MICRO FOCUS | SERENA

Deployment Automation
Integration Guide

Trademarks

U.S. Government Rights

Part number: 6.2.1

Publication date: 2018-11-19

Table of Contents

Chapter 1: Welcome to Deployment Automation

Deployment Automation enables you to automate the deployment of application changes. Benefits include continuous delivery and DevOps automation, reduction of development costs, and increased deployment frequency without increased risk.

About This Documentation

This documentation gives information on integrating with Deployment Automation and is intended for those who will configure the integrations. This information is also included in the online Help in HTML format.

Chapter 2: Integrating with Deployment Automation

The integrations provided by Deployment Automation enable you to execute deployment related tasks through many Serena and third-party products. Most integrations with Deployment Automation are implemented through the rich set of plugins provided with the product. Plugins are used in Deployment Automation process steps. For details on the plugins, see *Deployment Automation Plugins Guide*.

For information on additional integration mechanisms, configuring integrations, and writing plugins to create integrations of your own, see the following topics.

Chapter 3: Integrating with SBM [page 11]

Chapter 8: Integrating with Source Configuration Tools [page 53]

Chapter 4: Integrating with Dimensions CM [page 21]

Chapter 5: Integrating with ChangeMan ZMF [page 27]

Chapter 6: Integrating with Nolio [page 35]

Chapter 7: Creating Custom Plugins [page 41]

Chapter 3: Integrating with SBM

Communication between SBM and Deployment Automation enables release deployment automation from SBM solutions. Integration mechanisms that enable this communication are as follows:

- **REST Grid Widgets**

  You can select Deployment Automation RESTful service data and populate SBM REST Grid widgets directly from SBM Composer using Deployment Automation Composer Mode. This communication enables the creation and linking of Deployment Automation applications and environments and the access of Deployment Automation processes for automation deployment tasks.

- **User Auto-registration**

  When that user accesses functionality in Deployment Automation through SBM, the SBM Single Sign-On (SSO) token sends the sign on information, and Deployment Automation extracts the credentials from the SSO token. Those credentials are used to register the user in Deployment Automation. See the SBM documentation for more details on SSO.

- **ALF Events**

  ALF Events are another mechanism that can be used to integrate SBM with Deployment Automation.

  For information on ALF Events that can be emitted from Deployment Automation, see the *Deployment Automation User's Guide*.

For more information on integrating with SBM Solutions, see the following topics:

- Configuring the REST Grid Widgets [page 11]
- Methods Supporting Composer Mode [page 12]
- Single Sign-On (SSO) Configuration [page 17]

Configuring the REST Grid Widgets

In SBM Composer, in a Visual Design layout REST Grid widget, you can get data directly from Deployment Automation REST services.

To configure the REST grid widget in SBM Composer:

1. In the REST grid widget, in the Configure URL dialog, provide the REST service method URL and add the `composerMode=true` request parameter.

   For example:

   ```
   http://srademo:8080/da/rest/deploy/component/all?composerMode=true
   ```

2. If the given Deployment Automation GET REST service method supports Composer Mode, a sample JSON with returned property names will appear in the **Result** tab. These do not include real data, but are the JSON structure.

3. In the **Result** tab, pick the corresponding property names to construct your REST grid widget columns.
4. When you have finished picking the property names to construct your columns, change the REST service method URL `composerMode` parameter to `false`.

   For example:

   `http://srademo:8080/da/rest/deploy/component/all?composerMode=false`
5. Turn on SSO authentication.
6. Deploy the process app.
7. Verify the information in the SBM process app's user workspace.

The REST service methods that support Composer Mode are given in the following topic.

Methods Supporting Composer Mode

Only GET methods from the `da/rest/application.wadl` file are supported by REST Grid widgets in SBM Composer, and only some of them.

Many of the methods that support Composer Mode are given in the following list. This list is expanding, so please check the Knowledgebase if you don't see the method you need in the list, or just give the method you need a try to see if it supports Composer Mode.

1. Get Application

   /rest/deploy/application/{applicationId}
2. Get Applications

   /rest/deploy/application

   /rest/deploy/application/all
3. Get Application Components

   /rest/deploy/application/{applicationId}/components
4. Get Application Environments

   /rest/deploy/application/{applicationId}/environments/{inactive}

   /rest/deploy/application/environments/forComponent/{componentParam}

   /rest/deploy/application/{applicationId}/fullEnvironments
5. Get Application Process

   /rest/deploy/applicationProcess/{applicationProcessId}/{version}
6. Get Application Processes

   /rest/deploy/applicationProcess

   /rest/deploy/application/{applicationId}/processes/{inactive}

   /rest/deploy/application/processes/forComponent/{componentParam}

   /rest/deploy/application/{applicationId}/executableProcesses

/rest/deploy/application/{applicationId}/fullProcesses

7. Get Application Process Unfilled Properties
   /rest/deploy/applicationProcess/{applicationProcessId}/unfilledProps/{onlyRequired}
8. Get Application Properties
   /rest/deploy/application/{applicationId}/applicationProperties
9. Get Component
   /rest/deploy/component/{componentId}
10. Get Components
    /rest/deploy/component
    /rest/deploy/component/all
    /rest/deploy/component/allFull
11. Get Component Versions
    /rest/deploy/component/{componentId}/versions/{inactive}
12. Get Component Properties
    /rest/deploy/component/{componentId}/componentProperties
13. Get Component Version Properties
    /rest/deploy/component/{versionId}/componentVersionProperties
14. Get Component Process
    /rest/deploy/componentProcess/{componentProcessId}/{version}
15. Get Component Processes
    /rest/deploy/component/{componentId}/processes/{inactive}
    /rest/deploy/component/{componentId}/fullProcesses/{inactive}
    /rest/deploy/component/{componentId}/processesWithVersion
    /rest/deploy/component/{componentId}/executableProcesses
16. Get Standalone Process
    /rest/process/{processId}/{version}
17. Get Standalone Processes
    /rest/process/{inactive}
18. Get Resource
    /rest/resource/resource/{resourceId}
19. Get Resources
    /rest/resource/resource
    /rest/resource/resource/tree

/rest/resource/resource/treeWithInactive

/rest/resource/resource/{resourceId}/resources

20. Get Environment

    /rest/deploy/environment/{environmentId}

21. Get Environments

    /rest/deploy/environment/all

22. Get Applications For Environment

    /rest/deploy/environment/{environmentId}/applications

23. Get Environment Properties

    /rest/deploy/environment/ {environmentId}/environmentProperties

24. Get Environment Properties For Components

    /rest/deploy/environment/ {environmentId}/componentProperties

25. Get Environment Properties For Component

    /rest/deploy/environment/{environmentId}/{componentId}/
    propertiesForComponent

26. Get Active Agents

    /rest/agent

27. Get Agent

    /rest/agent/{agentId}

28. Get All Agents

    /rest/agent/all

29. Get Agents Assignable To License

    /rest/agent/assignableToLicense/{licenseId}

30. Get Agent Resources

    /rest/agent/{agentId}/resources

31. Get Agent Pools

    /rest/agent/{agentId}/pools

32. Get Component Config Templates

    /rest/deploy/component/{componentId}/configTemplates/{active}

33. Get Component Task Definitions

    /rest/deploy/component/{componentId}/taskDefinitions/{active}

34. Get All Status Plugins

    /rest/plugin/statusPlugin

35. Get Status Plugin

    /rest/plugin/statusPlugin/{statusPluginName}

36. Get Status Plugin Version Statuses

    /rest/plugin/statusPlugin/{statusPluginName}/versionStatuses

37. Get Status Plugin Inventory Statuses

    /rest/plugin/statusPlugin/{statusPluginName}/inventoryStatuses

38. Get Application Component Process Tree

    /rest/deploy/application/{applicationId}/componentProcessTree

39. Get Application Unused Components

    /rest/deploy/application/{applicationId}/unusedComponents

40. Get Application Task Definitions

    /rest/deploy/application/{applicationId}/taskDefinitions/{active}

41. Get Snapshots

    /rest/deploy/application/{applicationId}/snapshots/{inactive}

42. Get Component Process Prop Defs

    /rest/deploy/componentProcess/{componentProcessId}/{version}/propDefs

43. Get Component Process Activity Tree

    /rest/deploy/componentProcess/{componentProcessId}/activityTree

44. Get Component Process Change Log

    /rest/deploy/componentProcess/{componentProcessId}/changelog

45. Get Application Task Definition

    /rest/task/applicationTaskDefinition/{id}

46. Get Deployment Request

    /rest/deploy/deploymentRequest/{deploymentRequestId}

47. Get Deployment Requests

    /rest/deploy/deploymentRequest/table

48. Get Deployment Request Application Process Requests

    /rest/deploy/ deploymentRequest /{deploymentRequestId}/
    applicationProcessRequests

49. Get Deployment Request Non Compliancy By Resource

    /rest/deploy/deploymentRequest/{deploymentRequestId}/
    noncompliancyByResource

50. Get Config Template

/rest/deploy/configTemplate/{componentId}/{name}/{version}

/rest/deploy/configTemplate/byRequest/{requestId}/{name}

51. Get Application Process Request

    /rest/deploy/applicationProcessRequest/{applicationProcessRequestId}

52. Get Application Process Requests

    /rest/deploy/applicationProcessRequest/table

53. Get Application Process Request Properties

    /rest/deploy/applicationProcessRequest/{applicationProcessRequestId}/properties

54. Get Application Process Request Environment Properties

    /rest/deploy/applicationProcessRequest/{applicationProcessRequestId}/
    environmentProperties

55. Get Application Process Request Versions

    /rest/deploy/applicationProcessRequest/{applicationProcessRequestId}/versions

56. Get Active Global Environments

    - /rest/deploy/globalEnvironment

57. Get All Global Environments

    - /rest/deploy/globalEnvironment/all

58. Get Active Applications For Global Environment

    - /rest/deploy/globalEnvironment/{globalEnvironmentId}/applications

59. Get All Applications For Global Environment

    - /rest/deploy/globalEnvironment/{globalEnvironmentId}/applications/all

60. Get Inactive Global Environments

    - /rest/deploy/globalEnvironment/inactive

61. Get Global Environment

    - /rest/deploy/globalEnvironment/{globalEnvironmentId}

62. Get Global Environment Properties

    - /rest/deploy/globalEnvironment/{globalEnvironmentId}/
    globalEnvironmentProperties

63. Get Global Environment Resource Mappings

    - /rest/deploy/globalEnvironment/{globalEnvironmentId}/resources

64. Get Global Environment Not Mapped Resources

    - /rest/deploy/globalEnvironment/{globalEnvironmentId}/resourcesNotMapped

65. Get Global Environment Not Mapped Resource Groups

- /rest/deploy/globalEnvironment/{globalEnvironmentId}/resourceGroupsNotMapped

Single Sign-On (SSO) Configuration

SSO enables Deployment Automation to integrate more easily with other Serena products. Login information is passed automatically through SSO so that there is no need to prompt for login credentials as information flows between products.

For details on figuring SSO, see the following topics:

- Configuring Common Tomcat for SSO [page 17]
- Creating an SSO Authentication Realm [page 19]
- Sign On Using SSO [page 19]
- Single Sign Out [page 19]

Configuring Common Tomcat for SSO

To use a typical Deployment Automation installation with SBM, you must update configuration files to enable Common Tomcat to find and use the correct SBM SSO installation.

Before you can use SSO with Deployment Automation, you must have SBM installed and SSO must be enabled. You must have the Deployment Automation server installed on the same machine as the Common Tomcat.

1. On the Deployment Automation server, stop the Common Tomcat service.
2. Navigate to the Common Tomcat `conf` directory. For example:

   ```
   C:\Program Files\Micro Focus\common\tomcat\8.5\alfssogatekeeper\conf
   ```

3. In `gatekeeper-core-config.xml`, change the following parameters as necessary to replace the host and port values. Replace the placeholder variables shown here and in the default file as `$HTTP_OR_HTTPS`, `$HOSTNAME` and `$PORT`, with either HTTP or HTTPS, and the host name and port for your SBM SSO server. The default HTTP port number for the SBM SSO server is 8085, and the default HTTPS port number for the SBM SSO server is 8243.

   ```
   <parameter name="SecurityTokenService"
   Type="xsd:anyURI">$HTTP_OR_HTTPS://$HOSTNAME:$PORT/TokenService/
   services/Trust<parameter>

   <parameter name="SecurityTokenServiceExternal"
   Type="xsd:anyURI">$HTTP_OR_HTTPS://$HOSTNAME:$PORT/TokenService/
   services/Trust</parameter>

   <parameter name="FederationServerURL"
   Type="xsd:anyURI">$HTTP_OR_HTTPS://$HOSTNAME:$PORT/ALFSSOLogin/
   login</parameter>
   ```

   For example:

```
<parameter name="SecurityTokenService" Type="xsd:anyURI">
HTTPS://myserver:8243/TokenService/services/
Trust<parameter>

<parameter name="SecurityTokenServiceExternal" Type="xsd:anyURI">
HTTPS://myserver:8243/TokenService/services/
Trust</parameter>

<parameter name="FederationServerURL" Type="xsd:anyURI">
HTTPS://myserver:8243/ALFSSOLogin/login
</parameter>
```

CAUTION:

For the gatekeeper core configuration, you use the SBM SSO HTTP or HTTPS port number. Be careful not to confuse this with the port numbers for Deployment Automation, which are by default 8080 and 8443 for HTTP and HTTPS respectively.

4. Navigate to yourDeployment Automation server profile directory. For example:

   `C:\Users\username\.microfocus\da\conf\server`

   or

   `/opt/MicroFocus/da/username/.microfocus/da`/conf/server

5. Modify the `da_config.xml` to set the ssoEnabled property to true as follows:

   ```
   <ssoConfig>
       <ssoEnabled>true</ssoEnabled>
   </ssoConfig>
   ```

6. On the Deployment Automation server, start the Common Tomcat service.

7. Verify the configuration by invoking the Deployment Automation user interface through your implementation's URL, such as `http://sdaserver:8080/da`. If when attempting to sign on, you receive the following error, you will need to update your SSO STS certificates.

   ```
   ALF SSO Gatekeeper error has occurred: Error obtaining security token.

   Detail

   Validation of WS-Federation token failed with code 40:Token issuer not
   allowed.
   ```

   See Knowledgebase item S140637 for more information.

Reconfiguring for SSO After Upgrades

If you have configured Single Sign-On (SSO) with one version of Common Tomcat and have upgraded Deployment Automation to a version that uses a different version, you must configure Common Tomcat for SSO again, including setting the parameters in the `gatekeeper-core-config.xml` file. Otherwise, the SSO login will fail.

You must set these parameters by copying over the corresponding strings from earlier version of the `gatekeeper-core-config.xml` file. Copying and replacing the entire file from the earlier Common Tomcat installation does not work.

Creating an SSO Authentication Realm

You may need to create the Single Sign-On authentication realm in Deployment Automation. This is typically created for you automatically, although may need to be created for upgrades.

To configure to use SSO:

1. Log in to Deployment Automation as an administrative user.
2. Navigate to **Administration > Security**.
3. In the selection box, select **Authentication (Users)**.
4. Click the **Create Authentication Realm** button.
5. In the **Authorization Realm** field, select `Internal Security`.
6. In the **Type** field, select `Single Sign-On`.
7. In the **User Header Name** field, enter `ALFSSOAuthNToken`.
8. Click **Save**.

Deployment Automation allows sign on and sign out through SSO.

Sign On Using SSO

Try signing on to the Deployment Automation user interface URL: `http://<host>:<port>/da/`, where *port* is the Common Tomcat HTTP port.

Instead of the default Deployment Automation login page, the SBM Single Sign-On page should appear.

Enter your user name and password to access Deployment Automation.

Single Sign Out

When you use Single Sign-On (SSO), Single Sign Out will work correctly as long as you have the Deployment Automation server and the SSO server both configured to use the same host.

Chapter 4: Integrating with Dimensions CM

You can integrate Deployment Automation with Dimensions CM through the provided Dimensions CM plugin.

The Dimensions CM plugin can be used to retrieve a list of baselines for selection from Dimensions CM that are suitable for deployment, and then deploy the target baseline using the Dimensions CM deployment functionality. The Dimensions CM plugin that enables Deployment Automation and Dimensions CM to communicate uses the Dimensions CM web services and passes predefined credentials and selection information, such as the Dimensions CM product and stream.

Note: Dimensions CM also includes an option to use Deployment Automation for deployment. For information, see the Dimensions CM documentation.

The following topics describe the runtime communication and configuration of the Dimensions CM plugin for use with Deployment Automation.

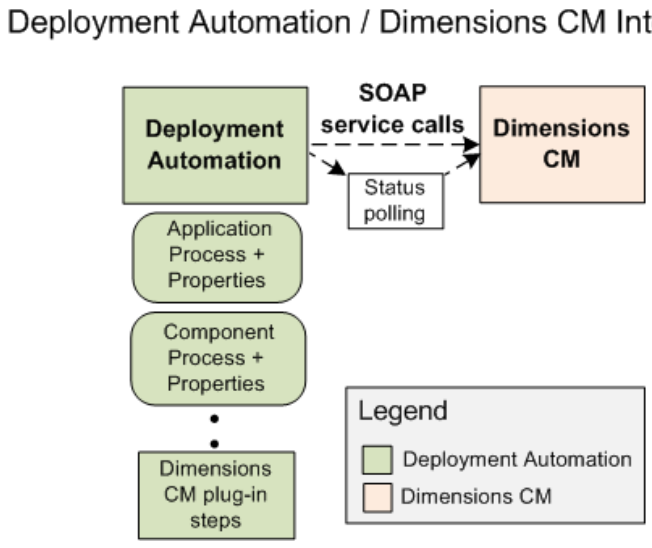
Documentation References

For more information on using plugins, including details on the plugin steps, see the *Deployment Automation Plugins Guide*.

Dimensions CM Integration Example

All of the information needed for Deployment Automation to communicate with Dimensions CM is embedded in the Deployment Automation application and component processes, which use the Dimensions CM plugin.

The flow of communication between Deployment Automation and Dimensions CM is shown in the following figure.



Dimensions CM Integration Runtime Communication

The communication between Deployment Automation and Dimensions CM proceeds as follows:

1. Deployment Automation processes are configured for the Dimensions CM processes to be executed.
2. When the Deployment Automation processes are run, they invoke the Dimensions CM processes. The Deployment Automation processes must contain all connection details for the target Dimensions CM server, product, stream, or other entities.
3. Deployment Automation requests information from Dimensions CM through SOAP service calls.
4. The activity on the Dimensions CM server is initiated and Deployment Automation polls the Dimensions CM server for the current job status.
5. Once the job status in Dimensions CM completes, either successfully or with a failure, the Deployment Automation process step that initiated the transaction completes.

Dimensions CM Plugin Installation

The Dimensions CM plugin must be extracted before it can be loaded into Deployment Automation. Extract the plugin as follows:

1. Download the plugin installation file from the Deployment Automation download location on the Support website. For example, `Dimensions_bundle_vvv.zip`, where `vvv` is the version.
2. Extract the files from the plugin bundle. It contains the plugin zip file and files needed to configure the plugin. The plugin zip file is named `DimensionsCM_vvv.zip`, where `vvv` is the version.

3. In Deployment Automation, navigate to **Administration > Automation**.
4. In the selection box, select **Plugins**.
5. Click the **Load Plugin** button.
6. Click **Choose File** and select the plugin zip file.
7. Click **Load**.
8. Configure the processes for the plugin that are required for the integration.

Configuring Dimensions CM Processes in Deployment Automation

The following topics describe how to configure the processes and properties for optimal use of the Dimensions CM plugin for use with Deployment Automation.

Importing the Dimensions CM Sample Environment

For the quickest and most reliable implementation, you should import the sample environment and application and modify the properties to suit your needs. Before you can import the sample application processes, you must first import the environment that is associated with the application.

One sample environment, UAT, is provided to use with all of the sample applications. If you have already imported the environment to use with one of the other sample applications, you should not import it again.

To import the sample UAT environment:

1. Navigate to the directory location where you downloaded the plugin bundle.
2. Extract the following JSON file if it is not already extracted:

   `Sample UAT.json`
3. If you want to change the name of the environment that will be imported, open the environment JSON file and change the name and description to whatever you want to call your environment.
4. To import the environment:
   a. In Deployment Automation, navigate to **Management > Environments**.
   b. Click the **Application Environments** button and then select **Import Environment**.
   c. Click **Choose File** and browse to the path of the `Sample UAT.json` file.

d. Click **Import**.

The environment should now be listed in your environments page, as UAT if you did not change the name, or under the name you specified when you changed it.

Importing the Sample Dimensions CM Application

There are several Deployment Automation processes necessary to create the operations needed for this plugin integration. To make it easier for you to configure your processes, an exported sample application is included in the plugin bundle.

The sample application includes all of the application and component information needed to get you started. You can import the exported file and modify the details to match your implementation. Otherwise, you must configure all of your processes and properties manually as described in the subsequent topics:

To import the sample application:

1. Navigate to the directory location where you downloaded the plugin bundle.
2. Extract the following JSON file if it is not already extracted:

   ```
   <product> Sample Application.json
   ```

3. If you changed the name and description of the sample environment that you imported, open the application JSON file and change the corresponding environment name and description to the ones you used in your environment JSON file.
4. To import the application processes:
   a. In Deployment Automation, navigate to **Management > Applications**.
   b. Click the **Application Actions** button and then select **Import Application**.
   c. Click **Choose File** and browse to the path of the JSON file.
   d. Click **Import**.

The application should now appear in the application list.

Configuring the Dimensions CM Application

An application process is used to run the component processes you need. Most of the properties that are needed for the component processes should be set at the application level, because many properties are used by more than one component process.

Tip: For the quickest implementation, import the sample environment and application and modify the properties to suit your needs.

To configure the application:

1. Create an application that will contain your properties and component processes or select an existing one. For example, DimCM Application.
2. If you imported the sample application, edit the application and change the application name and description to match your implementation's values.

3. Add properties to your application that are common to all component processes, or modify the existing imported values to match your system information. For example, the connection information property values are as follows:
   - DIMCM_DBCONNECTION: value <your DB connection name>
   - DIMCM_DBNAME value <your DB name>
   - DIMCM_SERVER value <your server name>
   - DIMCM_SERVICE_USER value <your service user name>
   - DIMCM_SERVICE_PASSWORD value <your service user password>
4. Add the following processes to your application if they have not already been imported.
   - Deploy Baseline
   - Get Baselines
   - Get Deployment Areas
   - Get Products
   - Get Projects
   - Get Projects and Streams
   - Get Stages
   - Get Streams
   - Promote Baseline
5. Add the properties to the application processes that the component processes will inherit, or change them in the imported application. Following are example properties for Deploy Baseline.
   - name DIMCM_PRODUCT, label Product, value `{applicationProcess:Get Products;displaycols:product}`
   - name DIMCM_PROJECT_NAME, label Project Name, value `{applicationProcess:Get Projects And Streams;properties:{[name:DIMCM_PRODUCT,value:DIMCM_PRODUCT]};displaycols:project_stream}`
   - name DIMCM_BASELINE_NAME, label Baseline Name, value `{applicationProcess:Get Baselines;properties:{[name:DIMCM_PRODUCT,value:DIMCM_PRODUCT]};displaycols:baseline}`
   - name DIMCM_STAGE_NAME, label Stage Name, value `RM${applicationProcess:Get Stages;displaycols:stage}`
   - name DIMCM_DEPLOYMENT_AREAS, label Deployment Areas, value `{applicationProcess:Get Deployment`

```
Areas;properties:{[name:DIMCM_PRODUCT,value:
DIMCM_PRODUCT],[name:DIMCM_PROJECT_FILTER,value:
DIMCM_PROJECT_NAME],[name:DIMCM_STAGE_NAME,value:
DIMCM_STAGE_NAME]};displaycols:deployment_area}
```

- name DIMCM_REASONS, label Reasons, value *none*

Configuring Dimensions CM Component Processes

Component processes are used to combine the Dimensions CM plugin steps into the processes needed to execute the set of Dimensions CM operations you need.

Note: The Dimensions CM plugin must be loaded and available before you design a component process.

Tip: For the quickest implementation, import the sample environment and application and modify the properties to suit your needs.

To configure the Dimensions CM component processes:

1. Create a component that will contain your component processes or select an existing one. For example, DimCM Components.
2. Add the following processes to your component if they have not already been imported.
   - Action Baseline
   - Demote Baseline
   - Deploy Baseline
   - Get Baselines
   - Get Deployment Areas
   - Get Products
   - Get Projects and Streams
   - Get Stages
   - Promote Baseline
3. Specify values or variables for each component process step property that will not be set by application properties.
4. Ensure that any properties that will be passed from the application processes are set to `Set a value here` so that those property values will be replaced with the application properties passed to them.

Chapter 5: Integrating with ChangeMan ZMF

You can integrate Deployment Automation with ChangeMan ZMF through the provided ChangeMan ZMF plugin.

The integration between Deployment Automation and ChangeMan ZMF is implemented through the ChangeMan ZMF plugin and the ALM Connector.

Documentation References

- For information on using the Deployment Automation ChangeMan ZMF plugin, including details on the plugin steps, see the *Deployment Automation Plugins Guide*.
- For details on configuring processes using the plugin, see the *Deployment Automation User's Guide*.

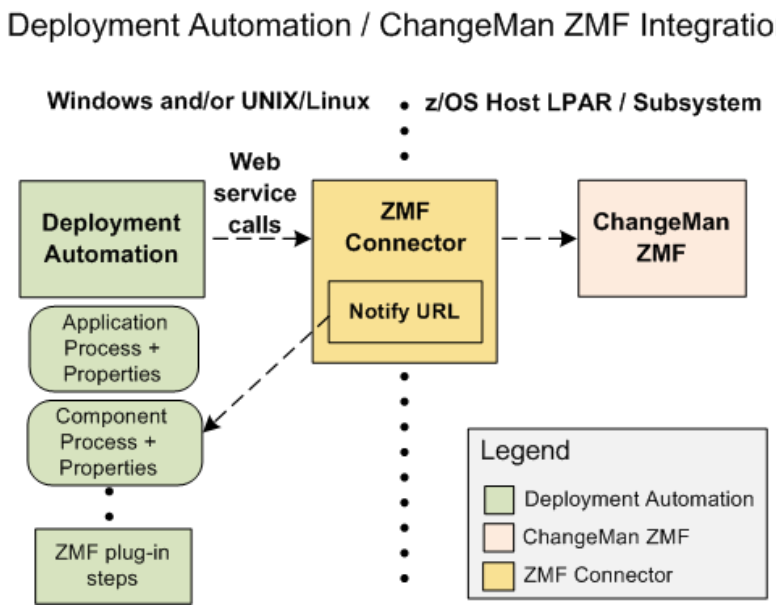
Important: The *ALM Connector Configuration Guide* primarily covers the integration between Release Control and ALM Connector. The configuration details for integration with Deployment Automation are slightly different. Use the Deployment Automation integration documentation rather than the ALM Connector documentation for configuring the integration with Deployment Automation.

The following topics describe the runtime communication and configuration of the ChangeMan ZMF plugin and give details on configuring the ALM Connector for use with Deployment Automation.

ChangeMan ZMF Integration Example

You must configure ChangeMan ZMF communication on the z/OS mainframe and on the Deployment Automation server before you use the ChangeMan ZMF plugin. The rest of the information needed for Deployment Automation to communicate with ChangeMan ZMF is embedded in the Deployment Automation application and component processes, which use the ChangeMan ZMF plugin.

The flow of communication between Deployment Automation, and ChangeMan ZMF is shown in the following figure.



ChangeMan ZMF Integration Runtime Communication

The communication between Deployment Automation and ChangeMan ZMF proceeds as follows:

1. Deployment Automation processes are configured for the ChangeMan ZMF processes to be executed.
2. When the Deployment Automation processes are run, they invoke the ChangeMan ZMF processes. The processes use a proxy ID to logon on behalf of a designated username, typically the Deployment Automation user name, to initiate the requested operations in ChangeMan ZMF.
3. When the ChangeMan ZMF operations complete, an event is sent from the SERNET NTFYURL and the Deployment Automation listener detects it. When the operations are complete, Deployment Automation retrieves ChangeMan ZMF information through the listener and puts it in the Deployment Automation execution log.
4. The Deployment Automation process is updated with the completion status and the component process step is flagged as successful or failed.

Configuring ALM Connector on the Mainframe

The mainframe portion of the ALM Connector should be configured by your ChangeMan ZMF administrator or by someone familiar with the IBM mainframe and ChangeMan ZMF. This part of the configuration is required for the integration between ChangeMan ZMF and Deployment Automation to work.

See the following topics for details on configuring ALM Connector support on the z/OS mainframe specifically for the Deployment Automation integration.

- Configuring a ChangeMan ZMF Proxy User ID [page 29]
- Configuring TSO User IDs and Permissions [page 29]

Configuring a ChangeMan ZMF Proxy User ID

A proxy user ID, or trusted user ID, is required for each ChangeMan ZMF host server, or LPAR. You specify these in the `zmf.properties` configuration file when you configure ChangeMan ZMF communication on the integrating server.

The purpose of the proxy user ID is to allow users to automatically access ChangeMan ZMF through the integration without logging on. The proxy ChangeMan ZMF user ID connects to the host server on behalf of the user.

Consider an example where a user wants to freeze a release unit. The orchestration invoked for the Freeze function requires access to the ChangeMan ZMF host server. The user's TSO user ID is on his SBM contact record and is associated with the proxy user ID; however, there is no password stored in the user's contact record. The proxy user ID (which does have a password) logs on to the ChangeMan ZMF host server on behalf of the user. The proxy user ID impersonates the user, but does not have access to other resources (such as performing ChangeMan ZMF functions). The authority levels of the user are in effect for the transaction.

The proxy user ID can be any SAF-defined user ID. No specific attributes are required. It is not necessary that this user ID be allowed to access TSO. This user ID must be given READ (or higher) access to the "trusted resource". The trusted resource is a SAF resource, by default `SERENA.SERNET.AUTHUSR` in the FACILITY class. The resource and class are user-modifiable by changing the names in the `SERLCSEC` module, which is delivered as source code with ChangeMan ZMF. This module is used for customizing a variety of security-related functions.

Note: It is not necessary to alter `SERLCSEC` to support the integration, as it is already coded for the preceding resource name and class. Be sure to use the version of `SERLCSEC` that is appropriate to your specific version of ChangeMan ZMF, including any customizations that you have applied.

Important: The *trusted resource* is not related to the RACF user ID TRUSTED attribute.

Configuring TSO User IDs and Permissions

TSO IDs used to access ChangeMan ZMF from Deployment Automation must have permission to access every resource required by ChangeMan ZMF functions that Deployment Automation uses.

For information on specifying connection details for ChangeMan ZMF, refer to the *Deployment Automation Plugins Guide*, in the step details in the "ChangeMan ZMF Plugin" section.

Configuration in ChangeMan ZMF

Configuration is required in ChangeMan ZMF for ALM Connector to work.

See the following topics for details.

- Adding Required Startup Parameters [page 30]
- Example ChangeMan ZMF Startup Parameters [page 30]
- Configuring Change Packages in ChangeMan ZMF [page 31]

Adding Required Startup Parameters

The ChangeMan ZMF server must be updated with startup parameters that will enable access to ALM Connector services.

These parameters are keyword options used with the SERNET started task. There are different ways of passing the parameters to SERNET. For details on setting SERNET parameters, refer to the ChangeMan ZMF documentation.

Add the parameters as follows:

- Add the `CMN=(,XXXX)` parameter to the ChangeMan ZMF startup, where `XXXX` is the SERNET TCP/IP port used to process SERNET requests.
- Add the SERNET `NTFYURL` parameter. This parameter is required for notifying your integration when an ALF event is emitted from ChangeMan ZMF, which indicates that ChangeMan ZMF has completed a requested task.

  It must be specified as follows:

  ```
  NTFYURL='<connectorHostname>:<port>/da/services/ZMFALFEventRouter'
  ```

  where `connectorHostname` is the server name where your ALM Connector services are installed and `port` is the port number for that server.

  Important: This parameter is case-sensitive; the non-variable text must be entered exactly as shown. Be sure to include the quotes around the variable string.

- ChangeMan ZMF must be restarted for these changes to take effect. After the restart, verify that the `CMN` port has been started in the SERPRINT listing data set.

Example ChangeMan ZMF Startup Parameters

Example ChangeMan ZMF startup parameters follow.

```
*********************************************************************
*                    MY SERNET MODULE
*                SUBSYSTEM ID 'A' (ALL SITE)
* Please only activate (1)EventRouter at a time..Comment out the
* ones not in use. SERNET can only handle EventRouter.03/19/09 - JN
*********************************************************************
STAX=YES                          /* DO NOT DISCONNECT ISPF APPLICATION
ESTAE=NO                          /* ESTAE RECOVERY
CMN=(,5314)                       /* CHANGE MAN TCP/IP PORT NUMBER
SUBSYS=I                          /* CHANGE MAN SUBSYS ID
SDNOTIFY=H8                       /* WATCH-DOG TIMER
EX003=N                           /* SERJES exit for security
AUTOMESSENGER=NOTIFY              /* ZDD, RLC notify
```

```
*TRACE=(SER,1,3)
*TRACE=(CMN,1,3)
TCPIP=TCPIP
XML=YES
XCH=6124,XCHMSG=6177              /* ZDD, RLC
NTFYURL='ConnectorHostName:8080/da/services/ZMFALFEventRouter'
```

CAUTION! If your site is a DP site, you must specify the same hostname and port in the NTFYURL parameter specified for the DP site and the P site. If not, the P site will continue to wake up looking for work and will fill up the JESMSGLG (JES message log).

Documentation References

- Refer to documentation on passing parameters to SERNET in "Passing Parameters to SERNET" in the *ChangeMan ZMF Installation Guide* .

Configuring Change Packages in ChangeMan ZMF

Change packages and their related ChangeMan ZMF entities are accessed through the integration. The following should be configured in ChangeMan ZMF as part of your ongoing administration and use of ChangeMan ZMF:

- Applications
- Sites
- Change Packages
- Approver lists
- Promotion levels

For details, refer to the ChangeMan ZMF documentation.

Installing the ALM Connector Services

The ALM Connector services must be installed into the Deployment Automation application server before the Deployment Automation ChangeMan ZMF plugin can be used to access ChangeMan ZMF.

Important: The ALM Connector services are supported only in a Tomcat application server.

Install the services as follows:

1. Download the ALM Connector bundle zip file from the Support website. For example, `ZMF_bundle_vvv.zip`, where `vvv` is the version of ALM Connector.
2. Extract the files from the zip file.
3. Stop the Common Tomcat application server under which Deployment Automation is running. For example, Micro Focus Common Tomcat.
4. Copy the ALM Connector `war` files to the application server location where the Deployment Automation `da.war` file is deployed. The default location is as follows:

   `C:\Program Files\Micro Focus\common\tomcat\8.5\webapps`

The ALM Connector `war` files are as follows:

`almzmf.war`

`almzmfalf.war`

`almzmfws.war`

`almsernet.war`

5. Start Common Tomcat.

Configuring the Integration Files

Additional files must be configured in the application server before the integration to ChangeMan ZMF can be used.

Configure the additional files as follows:

1. Download the Deployment Automation ChangeMan ZMF plugin bundle from the Deployment Automation downloads on the Support website. For example, `ChangeMan_ZMF_Bundle_vvv.zip`, where `vvv` is the version of Deployment Automation.

2. Extract the plugin `zip` file, such as `ZMF_6.1.4_v_bbb.zip`, from the bundle, where v is the version of the plugin and `bbb` is the build number.

3. Copy the `zmf-core-CURRENT.jar` from the plugin `zip` file to the Deployment Automation application server as follows:

   a. In the plugin bundle, navigate to the `lib` directory. For example:

   `C:\Users\bjoson\Downloads\ZMF_6.1.4_v_bbb.zip\lib`

   b. Copy the `zmf-core-CURRENT.jar` file to the Deployment Automation application server `WEB-INF\lib` directory. The default path is:

   `C:\Program Files\Micro Focus\common\tomcat\8.5\webapps\da\WEB-INF\lib`

4. In the Deployment Automation application server `WEB-INF` directory, such as

   `C:\Program Files\Micro Focus\common\tomcat\8.5\webapps\da\WEB-INF`,

   edit the `web.xml` file and add the following lines before the `</web-app>` tag.

```
<servlet>
      <servlet-name>ZMFALFEventRouter</servlet-name>
      <servlet-class>com.serena.servlet.ZMFALFEventRouter</servlet-class>
      <init-param>
          <param-name>redirectURL</param-name>
          <param-value>/</param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
      <servlet-name>ZMFALFEventRouter</servlet-name>
```

```
    <url-pattern>/servlet/ZMFALFEventRouter</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>ZMFALFEventRouter</servlet-name>
    <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

5. Restart Common Tomcat.

Loading the ChangeMan ZMF Plugin

The current version of the ChangeMan ZMF plugin must be loaded into Deployment Automation.

Note: If you have just restarted the application server, wait for it to start before starting this procedure.

Load the plugin as follows:

1. In Deployment Automation, navigate to **Administration > Automation**.
2. In the selection box, select **Plugins**.
3. Click the **Load Plugin** button.
4. Click **Choose File** and select the ChangeMan ZMF plugin zip file, such as `ZMF_6.1.4_v_bbb.zip`, that you extracted from the plugin bundle earlier.
5. Click **Load**.

Configuring ChangeMan ZMF Processes in Deployment Automation

After you have configured the integration and loaded the plugin, you should proceed with configuring the ChangeMan ZMF processes in Deployment Automation.

For information on configuring processes, see the *Deployment Automation User's Guide*.

For details on the plugin steps, see the *Deployment Automation Plugins Guide* or *Deployment Automation Plugin Index*.

Chapter 6: Integrating with Nolio

You can integrate Deployment Automation with CA Nolio through the provided Nolio plugin.

The Nolio plugin can be used to retrieve a list of Nolio processes for selection from Nolio that are suitable for execution, and then run the Nolio process on the target environment using the Nolio runProcess2 functionality. The Nolio plugin that enables Deployment Automation and Nolio to communicate uses the Nolio web services and will pass a number of predefined credentials and selection information for the Nolio application, environment, servers, and so on.

The following topics describe the runtime communication and configuration of the Nolio plugin for use with Deployment Automation.

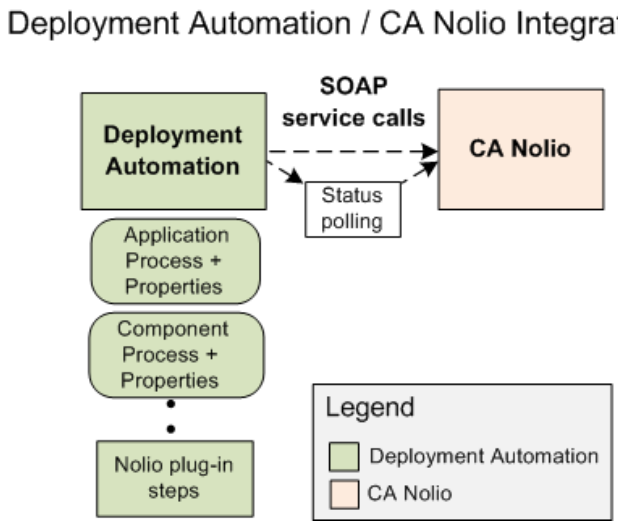
Documentation References

For more information on using plugins, including details on the plugin steps, see the *Deployment Automation Plugins Guide*.

Nolio Integration Example

All of the information needed for Deployment Automation to communicate with Nolio is embedded in the Deployment Automation application and component processes, which use the Nolio plugin.

The flow of communication between Deployment Automation and Nolio is shown in the following figure.



Nolio Integration Runtime Communication

The communication between Deployment Automation and Nolio proceeds as follows:

1. Deployment Automation processes are configured for the Nolio processes to be executed.
2. Deployment Automation requests information from Nolio through SOAP service calls.
3. Deployment Automation polls for the status of the Nolio processes.
4. Once the process in Nolio completes, either successfully or with a failure, the Deployment Automation process step that initiated the transaction completes.

Nolio Plugin Installation

The Nolio plugin must be extracted before it can be loaded into Deployment Automation. Extract the plugin as follows:

1. Download the plugin installation file from the Deployment Automation download location on the Support website. For example, `Nolio_bundle_vvv.zip`, where `vvv` is the version.
2. Extract the files from the plugin bundle. It contains the plugin zip file and files needed to configure the plugin. The plugin zip file is named `Nolio_vvv.zip`, where `vvv` is the version.
3. After the application server is started, in Deployment Automation, navigate to **Administration > Automation**.
4. In the selection box, select **Plugins**.
5. Click the **Load Plugin** button.
6. Click **Choose File** and select the plugin zip file.
7. Click **Load**.

8. Configure the processes for the plugin that are required for the integration.

Configuring Nolio Processes in Deployment Automation

The following topics describe how to configure the processes and properties for optimal use of the Nolio plugin for use with Deployment Automation.

Importing the Sample Nolio Environment

For the quickest and most reliable implementation, you should import the sample environment and application and modify the properties to suit your needs. Before you can import the sample application processes, you must first import the environment that is associated with the application.

One sample environment, UAT, is provided to use with all of the sample applications. If you have already imported the environment to use with one of the other sample applications, you should not import it again.

To import the sample UAT environment:

1. Navigate to the directory location where you downloaded the plugin bundle.
2. Extract the following JSON file if it is not already extracted:

   `Sample UAT.json`
3. If you want to change the name of the environment that will be imported, open the environment JSON file and change the name and description to whatever you want to call your environment.
4. To import the environment:
   a. In Deployment Automation, navigate to **Management > Environments**.
   b. Click the **Application Environments** button and then select **Import Environment**.
   c. Click **Choose File** and browse to the path of the `Sample UAT.json` file.
   d. Click **Import**.

The environment should now be listed in your environments page, as UAT if you did not change the name, or under the name you specified when you changed it.

Importing the Sample Nolio Application

There are several Deployment Automation processes necessary to create the operations needed for this plugin integration. To make it easier for you to configure your processes, an exported sample application is included in the plugin bundle.

The sample application includes all of the application and component information needed to get you started. You can import the exported file and modify the details to match your implementation. Otherwise, you must configure all of your processes and properties manually as described in the subsequent topics:

To import the sample application:

1. Navigate to the directory location where you downloaded the plugin bundle.
2. Extract the following JSON file if it is not already extracted:

   ```
   <product> Sample Application.json
   ```

3. If you changed the name and description of the sample environment that you imported, open the application JSON file and change the corresponding environment name and description to the ones you used in your environment JSON file.
4. To import the application processes:
   a. In Deployment Automation, navigate to **Management > Applications**.
   b. Click the **Application Actions** button and then select **Import Application**.
   c. Click **Choose File** and browse to the path of the JSON file.
   d. Click **Import**.

The application should now appear in the application list.

Configuring the Nolio Application

An application process is used to run the component processes you need. Most of the properties that are needed for the component processes should be set at the application level, because many properties are used by more than one component process.

Tip: For the quickest implementation, import the sample environment and application and modify the properties to suit your needs.

To configure the application:

1. Create an application that will contain your properties and component processes or select an existing one. For example, Nolio Application.
2. If you imported the sample application, edit the application and change the application name and description to match your implementation's values.
3. Add properties to your application that are common to all component processes, or modify the existing imported values to match your system information. For example, the connection information property values are as follows:
   - NOLIO_SERVER_URL: value <your server URL>

- NOLIO_SERVICE_USER: value <your admin username>
- NOLIO_SERVICE_PASSWORD: value <your admin password>

4. Add the following processes to your application if they have not already been imported.
   - Get Applications
   - Get Environments
   - Get Processes
   - Get Process Tags
   - Get Server Types
   - Run Process
5. Add the properties to the application processes that the component processes will inherit, or change them in the imported application. Most properties use variables, such as the following for Run Process:
   - Application: value `${p:NOLIO_APPLICATION}`
   - Environment: value `${p:NOLIO_ENVIRONMENT}`
   - Process: value `${p:NOLIO_PROCESS}`
   - Process Tag: value `${p:NOLIO_PROCESS_TAG}`
   - Servers: value `${p:NOLIO_SERVERS}`
   - Parameters: value `${p:NOLIO_PARAMETERS}`

Configuring Nolio Component Processes

Configure your Nolio component processes. Component processes are used to combine the plugin steps into the processes needed to execute the set of operations you need. You can either configure existing processes imported from JSON files or configure all of them manually.

Note: The Nolio plugin must be loaded and available before you design a component process.

Tip: For the quickest implementation, import the sample environment and application and modify the properties to suit your needs.

To configure the Nolio component processes:

1. Create a component that will contain your component processes or select an existing one. For example, Nolio Components.
2. Add the following processes to your component if they have not already been imported.
   - Get Agents

- Get Applications
- Get Environments
- Get Processes
- Get Process Tags
- Get Server Types
- Run Process

3. Specify values or variables for each component process step property that will not be set by the parent application properties.
4. Ensure that any properties that will be passed from the application processes are set to `Set a value here` so that those property values will be replaced with the application properties passed to them.

Chapter 7: Creating Custom Plugins

You can create your own plugins if there is not already one that meets your needs. See the following for details.

Plugin Creation Overview

A plugin consists of a ZIP file that contains a set of required and optional files in the root directory and supporting files located as needed. To make the plugin available for general use, this ZIP file must be loaded into Deployment Automation. The plugin files are described in the following section.

See also:

- A short tutorial to create a "HelloWorld" plugin, available from the Community website.

File	Description
plugin.xml	This file describes the steps provided by the new plugin. This file also contains informational elements such as description, name, and the location of the plugin in the Process Editor plugin list hierarchy. It is the main plugin file to create. (Required)
upgrade.xml	This file is used by Deployment Automation to upgrade plugins between versions. Plugins are versioned, like all Serena Deployment Automation entities, and this file is used to describe how to upgrade previous versions of the plugin to the latest. (Required)
info.xml	This file is used to detail the high-level plugin information such as who created the plugin and its current version. Although optional, it is important to use the `info.xml` file.
Other	Any supporting script files required by the plugin.

The `plugin.xml` file steps describe the functionality that can be used in the release process. Each step is defined by the use of the `<step-type>` element and contains the following supporting information:

Element	Description
<properties>	A container for <property> child elements, and can contain any number of <property> elements. Property values can be supplied at design-time or run-time. In addition to the properties defined locally in a step, a step can also access properties defined in other steps or even other plugins. This can be done by using the namespaces of the other steps or plugins to reference the property that is needed. For example, `<step-name>.<property-name>`
<command>	This element is used to detail the command that the plugin step is invoking. This command can be a shell script, an operating system command, or a program. It has a set of additional XML attributes that describe how the command is to be invoked.
<post-processing>	This element describes the logic that is to be invoked once the command has finished running and some kind of error-handling or post-command processing is desired.

Plugin steps are performed by an agent that has been configured to run on a target environment, so you must ensure that any step commands configured in the plugin are able to run on those agents. This may require additional software to be installed or licenses to be added as needed. If the appropriate software cannot be invoked correctly, an error message will be shown.

Once a plugin is created, load it into Deployment Automation to make it available to users.

To load a plugin:

1. Create a ZIP archive that contains the XML files (`plugin.xml`, `upgrade.xml`, and `info.xml`) along with any additional scripts required by the plugin.
2. Navigate to **Administration > Automation**.
3. In the selection box, select **Plugins**.
4. Click the **Load Plugin** button.
5. Click **Choose File** and select the ZIP file.
6. Click **Load**.

The plugin.xml File

The functionality that a plugin provides is defined in the `plugin.xml` file. The structure of this file consists of the following:

- elements used by all plugins: the document type declaration, and the `<plugin>` root element that identifies the XML schema type, `PluginXMLSchema_v1.xsd`
- a `header` element that provides the identity, version, and description of the plugin

- the step definitions; each step is delimited by a `<step-type>` element that defines the functionality and properties available to that step

Example

The following shows an example of a typical `plugin.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin xmlns="PluginXMLSchema_v1"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <header>
    <identifier id="plugin_id" version="version_number" name="Plugin Name"/>
    <description/>
    <tag>Plugin_type/Plugin_subtype/Plugin_name</tag>
  </header>
  <step-type name="Step_Name">
    <description/>
    <properties>
      <property name="property_name" required="true">
        <property-ui type="textBox" label="Driver Jar"
            description="The full path to the jdbc driver jar to use."
            default-value="${p:resource/sqlJdbc/jdbcJar}"/>
      </property>
    </properties>
    <post-processing>
      <![CDATA[
        if (properties.get("exitCode") != 0) {
            properties.put("Status", "Failure");
        }
        else {
            properties.put("Status", "Success");
        }
    ]]>
  </post-processing>
  <command program="${path_to_tool">
      <arg value="parameters_passed_to_tool"/>
      <arg path="${p:jdbcJar}"/>
      <arg file="command_to_run"/>
      <arg file="${PLUGIN_INPUT_PROPS}"/>
      <arg file="${PLUGIN_OUTPUT_PROPS}"/>
  </command>
 </step-type>
</plugin>
```

The following sections describe the elements of the `plugin.xml` file and their appropriate attributes.

- The Header: <header> Element [page 45]
- The Plugin Steps: <step-type> Element [page 45]
- Step Properties: <properties> Element [page 46]
- Step Commands: <command> Element [page 48]

- Step Post-Processing: <post-processing> Element [page 49]

The Header: <header> Element

<header> Element

The mandatory `header` element identifies the plugin and contains the following child elements:

<header> Child Elements	Description
`<identifier>`	This element's three attributes identify the plugin: • *version* API version (the version number used for upgrading plugins is defined in the info.xml file). • *id* Identifies the plugin. • *name* The plugin name that appears on the Automation Plugins pane in Deployment Automation. All values must be enclosed within single or double quotes.
`<description>`	Describes the plugin. It appears on the Automation Plugins pane in Deployment Automation.
`<tag>`	Defines where the plugin will appear on the process editor's hierarchy of available plugins. The location is defined by a string separated by slashes. For example, the Tomcat definition is: `Application Server/Java/Tomcat`. The Tomcat steps will be listed beneath the Tomcat item, which in turn is nested within the other two.

The following is a sample header definition:

```
<header>
  <identifier version="3" id="com.&company;.air.plugin.Tomcat"name="Tomcat"/>
  <description>
  The Tomcat plugin is used during deployments to execute Tomcat run-book
  automations and deploy or undeploy Tomcat applications.
    </description>
  <tag>Application Server/Java/Tomcat</tag>
</header>
```

The Plugin Steps: <step-type> Element

Plugin steps are defined with the `step-type` element; each `step-type` represents a single step in the Deployment Automation process editor. A `step-type` element has a name attribute and several child elements: `description`, `properties`, `command`, and `post-processing`.

The mandatory name attribute identifies the step. The description and name specified in the element will appear in the Process Editor.

```
<step-type name="Start">
  <description>Start Apache HTTP server</description>
```

Step Properties: <properties> Element

The `properties` element is a container for properties, which are defined with the `property` tag. Each step has a single `properties` element; a `properties` element can contain any number of `property` child elements.

A `property` tag has a mandatory `name` attribute, optional `required` attribute, and child elements, `property-ui` and `value`, which are defined in the following table.

<property> Element Table

<property> Child Elements	Description
`<property-ui>`	Defines how the property is presented to users in the Deployment Automation Process Editor. This element has several attributes: • `label` Identifies the name of the property shown in the Process Editor **Properties**. • `description` Help shown for the property in **Properties**. • `default-value` The default value of the property. This is displayed in **Properties** and is used by the step if left unchanged. • `type` Identifies the type of widget displayed to users. Possible values are: ▪ `textBox` Enables users to enter an arbitrary amount of text, limited to 4064 characters. ▪ `textAreaBox` Enables users to enter an arbitrary amount of text in a multi-line text box. The length of the text is limited to 4064 characters. ▪ `secureBox` Used for passwords. Similar to textBox except values are redacted. ▪ `checkBox` Displays a check box. If selected, a value of `true` will be used; otherwise the property is not set. ▪ `selectBox` Requires a list of one or more values that will be displayed in a drop-down list box. Configuring a value is described below.
`<value>`	Used to specify values for a selectBox. Each value has a mandatory `label` attribute which is displayed to users, and a value used by the property when selected. Values are displayed in the order they are defined.

Here is a sample `<property>` definition:

```
<property name="onerror" required="true">
 <property-ui type="selectBox"
    default-value="abort"
    description="Action to perform when statement fails: continue, stop, abort."
    label="Error Handling"/>
 <value label="Abort">abort</value>
 <value label="Continue">continue</value>
 <value label="Stop">stop</value>
</property>
```

Step Commands: <command> Element

Steps are executed by invoking the command line command specified by the <command> element. The <command> element's program attribute defines the location of the tool that will perform the command. It bears repeating that the tool must be located on the host and the agent invoking the tool must have access to it. In the following example, the location of the tool that will perform the command, the scripting tool groovy is being invoked, but any command can be run as long as it is in the path and available.

```
<command program='${GROOVY_HOME}/bin/groovy'>
```

The actual command and any parameters it requires are passed to the tool by the <command> element's <arg> child element. Any number of <arg> elements can be used. The <arg> element has several attributes:

<arg> Element Attributes Table

Attribute	Description
<value>	Specifies a parameter passed to the tool. Format is tool-specific; must be enclosed by single-quotes.
<path>	Path to files or classes required by the tool. Must be enclosed by single-quotes.
<file>	Specifies the path to any files required by the tool. Format is tool-specific; must be enclosed by single-quotes.

Because <arg> elements are processed in the order they are defined, ensured the order conforms to that expected by the tool.

```
<command program='${GROOVY_HOME}/bin/groovy'>
  <arg value='-cp' />
  <arg path='classes:${sdkJar}:lib/commons-codec.jar:
                lib/activation-1.1.1.jar:
                lib/commons-logging.jar:lib/httpclient-cache.jar:
                lib/httpclient.jar:lib/httpcore.jar:
                lib/httpmime.jar:lib/javamail-1.4.1.jar' />
```

```
  <arg file='registerInstancesWithLB.groovy' />
  <arg file='${PLUGIN_INPUT_PROPS}' />
  <arg file='${PLUGIN_OUTPUT_PROPS}' />
</command>
```

The `<arg file='${PLUGIN_INPUT_PROPS}' />`

specifies the location of the tool-supplied properties file.

The `<arg file='${PLUGIN_OUTPUT_PROPS}' />`

specifies the location of the file that will contain the step-generated properties.

Note: New lines are *not supported* by the `<arg>` element and are shown in this example only for presentation.

Step Post-Processing: <post-processing> Element

When a plugin step's `<command>` element finishes processing, the step's mandatory `<post-processing>` element is executed. The `<post-processing>` element optionally sets the step's output properties and error handling. The `<post-processing>` element can contain any valid JavaScript script (unlike the `<command>` element, `<post-processing>` scripts must be written in JavaScript). You can also provide your own scripts when defining the step in the Deployment Automation editor. Although not required, it is best practice for the scripts to be wrapped in a `CDATA` element.

You have access to a `java.util.Properties` variable called `properties`. The `properties` variable has several special properties: `exitCode` contains the process exit code, and `Status` contains the step's status. A `Status` value of `Success` means the step completed successfully.

Another available variable, `scanner`, can scan the step's output log on the agent and take actions depending on the results. The `scanner` variable may use the following public methods:

- `register(String regex, function call)` registers a function to be called when the regular expression is matched.
- `addLOI(Integer lineNumber)` adds a line to the lines of interest list, which are highlighted in the Log Viewer; implicitly called whenever scanner matches a line.
- `getLinesOfInterest()` returns a `java.util.List` of lines of interest. This can also be used to remove lines.
- `scan()` scans the log. Use after all regular expressions are registered.

The post-processing script can examine the step's output log and take actions based on the result. In the following code fragment, `scanner.register()` registers a string with a regular expression engine, then takes an action if the string is found. Once all strings are registered, it calls `scanner.scan()` on the step's output log line by line.

```
![CDATA[
    properties.put("Status", "Success");
```

```
if (properties.get("exitCode") != 0) {
    properties.put("Status", "Failure");
}
else {
  scanner.register("(?i)ERROR at line",  function(lineNumber, line) {
      var errors = properties.get("Error");
      if (errors == null) {
          errors = new java.util.ArrayList();
      }
      errors.add(line);
      properties.put("Error", errors);
      properties.put("Status", "Failure");
  });
  .
  .
  .
  scanner.scan();
  var errors = properties.get("Error");
  if (errors == null) {
      errors = new java.util.ArrayList();
  }
  properties.put("Error", errors.toString());
}
]]
```

You can also use post-processing scripts to set output properties that can then be used in other steps in the same process. This enables you to design complex workflows. Reference prior step output properties this way:

`${p:stepName/propName}`

The upgrade.xml file

To upgrade a plugin, you must create an upgrade XML file. This can be done as follows:

1. Increment the number of the `version` attribute of the `<identifier>` element in plugin.xml.
2. Create a `<migrate>` element in upgrade.xml with a `to-version` attribute containing the new number.
3. Place the property and step-type elements that match the updated plugin.xml file within this element, as shown in the following example.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin-upgrade
        xmlns="UpgradeXMLSchema_v1"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <migrate to-version="3">
    <migrate-command name="Run SQLPlus script">
      <migrate-properties>
```

```
        <migrate-property name="sqlFiles" old="sqlFile"/>
      </migrate-properties>
    </migrate-command>
  </migrate>
  <migrate to-version="4">
    <migrate-command name="Run SQLPlus script" />
  </migrate>
  <migrate to-version="5">
    <migrate-command name="Run SQLPlus script" />
  </migrate>
</plugin-upgrade>
```

Of course, you can also make a script-only upgrade, that is, an upgrade that contains changes to the step's associated scripts and files but does not change plugin.xml. This mechanism can be useful for plugin development and for minor bug-fixes/updates.

Any upgrade that does not change the step definitions or properties does not need to provide an `upgrade.xml`. You can simply load the new version of the plugin using the Automation Plugins pane in Deployment Automation.

The info.xml File

Use the optional `info.xml` file to describe the plugin and provide release notes to users. The file's `<release-version>` element can be used for version releases.

Chapter 8: Integrating with Source Configuration Tools

Deployment Automation provides built-in source configuration types that enable you to load artifacts from external tools into Deployment Automation as component versions. Loading artifacts into Deployment Automation enables you to track your artifacts as component versions as they are deployed into application environments.

For details on selections for the **Source Config Type** fields while creating or editing components, see "Creating Components" in the *Deployment Automation User's Guide*.

The source configuration tools that you can select are shown in the following table.

Source Config Type	Description
AnthillPro	Select this to load artifacts that are stored in AnthillPro into Deployment Automation as component versions.
Artifactory - Folder based	Select this to load artifacts that are stored in file folders managed by Artifactory into Deployment Automation as component versions.
Artifactory - Maven / NuGet	Select this to load artifacts that are stored in Maven or NuGet repositories that are managed by Artifactory into Deployment Automation as component versions.
ClearCaseUCM	Select this to load artifacts that are stored in ClearCase UCM into Deployment Automation as component versions.
Dimensions	Select this to load artifacts that are stored in Dimensions CM into Deployment Automation as component versions.
File System (Basic)	Select this to load artifacts into Deployment Automation from directories in your file system. This imports all files in the subdirectories and creates a component version either on a designated name or based on a version name pattern. Automatic import is not supported with this option.
File System (Versioned)	Select this to load artifacts into Deployment Automation from directories in your file system, creating a component version for each subdirectory in the base path.
Git	Select this to load artifacts that are stored in Git into Deployment Automation as component versions.

Source Config Type	Description
Jenkins	Select this to load artifacts that are stored in Jenkins into Deployment Automation as component versions. This does not display additional fields, but rather indicates that the Jenkins plugin for Deployment Automation is configured and activated.
Luntbuild	Select this to load artifacts that are stored in Luntbuild into Deployment Automation as component versions.
Maven	Select this to load artifacts that are stored in Maven into Deployment Automation as component versions.
PVCS	Select this to load artifacts that are stored in PVCS into Deployment Automation as component versions.
Perforce	Select this to load artifacts that are stored in a Perforce versioning engine into Deployment Automation as component versions.
StarTeam	Select this to load artifacts that are stored in Borland StarTeam into Deployment Automation as component versions.
Subversion	Select this to load artifacts that are stored in Subversion into Deployment Automation as component versions.
TFS	Select this to load artifacts that are stored in Microsoft Team Foundation Server (TFS) into Deployment Automation as component versions. Use this if using XAML build automation.
TFS vNext	Select this to load artifacts that are stored in Microsoft Team Foundation Server (TFS) 2015 and above into Deployment Automation as component versions. If you are using XAML build automation, use the TFS source configuration type.
TFS_SCM	Select this to load artifacts that are stored in TFS_SCM into Deployment Automation as component versions.
TeamCity	Select this to load artifacts that are stored in JetBrains TeamCity into Deployment Automation as component versions.
TeamForge	Select this to load artifacts that are stored in CollabNet TeamForge into Deployment Automation as component versions.
uBuild	Select this to load artifacts that are stored in uBuild into Deployment Automation as component versions.

Chapter 9: Creating Custom Source Configuration Types

You can create your own external source configuration type if there is not already one that meets your needs.

See Chapter 8: Integrating with Source Configuration Tools [page 53] for information on source configuration types that come built into Deployment Automation.

See the following for details on creating your own source configuration types.

Note: The implementation of custom source configuration types is done using Java. To implement these you must be proficient in Java programming.

Getting Started with Custom Source Configuration Types

You can use the provided Java files to create your own external source configuration types.

Some advantages of the external source configuration type architecture include the following:

- It is easy for Java programmers to implement.
- The source configuration types can be loaded into Deployment Automation without restarting the server and can be used immediately in Deployment Automation components.

Get started creating a custom source configuration type as follows:

1. Download the example project from Knowledgebase item S142533. This includes the `sct-commons` support files and the example .java file.
2. Build the example project and load the resulting source configuration type jar into a test Deployment Automation system to become familiar with the way it works.
3. Use the Javadoc to supplement this documentation and the example.
4. Create a project for your new source configuration type using any Java/J2EE IDE, such as Eclipse or IntelliJ IDEA. You can use the example project as a model for your new project.
5. Ensure that `sct-commons-1.1.0.jar` is in the IDE's current class path.
6. Implement the `CommonIntegrator` interface from `sct-commons-1.1.0.jar`. Package the implementation of `CommonIntegrator` in a jar file that you name according to your naming standard. This includes the source configuration type definitions, which are defined using annotations, and is the file that you will load into Deployment Automation when you are ready to use the source configuration type. See An Implementation of the CommonIntegrator Interface [page 57].
7. Define the source configuration type that you want to include in this jar file. For components to work successfully, all the runtime dependencies of the jar must be packaged together.

   Note: Although it is technically possible and more efficient to include multiple source configuration types per file, it is recommended to include only one source configuration type per file. This enables easier maintenance going forward. If you put multiple types per file, you cannot do things like upgrade or delete without impacting all of them.
8. Load your new source configuration type into a test Deployment Automation system. Configure a component to use it and test the functionality you implemented.
9. Load your new source configuration type into your production Deployment Automation system to make it available to your Deployment Automation users.
10. Ensure that those administrators who should have privileges to manage custom source configuration types are given the server role to do so.

Details are included in the subsequent sections of the documentation.

Minimum requirements:

- JDK 1.8 or later
- Deployment Automation 6.2 or later

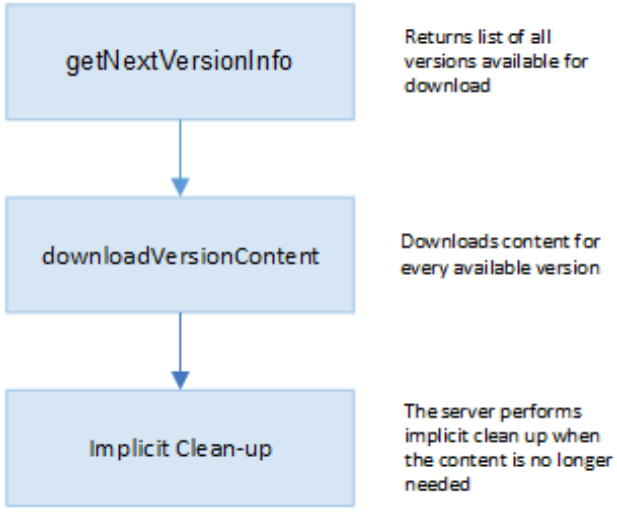
  

  Note:

  You can use custom source configuration types developed using the Deployment Automation 6.1.5 `CommonIntegrator` interface with Deployment Automation 6.2, but you should not use source configuration types developed with the latest `CommonIntegrator` interface with Deployment Automation 6.1.5.

The CommonIntegrator Lifecycle

The `CommonIntegrator` lifecycle has the following major phases that always occur in the given order:



1. **getNextVersionInfo**

   The server queries `CommonIntegrator` for the list of all available versions using `VersionInfo`. If there are any version parameters for `CommonIntegrator`, they are passed as properties during the `getNextVersionInfo` call.

2. **downloadVersionContent**

   The server goes through all the returned `VersionInfo` objects and downloads content for each of them by calling the `downloadVersionContent` method. It is your responsibility to download the version into the provided location. Failing to do so may result in the creation of empty versions. You must process versions properly when **Copy to CodeStation** is selected and when it is deselected.

3. **Implicit Clean-up (automatically done by the server)**

   The server puts all the provided content into CodeStation and performs clean-up on the downloaded folder if necessary.

The server guarantees that versions cannot be imported concurrently for the same component on the same server node.

An Implementation of the CommonIntegrator Interface

Custom source configuration types must implement the `CommonIntegrator` interface, as shown in bold in the following example.

```
package com.microfocus.da.sct.samples;

import com.microfocus.da.sct.annotations.SourceConfigTypeConfig;
import com.microfocus.da.sct.annotations.SourceConfigTypeParam;
```

```
import com.microfocus.da.sct.dtos.VersionInfo;
import com.microfocus.da.sct.interfaces.CommonIntegrator;
import com.microfocus.da.sct.interfaces.ComponentInfo;
import org.apache.commons.io.FileUtils;
import org.apache.commons.lang.StringUtils;

import java.io.File;
import java.util.*;

/**
 * Custom File System Versioned SCT
 *
 */
@SourceConfigTypeConfig(name="Sample File System (Versioned)", versionNumber = "1.0", descripti
public class FileSystemVersioned implements CommonIntegrator {


  <more example code here ...>


   }
```

Using the Annotations for Defining UI Properties

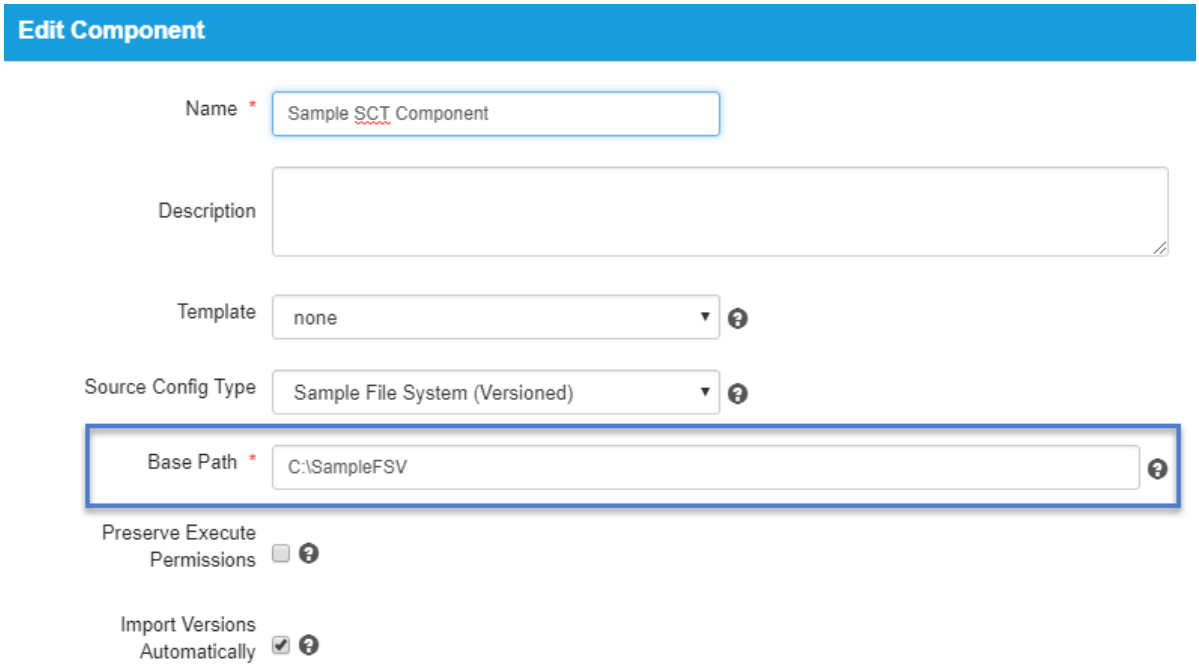
You can use annotations to define the source configuration type properties that appear in the Deployment Automation user interface during component configuration.

For details on the annotations, see the Javadoc.

In the following sample code you can see the annotation `SourceConfigTypeParam` defines a property called **Base Path**, as shown in bold:

```
.
.
.

public class FileSystemVersioned implements CommonIntegrator {

@SourceConfigTypeParam(displayName = "Base Path",
   description = "Base path for artifact storage", required = true)    private String basePath;

  <more example code here ...>

   }
```

After the example is compiled and loaded into Deployment Automation, the property **Base Path** is shown in a component when the example source configuration type is selected. This is shown in the following figure:



Methods to Use During Version Import

You must use specific methods and parameters in the `commonIntegrator` interface to control the behavior of a custom source configuration type.

The methods that define how the custom source configuration type behaves when manual or automatic import of a Deployment Automation component is triggered are as follows:

- `getNextVersionInfo()`
- `downloadVersionContent()`

To create component versions in Deployment Automation, you must implement the behavior accurately using:

`@VersionParams`

To define validation of component settings and show corresponding and locale-dependent messages in the UI, use:

`getAlerts()`

See the Javadoc and the example for more details.

For more information on these methods, see the following topics.

- getNextVersionInfo Method [page 60]
- downloadVersionContent Method [page 61]

getNextVersionInfo Method

The `getNextVersionInfo` method gets information about the next version to be imported.

`getNextVersionInfo` takes the `Properties` object as a parameter.

If you want specific properties to be used when creating the `VersionInfo` object, compose `VersionParams` so that those properties are passed to this method. The values of these parameters are supplied when importing versions.

For example:

```
@VersionParams
(
versionParams =
{
@VersionParam(displayName = "Specific version/tag", name = "revision",
 description = "Name of specific version or tag"), @VersionParam(displayName =
 "Name to create the version with", name = "name", description = "Name for
 identifying the version")
}
)
```

The above returns

`Collection<VersionInfo>`

This is the collection of `VersionInfo` objects that can be created using values from the `VersionParams` annotation.

```
public Collection<VersionInfo>
getNextVersionInfo(Properties properties) throws Exception
{

    // This is the name of the first VersionParam object
    String versionName = properties.getProperty("revision");
    VersionInfo info = new VersionInfo(versionName);

    // Set some additional properties to existing Properties Object
    // so that we can use it in
    // downloadVersionContent
    properties.put("myCustomProperty", "myValue");
    info.setVersionProps(properties); // SET PROPERTIES
    return Arrays.asList(info);
}

public void
downloadVersionContent(VersionInfo versionInfo, File processingDirectoryLocation)
    throws Exception
{
    // Extract properties from VersionInfo object
    Properties prop = versionInfo.getVersionProps();
    String myProp = prop.getProperty("myCustomProperty");
    Client someClient = new Client();
```

```
    Client.downloadContent(processingDirectoryLocation, myProp);
}
```

If you find that there is no version to create based on the `getNextVersionInfo()` call and do not need to call `downloadVersionContent()`, you should return a null or empty `Collections` object. When the Deployment Automation server receives an empty or null object instead of a collection of `VersionInfo` objects, it skips calling `downloadVersionContent()`.

```
public Collection<VersionInfo>
getNextVersionInfo(Properties properties) throws Exception
{

    // This is the name of the first VersionParam object
    String versionName = properties.getProperty("revision");

    // No need to go further, we are not creating the version.
    if(versionName == null || versionName. length() == 0)
     return null;

    VersionInfo info = new VersionInfo(versionName);

    // Set some additional properties to existing Properties Object
    // so that we can use it in
    // downloadVersionContent
    properties.put("myCustomProperty", "myValue");
    info.setVersionProps(properties); // SET PROPERTIES
    return Arrays.asList(info);
}
```

downloadVersionContent Method

The `downloadVersionContent` method downloads the artifact content after the version information has been obtained by a `getNextVersionInfo` method call.

`downloadVersionContent` takes `VersionInfo` and `File` objects as parameters.

This method extracts the information from `VersionInfo` and starts working on generating artifacts. Since the clean-up operation is handled implicitly, you must use `processingDirectoryLocation` for all of the processing so that the contents of that location are cleaned and deleted after the download version processing is done. If you use a location other than `processingDirectoryLocation`, the server does not guarantee the creation of a version. `processingDirectoryLocation` is provided as part of the integration in the server side code. As an implementer, you must put all of your artifacts in this location.

```
public void downloadVersionContent(VersionInfo versionInfo,
      File processingDirectoryLocation) throws Exception {
      SomeClient client = new SomeClient();
            // The below code will process and download the artifacts
            // under processingDirectoryLocation File Object.
```

```
    client.processAndDownloadArtifacts(processingDirectoryLocation);
}
```

Using ComponentInfo to Process Version Information

The `ComponentInfo` interface enables you to process version information.

With `ComponentInfo` you can:

- Determine whether to create a new version
- Create a new version based on version properties you set before
- Get file information for a particular version

After the `ComponentInfo` interface is initialized in the class definitions, you can use it for creating or identifying artifacts as follows.

Existing Versions

`getVersionByName()`

```
// For get version by name
VersionInfo versionInfo = componentInfo.getVersionByName(versionName);
// your logic here
```

`getAllVersions()`

```
// For all the existing versions
List<VersionInfo> existingVersions =
  (componentInfo == null) ? null : componentInfo.getAllVersions();

if(existingVersions != null && existingVersions.size() > 0) {
    for(VersionInfo info : existingVersions) {
        // your logic here
    }
}
```

Latest Versions

```
// For latest versions
VersionInfo singleLatestVersion =
  (componentInfo == null) ? null : componentInfo.getLatestVersion();
// your logic goes here.
```

See the Javadoc and the example for more details.

Using the CommonIntegrator getAlerts Method to Validate Field Values

After someone has configured a component using your custom source configuration type and saved the component, you can use the `CommonIntegrator` method `getAlerts()` to validate the provided values.

If the values are invalid, you can return a collection of alert messages to the UI. The default implementation of this method returns nothing. This method can be treated as a way to prevent integration failures by pre-checking all the required values.

An example implementation is as follows:

```
Private File location = null;

public Collection<String> getAlerts(Locale locale) throws Exception {
    if(location == null)
        return Arrays.asList("The Base Directory cannot be empty");

    File toCheck = new File(location);
    if(toCheck.exists())
        return null;

    return Arrays.asList("The Base Directory does not exist");
}
```

See the Javadoc and the example for more details.

Logging Messages to the Console

For debugging and informational purposes you can log the messages to the system console.

To define and use logging:

1. Define and initialize the `org.apache.log4j.` Logger class as follows:

   ```
   Logger log = Logger.getLogger(YourClass.class);
   ```

2. Include messages as follows:

   Informational Messages

   ```
   log.info(" Your informational message goes here ");
   ```

   Debugging Messages

   Debugging Mode must be enabled.

   ```
   log.debug(" Your debug message goes here ");
   ```

   For error messages

   ```
   log.error(" Your error message goes here ", exceptionVariable);
   ```

Compiling and Loading Custom Source Configuration Types

After you have created a Java class that implements `CommonIntegrator` with `downloadVersionContent` and `getNextVersionInfo` methods, you are ready to compile and generate the custom source configuration type jar file.

Compiling and Generating the Jar File

Using a build tool such as one included in your Java IDE, compile the code and generate the jar file.

Loading into Deployment Automation

After you have compiled and generated the jar file, you are ready to load the source configuration type into Deployment Automation and try it out.

You must have the **Manage Custom Source Config Types** server role to load custom source configuration types. See "Server Roles and System Security" in the *Deployment Automation User's Guide*.

To load a custom source configuration type:

1. In the Deployment Automation UI, navigate to **Administration > Automation**.
2. In the selection box, select **Source Config Types**.
3. Click the **Load Source Config Type** button.
4. Click **Choose File** and select the jar file you generated.
5. Click **Load**.

After they are successfully loaded, custom source configuration types appear in the **Source Config Type** list and you can click the name link to view more information.

Note:

After a source configuration type jar file is loaded into the system, it is copied to the profile directory. The default location is `<profile_location>/var/plugins/source/custom`. By default for Windows this is:

`C:\Users\username\.microfocus\da\var\plugins\source\custom`

Using Custom Source Configuration Types

You can select the custom source configuration types in the **Source Config Type** drop-down when you are creating or configuring a component.

Following are some things to expect when using an external source configuration type, whether or not it is custom.

- If you attempt to delete a source configuration type that is packaged in a jar with multiple custom source configuration types, you'll receive a warning message that this will delete all of the custom source configuration types that packaged with this

one. If you choose to delete, the source configuration type jar is physically deleted from the profile location.

- If a new version of a custom source configuration type is loaded that has additional required fields, you are notified about the missing required fields when you view or edit a component that is configured with that source configuration type.
- If a component is configured with a custom source configuration type that is missing, the component page has an alert message that says to reconfigure the component. This can happen if:
  - Someone deletes the source configuration type
  - The Deployment Automation server starts without loading the custom source configuration type, due to an error loading the jar file or jar files missing from the profile location
- Importing versions works the same as for built-in source configuration types. See "Importing Component Versions" in the *Deployment Automation User's Guide*.

Tip: You must select **Copy to CodeStation** if you want the versions to be imported into the Deployment Automation version file system. This is the preferred setting, as it enables you to track the inventory of your versions in environments.

See "Creating Components" in the *Deployment Automation User's Guide*.