Oracle Communications IP Service Activator™
Version 5.2.4

SDK Service Cartridge Developer Guide

Second Edition
December 2008

Contents

Preface

About this document

The *SDK Service Cartridge Developer Guide* explains how to use the Service Activator SDK to create service cartridges.

It consists of the following chapters:

- Chapter 1: *Overview* This chapter provides a brief overview of the concepts involved in creating service cartridges with the SDK.
- Chapter 2: *Building service cartridges* This chapter discusses service cartridge concepts and explains how to use the SDK to create service cartridges, and integrate them with configuration policies.
- Appendix A: *Service Cartridge generation properties* This Appendix provides details on the parameters you can configure in the skeleton.properties file used to generate service cartridge source files.
- Appendix B: *Generated Skeleton Service Cartridge Source Files* This appendix describes generated service cartridge source files.

Before contacting Oracle Global Customer Support (GCS)

If you have an issue or question, Oracle recommends reviewing the product documentation and articles on MetaLink in the Top Technical Documents section to see if you can find a solution. MetaLink is located at http://metalink.oracle.com.

In addition to MetaLink, product documentation can also be found on the product CDs and in the product set on Oracle E-Delivery.

Within the product documentation, the following publications may contain problem resolutions, work-arounds and troubleshooting information:

- Release Notes
- Oracle Installation and User's Guide
- README files

Contacting Oracle Global Customer Support (GCS)

You can submit, update, and review service requests (SRs) of all severities on MetaLink, which is available 24 hours a day, 7 days a week. For technical issues of an urgent nature, you may call Oracle Global Customer Support (GCS) directly.

Oracle prefers that you use MetaLink to log your SR electronically, but if you need to contact GCS by telephone regarding a new SR, a support engineer will take down the information about your technical issue and then assign the SR to a technical engineer. A technical support representative for the Oracle and/or former MetaSolv products will then contact you.

Note that logging a new SR in a language other than English is only supported during your local country business hours. Outside of your local country business hours, technical issues are supported in English only. All SRs not logged in English outside of your local country business hours will be received the next business day. For broader access to skilled technical support, Oracle recommends logging new SRs in English.

Oracle GCS can be reached locally in each country. Refer to the Oracle website for the support contact information in your country. The Oracle support website is located at http://www.oracle.com/support/contact.html.

Downloading products and documentation

To download the Oracle and/or former MetaSolv products and documentation, go to the Oracle E-Delivery site, located at http://edelivery.oracle.com.

You can purchase a hard copy of Oracle product documentation on the Oracle store site, located at http://oraclestore.oracle.com.

For a complete selection of Oracle documentation, go to the Oracle documentation site, located at http://www.oracle.com/technology/documentation.

Downloading a media pack

To download a media pack from Oracle E-Delivery

1. Go to http://edelivery.oracle.com.
2. Select the appropriate language and click **Continue**.
3. Enter the appropriate **Export Validation** information, accept the license agreements and click **Continue**.
4. For **Product Pack**, select **Oracle Communications Applications**.
5. For **Platform**, select the appropriate platform for your installation.

6. Click **Go**.
7. Select the appropriate media pack and click **Continue**.
8. Click **Download** for the items you wish to download.
9. Follow the installation documentation for each component you wish to install.

Service Activator publications

The Service Activator documentation suite includes a full range of publications. Refer to the Service Activator *Release Notes* for more information.

Service Activator schema online documentation

An online reference is provided containing schema documentation which outlines how to populate and read XML instance documents used by the Service Activator network processor. Configuration policy documentation is included as well.

It is accessed through:

<ServiceActivatorHome>\ipsaSDK\doc\schemaDoc\index.html

More information about accessing the schemaDoc files is provided in the *SDK Installation and Setup Guide*.

Chapter 1

Overview

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

Developing service cartridges with the SDK

A base cartridge provides 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. Refer to the *Base Cartridge Developer Guide* for details.

Base cartridges do not contain implementations of services. Additional services targeting specific vendor device types are added through integrated service cartridges.

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

Core cartridges

Note that Service Activator legacy cartridges, known as 'core' cartridges, included 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 and configuration instructions, refer to the *Software Development Kit 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 the SDK Overview and Setup Guide.

Chapter 2

Building service cartridges

This chapter discusses service cartridge concepts and explains how to use the SDK to create service cartridges, and integrate them with configuration policies.

References are made to sample service cartridge components packaged with the SDK.

Note: Refer to the Appendix A for details on all properties in the properties file.

This guide assumes:

- that Service Activator is deployed to a directory which will be referred to as `<ServiceActivatorHome>`. This directory is typically `C:\Program Files\Oracle Communications\IP Service Activator`
- that you have successfully installed the SDK to a directory which will be referred to as `<SDKHome>`.
- that the required versions of additional 3rd 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 3rd party tool versions, see the SDK Overview and Setup Guide.

General procedure to build a service cartridge

This section lists the steps required to build a service cartridge. A brief introduction to these steps follows. After that, each step is covered in detail.

The steps to build an XQuery-based service cartridge are:

- Creating a service cartridge source directory and skeleton properties file
- Defining the service and customizing the properties file
- Generating the cartridge source files
- Customizing the cartridge source files
- Compiling and packaging the cartridge
- Performing unit tests
- Performing end-to-end tests

The steps to build a Java-based service cartridge are:

Note: Service cartridges which use java-based transforms do not make use of the skeleton.properties file, or the source code generation capabilities of the SDK. Instead, sample source code is provided for you to base your project on.

- Copy one of the sets of sample source files to support your new development
- Customizing the cartridge source files
- Compiling and packaging the cartridge
- Performing unit tests
- Performing end-to-end tests

Creating a service cartridge source directory and skeleton properties file

To begin creating an XQuery-based service cartridge, you will need to establish a directory structure for the source files, and create the skeleton properties file that will be used to generate the starting source files.

Defining the service and customizing the properties file

This step involves determining the specific details of the service to be created and specifying the information needed to apply the desired configuration to the device.

You will need to select at least one modeled service, or configuration policy to be implemented by the service cartridge. As a starting point, you can specify one modeled service or one configuration policy as a subscription in the cartridge

skeleton properties file. This file is then used by the SDK service cartridge generator tool to generate a set of source files as a starting point for your service cartridge. The generated source files can be customized to add any additional subscriptions if the service cartridge will implement more than one modeled service and/or more than one configuration policy.

Each cartridge you create with the SDK will have a skeleton properties file. In this file, you will customize the property values that control the generation of the cartridge source files. For complete details on all properties, refer to Appendix A, *Service Cartridge generation properties* on page 31.

About Java based service cartridges

There are some special considerations to consider when creating service cartridges which use Java-based transforms as opposed to XQuery-based transforms. They do not make use of the skeleton.properties file, or the source code generation capabilities of the SDK. Instead, sample source code is provided for you to base your project on.

The parts of the procedures in this chapter dealing with XQuery, the skeleton.properties file, and generating and customizing XQuery cartridge source files in particular, do not directly apply to Java-based service cartridge development.

About Alcatel service cartridges

There are some special considerations to consider when creating service cartridges for Alcatel devices. In general terms, the concepts and procedures described in this guide apply, but certain you will have to take additional things into consideration for Alcatel service cartridges. Alcatel service cartridges should use Java-based transforms. (See *About Java based service cartridges* above.)

For details on building Alcatel service cartridges, refer to Appendix C, *Building Alcatel SAM service cartridges* on page 47.

Generating the 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 the majority of your development work takes place for XQuery-based service cartridges. Some of the key cartridge components you need to create are:

- Device Model (DM) schema definition
- Service Model (SM) to DM transform

- 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

Use the included script to compile and package the cartridge.

Performing unit tests

Unit tests are provided as part of the generated source files.

Performing end-to-end tests

Deploy the cartridge into a test Service Activator environment as an extension to a base or core cartridge to perform end-to-end testing.

About the provided sample service cartridges

The SDK includes a number of sample service cartridges.

XQuery-based sample service cartridges:

- ciscoBanner
- ciscoMartini
- ciscoStaticRoute

Java-based sample service cartridges:

- alcatelSamStaticRoute
- ciscoBannerJava

Purpose of the samples

You can use the samples in a variety of ways:

- you can inspect generated source files to see how a simple, working, service cartridge is constructed.
- you can complete, compile and package the pre-generated, customized sample source files into a working service cartridge and deploy it in a test system as an extension to the cisco base cartridge sample.
- you can take a copy of a provided skeleton properties file, relocate and rename it, and use it as the starting point to generate your own skeleton cartridge source files.

Included with each Cisco XQuery-based sample are:

- a **skeleton.properties** file

  This file is used to generate the source files for its sample service cartridge. See *Generating the cartridge source files* on page 14.

- **pre-generated, customized sample service cartridge source files** - these provided source files demonstrate the edits required to the generated source files to produce a working service cartridge.

As an example, the source files provided with the ciscoBanner sample include:

- `...\audit\auditTemplate.xml`
- `...\messages\successMessages.xml, errorMessages.xml, warningMessags.xml:` message patterns
- `...\schema\devicemodel.xsd`
- `...\test\TransformUnitTests.java`

- `...\transforms\sm2dm.xq, annotated-dm2cli.xq, dm-validation.xq`
- `...\xquerylib\ dm2cli-banner.xq, dm2cli-cisco.xq, dm-version.xq, sm2dm-banner.xq`

Note that the ***provided*** ciscoBanner sample source files are located in:

```
<SDKHome>\samples\serviceCartridge\ciscoBanner\...
```

while the ***pre-generated*** ciscoBanner sample cartridge source files are placed in:

```
<SDKHome>\serviceCartridges\ciscoBanner\...
```

Note: The other XQuery-based service cartridge sample files are organized in a similar fashion.

Sample XQuery-based service cartridge — ciscoBanner

Note: The details following details on the ciscoBanner service cartridge sample apply in a similar fashion to the other XQuery-based samples — ciscoMartini, and ciscoStaticRoute.

The ciscoBanner (and ciscoStaticRoute) sample cartridges illustrates the following concepts:

- filling out the skeleton properties for a service cartridge
- subscribing to configuration policies
- managing basic capabilities for configuration policies
- architecting the device model schema
- creating the cartridge Extension registry
- transformations (SM2DM and annotatedDM-CLI) that appropriately pass on the various IDs
- Device Model instance validation
- providing data so the network processor can recognize and correctly act on router responses
- providing command information to correctly enable NP audits for these services
- use of XQuery modules to improve compartmentalization, maintainability and readability of XQuery code

The ciscoBanner sample service cartridge implements the service modelled by the bannerSample sample configuration policy. The ciscoBanner sample contains a schema for a consolidated banner configuration policy that allows you to configure a sequence of one or more of five possible banner commands. It is assumed that only one occurrence of each banner type can exist on a device.

For details on the bannerSample configuration policy and its parameters and options, refer to the *Configuration Policy Extension Developer Guide*.

Process

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

You will need to create the source files for the corresponding configuration policies which implement the sample cisco services. See the *Configuration Policy Extension Developer Guide* for details.

ciscoBanner sample properties file

The sample skeleton properties file that is used to create the source files for the **ciscoBanner** sample service cartridge is called:

```
<SDKHome>\samples\serviceCartridge\ciscoBanner\skeleton.properties
```

This properties file is pre-populated with the information needed to construct the source files for the **ciscoBanner** sample service cartridge.

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

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

Refer to Appendix A for details on all the properties implemented in the sample properties files which create the source files for the samples.

Sample XQuery-based service cartridge — ciscoMartini

The ciscoMartini sample service cartridge implements a Martini layer two VPN service.

About the ciscoMartini sample

This service cartridge is written using XQuery-based transforms and illustrates how to implement a modeled Service Activator service using a service cartridge.

The provided set of pre-generated source files in the sample are based on the output of the supplied skeleton.properties file, but are pre-configured with some of the specific service configuration knowledge required for the service.

Key concepts illustrated are:

- filling out the skeleton.properties for a modeled service cartridge
- subscribing to a modeled service

- managing capabilities for a modeled service
- transformations (SM2DM and annotatedDM-CLI) that appropriately pass on various IDs

Note: this service cartridge does **not** implement a full-functional, or completed Martini service. It is intended only to demonstrate some of the concepts required when implementing such a service in a service cartridge.

Sample XQuery-based service cartridge — ciscoStaticRoute

The ciscoStaticRoute implements the service modelled by the sample static route configuration policy. It allows you to configure one or more static routes.

The ciscoStaticRoute sample cartridges illustrates the same concepts listed in *Sample XQuery-based service cartridge — ciscoBanner* plus the following concepts:

- specification of options schema
- use of options to influence transformations
- use of type-1 pre-check (to ensure a new static route service does not have a pre-existing conflicting static route on the router)
- use of type-2A pre-check (to ensure the next-hop address in the static route is reachable from the device - traps case when the network of next-hop is not reachable)
- use of type-2B pre-check (to ensure the next-hop address in the static route is reachable from the device - traps case when subnet is not reachable)
- use of post-check (to ensure that a provisioned static route has correctly updated the routing table)

For details on the ciscoStaticRoute configuration policy and its parameters and options, refer to the *Configuration Policy Extension Developer Guide*.

Sample Java-based service cartridge — alcatelSamStaticRoute

The Alcatel sample alcatelSamStaticRoute is a service cartridge to implement a static route service. The provided sample is pre-compiled.

Similar key concepts to those illustrated by the ciscoBannerJava sample are illustrated by this sample, with the inclusion of Alcatel SAM specific implementation details. (See *Building Alcatel SAM service cartridges* on page 47 for more details.)

Included with the Alcatel sample are:

- **sample service cartridge source files** - these provided source files demonstrate content required to produce a working service cartridge. The sample alcatelSamStaticRoute source files include:

- `...\audit\auditTemplate.xml`
- `...\javalib\*.java`: java source code demonstrating various actions
- `...\messages\successMessages.xml, errorMessages.xml, warningMessags.xml`: message patterns
- `...\schema\devicemodel.xsd`
- `...\transforms\`: key map xml file, Sm2dm, DmValidation and AnnotatedDm2Cli java source files

Sample Java-based service cartridge — ciscoBannerJava

The ciscoBannerJava sample service cartridge re-implements the bannerSample configuration policy for Cisco IOS and is similar to the ciscoBanner sample, except that it uses Java for model transformations instead of XQuery.

The provided sample is pre-compiled. Service cartridges with Java-based transforms do not make use of the generated XQuery source files, so no customization step for these files is needed.

The ciscoBannerJava sample implements the service modelled by the bannerSample configuration policy. The ciscoBannerJava sample contains a schema for a consolidated banner configuration policy that allows you to configure a sequence of one or more of five possible banner commands. It is assumed that only one occurrence of each banner type can exist on a device.

Key concepts illustrated are:

- file and directory structure for Java-based cartridges (since skeleton generation cannot be used)
- ant build scripting
- synonyms
- extension registry
- registry customization in the field
- use of options in Java transforms
- Java-based SM-DM and annotatedDM-CLI transforms
- Device Model upgrade transformation
- unit test harnesses for Java cartridges
- Java-based Device Model instance validation

For details on the bannerSample configuration policy and its parameters and options, refer to the *Configuration Policy Extension Developer Guide*.

Creating a service cartridge source directory and properties file

To create your own XQuery-based service 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. (For Java-based service cartridges, use one of the provides samples as a starting point and add your own Java code from there. None of the steps around creating and customizing XQuery files are required.)

Note: When deciding upon a directory structure for a new base cartridge 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.

Process

The simplest method is to copy one of the sample skeleton.properties files and then edit it for your own use.

1. Create a meaningful name for your new service cartridge. This name will be referred to as *this_service_name*.
2. Create a new directory for your cartridge source files:

   `<SDKHome>\serviceCartridges\`*this_service_name*
3. Copy a skeleton.properties file from one of the sample service cartridges into it:

   ```
   copy
   <SDKHome>\samples\serviceCartridge\ciscoBanner\skeleton.properties
   <SDKHome>\serviceCartridges\this_service_name
   ```
4. Edit your skeleton.properties file to change the sample cartridge name to *target_service_name* in the following entries:

```
## service cartridge name
   sdk_global_cartridgeName=this_service_name
. . .
## packaging structure
   sdk_global_package=com.metasolv.serviceactivator.cartridges.this_service_name
```

Defining the service and customizing the skeleton properties file

This step involves determining the specific details of the service to be created and specifying the information needed to apply the desired configuration to the device.

The service may require application across multiple OSs or device types. In this case, you will need to create multiple service cartridges.

This can affect the definition of the service and/or require separate services to be defined to achieve the desired end goal.

If your service requires an HTML-based GUI input form, you will also need to create a corresponding configuration policy. Note that a single configuration policy to implement a generic type of service may be subscribed to by multiple service cartridges, each implementing the service on a specific vendor's device.

You must edit the properties file to match the requirements of your service cartridge. Assuming you have used one of the supplied sample skeleton.properties files as a starting point, you will have to edit or remove properties that are not applicable to your service cartridge, and otherwise supply appropriate values for properties for your particular needs. Refer to Appendix A for details on the properties in the skeleton properties file.

Generating the cartridge source files

The SDK provides a tool for generating the service cartridge source files from the skeleton properties file. Once the source files are generated, you will need to edit them to complete your service 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 ciscoBanner service cartridge source files

The file `<SDKHome>\samples\serviceCartridge\ciscoBanner\skeleton.properties` is fully populated and can be used to construct the skeleton source files for the sample ciscoBanner service cartridge. You can use these files for reference, or as a starting point for your own service cartridge. The generated source files do not contain all the necessary modifications to complete the service. See the provided sample source files for the changes needed.

For the ciscoMartini service cartridge, you also have to modify the `Extension.xml` to point to the `default_caps.xml` file instead of the `empty_caps.xml` file.

Set the cartridge version string variable. If the cartridge version is 1.0, on a Windows host, type the command:

```
set VERSION_STRING=1.0
```

To generate the ciscoBanner sample service cartridge source files using the data from the skeleton properties file, go to the SDK directory:

```
cd <SDKHome>
```

and then either run the included batch file to run the cartridge generator script:

```
gensc samples\serviceCartridge\ciscoBanner\skeleton.properties
```

or type in the command to run the cartridge generator script:

```
ant -DtemplateType=serviceCartridge -
DpropFile=<SDKHome>\samples\serviceCartridge\ciscoBanner\skeleton.p
roperties
```

Note that to use the batch file, you must first add *<SDKHome>\bin* to your PATH variable where *<SDKHome>* is the SDK directory.

Result of generation process

The directory structure you created previously (see *Creating a service cartridge source directory and properties file* on page 12) has been extended by using the *<sdk_global_cartridgeName>* value from the skeleton properties file. The cartridge source files generated under *<SDKHome>*\serviceCartridges*<sdk_global_cartridgeName>*\ include:

- **build.xml** - ant build file to build the service 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 .xsd file for cartridge options
- **src\...\schema\devicemodel.xsd** - contains the stub service cartridge device model schema
- **src\...\test** - resources for testing the service cartridge
- **src\...\transforms** - transforms including pre- and post-check, SM to DM, annotated DM to CLI, and DM validation
- **src\...\xquerylib** - additional xqueries for DM to CLI, migration, validation, and version checking
- **src\...\cisco\Extension.xml** - identifies the service cartridge instance
- **src\...\cisco\Customization.xml** - can be used to override Extension.xml

A log file is also created within the logs directory:

<SDKHome>\logs\generator.log - log file from skeleton generation process.

To continue working with the sample ciscoBanner service cartridge, go to *Completing the sample service cartridge source files* on page 20.

Generating your service cartridge source files

When you create your own service cartridge, the cartridge name and the root folder for the generated source are based on the *<sdk_global_cartridgeName>* property value in the service cartridge skeleton properties file. It is incorporated into the cartridge source files in place of *<sdk_global_cartridgeName>* as shown below.

Set the cartridge version string variable. If the cartridge version is 1.0, on a Windows host, type the command:

```
set VERSION_STRING=1.0
```

To generate your skeleton service cartridge source files using your customized skeleton properties file, go to the SDK directory:

```
cd <SDKHome>
```

and then either run the included batch file to run the cartridge generator script:

```
gensc
\serviceCartridges\<sdk_global_cartridgeName>\skeleton.properties
```

or type in the command to run the cartridge generator script:

```
ant -DtemplateType=serviceCartridge -
DpropFile=<SDKHome>\serviceCartridges\<sdk_global_cartridgeName>\sk
eleton.properties
```

Note that to use the batch file, you must first add *<SDKHome>\bin* to your PATH variable where *<SDKHome>* is the directory where the SDK was installed.

Result of generation process

This extends the SDK directory structure in a similar manner to what was described in *Generating the sample ciscoBanner service cartridge source files* on page 14.

Note: It's 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 service cartridge generation

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

Access to jar files

The service cartridge needs to have access to the XMLbeans jar files. It must get these from the `ipsaSDK/3rdparty/lib` as shown in the setting of `cartridge.build.path` in `ipsaSDK/build/cartridge_build_imports.xml`. The use of this is shown in the ciscoBannerJava sample service cartridge's `build.xml` file.

In addition, if the service cartridge implements an SDK generated configuration policy, then the service cartridge will need access to the configuration policy's jar file as well. As an example, see how the `build.xml` file for ciscoBannerJava references the bannerSample configuration policy jar.

Using an alternate directory structure

If you are not using the standard directory structure to layout 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.

Service cartridge generator message logging

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

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

For details on troubleshooting property file attributes, see the *Base Cartridge Developer Guide*.

Completing your custom service cartridge source files

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

This section outlines the key source files you will need to edit to implement your service.

Device model (DM) schema definition

The service cartridge device model (`deviceModel.xsd`) extends the network processor's DM schema. It adds validation rules so that the new service(s) which the cartridge enables can be fully described between it and the network processor's Device Model.

Service model (SM) to device model transform

This XQuery or java source 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 Service Activator, flow through from the service model to the device model.

DM validation

If the cartridge 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 that would be caught by the device model validation using deviceMode.xsd.

Annotated DM 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 data from the annotated device model is transformed into a CLI document which contains the required device specific commands.

Service cartridge registry — Extension.xml

The service cartridge registry identifies the service model definitions to which the service cartridge subscribes and defines the transforms, audits, capabilities, options and messages of the service cartridge.

When the service cartridge is deployed, the service cartridge jar file will be deployed into the serviceCartridges sub-directory, the parent of which is the directory in which a base or core cartridge is already installed. Aspects of the service cartridge registry can be overridden after deployment by using a customization registry file.

The layout of the service cartridge registry file, `Extension.xml`, is defined by the schema `cartridge.xsd`. The service cartridge registry file is the first service cartridge file read by the network processor. It defines all of the important entry points into the service cartridge.

When you create a service cartridge, you must customize `Extension.xml` appropriately for your implementation.

The parameters specified in the service cartridge registry include:

- name - the name of the service cartridge. This must be unique across all service cartridges.
- smToDm - the service model to device model transform file
- dmValidation - the device model validation file
- dmToCli- the device model to CLI transform file
- dmMigration - the device model migration file. This is used to perform device model upgrades.
- success - the success message patterns file. These patterns are used to identify success messages sent by a device.
- warning - the warning message patterns file. These patterns are used to identify warning messages sent by a device.
- error - the error message patterns file. These patterns are used to identify error messages sent by a device.
- auditTemplateFile - the audit template file. It is used by the audit process for devices provisioned using a CLI.
- auditQueryFile - the audit query file. This file is used by the audit process for devices provisioned using an XML interface. Different audit template files can be specified for different device types and OS versions.
- optionsEntry - the options file. This file specifies the options file that will be passed to both the SM2DM and DM2CLI XQuery transforms. Different option files can be specified for different device types and OS versions.

- capsEntry - the capability file. This file specifies the capabilities that are supported by the service cartridge. Different capability files can be specified for different device types and OS versions. Capabilities for different service cartridges are ORed together by the network processor during a policy server caps fetch.
- subscriptions - specify the parts of the service model that the service cartridge is interested in.

For details on registry operations, refer to the SDK Overview and Setup Guide.

For information on customizing the registry, refer to *Customizing the registry - Registry.xml* in the *Base Cartridge Developer Guide*.

Message pattern definitions

Success, warning and error message pattern files can be defined for service cartridges in the same way as for base cartridges. (For example, device responses for commands sent by a service cartridge are analyzed and patterns are created in the message pattern files for that service cartridge. The difference is that for a base cartridge, the messages files are referenced from the Registry.xml file, and for a service cartridge, the messages files are reference from the Extension.xml file.) For an overview of the concepts behind message files, refer to the SDK Overview and Setup Guide.

For complete details on defining message patterns, refer to the *Base Cartridge Developer Guide*.

Completing the sample service cartridge source files

The generated sample cartridge source files are located in:

***<SDKHome>*\serviceCartridge\ciscoBanner**

To complete the sample, you must copy the files provided in ***<SDKHome>*\samples\serviceCartridge\ciscoBanner** over their counterparts in the generated source directory or you can edit the generated sample source files to complete their content development. The provided files demonstrate the modifications required to complete the generated sample source to produce a working sample service 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:

- `<SDKHome>\samples\serviceCartridge\ciscoBanner\src\...\audit\auditTemplate.xml`

- <SDKHome>\samples\serviceCartridge\ciscoBanner\src\...\messages*
- <SDKHome>\samples\serviceCartridge\ciscoBanner\src\...\schema\devicemodel.xsd
- <SDKHome>\samples\serviceCartridge\ciscoBanner\src\...\transforms*

Using options in the ciscoStaticRoute sample

Given a cartridge which supports a particular service on Cisco devices, here is an example of how to add an option to support a variation in a configuration command for certain device type(s) and OS version(s).

options.xsd file

In this case, the example adds an option for the **ciscostaticRoute** sample service cartridge supplied with the SDK to add the parameter `permanent` to the static route command for certain devices which require this.

Edit the options schema file `options.xsd` (based on the ***generated*** sample cartridge source file) and add the following statement to define support for a new boolean option type for this cartridge called `cartridge.ciscoStaticRoute.permanentOption`:

```
. . .
<xs:element name="cartridge.ciscoStaticRoute.permanentOption"
type="opt:BooleanValue" minOccurs="0" default="false"/>
. . .
```

Note that the default value is false.

options.xml file

Create the `options.xml` file. It specifies which options apply for the device types and device OS variants that this cartridge supports.

Create an entry using the data type defined in the `options.xsd` file - `cartridge.ciscoStaticRoute.permanentOption`.

The options.xml file follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<base:options
xsi:type="CartridgeOptions"
xmlns="http://www.metasolv.com/serviceactivator/cisco/staticroute/
options"
xmlns:base="http://www.metasolv.com/serviceactivator/options"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<!-- options for ciscoStaticRoute service cartridge -->
```

```
  <cartridge.ciscoStaticRoute.permanentOption>true
</cartridge.ciscoStaticRoute.permanentOption>
<!-- additional options could go here -->
</base:options>
```

xquerylib\dm2cli-staticroute.xq file

Now that the option has been defined, edit the annotated DM to CLI transform to implement it.

The first statment imports a standard library file of common XQuery functions (`options-common.xq`) to provide the ability to recognize and process 'if' clauses.

The if statement causes triggers the option evaluation and action.

```
  import module namespace options = "options-common-functions" at
"resource://metasolvcom/metasolv/serviceactivator/networkprocessor/
xquerylib/options-common.xq";
. . .
  if
(options:getBooleanOption("cartridge.ciscoStaticRoute.permanentOption"
,false()) = true()) then
    " permanent"
  else ())
. . .
```

When the annotated DM to CLI transform is run, an XQuery command to read the boolean option 'cartridge.ciscoStaticRoute.permanentOption' from the loaded `Extension.xml` file is run.

A check is made to see if there is an options file specified that matches the device type and OS variant for the target device the static route is being configured on.

In this case, there is an entry in the Extension.xml file to match devices of type 'Cisco.*' and OS '*.'. The file `options.xml` is specified for devices matching these characteristics. Because of the wildcards, the target device for the static route matches.

Next, the specified options file (**`options.xml`**) is searched for an entry matching the passed option (**cartridge.ciscoStaticRoute.permanentOption**). The entry exists, and the value associated with it is **true**.

This true value is returned back up to the if statement in the transform XQuery which initiated the option lookup:

```
  if
(options:getBooleanOption("cartridge.ciscoStaticRoute.permanentOption"
,false()) = true()) then
```

```
" permanent"
```

and the string " permanent" is added to the output CLI. The static route command for this target device, which is handled by this cartridge, will have " permanent" appended to it.

Extension.xml file

Edit the `Extension.xml` file for the cartridge to specify that the cartridge uses options, and to give the name of the options file (`options.xml`).

In addition, supply regular expressions to specify the device types and OS versions that this options file applies to. In this case, it applies to all Cisco device types, and all OS versions.

```
<options>
  <optionsEntry>
  <optionsFile>com/metasolv/serviceactivator/cartridges/
ciscostaticroute/options/options.xml</optionsFile>
  <appliesTo>
    <deviceTypes useRegex="true">Cisco.*</deviceTypes>
    <osVersions useRegex="true">.*</osVersions>
    </appliesTo>
  </optionsEntry>
</options>
```

Building the service cartridge

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=
```

Service 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.

If not already set, set the cartridge version string variable. For example:

```
set VERSION_STRING=1.0
```

Compiling the ciscoBanner sample source files

Compile the ciscoBanner sample service cartridge source files with the command:

```
ant package -f<SDKHome>\serviceCartridges\ciscoBanner\build.xml
```

Compiling your custom service cartridge source files

Once you have customized your service cartridge source files, compile the service cartridge with the command:

```
ant package -
f<SDKHome>\serviceCartridges\<sdk_global_cartridgeName>\build.xml
```

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

```
<SDKHome>\serviceCartridges\<sdk_global_cartridgeName>\
build.xml
AuditTrailsReports
beansrc
classes
lib
    <sdk_global_cartridgeName>.jar
    <sdk_global_cartridgeName>tests.jar
package
    <sdk_global_cartridgeName>-serviceCartridge-
${env.VERSION_STRING}.zip
       <sdk_global_cartridgeName>-serviceCartridge-
${env.VERSION_STRING}.manifest
```

Troubleshooting service cartridge building

Compilation issues

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

Manifest file

When a service cartridge is built, a manifest file is created listing all of the files that are packaged into the service cartridge zip file. Installation of the service cartridge places the manifest in the uninstall directory of the 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 pre-requisites 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 *Pre-and post-checks* in the SDK Overview and Setup Guide.

Testing in a standalone environment

Test scripts are created as part of the cartridge skeleton generation process. (See *Generating the cartridge source files* on page 14).

Unit tests

The unit test is generated with the skeleton service cartridge source files. After you have compiled the service cartridge, run unit tests with the command:

```
ant unittests
```

against

```
<SDKHome>\serviceCartridges\<sdk_global_cartridgeName>\build.xml
```

For example:

```
ant unittests -f=<SDKHome>\serviceCartridges\
<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.

Deploying service cartridges

Service cartridges are deployed as extensions to either base or core cartridges.

Core and base cartridges are deployed as jar files, along with their `MIPSA_registry.xml / Registry.xml` files, to the following subdirectory:

<ServiceActivatorHome>`\lib\java-lib\cartridges\`*<vendor>*

Service cartridges are placed below this:

<ServiceActivatorHome>`\lib\java-lib\cartridges\`*<vendor>*`\ServiceCartridges`

The `Extension.xml` file of each service cartridge must be placed at the root level in the service cartridge .jar file.

When an ant build is successfully completed from the service cartridge skeleton directory a service cartridge .zip file is produced in the skeleton package directory. The service cartrdige can then be installed into Service Activator by unzipping it to the *<ServiceActivatorHome>* directory. Upon restart of the network processor, the service cartridge is loaded. A notification appears in the Service Activator GUI fault pane to confirm that the service cartridge has been loaded.

To deploy the service cartridge in a Service Activator environment:

- unzip the cartridge file `<sdk_global_cartridgeName>-serviceCartridge-${env.VERSION_STRING}.zip` to the runtime environment of the network processor <NetworkProcessorHome>.
- Restart the network processor to load `<sdk_global_cartridgeName>.jar`

To observe cartridge loading operation see:

- **<*ServiceActivatorHome*>\logs\networkProcessor.log**
- **<ServiceActivatorHome>\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 service 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.

Deploying the sample service cartridge

Once you have compiled the ciscoBanner sample service cartridge, it can be deployed along with the sample configuration policy - bannerSample. (See the

Configuration Policy Extension Developer Guide for details on creating the bannerSample sample configuration policy.)

Deploy the service cartridge as described above.

Create and deploy the bannerSample configuration policy following the procedure in the *Configuration Policy Extension Developer Guide*.

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.

Each network processor maintains one current networkprocessor.log file and one current audit trail log file per base or core cartridge. Audit trail logging for a service cartridge is written to the audit trail log file for the base or core cartridge that the service cartridge extends. The network processor logging facilities are based on the log4j utility.

For more information on network processor logging, please refer to *Chapter 7* of the Service Activator *Administrator's Guide*.

For details on setting audit trail logging properties, refer to *Audit trail logging* in the *Base Cartridge Developer Guide*.

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.

For further details on device model upgrades, refer to the *Base Cartridge Developer Guide*.

Audit

Audit functionality is controlled by an audit template and an audit query file. The names of these files are specified in the `Extension.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 the SDK Overview and Setup Guide.

Uninstalling service cartridges

Service cartridges are uninstalled using the **uninstallCartridge.sh** script, which resides in the **bin** directory of the Service Activator installation. This script takes the name of the manifest file, which contains a list of all installed service cartridge files, as a parameter., and uses its contents to uninstall the service cartridge. (See *Manifest file* on page 25.)

You can include the base directory or the 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 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.

Note: you can use a relative path to specify the manifest file, but it must be relative to the current directory (i.e. 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.

The command is:

```
uninstallCartridge <manifest_file> [<ServiceActivatorHome>] [-k | -
v]
```

Use the **-k** option to leave empty directories. The **-v** (verbose) option produces extra output from the script.

Note: You must restart the network processor after the service cartridge is uninstalled.

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 service cartridge from the SDK

To remove a generated service cartridge from the SDK installation, delete all contents under and including:

<SDKHome>\serviceCartridges\<sdk_global_cartridgeName>

Uninstalling the SDK

To uninstall the SDK, delete all contents under *<SDKHome>*.

Appendix A

Service Cartridge generation properties

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

This file contains a number of properties that customize the generated service 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 (i.e. 'true', or 'false').

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

Note: Be aware that 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.

Audit

Property	Description	Example
sdk_audit_supported	Command sent to the device to determine if device is supported. Optional	true

Naming and packaging

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. Mandatory.	ciscoBanner
sdk_global_baseCartridgeName	The base or core cartridge that this service cartridge extends to provide support for a specific service for a specific vendor. Mandatory.	cisco
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.	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 source code path generation. The value is used in build scripts, java source code and support files. The generated files are placed under: `<SDKHome>\serviceCartridges\<sdk_global_cartridgeName>\src\<sdk_global_package>` Mandatory.	com.metasolv.serviceactivator.ciscoBanner would become com\metasolv\serviceactivator\ciscoBanner

Device type identification

These properties are used in the JUnit test environment.

Property	Description	Example
sdk_global_deviceName	Specify the device name that the base cartridge, that this service cartridge extends, was constructed for. This is used in the sample service model used by the junit tests for this service cartridge. Mandatory.	Cisco
sdk_global_deviceDescription	Device description. This is used in the sample service model used by the junit tests for this service cartridge.	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. This is used in the sample service model used by the junit tests for this service cartridge.	2611
sdk_global_deviceVersion	Device version. This is used in the sample service model by the junit tests for this service cartridge.	12.2(11)T8

Device Model schema

Property	Description	Example
sdk_deviceModel_namespace	Target namespace of the device model schema for this service cartridge. Mandatory.	http://www.metasolv.com/serviceactivator/devicemodel/ciscobanner
sdk_deviceModel_namespaceAbbr	Abbreviation of the target namespace of the device model schema for this service cartridge. This is used as a namespace prefix. Mandatory.	dmbanner
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 service cartridge. Mandatory.	CiscoBanner would become "CiscoBannerDevice"

Subscription

Subscriptions - properties which manage subscription to a configuration policy or modeled service definition.

Note that although a service cartridge can subscribe to more than one modeled service definition type and/or more than one configuration policy, the skeleton generator allows you to specify only one subscription. More subscriptions can be added by editing the generated file `Extension.xml` after the file generation step.

Note: Only one of sdk_subscription_configPolicy_supported and sdk_subscription_serviceDefinition_supported can be set to **true**.

Property	Description	Example
sdk_subscription_configPolicy_supported	Boolean value to indicate whether or not this service cartridge implements a configuration policy. When set to true, the value of sdk_subscription_configPolicyName identifies the configuration policy. Mandatory.	true
sdk_subscription_configPolicyName	Configuration policy subscription. Specify either a core configuration policy content type, or, for configuration policies created using the SDK, specify the configuration policy name that is registered in `ConfigPolicyRegistry.xml`.	bannerSample

Property	Description	Example
sdk_subscription_serviceDefinition_supported	Boolean value to indicate whether or not this service cartridge implements a modeled service definition. When set to true, the value of sdk_subscription_serviceDefinitionName idenitifies the modeled service definition. Mandatory.	false
sdk_subscription_serviceDefinitionName	Modeled service definition subscription. Specify a core definition type from: ■ ParameterSetDefinitionType ■ MqcDefinitionType ■ AccessRuleDefinitionType ■ GenericRuleDefinitionType ■ ServiceRuleDefinitionType ■ PolicingRuleDefinitionType ■ PhbGroupType ■ IBgpBaseDefinitionType ■ IBgpNeighbourDefinitionType ■ VrfTableDefinitionType ■ MartiniDefinitionType ■ CccDefinitionType ■ L2InterfaceCreationDefinitionType ■ TlsDefinitionType ■ SAADefinitionType	ParameterSetDefinitionType

Valid core configuration policy content types

Valid core configuration policy content types for the sdk_subscription_configPolicyName property include the following:

- atmSubInterfaceData
- frSubInterfaceData
- plSerialInterfaceData
- plPosInterfaceData
- dslInterfaceData
- loopbackInterfaceData
- ciscoUniversalInterface
- multilinkInterface
- pppMultilink
- virtualTemplateInterface
- dialerInterface
- ipPools
- prefixListEntries
- staticRoutes
- banners
- netflowParameters
- collectorParameters
- staticNats
- snmpCommunities
- snmpHosts
- saveConfig
- qosCosAttachment
- juniperQosCosAttachment
- ciscoQosPfcTxPortQueues
- vlanAccessPort
- vlanTrunkPort
- vlanDefinitions
- vlanInterface
- portCharacteristics
- customerIPsec
- publicIPsec
- rate-limit
- hsrp
- vlanSubInterfaceData
- keyChains
- multicastInterface
- multicastVrf
- multicastDevice
- multicastBootstrapRouter
- multicastAutoRp
- vrfRoutePolicy
- bgpRoutePolicy
- userData
- userAuth
- dialerList
- ipsecmodule
- stm1Controller
- t3Controller
- e1Controller
- e3Controller
- t1Controller
- stm1ChannelizedSerialInterface
- t3ChannelizedSerialInterface
- e1ChannelizedSerialInterface
- e3ChannelizedSerialInterface
- t1ChannelizedSerialInterface
- basicRateInterfaceData
- sgbp
- schedule
- atmPvcVcClass
- atmVcClass
- bgpCE
- extendedAcl
- backUpInterfacePolicy
- vrfCustomNaming
- vrfExportRouteFilter
- dlswDevice
- dlswEthernetInterface
- dlswTokenRingInterface
- alcatelSRL3Interface

Options schema

Property	Description	Example
sdk_options_namespace	Target namespace of the options schema for this service cartridge. Mandatory.	http:// www.metasolv.com/ serviceactivator/options
sdk_options_namespaceAbbr	Abbreviation of the target namespace of the options schema for this service cartridge. This is used as a namespace prefix. Mandatory.	cisopt

Appendix B

Generated Skeleton Service Cartridge Source Files

This appendix describes generated service cartridge source files.

About the generated skeleton service cartridge source files

The file `<SDKHome>\samples\serviceCartridge\ciscoBanner\skeleton.properties` is fully populated and can be used to construct the skeleton source files for the sample ciscoBanner service cartridge. You can use these files for reference, or as a starting point for your own service cartridge. The generated source files do not contain all the necessary modifications to complete the service. See the provided sample source files for the changes needed.

To generate the ciscoBanner sample service cartridge source files using the data from the skeleton properties file, go to the SDK directory:

```
cd <SDKHome>
```

and then either run the included batch file to run the cartridge generator script:

```
gensc samples\serviceCartridge\ciscoBanner\skeleton.properties
```

or type in the command to run the cartridge generator script:

```
ant -DtemplateType=serviceCartridge -
DpropFile=<SDKHome>\samples\serviceCartridge\ciscoBanner\skeleton.p
roperties
```

Note that to use the batch file, you must first add *<SDKHome>\bin* to your PATH variable where *<SDKHome>* is the SDK directory.

This results in the following directory structure, which is a skeleton ciscoBanner service cartridge:

```
<SDKHome>
```

```
logs
   generator.log
serviceCartridges
   <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>(....ciscobanner)
                        audit
                           auditTemplate.xml
                        capabilities
                           empty_caps.xml
                        messages
                           errorMessages.xml
                           successMessages.xml
                           warningMessages.xml
                        options
                           options.xsd
                        schema
                           devicemodel.xsd
                        test
                           models
                              upgradeFrom
                                 sampleDeviceModel.xml
                              sampleServiceModel.xml
                           DmUpgradeTests.java
                           TransformUnitTests.java
                        transforms
                           annotated-dm2cli.xq
                           dm2cli-postcheck.xq
                           dm2cli-precheck.xq
```

```
    dm-validation.xq
    sm2dm.xq
xquerylib
    dm-migration.xq
    DmUpgrade.xq
    dm-validation.xq
    dm-version.xq
Customization.xml
Extension.xml
```

Generated skeleton service cartridge source files details.

The following table describes the functionalities of the component files of the skeleton service cartridge.

Component file	Description
synonyms.xml	This file can be used to specify audit synonyms for commands delivered by this service cartridge. Audit synonyms can improve the success rate of a device audit, for devices that display some commands differently than how Service Activator sent them.
auditTemplate.xml	This file is an audit template. Audit templates define filter patterns to be applied to commands to identify configuration of interest, to affect their inclusion in the audit report, and to set attributes on the command results, which, when viewed using a stylesheet will affect how they are displayed to the Service Activator user.

Component file	Description
Extension.xml	The service cartridge registry is defined by the `schema cartridge.xsd`. The service cartridge registry file is the first service cartridge file read by the network processor. It defines all of the important entry points for the service cartridge. The name of the service cartridge registry file is `Extension.xml`. The parameters controlled by the service cartridge registry are: ■ **name**—defines the name of the service cartridge. This must be unique across all service cartridges. ■ **smToDmQuery**—defines the service model to device model transform XQuery ■ **dmValidation**—defines the device model validation XQuery ■ **dmToCliQuery**—defines the device model to CLI transform XQuery ■ **dmMigration**—defines the device model migration XQuery. This is used for device model upgrades. ■ **success**—defines success messages sent by a device ■ **warning**—defines warning messages sent by a device ■ **error**—defines error messages sent by a device ■ **auditTemplateFile**—used by the audit process for devices provisioned using a CLI. Different audit template files can be specified for different device types and OS versions. ■ **subscriptions**—specifies the parts of the service model that the service cartridge is interested in.

Component file	Description
Extension.xml (continued)	**auditQueryFile**—used by the audit process for devices provisioned using an XML interface. Different audit template files can be specified for different device types and OS versions. **optionsEntry**—specifies the options file that is passed to both the SM2DM and DM2CLI XQuery transforms. Different option files can be specified for different device types and OS versions. **capsEntry**—specifies the capabilities that are supported by the service cartridge. Different capability files can be specified for different device types and OS versions.
Customization.xml	This is used to customize the registry file. When packaged in the zip file, it will be named <sdk_global_cartridgeName>.xml.
empty_caps.xml	Capabilities provide privileges to the device and its subordinate interfaces to support various policies. The `empty_caps.xml` is a sample capabilities file. 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 Service Activator device discovery process.
errorMessages.xml	This file contains error patterns for commands generated by the service 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.

Component file	Description
warningMessages.xml	This file contains warning patterns (blocking or non-blocking) for commands generated by the service 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 service 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.xml	This is an XML schema file containing the configuration options that are required to define the cartridge jar file.
deviceModel.xsd	This file is used to validate the device model created during the service model to device model transformation. The `base_devicemodel` is owned by the network processor framework. The deviceModel is owned by the cartridge and can be extended as needed to support various polices and commands.
sampleDeviceModel.xml	This sample device model is used by DMUpgradeTests.java.
sampleServiceModel.xml	This sample service model is used for junit testing by `TransformUnitTesting.java`.

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 is the XQuery source code that transforms a service model to a device model.
annotated-dm2cli.xq	This is the XQuery source code that transforms a device model to a CLI document.
dm2cli-postcheck.xq	This is the XQuery source code that performs the post-check functionality which is used by the `annotatedDM2Cli.xq`.
dm2cli-precheck.xq	This is the XQuery source code that performs pre-check functionality, which is used by the `annotatedDM2Cli.xq`.
dm-validation.xq	This is the XQuery source code providing the ability to raise fault to the system console.
dm-migration.xq	This is the XQuery source code used to support device model upgrades.
DmUpgrade.xq	This is the XQuery source code used to support executing a DM upgrade if cartridge DM has been enhanced.

Component file	Description
dm-version.xq	This is the XQuery source code used to identify which cartridge version is in use.
dm-validation.xq	This file has the XQuery source code used for additional validation of the device model.

Appendix C

Building Alcatel SAM service cartridges

This chapter discusses Alcatel service cartridge concepts and explains how to use the SDK to create service cartridges for Alcatel devices.

This guide assumes:

- that Service Activator is deployed to a directory which will be referred to as `<ServiceActivatorHome>`. 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 `<SDKHome>`.
- that the required versions of additional 3rd 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 3rd party tool versions, see the SDK Overview and Setup Guide.

Overview of Alcatel service cartridges

The information in this guide builds on the concepts, details, and procedures contained in the *Service Cartridge Developer Guide*.

In general, the concepts and procedures described in the *Service Cartridge Developer Guide* apply. However, Alcatel service cartridges have some differences from typical service cartridges. This guide explains the additional things you need to know to develop Alcatel service cartridges.

As you refer to the procedures in the *Service Cartridge Developer Guide* keep the contents of this guide in mind, adapting as necessary to take into account the particular attributes of Alcatel service cartridges.

The following guidelines apply to Alcatel service cartridges:

- Alcatel service cartridges work in conjunction with the Alcatel 5620 SAM Cartridge. This base cartridge provides interface capabilities between the network processor and the Alcatel-Lucent 5620 Service Aware Manager (SAM). Refer to the *Alcatel 5620 SAM Cartridge Guide* for details.
- transforms created for Alcatel service cartridge should be implemented using Java instead of XQuery. The XQuery transform files created by the cartridge generation process must be replaced with your own Java code. The parts of the procedures in Chapter 2 dealing with XQuery, generating and customizing cartridge source files in particular, do not directly apply to Java-based service cartridge development.
- the Alcatel base cartridge does not communicate directly with Alcatel devices, but communicates through SAM-O, an open interface on the SAM through which an OSS client application can perform tasks such as configuring network management information in the SAM database and modifying managed objects.

References are made to sample service cartridge components packaged with the SDK.

Note: Refer to Appendix A in the *Service Cartridge Developer Guide* for details on all properties in the properties file.

Alcatel SAM ID Mapper - AlcatelSamIdMapper

The Alcatel SAM maintains its own set of unique IDs for the objects it manages. These IDs need to be referenced when modifying or deleting objects. The AlctatelSAMIdMapper acts as a transient cache of object IDs and a proxy, allowing the Alcatel Cartridge to query the SAM based on service specific data.

The ID Mapper generates its own set of keys based on the attributes of objects that can uniquely identify them in the SAM. For example, a subscr.Subscriber (Customer) object in the SAM can be uniquely identified by its subscriberName attribute. However, in order to uniquely identify a vprn.Vprn object you will need to refer to both its customerName and globalServiceName attributes. Note that the customerName and the subscriberName attributes refer to the same data (Service Activator Customer Name) and as a result the ID Mapper tries to normalize these differences through the use of a key map.

The ID Mapper Key Map document describes the various SAM objects and how to generate a unique key based on their attributes. A Key Map document contains a number of Type Definitions which describe the various SAM objects including their type name, the attributes that uniquely define a type and a "normalizing" key to identify each of these attributes in a consistent manner. Your Key Map XML file should be placed in the same directory that your sm2dm and annotatedDm2Cli transforms are in.

The following is an example of a Type as it would be defined in your Key Map XML document:

```
<tns:typeDefinition>
  <tns:name>rtr.VirtualRouter</tns:name>
  <tns:keySet>
    <tns:keyMap order="0">
      <tns:type>rtr.VirtualRouter1</tns:type>
      <tns:key>siteId</tns:key>
    </tns:keyMap>
    <tns:keyMap order="1">
      <tns:type>rtr.VirtualRouter2</tns:type>
      <tns:key>routerId</tns:key>
    </tns:keyMap>
  </tns:keySet>
</tns:typeDefinition>
```

The <typeDefnintion> has the following elements.

- `<name>` — name of the SAM schema element to create a key map for. **This name can only occur once within all Alcatel core and service cartridges.**
- `<keyMap>` — These elements list the composite identifiers that uniquely identify the object in SAM. Each keyMap in the typeDefinition must have a unique `order`

number and must run in sequence starting from 0. The `<type>` element is used to uniquely identify your keyMap entry within the typeDefinition. The `<key>` element is the actual name of the element in the SAM API. One use of the Key Map Document is to help the Command Executor find an object in SAM.

Below is an excerpt from a network processor log file showing how the above typeDefinition is used to get the objectFullName of a rtr.VirtualRouter.

```
<find xmlns="xmlapi_1.0">
  <fullClassName>rtr.VirtualRouter</fullClassName>
  <filter>
    <and>
      <equal name="siteId" value="10.1.1.53"/>
      <equal name="routerId" value="1"/>
    </and>
  </filter>
  <resultFilter>
    <attribute>objectFullName</attribute>
    <children/>
  </resultFilter>
</find>
```

Service Activator Service cartridges can define their own Key Map documents and further populate the existing AlcatelSAMIdMapper with service specific definitions.

The following example is an excerpt of java code that you would use in your sm2dm or annotatedDm2Cli java file showing how a service cartridge lists a Key Map file:

```
AlcatelSAMIdMapper mapper = AlcatelSAMIdMapper.getInstance();
    mapper.populateTypeMap("alcatelSamSampleKeyMap.xml",
AnnotatedDm2Cli.class);
```

The AlcatelSAMIdMapper operates as an MRU (most recently used) cache maintaining object mappings for a user-definable number of objects. If the cache grows beyond the defined size, the least referenced object will be discarded. The cache size can be defined in the `AlcatelSAMHosts.properties` file.

Another element that can be optionally defined in the Key Map Document is `CommandPreProcessors`. The value of the `<preCommandValidation>` is the name of a class that implements a Command Document pre-processor.

The following is an example from a Key Map XML file showing a CommandPreProcessor registration:

```
<tns:preCommandValidation>com.metasolv.serviceactivator.cartridges.alc
atelSam.commandExecutor.VprnCommandPreProcessor</
tns:preCommandValidation>
```

Implementation of a CommandPreProcessor is described in *CommandPreProcessor* on page 53.

Command Document Structure

Some specific guidelines must be followed when you create the Java transform (i.e. the annotatedDm2CLI) in your Alcatel service cartridge which creates the Command Document.

For generation of the Command Document, the Alcatel SAM cartridge uses <commandXML> instead of <commandString>. You should not use nested commands in the CommandSession Document.

The Alcatel SAM Cartridge Command Executor requires that a specific structure be used inside the commandXML. The command must contain the SAM configuration document wrapped in a samCommand document containing extra information required by the Command Executor to create the correct SOAP wrapper to send to SAM.

Below is an excerpt from a network processor log file showing a command that the annotatedDm2Cli would create:

```
<command dmId="15,16,17,18" xmlns:sam="http://www.oracle.com/
SAMCartridgeCommand">
  <assoc_id>4105</assoc_id>
  <assoc_id>14032</assoc_id>
  <commandXML>
    <sam:samCommand>
      <sam:command>
        <xa:generic.GenericObject.configureChildInstance>
            <xa:deployer>ignored</xa:deployer>
            <xa:synchronousDeploy>true</xa:synchronousDeploy>
            <xa:clearOnDeployFailure>true</xa:clearOnDeployFailure>
<xa:distinguishedName>subscr.Subscriber_#_acme_:_vprn.Vprn_#_vrfName_:
_vprn.Site_#_10.1.1.53</xa:distinguishedName>
          <xa:childConfigInfo>
            <xa:svt.Cloud>
              <xa:circuitTransport>ldp</xa:circuitTransport>
              <xa:actionMask>
                <xa:bit>create</xa:bit>
                <xa:bit>modify</xa:bit>
            </xa:actionMask>
          </xa:svt.Cloud>
        </xa:childConfigInfo>
      </xa:generic.GenericObject.configureChildInstance>
```

```
    </sam:command>
<sam:samType>subscr.Subscriber_#_acme_:_vprn.Vprn_#_vrfName_:_vprn.Sit
e_#_10.1.1.53_:_svt.Cloud</sam:samType>

<sam:samParentType>subscr.Subscriber_#_acme_:_vprn.Vprn_#_vrfName_:_vp
rn.Site_#_10.1.1.53</sam:samParentType>
    </sam:samCommand>
  </commandXML>
</command>
```

Within the commandXML, an element called <samCommand> will be created. This element has three child elements.

- `<command>` — This element contains the SAM configure document from the SAM API.
- `<samType>` — This element contains the ID Mapper value of the element being executed.
- `<samParentType>` — This element contains the ID Mapper value of the elements parent. This element is optional for SAM root elements.

When creating the `<xa:distinguishedName>` or `<xa:objectFullName>` elements in the SAM configure command, you should use the ID Mapper value instead of the real SAM values during transform. These values in the command will be replaced by the real SAM values during command execution of the command preprocessor.

CommandPreProcessor

As an extension, the ID Mapper can reference a command preprocess or prevalidation step. During the execution of the Command Document, each command can optionally also have preprocessing done on the command. This allow modification to the Command before it is executed by the Command Executor.

An example of such a use is the replacement of placeholder values in the command with values retrieved from SAM. During the SM2DM and AnnotatedDm2CLI transformations, there might be unpersisted data or IDs that you want to put into your command.

To minimize the communication between Service Activator and SAM during transformation, it is best to put SAM queries in the command preprocessor. Such a technique is already used for populating `<xa:distinguishedName>` and `<xa:objectFullName>`. These are unique keys that SAM uses to configure objects. These keys are generated by SAM when objects are created. In the cartridge device model, the cartridge stores a cartridge representation of the key that the pre-processor will replace during command execution. This prevents the transform from querying SAM for keys of every object, but instead will only query SAM for keys object objects that are in the command document.

To use the command preprocessor, you must create a new class that extends `BaseCommandPreProcessor` and registers the preprocessor in the ID mapper file. Two methods will needed to be implemented for the preprocessor class.

The first method is `testCommand()`. The Command Executor will run this method for each command in the Command Document to see if the command will be passed to your pre-processor. The example below show a simple but typical implementation of the method. The example first calls a method called `isMyCommand()`, which checks if the Command is for the SAM element of interest. The method also calls `isDelete()`, which checks if the SAM command is to be deleted.

Note: The `isMyCommand()` and `isDelete()` implementations are not shown and would be implemented as part of the custom preprocessor.

```
public synchronized boolean testCommand(Command command) throws
Exception {

       boolean myCommand = isMyCommand(command);

       return((myCommand) && (!isDelete(command)));

}
```

If no pre-processor class returns **true**, then the command will be passed to the GenericCommandPreProcessor which will replace the ID mapper values for the `<xa:distinguishedName>` and `<xa:objectFullName>` elements with real SAM IDs retrieved from SAM.

The second method is `processCommand()`. This method in the preprocessor will be called if the `testCommand()` method returned true. The Command from the Command Document will be passed in for processing. At this point the method can modify the Command or perform validation tasks against the SAM that you do not want to perform at transform time. There are two different ways that the `processCommand()` can return. The following are examples of the two options.

The first implementation is to perform some Command modification or validation and then tell the CommandExecutor to also run the GenericCommandPreProcessor when complete. If the Command includes `<xa:distinguishedName>` or `<xa:objectFullName>` that has ID Mapper values that need processing and this method did not do the substitution, then the method must return to the CommandExecutor to run the GenericCommandPreProcessor. For example:

```
public CommandProcessorResultCode processCommand(Command command)
throws Exception{
   // Perform my validation or modification here.
   return new
CommandProcessorResultCode(CommandProcessorResultCode.RUN_GENERIC);
}
```

The second implementation is to perform some Command modification or validation and then tell the CommandExecutor to *not* run the GenericCommandPreProcessor when complete. If the Command does not include `<xa:distinguishedName>` or `<xa:objectFullName>` that has ID Mapper values which need processing or this method is doing the substitution, then the method must return to the CommandExecutor such that the GenericCommandPreProcessor is not run. For example:

```
public CommandProcessorResultCode processCommand(Command command)
throws Exception{
   // Perform my validation or modification here.
   return new
CommandProcessorResultCode(CommandProcessorResultCode.NO_RUN_GENERIC);
}
```

Device Model Best Practices

The following are best practices to assist in efficient creation of a device model.

- The Alcatel SAM uses a hierarchical based schema that can be reflected in a device model. This same structure makes it very simple to create a device model. The first item is to make each SAM object a Changeable element in the device model. Since a Changeable element in the device model is supposed to represent a command, this holds true for SAM. The following example shows the declaration of a Changeable type for Epipe.

```
<xs:complexType name="ChangeableEpipe">
  <xs:complexContent>
    <xs:extension base="lib:Changeable">
      <xs:sequence>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
/xs:complexType>
```

- Create a container element for each SAM element. Between each layer in the device model, add a container which will assist in grouping and the addition of identifiable keys (described below). The following is an example of creating a container:

```
<xs:complexType name="Epipe.Container">
  <xs:complexContent>
    <xs:extension base="lib:Container">
      <xs:sequence>
      <xs:element name="epipe" type="dm:ChangeableEpipe"
minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:extension>
</xs:complexContent>
</xs:complexType>
```

- Each SAM Changeable will contain the three different types of children:
  - reference to the SAM schema for object data

    ```
    <xs:element ref="sam:epipe.Epipe" minOccurs="1" maxOccurs="1"/>
    ```

  - one or more containers for other SAM Changeable child elements

    ```
    <xs:element name="epipe.Site.Container"
    type="dm:Epipe.Site.Container" minOccurs="0" maxOccurs="1"/>
    ```

    - Optionally, an element that represents the identity. You should only need to add this element if the SAM reference object does not contain an element that is used as its unique identity in your device model.

```
      <xs:element name="name" type="xs:string" minOccurs="1"
      maxOccurs="1"/>
```

- Each Changeable should also be Identifiable. To have more efficient commands being sent to the SAM, it is best to give each SAM Changeable and Identifiable key. Without the Identifiable, the annotation of the device model would cause your element to be deleted and re-added in the SAM instead of being implemented with the more efficient **modify** command. This also means that if there is no identity, any change to any element in the SAM referenced object would cause a delete and re-add.

  If you have a SAM element for which you will never want to issue a modify command, then do not add the Identifiable key. The following is an example of setting the `epipe.Site` Identifiable key during the declaration of the `epipe.Site` parent Container.

```
<xs:element name="epipe.Site.Container"
type="dm:Epipe.Site.Container" minOccurs="0" maxOccurs="1">
  <xs:key name="epipeSiteKey">
    <xs:selector xpath="dm:epipe.Site"/>
    <xs:field xpath="sam:epipe.Site/sam:siteId"/>
  </xs:key>
</xs:element>
```

The following example implements all the best practices listed above. It shows how to use the repeatable pattern of Container/Changeable for an `Epipe` service. You would repeat the same pattern at each layer such as `epipe.Epipe->epipe.Site`, then `epipe.Site->vll.L2AccessInterface` and so on.

```
<xs:complexType name="Epipe.Container">
  <xs:complexContent>
    <xs:extension base="lib:Container">
      <xs:sequence>
        <xs:element name="epipe" type="dm:ChangeableEpipe" minOccurs="1"
maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="ChangeableEpipe">
  <xs:complexContent>
    <xs:extension base="lib:Changeable">
      <xs:sequence>
        <xs:element name="name" type="xs:string" minOccurs="1"
maxOccurs="1"/>
        <xs:element ref="sam:epipe.Epipe" minOccurs="1" maxOccurs="1"/>
```

```
      <xs:element name="epipe.Site.Container"
type="dm:Epipe.Site.Container" minOccurs="0" maxOccurs="1">
        <xs:key name="epipeSiteKey">
          <xs:selector xpath="dm:epipe.Site"/>
          <xs:field xpath="sam:epipe.Site/sam:siteId"/>
        </xs:key>
      </xs:element>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="Epipe.Site.Container">
  <xs:complexContent>
    <xs:extension base="lib:Container">
      <xs:sequence>
        <xs:element name="epipe.Site" type="dm:ChangeableEpipeSite"
minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Index