Oracle® Communications IP Service Activator

SDK Base Cartridge Developer Guide

Release 7.5
F59545-01
September 2022

Oracle Communications IP Service Activator SDK Base Cartridge Developer Guide, Release 7.5

F59545-01

Contents

Preface

This guide explains how to use the Oracle Communications IP Service Activator SDK to create base cartridges which integrate with the network processor to enable basic communications with devices.

Audience

This guide is intended for system developers developing base cartridges using the SDK.

Before reading this guide, you should have familiarity with IP Service Activator.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

1

Base Cartridge Guide Overview

This chapter provides a brief overview of the concepts involved in creating base cartridges with the SDK.

Developing Base Cartridges with the SDK

Base cartridges provide a framework to allow the Network Processor to perform basic communication functions with a device. These functions include logging in and out of the device, sending commands or configlets, performing audits, and interpreting responses from the device as successes, warnings, or failures.

Base cartridges do not contain implementations of services. Additional services targeting specific vendor device types are added through integrated service cartridges. Refer to *IP Service Activator SDK Service Cartridge Developer Guide* for details.

Configuration policies are implemented in conjunction with supporting service cartridges. For additional information on creating configuration policies, refer to *IP Service Activator SDK Configuration Policy Extension Developer Guide*.

Core Cartridges

Oracle Communications IP Service Activator existing cartridges, known as core cartridges, include the functions provided by both a base and service cartridges all in the same package.

The base cartridge with separate related service cartridges is the preferred method of supporting new services to maximize scalability and flexibility.

Vendor Cartridges

A base or core cartridge can be combined with a number of service cartridges to create a vendor cartridge, which contains the functionality to connect to a specific device type, and apply the services provided by the service cartridges.

SDK Installation

For SDK installation, configuration, and custom cartridge upgrade instructions, refer to *IP Service Activator SDK Installation and Setup Guide*.

Additional Documentation

Additional documentation is available on the SDK, its concepts and documents, and how to create cartridges and configuration policies.

Refer to Next steps in learning about the SDK in *IP Service Activator SDK Installation and Setup Guide*.

2

Building Base Cartridges

This chapter explains how to build base cartridges using the SDK. It discusses the sample Cisco base cartridge source files provided with the SDK. A brief overview of the steps required to build base cartridges is given, followed by detailed sections explaining all the required activities. Included are steps to try out the procedures on the supplied sample Cisco base cartridge source files.

> **Note:**
>
> For details on installing the SDK and the required third party tools, plus a detailed overview of all SDK concepts, a discussion of cartridge components, and an explanation of how cartridges integrate with the network processor, refer to *IP Service Activator SDK Installation and Setup Guide*.

This guide assumes:

- That Oracle Communications IP Service Activator is deployed to a directory which will be referred to as *Service_Activator_home*. This directory is typically

  C:\Program Files\Oracle Communications\Service Activator
- That you have successfully installed the SDK to a directory which will be referred to as *SDK_home*
- That the required versions of additional third party tools to support the SDK are installed correctly
- That you have set up the required environment variables to support the SDK functions

For details on installing the SDK and the third party tool versions, refer to *IP Service Activator SDK Developer Overview Guide*.

Building a Base Cartridge

This section lists the steps required to build a base cartridge. Following a brief introduction to these steps, each step and all the activities required to execute it are covered in detail.

The steps to build a base cartridge are:

- Creating a Base Cartridge Source Directory and Skeleton Properties File
- Performing Device Characterization and Customizing the Skeleton Properties File
- Generating Cartridge Source Files
- Customizing the Cartridge Source Files
- Compiling and Packaging the Cartridge
- Performing Standalone Tests
- Performing End-to-End Tests

Creating a Base Cartridge Source Directory and Skeleton Properties File

In order to create your own base cartridge, you need to establish a uniquely named directory structure for the source files, and create the skeleton properties file that will be used to generate the starting source files.

Performing Device Characterization and Customizing the Skeleton Properties File

This step is an analysis of how the device performs the functions that the base cartridge is going to manage. You will need to gather information on login and logout sequences, prompts, and more.

Each cartridge you create with the SDK requires a skeleton properties file. In this file, you will edit the property values that control the generation of the cartridge source files. For complete details on all properties, refer to "Base Cartridge Generation Properties".

Generating Cartridge Source Files

This step uses the SDK tools to read the skeleton properties file and create the skeleton cartridge source files.

Customizing the Cartridge Source Files

This step is where most of your development effort will be spent. The key cartridge source components include:

- Device Model (DM) schema definition
- Service Model (SM) to DM transform
- DM validation
- Annotated DM to CLI transform
- Message (success/warning/error) pattern definitions

There are many other source file components you may need to create or modify including files that support audit services, options, capabilities, and pre- and post-checks. These are described later in this chapter.

Compiling and Packaging the Cartridge

This step uses the SDK tools to compile and package the cartridge.

Performing Standalone Tests

Unit and device tests are created as part of the generated skeleton source files and are run using the SDK tools.

Performing End-to-End Tests

To perform end-to-end testing, you'll need to deploy your cartridge into a test IP Service Activator system.

About the Provided Sample Base Cartridge

This section describes the provided sample base cartridge.

Components of the Provided Sample Base Cartridge

A sample working base cartridge for Cisco IOS devices, called cisco, is provided with the SDK in the form of:

- A **skeleton.properties** file

  This file is used to generate the source files for the sample base cartridge. For more information, see "Generating the Sample Base Cartridge Source Files".

- Three pre-edited source files that demonstrate the edits required by the generated source files to produce a working cisco sample base cartridge.
  - **...\audit\auditTemplate.xml**
  - **...\messages\successMessages.xml**
  - **...\xquerylib\dm2cli-common.xq**

The provided sample source files are located in

*SDK_home***samples\baseCartridge\cisco\...**

while the generated sample cartridge source files are placed in

*SDK_home***baseCartridges\cisco\...**

Completing the Sample

To complete the sample, you can copy the provided files over their generated counterparts; or you can edit the generated files.

Purpose of the Provided Sample Base Cartridge

The pre-completed skeleton properties file illustrates how to populate the fields in the file for a working base cartridge. The provided sample source files are based on those generated by the sample base cartridge skeleton.properties file, but are customized further for Cisco IOS. This illustrates the typical post-generation steps you need to perform.

It can be useful to perform a diff between the customized sample source files and a set of generated sample skeleton source files, and then examine the difference between them.

The main values provided by the base cartridge sample are:

- Shows how to fill out the skeleton properties for a base cartridge
- Provides material to increase your understanding of the roles of the various prompt-related regular expressions

- Demonstrates implementation of the config version check optional function
- Lets you inspect the generated source files to see how a simple, working, base cartridge is constructed.
- Lets you complete, compile and package the overwritten generated sample source files into a working base cartridge and deploy it in a test system.
- Lets you take a copy of the provided skeleton properties file, relocate and rename it, and use it as the starting point to generate your own skeleton cartridge source files.

Sample Skeleton Properties File

The sample skeleton properties file that is used to create the source files for the cisco sample base cartridge is called

*SDK_home***samples\baseCartridge\cisco\skeleton.properties**

This properties file is pre-populated with the information needed to construct the starting source files for the cisco sample base cartridge - a working base cartridge that will support Cisco IOS devices.

Some of the generated source files will require editing or you can overwrite them with the provided source files.

As you read through the base cartridge creation steps, instructions are given on how to use the sample to test some of the SDK tools and commands.

Refer to "Base Cartridge Generation Properties" for details on all the properties implemented in the sample properties file which create the source files for the sample.

Creating a Base Cartridge Source Directory and Skeleton Properties File

To create your own base cartridge, you will need to establish a directory structure for the source files, and create a skeleton properties file to generate the starting source files.

> **Note:**
>
> When deciding upon a directory structure for a new base or service cartridge care must be taken to choose a unique base directory name. If the path of a file in the new cartridge is the same as the path of a file in a deployed cartridge, undesirable behavior could occur.

The simplest method is to copy the sample **skeleton.properties** file and edit it for your own use.

To copy and edit the sample file:

1. Create a unique name that identifies the type of device the cartridge will support for your new base cartridge. This name will be referred to as *target_device_name*.

2. Create a new directory to hold your cartridge source files. For example:

   SDK_home\baseCartridges*target_device_name*

3. Copy the sample **skeleton.properties** file into your directory:

   ```
   copy SDK_home\samples\baseCartridge\cisco\skeleton.properties
   SDK_home\baseCartridges\target_device_name
   ```

4. Edit your **skeleton.properties** file and change cisco to the *target_device_name* in the following entries:

   ```
   ## base cartridge name
   sdk_global_cartridgeName=target_device_name
   . . .
   ## packaging structure
   sdk_global_package=com.metasolv.serviceactivator.cartridges.target_device_name
   ```

Performing Device Characterization and Editing the Skeleton Properties File

To construct a base cartridge, you will need to identify the key aspects of a device's characteristics in order to configure the cartridge to appropriately interface with the device. This information is used to edit the skeleton properties file. The skeleton properties file is used to generate a set of customized cartridge source files for you to use as the starting point for your base cartridge.

As you gather the information in the categories below, refer to "Base Cartridge Generation Properties" for details on the how the information you collect is specified in the skeleton properties file.

Basic device characteristics include:

- **Discovery method**: how IP Service Activator will be able to discover the device
- **Access**: how to log in and log out of the device
- **Configuration mode**: commands for entering and exiting a read/write configuration-type mode (e.g. privileged mode)
- **Prompts**: the characters used by the device to represent a user-input prompt in various modes and circumstances
- **Audit commands**: commands to retrieve the entire running configuration for the device for the purposes of comparing a persisted expected configuration with the actual configuration that is on the device
- **Saving the running configuration**: commands to manage start-up and running configurations

Discovery Method

IP Service Activator primarily uses SNMP for discovery. The policy server component queries the device using SNMP MIBs and constructs the device topology, its interfaces and sub-interfaces in the IP Service Activator object model.

The amount of detail on the device retrieved through this basic SNMP discovery process may be sufficient to start construction of your base cartridge. You can test this by discovering your device as you normally would in the IP Service Activator client. Refer to the Service Activator online Help for more information about discovering one or more devices.

For additional information, see the discussion of network discovery and representation in *IP Service Activator Concepts*.

External Discovery

If the basic IP Service Activator SNMP discovery process does not properly discover the device, or leaves out critical details (such as interfaces, sub-interfaces, or VCs), consider developing an external discovery process.

An external discovery process consists of an application, typically written in Java, that connects to the device and then constructs the device topology and its parts in IP Service Activator using one of the available programming interfaces. The connection method can be telnet, SNMP, or whatever else is appropriate.

For Java-based external discovery processes, OJDL is the appropriate programming interface. If the external discovery process is authored using a shell script or perl, for example, then OSS Integration Manger (OIM) CLI commands would be used to interface with IP Service Activator.

> **Note:**
>
> It is beyond the scope of this document to describe exactly how to program an external discovery process. However, relevant programming interface information can be found in *IP Service Activator OSS Java Development Library Guide* and *IP Service Activator OSS Integration Manager Guide*.

The parsing of information retrieved from the device for relevant information is in the domain of the external discovery process.

At a minimum, an external discovery process should create a device object in IP Service Activator with the required attributes set (such as device type, version, and so forth). It should create the main interfaces under the device with appropriate attributes such as interface type, bandwidth, speed, etc. If necessary, it should populate IP Service Activator with any sub-interfaces present on the device.

Access: Logging In

This section describes the two aspects of the access device characteristic.

Logging In

When you create your cartridge's skeleton properties file, it must be configured for one of the login methods supported by the SDK. These are:

- TACACS
- SSH
- Anonymous
- Password-only

All of these login methods have similarities. Typically, the device presents a prompt for a username, and then a prompt for a password. The anonymous login method

involves no prompts. The password-only login method prompts only for a password without a user name.

The skeleton properties which indicate the login methods supported by the device are:

- sdk_authenticationTacacs_supported
- sdk_authenticationSsh_supported
- sdk_authenticationAnonymous_supported
- sdk_authenticationPasswordOnly_supported

> **Note:**
>
> At least one of these properties must be set to **true**.

The login prompts used by the device are defined in the properties file as regular expressions. For example:

```
sdk_authenticationTacacs_useridPrompt=.*Username:
```

In this example, .*Username: is a regular expression that matches any line ending with the text "Username:". When the network processor matches input received from the device using this regular expression, it is recognized as a prompt for a TACACS username. The network processor then sends the username to the device.

Depending on the login procedure for the device, supply values appropriately for related login properties. For example, if you set sdk_authenticationTacacs_supported to true in the properties file, you will also need to specify values for many of the following additional properties:

- sdk_authenticationTacacs_useridPrompt
- sdk_authenticationTacacs_passwdPrompt
- sdk_authenticationTacacs_errorPrompt
- sdk_authenticationTacacs_enablePasswdPrompt
- sdk_authenticationTacacs_enablePasswdPasswdPrompt
- sdk_authenticationTacacs_enablePasswdErrorPrompt

If the device has two levels of authentication, then the **..._enablePasswdPrompt** property should also be defined. The device presents this prompt when a user attempts to enter privileged mode (e.g. by entering the enable command on a Cisco device).

> **Tip:**
>
> To get the required prompt information for setting the login properties, connect to an actual device and take note of the prompts the device presents. Be sure to test invalid parameters so you can capture error messages returned, and use these in setting the various login error properties.

Logging Out

Set the **sdk_logoutCommand_cmd** property to the exact command used to log out of the device.

Configuration Mode

Most devices have two modes of operation: user (read-only), and configuration (read-write) mode. User mode allows you to query the device about its hardware, configuration or statistics, but bars you from making changes. Configuration mode allows modification of the device's configuration. For example, Cisco devices use the command configure terminal (or conf t for short).

In the skeleton properties file, specify the exact commands to enter and exit configuration mode on the device:

- sdk_configMode_cmd
- sdk_configModeTerminate_cmd

Prompt Matching

The Network Processor determines when a command it has sent to a device has been received by the reception of a prompt from the device.

The response time varies by device and command. On a slow connection, sending and receiving the command, and receiving the prompt may take a number of seconds. The Network Processor waits until it has received the return prompt to be sure that the command was received correctly.

Care must be taken when configuring a regular expression to match a prompt because prompts are configurable on a per-device basis. Additionally, prompts can change throughout the communication session.

For example, when a device is being configured, the prompt may change to show which area of the configuration is currently being worked on.

In a session:

- The device presents the prompt **mydevice>** after successful login.
- The prompt changes to **mydevice#** when the user enters privileged mode.
- The prompt changes to **mydevice(config)#** when configuring.
- The prompt may change to variants of **mydevice(config-*)#** when configuring.

In order to match a prompt, the Network Processor attempts to discover the prompt upon logging in. It remembers the original prompt and uses it throughout the communication session.

When configuring your base cartridge, the **sdk_getConfiguration_matchPattern** property defines a regular expression to be used by the Network Processor to discover the prompt upon login by matching the pattern with the responses from the device. There are also properties to define text which precedes and text which follows the prompt.

Table 2-1 lists the prompt definition properties and their matched regular expression.

Table 2-1 Prompt Definition Properties

Prompt Definition Property	Description of Regular Expression
sdk_getConfiguration_matchPattern	After a successful login, Network Processor uses this expression to match on the prompt. The expression must contain one group that matches the core prompt. The network processor then uses this core prompt as a pattern to match on. Example: `sdk_getConfiguration_matchPattern=([^>#\\n]*)[>#]` This specifies that the prompt includes everything except the characters ">" and "#" or a new line that comes before one of the characters ">" or "#". The prompt "mydevice>" would match and the word "mydevice" would become the core prompt pattern.
sdk_getConfiguration_prePendPattern	Regular expression to match text which prepends the prompt, such as a command number (e.g. “10: mydevice>:), or whites pace. Example: `sdk_getConfiguration_prePendPattern=\\n` This specifies that the prompt could be preceded by a blank line.

sdk_getConfiguration_appendPattern	Regular expression to match text which follows the prompt, such as a special character
	Example:
	`sdk_getConfiguration_appendPattern=(([>#])|(\\(.*\\)#))`
	This specifies that the prompt could be followed by the characters ">" or "#", or some context in parentheses followed by the character "#". It would match with:
	`mydevice>`
	`mydevice#`
	`mydevice(config)#`
	`mydevice(config-pmap)#`

sdk_getConfiguration_errorPattern	Regular expression to match error response from login.
sdk_getConfiguration_hostPattern	Pattern to match the hostname within the prompt.

For complete details on all properties, refer to "Base Cartridge Generation Properties".

Audit Commands

An audit compares the configuration that the Network Processor has persisted for the device with the actual configuration that is retrieved from the device. Commands not administered by IP Service Activator are filtered out.

In order to perform an audit, the Network Processor must be able to parse all of the commands on the device. Most devices allow this by providing a command that returns the entire configuration of the device (e.g. **show-running-configuration** on Cisco).

The required properties to configure auditing for a base cartridge are:

- sdk_audit_supported
- sdk_showRunningConfig_supported
- sdk_showRunningConfig_cmd

To enable auditing, the base cartridge properties **sdk_audit_supported** and **sdk_showRunningConfig_supported** should be set to **true**. The property **sdk_showRunningConfig_cmd** should be set to the exact command to retrieve the current configuration of the device.

For complete details on these properties, refer to "Base Cartridge Generation Properties".

Saving the Running Configuration

Some devices have both a running configuration — the configuration that is currently executing on the device, and a startup configuration — the configuration that is loaded upon restart of the device. The startup configuration is stored on disk whereas the running configuration is usually only stored in memory.

Devices with running and startup configurations typically support the copying of one configuration to the other. For example, when a new configuration is applied to a device it may then be backed up to the startup configuration.

The properties that are used by the SDK to support this are:

- sdk_saveRunningConfig_supported: set this to **true** to indicate support for copying the running configuration to the startup configuration
- sdk_saveRunningConfig_cmd: set this to the exact command to copy the running configuration to the startup configuration

Configuration Version

IP Service Activator, when generating device models, generates a new configuration version statement that conforms to the following format:

```
YYYY-MM-DDTHH:MM:SS.sssZ
```

This configuration version is stored in the device model and is written to the device along with the general configuration commands.

The configuration version, read back from the device during audit and subsequent configuration updates, is used to determine if the last persisted device model is in sync with what is on the device. If the device and device model configuration versions match, then the last persisted device model should match with the commands on the device. If the configuration versions don't match, then either the last persisted device model is out of date or the device is out of sync with IP Service Activator.

The configuration version functionality consists of several sub-functions that an SDK developer needs to be aware of.

- Configuration versioning in the target and last device models
- Configuration versioning on the network devices
- Configuration version pre-check prior to committing configuration changes to a device

- Configuration version auditing

The first function is always performed by the cartridge framework. In other words, the target, and consequently, the last device models automatically contain configuration version information.

The last three functions are optional in the SDK. An SDK developer can enable them with the **sdk_configurationversion_supported** property. When set to **true**, the source file generator generates XQuery functions responsible for the following:

- Transforming the configuration version in the annotated device model into CLI commands.
- Retrieving the configuration version from the device, comparing it with the configuration version in the last device model and, when out-of-sync, raising faults that prevent making further configuration changes to a device that is out of sync.

SkeletonGeneratorProperties

The configuration version functions are further parameterized using 4 more properties which are only relevant when **sdk_configversion_supported** is set to **true**.

These properties are:

- sdk_configversion_updateCmd=alias exec IpsaConfigVersion: the prefix of the CLI command that updates the configuration version on the device. The entire command is formed by appending the configuration version to it. For example:

  ```
  alias exec IpsaConfigVersion 2007-08-12T12:59:00.234Z
  ```

- sdk_configversion_removeCmd=no alias exec IpsaConfigVersion: the prefix of the CLI command that removes the configuration version from the device. The entire command is formed by appending the configuration version to it. For example:

  ```
  no alias exec IpsaConfigVersion? 2007-08-12T12:59:00.234Z
  ```

- sdk_configversion_showCmd=show aliases exec | include IpsaConfigVersion: the CLI command that retrieves the configuration version from the device. The result from this command is expected to be a single line which, among other things, must contain the configuration version.
- sdk_configversion_extractCmd=IpsaConfigVersion? +(.+): a regular expression that, when applied against the result from **sdk_configversion_showCmd** must match, so that its first group is bound to the configuration version string.

The values assigned to the above properties illustrate what is suitable for generating a skeleton cisco IOS cartridge.

Generating Cartridge Source Files

The SDK provides a tool to generate the base cartridge source files from the skeleton properties file. Once the source files are generated, you will need to edit the source files to complete your base cartridge.

> **Note:**
>
> Ensure that you save copies of any cartridge source files you alter prior to re-generating from the **skeleton.properties** file to ensure that you do not lose customization work. Alternatively, modify the **skeleton.properties** file so that a new target directory name is used. In either case, you will need to manually merge any alterations you made in the previous iteration if you want those changes to persist.

Generating the Sample Base Cartridge Source Files

To generate the sample base cartridge source files using the data from the sample skeleton properties file:

1. Set the cartridge version string variable. For example, if the cartridge version is 1.0, on a Windows host, type the command:

   ```
   set VERSION_STRING=1.0
   ```

2. In *SDK_home*, either:

   - Run the included batch file to run the cartridge generator script:

     ```
     genbc samples\baseCartridge\cisco\skeleton.properties
     ```

     or

   - Type in the command to run the cartridge generator script:

     ```
     ant -DtemplateType=baseCartridge -
     DpropFile=SDK_home\samples\baseCartridge\cisco\skeleton.properties
     ```

> **Note:**
>
> To use the batch file, you must first add *SDK_home***bin** to your PATH variable where *SDK_home* is the SDK directory.

Result of Generation Process

The directory structure you created previously (see "Creating a Base Cartridge Source Directory and Skeleton Properties File") has been extended by using the *sdk_global_cartridgeName* value from the skeleton properties file. The cartridge source files generated under *SDK_home***baseCartridges***sdk_global_cartridgeName*\ include:

- **build.xml**: ant build file to build the base cartridge
- **src\synonyms.xml**: used by the audit process
- **src\...\audit\auditTemplate.xml**: stub file for audit commands
- **src\...\capabilities\empty_caps.xml**: stub file for capabilities information
- **src\...\messages**: contains .xml files with success, error and warning message patterns

- **src\...\options\options.xsd**: stub schema file for cartridge options
- **src\...\schema\devicemodel.xsd**: contains the stub base cartridge device model schema
- **src\...\test**: resources for testing the base cartridge
- **src\...\transforms**: transforms including pre-check, SM to DM, annotated DM to CLI, DM validation, and a restore template for Configuration Management support
- **src\...\xquerylib**: additional XQueries for SM to DM, DM to CLI, DM upgrade, DM version and pre-check.
- **src\...\cisco\auditLogging.properties**: used to set property values for audit logging
- **src\...\cisco\Registry.xml**: identifies the base cartridge instance
- **src\...\cisco\Customization.xml**: can be used to override **Registry.xml**

A log file is also created within the logs directory:

- *SDK_home***logs\generator.log**

To continue working with the sample base cartridge, go to "Completing the Sample Cartridge Source Files".

Generating Your Base Cartridge Source Files

When you create your own base cartridge, the cartridge name and the root folder for the generated source are based on the *sdk_global_cartridgeName* property value in the skeleton properties file. See "Creating a Base Cartridge Source Directory and Skeleton Properties File" for details. The property value is incorporated into the cartridge source files in place of *sdk_global_CartridgeName*.

To generate your skeleton base cartridge source files using your customized skeleton properties file:

1. Set the cartridge version string variable. For example, if the cartridge version is 1.0, on a Windows host, type the command:

   ```
   set VERSION_STRING=1.0
   ```

2. In *SDK_home*, either:

   - Run the included batch file to run the cartridge generator script:

     ```
     genbc \baseCartridges\<sdk_global_cartridgeName>\skeleton.properties
     ```

     or

   - Type in the command to run the cartridge generator script:

     ```
     ant -DtemplateType=baseCartridge -
     DpropFile=SDK_home\baseCartridges\sdk_global_cartridgeName\skeleton.properties
     ```

> **Note:**
>
> To use the batch file, you must first add *SDK_home***bin** to your PATH variable where *SDK_home* is the SDK directory.

Result of the Generation Process

This extends the SDK directory structure in a similar manner to what is described in "Generating the Sample Base Cartridge Source Files".

> **Note:**
>
> It is possible to use a different name for the skeleton properties file. If you choose to do this, supply the new name instead of **skeleton.properties** in the ant commands.

Troubleshooting the Cartridge Generation

This section discusses where to find information to help you resolve cartridge generation issues.

Using an Alternate Directory Structure

If you are not using the standard directory structure to lay out all the configuration policies, base cartridges and service cartridges being developed using the SDK, then you must modify the Java sample **build.xml** file to ensure that all instances of **sdkDir** are replaced with valid paths to the respective files. The preferred way to do this is to set **sdkDir** to the top level directory for all the SDK-based artifacts.

Base Cartridge Generator Message Logging

The logging level of the cartridge generator can be controlled by editing the settings in the *SDK_home***\config\logging.properties** file.

The default is to log debug level messages. Output is sent to both stdout and a logging file: *SDK_home***\logs\generator.log**.

Troubleshooting Property File Attributes

If a mandatory attribute in the cartridge properties file is missing, source file generation will terminate prematurely. Refer to the log file to ensure that mandatory values are set. Lines such as:

```
Verify property set: propertyName propertyValue
```

indicate that the mandatory attribute named is being verified.

Lines such as:

```
Testing property: propertyName propertyValue
```

indicate that the propertyValue is being tested to ensure it compiles as a proper regular expression. If propertyValue does not compile correctly, code generation does not stop, but the following message is generated:

```
Testing property (suspicious pattern): sdk_testBadRegex_Pattern |)_+
```

Customizing the Cartridge Source Files

When creating your cartridges, you will need to make appropriate edits to the skeleton source files to support the particular functionality you want to implement for your device.

The key cartridge source components you need to create and/or modify include:

- Device Model (DM) schema definition
- Service Model (SM) to Device Model transform
- Device Model validation
- Annotated Device Model to CLI transform
- Message (success/warning/error) pattern definitions

There are many other source file components you may need to create or modify including files that support audit services, options, capabilities, and pre- and post-checks. These are described later in this chapter.

Device Model Schema Definition

This device model extends the Network Processor's base device model to realize the new services being administered by the cartridge.

Service Model to Device Model Transform

This XQuery or java transform transforms the device-independent service model to a device-specific device model.

It is essential that the service model Definition IDs, which identify policy definitions, and Association IDs, which identify the links between defined policies and their target objects and their representative concretes in IP Service Activator, flow through from the service model to the device model.

Device Model Validation

If the cartridge registry entry <dmValidation> contains a dmValidation entry, the Network Processor will invoke this function to validate the transformed device model. This would capture logical faults as opposed to syntax faults which would be caught by the device model validation using **deviceModel.xsd**.

Annotated Device Model to CLI Transform

The Network Processor compares the target device model with the last device model that was persisted to the database after the last successful push to the device. The Network Processor annotates the target device model. For each policy object, the annotation includes the smId, a dmId that is generated by the Network Processor and a changeType which indicates whether configuration is being added, deleted or modified on the device.

The Network Processor invokes the <dmToCli> entry in the cartridge instance. This transforms the annotated device model into a CLI document that is a list of native configuration commands to be sent to the device.

Message Pattern Definitions

As part of the creation of a base cartridge, you will need to create success, error, and warning message pattern files. This section explains how to analyze device responses and create the appropriate entries in the message XML files.

For an overview of the concepts behind message files, refer to *IP Service Activator SDK Developer Overview Guide*.

Device Response Analysis

To analyze the device's response to issued commands, the responses are matched against known success, warning, and error patterns. Note the following:

- A response from the execution of a single command may include multiple messages, and may be split across more than one line.
- The response is treated as a single line for pattern matching.
- In order to handle cases where responses match multiple types of response, patterns, a comparison order is employed as follows:
  - Success patterns are matched first. If the command response matches a success pattern, the command response is a success response.
  - If the command response does not match any success pattern, warning patterns are matched. The command response may be a blocking or non-blocking warning response.
  - If the command response does not match any success pattern or warning pattern, error patterns are matched. The command response may be an error response, if a match is found, or unknown error response, if a match is not found. The difference is that the fault message for a known error response will include the message associated with the error pattern.
  - If the command response does not match any success pattern, or warning pattern, or error pattern, the response is considered an unknown error response.

Example Device Response Pattern Match

This section contains an example of a response to a command which matches multiple types of response patterns.

Command sent:

```
rate-limit output access-group 100 8000 2000 2000 conform-action set-mpls-exp-
transmit 0 exceed-action set-mpls-exp-transmit 1
```

Response received:

```
 Illegal normal burst size
    Increasing normal burst size to 4470
    Illegal extended burst size
    Increasing extended burst size to 4470
    "set-mpls-exp-imposition-transmit" and "set-mpls-exp-imposition-continue"
    not allowed in output rate-limit command.
```

Blocking warning pattern matched:

```
<!-- in theory, we could ignore this and continue. However, we receive the same
message when removing the policing rule. This means a user could see the warning,
correct the problem, and then see the warning return. -->
  <cmd:warningPattern blocking="true">
  <cmd:pattern>(?s).*Illegal .* burst size.*Increasing .* burst size.*</cmd:pattern>
  </cmd:warningPattern>
<cmd:warningPattern>
```

In this example the actual response pattern to match against the command response is:

```
(?s).*Illegal .* burst size.*Increasing .* burst size.*
```

The response also matches this unknown error response:

```
"set-mpls-exp-imposition-transmit" and "set-mpls-exp-imposition-continue"
    not allowed in output rate-limit command.
```

However, blocking warning patterns are matched first, so the outcome is:

```
Response is a blocking warning response.
```

Defining Device Response Patterns

As part of the base cartridge creation process, you must define the list of device command response patterns. Valid response pattern files are in XML format and conform to the **cliModel.xsd** schema.

> **Note:**
>
> Success, warning, and error response patterns are all defined as regular expressions.

Defining Success Response Patterns

The location of the **successMessages.xml** file is referenced in the successMessages element in each registry entry in the **Registry.xml** file.

To define success patterns:

1. Edit **successMessages.xml** and add success patterns as necessary, based on your observations of responses received from the actual device when particular commands are sent.

Success patterns are specified as regular expressions. For example, the following success pattern contains a comment between <!-- and --> and a pattern which will match a response containing the text `QoS Reserved Bandwidth has been modified. Configuration may be affected.`"

```
<cmd:successPattern>
  <!-- -undo qos reserved-bandwidth command being issued -->
  <cmd:pattern>(?s).*QoS Reserved Bandwidth has been modified. Configuration may be
affected.*</cmd:pattern>
</cmd:successPattern>
```

Defining Warning Response Patterns

The location of the **warningMessages.xml** file is referenced in the warningMessages element in each registry entry in the **Registry.xml** file.

To define warning patterns:

1. Edit **warningMessages.xml** and add warning patterns as necessary.

   A warning pattern is non-blocking by default.

   For example:

```
<cmd:warningPattern>
  <cmd:pattern>(?s).*startup-config file open failed.*</cmd:pattern>
</cmd:warningPattern>
```

To define a warning pattern as blocking:

1. In the warning pattern definition, set blocking="true"

   For example:

```
<cmd:warningPattern blocking="true">
  <cmd:pattern>(?s).*Illegal .* burst size.*Increasing .* burst size.*</
cmd:pattern>
  </cmd:warningPattern>
<cmd:warningPattern>
```

Defining Error Response Patterns

The location of the **errorMessages.xml** file is referenced in the errorMessages element in each registry entry in the **Registry.xml** file.

To define error patterns:

1. Edit **errorMessages.xml** and add error patterns as necessary.

   For example, the following error pattern contains a comment between <!-- and --> and a pattern which will match a response containing the text `Invalid input`. The fault message will include the rejected command and the message `Invalid input` associated with the error pattern.

```
<cmd:errorPattern>
  <cmd:pattern>(?s).*Invalid input.*</cmd:pattern>
  <cmd:message>Invalid input</cmd:message>
</cmd:errorPattern>
```

Options

Using the options framework in IP Service Activator, you can customize the configuration style for different device types and IOS combinations.

To do this, you define and document configuration options for a cartridge, and implement the variations in the service model to device model transform, and the annotated device model to CLI transform, based on the option values.

Option values for specific device type and IOS combinations are specified in option configuration files, which are registered by cartridge units in the **Registry.xml** file. The

option configuration files, and the registry entries that reference them, may be customized by the system administrator once the cartridge is deployed.

Defining Options

When you use the cartridge skeleton generator tool to generate the base cartridge source files, a skeleton XML schema file named **options.xsd** is automatically created. This file contains the schema for the cartridge configuration options. Occurrences of ${sdk_global_cartridgename} are substituted with your cartridge's name in the resultant **options.xsd file**.

Add configuration option definitions to an options file to handle such things as device command format variants.

Using **options.xsd** as a starting point, create one configuration option file for each set of device type, and IOS combinations for which the same set of option values are required. For example, if option values correspond to command variants, then a configuration option file defines the set of command variants for a set of device type and IOS combinations.

Configuration options defined in **options.xsd** include:

- Name: xml element name such as **cartridge.***sdk_global_cartridgeName***.sampleEnumerationOption**
- Type: defines restrictions on the allowable values
- Default value: the value to be used if the option value is unspecified

Example option definition from **options.xsd**:

```
<xs:element name="cartridge.<sdk_global_cartridgeName>.sampleEnumerationOption"
minOccurs="0" default="value3">
  <xs:simpleType>
    <xs:restriction base="opt:StringValue">
    <xs:enumeration value="value1"/>
    <xs:enumeration value="value2"/>
    <xs:enumeration value="value3"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Document the details of the configuration options for the cartridge such that the options may be configured at deployment time. For each option, this includes:

- The option name
- Possible values
- Default value
- The effect of setting the option values such as command variants

Registering Options

Option files for base cartridges are registered by cartridges in the **Registry.xml** file and options files for service cartridges are registered in the **Extension.xml** file. A valid option configuration file for a cartridge is in XML format and conforms to the cartridge options schema.

Here is an example of an option configuration file which uses the options defined in the sample **options.xsd** schema. Substitute the cartridge name for $

{sdk_global_cartridgeName}. In this example, **value1** is set for **sampleEnumerationOption**, and the default values defined for the remaining options will be used.

```
<?xml version="1.0" encoding="UTF-8"?>
  <base:options xsi:type="CartridgeOptions" xmlns="http://www.metasolv.com/
serviceactivator/${sdk_global_cartridgeName}/options" xmlns:base="http://
www.metasolv.com/serviceactivator/options" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance">
     <cartridge.${sdk_global_cartridgeName}.sampleEnumerationOption>value1</
sampleEnumerationOption>
  </base:options>
```

To register an option configuration file, add a reference to it in the <**options**> entries in the **Registry.xml** file.

Implementing Options

Variations in the service model to device model, and device model to CLI transforms are implemented based on option values. For this purpose, the option values are made available to the transforms at execution time. The **options-common.xq** module in the **networkprocessor.jar** file can be imported by the XQuery modules implementing the transforms, to provide methods for retrieving the option values.

To import **options-common.xq**:

1. Enter the following command syntax:

   ```
   import module namespace options = "options-common-functions" at "resource://
   metasolvcom/metasolv/serviceactivator/networkprocessor/xquerylib/options-
   common.xq";
   ```

A sample XQuery code to retrieve sample option value (where value3 is the default value) follows:

```
if (options:getStringOption("cartridge.$
{sdk_global_cartridgeName}.sampleEnumerationOption", "value3") = "value1") then
. . .
```

Customizing the Registry

When a cartridge is installed, by extracting it to the *Service_Activator_home* directory, it creates a sample registry configuration directory under it such as *Service_Activator_home***\Config\networkprocessor\ciscoSampleRegistry**. The name of the sample directory is *sdk_global_cartridgeName***SampleRegistry**. This directory contains samples of the various configuration files that you may want or need to customize.

One of the main files that you will need to edit, to customize any SDK registry entry, is the customization registry file. A sample of this file exists in the sample directory. The name of the file is *sdk_global_cartridgeName***.xml**. To use the customization registry file to customize the SDK registries, edit the file and then copy it to the directory *Service_Activator_home***\Config\networkprocessor\Custom\Registries**. The next time the Network Processor is restarted, the customizations will override the cartridge's registry entries.

The name of the file, as it exists in the *Service_Activator_home***\Config\networkprocessor\Custom\Registries** directory is

not actually important. It is the name field within the file that indicates which SDK registry the customization will edit.

The customization file must conform to the **cartridge.xsd** schema.

Cisco Sample

A sample custom Cisco registry follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<registry xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.metasolv.com/serviceactivator/networkprocessor/
cartridgeregistry/ ../cartridge.xsd"
xmlns="http://www.metasolv.com/serviceactivator/networkprocessor/cartridgeregistry/">
  <customizations>
    <name>cisco</name>
    <audit>
      <auditTemplate>
        <auditTemplateEntry>
          <auditTemplateFile>com/metasolv/serviceactivator/
cartridges/cisco/units/cu1/audit/auditTemplate.xml</auditTemplateFile>
          <appliesTo>
            <deviceTypes useRegex="true">.*</deviceTypes>
            <osVersions useRegex="true">.*</osVersions>
          </appliesTo>
        </auditTemplateEntry>
      </auditTemplate>
    </audit>
    <messages>
      <success>com/metasolv/serviceactivator/cartridges/cisco/messages /
successMessages.xml</success>
      <warning>com/metasolv/serviceactivator/cartridges/cisco/ messages/
warningMessages.xml</warning>
      <error>com/metasolv/serviceactivator/ cartridges/cisco/messages /
errorMessages.xml</error>
    </messages>
    <capabilities>
      <capabilitiesEntry>
        <capsFile>com/metasolv/serviceactivator/cartridges /cisco/capabilities/
cisco_default.xml</capsFile>
        <appliesTo>
          <deviceTypes useRegex="true">Cisco.*</deviceTypes>
          <osVersions useRegex="true">.*</osVersions>
        </appliesTo>
      </capabilitiesEntry>
    </capabilities>
    <options>
      <optionsEntry>
        <optionsFile>com/metasolv/serviceactivator/cartridges/cisco/options/
cisco_options.xml</optionsFile>
        <appliesTo>
          <deviceTypes useRegex="true">Cisco.*</deviceTypes>
          <osVersions useRegex="true">.*</osVersions>
        </appliesTo>
      </optionsEntry>
    </options>
  </customizations>
</registry>
</registry>
```

Customization File Entries

The relevant parts of the customization file are:

- **AuditTemplate**
  - Specifies a list of **auditTemplateEntry**. Each entry specifies an **auditTemplateFile** and the **deviceTypes** and **osVersions** that the audit template is to be used for.
  - The **auditTemplateFile** specifies the path to the auditTemplate file relative to the directory *Service_Activator_home***\Config\networkprocessor**.
  - When multiple **auditTemplateEntry** are specified, the first matching entry is used for a device. For this reason, more specific device specifications should be placed before less specific device specifications.
  - When a custom auditTemplate entry is specified, it replaces all auditTemplate entries that may have been specified in the default registry. The custom entries are not merged with the default entries; they replace the default entries.
- **AuditQuery**
  - Specifies a list of **auditQueryEntry**. Each entry specifies an **auditQueryFile** and the **deviceTypes** and **osVersions** that the audit query file is to be used for.
  - The **auditQueryFile** specifies the path to the auditQuery file relative to the directory *Service_Activator_home***\Config\networkprocessor**.
  - When multiple **auditQueryEntry** are specified, the first matching entry is used for a device. For this reason, more specific device specifications should be placed before less specific device specifications.
  - When a custom auditQuery entry is specified, it replaces all auditQuery entries that may have been specified in the default registry. The custom entries are not merged with the default entries; they replace the default entries.
  - auditQuery is only be valid for specific cartridges.
- **Success**
  - Specifies a custom success messages file. The path to the file is specified relative to the directory *Service_Activator_home***\Config\networkprocessor**.
- **Warning**
  - Specifies a custom warning messages file. The path to the file is specified relative to the directory *Service_Activator_home***\Config\networkprocessor**.
- **Error**
  - Specifies a custom error messages file. The path to the file is specified relative to the directory *Service_Activator_home***\Config\networkprocessor**.
- **Capabilities**
  - Specifies a list of **capabilitiesEntry**. Each entry specifies a **capabilitiesFile** and the **deviceTypes** and **osVersions** that the capabilities file is to be used for.
  - The **capabilitiesFile** specifies the path to the capabilities file relative to the directory *Service_Activator_home***\Config\networkprocessor**.

  - When multiple **capabilitiesEntry** are specified, the first matching entry is used for a device. For this reason, more specific device specifications should be placed before less specific device specifications.
  - When a custom capabilities entry is specified, it replaces all capabilities entries that may have been specified in the default registry. The custom entries are not merged with the default entries; they replace the default entries.
- **Options**
  - Specifies a list of **optionsEntry**. Each entry specifies an **optionsFile** and the **deviceTypes** and **osVersions** that the options file is to be used for.
  - The **optionsFile** specifies the path to the options file relative to the directory *Service_Activator_home***Config\networkprocessor**.
  - When multiple **optionsEntry** are specified, the first matching entry is used for a device. For this reason, more specific device specifications should be placed before less specific device specifications.
  - When a custom options entry is specified, it replaces all options entries that may have been specified in the default registry. The custom entries are not merged with the default entries; they replace the default entries.

Raising Faults

You can have your cartridge raise a fault to the Network Processor using the method **AddFaultByThreadAndAbort()** which is part of the FaultCollector. This method raises a fault and causes transformation to abort immediately.

The **AddFaultByThreadAndAbort()** method throws an **AbortTransformException** exception.

Since this java exception is invoked in an XQuery, Saxon throws it to syserr. On UNIX, the Network Processor shell script is modified to discard all syserr output.

Completing the Sample Cartridge Source Files

To complete the sample cartridge source files:

1. Do one of the following:
   - Copy the files provided in *SDK_home***samples\baseCartridge\cisco** over their counterparts in the generated source directory (*SDK_home***baseCartridge\cisco**)

     or
   - Edit the generated sample source files to complete their content development.

     The provided files demonstrate the edits required to complete the generated sample source to produce a working sample base cartridge.

     You can examine the contents of the sample files and by highlighting in some manner (change bars, etc.), you can observe what was added, or modified to complete the sample.

The files to be copied or edited are:

- **successMessages.xml**
- **dm2cli-common.xq.xml**

- **auditTemplate.xml**

Compiling the Cartridge

Base cartridge source files are compiled using ant. The compilation process creates the required XML beans for the cartridge and packages them into a .zip file.

> **Note:**
>
> An existing CLASSPATH environment variable may interfere with the CLASSPATH required by the SDK. It is therefore recommended that the CLASSPATH environment variable be unset in the session where the SDK is being used. For example:
>
> ```
> set CLASSPATH=
> ```

To compile the cartridge:

1. Set the cartridge version string variable. For example, if the cartridge version is 1.0, on a Windows host, type the command:

   ```
   set VERSION_STRING=1.0
   ```

2. Compile the ciscoBanner sample service cartridge source files using the following command:

   ```
   ant package -f SDK_home\baseCartridges\cisco\build.xml
   ```

3. Once you have completed editing your base cartridge source files, compile them using the following command:

   ```
   ant package -f SDK_home\baseCartridges\sdk_global_cartridgeName\build.xml
   ```

   This results in the following additions to the cartridge directory structure:

   ```
   SDK_home\baseCartridges\sdk_global_cartridgeName\
   build.xml
   AuditTrailsReports
   beansrc
   classes
   lib
     sdk_global_cartridgeName.jar
     sdk_global_cartridgeNametests.jar
   package
     sdk_global_cartridgeName-baseCartridge-${env.VERSION_STRING}.zip
       sdk_global_cartridgeName-baseCartridge-${env.VERSION_STRING}.manifest
   ```

Troubleshooting Cartridge Compilation

Compilation problems are caused by schema or XQuery errors. To debug these problems, load the schema into an XML schema aware editor. This makes it much easier to find and correct problems in the schema.

Manifest File

When an SDK cartridge is built, a manifest file is created listing all of the files that are packaged into the cartridge zip file. Installation of the cartridge places the manifest file into the uninstall directory of the IP Service Activator installation.

Implementing Pre- and Post-Checks

Pre- and post-checks provide the ability verify information on a device when the annotated DM to CLI transform executes, before the general configuration is sent. This allows you to confirm that prerequisites to the configuration are met.

After configuration is sent, you have the opportunity to have a post-check invoked to verify some aspect of the commands that were sent to the device.

For further information on pre- and post-checks, see *IP Service Activator SDK Developer Overview Guide*.

Testing in a Standalone Environment

Test scripts are created as part of the base cartridge skeleton generation process. For more information, see "Generating Cartridge Source Files".

Unit Tests

The unit test is generated with the skeleton cartridge source files.

To run unit tests:

1. After you have compiled the cartridge, enter the following command:

```
ant unittests -f=SDK_home\baseCartridges\sdk_global_cartridgeName\build.xml
```

This runs tests which are intended to prove that the main transform stages of the cartridge (i.e. service model to device model and annotated device model to CLI) will generate the output documents correctly.

Device Tests

The device test is generated with the skeleton cartridge source files. Its configuration can be modified after the skeleton has been generated. The device test configuration file is:

*SDK_home***cartridges***sdk_global_cartridgeName***test\devicetests\DeviceTests.properties**

DeviceTests.properties contains parameters for running the device test, including the host to connect to and the userid and password.

To run device tests:

1. Enter the following command:

```
ant devicetests
SDK_home\cartridges\sdk_global_cartridgeName\test\devicetests\DeviceTests.properties
```

The device tests validate that the login and prompt checks are correct for the actual device. If any are incorrect, one of the following errors results:

- Incorrect login prompt
- Incorrect password prompt
- Timeout attempting to read the default prompt. This indicates a problem with the promptMatchPattern

> **Note:**
>
> When managing a Cisco device with old alias information for **IpsaConfigVersion** configured on the device, the device state changes to Intervention Required due to IP Service Activator validation pre-check. If this behavior is undesired during device testing, manually remove the alias command from the device, delete the error message in the IP Service Activator client, and re-commit the transaction.

Troubleshooting the Standalone Tests

Typical login and prompt problems include:

- Connectivity to the device.

  If there are problems connecting to the device, ensure the device supports telnet connections to the default port (23).
- If the command executor is unable to determine the login or password prompt, ensure its regular expression matches the responses seen when logging in manually.
- Timeouts waiting for prompts.

The command executor can time-out waiting for a prompt if the regular expression for it is incorrect. Ensure the regular expression matches the device responses correctly.

Success/error message problems:

- Errors while sending commands.

  If errors are encountered while sending commands, the device may be sending responses that are not defined in the success messages file. This can apply to any non-login command sent to the device. Any responses outside of the prompt that are not defined in the success messages file are considered errors.

Testing in an IP Service Activator Environment

To run the cartridge in an IP Service Activator environment:

1. Unzip the cartridge file *sdk_global_cartridgeName***-baseCartridge-${env.VERSION_STRING}.zip** to the runtime environment of the Network Processor *Network_Processor_home*.
2. Restart the network processor to load *sdk_global_cartridgeName***.jar**.

To observe the cartridge loading operation see:

- *Service_Activator_home***logs\networkProcessor.log**
- *Service_Activator_home***AuditTrails\np***sdk_global_cartridgeName*.**log**

Verification of Deployment

Once the Network Processor has started, it will raise information faults in the system indicating each cartridge registered. The new base cartridge should be indicated. If this does not happen, check the network processor log; it will contain the details on why the cartridge was not loaded. If the log does not indicate the problem, check that the cartridge was deployed to the correct location.

Free-form Testing Using CTM

If the CTM module is installed, it can be used to send commands to the base cartridge.

To configure CTM:

1. Find TABLE in database called XTM_DRIVER_TYPE
2. Add the name of your new cartridge's driverType to the list

   For example, from the sqlplus command line:

   ```
   insert into xtm_driver_type values ('cisco');
   ```

Problem Isolation and Resolution

Problems with the base cartridge should fall into one of the following categories:

- Environmental/loading of the base cartridge
- Cartridge transform errors: if the transform has been customized, it must produce the correct results. If not, this can cause configuration to be rejected and faults to be raised.
- Device discovery: ensure that the core IP Service Activator system can discover the device. If the SNMP discovery works but the capabilities fetch fails, this can indicate the cartridge is not registered for the correct device type/OS combination.
- Device access: ensure the correct session type (SSH/telnet) has been selected as well as the correct userid and password. If problems persist, ensure the same prompts are used in the device tests. These can be used to help isolate the problem.
- Command problems:
  - The auditTrail log shows commands being sent to the device. Use the auditTrail log to identify commands begin sent and whether or not they are being accepted.
  - Attempt the same commands manually on the device. Take note of any responses from the device. These may need to be added to the success messages for the cartridges.

Audit Trail Logging

Audit trail logging records the commands sent to devices by the base cartridge, and any service cartridges that extend the services of the base cartridge.

Setting Audit Trail Logging Properties

The default audit logging properties are defined in **auditLogging.properties** in the cartridge jar file.

Once the cartridge is deployed, the audit logging properties can be overridden by the system administrator. For this purpose, the cartridge .zip file includes a copy of the auditLogging.properties file in the vendor specific sampleRegistry directory located in:

*Service_Activator_home***Config\networkProcessor***sdk_global_cartridgeName***SampleRegistry\auditLogging.properties**

where *sdk_global_cartridgeName* is the name of the cartridge.

When the skeleton cartridge source files are generated, occurrences of *sdk_global_deviceNameLowerCase*, as shown below in the source **auditLogging.properties** template file, are replaced with value of the property sdk_global_deviceName converted to all lowercase characters. The property sdk_global_deviceName is a property in the **skeleton.properties** file. For example, if `sdk_global_deviceName=Cisco` then **cisco** will be substituted for *sdk_global_deviceNameLowerCase* in the output auditLogging.properties file.

To set different default values for the audit logging properties:

1. edit auditLogging.properties. For example:

```
# Audit output
#  There are alternatives for file rollover.
#
#  The current default is to rollover a log when it reaches 8MB.
#
log4j.appender.sdk_global_deviceNameLowerCaseAudit=org.apache.log4j.RollingFi
leAppender
# log4j.appender.sdk_global_deviceNameLowerCaseAudit.MaxFileSize=8MB
# log4j.appender.sdk_global_deviceNameLowerCaseAudit.MaxBackupIndex=1
#
#   An alternative would be to rollover at the end of each day.
#   To do this, replace the 3 lines shown above with the following.
#
log4j.appender.sdk_global_deviceNameLowerCaseAudit=org.apache.log4j.DailyRoll
ingFileAppender
# log4j.appender.sdk_global_deviceNameLowerCaseAudit.DatePattern='.'yyyy-MM-
dd
#
log4j.loggerFactory=com.metasolv.serviceactivator.util.logging.TraceLoggerFac
tory
log4j.logger.com.metasolv.serviceactivator.networkprocessor.AuditLogger.sdk_g
lobal_deviceNameLowerCase=info, sdk_global_deviceNameLowerCaseAudit
log4j.appender.sdk_global_deviceNameLowerCaseAudit=org.apache.log4j.RollingFi
leAppender
log4j.appender.sdk_global_deviceNameLowerCaseAudit.File=AuditTrails/
npsdk_global_cartridgeName.audit.log
log4j.appender.sdk_global_cdeviceNameLowerCaseAudit.MaxFileSize=8MB
log4j.appender.sdk_global_cdeviceNameLowerCaseAudit.MaxBackupIndex=1
log4j.appender.sdk_global_deviceNameLowerCaseAudit.layout=org.apache.log4j.Pa
tternLayout
log4j.appender.sdk_global_deviceNameLowerCaseAudit.layout.ConversionPattern=%
d{yyyy-MM-dd HH:mm:ss}%m%n
```

To override the default audit logging properties once the cartridge is deployed:

1. Append the contents of **auditLogging.properties** found in the install directory at

   *Service_Activator_home***Config\networkProcessor***sdk_global_cartridgeName**Sample Registry\auditLogging.properties**

   to the logging.properties file located in

   *Service_Activator_home***Config\networkProcessor\com\metasolv\serviceactivator\networkprocessor\logging.properties**

2. Adjust audit logging properties as necessary.

Adjusting the Audit Trail Logging Level

To change the audit trail logging level from debug to info:

1. Modify the line:

   ```
   log4j.logger.com.metasolv.serviceactivator.networkprocessor.AuditLogger.sdk_global_cartridgeName=debug, ${sdk_global_cartridgeName}Audit
   ```

   to:

   ```
   log4j.logger.com.metasolv.serviceactivator.networkprocessor.AuditLogger.sdk_global_cartridgeName=info, sdk_global_cartridgeNameAudit
   ```

Adjusting the Audit Trail Log File Size and Number of Previous Versions

To change the audit log file size:

1. Modify the value of MaxFileSize. For Example:

   ```
   log4j.appender.sdk_global_cartridgeNameAudit.MaxFileSize=8MB
   ```

To change the number of previous versions of the audit log file:

1. Modify the value of MaxBackupIndex. For example:

   ```
   log4j.appender.sdk_global_cartridgeNameAudit.MaxBackupIndex=1
   ```

Adjusting the Audit Trail Log File Rollover Strategy

IP Service Activator uses a rollover strategy to prevent log files from becoming too large.

To configure a log file to roll over when it reaches a specified size:

1. Use the following settings in the **auditLogging.properties** file:

   ```
   log4j.appender.sdk_global_cartridgeNameAudit=org.apache.log4j.RollingFileAppender
   log4j.appender.sdk_global_cartridgeNameAudit.MaxFileSize=8MB
   log4j.appender.sdk_global_cartridgeNameAudit.MaxBackupIndex=1
   ```

To configure a log file to roll over at the end of each day:

1. Use the following settings in the **auditLogging.properties** file:

   ```
   log4j.appender.sdk_global_cartridgeNameAudit=org.apache.log4j.DailyRollingFileAppender
   log4j.appender.sdk_global_cartridgeNameAudit.DatePattern='.'yyyy-MM-dd
   ```

Device Model Upgrades

Once a cartridge is constructed and deployed, it will carry with it a device model version identifier (e.g 1.0). If a subsequent release of the cartridge is constructed which involves a non-trivial device model change, then the device model version would be incremented to 2.0, as an example, to distinguish it from the predecessor cartridge.

In order to deploy the updated cartridge into the production environment, an upgrade path for any existing persisted device models is required.

The cartridge developer is responsible for identifying that a device model upgrade is required, and if so:

- Updating the cartridge version
- Updating and unit testing the upgrade source code related to the modified device model

The following sections describe this process in detail.

Identifying that a Device Model Upgrade is Required

Once a cartridge is deployed in production, a non-trivial change to the **devicemodel.xsd** in the next version of the cartridge will require a custom upgrade transform.

Examples of non-trivial changes are:

- You introduce mandatory elements
- You remove elements
- You completely modify an existing elements
- You reorder elements

Updating the Cartridge Version

When you create a cartridge, within the **skeleton.properties** file, the cartridge version is set in the <sdk_cartridgeVersion> property. This cartridge version is used to identify the version of the device model as seen in the cartridge file:

*SDK_Home***basecartridges\cisco\src\com\metasolv\serviceactivator\cartridges\cisco\xquerylib\dm-version.xq**

Increment the cartridge version whenever a significant change to the device model has occurred which requires a non-trivial upgrade to an existing device model. For more information about Device Model upgrades, see *IP Service Activator SDK Developer Overview Guide*.

The cartridge version is set in the file **dm-version.xq**:

```
declare variable $dmver:version := "2.0";
```

The version format can be a, a.b, or a.b.c (i.e. major, major.minor, major.minor.sub-minor), where a, b, c represent numeric values.

Updating the Device Model Upgrade Transform

When you use the cartridge skeleton generator tool to generate the base cartridge source files, a skeleton XQuery file called DmUpgrade.xq is automatically created.

To implement the upgrade transform:

1. Edit **DmUpgrade.xq**.

   For example, if the device model has an existing element named 'myelement' which requires an upgrade transform, implement the upgrade transform in **DmUpgrade.xq** as follows:

```
(: -*- nxml -*- :)
  import module namespace dmver = "dm-<sdk_global_cartridgeName>-version" at "dm-
version.xq";

declare namespace dm="http://www.metasolv.com/serviceactivator/
<sdk_global_cartridgeName>";
  declare namespace lib="http://www.metasolv.com/serviceactivator/devicemodel";

<lib:device xsi:type="dm:<sdk_global_cartridgeName>Device" xmlns:dm="http://
www.metasolv.com/serviceactivator/<sdk_global_cartridgeName>">
  {
    dmver:getUpgradeAppInfo(),
    for $element in /lib:device/*
    return
      if (fn:local-name($element) = 'appInfo') then ()
      else if (fn:local-name($element) = 'myelement') then
      (: implement transform for 'myelement' here :)
      else $element
  }
</lib:device>
```

Unit Testing the Device Model Upgrade Transform

When you use the cartridge skeleton generator tool to generate the base cartridge source files, a skeleton java file named **DmUpgradeTests.java** is automatically created. This file contains the device model upgrade unit tests.

To test the upgrade transform for an existing element in the device model named 'myelement' that requires an upgrade transform implemented in DmUpgrade.xq:

1. Create a sample device model XML file that includes data requiring an upgrade. In the sample code below, the file is named **sampleDeviceModel.xml**.
2. Implement a test in **DmUpgradeTests.java**.

   The test will load **sampleDeviceModel.xml**, transform the XML using the upgrade transform, and use methods in BaseSdkTest to verify the resulting upgraded device model, thereby validating the upgrade transform.

   In the sample code below, the test method is named **testUpgradeFromVersionX**.
3. Run the unit tests using the following command syntax:

```
ant unittests SDK_home\cartridges\sdk_global_cartridgeNamebuild.xml
```

A code sample follows:

```
  package ${sdk_global_package}.test;

  import com.metasolv.serviceactivator.sdk.test.TestUtilsInterface;
  import com.metasolv.serviceactivator.sdk.test.BaseSdkTest;
  import com.metasolv.serviceactivator.util.XmlUtil;
  import org.apache.xmlbeans.XmlObject;
  import java.io.File;

  public class DmUpgradeTests extends BaseSdkTest {
    static private final String testUpgradeFromPackageName = "$
{sdk_global_customPackage}/test/models/upgradeFrom";
    public static final String dmUpgradeTransform = "${sdk_global_customPackage}/
xquerylib/DmUpgrade.xq";
    static private final String upgradeFromDeviceModelXmlFile =
testUpgradeFromPackageName + "/sampleDeviceModel.xml";
    static private TestUtilsInterface testUtils = null;

    protected void setUp() throws Exception {
        testUtils = getTestUtils();
        }

    public void testUpgradeFromVersionX() throws Exception {
      XmlObject oldDm = loadXml(DmUpgradeTests.upgradeFromDeviceModelXmlFile);
      XmlObject upgradedDm = transform(oldDm, null,
DmUpgradeTests.dmUpgradeTransform, false);
  //    System.out.println("Transformed:\n" + XmlUtil.bean2xml(upgradedDm));

   // use methods in BaseSdkTest to verify upgradedDm here
    }
 }
```

Network Processor NpUpgrade

DMUpgrade.xq is executed by the Network Processor when an upgrade procedure is invoked.

Audit

Audit functionality is controlled by an audit template and an audit query file. The names of these files are specified in the **Registry.xml** file.

The audit template file is used by the audit process for devices provisioned using a command line interface (CLI).

The audit query file is used by the audit process for devices provisioned using an XML interface. Different audit template and audit query files can be specified for different device types and OS versions.

For complete details on audit, refer to *IP Service Activator SDK Developer Overview Guide*.

Uninstalling Base Cartridges

Cartridges are uninstalled using the **uninstallCartridge.sh** script, which resides in the bin directory of the IP Service Activator installation. This script takes the name of the manifest file, which contains a list of all installed cartridge files, as a parameter, and uses its contents to uninstall the cartridge. (See "Manifest File".)

You can include the base directory of the IP Service Activator installation as a parameter to the script. If you do not, the script queries the ORCHcore package to locate the base directory of the IP Service Activator installation.

The **uninstallCartridge.sh** script sorts the manifest file in reverse order, then deletes files, and then directories. Only empty directories are removed; this ensures that the script will not remove directories used by other cartridges.

You can use a relative path to specify the manifest file, but it must be relative to the current directory (where you are running the uninstall script from). You can also use an absolute path. To verify that the manifest file is in the directory, use the command “ls<manifest>" using the same value that is provided to the script.

To uninstall the base cartridge:

1. Enter the following command:

   ```
   uninstallCartridge manifest_file [Service_Activator_Home] [-k | -v]
   ```

   Use the -k option to leave empty directories. The -v (verbose) option produces extra output from the script.
2. After the cartridge is uninstalled, restart the Network Processor.

> **Note:**
>
> Uninstalling a cartridge or configuration policy developed using the SDK does not remove the Network Processor's device model entries that reference this cartridge or configuration policy. This information is maintained because it is unknown whether you are uninstalling the cartridge or configuration policy to remove it or to upgrade it.

Removing a Generated Cartridge from the SDK

To remove a generated cartridge from the SDK installation:

1. Delete all contents under *SDK_home***cartridges***sdk_global_cartridgeName*.

Uninstalling the SDK

To uninstall the SDK:

1. Delete all contents under *SDK_home*.

A

Base Cartridge Generation Properties

This appendix provides details on the parameters you can configure in the **skeleton.properties** file used to generate base cartridge source files.

This file contains a number of properties that customize the generated base cartridge source.

Property names are of the form sdk_*context*_*type* and are composed of three parts:

- sdk: indicates an SDK variable
- *context*: describes of the context in which the variable applies
- *type*: indicates how the variable is being used, and may imply a restriction on the possible values:
  - If **supported** appears in the *type*, a boolean value should be entered.
  - If **pattern** appears in the *type*, a regular expression (regex) pattern should be entered.
  - If **prompt** appears in the *type*, a device response should be entered in the form of a regex pattern.
  - If **cmd** appears in the *type*, a device specific command should be entered.

Boolean variables are validated to ensure that the values conform to boolean values (**true** or **false**).

Regex patterns are validated to ensure that they can be compiled.

> **Note:**
>
> For certain regular expressions in the **skeleton.properties** file, it maybe necessary to use an escape character to precede certain special characters in order for them to be translated to the generated source code correctly. This is dependent on whether you are using XQuery or Java based transforms. For example, many regexes specifying prompt string matches appear in **dm2cli-common.xq**.

Table A-1 shows the naming and packaging properties.

Table A-1 Naming and Packaging Properties

Property	Description	Example
sdk_global_cartridgeName	This is the cartridge name. This variable is used throughout the cartridge code in generating file names and source code variable names. This property is mandatory.	`cisco`

Table A-1 (Cont.) Naming and Packaging Properties

Property	Description	Example
sdk_global_cartridgeVersion	This is the cartridge version that is being developed. It is used at run time to verify that a device model is still valid in the event of an upgrade of the cartridge. This property is mandatory.	`1.0`
sdk_global_package	This is the cartridge path in dotted notation used for packaging. Its value is translated to a directory structure for the source files path generation. The value is used in build scripts, java source code and support files. The generated files are placed in: *SDK_home***baseCartridges***sdk_global_cartridgeName***src***sdk_global_package* This property is mandatory.	`com.metasolv.serviceactivator.cisco` becomes `com\metasolv\serviceactivator\cisco`

Table A-2 shows the device type identification properties used in the JUnit test environment.

Table A-2 Device Type Identification Properties

Property	Description	Example
sdk_global_deviceName	Device name as it will be used with the **Registry.xml** for which this cartridge is being constructed. This property is a key property that is used to assign the driverType value for this base cartridge. The driverType value will be set to the value of this property converted to all lowercase characters. For more details about base cartridge **Registry.xml**, see *IP Service Activator SDK Developer Overview Guide* for more details. This property is mandatory.	`Cisco`
sdk_global_deviceDescription	Device description as it will be used with the Service Model for which this cartridge is being constructed. This property is mandatory.	`Cisco Internetwork Operating System Software IOS (tm) RSP Software (RSP-PV-M), Version 12.2(8)T, RELEASE SOFTWARE (fc2) TAC`
sdk_global_deviceModel	Device Model that this cartridge is being constructed for. Used in **Registry.xml** and in the test environment. This property is mandatory	`2611`
sdk_global_deviceVersion	Device version that this cartridge is being constructed for. This property is mandatory.	`12.2(11)T8`

Table A-3 shows the Device Model schema properties.

Table A-3 Device Model Schema Properties

Property	Description	Example
sdk_deviceModel_namespace	Target namespace of the device model schema for this base cartridge. This property is mandatory.	--
sdk_deviceModel_namespaceAbbr	Abbreviation of the target namespace of the device model schema for this base cartridge. This is used as a namespace prefix. This property is mandatory.	`dmcisco`
sdk_deviceModel_prefix	A complex type with the name *sdk_deviceModel_prefix*Device which extends BaseDevice will be generated in the deviceModel schema for this base cartridge. This property is mandatory.	Cisco becomes `CiscoDevice`

Table A-4 shows the options schema properties.

Table A-4 Options Schema Properties

Property	Description	Example
sdk_options_namespace	Target namespace of the options schema for this service cartridge. This property is mandatory.	--
sdk_options_namespaceAbbr	Abbreviation of the target namespace of the options schema for this cartridge. This is used as a namespace prefix. This property is mandatory.	`ciscopt`

Table A-5 shows the test environment properties.

> **Note:**
>
> Be aware of the final usage of properties and code appropriately - they could end up in either **dm2cli-common.xq** or in java.

Table A-5 Test Environment Properties

Property	Description	Example
sdk_test_userName	Login userid for device to be used in testing the cartridge. **Note:** TACACS access is only supported for testing. This property is mandatory	`userid`

Table A-5 (Cont.) Test Environment Properties

Property	Description	Example
sdk_test_userPasswd	Login password for device to be used in testing the cartridge. **Note:** TACACS access is only supported for testing. This property is mandatory.	`userPasswd`
sdk_test_hostip	IP address for the device to be used in testing the cartridge. This property is mandatory.	`2.2.2.2`
sdk_test_cmd	A simple command to be used to determine if device access is available. This property is mandatory.	`show run`
sdk_test_matchPattern	Pattern to match the general device prompt. The sample matchPattern matches zero or more characters with the exception of: >, #, or \n (newline), and is followed by either one of > or #. The group of characters preceding the > or # is the hostname. **Note:** You must use a syntax that conforms to java. This property is mandatory.	`([^>#\\n]*)[>#]`
sdk_test_prePendPattern	Pattern that can prefix the general prompt. The prepend pattern is prepended to the prompt pattern. It should be used if the prompt has something at its beginning that must be explicitly matched. The sample prepend pattern matches \n (the newline character). **Note:** You must use a syntax that conforms to java. This property is mandatory.	`\\n`
sdk_test_appendPattern	Pattern to match the **config t** prompt. The append pattern is appended to the prompt pattern. It should be used if the prompt has something at its end that must be explicitly matched. The sample append pattern matches one of the following: >, #, or zero or more characters in parentheses followed by # **Note:** You must use a syntax that conforms to java. This property is mandatory.	`(([>#])|(\\\\(.*\\\\)#))`

Table A-6 shows the properties to support saving of the running configuration.

Table A-6 Properties to Support Saving of the Running Configuration

Property	Description	Example
sdk_saveRunningConfig_supported	Boolean value to indicate if running config should be saved to the device after each configuration change. This property is mandatory.	`True`

Table A-6 (Cont.) Properties to Support Saving of the Running Configuration

Property	Description	Example
sdk_saveRunningConfig_cmd	Command to send to the device to save running config. This property is optional.	`copy running-config startu-config`
sdk_startUpSavedConfig_timeout	Timeout in seconds for sdk_start command. This property is optional.	`600`

Table A-7 shows the Audit properties.

Table A-7 Audit Properties

Property	Description	Example
sdk_audit_supported	Command sent to the device to determine if device is supported. This property is mandatory	`True`
sdk_auditTerminalLengthZero_supported	Boolean value to indicate if the device needs to have a command sent that sets the terminal length to 0. This property is mandatory.	`True`
sdk_auditTerminalLengthZero_cmd	Command to send to the device to set terminal length to 0. This property is optional.	`terminal length 0`
sdk_auditShowRunningConfig_cmd	Command to instruct the device to display all of the running configuration This property is optional.	`show running-config`
sdk_auditShowRunningConfig_conditionalPrompt	Regular expression which matches the returned running configuration - the successful response to **sdk_auditShowRunningConfig_cmd** (see above). The example matches multiple lines of text consisting of any characters followed by the text `end` on a separate line followed by zero or more characters. This property is optional.	`.*(?m)^end$.*`
sdk_auditShowCommandsLogout_cmd	Command to send to the device to log out after executing **sdk_auditshowRunningConfig_cmd**. This property is optional	`Logout`
sdk_auditShowCommandsLogout_conditionalPrompt	Regular expression to match the device's successful response after executing the **sdk_auditShowCommandsLogout_cmd** to logout after displaying the running configuration. The example matches any character zero or more times This property is optional.	`.*`

Table A-8 shows the Restore properties.

Table A-8 Restore Properties

Property	Description	Example
sdk_restore_supported	Boolean value to indicate if the device supports the ability to copy all configuration from a network server to the device startup configuration. This property is optional.	`True`
sdk_restore_copyTftpCmd	Command to send to the device to copy all configuration from a network server. This property is optional.	`copy tftp startup-config`
sdk_restore_copyTftpAddressPrompt	Device prompt for the location after **executing sdk_restore_copyTftpCmd**. This property is optional.	`.*Address or name of remote host.*`
sdk_restore_copyTftpSourceFilePrompt	Device prompt for the source after executing **sdk_restore_copyTftpCmd** This property is optional.	`.*Source filename.*`
sdk_restore_copyTftpDestinationFilePrompt	Device prompt for the destination after executing **sdk_restore_copyTftpCmd**. This property is optional.	`.*Destination filename.*`
sdk_restore_copyTftpDestinationFileCmd	Destination to send to the device to set the destination for the **sdk_restore_copyTftpCmd** command to be the device startup configuration This property is optional.	`startup-config`
sdk_restore_reloadCmd	Command to send to the device to reload the operating system after all configuration is restored to the startup configuration. This property is optional.	`reload`
sdk_restore_reloadPrompt	Device prompt for confirmation to proceed with reloading the operating system after executing **sdk_restore_reloadCmd**. This property is optional.	`.*Proceed with reload.*`
sdk_restore_reloadConfirmationCmd	Command to send to the device to confirm that the reload should proceed after receiving the device prompt **sdk_restore_reloadPrompt**. This property is optional.	`y`

Table A-9 shows the properties for support of Network Processor DM synchronization with the device.

Table A-9 Properties for Support of Network Processor DM Synchronization with the Device

Property	Description	Example
sdk_configversion_supported	Boolean value to indicate if the cartridge should support the ability to specify configuration versions. This property is mandatory	`True`

Table A-9 (Cont.) Properties for Support of Network Processor DM Synchronization with the Device

Property	Description	Example
sdk_configversion_text	The text that specifies the configuration version on the device. When reading back the configuration version, the configuration version is found by searching for the variable's value. This property is optional.	`IpsaConfigVersion`
sdk_configversion_updateCmd	Prefix of the CLI command that updates the configuration version on the device. The entire command is formed by appending the configuration version to it. This propety is optional.	`alias exec IpsaConfigVersion`
sdk_configversion_removeCmd	Prefix of the CLI command that removes the configuration version from the device. The entire command is formed by appending the configuration version to it. This property is optional.	`no alias execc IpsaConfigVersion`
sdk_configversion_showCmd	The CLI command that retrieves the configuration version from the device. The result from this command is expected to be a single line which among other tokens must contain the configuration version. This property is optional.	`show aliases exec | include IpsaConfigVersion`
sdk_configversion_extractCmd	This is a regular expression that when applied against to the result from **sdk_configversion_showCmd** must match, so that its first group is bound to the configuration version string. This property is optional	`IpsaConfigVersion +(.+)`

Table A-10 shows Prompt Matching properties.

Table A-10 Prompt Matching Properties

Property	Description	Example
sdk_getConfiguration_matchPattern	Pattern to match the general device prompt. **Note:** You must use syntax that conforms with XM This property is optional.	`([^>#\\n]*) [>#]`
sdk_getConfiguration_prePendPattern	Pattern that can prefix the general prompt. **Note**: You must use syntax that conforms with XML. This property is optional.	`\\n`
sdk_getConfiguration_appendPattern	Pattern to match the config t prompt **Note**: You must use syntax that conforms with XML. This property is optional.	`(([>#])|(\\(.*\ \)#))`

Table A-10 (Cont.) Prompt Matching Properties

Property	Description	Example
sdk_getConfiguration_errorPattern	Prompt from device if login failed. This property is optional.	`.*Username:`
sdk_getConfiguration_hostPattern	Pattern to match the hostname within the prompt **Note**: You must use syntax that conforms with XML. This property is optional.	`[\\w\\.\\-]+`

Table A-11 shows the TACACS authentication support properties.

Table A-11 TACACS Authentication Support Properties

Property	Description	Example
sdk_authenticationTacacs_supported	Boolean value to indicate if cartridge is to support this type of login access to device. This property is mandatory.	`True`
sdk_authenticationTacacs_useridPrompt	Prompt from device to enter userid. This property is optional.	`.*Username:`
sdk_authenticationTacacs_passwdPrompt	Prompt from device to enter password. This property is optional.	`.*Password:`
sdk_authenticationTacacs_errorPrompt	Prompt from device if userid or password is bad. This property is optional.	`.*Username:`
sdk_authenticationTacacs_enablePasswdPrompt	Prompt from device to enter enable userid if applicable. This property is optional.	`.+`
sdk_authenticationTacacs_enablePasswdPasswdPrompt	Prompt from device to enter enable password if applicable. This property is optional.	`.*Password:`
sdk_authenticationTacacs_enablePasswdErrorPrompt	Prompt from device if enable userid\password is bad. This property is optional.	`.*Bad secrets`

Table A-12 shows the SSH authentication support properties.

Table A-12 SSH Authentication Support Properties

Property	Description	Example
sdk_authenticationSsh_supported	Boolean value to indicate if cartridge is to support this type of login access to device. This property is mandatory.	`True`
sdk_authenticationSsh_useridPrompt	Prompt from device to enter userid. This property is optional.	`.+`

Table A-12 (Cont.) SSH Authentication Support Properties

Property	Description	Example
sdk_authenticationSsh_passwdPrompt	Prompt from device to enter password. This property is optional.	`.*Password:`
sdk_authenticationSsh_enablePasswdErrorPrompt	Prompt from device if enable password is bad. This property is optional.	`.*Bad secrets`

Table A-13 shows anonymous authentication support properties.

Table A-13 Anonymous Authentication Support Properties

Property	Description	Example
sdk_authenticationAnonymous_supported	Boolean value to indicate if cartridge is to support this type of login access to device. This property is mandatory.	`True`
sdk_authenticationAnonymous_passwdPrompt	Prompt from device to enter password. This property is optional.	`.*Password:`
sdk_authenticationAnonymous_useridPrompt	Prompt from device to enter the userid. This property is optional.	`.+`
sdk_authenticationAnonymous_enablePasswdPrompt	Prompt from device to enter enable password if applicable. This property is optional.	`.*Password:`
sdk_authenticationAnonymous_enablePasswdErrorPrompt	Prompt from device if enable password is bad. This property is optional.	`.*Bad secrets`

Table A-14 shows the password only authentication support properties.

Table A-14 Password Only Authentication Support Properties

Property	Description	Example
sdk_authenticationPasswordOnly_supported	Boolean value to indicate if cartridge is to support this type of login access to device. This property is mandatory.	`True`
sdk_authenticationPasswordOnly_passwdPrompt	Prompt from device to enter password. This property is optional.	`.*Password:`

> **Note:**
>
> You must set at least one of the **sdk_authentication***authentication_style*_**supported** parameters from to true.

Table A-15 shows the logout properties.

Table A-15 Logout Properties

Property	Description	Example
sdk_logoutCommand_cmd	Command to exit the device session.	`Exit`

Table A-16 shows the configuration behavior properties.

Table A-16 Configuration Behavior Properties

Property	Definition	Example
sdk_configMode_retryNum	Property to tell the network processor how many times to retry a command. This property is mandatory.	`2`
sdk_configMode_waitTime	Property to tell the network processor how long to wait to see if a command is successful or not. This property is mandatory.	`20`
sdk_configMode_cmd	Command to enter the device config entry mode. This property is optional.	`config`
sdk_configModeTerminate_cmd	Command to exit the device config entry mode. This property is optional.	`end`
sdk_testBadRegex_Pattern	This property is not used in the generated skeleton files. It is for testing the SkeletonGenerator to determine if bad regexes are being caught. You can remove the property from the file if desired. This property is optional.	`|)_*`

B

Generated Skeleton Base Cartridge Source Files

This appendix describes the generated base cartridge source files.

About the Generated Skeleton Base Cartridge Source Files

The directory, *SDK_home***samples\baseCartridge***sdk_global_cartridgeName***skeleton.properties**, contains the list of editable properties that control base cartridge construction. For example, *SDK_home***samples\baseCartridge\cisco\skeleton.properties** contains the list of editable properties that control the base cartridge construction for the sample Cisco base Cartridge. This **skeleton.properties** file is fully populated and can be used to construct a Cisco demonstration cartridge.

To generate the sample base cartridge source files using the data from the skeleton properties file:

1. Go to the SDK directory:

   ```
   cd SDK_home
   ```

2. Do one of the following:

   - Run the included batch file to run the cartridge generator script:

     ```
     genbc samples\baseCartridge\cisco\skeleton.properties
     ```

     or

   - Type in the command to run the cartridge generator script:

     ```
     ant -DtemplateType=baseCartridge -
     DpropFile=SDK_home\samples\baseCartridge\cisco\skeleton.properties
     ```

> **Note:**
>
> to use the batch file, you must first add *SDK_home***bin** to your PATH variable where *SDK_home* is the directory where the SDK was installed (typically **C:\Program Files\Oracle Communications\IP Service Activator\ipsaSDK**).

This results in the following directory structure:

```
SDK_home
  logs
    generator.log
  baseCartridges
    sdk_global_cartridgeName
      build.xml
      src
        synonyms.xml
```

```
<sdk_global_package> (com)
  <sdk_global_package> (.metasolv)
    <sdk_global_package> (..serviceactivator)
      <sdk_global_package> (...cartridges)
        <sdk_global_package> (....cisco)
          audit
            auditTemplate.xml
          capabilities
            empty_caps.xml
          messages
            errorMessages.xml
            successMessages.xml
            warningMessages.xml
          options
            options.xsd
          schema
            devicemodel.xsd
          test
            devicetests
              DeviceAccessTests.java
              DeviceTests.properties
            models
              sampleServiceModel.xml
              upgradeFrom
                sampleDeviceModel.xml
            DMUpgradeTests.java
            transformUnitTests.java
          transforms
            annotatedDm2Cli.xq
            dm2cli-precheck.xq
            dm-validation.xq
            restoreTemplate.xml
            sm2dm.xq
          xquerylib
            dm2cli-common.xq
            dmUpgrade.xq
            dm-version.xq
            sm2dm-common.xq
            precheck-cfg-version.xq
          auditLogging.properties
          Customization.xml
          Registry.xml
```

Generated Skeleton Base Cartridge Source File Details

Table B-1 shows the details for the generated skeleton base cartridge source file.

Table B-1 Generated Base Cartridge Source File Details

Component File	Description
synonyms.xml	This file is used to identify the synonyms for configuration commands that are displayed in a different manner after coming back from a device. Specifying a synonym for a command enables the audit functionality to correctly determine if a configuration command sent to the device is equivalent to the command coming back from the device. This file is coordinated with content in **auditTemplate.xml**.

Table B-1 (Cont.) Generated Base Cartridge Source File Details

Component File	Description
auditTemplate.xml	This file is used to identify each and every command pattern that could be sent to the device. This file is used by the audit mechanism to filter out commands that are not administered by the cartridge.
auditLogging.properties	This file is used to control the following attributes of the cartridge **auditTrail log**: • filename • rollover strategy • debug level
Registry.xml	This file is used to register cartridges with the Network Processor. When the Network Processor executes a configuration change on a particular device, it finds the cartridge that administers the device in question through the registration process.
Customization.xml	This file is used to override the **Registry.xml** and serves as a sample.
empty_caps.xml	This file is a aample capabilities file. Capabilities provide privileges to the device and its subordinate interfaces to support various policies. The sample, being empty, will provide no capabilities to the device and it subordinate interfaces. The user needs to provide capability entries in order to provide privileges to the device during the IP Service Activator device discovery process.
errorMessages.xml	This file contains error patterns for commands generated by the base cartridge. If the response from the device matches one of the known error patterns, then a fault (`Error`) is raised against the device itself, all the concretes affected by that transaction are rejected and the partially implemented configuration is rolled back.
warningMessages.xml	This file contains warning patterns (blocking or non-blocking) for commands generated by the base cartridge. If the response from the device matches a non-blocking warning pattern, a fault (`Warning`) is raised. If the response from the device matches a blocking warning pattern, a fault is raised, and all concretes affected by that transaction are rejected and the partially implemented configuration is rolled back.
successMessages.xml	This file contains success patterns for commands generated by the base cartridge. If the device response to sending a command matches a success pattern, or there is no response at all (only a prompt), then the command is considered successful.
options.xsd	This file is a XML schema file that defines the base options elements and types. The cartridge developer must extend this schema with their own cartridge specific options schema. A sample options document is provided when the cartridge generation tool executed.
deviceModel.xsd	This file is an extension of the **base_deviceModel.xsd** owned by the network processor framework. The **deviceModel.xsd** is owned by the cartridge and can be edited to add content to enable the support of new services that the cartridge will be administering.
DeviceTest.properties	This file contains properties used by the device tests. This file can be edited after the generation of skeleton file.
DeviceAccessTests.java	This file is a java class that tests the device connectivity.
sampleServiceModel.xml	This file is a sample service model used for JUnit testing by **TransformUnitTesting.java** and is an instance of a run time service model.
sampleDeviceModel.xml	This file is a sample device model used by **DMUpgradeTests.java**.

Table B-1 (Cont.) Generated Base Cartridge Source File Details

Component File	Description
TransformUnitTests.java	• method `testBasicServiceModelToDeviceModelTransform`: tests the ability to transform the sampleServiceModel to a proper deviceModel • method `testBasicDeviceModelToCommandDocumentAddTransform`: tests the ability to transform the deviceModel to a proper cliDocument which is adding cmds to the device • method `testBasicDeviceModelToCommandDocumentDeleteTransform`: tests the ability to transform the deviceModel to a proper cliDocument which is deleting cmds from the device
DMUpgradeTests.java	This file is used for testing cartridge upgrade scenarios.
sm2dm.xq	This file contains the XQuery source code that transforms a service model to a device model.
annotatedDM2Cli.xq	This file contains the XQuery source code that transforms an annotated device model to a CLI document.
dm2cli-precheck.xq	This file contains the XQuery source code that performs pre-check functionality, which is used by the **annotatedDM2Cli.xq**.
dmValidation.xq	This file contains the XQuery source code providing the ability to raise fault to the system console.
sm2dm-common.xq	This file contains the XQuery source code used to support **sm2dm.xq**.
dm2cli-common.xq	This file contains the XQuery source code used to support **annoatedDM2Cli.xq**.
dmUpgrade.xq	This file contains the XQuery source code used to support executing a DM upgrade if cartridge DM has been enhanced.
dm-version.xq	This file contains the XQuery source code used to identify which cartridge version is in use.
precheck-cfg-version.xq	This file contains the XQuery source code used to write version information to the device in order to ensure that the network processor device model is in sync with the device.