as a default, but it can be changed) in order to keep the intended behavior of the Action in the operation.

The signature of an @Action method can become quite long so it is suggested to follow an indentation scheme where there is one parameter (*@Param*) on each line (improves code readability).

```
(@Param(value = "volumeId") String volumeId,
 @Param(value = "attribute") String attribute,
 @Param(value = "accessKey") String accessKey,
 @Param(value = "accessKeyId", encrypted = true) String
 accessKeyId,
 @Param(value = "proxyHost") String proxyHost,
 @Param(value = "proxyPort") String proxyPort,
 @Param(value = "proxyUsername") String proxyUsername,
 @Param(value = "proxyPassword", encrypted = true ) String
 proxyPassword
 )
```

Best practices for @Action outputs

An operation should have at least three outputs (all of them available in the *OutputNames* class in the SDK):

- *returnResult* - should contain the main result of the operation. It contains the output of the execution for the success scenario or the error message in case of a failure.
- *returnCode* - used to establish the response of the operation.
- *exception* - should contain the exception stack trace in case of failure.

Example:

```
outputs = {
            @Output(OutputNames.RETURN_CODE),
            @Output(OutputNames.RETURN_RESULT),
            @Output(OutputNames.EXCEPTION),
           @Output("sessionKey")
}
```

Use the SDK constant class: *com.hp.oo.sdk.content.constants.OutputNames* as much as possible. Use the constants from *com.hp.oo.sdk.content.constants.ReturnCodes* for *returnCode* output values.

The order of the outputs mentioned in the @Action method is inherited when the operation is created, thus it is recommended to follow an ordering rule, such as an alphabetical order or an order based on importance of the outputs.

Best practices for @Action responses

Use the *field* attribute of the annotation in order to create a rule to determine the response type. The *field* value is one of the outputs of the operation and will be matched with the rule defined by the *matchType* attribute against the *value* attribute. If the match succeeds, the *responseType* attribute will be the actual response in that particular situation.

The default response should be failure. This way, an incomplete Action execution shows as a failure during flow debugging and points the author to the problem before the flow goes into production. Use the *isDefault* attribute of the @Response annotation to set the default operation response.

In a **success** scenario, *field* is *returnCode* and it is equal to the *RETURN_CODE_SUCCESS* constant from the *ReturnCodes* Class in the SDK. The *matchType* rule is *COMPARE_EQUAL* and the *responseType* is *RESOLVED.* In the @Action method, populate the *RETURN_CODE* Output Name with the *RETURN_CODE_SUCCESS* value on the success scenario.

```
Example:
@Response(isDefault = true, text = ResponseNames.SUCCESS, field =
OutputNames.RETURN_CODE, value = ReturnCodes.RETURN_CODE_SUCCESS,
matchType = MatchType.COMPARE_EQUAL, responseType =
ResponseType.RESOLVED)
```

In a **failure** scenario, the *returnCode* field is equal to *RETURN_CODE_ERROR;* the response type of the operation is *ERROR.* In the @Action, populate the *RETURN_CODE* Output with *RETURN_CODE_ERROR* on the failure scenario.

```
Example:
@Response(isOnFail = true, text = ResponseNames.FAILURE, field =
OutputNames.RETURN_CODE, value = ReturnCodes.RETURN_CODE_FAILURE,
matchType = MatchType.COMPARE_EQUAL, responseType =
ResponseType.ERROR)
```

Use the SDK constant class: *com.hp.oo.sdk.content.constants.ResponseNames* as much as possible.

Best practices for designing the @Action code

Write as little logic as possible in the @Action itself. Typically the logic should be limited to passing the @Action parameters (inputs of the operation) to a service method (which is responsible for the business logic) and populating the result map of the @Action with the results of the service method invocation.

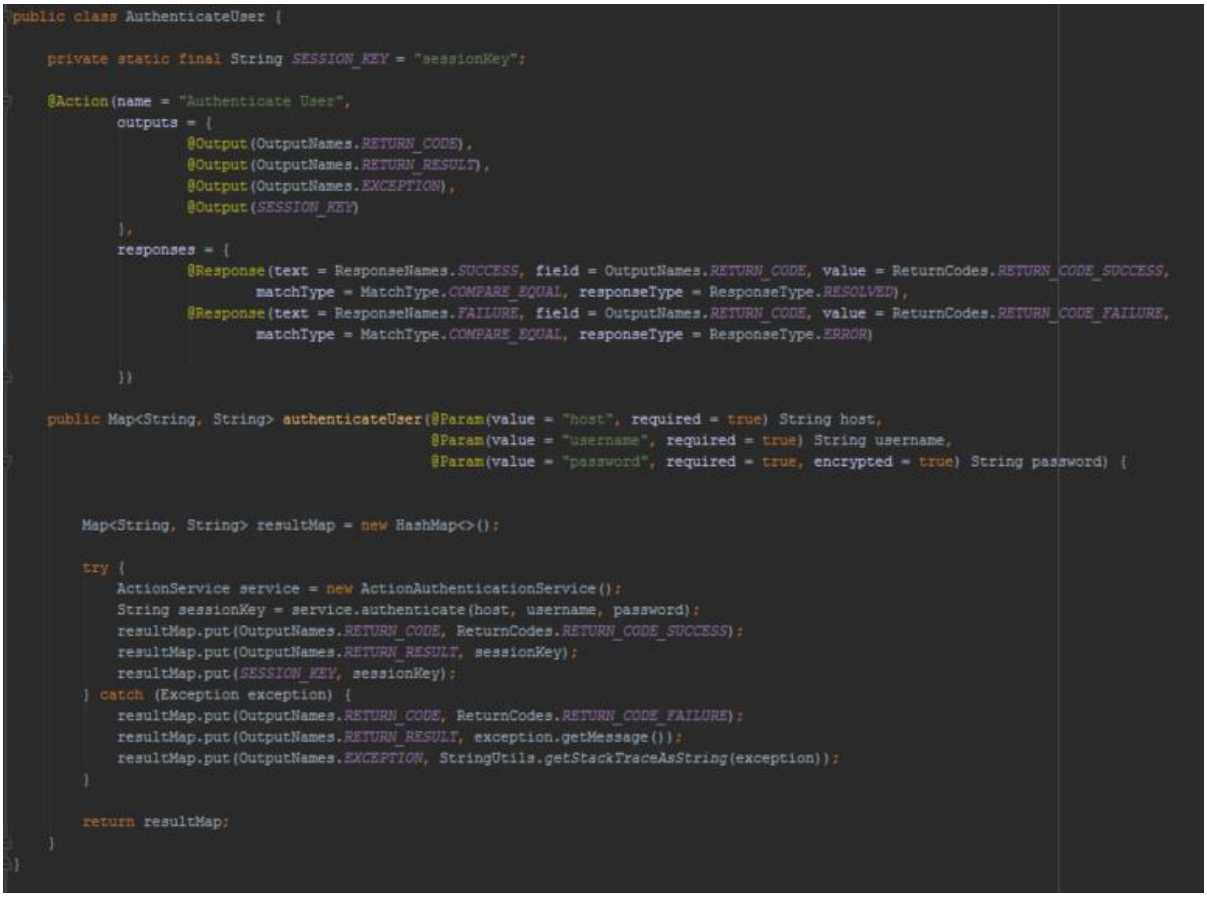
```
Example:

Map params = createMapParams(input1, input2, <…>  inputn);
ResponseWrapper response = service.describeEntityAttribute(params);
resultMap.put(OutputNames.RETURN_RESULT, response.getVolumeId());
```

Use third party libraries for common generic tasks instead or your own implementations.

Avoid static members because a @Action can be instantiated in two steps in a flow and could be executed on different workers. This makes static members unreliable for data transfer.

One best practice is for the client (which in this case is the Action itself) to not be required to know how to construct the services, but only to know about the interfaces of the services, which define how the client may use the services.This enables separating the responsibilities of use and construction. Also, it is recommended to use as much as possible design patterns when writing the code. The following example shows how an @Action should appear:



```
public class AuthenticateUser {

    private static final String SESSION_KEY = "sessionKey";

    @Action(name = "Authenticate User",
            outputs = {
                    @Output(OutputNames.RETURN_CODE),
                    @Output(OutputNames.RETURN_RESULT),
                    @Output(OutputNames.EXCEPTION),
                    @Output(SESSION_KEY)
            },
            responses = {
                    @Response(text = ResponseNames.SUCCESS, field = OutputNames.RETURN_CODE, value = ReturnCodes.RETURN_CODE_SUCCESS,
                            matchType = MatchType.COMPARE_EQUAL, responseType = ResponseType.RESOLVED),
                    @Response(text = ResponseNames.FAILURE, field = OutputNames.RETURN_CODE, value = ReturnCodes.RETURN_CODE_FAILURE,
                            matchType = MatchType.COMPARE_EQUAL, responseType = ResponseType.ERROR)

            })

    public Map<String, String> authenticateUser(@Param(value = "host", required = true) String host,
                                                @Param(value = "username", required = true) String username,
                                                @Param(value = "password", required = true, encrypted = true) String password) {


        Map<String, String> resultMap = new HashMap<>();

        try {
            ActionService service = new ActionAuthenticationService();
            String sessionKey = service.authenticate(host, username, password);
            resultMap.put(OutputNames.RETURN_CODE, ReturnCodes.RETURN_CODE_SUCCESS);
            resultMap.put(OutputNames.RETURN_RESULT, sessionKey);
            resultMap.put(SESSION_KEY, sessionKey);
        } catch (Exception exception) {
            resultMap.put(OutputNames.RETURN_CODE, ReturnCodes.RETURN_CODE_FAILURE);
            resultMap.put(OutputNames.RETURN_RESULT, exception.getMessage());
            resultMap.put(OutputNames.EXCEPTION, StringUtils.getStackTraceAsString(exception));
        }

        return resultMap;
    }
}
```

Context @Actions

A dynamic map of parameters can be passed in a @Action instead of passing specific @*Param* annotated parameters.
Context Actions are useful in situations when the actual inputs of the operations need to be customizable for each instance of the operation (dynamically defined inputs at step level).

This can be achieved by replacing the parameters of the @Action method with a Map parameter.

```
      @Action (…)
      public Map<String, String> authenticateUser(Map<String,
String> inputs){}
```

Such a @Action should be placed in a plugin which follows a naming convention of:
oo-<name>-context-plugin
By following this naming convention, the user would immediately gain the information that this plugin contains context based Actions. This type of plugin is not compatible with the *oo-action-plugin-maven-plugin* because it requires the *oo-context-action-plugin* (from the SDK) to be generated.

Example (pom.xml):

```
<artifactId>oo-base-context-plugin</artifactId>
<packaging>maven-plugin</packaging>
<properties>

  <action-plugin.goal>
      generate-context-action-plugin
 </action-plugin.goal>
</properties>

<dependencies>
      <dependency>
         <groupId>${oo-sdk.group}</groupId>
         <artifactId>oo-context-action-plugin</artifactId>
      </dependency>
</dependencies>

<build>
   <plugins>
       <plugin>
           <groupId>${oo-sdk.group}</groupId>
           <artifactId>oo-action-plugin-maven-plugin</artifactId>
        </plugin>
   </plugins>
</build>
```

Best Practices for Exception Handling

This section describes practices and guidelines with regards to exception handling inside @Actions.

- *Only change exception messages* if semantic value must be added to the original exception that the integration product returns.
- *Use standard Java exceptions as much as possible*. `java.net.SocketException`, `java.util.concurrent.TimeoutException`
- *Catch specific exceptions* in @Actions and set different return codes for each specific exception.
- Keep the exception messages returned by the systems OO is integrating with, do not replace or wrap these messages with your own error messages. This way will allow a certain level of consistency between error messages returned by operations integrating with the same system.
- Do not rely on exception messages which are thrown by integrating systems because they sometimes change from one version to another, breaking backward compatibility.
- *Throw exceptions early*: an exception should be thrown as soon as possible from the service classes

Exceptions should be caught in the code of the @Action, where it can be handled properly. Usually handling means setting the specific return code and error message on the operation output results.

Example:

```
import com.hp.oo.sdk.content.constants.ReturnCodes;
//…
try {
  customService = sshLogOff(parameterList);
}
catch (SSHException) {
  resultMap.put(OutputNames.RETURN_CODE,
                ReturnCodes.RETURN_CODE_FAILURE);
}
```

When an exception is caught, the *exception* result should be populated with the stack trace and the *returnResult* result should be populated with the exception message.

Example:

```
catch (SpecificException1 e) {
  setReturnCode();
  setException(exception, e.getStackTrace ());
  setFailureMessage(returnResult, e.getMessage());
}
```

Best Practices for Date and Time processing

The preferred date and time format is the one provided by the API of the integration product.

If there is a need to convert between different date formats, the recommended way is to do that at authoring time, using *Date and Time* operations provided in Base Content Pack or recommended libraries for date and time manipulation such as *Java's API* (*java.util*).

Best Practices for Timeouts

Use the following types of timeout *inputs***,** when there is a need for each type of behavior:

- *connectTimeout* – represents the timeout to connect to the target server or URL.
- *socketTimeout* - represents the timeout for waiting for the data. In other words, the maximum period of inactivity between two consecutive data packets. During the execution of an operation, there can be multiple packets sent and sockets opened between the source and the target.
- *executionTimeout* (or simply *timeout*) – the time for the actual action to finish the execution.

Types of timeout *outputs*:

- *TimedOut* - OO 10.x exposes the *TimedOut* output. When a timeout situation occurs, the *TimedOut* result should be set to "true", otherwise it should be "false".

- *exception* – In case of a timeout, this output should contain the execution stack trace with one of the *standard Java timeout exceptions: ConnectionTimeoutException*, *TimeoutException*, *SocketTimeoutException.*

Currently, there is no global execution timeout in OO 10.x.

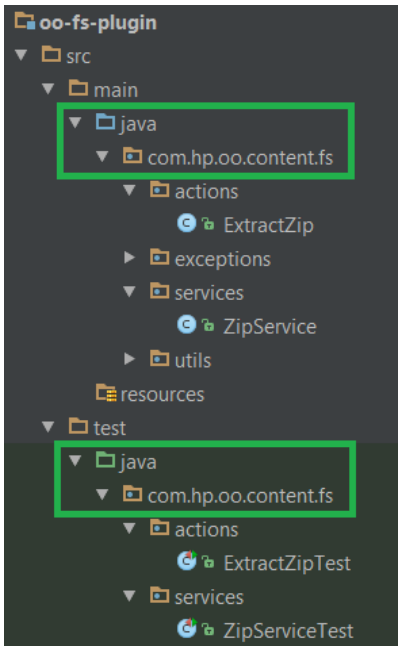
Best Practices for Unit Testing

Apply the following rules when writing unit tests for the @Actions.

Create a consistent package structure between the tests and the classes under test

If you want to develop a unit test for the class *ExtractZip* located in the *com.hp.oo.content.fs* folder, under the *test* folder it is recommended to create the same package structure like the one the *ExtractZip* class is located in.

This is the Apache Software Foundation's standard directory structure, and it enables the user to transition more easily from a plugin to another, while also be able to leverage modern IDE features that calculate unit test code coverage.



Mock server calls

In your test methods, don't use real calls to servers. If you want to test a method like *executeHttpRequest*(Url url), don't perform a real HTTP request, instead use one of the industry established mocking frameworks (e.g. Mockito, EasyMock, Powermock) to mock the request and create isolation unit tests using a framework like JUnit or TestNG.

Best Practices for working with 3rd party APIs

Typically, the most common types of APIs used in the industry are REST and SOAP, with some product also exposing Java based APIs or SSH APIs. When choosing the right API to use there are several factors that needs to be considered:

- The API is *GA*-ed and officially supported (the exposed API used is not in Beta)
- Apache 2, LGPL licenses are preferred over GPL

- *REST* APIs are preferred over *SOAP*, Java and SSH APIs (in this order).

REST APIs advantages over SOAP are that they rely upon the HTTP standard, they are lightweight, format-agnostic (can use any one of the XML, JSON, HTML formats) and have better support for API versioning. All these features tremendously help during the development process of the integration.

Avoid tightly coupling @Actions classes to integration specific APIs.This enables easier backward compatibility support in the future, when trying to support a new version of a certain API that by itself broke backward compatibility. Expect this behavior from APIs you are developing integrations for and design your operations accordingly.

Consider the following scenarios and possible ways to handle them for operations that need to support multiple versions of the Integration Product that they integrate with:

- A version input could be added to all of the operations/flows. The default value for the version input would be the first version supported. In this way, existing users that use the old operations are not affected by an upgrade of content.

- A new folder containing operations for the new integration version should be created if there are major changes between the last supported version and the new version.

References

- Action Developers Guide: document under <INSTALLATION_FOLDER>/docs folder
- Apache Maven Site Plugin: https://maven.apache.org/plugins/maven-compiler-plugin/compile-mojo.html
- Introduction to Dependency Mechanism: https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html
- Best Practices for Content Authoring: https://hpln.hp.com/node/21182/attachment
- Concepts Guide: document under <INSTALLATION_FOLDER>/docs folder
- Maven plugins: https://maven.apache.org/plugins/
- Coding Styles: https://google-styleguide.googlecode.com/svn/trunk/javaguide.html
- Unit Testing Practices: http://www.oracle.com/technetwork/articles/adf/part5-083468.html

If you have any questions or feedback, then please post these on the OO community forum: http://www.hp.com/go/OOPractitionerForum