Cisco Prime Cable Provisioning 7.4 Integration Developers Guide

July 27, 2025

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883

THE SPECIFICATIONS AND INFORMATION REGARDING THE PRODUCTS IN THIS MANUAL ARE SUBJECT TO CHANGE WITHOUT NOTICE. ALL STATEMENTS, INFORMATION, AND RECOMMENDATIONS IN THIS MANUAL ARE BELIEVED TO BE ACCURATE BUT ARE PRESENTED WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. USERS MUST TAKE FULL RESPONSIBILITY FOR THEIR APPLICATION OF ANY PRODUCTS.

THE SOFTWARE LICENSE AND LIMITED WARRANTY FOR THE ACCOMPANYING PRODUCT ARE SET FORTH IN THE INFORMATION PACKET THAT SHIPPED WITH THE PRODUCT AND ARE INCORPORATED HEREIN BY THIS REFERENCE. IF YOU ARE UNABLE TO LOCATE THE SOFTWARE LICENSE OR LIMITED WARRANTY, CONTACT YOUR CISCO REPRESENTATIVE FOR A COPY.

The Cisco implementation of TCP header compression is an adaptation of a program developed by the University of California, Berkeley (UCB) as part of UCB's public domain version of the UNIX operating system. All rights reserved. Copyright © 1981, Regents of the University of California.

NOTWITHSTANDING ANY OTHER WARRANTY HEREIN, ALL DOCUMENT FILES AND SOFTWARE OF THESE SUPPLIERS ARE PROVIDED "AS IS" WITH ALL FAULTS. CISCO AND THE ABOVE-NAMED SUPPLIERS DISCLAIM ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THOSE OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OR ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE.

IN NO EVENT SHALL CISCO OR ITS SUPPLIERS BE LIABLE FOR ANY INDIRECT, SPECIAL, CONSEQUENTIAL, OR INCIDENTAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOST PROFITS OR LOSS OR DAMAGE TO DATA ARISING OUT OF THE USE OR INABILITY TO USE THIS MANUAL, EVEN IF CISCO OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Cisco and the Cisco logo are trademarks or registered trademarks of Cisco and/or its affiliates in the U.S. and other countries. To view a list of Cisco trademarks, go to this URL: www.cisco.com/go/trademarks. Third-party trademarks mentioned are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company. (1721R)

Any Internet Protocol (IP) addresses and phone numbers used in this document are not intended to be actual addresses and phone numbers. Any examples, command display output, network topology diagrams, and other figures included in the document are shown for illustrative purposes only. Any use of actual IP addresses or phone numbers in illustrative content is unintentional and coincidental.

Cisco Prime Cable Provisioning 7.4 Integration Developers Guide
© 2025 Cisco Systems, Inc. All rights reserved.

CONTENTS

Preface vii

Audience vii

Product Documentation vii

Related Documentation vii

Obtaining Documentation and Submitting a Service Request viii

CHAPTER 1 **Introduction** 1-1

Overview 1-1

API Functions 1-2

PWS Functions 1-3

CHAPTER 2 **Prime Cable Provisioning Architecture** 2-1

Regional Distribution Unit 2-1

Device Provisioning Engine 2-1

Provisioning Group 2-2

Client API 2-3

Cisco Prime Network Registrar 2-4

Key Distribution Center 2-4

Prime Cable Provisioning Process Watchdog 2-4

SNMP Agent 2-5

Provisioning Web Services (PWS) 2-5

Web User Interface 2-5

CHAPTER 3 **Client and RDU Communication** 3-1

Overview 3-1

Establishing a Connection 3-2

Maintaining a Connection 3-2

Connection Concurrency 3-2

Closing a Connection 3-2

CHAPTER 4 **Batches and Commands** 4-1

Overview 4-1

Batch Rules 4-2
Identifying a Batch 4-2
Batch Processing Flags 4-3
Setting the Reliable Flag 4-4
Setting the Activation Flag 4-5
Setting the Confirmation Flag 4-6
Setting the Publishing Flag 4-6
Setting the Optimistic Locking Flag 4-7
Submitting the Batch 4-8
Submitting in Synchronous Mode 4-8
Submitting in Asynchronous Mode 4-9
Batch Processing Modes 4-9
Batch Results 4-10
Queuing a Batch 4-13
Retrying a Batch 4-14
Handling Errors 4-16
Types of Errors 4-16
Connection Errors 4-16
Batch and Command Errors 4-16
Batch Warnings 4-16

CHAPTER 5 **Events** 5-1
Overview 5-1
Event Registration 5-1
Event Handling 5-3
Event Reliability 5-3

CHAPTER 6 **Getting Started with the Prime Cable Provisioning API** 6-1
Startup Process for API Client 6-1
Configuring the System 6-1
Executing the API Client 6-2
Processing a Batch 6-2

CHAPTER 7 **Java API Client Use Cases** 7-1
Provisioning Operations 7-1
Provisioning API Use Cases 7-3
How to Create an API Client 7-3

Use Cases 7-6
Self-Provisioned Modem and Computer in Fixed Standard Mode 7-7
Adding a New Computer in Fixed Standard Mode 7-10
Disabling a Subscriber 7-13
Preprovisioning Modems/Self-Provisioned Computers 7-15
Modifying an Existing Modem 7-17
Unregistering and Deleting a Subscriber's Devices 7-18
Self-Provisioning First-Time Activation in Promiscuous Mode 7-22
Bulk Provisioning 100 Modems in Promiscuous Mode 7-25
Preprovisioning First-Time Activation in Promiscuous Mode 7-27
Replacing an Existing Modem 7-29
Adding a Second Computer in Promiscuous Mode 7-31
Self-Provisioning First-Time Activation with NAT 7-31
Adding a New Computer Behind a Modem with NAT 7-32
Move Device to Another DHCP Scope 7-32
Log Device Deletions Using Events 7-33
Monitoring an RDU Connection Using Events 7-34
Logging Batch Completions Using Events 7-35
Getting Detailed Device Information 7-35
Searching Using the Device Type 7-40
Searching for Devices Using Vendor Prefix or Class of Service 7-41
Preprovisioning PacketCable eMTA/eDVA 7-42
SNMP Cloning on PacketCable eMTA/eDVA 7-44
Incremental Provisioning of PacketCable eMTA/eDVA 7-45
Preprovisioning DOCSIS Modems with Dynamic Configuration Files 7-48
Optimistic Locking 7-49
Temporarily Throttling a Subscriber's Bandwidth 7-51
Preprovisioning CableHome WAN-MAN 7-53
CableHome with Firewall Configuration 7-54
Retrieving Device Capabilities for CableHome WAN-MAN 7-56
Self-Provisioning CableHome WAN-MAN 7-57
RBAC Administration and Operational Access Control 7-59
Update Protection for Properties at Device Level 7-65
Domain Administration and Instance Level Access Control 7-68
CRS Management 7-74

CHAPTER 8 **Provisioning Web Services (PWS) 8-1**
Overview 8-1
PWS Concepts 8-2

PWS Data Types 8-2
PWS Operations 8-6
Session Operations 8-6
Device Provisioning Operations 8-7
DeviceType Operations 8-17
Generic Device Operation 8-19
Class Of Service Operations 8-20
DHCP Criteria Operations 8-22
File Operations 8-25
Group Operations 8-28
Custom Property Operations 8-30
Poll Operation Status 8-31
Search Operation 8-32
REST PWS Operations 8-33
Server Operations 8-36
PWS Samples 8-41
Create Session 8-42
Add Operation 8-43
Update Operation 8-47
Change Operation 8-53
Get Operation 8-55
Delete Operation 8-62
Status, Reboot, Unregister, Search Operations 8-64
Close Session 8-72
PWS Use Cases 8-79
Registering a New Device 8-80
Unregistering a Device 8-84
Getting DHCP Lease Information of a Device 8-86
Updating Device Details 8-89
Searching a Device 8-93
Supported Query Elements 8-97
Deleting a Device 8-100
Multiple Devices Operations in a Single Request 8-102
Reboot of Device or Devices 8-108

GLOSSARY

INDEX

Preface

This document describes the Cisco Prime Cable Provisioning Application Programming Interface (API) and Provisioning Web Services (PWS), which can be used to integrate Prime Cable Provisioning with Business Support Systems (BSS) and Operational Support Systems (OSS).

Audience

System integrators, network administrators, and network technicians can use this integration guide to integrate the various BSS and OSS with Prime Cable Provisioning. Only experienced users should use these instructions. To use the instructions in this guide, you must be familiar with:

- Prime Cable Provisioning architecture.
- Java programming.

Product Documentation

Note We sometimes update the printed and electronic documentation after original publication. Therefore, you should also review the documentation on http://www.cisco.com for any updates.

See the *Cisco Prime Cable Provisioning 7.4 Documentation Overview* for the list of Prime Cable Provisioning guides.

Related Documentation

See the *Cisco Prime Network Registrar 11.x Documentation Overview* for the list of Cisco Prime Network Registrar guides.

Communications, Services, and Additional Information

- To receive timely, relevant information from Cisco, sign up at Cisco Profile Manager.
- To get the business impact you're looking for with the technologies that matter, visit Cisco Services.
- To submit a service request, visit Cisco Support.

- To discover and browse secure, validated enterprise-class apps, products, solutions and services, visit Cisco Marketplace.
- To obtain general networking, training, and certification titles, visit Cisco Press.
- To find warranty information for a specific product or product family, access Cisco Warranty Finder.

Cisco Bug Search Tool

Cisco Bug Search Tool (BST) is a web-based tool that acts as a gateway to the Cisco bug tracking system that maintains a comprehensive list of defects and vulnerabilities in Cisco products and software. BST provides you with detailed defect information about your products and software.

Introduction

This chapter provides an overview of the Cisco Prime Cable Provisioning Application Programming Interface (API) and Provisioning Web Services (PWS). This chapter also describes the PWS and API functions that you can use to perform the Regional Distribution Unit (RDU) tasks.

Overview

Prime Cable Provisioning automates the tasks of provisioning and managing Customer Premises Equipment (CPE) in a broadband service-provider network. It can be integrated into a new or existing environment through a provisioning application programming interface (API) that lets you control how Prime Cable Provisioning operates. You can use the provisioning API to register devices in Prime Cable Provisioning, assign device configurations, and configure the entire Prime Cable Provisioning provisioning system.

You can also perform the provisioning operations using the Provisioning Web Services (PWS). For information on PWS operations, see Chapter 8, “Provisioning Web Services (PWS).”

Prime Cable Provisioning supports provisioning and managing of CPE that is compliant with the DOCSIS 3.0 specification. It also supports other devices like Computer, PacketCable Multimedia Terminal Adaptor (Packetcable MTA), Set Top Box (STB), CableHomeWanData, and CableHomeWanMan.

Using the Java API, you can integrate various Business Support Systems (BSS) and Operational Support Systems (OSS) with Prime Cable Provisioning. The PWS and Java API are the programmatic interfaces through which the various BSS and OSS clients connect to the RDU, which is the central server in a Prime Cable Provisioning deployment.

You can use the Prime Cable Provisioning API to:

- Register devices in the RDU database.
- Assign configuration policies for devices.
- Execute set operations on the CPE.
- Configure the Prime Cable Provisioning provisioning system.
- Manage CRS requests

Note Use this guide along with the following resources that are integrated with the Prime Cable Provisioning software:

- API Javadocs,
- Sample API client code

These resources are located in the home directory. The default home directory is */opt/CSCObac*.

API Functions

Using the Prime Cable Provisioning API, you can perform the following operations:

- Provisioning operations.

  You can:

  - Add, modify, and search device records in the RDU database.
  - Associate device records with Classes of Service in the RDU database.
  - Associate device records with the groups in the RDU database.
  - Retrieve discovered device data stored in the RDU database.
  - Retrieve device operation history from the RDU database.
  - Retrieve device faults from the Prime Cable Provisioning servers.

- Device management operations.

  You can:

  - Retrieve live data, such as statistics, from a device.
  - Execute diagnostics on a device.
  - Reset the device.
  - Reset the device settings to default configuration.
  - Perform individual sets on a device.

- System configuration and management operations.

  You can:

  - Configure Class of Service objects in the RDU.
  - Manage firmware rules, configuration templates, and other files.
  - Configure device grouping objects in the RDU.
  - Configure licenses.
  - Configure users.
  - Configure system settings for Prime Cable Provisioning.
  - Retrieve Prime Cable Provisioning server status and statistics.
  - Configure user groups, roles, domains, and privileges.
  - Configure CRS

Note You can also perform all system configuration and management operations from the Prime Cable Provisioning administrator user interface. For details on how to perform these operations, see the *Cisco Prime Cable Provisioning 7.4User Guide*.
For more details on how to perform provisioning and device management operations using Java API, see Chapter 7, "Java API Client Use Cases."

PWS Functions

For provisioning operations, you can also use PWS which is a REST based web service. For information on supported provisioning operations using PWS, see Chapter 8, "Provisioning Web Services (PWS)".

Prime Cable Provisioning Architecture

This chapter describes the basic Prime Cable Provisioning architecture, and the components involved in Prime Cable Provisioning.

Regional Distribution Unit

The RDU is the primary server in the Prime Cable Provisioning setup. You must install the RDU on a server running the Linux operating system.

The functions of the RDU include:

- Managing device configuration generation
- Generating configurations for devices and distributing them to DPEs for caching
- Synchronizing with DPEs to keep device configurations up to date
- Processing API requests for all Prime Cable Provisioning functions
- Managing the Prime Cable Provisioning system

The RDU supports the addition of new technologies and services through an extensible architecture.

Cisco Prime Cable provisioning is deployed in High Availability (HA) environment to provide an error-free and continued provisioning service. The HA environment involves setting up the critical components in a failover pair. This ensures that the service continuity is maintained even if a critical component fails to respond, and also helps in disaster recovery. For a medium or large scale setup, it is required to have a redundant service to adhere to the Service Level Agreements (SLAs) and Service Level Objectives (SLOs). The RDU redundancy in Prime Cable Provisioning facilitates setting up RDU in HA environment with a two node failover pair. If the primary RDU node fails or becomes unresponsive, the secondary RDU node controls the provisioning service. For more information on RDU redundancy, see the *Cisco Prime Cable Provisioning 7.4 User Guide*.

Device Provisioning Engine

The Device Provisioning Engine (DPE) communicates with CPE to perform provisioning and management functions.

The RDU generates DHCP instructions and device configuration files, and distributes them to the relevant DPE servers. The DPE caches these DHCP instructions and device configuration files. The DHCP instructions are then used during interactions with the Network Registrar extensions, and configuration files are delivered to the device through the TFTP service.

Prime Cable Provisioning supports multiple DPEs. You can use multiple DPEs to ensure redundancy and scalability.

The DPE handles all configuration requests, including providing configuration files for devices. It is integrated with the Network Registrar DHCP server to control the assignment of IP addresses for each device. Multiple DPEs can communicate with a single DHCP server.

In the DPE, the configurations are compressed using Delta Compression technique of RFC 328 to reduce overall DPE cache size for better scalability.

The DPE manages these activities:

- Synchronizes with the RDU to retrieve the latest configurations for caching.
- Generates last-step device configuration (for instance, DOCSIS timestamps).
- Provides the DHCP server with instructions controlling the DHCP message exchange.
- Delivers configuration files through TFTP.
- ToD server.
- Integrates with Network Registrar.
- Provisions voice-technology services.

You must install the DPE on a server that runs the Linux operating system. Configure and manage the DPE from the CLI, which you can access locally or remotely through Telnet. For specific information on the CLI commands that DPE supports, see the *Cisco Prime Cable Provisioning 7.2.1 DPE CLI Reference*.

Note During installation, you must configure each DPE for the:

- Name of the provisioning group to which the DPE belongs. This name determines the logical group of devices that the DPE services.
- IP address and port number of the RDU.

For more information on DPE, see the *Cisco Prime Cable Provisioning 7.4 User Guide*.

Provisioning Group

A provisioning group is designed to be a logical (typically geographic) grouping of servers that usually consists of one or more DPEs and a failover pair of DHCP servers. Each DPE in a given provisioning group caches identical sets of configurations from the RDU, thus enabling redundancy and load balancing. As the number of devices grows, you can add additional provisioning groups to the deployment.

Note The servers for a provisioning group are not required to reside at a regional location. They can just as easily be deployed in the central network operations center.

Provisioning groups enhance the scalability of the Prime Cable Provisioning deployment by making each provisioning group responsible for only a subset of devices. This partitioning of devices can be along regional groupings or any other policy that the service provider defines. To scale a deployment, the service provider can:

- Upgrade existing DPE server hardware
- Add DPE servers to a provisioning group
- Add provisioning groups

To support redundancy and load sharing, each provisioning group can support any number of DPEs. As the requests come in from the DHCP servers, they are distributed between the DPEs in the provisioning group and an affinity is established between the devices and a specific DPE. This affinity is retained as long as the DPE state within the provisioning group remains stable.

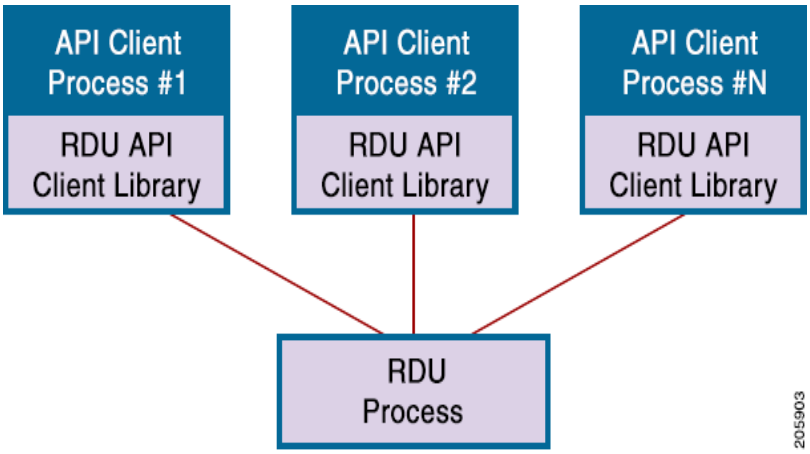
For more information on provisioning groups, see the *Cisco Prime Cable Provisioning 7.4 User Guide*.

Client API

The client API provides total client control over Prime Cable Provisioning capabilities. The API enables the client on a remote host to communicate with the RDU.

The API client library exposes the client to a single logical interface. For information on the objects and functions of this interface, see the API Javadocs in the Prime Cable Provisioning installation directory. Figure 2-1 shows three remote clients accessing the RDU using the API client library.

Figure 2-1 Embedded Client Library



The API client library is packaged in the bpr.jar, bacbase.jar, and bac-common.jar files, located at *BPR_HOME/lib*, where *BPR_HOME* refers to the home directory on which you install Prime Cable Provisioning.

Note For successful communication between Prime Cable Provisioning RDU and 4.x and 4.x.x clients, ensure that the bpr.jar, bacbase.jar, and bac-common.jar files are copied to the 4.x and 4.x.x API client setup. These jars are loaded to the appropriate classpaths. We recommend that you use Java version 1.6.0_32 or later to support the client API in Prime Cable Provisioning.

Cisco Prime Network Registrar

Cisco Prime Network Registrar provides the DHCP and DNS functionality in Prime Cable Provisioning. The DHCP extension points on Prime Network Registrar integrate Prime Cable Provisioning with Prime Network Registrar. Using these extensions, Prime Cable Provisioning examines the content of DHCP requests to detect device type, manipulates the content according to its configuration, and delivers customized configurations for devices that it provisions.

For more information on Cisco Prime Network Registrar, see the *Cisco Prime Network Registrar 9.x User Guide*.

Note We recommend you to use the CPNR 9.x version, however CPNR 8.2.x or later versions are also supported with Cisco Prime Cable Provisioning 6.0.

Key Distribution Center

The Key Distribution Center (KDC) authenticates PacketCable MTAs and also grants service tickets to MTAs. As such, it must check the MTA certificate, and provide its own certificates so that the MTA can authenticate the KDC. It also communicates with the DPE (the provisioning server) to validate that the MTA is provisioned on the network.

The KDC requires a license to function. Obtain a KDC license from your Cisco representative and install it in the correct directory.

You must install the KDC on a server that runs the Linux operating system.

The certificates used to authenticate the KDC are not shipped with Prime Cable Provisioning. You must obtain the required certificates from Cable Television Laboratories, Inc. (CableLabs), and the content of these certificates must match those that are installed in the MTA.

During PCP installation, the KDC has several default properties that are populated into the kdc.ini properties file located at BPR_HOME/kdc/linux directory.You can edit this file to change values as operational requirements dictate.

The KDC also supports the management of multiple realms. For more information on KDC, see the *Cisco Prime Cable Provisioning 7.4 User Guide*.

Prime Cable Provisioning Process Watchdog

The Prime Cable Provisioning process watchdog is an administrative agent that monitors the runtime health of all Prime Cable Provisioning processes. This watchdog process ensures that if a process stops unexpectedly, it is automatically restarted. One instance of the Prime Cable Provisioning process watchdog runs on every system which runs Prime Cable Provisioning components.

You can use the Prime Cable Provisioning process watchdog as a command-line tool to start, stop, restart, and determine the status of any monitored processes.

For more information on how to manage the monitored processes, see the *Cisco Prime Cable Provisioning 7.4 User Guide*.

SNMP Agent

Prime Cable Provisioning provides basic SNMP v2-based monitoring of the RDU and DPE servers. The Prime Cable Provisioning SNMP agents support SNMP informs and traps, collectively called notifications.

You can configure the SNMP agent:

- On the RDU, using the SNMP configuration command-line tool or from the API, see the *Cisco Prime Cable Provisioning 7.4 User Guide*.
- On the DPE, using the snmp-server CLI commands, see the *Cisco Prime Cable Provisioning 7.4 DPE CLI Reference*.

Provisioning Web Services (PWS)

Provisioning Web Services (PWS) is a REST based web service that supports provisioning operations. A separate web server is installed in the network infrastructure to host PWS.

For more information on PWS, see Chapter 8, "Provisioning Web Services (PWS)".

Web User Interface

The Prime Cable Provisioning administrator user interface is a web-based application for central management of the Prime Cable Provisioning system. You can use this system to:

- Configure global defaults
- Define custom properties
- Add, modify, and delete Class of Service
- Add, modify, and delete DHCP Criteria
- Add, modify, and delete devices
- Group devices
- View server status and server logs
- Manage roles, user groups, and domains
- Enable, disable, pause, and resume CRS
- View, filter, and delete any CRS request

For more information on Prime Cable Provisioning administrator user interface, see the *Cisco Prime Cable Provisioning 7.4 User Guide*.

Client and RDU Communication

This chapter describes the communication between the RDU Java client library and the RDU, and describes how to establish, maintain, and close the connection between the RDU Java client library and the RDU.

Overview

The Prime Cable Provisioning API communicates with the RDU in a Prime Cable Provisioning deployment over TCP/IP. The TCP/IP based interactions are secured using the Secured Socket Layer (SSL) protocol. For client and RDU communication, SSL secures the following interactions:

- Clients using the Prime Cable Provisioning API to interact with the RDU.
- Client interaction with the PWS interface.

The API client library initiates the connection between the API and the RDU. The RDU does not try to establish a connection between itself and the API.

When the RDU Java client library initiates and establishes connectivity between API and RDU, the information flows in both the directions; with the RDU Java client library submitting requests to the RDU, and the RDU responding to those requests. The bilateral heartbeat messages enable the API client and the RDU to maintain a bidirectional connection.

Note The network administrator must ensure that:

- IP connectivity exists between the client and the RDU.
- The TCP port that the RDU listens on is opened through a firewall between the client API and the RDU. The default TCP port is 49187 for nonsecured communication with RDU and 49188 for secured communication with RDU. The RDU uses these TCP ports to bind itself to all network interfaces.
- For secure communication, the SSL function must be established between the Prime Cable provisioning components. The SSL secures all inbound communication within Prime Cable Provisioning components through the use of Secure Socket Layer (SSL) 3.0 or and TLS 1.0 protocols. The SSL facilitates encryption of default and custom properties and supports both signed and unsigned certificates. For more information on SSL, see the *Cisco Prime Cable Provisioning 7.4 User Guide*.

Establishing a Connection

The client establishes a connection with the RDU by passing the following parameters:

- Hostname of the RDU; for example, rdu.mso.com
- Selected mode of communication with the RDU; non-secured or secured. For secured communication, you must configure the SSL function. For information on how to configure SSL, see the *Cisco Prime Cable Provisioning 7.4 User Guide*.
- Port for communication with the RDU; the default non-secured port is 49187 and secured port is 49188.
- Administrator username; the default administrator username is **admin**.
- Administrator password; the default administrator password is **changeme**.

You can use the following code to establish a connection between the RDU and the RDU Java client library:

```
final PACEConnection connection =
   PACEConnectionFactory.newInstance(
         "rdu.mso.com", 49187, "admin", "changeme");
```

The connection between the RDU Java client library and RDU is maintained until it is explicitly closed. See Closing a Connection, page 3-2 for information on how to close a connection.

Maintaining a Connection

The RDU Java client library automatically maintains the connection between the client and RDU. In case the connection breaks in the network layer because of congestion, routing problems, or other issues, the RDU Java client library automatically reconnects to the RDU. The RDU Java client library tries to reconnect to the RDU until the connectivity is restored.

The reconnection process is automatic and does not impact your code while the RDU interacts with the library. For example, a synchronous call to submit a batch blocks the thread and returns the results when the results are available as usual; even if the RDU Java client library had to automatically reconnect to the RDU.

Connection Concurrency

The RDU Java client library maintains a single TCP connection to the RDU. This connection can be used for any number of requests and responses. Multiple threads can use the same single connection object.

While there is only a single underlying TCP connection, many Provisioning API Command Engine (PACE) connection instances can be created. If there is a need for multiple Prime Cable Provisioning users in a single client, then multiple PACE connections are required.

Closing a Connection

The connection between the RDU and the RDU Java client library is maintained until you explicitly close the connection. You can use the following code to close the connection:

```
connection.releaseConnection();
```

Batches and Commands

This chapter provides an overview of a batch, the commands contained in a batch, and how a batch is processed in Prime Cable Provisioning.

Overview

A batch object:

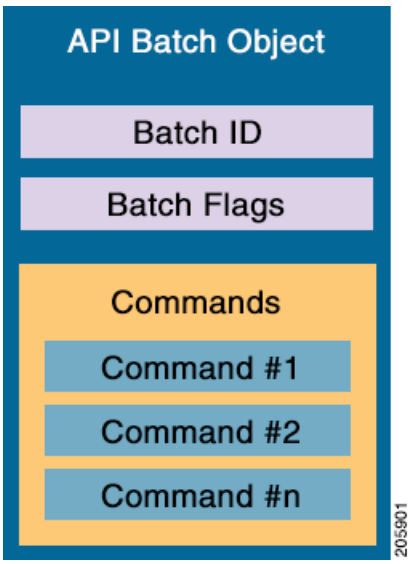
- Is a container for commands that the RDU must execute.
- Contains methods that control how the RDU executes the commands and returns results.

A command represents an operation that is performed on an object in the RDU database. For example, to add a new device, the client issues an add command from the API to the RDU.

The batch lifecycle (create, post, execute, return results) demands two entities to communicate over a network. For this communication, a provisioning client in Prime Cable Provisioning submits API requests to the RDU in the form of batches that contain single or multiple commands.

Figure 4-1 illustrates the concept of batch processing.

Figure 4-1 API Batch Object



Batches are atomic units; either all the commands in the batch succeed or none of the commands succeeds. If the batch fails, the RDU restores changes that were made to its database. The RDU executes the commands in the same sequence in which they are added to the batch. For more information on batch identification, see Identifying a Batch, page 4-2. For more information on batch flags, see Batch Processing Flags, page 4-3.

Batch Rules

To execute a batch successfully, ensure that you follow rules listed below:

- A batch must contain between 1 and 100 commands. You cannot execute a batch with no commands, or one with more than 100 commands.
- Commands in a batch must either be read or write. You cannot combine read and write commands in a batch. For example, the same batch cannot contain a get device details command (read) as well as an add device command (write).

  Note Commands that perform device operations are write commands.

- Batch commands must relate to device or system configuration. You cannot combine device-related and system-related commands in a batch. For example, you cannot combine a modify Class of Service command (system) and an add device command (device) in the same batch.
- When a batch includes a command that interacts with a device record in the RDU through a device operation or an automatic activation flag, all commands in the batch must relate to the same device record in the RDU.
- If you have multiple device operations, each device operation should be submitted in a single batch.

Identifying a Batch

Every batch that the RDU executes has a unique batch identifier. The batch identifier that the RDU Java client library generates includes the hostname of the local client server and a random number that increments.

The batch identifier helps you to:

- Retrieve batch status from the RDU.
- Correlate the respective batch events in the RDU.

While the RDU Java client library automatically generates a batch identifier, you can specify your own batch identifier based on your requirements.

Note We recommend that you use the batch identifiers that the RDU Java client library generates for you.

If you generate your own batch identifier, ensure that you clearly identify the local client server.

Tip If you have a global transaction identifier, it can be a good idea to include it in the batch identifier in order to monitor the transaction throughout the entire system.

If the RDU detects a duplicate batch identifier, it rejects that batch. Submitting batches with batch identifiers that have already been processed may lead to failure and unexpected results.

You can generate a batch identifier in one of two following ways:

- Using the RDU Java client library —To use the RDU Java client library, use the newBatch methods on the Provisioning API Command Engine (PACE) connection object for a batch without the batch identifier parameter.

  Use the following code to generate a batch identifier using a RDU Java client library:

```
public Batch newBatch()

public Batch newBatch(ActivationMode activation)

public Batch newBatch(PublishingMode publishing)

public Batch newBatch(ActivationMode activation, ConfirmationMode confirmation)

public Batch newBatch(ActivationMode activation, ConfirmationMode confirmation,
                            PublishingMode publishing)

public Batch newBatch(ActivationMode activation,
                            PublishingMode publishing)
```

- By specifying your own identifier — To generate your own batch identifier, use the newBatch methods on the PACE connection object containing the batch identifier parameter.

  Use the following code to generate a batch identifier by specifying your own identifier:

```
public Batch newBatch(String batchId)

public Batch newBatch(String batchId,
                            ActivationMode activation)

public Batch newBatch(String batchId,
                            PublishingMode publishing)

public Batch newBatch(String batchId,
                            ActivationMode activation,
                            ConfirmationMode confirmation)

public Batch newBatch(String batchId,
                            ActivationMode activation,
                            ConfirmationMode confirmation,
                            PublishingMode publishing)

public Batch newBatch(String batchId,
                            ActivationMode activation,
                            PublishingMode publishing)
```

Batch Processing Flags

Batch processing flags control:

- Batch interaction with a device.

- Notifications of batches to external systems. These notifications detail the changes that are made by various operations in a batch.

Prime Cable Provisioning supports the following processing flags, each of which is described in subsequent sections:

- Reliable, see Setting the Reliable Flag, page 4-4.
- Activation, see Setting the Activation Flag, page 4-5.
- Confirmation, see Setting the Confirmation Flag, page 4-6.
- Publishing, see Setting the Publishing Flag, page 4-6.
- Optimistic Locking, see Setting the Optimistic Locking Flag, page 4-7.

Setting the Reliable Flag

Communication between the client and the RDU breaks if:

- The client restarts after posting a batch.
- The RDU restarts after receiving a batch.
- The network connection breaks when the results are being sent. Subsequently, the results are lost.

To handle such issues, Prime Cable Provisioning provides a reliable batch flag. When you enable the reliable flag for a batch, the RDU stores the batch on receiving it, and even if the RDU restarts, the batch is guaranteed to be executed after the restart.

Note You can enable the reliable batch flag for batches that contain write commands, such as add, change, or delete. The get operation is not supported in the reliable mode.

After the batch is executed, the RDU stores the results in its database. Subsequently, the client can obtain results for the batches even after an RDU restart. To obtain the results, the client uses a join operation and the thread blocks till the results are returned or a timeout occurs. If the RDU did not receive the batch, or cleared the results from its database, an error appears. At a time, the RDU stores the results of 1000 reliable batches that were last executed.

Note We recommend that you store all batch identifiers of reliable batches to the disk, before you post a batch. By storing the batch identifiers, the RDU Java client library can query for results even if a client restart occurs.

- To join a reliable batch with a batch identifier using the PACE Connection object:
  - With a timeout:

```
final BatchStatus batchStatus = connection.join(batchId, 5000);
```

Note We recommend that you use a timeout value when using the join feature for reliable batches. Also, because reliable batches add a significant load to the RDU, use it only when client and network reliability outweigh the performance impact.

  - Without a timeout:

```
final BatchStatus batchStatus = connection.join(batchId);
```

- To force a batch to be reliable before submitting a synchronous or asynchronous post, use the following code:

```
// make it reliable
batch.forceBatchReliable();
```

For information on synchronous and asynchronous batches, see Batch Processing Modes, page 4-9.

Setting the Activation Flag

You can use the activation flag in batches that contain write commands and operate on a single device. The activation flag is of two types:

- No Activation—Executes by updating the RDU database and the appropriate DPE caches.

  Batches that include commands for on-connect device operations must use the no-activation flag.

- Automatic Activation—Executes by persisting the changes in the RDU database and by trying to establish contact with the device to obtain the latest configuration.

  Batches that include commands for all immediate device operations must use the automatic-activation flag.

Note Activation flag is not applicable for batches that contain delete commands.

You can mark a batch using the no-activation flag or the automatic-activation flag.

For example, consider a batch that contains a change Class of Service command for a device. If you execute the batch with the no-activation flag, the Class of Service of the device is changed, and the resulting new configuration is sent to the DPEs in the provisioning group. The new data is available in the appropriate DPEs for the next device session.

On the other hand, if you execute the same batch with an automatic-activation flag (to achieve device reset), the RDU sends the new configuration to the provisioning group. Activation does not verify if the configuration was successfully applied on the device. When you execute a batch with the automatic-activation flag, the batch becomes reliable. Also, activation involves updating the RDU database and pushing the updated configuration for the device to the DPE, automatically.

The default device reset (Activation) is done by RDU by sending a SNMP set message from RDU to the device. Remote SNMP reset can also be enabled, in which the device SNMP reset request can be sent from DPE rather than RDU. During reset operation, RDU sends a reset request to DPE and in turn, DPE sends SNMP set to device. For more information, see Configuring Remote SNMP Reset section of *Cisco Prime Cable Provisioning 7.4 User Guide*.

For details on controlling this behavior using the Confirmation flag, see Setting the Confirmation Flag, page 4-6.

Note You can augment or replace the activation logic in the RDU during deployment using an extension. For more information, see the *Cisco Prime Cable Provisioning 7.4 User Guide*.

- You can create a batch with no activation in one of two following ways:
  - Without specifying the flag. Because no-activation is the default, batches are created with the no-activation flag.

```
final Batch batch = connection.newBatch();
```

  - By explicitly setting the flag.

```
final Batch batch = connection.newBatch(
                            ActivationMode.NO_ACTIVATION);
```

- You can create a batch with automatic activation using the following code:

```
final Batch batch = connection.newBatch(
                            ActivationMode.AUTOMATIC);
```

Setting the Confirmation Flag

You can use the confirmation flag to control the behavior of batch activation. You must use the confirmation flag only in batches that have the automatic-activation flag set.

The confirmation flag communicates with the RDU on how the processing of a batch should proceed if there are warnings or errors during activation. For more information on warnings or errors during activation, see Batch Warnings, page 4-16.

Prime Cable Provisioning supports two types of confirmation flags:

- No confirmation
- Custom confirmation.

Unless you specify otherwise, a batch is created with the no confirmation flag.

When you execute a batch with the no-confirmation flag, warnings or errors during activation do not cause the batch to fail. Instead, the batch results contain a warning indicating that activation issues occurred. The batch proceeds and database updates are committed.

When you execute a batch with the custom-confirmation flag and a warning occurs during activation, the batch results contain the warning. The batch proceeds, committing the database updates. However, if an error occurs during activation, and the batch results contain the error, the batch fails, and the database updates get rolled back.

Note You can replace or augment the activation code in the RDU so that the errors or warnings that appear depend on the code in use.

You can create a batch with a no-confirmation flag or a custom-confirmation flag.

- You can create a batch with the no-confirmation flag, using the following code:

```
final Batch batch = connection.newBatch(
                                ActivationMode.AUTOMATIC);
```

- You can create a batch with the custom-confirmation flag, using the following code:

```
final Batch batch = connection.newBatch(
                            ActivationMode.AUTOMATIC,
                            ConfirmationMode.CUSTOM_CONFIRMATION);
```

Setting the Publishing Flag

You can use publishing plug-ins to include custom code that helps notify the external entities of changes the batch make to the RDU database. For information on creating publishing plug-ins in the RDU, see the *Cisco Prime Cable Provisioning 7.4 User Guide*.

You can set the publishing flag in one of three ways:

- No publishing—The publishing plug-in is not called within the batch.
- Publishing with no confirmation—The publishing plug-in is executed. If an error occurs, the batch proceeds and any database change is updated.
- Publishing with confirmation—The publishing plug-in is executed. If an errors occurs, the batch fails and the database updates are rolled back.

Note When you mark a batch with the publishing with confirmation flag, the batch automatically becomes reliable.

You must explicitly specify if a batch is to be created with publishing; otherwise, batches are created using the no-publishing flag.

- You can create a batch with the no-publishing flag in one of two following ways:
  - Without setting any flag. Because the no-publishing flag is the default setting, a batch is thus created:

```
final Batch batch = connection.newBatch();
```

  - By explicitly setting the no-publishing flag:

```
final Batch batch = connection.newBatch(
                          PublishingMode.NO_PUBLISHING);
```

- You can create a batch with the publishing no-confirmation flag using:

```
final Batch batch = connection.newBatch(
                        PublishingMode.PUBLISHING_NO_CONFIRMATION);
```

- You can create a batch with the publishing-with-confirmation flag using:

```
final Batch batch = connection.newBatch(
                        PublishingMode.PUBLISHING_CONFIRMATION);
```

Setting the Optimistic Locking Flag

Because the API client executes in a client-server model, a time interval occurs between a get and a modify cycle. You can use the optimistic locking flag to prevent inconsistent changes being made to devices by different clients, simultaneously.

When you perform a get operation for an object (such as a device), the details map contains the GenericObjectKeys.OID_REVISION_NUMBER key. The value for this key is an object identifier that is encoded with the current revision number for the object. You can add this revision number to the batch to ensure that the object is not changed before the changes in your batch are applied. If the object has changed, as indicated by a different revision number, the batch returns the following error: `BatchStatusCodes.BATCH_NOT_CONSISTENT`.

For example, consider a batch that retrieves a device and change its Class of Service using optimistic locking:

Note This example uses the MAC address 1,6,00:11:22:33:44:55 as device ID.

```
final DeviceID deviceId = DeviceID.getInstance("1,6,00:11:22:33:44:55",
KeyType.MAC_ADDRESS);
```

```
final Batch batchForGet = connection.newBatch();
batchForGet.getDetails(deviceId, null);

final BatchStatus batchStatusForGet = batchForGet.post(10000);

if (batchStatusForGet.isError())
{
    // handle error
}

// we know that we only submitted one command in the
// batch so we can get the first command status

final CommandStatus commandStatus =
             batchStatusForGet.getCommandStatus(0);

// we know we submitted a get details command so we are
// expecting a result of a map
if (commandStatus.getDataTypeCode != CommandStatus.DATA_MAP)
{
      // throw an exception or log a message
      // we are expecting a map and didn't get one
}

final Map<String, Object> result =
    (Map<String, Object>)commandStatus.getData();

final Object consistencyValue = result.get(
            GenericObjectKeys.OID_REVISION_NUMBER);

// change the class of service
final Batch batchForMod = connection.newBatch();
batchForMod.changeClassOfService(deviceId, "gold");
// now do the optimistic locking
final List<Object> list = new ArrayList<Object>();
list.add(consistencyValue);
batchForMod.ensureConsistency(list);

// now when we post we know the device has not been changed
// since our get and our change
// if it has it  be an error
```

Submitting the Batch

The API client submits batches to the RDU synchronously or asynchronously. The API submits batches to the RDU in two modes:

- Submitting in Synchronous Mode, page 4-8
- Submitting in Asynchronous Mode, page 4-9

Submitting in Synchronous Mode

When the API client submits a synchronous batch, the batch blocks the current thread till:

- The RDU returns the results on the batch.
- The batch times out before the RDU returns results.

If the RDU Java client library does not receive a response from the RDU within the specified timeout, a `ProvTimeoutException` is thrown. The error message in the exception indicates that the RDU Java client library did not receive the batch result in the specified time but that the batch execution did not necessarily fail.

You can submit your batch to the RDU in synchronous mode with or without a timeout.

- You can submit a synchronous batch on a PACE connection object with a timeout, using:

```
// posting with timeout (in milliseconds)
final BatchStatus batchStatus = connection.postBatch(batch, 5000);
```

Note We recommend that you post a batch in synchronous mode with a timeout configured. For batches that read or update the database, you can configure a timeout of 30,000 milliseconds (msec). For batches that perform operations on live devices, you can configure a timeout of 60,000 msec.

- You can submit a synchronous batch on a PACE connection object without a timeout, using:

```
// posting with no timeout
final BatchStatus batchStatus = connection.postBatch(batch);
```

Submitting in Asynchronous Mode

When the client submits an asynchronous batch, the RDU Java client library thread that posts a batch to the RDU becomes active again. The RDU Java client library obtains the results using the batch events or, if preferred, does not obtain results at all.

You can submit an asynchronous batch on a PACE connection object, using:

```
// posting async
connection.postBatchNoStatus(batch);
```

To obtain batch results from batch events, the RDU Java client library registers a listener class that implements batch listener through the PACE connection with an appropriate qualifier. The batch listener interface exposes a completed method that has a batch event as its argument, and this method is called for each qualified batch when it completes. The batch event, in turn, provides access to the batch status object, which contains the results of the batch. To correlate between the submitted batch and the results, use the batch identifier.

To receive the results, ensure that the listener is registered before the batch is submitted. See Events to view the various events posted by Prime Cable Provisioning.

Batch Processing Modes

Depending on the commands contained in the batch, the RDU executes the batch in one of two following modes:

- Concurrent
- Nonconcurrent

The concurrent and nonconcurrent modes provide higher throughput at the RDU, without losing data integrity.

When the RDU receives a batch, the commands in the batch determine the mode in which a batch is executed. The RDU executes most batches in concurrent mode. A batch must include either concurrent or nonconcurrent commands; the RDU does not process a mix of concurrent and nonconcurrent commands in a single batch. When running one concurrent batch, you can execute other concurrent batches as well.

If the RDU has to process a batch in nonconcurrent mode, all the batches currently being run in the RDU must have completed execution, and no new batches must have started. Batches you submit at this time are queued. The RDU executes the new batches in the mode in which they are marked, after completing the processing of the nonconcurrent batch; by so doing, the RDU avoids lock conflicts and consistency issues.

Only a few commands cause a batch to run in nonconcurrent mode. These commands relate to the following system configuration operations:

- Configuring Class of Service objects in the RDU.
- Managing firmware rules, configuration templates and other files.
- Configuring device grouping objects in the RDU.
- Configuring licenses.
- Configuring users.
- Configuring system settings.
- Configuring user groups, roles, and domains.

Batch Results

A batch result is the outcome of a batch that the RDU executes. Results are returned either as exceptions or as batch status objects.

When posting a batch, an exception is thrown if:

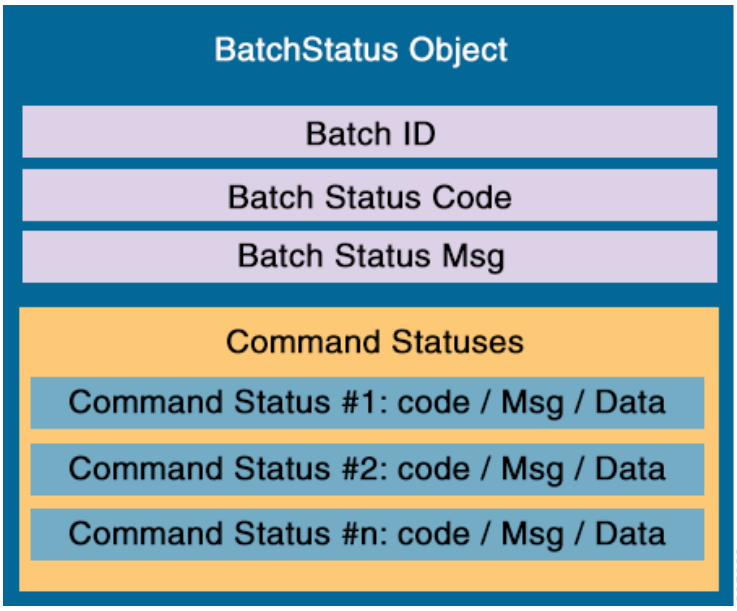
- The batch has already been posted.
- A connection to the RDU cannot be established.
- A timeout occurred when submitting a batch in synchronous mode.

Note These exceptions are rare and are raised as a `ProvisioningException` object.

If there is no `ProvisioningException` thrown, a batch status object is returned. Similar to batches and commands, there are batch status objects and command status objects. A batch status object contains command status entries for each of the commands in the corresponding batch object that was executed. The order of the command status entries matches that of the commands in the batch object.

Figure 4-2 illustrates the structure of a batch status object.

Figure 4-2 Batch Status Object



The batch status object, like a batch, serves as a container. If a single command fails, you can query the batch status to determine if there was a failure and to obtain the command status that contains the details. You can also check the batch status to determine if all the commands succeeded.

Note A batch status object does not always contain a command status. An invalid batch construction, for example, one with a combination of read and write commands, returns a batch status object without command status objects.

- You can query the batch status object to determine:
  - If a single command in a batch failed.
  - The success of all commands in the batch.
- You can query the command status object to determine the details of a command failure. For more information on the status objects, see Batch and Command Errors, page 4-16.

To check whether the batch successfully passes, and to handle errors, if any, use the following code:

```
final BatchStatus batchStatus = connection.post(batch); if (!batchStatus.isError())
{
    // batch passed so all commands passed
}
else
{
    // we need to determine if it was a batch error or a
    // command error that caused this failure

    if (batchStatus.getFailedCommandIndex() == -1)
    {
        // this is a batch only error
        // get the error code and get the error message
        final StringBuilder msg = new StringBuilder(128);
        msg.append("Batch with ID [");
        msg.append(batchStatus.getBatchID());
```

```
            msg.append("] failed with error code [");
            msg.append(batchStatus.getStatusCode());
            msg.append("]. [");
            msg.append(batchStatus.getErrorMessage());
            msg.append("].");

            // throw an exception or log the message
        }
        else
        {
            // this is a batch error caused by a command
            final CommandStatus commandStatus =
                    batchStatus.getFailedCommandIndex();

            // get the error code and get the error message
            final StringBuilder msg = new StringBuilder(128);
            msg.append("Batch with ID [");
            msg.append(batchStatus.getBatchID());
            msg.append("] failed with command error code [");
            msg.append(commandStatus.getStatusCode());
            msg.append("]. [");
            msg.append(commandStatus.getErrorMessage());
            msg.append("].");

            // throw an exception or log the message
        }
    }
```

If a batch successfully passed and you want to view the results before retrieving the details of a device, use the following code.

```
final BatchStatus batchStatus = connection.post(batch); if (batchStatus.isError())
 {
     // handle error
 }
 else
 {
     // we know that we only submitted one command in the
     // batch so we can get the first command status

     final CommandStatus commandStatus =
                 batchStatus.getCommandStatus(0);

     // we know we submitted a get details command so we are
     // expecting a result of a map
     if (commandStatus.getDataTypeCode !=
         CommandStatus.DATA_MAP)
     {
         // throw an exception or log a message
         // we are expecting a map and didn't get one
     }
     else
     {
         final Map<String, Object> result =
                 (Map<String, Object>)commandStatus.getData();
         // now handle the result
     }
 }
```

Queuing a Batch

When the RDU receives a batch from a client, it queues the batch for execution. The priority of a batch determines the queue that the RDU uses for a successful execution of the batch. In case the selected queue is full, the batch is dropped, and the client is notified.

There are ten batch queues, each with the capacity to hold 1000 batches in the order that they were received. Each queue has a different priority. Each queue could contain batches that originate internally or externally. Internal batches are those designated from the DPE and the RDU, and the batches submitted to the RDU Java client library. External batches are those designated from the API client.

Of the ten batch queues:

- Four queues are meant for RDU API client batches (for example, those relating to the administrator user interface and the OSS).
- Six queues are meant for internal batches that relate to:
  - Configuration generation of CPNR DHCP extensions
  - Prime Cable Provisioning server registration
  - DPE cache synchronization
  - DPE configuration regeneration
  - Posted by RDU itself to coordinate access to the database
  - User authentication

The RDU has 100 threads dedicated to execute batches. At a time, the server can execute a maximum number of threads as defined in Table 4-1.

PACE also processes batches from the Configuration Regeneration Service (CRS) and a maximum of one CRS batch is executed for every five batches from the RDU batch queues.

Table 4-1 lists the various batch queues, with the maximum executing threads for each queue.

Table 4-1 Batch Queue

Queue	Batch Origin	Maximum Executing Threads
No Activation	External	25
Automatic Activation		50
Search command		1
Lease Query		25
Configuration Generation	Internal	25
Configuration Regeneration		25
DPE Synchronization		1
Server Registration		1
RDU internal		4
User Authentication		1

Retrying a Batch

If you are unable to receive results, you have to retry the batch posting. You not receive results if:

- A timeout occurred.
- Issues exist in batch submission.
- The client that posted the batch restarts.

Though the RDU Java client library allows you to submit batches only once, you can create a copy of the original batch and re-post it.

There are four basic groups of commands for retrying a batch. Commands that:

- Add new objects to the RDU, such as add a device or a Class of Service.
- Delete objects from the RDU, such as delete a device or a Class of Service.
- Manipulate existing objects in the RDU, such as change the Class of Service for a device, get device details, or get details on a Class of Service.

Note While batches support running commands across groups, mixing commands from different groups adversely impacts batch retrying.

Table 4-2 describes the four different command groups for retrying a batch.

Table 4-2 Command Groups for Retrying a Batch

Command Group	Description
Add new objects to the RDU	For batches that contain commands to add new objects to the RDU, retrying causes issues if the original batch succeeds. You get a command error code that the object already exists. For example, if you try to add objects that already exist, the following batch and command status codes are returned: Batch status code: `BatchStatusCodes.BATCH_FAILED_WRITE` Command status code: `CommandStatusCodes.CMD_ERROR_DEVICEID_EXISTS` **Note** Any other errors that you receive indicates a validate error that is not related to retrying the original batch.
Delete objects in the RDU	For batches that contain commands to delete objects existing in the RDU, retrying is acceptable even if the original batch succeeds. You get a command error code that the object is unknown. For example, if you try to delete an object that has already been deleted, the following batch and command status codes are returned: Batch status code: `BatchStatusCodes.BATCH_FAILED_WRITE` Command status code: `CommandStatusCodes.CMD_ERROR_DEVICEID_UNKNOWN` **Note** Any other errors that you receive indicate a validate error that is not related to retrying the original batch.
Manipulate objects in the RDU	For batches that contain commands that manipulate objects existing in the RDU, retrying does not make any difference. **Note** Any errors that you receive indicate a validate error that is not related to retrying the original batch.
Communicate with live devices	For batches that contain commands that perform operations on live devices, retrying depends on the operation. For example, if an operation adds a new object to the device, deletes an object from the device, or modifies an object from the device, retrying may cause a problem, similar to what an add device command does with the RDU.

Note When retrying a batch for which you created your own batch identifier, ensure that you use the identifier of the original batch. In case you receive a `Duplicate BatchID` error, wait until the original batch has finished execution (for example, using the batch join feature), then submit the batch, if required.

Handling Errors

Troubleshooting integration issues involve handling errors and warnings.

Integration errors may occur because of a:

- Failed RDU Java client library connection to the RDU.
- Failed batch posted in the RDU.

When the connection between the RDU Java client library and the RDU fails, the RDU Java client library tries to reconnect to the RDU. When a batch fails, all database changes are rolled back; a batch status object is returned, indicating that an error occurred.

Batch warnings indicate that the batch succeeded and the changes were committed to the database.

Types of Errors

The two types of errors that occur while integrating the OSS and BSS components to Prime Cable Provisioning are:

- Connection Errors, page 4-16.
- Batch and Command Errors, page 4-16.

Connection Errors

Connection errors are those that occur when the API client library tries to restore a broken connection with the RDU. In general, you can ignore connection errors because the RDU Java client library tries to reconnect to the RDU until the connection is restored. After a connection is restored, processing continues as usual.

You must, however, explicitly address authentication connection errors, such as an `AuthenticationException`. Prime Cable Provisioning does not automatically recover from an authentication error. As an administrator, you must confirm the authentication credentials of the user (username and password).

Batch and Command Errors

To check batch and command errors, see Step 5 in Getting Started with the Prime Cable Provisioning API.

The status objects, BatchStatus and CommandStatus, have methods to return the error code along with a detailed error message. See the API constants *BatchStatusCodes.java* and *CommandStatusCodes.java* in the API Javadocs in the installation directory of the product for the methods that return the error code along with the detailed error message.

Batch Warnings

A warning indicates that:

- The batch has succeeded and the changes have been committed.
- Something of interest has occurred.

The RDU may return warnings for successful batches in two instances:

- When the batch has altered high-level RDU objects, such as a Class of Service or a group. The devices related to these objects must have configurations regenerated from the CRS. The warning indicates the need for configuration regeneration and that this activity occur. The RDU automatically regenerates configurations for these devices.
- During the activation of a batch marked with the default no-confirmation batch flag, if an error occurs, the error appears as a warning, and the batch succeeds.
-

When you execute a batch with the custom-confirmation flag and a warning occurs during activation, the batch results contain the warning. The batch proceeds, committing the database updates. However, if an error occurs during activation, and the batch results contain the error, the batch fails, and the database updates get rolled back.

Events

This chapter provides an overview of the events that the RDU and DPEs provide, and explains how to register and handle these events.

Overview

Using the Prime Cable Provisioning RDU Java client library, you can register for numerous types of events, which are sourced from the RDU and the DPEs. The events that are sourced include:

- Device notification.
- Asynchronous operation notification.
- Batch status events.
- Custom extension events.
- Policy related events.

Event Registration

Events are registered by implementing the appropriate event listener interface. The resulting class is then registered from the PACE connection along with a qualifier.

The qualifier further filters the events that the client receives. If the client wants to receive all events, you can use the QualifyAll qualifier. For an object that can be modified in the RDU, a corresponding event is available in the API. For a complete list of available events, see the *package.com.cisco.provisioning.cpe.events* section in the API Javadocs.

Each event class has a specific qualifier with methods that allow you to refine the events that are to be delivered to the registered listener.

Note You can use only the qualifiers that the RDU Java client library provides. Prime Cable Provisioning does not support implementing your own qualifiers.

For example, to handle all asynchronous operation events that are fired when an on-connect device operation completes:

Step 1 Create the listener class using:

```
public class AsyncEventHandler implements AsyncOperationListener
{
    private boolean m_isOneShot;

    /**
     * The method invoked when a {@link AsyncOperationEvent
     * AsyncOperationEvent} arrives as a result of an async
     * operation completing.
     *
     * <P>
     * @param ev The object containing the {@link AsyncOperationEvent
     * AsyncOperationEvent} data.
     */
    public void completed(final AsyncOperationEvent ev)
    {
        // handle the incoming event
    }

    /**
     * Gets oneShot mode value, specifying whether or not the listener
     * is registered for just one occurrence of the Event.
     *
     * <P>
     * @return <TT>true</TT> if oneShot mode has been set.
     */
    public boolean getOneShot()
    {
        return m_isOneShot;
    }

    /**
     * Sets oneShot mode, specifying that the registration request is
     * for just one occurrence of the Event.
     *
     * <P>
     * @param flg <TT>true</TT> if oneShot mode is being set.
     */
    public void setOneShot(final boolean flg)
    {
        m_isOneShot = flg;
    }
}
```

Step 2 Register the created listener class using the PACE connection:

```
final AsyncEventHandler handler = new AsyncEventHandler();
// use a qualifier that filters all events
final Qualifier qualifier = new QualifyAll();

// register the listener, this contact the RDU
// and from now on we start receiving events
connection.addAsyncOperationLister(handler, qualifier);
```

Note If the connection breaks after the listener is registered, you do not have to reregister the listener. The RDU Java client library automatically registers the listener again.

Step 3 Receive the events. The listener class can be executed when the event arrives.

Step 4 Remove the listener that is created.

You can use any of the following methods to remove a listener:

- Where the implementing class can specify if the listener is one shot. This means that the listener receive only the first qualified event and is removed after receiving its first event.
- By using the PACE connection with an explicit remove listener call.

To explicitly remove the event listener that was created in Step 1:

```
// unregister the listener
// note we must use the same references for the handler
// and the qualifier from the addAsyncOperationListener
// method call
connection.removeAsyncOperationLister(handler, qualifier);
```

Event Handling

When an event is delivered to your registered listener, you must execute any logic that is required. However, because the thread delivering the event does so from the Prime Cable Provisioning RDU Java client library, you must exercise caution.

When running any logic for handling events:

- Avoid any complex logic for your registered listener that uses a Prime Cable Provisioning RDU Java client library thread. If the thread is busy processing the listener, the thread may not be able to deliver events to other listeners or batch results to threads that have completed synchronous posting.
- Re-accessing the PACE connection can cause a deadlock. For example, if you receive an event and then try to submit a new batch while handling the event with the current thread, a deadlock can occur in the RDU Java client library.

To avoid these issues, we recommend that you:

- Keep the logic in your listener short.
- Avoid re-accessing the PACE connection. If you require a more complex logic, you can notify any one of your threads for the processing.

Event Reliability

The RDU Java client library receives events when it maintains a connection with the RDU. If the connection is lost (for example, because of a network crash), events may be lost. You cannot retrieve missed events.

You may also lose events that are generated from the DPE. For example, an interruption in the connection from the DPE to the RDU makes it impossible for the DPE to forward the events to the RDU, and from there, to the client.

For more information on how the RDU Java client library communicates with the RDU, see:

- Chapter 6, “Getting Started with the Prime Cable Provisioning API.”
- Chapter 7, “Java API Client Use Cases.”

Getting Started with the Prime Cable Provisioning API

This chapter describes how to start the API clients and process a batch.

Startup Process for API Client

The startup process for an API client interaction involves:

- Configuring the System, page 6-1.
- Executing the API Client, page 6-2.

Configuring the System

Before executing a simple client, ensure that you have completed the tasks listed in this section.

Note These tasks are part of an initial configuration workflow that you must complete before executing a simple client for the first time. Thereafter, you can execute any number of simple clients.

Table 6-1 System Configuration Workflow

Task	Refer to
1. Install Java Development Kit version 1.6.	Sun Microsystems support site
2. Ensure that files bpr.jar, bacbase.jar, and bac-common.jar are available in the classpath. These *.jar* files are located in the *BPR_HOME/lib* directory.	—
3. Access the Prime Cable Provisioning administrator user interface and ensure that the password that you set for the default **admin** username matches the password that you set on the RDU. The default password is **changeme**.	*Cisco Prime Cable Provisioning 7.4 User Guide*

Executing the API Client

To execute a simple API client:

Note This procedure uses the *AddDeviceExample.java* classfile as an example.

Step 1 Compile the API classfile using the following code:

```
javac -classpath .:bpr.jar:bacbase.jar:bac-common.jar java_source_file
```

For example:

```
javac -classpath .:bpr.jar:bacbase.jar:bac-common.jar AddDeviceExample
```

Note This example assumes that the bpr.jar, bacbase.jar and bac-common.jar files exist in the local directory.

Step 2 Execute the API classfile using the following code:

```
java -cp .:bpr.jar:bacbase.jar:bac-common.jar class_file
```

For example:

```
java -cp .:bpr.jar:bacbase.jar:bac-common.jar AddDeviceExample
```

Step 3 Verify the results.

For example, the AddDeviceExample print success or failure messages. If there is no error, the following message appears:

```
Successfully provisioned device with identifier [OUI-serial-12345]
```

You can also verify the results for the device record from the administrator user interface from the **Devices > Manage Device** page. For more information, see the *Cisco Prime Cable Provisioning 7.4 User Guide*.

Processing a Batch

This section describes how you can connect to the RDU, create a batch, post the batch to the RDU, and verify the result.

Note This procedure uses the *AddDeviceExample.java* classfile as an example.

Step 1 Create a connection to the Provisioning API Command Engine (PACE).

```
// The PACE connection to use throughout the example.  When
// executing multiple batches in a single process, it is advisable
// to use a single PACE connection that is retrieved at the start
// of the application.  When done with the connection, YOU MUST
// explicitly close the connection with the releaseConnection()
// method call.
```

```
PACEConnection connection = null;

// ------------------------------------------------------------------
//
// 1) Connect to the Regional Distribution Unit (RDU).
//
//    The parameters defined at the beginning of this class are
//    used here to establish the connection.  Connections are
//    maintained until releaseConnection() is called.  If
//    multiple calls to getInstance() are called with the same
//    arguments, you must still call releaseConnection() on each
//    connection you received.
//
//    The call can fail for one of the following reasons:
//    - The hostname / port is incorrect.
//    - The authentication credentials are invalid.
//
// ------------------------------------------------------------------
try
{
    connection = PACEConnectionFactory.getInstance(
        // RDU host
        rduHost,
        // RDU port
        rduPort,
        // User name
        userName,
        // Password
        password);
}
catch (PACEConnectionException pce)
{
    // failed to get a connection
    System.out.println("Failed to establish a PACEConnection to ["
            + userName + "@" + rduHost + ":" + rduPort + "]; " +
            pce.getMessage());
    throw new RuntimeException(pce.getMessage());
}
catch (RDUAuthenticationException bae)
{
    // failed to get a connection
    System.out.println("Failed to establish a PACEConnection to ["
            + userName + "@" + rduHost + ":" + rduPort + "]; " +
              bae.getMessage());
       throw new RuntimeException(bae.getMessage());
}
// ------------------------------------------------------------------
```

Step 2 Get a new batch instance.

```
// ------------------------------------------------------------------
//
// 2) Get a new batch instance.
//
//    To perform any operations in the Provisioning API, you must
//    first start a batch.  As you make commands against the batch,
//    nothing  actually start until you post the batch.
//    Multiple batches can be started concurrently against a
//    single connection to the RDU.
//
// ------------------------------------------------------------------
Batch myBatch = connection.newBatch(
                        // No reset
                        ActivationMode.NO_ACTIVATION,
```

```
                        // No need to confirm activation
                        ConfirmationMode.NO_CONFIRMATION,
                        // No publisining to external database
                        PublishingMode.NO_PUBLISHING);
// ------------------------------------------------------------------
```

Step 3 Register the `AddDeviceExample( )` call with the batch.

```
// ------------------------------------------------------------------
//
// 3) Register the add(...) call with the batch.
//
//    Add to the batch the add(...) call.  This  make
//    the batch add the device during the post() operation.  If
//    multiple methods are added to a batch, they  be executed
//    in the order they are registered.  For example, you could
//    add a device and then modify it successfully in a batch.
//
//    The host name and domain name only needs to be specified if the
//    device should have an explicit name assigned to it -- and this is
//    only really useful if you have dynamic DNS enabled in CNR.
//    Properties can be used to store additional information that
//    should be maintained by BPR.  This data  be returned as a
//    response to a query for device details.
//
// ------------------------------------------------------------------
myBatch.add(
// Device type
DeviceType.DOCSIS,
// deviceID list with MACAddress
deviceIDList,
// Host name - Not used in this example
null,
// Domain Name - Not used in this example
null,
// ownerID
accountNumber,
// classOfService - Use default COS
null,
// dhcpCriteria - Use default DHCP Criteria
null,
// properties
null);

        // ------------------------------------------------------------------
```

Step 4 Post a batch to the RDU.

```
//
// 4) Post the batch to the server.
//
//    Executes the batch against the RDU.  All of the
//    methods are executed in the order entered and the data
//    changes are applied against the embedded database in RDU.
//
// ------------------------------------------------------------------
BatchStatus batchStatus = null;
try
{
    batchStatus = myBatch.post();
}
catch (ProvisioningException pe)
{
    System.out.println("Failed to provision device with identifer ["
```

```
            + deviceId + "]; " + pe.getMessage());

        throw new RuntimeException(pe.getMessage());
    }

    // ------------------------------------------------------------------
```

Step 5 Verify the result of the connection.

```
    //
    // 5) Check to see if the batch was successfully posted.
    //
    //    Verify if any errors occurred during the execution of the
    //    batch.  Exceptions occur during post() for truly exception
    //    situations such as failure of connectivity to RDU.
    //    Batch errors occur for inconsistencies such as no lease
    //    information for a device requiring activiation.  Command
    //    errors occur when a particular method has problems, such as
    //    trying to add a device that already exists.
    //
    // ------------------------------------------------------------------
    if (batchStatus.isError())
    {
        // Batch error occurred.
        // we need to determine if it was a batch error or a
        // command error that caused this failure

        if (batchStatus.getFailedCommandIndex() == -1)
        {
            // this is a batch only error
            // get the error code and get the error message
            final StringBuilder msg = new StringBuilder(128);
            msg.append("Batch with ID [");
            msg.append(batchStatus.getBatchID());
            msg.append("] failed with error code [");
            msg.append(batchStatus.getStatusCode());
            msg.append("]. [");
            msg.append(batchStatus.getErrorMessage());
            msg.append("].");


            // throw an exception or log the message
            System.out.println("Failed to add device with identifier ["
                    + deviceId + "]; " +  msg.toString());
        }
        else
        {
            // this is a batch error caused by a command
            final CommandStatus commandStatus =
                batchStatus.getFailedCommandStatus();

            // get the error code and get the error message
            final StringBuilder msg = new StringBuilder(128);
            msg.append("Batch with ID [");
            msg.append(batchStatus.getBatchID());
            msg.append("] failed with command error code [");
            msg.append(commandStatus.getStatusCode());
            msg.append("]. [");
            msg.append(commandStatus.getErrorMessage());
            msg.append("].");

            // throw an exception or log the message
            System.out.println("Failed to add device with identifier ["
                    + deviceId + "]; " + msg.toString());
        }
```

```
}
else
{
    // Successfully added device
    System.out.println("Successfully added device with identifier ["
        + deviceId + "]");
}
```

Step 6 Release the connection to the RDU.

```
// ------------------------------------------------------------------
//
// 6) Release the connection to the RDU.
//
//    Once the last batch has been executed, the connection can
//    be closed to the RDU.  It is important to explictly
//    close connections since it helps ensure clean shutdown of
//    the Java virtual machine.
//
// ------------------------------------------------------------------
connection.releaseConnection();
```

Java API Client Use Cases

This chapter describes the most common Prime Cable Provisioning API use cases that are directly related to device provisioning and device management.

Many system configuration and management operations, such as managing Class of Service, DHCP Criteria, and licenses, are not addressed here because these operations do not require integration with BSS and OSS. You can use the Prime Cable Provisioning admin UI to perform most of these activities. For more information on Prime Cable Provisioning admin UI, see the *Cisco Prime Cable Provisioning 7.4 User Guide*.

For more details on related API calls and sample API client code segments explaining individual API calls and features, refer to these resources that are available in the Prime Cable Provisioning installation directory:

- API Javadoc located at *BAC_61_Linux/docs/BAC_Javadoc_API_Provisioning* on a Linux machine.
- Sample API client code, located at *BPR_HOME/rdu/samples/provapi*.

BPR_HOME is the home directory in which you install Prime Cable Provisioning. The default home directory is */opt/CSCObac*.

This chapter lists various API constants and their functions. To execute any API, you must follow the steps described in the Getting Started with the Prime Cable Provisioning API chapter.

Provisioning Operations

This section describes the following provisioning operation use cases:

Note The classfiles referenced in these use cases; for example, the *AddDeviceExample.java* classfile that illustrates how you can add a device record to the RDU, are only samples that are bundled with the Prime Cable Provisioning software.

- Adding a device record to the RDU—See Table 7-1.
- Modifying a device record in the RDU—See Table 7-2.
- Retrieving discovered device data from the RDU—See Table 7-3.
- Deleting device from the RDU—See Table 7-4.
- Retrieve Devices Matching Vendor Prefix—See Table 7-5.

Table 7-1 Adding a Device Record to the RDU

Classfile	API
`AddDeviceExample.java`	IPDevice.add()
Adds a new device record to the RDU database. Uses the IPDevice.add() API and submits the batch synchronously with the NO_ACTIVATION flag. This operation causes the RDU to generate instructions for the device, which are then cached in the DPE. Figure 7-1 explains adding or modifying a device record in the RDU with Activation mode = No_ACTIVATION.	

Figure 7-1 Change Device Class of Service (Activation mode= NO_ACTIVATION)

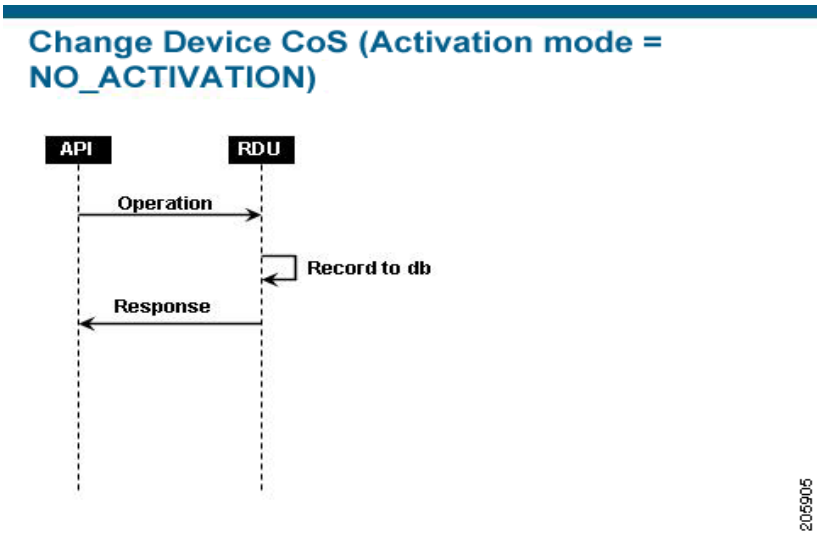


Table 7-2 Modifying a Device Record in the RDU

Classfile	API
`ModifyDeviceExample.java`	IPDevice.changeProperties()
Changes the properties of a device record stored in the RDU. Uses the IPDevice.changeProperties() API and submits the batch synchronously with the NO_ACTIVATION flag. This operation causes the RDU to generate instructions for the device, which are then cached in the DPE.	

Table 7-3 Retrieving Discovered Device Data in the RDU

Classfile	API
`QueryDeviceExample.java`	IPDevice.getDetails()
Retrieves the discovered data of a device that is stored in the RDU. Uses the IPDevice.getDetails() API and submits the batches synchronously using the on-connect mode with the NO_ACTIVATION flag.	

Table 7-4 Delete Device from the RDU

Classfile	API
DeleteDeviceExample.java	IPDevice.delete()
Deletes a device from the RDU. Uses the IPDevice.delete() API and submits the batch synchronously with the NO_ACTIVATION flag.	

Table 7-5 Retrieve Devices Matching Vendor Prefix

Classfile	API
RetriveDevicesMatchingVendorPrefix.java	IPDevice.searchDevice()
Searches for devices that exist in the database. Uses the IPDevice.searchDevice() API to query a list of devices that exist in the database.	

Provisioning API Use Cases

This section presents a series of the most common provisioning application programming interface (API) use cases. See the Cisco Prime Cable Provisioning 7.2.1 API Javadoc for more details and sample Java code segments explaining individual API calls and features.

These use cases are directly related to device provisioning, service provisioning, or both. Many administrative operations, such as managing Class of Service, DHCP Criteria, and licenses are not addressed here. We recommend that you go through the API javadoc for more details on the related API calls. You can also use the administrator user interface to perform most of these activities.

This section describes:

- How to Create an API Client, page 7-3
- Use Cases, page 7-6

How to Create an API Client

Before going through the use cases, you must familiarize yourself with how to create an API client. Use the workflow described in this section to create the API client.

Step 1 Create a connection to the Provisioning API Command Engine (PACE).

```
// The PACE connection to use throughout the example. When
// executing multiple batches in a single process, it is advisable
// to use a single PACE connection that is retrieved at the start
// of the application. When done with the connection, YOU MUST
// explicitly close the connection with the releaseConnection()
// method call.

PACEConnection connection = null;

// Connect to the Regional Distribution Unit (RDU).
//
// The parameters defined at the beginning of this class are
// used here to establish the connection. Connections are
// maintained until releaseConnection() is called. If
```

```
// multiple calls to getInstance() are called with the same
// arguments, you must still call releaseConnection() on each
// connection you received.
//
// The call can fail for one of the following reasons:
// - The hostname / port is incorrect.
// - The authentication credentials are invalid.
// - The maximum number of allowed sessions for the user
// has already been reached.

// However, the number of session validation for an user can be
// avoided by using the following overloaded method :
//
// PACEConnectionFactory.getInstance(
    // RDU host     rduHost,
    // RDU port     rduPort,
    // User name    userName,
    // Password     password
    // Is immediate authentication required authenticateImmediately
    // create a session forcefully          forceLogin
// )
try
{
    connection = PACEConnectionFactory.getInstance(

         // RDU host      rduHost,
         // RDU port      rduPort,
         // User name     userName,
         // Password       password);
}
catch (PACEConnectionException e)
{
// Handle connection error:

System.out.println("Failed to establish a PACEConnection to [" +
    userName + "@" + rduHost + ":" + rduPort + "]; " +
    e.getMessage());

System.exit(1);
}
```

Step 2 Create a new instance of a batch.

```
// To perform any operations in the Provisioning API, you must
// first create a batch. As you add commands to the batch,
// nothing gets executed until you post the batch.
// Multiple batches can be started concurrently against a
// single connection to the RDU.

Batch myBatch = connection.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);
```

Step 3 Register an API command with the batch. The example featured in this step uses the *getDetails(...)* call.

```
// Use the Provisioning API to get all of the information for
// the specified MAC address. Since methods aren't actually
// executed until the batch is posted, the results are not
// returned until after post() completes. The getCommandStatus()
// followed by getData() calls must be used to access the results
// once the batch is posted.

final DeviceID modemMACAddress = DeviceID.getInstance("1,6,00:11:22:33:44:55",
    KeyType.MAC_ADDRESS);

List options = new ArrayList();
options.add(DeviceDetailsOption.INCLUDE_LEASE_INFO);

myBatch.getDetails(modemMACAddress, options);
```

Step 4 Post the batch to the Regional Distribution Unit (RDU) server.

```
// Executes the batch against the RDU. All of the
// methods are executed in the order entered.

BatchStatus bStatus = null;
try
{
// Post batch in synchronous fashion without a timeout. This method will block until
// results are returned. Other API calls are available to submit a batch with timeout
// or in asynchronous (non-blocking) fashion.

bStatus = myBatch.post();
}
catch (ProvisioningException pe)
{
System.out.println("Failed to query for modem with MAC address [" +
    modemMACAddress + "]; " + pe.getMessage());

System.exit(2);
}
```

Step 5 Check the status of the batch.

```
// Check if any errors occurred during the execution of the
// batch. Exceptions occur during post() for truly exceptional
// situations such as failure of connectivity to RDU.
// Batch errors occur for inconsistencies such as no lease
// information for a device requiring activation. Command
// errors occur when a particular method has problems, such as
// trying to add a device that already exists.

//check batchStatus and commandStatus
//for any error

CommandStatus commandStatus = null;
if (batchStatus.getCommandCount() > 0)
{
     commandStatus = batchStatus.getCommandStatus(0);
}
if (batchStatus.isError()
    || commandStatus == null
    || commandStatus.isError())
{
        System.out.println("Failed to query for modem with MAC address [" +
           modemMACAddress + "]; " + bs.getStatusCode().toString() + ", "
           + bs.getErrorMessage());
        for (int i = 0; i < bs.getCommandCount(); i++)
        {
```

```
            CommandStatus cs = bs.getCommandStatus(i);
            System.out.println("Cmd " + i + ": status code "
                    + cs.getStatusCode().toString() + ", " + cs.getErrorMessage());
        }
}
```

If there is no error, the batch call returns a successful result.

```
// Successfully queried for device.

System.out.println("Queried for DOCSIS modem with MAC address ["+
    modemMACAddress + "]");

// Display the results of the command (TreeMap is sorted). The
// data returned from the batch call is stored on a per-command
// basis. In this example, there is only one command, but if
// you had multiple commands all possibly returning results, you
// could access each result by the index of when it was added.
// The first method added is always index 0. From the status of
// each command, you can then access the accompanying data by
// using the getData() call. Since methods can return data of
// different types, you will have to cast the response to the
// type indicated in the Provisioning API documentation.

Map deviceData = (Map)bStatus.getCommandStatus(0).getData();

// Created a sorted map view

Map<String, Object> deviceDetails = new TreeMap(deviceData);
    for(String key: deviceDetails.keySet())
{
    System.out.println(" " + key + "=" + deviceDetails.get(key));
}
```

Step 6 Release the connection.

```
// Once the last batch has been executed, the connection can
// be closed to the RDU. It is important to explicitly
// close connections since it helps ensure clean shutdown of
// the Java virtual machine.

connection.releaseConnection();
```

Use Cases

This section includes these use cases:

- Self-Provisioned Modem and Computer in Fixed Standard Mode, page 7-7
- Adding a New Computer in Fixed Standard Mode, page 7-10
- Disabling a Subscriber, page 7-13
- Preprovisioning Modems/Self-Provisioned Computers, page 7-15
- Modifying an Existing Modem, page 7-17
- Unregistering and Deleting a Subscriber's Devices, page 7-18
- Self-Provisioning First-Time Activation in Promiscuous Mode, page 7-22
- Bulk Provisioning 100 Modems in Promiscuous Mode, page 7-25

- Preprovisioning First-Time Activation in Promiscuous Mode, page 7-27
- Replacing an Existing Modem, page 7-29
- Adding a Second Computer in Promiscuous Mode, page 7-31
- Self-Provisioning First-Time Activation with NAT, page 7-31
- Adding a New Computer Behind a Modem with NAT, page 7-32
- Move Device to Another DHCP Scope, page 7-32
- Log Device Deletions Using Events, page 7-33
- Monitoring an RDU Connection Using Events, page 7-34
- Logging Batch Completions Using Events, page 7-35
- Getting Detailed Device Information, page 7-35
- Searching Using the Device Type, page 7-40
- Searching for Devices Using Vendor Prefix or Class of Service, page 7-41
- Preprovisioning PacketCable eMTA/eDVA, page 7-42
- SNMP Cloning on PacketCable eMTA/eDVA, page 7-44
- Incremental Provisioning of PacketCable eMTA/eDVA, page 7-45
- Preprovisioning DOCSIS Modems with Dynamic Configuration Files, page 7-48
- Optimistic Locking, page 7-49
- Temporarily Throttling a Subscriber's Bandwidth, page 7-51
- Preprovisioning CableHome WAN-MAN, page 7-53
- CableHome with Firewall Configuration, page 7-54
- Retrieving Device Capabilities for CableHome WAN-MAN, page 7-56
- Self-Provisioning CableHome WAN-MAN, page 7-57
- RBAC Administration and Operational Access Control, page 7-59
- Update Protection for Properties at Device Level, page 7-65
- Domain Administration and Instance Level Access Control, page 7-68

Self-Provisioned Modem and Computer in Fixed Standard Mode

The subscriber has a computer installed in a single-dwelling unit and has purchased a DOCSIS cable modem. The computer has a web browser installed.

Desired Outcome

Use this workflow to bring a new unprovisioned DOCSIS cable modem and computer online with the appropriate level of service.

Step 1 The subscriber purchases and installs a DOCSIS cable modem at home and connects a computer to it.

Step 2 The subscriber powers on the modem and the computer, and Prime Cable Provisioning gives the modem restricted access. The computer and modem are assigned IP addresses from restricted access pools.

Step 3 The subscriber starts a web browser, and a spoofing DNS server points the browser to a service provider's registration server (for example, an OSS user interface or a mediator).

Step 4 The subscriber uses the service provider's user interface to complete the steps required for registration, including selecting Class of Service.

Step 5 The service provider's user interface passes the subscriber's information, such as the selected Class of Service and computer IP address, to Prime Cable Provisioning, which then registers the subscriber's modem and computer.

```
// First we query the computer's information to find the
// modem's MAC Address. We use the computer IP Address (the web browser
// received this when the subscriber opened the service provider's
// web interface

PACEConnection connection = PACEConnectionFactory.getInstance(
            "localhost", 49187, "admin", "changeme");

// NO_ACTIVATION is the activation mode because this is a query.
// NO_CONFIRMATION is the confirmation mode because we are not
// attempting to reset the device.
// NO_PUBLISHING is the publishing mode because we are not attempting
// to publish to external database.

Batch batch = connection.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// register getAllForIPAddress to the batch

batch.getAllForIPAddress("10.0.14.38");

BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
batchStatus = batch.post();
}
catch(ProvisioningException e)
{
e.printStackTrace();
}
// Get the LeaseResults object after posting a batch.

CommandStatus commandStatus = batchStatus.getCommandStatus(0);

LeaseResults computerLease = (LeaseResults)commandStatus.getData();

// Derive the modem MAC address from computer's network
// information. The "1,6" is a standard prefix for an Ethernet
// device. The fully qualify MAC Address is required by PCP

StringBuffer modemMACAddress = new StringBuffer();
modemMACAddress.append("1,6,");
modemMACAddress.append(computerLease.getSingleLease().get("relay-agent-remote-id"));
```

```
// Create MacAddress object from the string

MACAddress modemMACAddressObject = new MACAddress(modemMACAddress.toString());

List<DeviceID> modemDeviceIDList = new ArrayList<DeviceID>();
    modemDeviceIDList.add(modemMACAddressObject);

// Create a new batch to add modem device

batch = connection.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// Register add API to the batch

batch.add(DeviceType.DOCSIS, modemDeviceIDList,
    null, null, "0123-45-6789", "silver", "provisioned-cm", null);

// post the batch to RDU server

// Derive computer MAC address from computer's network information.

String computerMACAddress =
    (String)computerLease.getSingleLease().get(DeviceDetailsKeys.MAC_ADDRESS);

// Create a map for computer property.

Map<String, Object> properties = new HashMap<String, Object>();
properties.put(IPDeviceKeys.MUST_BE_BEHIND_DEVICE, modemMACAddress.toString());

List<DeviceID> compDeviceIDList = new ArrayList<DeviceID>();
MACAddress computerMACAddressObject = new MACAddress(computerMACAddress);
compDeviceIDList.add(computerMACAddressObject);

// Register add API to the batch

batch.add(DeviceType.COMPUTER, compDeviceIDList,
    null, null, "0123-45-6789", null, "provisioned-cpe", properties);
try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
}
```

Step 6 The provisioning client calls *performOperation(DeviceOperation deviceOperation, DeviceID deviceID, Map<String,Object> parameters* to reboot the modem and gives the modem provisioned access.

```
// Reset the computer
// Create a new batch
```

```
        batch = connection.newBatch(

            // No reset

            ActivationMode.AUTOMATIC,

            // No need to confirm activation

            ConfirmationMode.NO_CONFIRMATION);

        // Register performOperation command to the batch

        batch.performOperation(DeviceOperation.RESET, modemMACAddressObject, null);

        // Post the batch to RDU server

        try
        {
            batchStatus = batch.post();
        }
        catch(ProvisioningException e)
        {
            e.printStackTrace();
            }
        }
}
```

Step 7 The user interface prompts the subscriber to reboot the computer.

After rebooting, the computer receives a new IP address, and both cable modem and computer are now provisioned devices. The computer has access to the Internet through the service provider's network.

Adding a New Computer in Fixed Standard Mode

A Multiple System Operator (MSO) lets a subscriber have two computers behind a cable modem. The subscriber has one computer already registered and then brings home a laptop from work and wants access. The subscriber installs a hub and connects the laptop to it.

Desired Outcome

Use this workflow to bring a new unprovisioned computer online with a previously provisioned cable modem so that the new computer has the appropriate level of service.

Step 1 The subscriber powers on the new computer and Prime Cable Provisioning gives it restricted access.

Step 2 The subscriber starts a web browser on the new computer and a spoofing DNS server points it to the service provider's registration server (for example, an OSS user interface or a mediator).

Step 3 The subscriber uses the service provider's user interface to complete the steps required to add a new computer.

Step 4 The service provider's user interface passes the subscriber's information, such as the selected Class of Service and computer IP address, to Prime Cable Provisioning, which then registers the subscriber's modem and computer.

```
// First we query the computer's information to find the
// modem's MAC Address. We use the computer IP address (the web browser
// received this when the subscriber opened the service provider's
// web interface.
```

```
PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeme");

// NO_ACTIVATION is the activation mode because this is a query
// NO_CONFIRMATION is the confirmation mode because we are not
// attempting to reset the device
// NO_PUBLISHING is the publishing mode because we are not attempting
// to publish to external database.

Batch batch = connection.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// register getAllForIPAddress to the batch

batch.getAllForIPAddress("10.0.14.39");
BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
}
// Get the LeaseResults object after posting a batch.

CommandStatus commandStatus = batchStatus.getCommandStatus(0);

LeaseResults computerLease = (LeaseResults)commandStatus.getData();

// derive the modem MAC address from computer's network
// information. The "1,6" is a standard prefix for an Ethernet
// device. The fully qualify MAC Address is required by PCP

StringBuffer modemMACAddress = new StringBuffer();
modemMACAddress.append("1,6,");
modemMACAddress.append(computerLease.getSingleLease().get("relay-agent-remote-id"));

// derive computer MAC address from computer's network information.

String computerMACAddress =
    (String)computerLease.getSingleLease().get(DeviceDetailsKeys.MAC_ADDRESS);

//Create a map for computer property.

Map<String, Object> properties = new HashMap<String, Object>();

// setting IPDeviceKeys.MUST_BE_BEHIND_DEVICE on the computer ensures
// that when the computer boots, it will only receive its provisioned
// access when it is behind the given device. If it is not behind
```

```
        // the given device, it will receive default access (unprovisioned)
        // and hence fixed mode.

        properties.put(IPDeviceKeys.MUST_BE_BEHIND_DEVICE, modemMACAddress);

        // the IPDeviceKeys.MUST_BE_IN_PROV_GROUP ensures that the computer
        // will receive its provisioned access only when it is brought up in
        // the specified provisioning group. This prevents the computer
        // (and/or) the modem from moving from one locality to another
        // locality.

        properties.put(IPDeviceKeys.MUST_BE_IN_PROV_GROUP, "bostonProvGroup");

        List<DeviceID> compDeviceIDList = new ArrayList<DeviceID>();
        MACAddress computerMACAddressObject = new MACAddress(computerMACAddress);
        compDeviceIDList.add(computerMACAddressObject);

        batch = connection.newBatch(

            // No reset

            ActivationMode.NO_ACTIVATION,

            // No need to confirm activation

            ConfirmationMode.NO_CONFIRMATION,

            // No publishing to external database

            PublishingMode.NO_PUBLISHING);

        // register add API to the batch

        batch.add(
            DeviceType.COMPUTER,    // deviceType: Computer
            compDeviceIDList,       // compDeviceIDList: the list of DeviceIDs derived from
                                    // computerLease
            null,                   // hostName: not used in this example
            null,                   // domainName: not used in this example
            "0123-45-6789",         // ownerName
            null,                   // class of service: get the default COS
            "provisionedCPE",       // dhpcCriteria: Network Registrar uses this to
                                      // select a modem lease granting provisioned IP address
            properties                // device properties
        );


        // post the batch to RDU server

        try
        {
            batchStatus = batch.post();
        }
        catch(ProvisioningException e)
        {
            e.printStackTrace();
            }
        }
    }
```

Step 5 The user interface prompts the subscriber to reboot the new computer so that Prime Cable Provisioning can give the computer its registered service level.

The computer is now a provisioned device with access to the appropriate level of service.

Disabling a Subscriber

A service provider needs to disable a subscriber from accessing the Internet due to recurring nonpayment.

Desired Outcome

Use this workflow to disable an operational cable modem and computer, so that the devices temporarily restrict Internet access for the user. Additionally, this use case can redirect the user's browser to a special page that could announce:

```
You haven't paid your bill so your Internet access has been disabled.
```

Step 1 The service provider's application uses a provisioning client program to request a list of all of the subscriber's devices from Prime Cable Provisioning.

Step 2 The service provider's application then uses a provisioning client to individually disable or restrict each of the subscriber's devices.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

//get all for owner ID

Batch batch = conn.newBatch();
batch.getAllForOwnerID("0123-45-6789");

BatchStatus batchStatus = null;
try
{
    batchStatus = batch.post();
}
catch(Exception e)
{
    e.printStackTrace();
}
CommandStatus commandStatus = batchStatus.getCommandStatus(0);


//batch success without error, retrieve the result

RecordSearchResults rcSearchResult = (RecordSearchResults)commandStatus.getData();
List<RecordData> resultList = rcSearchResult.getRecordData();

if (resultList != null)
{
// getting the data

for (int i=0; i<resultList.size(); i++)
{
    RecordData rd = resultList.get(i);
    Map<String, Object> detailMap = rd.getDetails();

    //get the deviceType from the detail map

    String deviceType =
        (String)detailMap.get(DeviceDetailsKeys.DEVICE_TYPE);
```

```
            Key primaryKey = rd.getPrimaryKey();

            //only interest in DOCSIS
            if (DeviceType.getDeviceType(deviceType)
                .equals(DeviceType.DOCSIS))
            {

            //change COS

            batch = conn.newBatch();
            batch.changeClassOfService((DeviceID)primaryKey, "DisabledCOS");

            //change DHCPCriteria

            batch.changeDHCPCriteria((DeviceID)primaryKey, "DisabledDHCPCriteria");

            batchStatus = null;
            try
            {
                 batchStatus = batch.post();
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }

        }
        //disable computer

        else if (DeviceType.getDeviceType(deviceType)
            .equals(DeviceType.COMPUTER))
        {
            //change DHCPCriteria

            batch = conn.newBatch();
            batch.changeClassOfService((DeviceID)primaryKey,
                "DisabledComputerCOS");
            batch.changeDHCPCriteria((DeviceID)primaryKey,
                "DisabledComputerDHCPCriteria");

            batchStatus = null;
            try
            {
                batchStatus = batch.post();
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }
    }
```

Note You may need to consider the impact on the CPE behind the modem when defining the characteristics of DisabledCOS and resetting the modem. This is especially important if you have voice endpoints behind the modem, because disrupting the cable modem might affect the telephone conversation in progress at that time.

The subscriber is now disabled.

Preprovisioning Modems/Self-Provisioned Computers

A new subscriber contacts the service provider and requests service. The subscriber has a computer installed in a single-dwelling unit. The service provider preprovisions all its cable modems in bulk.

Desired Outcome

Use this workflow to bring a preprovisioned cable modem, and an unprovisioned computer, online in the roaming standard mode. This must be done so that both devices have the appropriate level of service and are registered.

Step 1 The service provider chooses a subscriber username and password for the billing system.

Step 2 The service provider selects services that the subscriber can access.

Step 3 The service provider's field technician installs the physical cable to the new subscriber's house and installs the preprovisioned device, connecting it to the subscriber's computer.

Step 4 The technician turns on the modem and Prime Cable Provisioning gives it a provisioned IP address.

Step 5 The technician turns on the computer and Prime Cable Provisioning gives it a private IP address.

Step 6 The technician starts a browser application on the computer and points the browser to the service provider's user interface.

Step 7 The technician accesses the service provider's user interface to complete the steps required for registering the computer behind the provisioned cable modem.

```
// First we query the computer's information to find the
// modem's MAC Address. We use the computer IP address (the web browser
// received this when the subscriber opened the service provider's
// web interface

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeme");

// NO_ACTIVATION is the activation mode because this is a query
// NO_CONFIRMATION is the confirmation mode because we are not
// attempting to reset the device
// NO_PUBLISHING is the publishing mode because we are not attempting
// to publish to external database.

Batch batch = connection.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// register getAllForIPAddress to the batch

batch.getAllForIPAddress("10.0.14.38");

BatchStatus batchStatus = null;

// post the batch to RDU server
```

```
try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
}

// Get the LeaseResults object after posting a batch.

CommandStatus commandStatus = batchStatus.getCommandStatus(0);

LeaseResults computerLease = (LeaseResults)commandStatus.getData();

// derive computer MAC address from computer's network information.
String computerMACAddress =
    (String)computerLease.getSingleLease().get(DeviceDetailsKeys.MAC_ADDRESS);


    List<DeviceID> compDeviceIDList = new ArrayList<DeviceID>();
    MACAddress computerMACAddressObject = new MACAddress(computerMACAddress);
    compDeviceIDList.add(computerMACAddressObject);


// NO_ACTIVATION will generate new configuration for the computer,
// however it will not attempt to reset it.
// NO_CONFIRMATION is the confirmation mode because we are not
// attempting to reset the computer because this cannot be done.

batch = connection.newBatch(
    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// register add API to the batch

batch.add(
    DeviceType.COMPUTER,  // deviceType: Computer
    compDeviceIDList,     // compDeviceIDList: the list of DeviceIDs derived from
                          // computerLease
    null,                 // hostName: not used in this example
    null,                 // domainName: not used in this example
    "0123-45-6789",       // ownerName
    null,                 // class of service: get the default COS
    "provisionedCPE",     // dhpcCriteria: Network Registrar uses this to
                          // select a modem lease granting provisioned IP address
    null                  // properties: not used
);

// post the batch to RDU server

try
{
    batchStatus = batch.post();
```

```
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Note The *IPDeviceKeys.MUST_BE_BEHIND_DEVICE* property is not set on the computer and this allows roaming from behind one cable modem to another.

Step 8 The technician restarts the computer and the computer receives a new provisioned IP address.

The cable modem and the computer are now both provisioned devices. The computer has access to the Internet through the service provider's network.

Modifying an Existing Modem

A service provider's subscriber currently has a level of service known as **Silver** and has decided to upgrade to **Gold** service. The subscriber has a computer installed at home.

Note The intent of this use case is to show how to modify a device. You can apply this example to devices provisioned in modes other than roaming standard.

Desired Outcome

Use this workflow to modify an existing modem's Class of Service and pass that change of service to the service provider's external systems.

Step 1 The subscriber phones the service provider and requests to have service upgraded. The service provider uses its user interface to change the Class of Service from **Silver** to **Gold**.

Step 2 The service provider's application makes these API calls in Prime Cable Provisioning:

```
// NO_ACTIVATION is the activation mode because this is a query
// NO_CONFIRMATION is the confirmation mode because we are not
// attempting to reset the device
// NO_PUBLISHING is the publishing mode because we are not attempting
// to publish to external database.

Batch batch = connection.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// replace changeClassOfService to this. Make sure the comment
// on top of this line is still there.
```

```
    batch.changeClassOfService(new MACAddress("1,6,00:11:22:33:44:55")

        // the MACAddress object

        , "Gold");


    // post the batch to the RDU

    BatchStatus batchStatus = null;
    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

The subscriber can now access the service provider's network with the *Gold* service.

Unregistering and Deleting a Subscriber's Devices

A service provider needs to delete a subscriber who has discontinued service.

Desired Outcome

Use this workflow to permanently remove all the subscriber's devices from the service provider's network.

Step 1 The service provider's user interface discontinues service to the subscriber.

Step 2 This step describes how to unregister a subscriber's device and delete a subscriber's device. Deleting a device is optional because some service providers prefer to keep the cable modem in the database unless it is broken. Note however that if you unregister a device using Step 2-a, you cannot delete the device using Step 2-b.

a. To unregister a device, the service provider's application uses a provisioning client program to request a list of all the subscriber's devices from Prime Cable Provisioning, and unregisters and resets each device so that it is brought down to the default (unprovisioned) service level.

Note If the device specified as the parameter to the "unregister" API is already in unregistered state then the status code from the API call will be set to `CommandStatusCodes.CMD_ERROR_DEVICE_UNRE- GISTER_UNREGISTERED_ERROR`. This is normal/expected behavior.

```
// MSO admin UI calls the provisioning API to get a list of
// all the subscriber's devices.

// Create a new connection

PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");
```

```
// NO_ACTIVATION is the activation mode because this is a query
// NO_CONFIRMATION is the confirmation mode because we are not
// attempting to reset the device
// NO_PUBLISHING is the publishing mode because we are not attempting
// to publish to external database.

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

batch.getAllForOwnerID("0123-45-6789"

//  query all the devices for this account number
);

BatchStatus batchStatus = null;
try
{
    batchStatus = batch.post();
}
catch(Exception e)
{
    e.printStackTrace();
}
CommandStatus commandStatus = batchStatus.getCommandStatus(0);


//batch success without error, retrieve the result

RecordSearchResults rcSearchResult = (RecordSearchResults)commandStatus.getData();
List<RecordData> resultList = rcSearchResult.getRecordData();

//  We need to unregister all the devices behind each modem(s) or else the
//  unregister call for that modem will fail.

if (resultList != null)
{
  //Unregister the COMPUTER
  for (int i=0; i<resultList.size(); i++)
{
    RecordData rd = resultList.get(i);
    Map<String, Object> detailMap = rd.getDetails();

    //get the deviceType from the detail map

    String deviceType = (String)detailMap.get(DeviceDetailsKeys.DEVICE_TYPE);


    //only interest in DOCSIS

    if (DeviceType.getDeviceType(deviceType) .equals(DeviceType.COMPUTER))
    {
        Key primaryKey = rd.getPrimaryKey();
```

```
            batch = conn.newBatch();
            batch.unregister((DeviceID)primaryKey);

            batchStatus = null;
            try
            {
               batchStatus = batch.post();
            }
            catch(ProvisioningException e)
            {
               e.printStackTrace();
            }
         }
    }
}
    //  for each modem in the retrieved list:

    for (int i=0; i<resultList.size(); i++)
    {
    RecordData rd = resultList.get(i);
    Map<String, Object> detailMap = rd.getDetails();

    //get the deviceType from the detail map

    String deviceType = (String)detailMap.get(DeviceDetailsKeys.DEVICE_TYPE);


    //only interest in DOCSIS

    if (DeviceType.getDeviceType(deviceType) .equals(DeviceType.DOCSIS))
    {
        Key primaryKey = rd.getPrimaryKey();
        batch = conn.newBatch();
        batch.unregister((DeviceID)primaryKey);
        batchStatus = null;
        try
        {
            batchStatus = batch.post();
        }
        catch(ProvisioningException e)
        {
            e.printStackTrace();
        }
    }
}
```

b. To delete a device, the service provider's application uses a provisioning client program to delete each of the subscriber's remaining devices individually from the database.

```
// Create a new connection

PACEConnection conn =
    PACEConnectionFactory.getInstance("localhost", 49187, "admin", "password1");


// NO_ACTIVATION is the activation mode because this is a query
// NO_CONFIRMATION is the confirmation mode because we are not
// attempting to reset the device
// NO_PUBLISHING is the publishing mode because we are not attempting
// to publish to external database.

Batch batch = conn.newBatch(

    // No reset
```

```
ActivationMode.NO_ACTIVATION,

// No need to confirm activation

ConfirmationMode.NO_CONFIRMATION,

// No publisining to external database

PublishingMode.NO_PUBLISHING);

batch.getAllForOwnerID("0123-45-6789"   //  query all the devices for this account
                                        // number
);

BatchStatus batchStatus = null;
try
{
    batchStatus = batch.post();
}
catch(Exception e)
{
    e.printStackTrace();
}

CommandStatus commandStatus = batchStatus.getCommandStatus(0);


//batch success without error, retrieve the result

RecordSearchResults rcSearchResult = (RecordSearchResults)commandStatus.getData();
List<RecordData> resultList = rcSearchResult.getRecordData();

if (resultList != null)
{

//  for each modem in the retrieved list, delete it

for (int i=0; i<resultList.size(); i++)
{
    RecordData rd = resultList.get(i);
    Map<String, Object> detailMap = rd.getDetails();

    //get the deviceType from the detail map

    String deviceType = (String)detailMap.get(DeviceDetailsKeys.DEVICE_TYPE);


    //only interest in DOCSIS

    if (DeviceType.getDeviceType(deviceType) .equals(DeviceType.DOCSIS))
    {
         Key primaryKey = rd.getPrimaryKey();

         //change COS

         batch = conn.newBatch();
         batch.delete((DeviceID)primaryKey, true);

         batchStatus = null;
         try
        {
             batchStatus = batch.post();
        }
```

```
                catch(ProvisioningException e)
               {
                    e.printStackTrace();
                }
            }
    }
}
```

Self-Provisioning First-Time Activation in Promiscuous Mode

The subscriber has a computer (with a browser application) installed in a single-dwelling unit and has purchased a DOCSIS cable modem.

Desired Outcome

Use this workflow to bring a new unprovisioned DOCSIS cable modem and computer online with the appropriate level of service.

Step 1 The subscriber purchases a DOCSIS cable modem and installs it at home.

Step 2 The subscriber powers on the modem, and Prime Cable Provisioning gives it restricted access.

Step 3 The subscriber starts a browser application on the computer and a spoofing DNS server points the browser to the service provider's registration server (for example, an OSS user interface or a mediator).

Step 4 The subscriber uses the service provider's user interface to complete the steps required for registration, including selecting a Class of Service.

The service provider's user interface passes the subscriber's information to Prime Cable Provisioning, including the selected Class of Service and computer IP address. The subscriber's cable modem and computer are then registered with Prime Cable Provisioning.

Step 5 The user interface prompts the subscriber to reboot the computer.

```
// Create a new connection
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");


Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);


// NO_ACTIVATION is the activation mode because this is a
// query. NO_CONFIRMATION is the confirmation mode because
// we are not attempting to reset the device.
// First we query the computer's information to find the
// modem's MAC address.
// We use the computer's IP address (the web browser
```

```
// received this when the subscriber opened the service
// provider's web interface).
// We also assume that "bostonProvGroup"
// is the provisioning group used in that locality.

List<String> provGroupList = new ArrayList<String>();

provGroupList.add("bostonProvGroup");


batch.getAllForIPAddress("10.0.14.38",

    // ipAddress: restricted access computer lease
    provGroupList
    // provGroups: List containing provgroup
    );

BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
}

// Get the LeaseResults object after posting a batch.

CommandStatus commandStatus = batchStatus.getCommandStatus(0);

LeaseResults computerLease = (LeaseResults)commandStatus.getData();

// Derive the modem MAC address from the computer's network
// information. The 1,6, is a standard prefix for an Ethernet
// device. The fully qualified MAC address is required by PCP

StringBuffer modemMACAddress = new StringBuffer();
modemMACAddress.append("1,6,");
modemMACAddress.append(computerLease.getSingleLease().get("relay-agent-remote-id"));


//create MacAddress object from the string

MACAddress modemMACAddressObject = new MACAddress(modemMACAddress.toString());

List<DeviceID> modemDeviceIDList = new ArrayList<DeviceID>();
modemDeviceIDList.add(modemMACAddressObject);


// NO_ACTIVATION is the activation mode because this is a query
// NO_CONFIRMATION is the confirmation mode because we are not
// attempting to reset the device
// NO_PUBLISHING is the publishing mode because we are not attempting
// to publish to external database.

batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,
```

```
    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

Map<String, Object> properties = new HashMap<String, Object>();

// Set the property PolicyKeys.COMPUTER_PROMISCUOUS_MODE_ENABLED
// to enable promiscuous mode on modem

properties.put(PolicyKeys.COMPUTER_PROMISCUOUS_MODE_ENABLED, Boolean.TRUE);

properties.put(PolicyKeys.COMPUTER_DHCP_CRITERIA, "provisionedCPE");


// enable promiscuous mode by changing the technology default

batch.changeDefaults(DeviceType.DOCSIS,
    properties, null);


// post the batch to RDU server

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
}
batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);


batch.add(
    DeviceType.DOCSIS,    // deviceType: DOCSIS
    modemDeviceIDList,    // macAddress: derived from computer lease
    null,                 // hostName: not used in this example
    null,                 // domainName: not used in this example
    "0123-45-6789",       // ownerID: here, account number from billing system
    "Silver",             // ClassOfService
    "provisionedCM",      // DHCP Criteria: Network Registrar uses this to
                          // select a modem lease granting provisioned IP address
    null                  // properties:
);
```

Step 6 The provisioning client calls *performOperation(…)* to reboot the modem and gives the modem provisioned access.

```
    // Reset the computer
    // create a new batch
    batch = conn.newBatch(

        // No reset

        ActivationMode.AUTOMATIC,

        // No need to confirm activation

        ConfirmationMode.NO_CONFIRMATION);

    // register performOperation command to the batch

    batch.performOperation(DeviceOperation.RESET,
        modemMACAddressObject, null);

    // post the batch to RDU server

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 7 When the computer is rebooted, it receives a new IP address.

The cable modem is now a provisioned device. The computer is not registered with Prime Cable Provisioning, but it gains access to the Internet through the service provider's network. Computers that are online behind promiscuous modems are still available using the provisioning API.

Bulk Provisioning 100 Modems in Promiscuous Mode

A service provider wants to preprovision 100 cable modems for distribution by a customer service representative at a service kiosk.

Desired Outcome

Use this workflow to distribute modem data for all modems to new subscribers. The customer service representative has a list of modems available for assignment.

Step 1 The cable modem's MAC address data for new or recycled cable modems is collected into a list at the service provider's loading dock.

Step 2 Modems that are assigned to a particular kiosk are bulk-loaded into Prime Cable Provisioning and are flagged with the identifier for that kiosk.

Step 3 When the modems are distributed to new subscribers at the kiosk, the customer service representative enters new service parameters, and changes the Owner ID field on the modem to reflect the new subscriber's account number.

```
    // Create a new connection
```

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");


Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);



// The activation mode for this batch should be NO_ACTIVATION.
// NO_ACTIVATION should be used in this situation because no
// network information exists for the devices because they
// have not booted yet. A configuration can't be generated if no
// network information is present. And because the devices
// have not booted, they are not online and therefore cannot
// be reset. NO_CONFIRMATION is the confirmation mode because
// we are not attempting to reset the devices.
// Create a Map for the properties of the modem

Map properties;

// Set the property PolicyKeys.COMPUTER_PROMISCUOUS_MODE_ENABLED to
// enable promiscuous mode on modem.
// This could be done at a system level if promiscuous mode
// is your default provisioning mode.

properties.put(PolicyKeys.COMPUTER_PROMISCUOUS_MODE_ENABLED, Boolean.TRUE);

// The PolicyKeys.CPE_DHCP_CRITERIA is used to specify the DHCP
// Criteria to be used while selecting IP address scopes for
// CPE behind this modem in the promiscuous mode.

properties.put(PolicyKeys.COMPUTER_DHCP_CRITERIA, "provisionedCPE");

// enable promiscuous mode by changing the technology default

batch.changeDefaults(DeviceType.DOCSIS,properties, null);

BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
        }

// for each modem MAC-address in list:
```

```
    ModemLoop:
    {
        batch = conn.newBatch(

           // No reset

           ActivationMode.NO_ACTIVATION,

           // No need to confirm activation

           ConfirmationMode.NO_CONFIRMATION,

           // No publishing to external database

           PublishingMode.NO_PUBLISHING);


    batch.add(
        DeviceType.DOCSIS,   // deviceType: DOCSIS
        modemMACAddressList, // modemMACAddressList: the list of deviceID
        null,                // hostName: not used in this example
        null,                // domainName: not used in this example
        "0123-45-6789",      // ownerID: here, account number from billing system
        "Silver",            // ClassOfService
        "provisionedCM",     // DHCP Criteria: Network Registrar uses this to
                             // select a modem lease granting provisioned IP address
        properties           // properties:
    );

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
    // end ModemLoop.
}
```

Preprovisioning First-Time Activation in Promiscuous Mode

A new subscriber contacts the service provider and requests service. The subscriber has a computer installed in a single-dwelling unit.

Desired Outcome

Use this workflow to bring a new unprovisioned cable modem and computer online with the appropriate level of service.

Step 1 The service provider chooses a subscriber username and password for the billing system.

Step 2 The service provider selects the services that the subscriber can access.

Step 3 The service provider registers the device using its own user interface.

Step 4 The service provider's user interface passes information, such as the modem's MAC address and the Class of Service, to Prime Cable Provisioning. Additionally, the modem gets a CPE DHCP Criteria setting that lets Network Registrar select a provisioned address for any computers to be connected behind the modem. The new modem is then registered with Prime Cable Provisioning.

Step 5 The service provider's field technician installs the physical cable to the new subscriber's house and installs the preprovisioned device, connecting it to the subscriber's computer.

```
//  MSO admin UI calls the provisioning API to pre-provision
//  an HSD modem.

// Create a new connection
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");


Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);


// The activation mode for this batch should be NO_ACTIVATION.
// NO_ACTIVATION should be used in this situation because no
// network information exists for the modem because it has not
// booted. A configuration cannot be generated if no network
// information is present. And because the modem has not booted,
// it is not online and therefore cannot be reset.
// NO_CONFIRMATION is the confirmation mode because we are not
// attempting to reset the modem.
// Create a map for the properties of the modem.


Map<String, Object> properties = new HashMap<String, Object>();

// Set the property PolicyKeys.COMPUTER_PROMISCUOUS_MODE_ENABLED
// to enable promiscuous mode on modem

properties.put(PolicyKeys.COMPUTER_PROMISCUOUS_MODE_ENABLED, Boolean.TRUE);

properties.put(PolicyKeys.COMPUTER_DHCP_CRITERIA, "provisionedCPE");


// enable promiscuous mode by changing the technology default

batch.changeDefaults(DeviceType.DOCSIS, properties, null);

BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
```

```
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
    batch = conn.newBatch(

        // No reset

        ActivationMode.NO_ACTIVATION,

        // No need to confirm activation

        ConfirmationMode.NO_CONFIRMATION,

        // No publishing to external database

        PublishingMode.NO_PUBLISHING);


    MACAddress macAddressObject = new MACAddress("1,6,00:11:22:33:44:55");
    List<DeviceID> modemDeviceIDList = new ArrayList<DeviceID>();
    modemDeviceIDList.add(macAddressObject);

    batch.add(
        DeviceType.DOCSIS,      // deviceType: DOCSIS
        modemDeviceIDList,      // macAddress: derived from computer lease
        null,                   // hostName: not used in this example
        null,                   // domainName: not used in this example
        "0123-45-6789",      // ownerID: here, account number from billing system
        "Silver",            // ClassOfService
        "provisionedCM",     // DHCP Criteria: Network Registrar uses this to
                             // select a modem lease granting provisioned IP address
        null                 // properties:
    );


    // post the batch to RDU server

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 6 The technician powers on the cable modem and Prime Cable Provisioning gives it provisioned access.

Step 7 The technician powers on the computer and Prime Cable Provisioning gives it provisioned access.

The cable modem and the computer are now both provisioned devices. The computer has access to the Internet through the service provider's network.

Replacing an Existing Modem

A service provider wants to replace a broken modem.

Note If the computer has the option to restrict roaming from one modem to another, and the modem is replaced, the computer's MAC address for the modem must also be changed.

Desired Outcome

Use this workflow to physically replace an existing cable modem with a new modem without changing the level of service provided to the subscriber.

Step 1 The service provider changes the MAC address of the existing modem to that of the new modem.

```
    // Create a new connection
    PACEConnection conn = PACEConnectionFactory.getInstance(
        "localhost", 49187, "admin", "password1");

    batch = conn.newBatch(

        // No reset

        ActivationMode.NO_ACTIVATION,

        // No need to confirm activation

        ConfirmationMode.NO_CONFIRMATION,

        // No publishing to external database

        PublishingMode.NO_PUBLISHING);


    MACAddress macAddressObject = new MACAddress("1,6,00:11:22:33:44:55");
    List<DeviceID> modemDeviceIDList = new ArrayList<DeviceID>();
    modemDeviceIDList.add(macAddressObject);

    //  old macAddress: unique identifier for the old modem
    MACAddress oldMacAddress = new MACAddress("1,6,00:11:22:33:44:55");

    // new macAddress: unique identifier for the new modem
    MACAddress newMacAddress = new MACAddress("1,6,00:11:22:33:44:66");
    List<DeviceID> newDeviceIDs = new ArrayList<DeviceID>();
    newDeviceIDs.add(newMacAddress);

    batch.changeDeviceID(oldMacAddress, newDeviceIDs);

    // post the batch to RDU server

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 2 The service provider replaces the cable modem and turns it on.

Step 3 The computer must also be turned on.

The cable modem is now a fully provisioned device with the appropriate level of service, as is the computer behind the cable modem.

Adding a Second Computer in Promiscuous Mode

A subscriber wants to connect a second computer behind an installed cable modem. This case does not require calls to the provisioning API.

Desired Outcome

Use this workflow to ensure that the subscriber's selected service permits the connection of multiple sets of CPE, and that the subscriber has network access from both connected computers.

Step 1 The subscriber connects a second computer behind the cable modem.

Step 2 The subscriber turns on the computer.

If the subscriber's selected service permits connecting multiple sets of CPE, Prime Cable Provisioning gives the second computer access to the Internet.

Self-Provisioning First-Time Activation with NAT

A university has purchased a DOCSIS cable modem with network address translation (NAT) and DHCP capability. The five occupants of the unit each have a computer installed with a browser application.

Desired Outcome

Use this workflow to bring a new unprovisioned cable modem (with NAT) and the computers behind it online with the appropriate level of service.

Step 1 The subscriber purchases a cable modem with NAT and DHCP capability and installs it in a multiple-dwelling unit.

Step 2 The subscriber turns on the modem and Prime Cable Provisioning gives it restricted access.

Step 3 The subscriber connects a laptop computer to the cable modem, and the DHCP server in the modem provides an IP address to the laptop.

Step 4 The subscriber starts a browser application on the computer and a spoofing DNS server points the browser to the service provider's registration server (for example, an OSS user interface or a mediator).

Step 5 The subscriber uses the service provider's user interface to complete the steps required for cable modem registration of the modem. The registration user interface detects that the modem is using NAT and registers the modem, making sure that the modem gets a Class of Service that is compatible with NAT. For details, see Self-Provisioned Modem and Computer in Fixed Standard Mode, page 7-7.

Note Certain cable modems with NAT may require you to reboot the computer to get the new Class of Service settings. If the cable modem and NAT device are separate devices, the NAT device must also be registered similarly to registering a computer.

Adding a New Computer Behind a Modem with NAT

The landlord of an apartment building has four tenants sharing a modem and accessing the service provider's network. The landlord wants to provide Internet access to a new tenant, sharing the building's modem. The modem has NAT and DHCP capability. The new tenant has a computer connected to the modem.

Note This case does not require calls to the provisioning API.

Desired Outcome

Use this workflow to bring a new unprovisioned computer online with a previously provisioned cable modem so that the new computer has the appropriate level of service.

Step 1 The subscriber turns on the computer.

Step 2 The computer is now a provisioned device with access to the appropriate level of service.

The provisioned NAT modem hides the computers behind it from the network.

Move Device to Another DHCP Scope

A service provider is renumbering its network causing a registered cable modem to require an IP address from a different Network Registrar scope.

Desired Outcome

A provisioning client changes the DHCP Criteria, and the cable modem receives an IP address from the corresponding DHCP scope.

Step 1 Change the DOCSIS modem's DHCP Criteria to "newmodemCriteria".

```
// Create a new connection
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.AUTOMATIC,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// AUTOMATIC is the Activation mode because we are attempting
// to reset the modem so that a phone line is disabled
// NO_CONFIRMATION is the Confirmation mode because we don't
// want the batch to fail if we can't reset the modem.
// This use case assumes that the DOCSIS modem has been
// previously added to the database
```

```
    batch.changeDHCPCriteria(
        new MACAddress("1,6,ff:00:ee:11:dd:22"),  // Modem's MAC address or FQDN
              "newmodemCriteria"
    );

    // post the batch to RDU server

    BatchStatus batchStatus = null;
    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 2 The modem gets an IP address from the scope targeted by "newmodemCriteria."

Log Device Deletions Using Events

A service provider has multiple provisioning clients and wants to log device deletions.

Desired Outcome

When any provisioning client deletes a device, the provisioning client logs an event in one place.

Step 1 Create a listener for the device deletion event. This class must extend the *DeviceAdapter* abstract class or, alternatively, implement the *DeviceListener* interface. This class must also override the *deletedDevice(DeviceEvent ev)* method in order to log the event.

```
public DeviceDeletionLogger
        extends DeviceAdapter

        //Extend the DeviceAdapter class.
{
    public void deletedDevice(DeviceEvent ev)

    //Override deletedDevice.
    {
        logDeviceDeletion(ev.getDeviceID());

        //Log the deletion.
    }
}
```

Step 2 Register the listener and the qualifier for the events using the *PACEConnection* interface.

```
DeviceDeletionLogger deviceDeletionLogger =
    new DeviceDeletionLogger();

        // Modem's MAC address or FQDN "newmodemCriteria"

DeviceEventQualifier qualifier = new DeviceEventQualifier();

// We are interested only in device deletion.
```

```
qualifier.setDeletedDevice ();

// Add device listener using PACEConnection

connection.addDeviceListener(deviceDeletionLogger, qualifier
);
```

Step 3 When a device is deleted from the system, the event is generated, and the listener is notified.

Monitoring an RDU Connection Using Events

A service provider is running a single provisioning client and wants notification if the connection between the provisioning client and the RDU breaks.

Desired Outcome

Use this workflow to have the event interface notify the service provider if the connection breaks.

Step 1 Create a listener for the messaging event. This class must extend the *MessagingAdapter* abstract class or, alternatively, implement the *MessagingListener* interface. This class must override the *connectionStopped(MessagingEvent ev)* method.

```
// Extend the service provider's Java program using the
// provisioning client to receive Messaging events.
public MessagingNotifier
    extends MessagingAdapter

    //Extend the MessagingAdapter class.
{
    public void connectionStopped(MessagingEvent ev)

    //Override connectionStopped.
    {
        doNotification(ev.getAddress(), ev.getPort());

        //Do the notification.
    }
}
```

Step 2 Register the listener and the qualifier for the events using the *PACEConnection* interface.

```
MessagingQualifier qualifier =new MessagingQualifier();
qualifier.setConnectionDown();
MessagingNotifier messagingNotifier = new MessagingNotifier();
connnection.addMessagingListener(messagingNotifier, qualifier
);
```

Step 3 If a connection breaks, the event is generated, and the listener is notified. Whenever connectivity is interrupted, the PACE Connection automatically reconnects to the RDU.

Logging Batch Completions Using Events

A service provider has multiple provisioning clients and wants to log batch completions.

Desired Outcome

When any provisioning client completes a batch, an event is logged in one place.

Step 1 Create a listener for the event. This class must extend the *BatchAdapter* abstract class or implement the *BatchListener* interface. This class must override the *completion(BatchEvent ev)* method in order to log the event.

```
public BatchCompletionLogger
    extends BatchAdapter

    //Extend the BatchAdapterclass.
{
    public void completion(BatchEvent ev)
    //Override completion.
    {
        logBatchCompletion(ev.BatchStatus().getBatchID());
        //Log the completion.
    }
}
```

Step 2 Register the listener and the qualifier for the events using the *PACEConnection* interface.

```
BatchCompletionLogger batchCompletionLogger = new BatchCompletionLogger();
BatchEventQualifier qualifier = new BatchEventQualifier();
connection.addBatchListener(batchCompletionLogger , qualifier
);
```

Step 3 When a batch completes, the event is generated, and the listener is notified.

Getting Detailed Device Information

A service provider wants to allow an administrator to view detailed information for a particular device.

Desired Outcome

The service provider's administrative application displays all known details about a given device, including MAC address, lease information, provisioned status of the device, and the device type (if known).

Step 1 The administrator enters the MAC address for the device being queried into the service provider's administrator user interface.

Step 2 Prime Cable Provisioning queries the embedded database for the device details.

```
    // The host name or IP address of the RDU. It is
    // recommended that you normally use a fully-qualified domain name
    // since it lends itself to the greatest flexibility going forward.
    // For example, you could change the host running RDU without
    // having to reassign IPs. For that reason, having an alias for
    // the machine is better than a specific name.

    final String rduHost = "localhost";
```

```
// The port number of RDU on the server.

final int rduPort = 49187;

// The user name for connecting to RDU.

final String userName = "admin";

// The password to use with the username.

final String password = "changeme";

// ------------------------------------------------------------------
// DEVICE PARAMETERS, see IPDevice.getDetails()
// ------------------------------------------------------------------

// The MAC address of the modem to be queried. MAC addresses in BAC
// must follow the simple "1,6,XX:XX:XX:XX:XX:XX" format.

final DeviceID modemMACAddress = DeviceID.getInstance("1,6,00:11:22:33:44:55",
    KeyType.MAC_ADDRESS);

// The PACE connection to use throughout the example. When
// executing multiple batches in a single process, it is advisable
// to use a single PACE connection that is retrieved at the start
// of the application. When done with the connection, YOU MUST
// explicitly close the connection with the releaseConnection()
// method call.

PACEConnection connection = null;

// 1) Connect to the Regional Distribution Unit (RDU).
//
// The parameters defined at the beginning of this class are
// used here to establish the connection. Connections are
// maintained until releaseConnection() is called. If
// multiple calls to getInstance() are called with the same
// arguments, you must still call releaseConnection() on each
// connection you received.
//
// The call can fail for one of the following reasons:
// - The hostname / port is incorrect.
// - The authentication credentials are invalid.
// - The maximum number of allowed sessions for the user
// has already been reached.
try
{
connection = PACEConnectionFactory.getInstance(
// RDU host   rduHost,
// RDU port   rduPort,
// User name  userName,
// Password   password
);
}
catch (PACEConnectionException e)
{
// failed to get a connection

System.out.println("Failed to establish a PACEConnection to [" +
    userName + "@" + rduHost + ":" + rduPort + "]; " +
    e.getMessage());

System.exit(1);
}
```

```
// 2) Create a new batch instance.
//
// To perform any operations in the Provisioning API, you must
// first start a batch. As you make commands against the batch,
// nothing will actually start until you post the batch.
// Multiple batches can be started concurrently against a
// single connection to the RDU.

Batch myBatch = connection.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// 3) Register the getDetails(...) with the batch.

// Use the Provisioning API to get all of the information for
// the specified MAC address. Since methods aren't actually
// executed until the batch is posted, the results are not
// returned until after post() completes. The getCommandStatus()
// followed by getData() calls must be used to access the results
// once the batch is posted.

final DeviceID modemMACAddress = DeviceID.getInstance("1,6,00:11:22:33:44:55",
        KeyType.MAC_ADDRESS);

List options = new ArrayList();
     options.add(DeviceDetailsOption.INCLUDE_LEASE_INFO);

myBatch.getDetails(modemMACAddress, options);

// 4) Post the batch to the server.
//
// Executes the batch against the RDU. All of the
// methods are executed in the order entered and the data
// changes are applied against the embedded database in RDU.

BatchStatus bStatus = null;
try
{
    bStatus = myBatch.post();
}
catch (ProvisioningException pe)
{
System.out.println("Failed to query for modem with MAC address [" +
    modemMACAddress + "]; " + pe.getMessage());

System.exit(2);
}
// 5) Check to see if the batch was successfully posted.
//
// Verify if any errors occurred during the execution of the
// batch. Exceptions occur during post() for truly exception
// situations such as failure of connectivity to RDU.
// Batch errors occur for inconsistencies such as no lease
// information for a device requiring activation. Command
```

```
// errors occur when a particular method has problems, such as
// trying to add a device that already exists.

if (bStatus.isError())
{
// Batch error occurred.

System.out.println("Failed to query for modem with MAC address [" +
    modemMACAddress + "]; " + bStatus.getErrorMessage());

System.exit(3);
}
```

Step 3 The service provider's application presents a page of device data details, which can display everything that is known about the requested device. If the device was connected to the service provider's network, this data includes lease information (for example, IP address and relay agent identifier). The data indicates whether the device was provisioned, and if it was, the data also includes the device type.

```
// Successfully queried for device.
System.out.println("Queried for DOCSIS modem with MAC address ["+
    modemMACAddress + "]");

// Display the results of the command (TreeMap is sorted). The
// data returned from the batch call is stored on a per-command
// basis. In this example, there is only one command, but if
// you had multiple commands all possibly returning results, you
// could access each result by the index of when it was added.
// The first method added is always index 0. From the status of
// each command, you can then access the accompanying data by
// using the getData() call. Since methods can return data of
// different types, you will have to cast the response to the
// type indicated in the Provisioning API documentation.

Map<String, Object> deviceDetails = new HashMap<String,
    Object>((Map)bStatus.getCommandStatus(0).getData());

String deviceType = (String)deviceDetails.get(DeviceDetailsKeys.DEVICE_TYPE);
String macAddress = (String)deviceDetails.get(DeviceDetailsKeys.MAC_ADDRESS);
String fqdn = (String)deviceDetails.get(DeviceDetailsKeys.FQDN);
String duid = (String)deviceDetails.get(DeviceDetailsKeys.DUID);
String host = (String)deviceDetails.get(DeviceDetailsKeys.HOST);
String domain = (String)deviceDetails.get(DeviceDetailsKeys.DOMAIN);

// if the device is DocsisModem, get the COS

String cos   = (String)deviceDetails.get(DeviceDetailsKeys.CLASS_OF_SERVICE);
String dhcpCriteria   = (String)deviceDetails.get(DeviceDetailsKeys.DHCP_CRITERIA);
String provGroup = (String)deviceDetails.get(DeviceDetailsKeys.PROV_GROUP);
Boolean isProvisioned = (Boolean)deviceDetails.get(DeviceDetailsKeys.IS_PROVISIONED);
String ownerID = (String)deviceDetails.get(DeviceDetailsKeys.OWNER_ID);
Boolean isRegistered = (Boolean)deviceDetails.get(DeviceDetailsKeys.IS_REGISTERED);
String oidNumber = (String)deviceDetails.get(GenericObjectKeys.OID_REVISION_NUMBER);

// if the device is a modem, get the device behind

String relayAgentMacAddress =
    (String)deviceDetails.get(DeviceDetailsKeys.RELAY_AGENT_MAC);
String relayAgentDUID = (String)deviceDetails.get(DeviceDetailsKeys.RELAY_AGENT_DUID);

// get the map of Device property

Map deviceProperties = (Map)deviceDetails.get(DeviceDetailsKeys.PROPERTIES);
```

```
// get the map of discovery data v4

Map dhcpdiscovermapv4 =
    (Map)deviceDetails.get(DeviceDetailsKeys.DISCOVERED_DATA_DHCPV4);

// if discovery data is not null, get the inform, response, request and environment
// map from discovery data map

Map dhcpInformMap = (Map)dhcpdiscovermapv4.get("INFORM");
Map dhcpRespMap = (Map)dhcpdiscovermapv4.get("RESPONSE");
Map dhcpReqMap = (Map)dhcpdiscovermapv4.get("REQUEST");
Map dhcpEnvMap = (Map)dhcpdiscovermapv4.get("ENVIRONMENT");

// get the map of lease query v4

Map leasemapv4 = (Map)deviceDetails.get(DeviceDetailsKeys.LEASE_QUERY_DATA_DHCPV4);
String leaseTime = (String)leasemapv4.get(CNRNames.DHCP_LEASE_TIME.toString());
String rebindingTime =
    (String)leasemapv4.get(CNRNames.DHCP_REBINDING_TIME.toString());

String clientLastTransTime =
    (String)leasemapv4.get(CNRNames.CLIENT_LAST_TRANSACTION_TIME.toString());
String clientIPAddress= (String)leasemapv4.get(CNRNames.CLIENT_IPADDRESS.toString());
String relayAgentRemoteID=
    (String)leasemapv4.get(CNRNames.RELAY_AGENT_REMOTE_ID.toString());
String relayAgentCircuitID=
    (String)leasemapv4.get(CNRNames.RELAY_AGENT_CIRCUIT_ID.toString());

// get the map of discovery DHCP v6

Map dhcpdiscovermapv6 =
    (Map)deviceDetails.get(DeviceDetailsKeys.DISCOVERED_DATA_DHCPV6);

// if discovery data is not null , get the inform, response, request and environment
// map from discovery data map

Map dhcpv6InformMap = (Map)dhcpdiscovermapv6.get("INFORM");

Map dhcpv6RespMap = (Map)dhcpdiscovermapv6.get("RESPONSE");

Map dhcpv6ReqMap = (Map)dhcpdiscovermapv6.get("REQUEST");

Map dhcpv6RelReqMap = (Map)dhcpdiscovermapv6.get("RELAY_REQUEST");

Map dhcpv6EnvMap = (Map)dhcpdiscovermapv6.get("ENVIRONMENT");

// get the map of lease query V6

Map leasemapv6 = (Map)deviceDetails.get(DeviceDetailsKeys.LEASE_QUERY_DATA_DHCPV6);

String iaprefixkey = (String)leasemapv6.get(CNRNames.IAPREFIX.toString());
String iaaddrkey = (String)leasemapv6.get(CNRNames.IAADDR.toString());
String leasetimev6 = (String)leasemapv6.get(CNRNames.VALID_LIFETIME.toString());
String renewaltimev6 = (String)leasemapv6.get(CNRNames.PREFERRED_LIFETIME.toString());
String dhcplasttranstimev6 =
    (String)leasemapv6.get(CNRNames.CLIENT_LAST_TRANSACTION_TIME);
String clientIpAddressv6 = (String)leasemapv6.get(CNRNames.CLIENT_IPADDRESS);
String relayagentremoteidv6 = (String)leasemapv6.get(CNRNames.RELAY_AGENT_REMOTE_ID);
String relayagentcircuitidv6 =
    (String)leasemapv6.get(CNRNames.RELAY_AGENT_CIRCUIT_ID);
```

Searching Using the Device Type

A service provider wants to allow an administrator to view data for all DOCSIS modems.

Desired Outcome

The service provider's administrative application returns a list of DOCSIS devices.

Step 1 The administrator selects the search option in the service provider's administrator user interface.

Step 2 Prime Cable Provisioning queries the embedded database for a list of all MAC addresses for the DOCSIS modems.

```
public static void getAllDevicesByDeviceType() throws Exception  {
      DeviceSearchType dst = DeviceSearchType.getByDeviceType(
          DeviceType.getDeviceType(DeviceTypeValues.DOCSIS_MODEM),
          ReturnParameters.ALL);

      RecordSearchResults rs = null;

      SearchBookmark sb = null;

      rs = searchDevice(dst, sb);
      sb = rs.getSearchBookmark();

      while (sb != null)
      {
          // print out the data in the record search result.
          sb = printRecordSearchResults(rs);

          // call the search routine again
          rs = searchDevice(dst, sb);
      }
 }
 private static RecordSearchResults searchDevice(DeviceSearchType dst,
                SearchBookmark sb) throws Exception  {
      RecordSearchResults rs = null;
      final Batch batch = s_conn.newBatch();
      final int numberOfRecordReturn = 10;

      //calling the search API
      batch.searchDevice(dst, sb, numberOfRecordReturn);

      // Call the RDU.
      BatchStatus batchStatus = batch.post();

      // Check for success.
      CommandStatus commandStatus = null;

      if (0 < batchStatus.getCommandCount())
      {
          commandStatus = batchStatus.getCommandStatus(0);
      }
      //check to see if there is an error
      if (batchStatus.isError()
      || batchStatus.isWarning()
      || commandStatus == null
      || commandStatus.isError())
      {
          System.out.println("report batch error.");
          return null;
      }
```

```
        //batch success without error, retrieve the result
        //this is a list of devices
        rs = (RecordSearchResults)commandStatus.getData();
        return rs;
    }


    private static SearchBookmark printRecordSearchResults(RecordSearchResults rs)  throws
Exception  {

        SearchBookmark sb = rs.getSearchBookmark();

        List<RecordData> rdlist = rs.getRecordData();
        Iterator<RecordData> iter = rdlist.iterator();

        while (iter.hasNext())
        {
            RecordData rdObj = iter.next();
            Key keyObj = rdObj.getPrimaryKey();

             System.out.println("DeviceOID: " + ((DeviceID)keyObj).getDeviceId());

            //this is for secondary keys.
            List<Key> deviceList = rdObj.getSecondaryKeys();

            if (deviceList != null && !deviceList.isEmpty())
            {
                for (int i=0; i<deviceList.size(); i++)
                {
                    Key key = deviceList.get(i);
                    System.out.println("DeviceID : " + key.toString());
                }
            }
        }
        return sb;

    }
```

Searching for Devices Using Vendor Prefix or Class of Service

A service provider wants to allow an administrator to search for all devices matching a particular vendor prefix or a particular Class of Service.

Desired Outcome

The service provider's administrative application returns a list of devices matching the requested vendor prefix or the Class of Service.

Step 1 The administrator enters the substring matching the desired vendor prefix into the service provider's administrator user interface.

Step 2 Prime Cable Provisioning queries the embedded database for a list of all MAC addresses for the devices that match the requested vendor prefix or Class of Service. This example illustrates how you can build the search query to retrieve devices using the MAC address. Also see Searching Using the Device Type, page 7-40.

```
DeviceIDPattern pattern = new MACAddressPattern("1,6,22:49:*");
```

```
DeviceSearchType dst = DeviceSearchType.getDevices(pattern, ReturnParameters.ALL);

// To set up search for class of service:

DeviceSearchType searchType = DeviceSearchType.getByClassOfService(

new ClassOfServiceName(name), AssociationType

.valueOf(association), ReturnParameters.ALL);
```

Step 3 The service provider's application requests details on these devices from Prime Cable Provisioning, and presents a page of device data. For each device, the code displays the device type, MAC address, client class, and provisioned status of the device. One device is identified per line.

```
// calling the search procedure

rs = searchDevice(connection, dst, sb);
sb = processRecordSearchResults(rs);

if (rs != null)
{
    while (sb != null)
         {
            // The search returns a search bookmark, which can be used to make
            // the next search call that would return next set of results

            rs = searchDevice(connection, dst, sb);
            sb = processRecordSearchResults(rs);
        }
    }
}
}
```

Preprovisioning PacketCable eMTA/eDVA

A new customer contacts a service provider to order PacketCable voice service. The customer expects to receive a provisioned embedded MTA or embedded DVA.

Desired Outcome

Use this workflow to preprovision an embedded MTA or embedded DVA so that the modem MTA/DVA component has the appropriate level of service when brought online.

Note This use case skips the call agent provisioning that is required for making telephone calls from eMTA/eDVAs.

Step 1 Choose a subscriber username and password for the billing system.

Step 2 Choose the appropriate Class of Service and DHCP Criteria for the modem component and add it to Prime Cable Provisioning.

```
// Create a new connection

PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");
```

```
Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION);

// Let's provision the modem and the MTA component in the same
// batch. This can be done because the activation mode of this
// batch is NO_ACTIVATION. More than one device can be operated
// on in a batch if the activation mode does not lead to more
// than one device being reset.
// To add a DOCSIS modem:

List<DeviceID> modemDeviceIDList = new ArrayList<DeviceID>();
modemDeviceIDList.add(new MACAddress("1,6,01:02:03:04:05:06"));
batch.add(
    DeviceType.DOCSIS,     // deviceType: DOCSIS
    modemDeviceIDList,     // macAddress: scanned from the label
    null,                  // hostName: not used in this example
    null,                  // domainName: not used in this example
    "0123-45-6789",        // ownerID: here, account number from billing system
    "Silver",              // classOfService
    "provisionedCM",       // DHCP Criteria: Network Registrar uses this to
                           // select a modem lease granting provisioned IP address
    null                   // properties: not used
);
```

Step 3 Choose the appropriate Class of Service and DHCP Criteria for the MTA/DVA component and add it to Prime Cable Provisioning.

```
List<DeviceID> packetcableMTADeviceIDList = new ArrayList<DeviceID>();
packetcableMTADeviceIDList.add(new MACAddress("1,6,01:02:03:04:05:07"));

// Continuation of the batch in Step2
// To add the MTA component:

batch.add(
    DeviceType.PACKET_CABLE_MTA,  // deviceType: PACKET_CABLE_MTA
    packetcableMTADeviceIDList,   // macAddress: scanned from the label
    null,                         // hostName: not used in this example, will be auto
                                  // generated
    null,                         // domainName: not used in this example, will be
                                  // auto generated. The FqdnKeys.AUTO_FQDN_DOMAIN
                                  // property must be set somewhere in the property
                                  // hierarchy.
    "0123-45-6789",               // ownerID: here, account number from billing system
    "Silver",                     // ClassOfService
    "provisionedMTA",             // DHCP Criteria: Network Registrar uses this to
                                  // select an MTA lease granting provisioned IP
                                  // address
    null                          // properties: not used
);

BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
```

```
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 4 The embedded MTA or embedded DVA is shipped to the customer.

Step 5 The customer brings the embedded MTA or embedded DVA online and makes telephone calls using it.

SNMP Cloning on PacketCable eMTA/eDVA

An administrator wants to grant SNMP Element Manager access to a PacketCable eMTA/eDVA.

Desired Outcome

An external Element Manager is granted secure SNMPv3 access to the PacketCable eMTA/eDVA.

Note Changes made to RW MIB variables are not permanent and are not updated in the Prime Cable Provisioning configuration for the eMTA/eDVA. The information written into the eMTA/eDVA MIB is lost the next time the MTA powers down or resets.

Step 1 Call the provisioning API method, *performOperation(...)*, passing in the MAC address of the MTA/DVA and the username of the new user to create on the MTA/DVA. This will be the username used in subsequent SNMP calls by the Element Manager.

```
// Create a new connection
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION);

// NO_ACTIVATION is the activation mode because we don't want to
// reset the device.
// NO_CONFIRMATION is the confirmation mode because we are
// not attempting to reset the device.
// The goal here is to create a new user on the MTA indicated
// by the MAC address. The other parameter needed here is the new
// user name, which is passed in the Map.
// Create a map that contains one element - the name of
// the new user to be created on the MTA

HashMap<String, Object> map = new HashMap<String, Object>();
map.put( SNMPPropertyKeys.CLONING_USERNAME, "newUser" );

// The first param is the actual device operation to perform.

batch.performOperation(
```

```
        DeviceOperation.ENABLE_SNMPV3_ACCESS,    // deviceOperation : ENABLE_SNMPV3_ACCESS
        new MACAddress("1,6,00:00:00:00:00:99"), // macORFqdn : MAC Address of the modem
        map                                      // parameters: operation specific
                                                 // parameters
    );

    BatchStatus batchStatus = null;

    // post the batch to RDU server

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 2 The provisioning API attempts to perform an SNMPv3 cloning operation to create an entry on the MTA/DVA for the new user passed in Step 1. The keys used in the new user entry row are a function of two passwords defined within Prime Cable Provisioning. These passwords will be made available to the customer and the RDU command passes these passwords (the auth and priv password) through a key localization algorithm to create an auth and priv key. These are stored, along with the new user, in the eMTA's/eDVA's user table.

Note The auth and priv passwords mentioned in this step may be changed by setting `SNMPPropertyKeys.CLONING_AUTH_PASSWORD` (*/snmp/cloning/auth/password*) and `SNMPPropertyKeys.CLONING_PRIV_PASSWORD` (*/snmp/cloning/priv/password*) properties, respectively, in the *rdu.properties* configuration file.

Step 3 The customer issues SNMPv3 requests using the specified username, passwords, and key localization algorithm to allow for secure communication with the MTA/DVA.

Incremental Provisioning of PacketCable eMTA/eDVA

A customer has a PacketCable eMTA/eDVA in service with its first line (endpoint) enabled. The customer wants to enable the second telephone line (endpoint) on the eMTA/eDVA and connect a telephone to it.

Desired Outcome

The customer should be able to connect a telephone to the second line (endpoint) on the eMTA/eDVA and successfully make phone calls from it without any service interruption.

Note In order to use the second line on the eMTA/eDVA, the Call Agent needs to be configured accordingly. This use case does not address provisioning of call agents.

Step 1 The service provider's application invokes the Prime Cable Provisioning API to change the Class of Service of the eMTA/eDVA. The new Class of Service supports two endpoints on the eMTA/eDVA. This change in Class of Service does not take effect until the eMTA/eDVA is reset. Disrupting the eMTA/eDVA is not desirable; therefore, incremental provisioning is undertaken in the next step.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION);

// NO_ACTIVATION is the activation mode because we don't want to
// reset the device.
// NO_CONFIRMATION is the Confirmation mode because we are not
// disrupting the device.

batch.changeClassOfService(
    new MACAddress("1,6,ff:00:ee:11:dd:22"),  / eMTA's MAC address or FQDN
    "twoLineEnabledCOS"                      // This COS supports two lines.
);
BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
    }
```

Step 2 The service provider's application uses the Prime Cable Provisioning incremental update feature to set SNMP objects on the eMTA/eDVA and thereby enabling the service without disrupting the eMTA/eDVA.

```
// The goal here is to enable a second phone line, assuming one
// phone line is currently enabled.   We will be adding a new
// row to the pktcNcsEndPntConfigTable.

batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION);

// NO_ACTIVATION is the activation mode because we don't want to
// reset the device.
// NO_CONFIRMATION is the confirmation mode because we are
// not attempting to reset the device.
// Create a map containing one element - the list of SNMP
// variables to set on the MTA
```

```
    HashMap<String, Object> map = new HashMap<String, Object>();

    // Create an SnmpVarList to hold SNMP varbinds

    SnmpVarList list = new SnmpVarList();

    // An SnmpVariable represents an oid/value/type triple.
    // pktcNcsEndPntConfigTable is indexed by the IfNumber, which in this case we will
    // assume is interface number 12 (this is the last number in each of the oids below).
    // The first variable represents the creation of a new row in
    // pktcNcsEndPntConfigTable we are setting the RowStatus
    // column (column number 26). The value of 4 indicates that
    // a new row is to be created in the active state.

    SnmpVariable variable = new SnmpVariable( ".1.3.6.1.4.1.4491.2.2.2.1.2.1.1.26.12",
        "4", SnmpType.INTEGER );
    list.add( variable );

    // The next variable represents the call agent id for this new
    // interface, which we'll assume is 'test.com'

    variable = new SnmpVariable( ".1.3.6.1.4.1.4491.2.2.2.1.2.1.1.1.12", "test.com",
        SnmpType.STRING );
    list.add( variable );

    // The final variable represents the call agent port

    variable = new SnmpVariable( ".1.3.6.1.4.1.4491.2.2.2.1.2.1.1.2.12", "2728",
        SnmpType.INTEGER );
    list.add( variable );

    // Add the SNMP variable list to the Map to use in the API call

    map.put( SNMPPropertyKeys.SNMPVAR_LIST, list );

    // Invoke the PCP API to do incremental update on the eMTA.

    batch.performOperation(
        DeviceOperation.INCREMENTAL_UPDATE,      // device operation
        new MACAddress("1,6,00:00:00:00:00:99"), // MAC Address
        map                                      // Parameters for the operation
    );

    // post the batch to RDU server

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 3 The eMTA/eDVA is enabled to use the second telephone line. The eMTA/eDVA continues to receive the same service, after being reset, because the Class of Service was changed in Step 1.

Preprovisioning DOCSIS Modems with Dynamic Configuration Files

A new customer contacts a service provider to order a DOCSIS modem with high-speed *Gold* data service for two sets of CPE behind it.

Desired Outcome

Use this workflow to preprovision a DOCSIS modem with a Class of Service that uses DOCSIS templates. The dynamic configuration file generated from the templates is used while the modem comes online.

Step 1 The service provider chooses a subscriber username and password for the billing system.

Step 2 The service provider chooses Gold Class of Service, and the appropriate DHCP Criteria, and then adds the cable modem to Prime Cable Provisioning.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION);


Map<String, Object> properties = new HashMap<String, Object>();

// Set the property PolicyKeys.COMPUTER_PROMISCUOUS_MODE_ENABLED to enable
// promiscuous mode on modem

properties.put(PolicyKeys.COMPUTER_PROMISCUOUS_MODE_ENABLED, Boolean.TRUE);

// enable promiscuous mode by changing the technology default

batch.changeDefaults(DeviceType.DOCSIS,properties, null);

BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
}
// No CPE DHCP Criteria is specified.
// The CPE behind the modem will use the default provisioned
// promiscuous CPE DHCP criteria specified in the system defaults.
// This custom property corresponds to a macro variable in the
// DOCSIS template for "gold" class of service indicating the
// maximum number of CPE allowed behind this modem. We set it
// to two sets of CPE from this customer.

properties = new HashMap<String, Object>();
```

```
    properties.put("docsis-max-cpes", "2");

    batch = conn.newBatch(

        // No reset

        ActivationMode.NO_ACTIVATION,

        // No need to confirm activation

        ConfirmationMode.NO_CONFIRMATION,

        // No publishing to external database

        PublishingMode.NO_PUBLISHING);

    // To add a DOCSIS modem:

    List<DeviceID> deviceIDList = new ArrayList<DeviceID>();
    deviceIDList.add(new MACAddress("1,6,01:02:03:04:05:06"));
    batch.add(
        DeviceType.DOCSIS,     // deviceType: DOCSIS
        deviceIDList,          // macAddress: scanned from the label
        null,                  // hostName: not used in this example
        null,                  // domainName: not used in this example
        "0123-45-6789",        // ownerID: here, account number from billing system
        "gold",                // classOfService:
        "provisionedCM",       // DHCP Criteria: Network Registrar uses this to
                               // select a modem lease granting provisioned IP address
        properties             // properties:
    );

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 3 The cable modem is shipped to the customer.

Step 4 The customer brings the cable modem online and connects the computers behind it.

Optimistic Locking

An instance of the service provider application needs to ensure that it is not overwriting the changes made by another instance of the same application.

Desired Outcome

Use this workflow to demonstrate the optimistic locking capabilities provided by the Prime Cable Provisioning API.

Note Locking of objects is done in multiuser systems to preserve integrity of changes, so that one person's changes do not accidentally get overwritten by another. With optimistic locking, you write your program assuming that any commit has a chance to fail if at least one of the objects being committed was changed by someone else since you began the transaction.

Step 1 The service representative selects the search option in the service provider's user interface and enters the cable modem's MAC address.

Step 2 Prime Cable Provisioning queries the embedded database, gets the details of the device, and the MSO user interface displays the information.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);


final DeviceID modemMACAddress = DeviceID.getInstance("1,6,00:11:22:33:44:55",
        KeyType.MAC_ADDRESS);
List<DeviceDetailsOption> options = new ArrayList<DeviceDetailsOption>();

options.add(DeviceDetailsOption.INCLUDE_LEASE_INFO);

// MSO admin UI calls the provisioning API to query the details
// for the requested device. Query may be performed based on MAC
// address or IP address, depending on what is known about the
// device.

batch.getDetails(modemMACAddress, options);

// post the batch to RDU server

BatchStatus batchStatus = null;

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
}
```

Step 3 The service representative attempts to change the Class of Service and the DHCP Criteria of the modem using the user interface. This in turn invokes the Prime Cable Provisioning API.

```
Map<String, Object> deviceDetails = new TreeMap((Map<String,
    Object>)batchStatus.getCommandStatus(0).getData());
```

```
    //  extract device detail data from the map

    String deviceType = (String)deviceDetails.get(DeviceDetailsKeys.DEVICE_TYPE);
    String macAddress = (String)deviceDetails.get(DeviceDetailsKeys.MAC_ADDRESS);
    String relayAgentID = (String)deviceDetails.get(DeviceDetailsKeys.RELAY_AGENT_MAC);
    Boolean isProvisioned = (Boolean)deviceDetails.get(DeviceDetailsKeys.IS_PROVISIONED);

    //  Let's save the OID_REVISION_NUMBER property so that we can set it in
    //  step 3.

    String oidRevisionNumber =
        (String)deviceDetails.get(GenericObjectKeys.OID_REVISION_NUMBER);

    // We need a reference to Batch instance so that ensureConsistency()
    // method can be invoked on it.

    batch = conn.newBatch() ;
    List<String> oidList = new ArrayList<String>();

    // Add the oid-rev number saved from step 2 to the list

    oidList.add(oidRevisionNumber);

    // Sends a list of OID revision numbers to validate before processing the
    // batch. This ensures that the objects specified have not been modified
    // since they were last retrieved.

    batch.ensureConsistency(oidList);
    batch.changeClassOfService (
        new MACAddress("1,6,00:11:22:33:44:55"),  // macORFqdn: unique identifier for the
                                                  // device.
        "gold"                                    // newCOSName : Class of service name.
    );

    batch.changeDHCPCriteria (
        new MACAddress("1,6,00:11:22:33:44:55"),  // macORFqdn: unique identifier for the
                                                  // device.
        "specialDHCPCriteria"                     // newDHCPCriteria : New DHCP Criteria.
    );

    // This batch fails with BatchStatusCodes.BATCH_NOT_CONSISTENT,
    // in case if the device is updated by another client in the meantime.
    // If a conflict occurs, then the service provider client
    // is responsible for resolving the conflict by querying the database
    // again and then applying changes appropriately.
    }
}
```

Step 4 The user is ready to receive Gold Class of Service with appropriate DHCP Criteria.

Temporarily Throttling a Subscriber's Bandwidth

An MSO has a service that allows a subscriber to download only 10 MB of data a month. Once the subscriber reaches that limit, their downstream bandwidth is turned down from 10 MB to 56 K. When the month is over they are moved back up to 10 MB.

Note You may want to consider changing upstream bandwidth as well, because peer-to-peer users and users who run websites tend to have heavy upload bandwidth.

Desired Outcome

Use this workflow to move subscribers up and down in bandwidth according to their terms of agreement.

Step 1 The MSO has a rate tracking system, such as *NetFlow*, which keeps track of each customer's usage by MAC address. Initially a customer is provisioned at the *Gold* Class of Service level with 1 MB downstream.

Step 2 When the rate tracking software determines that a subscriber has reached the 10-MB limit it notifies the OSS. The OSS then makes a call into the Prime Cable Provisioning API to change the subscriber's Class of Service from *Gold* to *Gold-throttled*.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);


// AUTOMATIC is the activation mode because we are
// attempting to reset the modem so that it
// receives low bandwidth service.
// NO_CONFIRMATION is the confirmation mode
// because we do not want the batch to fail if we cannot
// reset the modem. If the modem is off, then it will
// be disabled when it is turned back on.
// Let's change the COS of the device so that it restricts
// bandwidth usage of the modem.

batch.changeClassOfService(
    new MACAddress("1,6,00:11:22:33:44:55"),  // macAddress: unique identifier for
                                              // this modem
    "Gold-throttled"                          // newClassOfService: restricts
                                              // bandwidth usage to 56k
);

BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
```

```
            e.printStackTrace();
        }
    }
```

Step 3 At the end of the billing period, the OSS calls the Prime Cable Provisioning API to change the subscriber's Class of Service back to *Gold*.

Preprovisioning CableHome WAN-MAN

A new customer contacts a service provider to order home networking service. The customer expects a provisioned CableHome device.

Desired Outcome

Use this workflow to preprovision a CableHome device so that the cable modem and WAN-MAN components on it will have the appropriate level of service when brought online.

Step 1 The service provider chooses a subscriber username and password for the billing system.

Step 2 The service provider chooses the appropriate Class of Service and the DHCP Criteria for the modem component, then adds it to Prime Cable Provisioning.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation
    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// Let's provision the modem and the WAN-Man component in the same
// batch.
// To add a DOCSIS modem:

List<DeviceID> docisDeviceIDList = new ArrayList<DeviceID>();
docisDeviceIDList.add(new MACAddress("1,6,01:02:03:04:05:06"));
batch.add(
    DeviceType.DOCSIS,      // deviceType: DOCSIS
    docisDeviceIDList,      // macAddress: scanned from the label
    null,                   // hostName: not used in this example
    null,                   // domainName: not used in this example
    "0123-45-6789",         // ownerID: here, account number from billing system
    "Silver",               // classOfService
    "provisionedCM",        // DHCP Criteria: Network Registrar uses this to
                            // select a modem lease granting provisioned IP address
    null                    // properties: not used
);
```

Step 3 The service provider chooses the appropriate Class of Service and DHCP Criteria for the WAN-MAN component, then adds it to Prime Cable Provisioning.

```
    List<DeviceID> wanManDeviceIDList = new ArrayList<DeviceID>();
    wanManDeviceIDList.add(new MACAddress("1,6,01:02:03:04:05:07"));
    batch.add(
        DeviceType.CABLEHOME_WAN_MAN,   // deviceType: CABLEHOME_WAN_MAN
        wanManDeviceIDList,             // macAddress: scanned from the label
        null,                           // hostName: not used in this example
        null,                           // domainName: not used in this example
        "0123-45-6789",                 // ownerID: here, account number from billing
                                        // system
        "silverWanMan",                 // classOfService
        "provisionedWanMan",            // DHCP Criteria: Network Registrar uses this to
                                        // select a modem lease granting provisioned IP
                                        // address
        null                            // properties: not used
    );
}
```

Step 4 The CableHome device is shipped to the customer.

Step 5 The customer brings the CableHome device online.

CableHome with Firewall Configuration

A customer contacts a service provider to order a home networking service with the firewall feature enabled. The customer expects to receive a provisioned CableHome device.

Desired Outcome

Use this workflow to preprovision a CableHome device so that the cable modem and the WAN-MAN components on it have the appropriate level of service when brought online.

Step 1 The service provider chooses a subscriber username and password for the billing system.

Step 2 The service provider chooses the appropriate Class of Service and DHCP Criteria for the cable modem component, then adds it to Prime Cable Provisioning.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// Let's provision the modem and the WAN-Man component in the same
// batch.
// To add a DOCSIS modem:

List<DeviceID> docisDeviceIDList = new ArrayList<DeviceID>();
docisDeviceIDList.add(new MACAddress("1,6,01:02:03:04:05:06"));
```

```
    batch.add(
        DeviceType.DOCSIS,      // deviceType: DOCSIS
        docisDeviceIDList,      // macAddress: scanned from the label
        null,                   // hostName: not used in this example
        null,                   // domainName: not used in this example
        "0123-45-6789",         // ownerID: here, account number from billing system
        "Silver",               // classOfService
        "provisionedCM",        // DHCP Criteria: Network Registrar uses this to
                                // select a modem lease granting provisioned IP address
        null                    // properties: not used
    );
```

Step 3 The service provider chooses the appropriate Class of Service and DHCP Criteria for the WAN-MAN component and adds it to Prime Cable Provisioning.

```
    //      Continuation of the batch in Step 2
    // To add the WAN-Man component:
    // Create a Map to contain WanMan's properties

    Map<String, Object> properties = new HashMap<String, Object>();

    // The fire wall configuration for the Wan Man component is specified
    // using the CableHomeKeys.CABLEHOME_WAN_MAN_FIREWALL_FILE property.
    // This use case assumes that the firewall configuration file named
    // "firewall_file.cfg" is already present in the RDU database and the
    // firewall configuration is enabled in the Wan Man configuration file
    // specified with the corresponding class of service.

    properties.put(CableHomeKeys.CABLEHOME_WAN_MAN_FIREWALL_FILE, "firewall_file.cfg");

    List<DeviceID> wanManDeviceIDList = new ArrayList<DeviceID>();
    wanManDeviceIDList.add(new MACAddress("1,6,01:02:03:04:05:07"));
    batch.add(
        DeviceType.CABLEHOME_WAN_MAN,    // deviceType: CABLEHOME_WAN_MAN
        wanManDeviceIDList,              // macAddress: scanned from the label
        null,                         // hostName: not used in this example
        null,                         // domainName: not used in this example
        "0123-45-6789",               // ownerID: here, account number from billing system
        "silverWanMan",               // classOfService
        "provisionedWanMan",          // DHCP Criteria: Network Registrar uses this to
                                      // select a modem lease granting provisioned IP
                                      // address
        null                          // properties: not used
    );

    BatchStatus batchStatus = null;

    // post the batch to RDU server

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 4 The CableHome device is shipped to the customer.

Step 5 The customer brings the CableHome device online and the cable modem and the WAN-MAN component get provisioned IP addresses and proper configuration files.

Retrieving Device Capabilities for CableHome WAN-MAN

A service provider wants to allow an administrator to view capabilities information for a CableHome WAN-MAN device.

Desired Outcome

The service provider's administrative application displays all known details about a given CableHome WAN-MAN component, including MAC address, lease information, provisioned status, and the device capabilities information.

Step 1 The administrator enters the MAC address of the WAN-MAN being queried into the service provider's user interface.

Step 2 Prime Cable Provisioning queries the embedded database for details of the device identified using the MAC address entered.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");


Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);


final DeviceID modemMACAddress = DeviceID.getInstance("1,6,00:11:22:33:44:55",
        KeyType.MAC_ADDRESS);

List<DeviceDetailsOption> options = new ArrayList<DeviceDetailsOption>();
options.add(DeviceDetailsOption.INCLUDE_LEASE_INFO);

// MSO admin UI calls the provisioning API to query the details
// for the requested device. Query may be performed based on MAC
// address or IP address, depending on what is known about the
// device.

batch.getDetails(modemMACAddress, options);

// post the batch to RDU server

BatchStatus batchStatus = null;
try
{
    batchStatus = batch.post();
}
```

```
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 3 The service provider's application then presents a page of device data details, which can display everything that is known about the requested device. If the device was connected to the service provider's network, this data includes lease information, such as the IP address or the relay agent identifier. This data indicates whether the device is provisioned. If it is provisioned, the data also includes the device type and device capabilities information.

```
    Map<String, Object> deviceDetails = new TreeMap((Map<String,
        Object>)batchStatus.getCommandStatus(0).getData());

    //  extract device detail data from the map

    String deviceType = (String)deviceDetails.get(DeviceDetailsKeys.DEVICE_TYPE);
    String macAddress = (String)deviceDetails.get(DeviceDetailsKeys.MAC_ADDRESS);
    String relayAgentID = (String)deviceDetails.get(DeviceDetailsKeys.RELAY_AGENT_MAC);
    Boolean isProvisioned = (Boolean)deviceDetails.get(DeviceDetailsKeys.IS_PROVISIONED);

    String deviceID = (String) deviceDetails.get(CNRNames.DEVICE_ID.toString());
    String serNum = (String) deviceDetails.get(CNRNames.DEVICE_SERIAL_NUMBER.toString());
    String hwVer = (String)
        deviceDetails.get(CNRNames.HARDWARE_VERSION_NUMBER.toString());
    String swVer = (String)
        deviceDetails.get(CNRNames.SOFTWARE_VERSION_NUMBER.toString());
    String brVer = (String) deviceDetails.get(CNRNames.BOOT_ROM_VERSION.toString());
    String vendorOui = (String) deviceDetails.get(CNRNames.VENDOR_OUI.toString());
    String modelNum = (String) deviceDetails.get(CNRNames.MODEL_NUMBER.toString());
    String vendorNum = (String) deviceDetails.get(CNRNames.VENDOR_NAME.toString());

    //  The admin UI now formats and prints the detail data to a view page
    }
}
```

Self-Provisioning CableHome WAN-MAN

A subscriber has a computer with a browser application installed in a single-dwelling unit and has purchased an embedded CableHome device.

Desired Outcome

Use this workflow to bring a new unprovisioned embedded CableHome device online with the appropriate level of service, and give the subscriber internet access from computers connected to the embedded CableHome device.

Step 1 The subscriber purchases an embedded CableHome device and installs it at home.

Step 2 The subscriber powers on the embedded CableHome device. Prime Cable Provisioning gives the embedded cable modem restricted access, allowing two sets of CPE: one for the CableHome WAN-MAN and the other for the computer.

Note This use case assumes an unprovisioned DOCSIS modem allows two sets of CPE behind it. Until configured to do otherwise, Prime Cable Provisioning supports only a single device behind an unprovisioned DOCSIS modem. You can change this behavior by defining an appropriate Class of Service that supports two sets of CPE and then using it as the default Class of Service for DOCSIS devices.

Step 3 Prime Cable Provisioning configures the CableHome WAN-MAN, including IP connectivity and downloading the default CableHome boot file. The default CableHome boot file configures the CableHome device in passthrough mode. The CableHome device is still unprovisioned.

Step 4 The subscriber connects the computer to the CableHome device. The computer gets an unprovisioned (restricted) IP address. The subscriber starts a browser application on the computer. A spoofing DNS server points the browser to the service provider's registration server (for example, an OSS user interface or a mediator).

Step 5 The subscriber uses the service provider's user interface to complete the steps required for cable modem registration, including selecting a Class of Service. The subscriber also selects a CableHome Class of Service.

Step 6 The service provider's user interface passes the subscriber's information to Prime Cable Provisioning, including the selected Class of Service for cable modem and CableHome, and computer IP address. The subscriber is then registered with Prime Cable Provisioning.

Step 7 The user interface prompts the subscriber to reboot the computer.

Step 8 The provisioning client calls *performOperation(…)* to reboot the modem and gives the modem provisioned access.

```
    // create a new batch

    batch = conn.newBatch(

        // No reset

        ActivationMode.AUTOMATIC,

        // No need to confirm activation

        ConfirmationMode.NO_CONFIRMATION);


    // register performOperation command to the batch

    batch.performOperation(DeviceOperation.RESET, modemMACAddressObject, null);

    // post the batch to RDU server

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 9 When the computer is rebooted, it receives a new IP address from the CableHome device's DHCP server. The cable modem and the CableHome device are now both provisioned. Now the subscriber can connect a number of computers to the Ethernet ports of the CableHome device and they have access to the Internet.

Note If the configuration file supplied to the WAN-MAN component enables the WAN-Data component on the box, it will be provisioned in the promiscuous mode. This assumes that the promiscuous mode is enabled at the technology defaults level for the DeviceType.CABLEHOME_WAN_DATA device type.

RBAC Administration and Operational Access Control

Roles and privileges provide fine grain authorization to manage Prime Cable Provisioning. A privilege represents an authority granted for an operation that can be performed and roles are composed of these privileges.

This section includes these use cases:

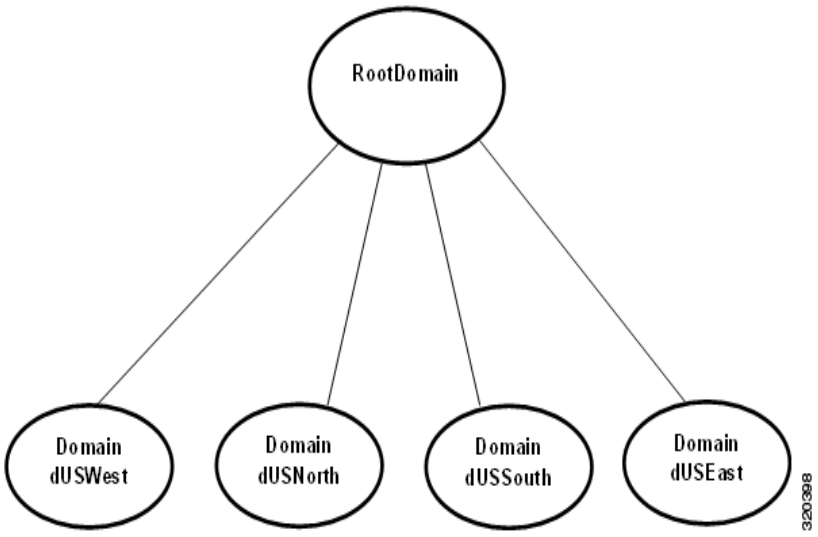
- Role Based Access Control Administration, page 7-59
- Operational Access Control, page 7-63

Role Based Access Control Administration

An administrator wants to use role based access control (RBAC) in the RDU server. The administrator has to first create various configurations.

Figure 7-2 represents the domain hierarchy for RBAC.

Figure 7-2 RBAC Domain Hierarchy



Desired Outcome

Use this work flow to create various configurations required for Role Based Access Control (Domain, Role, User Group, User).

Step 1 Create a domain dUS under the out of the box domain RootDomain. RootDomain is the top most domain defined in the RDU server.

```
//Create a new connection
PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");
Batch batch = connection.newBatch();
//Create a new domain dUS under RootDomain
bsatch.addDomain("dUS", "US", DomainTypeValues.ROOT_DOMAIN, null);
BatchStatus bStatus = null;
//Post the batch to RDU Server
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 2 Create two other domains dUSEast and dUSWest with dUS as their parent.

```
//Create a new batch

batch = connection.newBatch();

//Create two child domains dUSEast and dUSWest under the parent domain dUS

batch.addDomain("dUSEast", "US East", "dUS", null);
batch.addDomain("dUSWest", "US West", "dUS", null);
bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 3 Create a role rAllDevice with all device related privileges associated with it. Also, associate different properties which can be modified by a user with this role.

```
//Create a new batch

batch = connection.newBatch();

//Add privileges to the role

List <String> allDevicePrivileges=new ArrayList<String>();
allDevicePrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_CREATE);
allDevicePrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_DELETE);
allDevicePrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_OPERATION);
allDevicePrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_READ);
allDevicePrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_REGEN);
allDevicePrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_UPDATE);

//Add properties to the role
```

```
Map<String, Object> properties = new HashMap<String, Object>();

//Add device related properties which can be administered by the role.

List<String> devicesProperties = new ArrayList<String>();

devicesProperties.add(FqdnKeys.AUTO_FQDN_ENABLE);

devicesProperties.add(FqdnKeys.AUTO_FQDN_DOMAIN);

properties.put(RoleDetailsKeys.MODIFIABLE_DEVICE_PROPERTIES,
     devicesProperties);

//Create the role

batch.addRole("rAllDevice", "Role with all device privileges",
    allDevicePrivileges, properties);

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 4 Create a new user group gDeviceAdmin and associate it with the newly created role rAllDevice and other out of the box roles (for example, File Admin Role) defined in the RDU server.

```
//Create a new batch

batch = connection.newBatch();

List<String> roleList = new ArrayList<String>();

//Associate the role "rAllDevice" to the user group

roleList.add("rAllDevice");

//Associate the out of the box roles already defined in the system to the user group

roleList.add(RoleTypeValues.FILE_ADMIN_ROLE);

//add the user group

batch.addUserGroup("gDeviceAdmin", "Device admin user group", roleList);

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 5 Create another user group gCosDhcpAdmin which is associated with the out of the box roles defined in the RDU server.

```
//Create a new batch
```

```
batch = connection.newBatch();

roleList = new ArrayList<String>();

//Associate the out of the box roles already defined in the RDU Server

roleList.add(RoleTypeValues.COS_ADMIN_ROLE);
roleList.add(RoleTypeValues.DHCP_ADMIN_ROLE);

//Add a user group

batch.addUserGroup("gCosDhcpAdmin", "COS and DHCP Criteria admin user group", roleList);

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 6 Create a new user uOperator and assign roles, user groups and domains to the user.

```
//Create a new batch

batch = connection.newBatch();

Map<String, Object> userProperties = new HashMap<String, Object>();

//Associate the role rAllDevice as well as other out of the box roles with the user

List<String> userRoleList = new ArrayList<String>();
userRoleList.add("rAllDevice");
userRoleList.add(RoleTypeValues.FILE_ADMIN_ROLE);

//Associate the user group gCosDhcpAdmin with the user

List<String> userGroupList = new ArrayList<String>();
userGroupList.add("gCosDhcpAdmin");

//Associate the user with the domains dUSEast and dUSWest.

List<String> userDomainList = new ArrayList<String>();
userDomainList.add("dUSEast");
userDomainList.add("dUSWest");

userProperties.put(UserDetailsKeys.ROLES_ASSIGNED, userRoleList);
userProperties.put(UserDetailsKeys.GROUPS_ASSIGNED, userGroupList);
userProperties.put(UserDetailsKeys.DOMAINS_ASSIGNED, userDomainList);

//Add the user

batch.addUser("uOperator", "changeit", userProperties);

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
```

```
{
    System.exit(1);
}
```

Step 7 Create a mapping between a user group defined in an external AAA server and a user group defined in the RDU server.

```
//Create a new batch

batch = connection.newBatch();

Map<String, Object> rduDefaultsProperties =
    new HashMap<String, Object>();

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

//Map the user group gDeviceAdmin defined in RDU to the user group externalDeviceAdmin
//defined in external AAA server

externalToInternalUserGroupMapping.put("externalDeviceAdmin",  "gDeviceAdmin");

//Map the user group gCosDhcpAdmin defined in RDU to the user group externalCosDhcpAdmin
//defined in external AAA server

externalToInternalUserGroupMapping.put("externalCosDhcpAdmin", "gCosDhcpAdmin");

rduDefaultsProperties.put(UserAuthenticationKeys.USER_GROUP_MAPPING,
    externalToInternalUserGroupMapping);

//Save the mapping information at the RDU Defaults property level
batch.changeRDUDefaults(rduDefaultsProperties, null);

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Operational Access Control

A service provider has created configurations for administering Role Based Access Control. The service provider wants to use the operational level access control.

Desired Outcome

The operator should not be able to perform an operation unless the appropriate privileges necessary for the operation are assigned to the operator.

Step 1 Create a user uCoSAdmin with class of service admin role.

```
//Create a new connection

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

//Create a new batch
```

```
Batch batch = connection.newBatch();

//Create a user with class of service admin role

Map<String, Object> userProperties = new HashMap<String, Object>();

//Add Class of Service Admin Role to the user's property

List<String> userRoleList = new ArrayList<String>();
userRoleList.add(RoleTypeValues.COS_ADMIN_ROLE);

userProperties.put(UserDetailsKeys.ROLES_ASSIGNED, userRoleList);

//Add the user

batch.addUser("uCoSAdmin", "changeit", userProperties);

BatchStatus bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
   System.exit(1);
}
```

Step 2 Since the user uCoSAdmin has class of service admin privilege, the user is able to create a class of service successfully.

```
//Create a new PACE Connection with the user uCoSAdmin

PACEConnection connectionUCoSAdmin = PACEConnectionFactory.getInstance(
    "localhost", 49187, "uCoSAdmin", "changeit");

//Create a new batch

batch = connectionUCoSAdmin.newBatch();

//Create a class of service

Map<String, Object> cosProperties = new HashMap<String, Object>();
cosProperties.put(ClassOfServiceKeys.COS_DOCSIS_FILE, "gold.cm");
cosProperties.put(ClassOfServiceKeys.COS_ASSIGNED_DOMAIN,
    DomainTypeValues.ROOT_DOMAIN);

batch.addClassOfService(DeviceType.DOCSIS, "sampleCoS", cosProperties);

//The operation should be successful

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 3 The user uCoSAdmin tries to set the newly created class of service sampleCoS as the default class of service for DOCSIS technology. But, since the user does not have necessary privilege(s), the operation fails with command status CMD_ERROR_USER_DOES_NOT_HAVE_PRIVILEGE.

```
//User uCoSAdmin creates a new batch

batch = connectionUCoSAdmin.newBatch();

//Try to set sampleCoS  as the default class of service for DOCSIS

Map<String, Object> docsisDefaultsProperties = new HashMap<String, Object>();
docsisDefaultsProperties.put(TechnologyDefaultsKeys.DEFAULT_CLASS_OF_SERVICE,
"sampleCoS");

batch.changeDefaults(DeviceType.DOCSIS, docsisDefaultsProperties, null);

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}

CommandStatusCode commandStatus = (CommandStatusCode)
bStatus.getCommandStatus(0).getStatusCode();

//Since the user does not have privilege for changing the technology defaults, the
operation fails

if (bStatus.isError() &&
CommandStatusCode.CMD_ERROR_USER_DOES_NOT_HAVE_PRIVILEGE.equals(commandStatus))  {
    // Batch error occurred.

    System.out.println("Failed to change the Docsis technology defaults with the user
sampleCoS"
        + "since the user does not have sufficient privilges");
    System.exit(1);
}
```

Update Protection for Properties at Device Level

A service provider wants to control its operator from updating properties of devices.

Desired Outcome

A role can be associated with a set of device related properties (including custom properties). An operator assigned with that role, is able to administer only those properties to a device.

Step 1 Create a custom property at the RDU Server. This property can be assigned to a device.

```
//Create a PACE Connection

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

//Create a new batch
```

```
Batch batch = connection.newBatch();

//Add a custom property to the RDU Server

batch.addCustomPropertyDefinition("/role/owner", DataType.STRING);

BatchStatus bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    //System.exit(1);
}
```

Step 2 Create a role rDevAdmin with different device related properties and custom properties. An operator with the role is able to administer only those properties to a device.

```
//Create a new batch

batch = connection.newBatch();

//Assign all the device related privileges to the role

List <String> allPrivileges = new ArrayList<String>();

allPrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_CREATE);
allPrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_DELETE);
allPrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_OPERATION);
allPrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_READ);
allPrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_REGEN);
allPrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_UPDATE);

Map<String, Object> properties = new HashMap<String, Object>();

List<String> devicesProperties = new ArrayList<String>();

//Add device related properties to the role

//Add Class Of Service as a modifiable property

devicesProperties.add(DeviceDetailsKeys.CLASS_OF_SERVICE);

//Add the following properties as modifiable properties

devicesProperties.add(FqdnKeys.AUTO_FQDN_PREFIX);
devicesProperties.add(FqdnKeys.AUTO_FQDN_SUFFIX);
devicesProperties.add(FqdnKeys.AUTO_FQDN_ENABLE);
devicesProperties.add(FqdnKeys.AUTO_FQDN_DOMAIN);

//Add the custom property "/role/owner" as a modifiable property
devicesProperties.add("/role/owner");

//Add modifiable device related properties to the role

properties.put(RoleDetailsKeys.MODIFIABLE_DEVICE_PROPERTIES,
     devicesProperties);

//Add the role

batch.addRole("rDevAdmin", "Role with all device privileges",
    allPrivileges, properties);
```

```
bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    //System.exit(1);
}
```

Step 3 Create an operator with rDevAdmin role.

```
//Create a new batch

batch = connection.newBatch();

Map<String, Object> userProperties = new HashMap<String, Object>();

//Associate the operator with the role rDevAdmin

List<String> userRoleList = new ArrayList<String>();
userRoleList.add("rDevAdmin");

userProperties.put(UserDetailsKeys.ROLES_ASSIGNED, userRoleList);

//Create the user

batch.addUser("uDeviceAdminOperator", "changeit", userProperties);

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    //System.exit(1);
}
```

Step 4 Operator uDeviceAdminOperator tries to add a device with few properties all of which are associated with its role as modifiable device related properties.

The device is added successfully.

```
PACEConnection connectionUser = PACEConnectionFactory.getInstance(
    "localhost", 49187, "uDeviceAdminOperator", "changeit");

batch = connectionUser.newBatch();

DeviceID modemMacAddress = DeviceID.getInstance("1,6,00:11:00:00:00:92",
KeyType.MAC_ADDRESS);
List<DeviceID> deviceIDs = new ArrayList<DeviceID>();
deviceIDs.add(modemMacAddress);

Map<String, Object> prop = new HashMap<String, Object>();
prop.put(FqdnKeys.AUTO_FQDN_DOMAIN, "false");
prop.put("/role/owner", "abcd");

batch.add(DeviceType.DOCSIS, deviceIDs, null, null, null, "sample-bronze-docsis", null,
prop);

bStatus = null;
try
```

```
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 5 Operator uDeviceAdminOperator tries to modify the DHCP Criteria of the device.

The operation fails as this role does not have DHCP Criteria as modifiable property of a device.

```
//Create a new batch

batch = connectionUser.newBatch();

//Change the DHCP Criteria of the device

batch.changeDHCPCriteria(modemMacAddress, "sample-provisioned");

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}

CommandStatus commandStatus = bStatus.getCommandStatus(0);
StatusCode statusCode = commandStatus.getStatusCode();

System.out.println(bStatus);

// Batch error occurred.

if (bStatus.isError())  {
    System.out.println("Failed to modify the DHCP Criteria of the device with MAC Address
" + modemMacAddress);

//The operation should fail with CMD_ERROR_USER_CAN_NOT_UPDATE_PROPERTY command
//status code.

    if (CommandStatusCode. CMD_ERROR_USER_CAN_NOT_UPDATE_PROPERTY.equals(statusCode)) {
        System.out.println("The user uDeviceAdminOperator can't update the DHCP Criteria
of a device");
    }
    System.exit(1);
}
```

Domain Administration and Instance Level Access Control

While operational level access control defines what actions a user can perform, instance level access control determines whether or not those actions can be performance on a specific instances. Domain and instance level authorization are independent of roles and privileges.

This section includes these use cases:

- Assigning Existing Configuration Objects to a Domain, page 7-69

- Instance Level Authorization, page 7-71

Assigning Existing Configuration Objects to a Domain

An administrator wants to assign the existing resources such as devices, Classes of Service, DHCP Criterion, files, DPEs, CPNRs, and Provisioning Groups to a domain.

Desired Outcome

The resources are being assigned to the specified domain. The service provider's administrative application displays the changed domain information for each of the resources.

1. **Use Case 1**: Assign the existing classes of service sample-bronze-docsis, sample-gold-docsis, and unprovisioned-stb to the domain dUS.

```
//Create a new connection

PACEConnection connection = PACEConnectionFactory.getInstance("localhost", 49187, "admin",
"changeit");

//Create a new batch

Batch batch = connection.newBatch();

//List of classes of service to be assigned to the domain dUS

List<String> cosList = new ArrayList<String>();
cosList.add("sample-bronze-docsis");
cosList.add("sample-gold-docsis");
cosList.add("unprovisioned-stb");

//Assign the classes of service to the domain

batch.changeDomainProperties("dUS", null, cosList, ResourceType.CLASS_OF_SERVICE);

BatchStatus bStatus = null;

//Post the batch to RDU Server
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

2. **Use Case 2:** Assign the existing DHCP criterion sample-provisioned and unprovisioned-cablehome-wan-data to the domain dUSEast.

```
//Create a new batch

batch = connection.newBatch();

//List of DHCP Criterion to be assigned to the domain dUSEast

List<String> dhcpList = new ArrayList<String>();
dhcpList.add("sample-provisioned");
dhcpList.add("unprovisioned-cablehome-wan-data");

// Assign the DHCP Criterion to the domain

batch.changeDomainProperties("dUSEast", null, dhcpList, ResourceType.DHCP_CRITERIA);
```

```
bStatus = null;

//Post the batch to RDU Server

try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

3. **Use Case 3:** Assign the existing files bronze.cm, gold.cm, unprov.cm to the domain dUSWest.

```
//Create a new batch

batch = connection.newBatch();

//List of files to be assigned to the domain dUSWest

List<String> fileList = new ArrayList<String>();
fileList.add("bronze.cm");
fileList.add("gold.cm");
fileList.add("unprov.cm");

// Assign the files to the domain

batch.changeDomainProperties("dUSWest", null, fileList, ResourceType.FILE);

bStatus = null;

//Post the batch to RDU Server

try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

4. **Use Case 4**: Assign the devices already existing in the RDU server to the domain dUSWest.

```
//Create a new batch

batch = connection.newBatch();

// List of devices to be assigned to the domain dUSWest

List<DeviceID> deviceList = new ArrayList<DeviceID>();
deviceList.add(DeviceID.getInstance("1,6,00:11:00:00:00:01", KeyType.MAC_ADDRESS));
deviceList.add(DeviceID.getInstance("1,6,00:11:00:00:00:02", KeyType.MAC_ADDRESS));
deviceList.add(DeviceID.getInstance("03:00:00:01:00:00", KeyType.DUID));

// Assign the devices to the domain

batch.changeDeviceDomainProperties("dUSWest", null, deviceList);

bStatus = null;

//Post the batch to RDU Server
```

```
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

5. **Use Case 5**: Assign a DPE already in ready state to the domain dUS.

```
//Create a new batch

batch = connection.newBatch();

// List of DPEs to be assigned to the domain dUS

List<String> dpeList = new ArrayList<String>();
dpeList.add("cpc-rhel-test.cisco.com");

//Assign the DPEs to the domains

batch.changeDomainProperties("dUS", null, dpeList, ResourceType.DPE);

bStatus = null;

//Post the batch to RDU Server

try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Instance Level Authorization

A service provider has created configurations for administering Role Based Access Control. The service provider wants to use the instance level authorization for various resources such as Device, Class of service, DHCP Criteria, File, DPE, CPNR, and Provisioning Group.

Desired Outcome

Only those operators who have access to a specific resource must be allowed to operate on it.

Step 1 Enable instance level authorization feature at the RDU Server.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

//Create a new batch

Batch batch = connection.newBatch();

//Enable instance level authorization at the RDU Server.

Map<String, Object> properties = new HashMap<String, Object>();
```

```
properties.put(ServerDefaultsKeys.INSTANCE_LEVEL_AUTH_ENABLE, true);
batch.changeRDUDefaults(properties, null);

BatchStatus bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 2 Create a user uDUS with administrator role and associate the user to the domain dUS and not to the domain RootDomain.

Step 3 Create a class of service cosDUSEast associated with the domain dUSEast and another class of service cosRootDomain associated with the domain RootDomain.

Since instance level authorization is enabled, the user uDUS is be able to access only the resources associated to the domain dUS and its children (dUSEast, dUSWest). But, the user is not authorized to access resources associated to RootDomain.

```
//Create a batch

batch = connection.newBatch();

Map<String, Object> userProperties = new HashMap<String, Object>();

//Assign the administrator role to the user

List<String> userRoleList = new ArrayList<String>();
userRoleList.add(RoleTypeValues.SUPER_ADMIN_ROLE);

//Associate the user to the domain dUS

List<String> userDomainList = new ArrayList<String>();
userDomainList.add("dUS");

userProperties.put(UserDetailsKeys.ROLES_ASSIGNED, userRoleList);
userProperties.put(UserDetailsKeys.DOMAINS_ASSIGNED, userDomainList);

//add the user

batch.addUser("uDUS", "changeit", userProperties);

//Create a class of service and associate it to the domain dUSEast

Map<String, Object> cosProperties = new HashMap<String, Object>();
cosProperties.put(ClassOfServiceKeys.COS_ASSIGNED_DOMAIN, "dUSEast");
cosProperties.put(ClassOfServiceKeys.COS_DOCSIS_FILE, "gold.cm");

batch.addClassOfService(DeviceType.DOCSIS, "cosDUSEast", cosProperties);

//Create another class of service and associate it to the domain RootDomain.

cosProperties = new HashMap<String, Object>();
cosProperties.put(ClassOfServiceKeys.COS_ASSIGNED_DOMAIN,
DomainTypeValues.ROOT_DOMAIN);
cosProperties.put(ClassOfServiceKeys.COS_DOCSIS_FILE, "gold.cm");

batch.addClassOfService(DeviceType.DOCSIS, "cosRootDomain", cosProperties);
```

```
bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 4 The user uDUS tries to get the details of the class of service cosDUSEast belonging to the domain dUSEast. Since the user has access to the domain dUS and its children (dUSEast, dUSWest), user can successfully get the class of service details.

```
//Create a PACEConnection by the user uDUS

PACEConnection connectionByDUS = PACEConnectionFactory.getInstance(
    "localhost", 49187, "uDUS", "changeit");

//Create a new batch

batch = connectionByDUS.newBatch();

//User uDUS tries to get the details of class of service cosDUSEast

batch.getClassOfServiceProperties("cosDUSEast");

bStatus = null;

//The batch should successfully return the class of service details.

try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 5 The user uDUS tries to get the details of the class of service cosRootDomain which is associated with the domain RootDomain. Since the user does not have access to RootDomain, the user is not authorized to get details of the class of service.

```
//Create a new batch from the PACEConnection connectionByDUS

batch = connectionByDUS.newBatch();

//User uDUS tries to get the details of class of service cosRootDomain

batch.getClassOfServiceProperties("cosRootDomain");

bStatus = null;

//Post the batch

try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
```

```
    System.exit(1);
}

CommandStatus commandStatus = bStatus.getCommandStatus(0);
StatusCode statusCode = commandStatus.getStatusCode();

//The batch should fail with Command Status code CMD_ERROR_USER_NOT_AUTHORIZED
//as the user uDUS does not have access to the class of service cosRootDomain

if (bStatus.isError() &&
    CommandStatusCode.CMD_ERROR_USER_NOT_AUTHORIZED.equals(statusCode))  {
    // Batch error occurred.
System.out.println("User uDUS failed to get the details of the class of service
cosRootDomain");
    System.exit(1);
}
```

CRS Management

Prime Cable Provisioning provides greater control on CRS, such as you can enable, disable, pause, and resume CRS without restarting RDU. You can also view, filter, and delete any request queued by CRS.

Desired Outcome

The operator must be able to control CRS without restating RDU.

Step 1 Enable CRS.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
final Map<String, Object> map = new HashMap<String, Object>();
//Enabling CRS using SERVER_CRS_ENABLE
map.put(ServerDefaultsKeys.SERVER_CRS_ENABLE, Boolean.TRUE);
batch.changeRDUDefaults(map, null);
BatchStatus bStatus = null;
//Post the batch to RDU Server
try
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```

Step 2 Change CoS to add CRS request to the queue.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
```

```
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
Map propAdd1 = new HashMap();
propAdd1.put(ClassOfServiceKeys.COS_DOCSIS_FILE, "gold.cm");
//Adding request to the CRS queue
batch.changeClassOfServiceProperties("bac1", propAdd1, null);
BatchStatus bStatus = null;
//Post the batch to RDU Server
try
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```

Step 3 Fetch CRS Statistics using getRDUDetails() API.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
//Fetch CRS Statistics using getRDUDetails
batch.getRDUDetails("localhost");
BatchStatus bStatus = null;
//Post the batch to RDU Server
try
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```

Step 4 Get details of all the requests in the queue.

```
private static SearchBookmark printRecordSearchResults(RecordSearchResults rs)
    throws Exception
    {
        SearchBookmark sb = rs.getSearchBookmark();
        List<RecordData> rdlist = rs.getRecordData();
        Iterator<RecordData> iter = rdlist.iterator();
```

```
        while (iter.hasNext())
        {
            RecordData rdObj = iter.next();
            Key keyObj = rdObj.getPrimaryKey();
           System.out.println("CrsRequestId: " + ((CrsRequestId)keyObj).getRequestId());
            Map crsDetails = rdObj.getDetails();
        }
        return sb;
    }

    public static void getAllCRSRequestsQueuedBasedOnDHCPCriteria() throws Exception
    {
        ResourceNamePattern pattern = new ResourceNamePattern("*",
CRSRequestType.DHCP_CRITERIA);
        CRSSearchType cst = new CRSSearchType(ReturnParameters.ALL);
        RecordSearchResults rs = null;
        SearchBookmark sb = null;

        rs = searchCRSRequest(cst, sb);
        sb = rs.getSearchBookmark();

        while (sb != null)
        {
            // print out the data in the record search result.
            sb = printRecordSearchResults(rs);
            // call the search routine again
            rs = searchCRSRequest(cst, sb);
        }
    }

   private static RecordSearchResults searchCRSRequest(CRSSearchType cst, SearchBookmark
sb) throws Exception
    {
        RecordSearchResults rs = null;
        PACEConnection s_conn = null;
        final Batch batch = s_conn.newBatch();
        final int numberOfRecordReturn = 10;
```

```
        //calling the search API
        batch.searchCRSRequest(cst, sb, numberOfRecordReturn);

        // Call the RDU.
        BatchStatus batchStatus = batch.post();

        // Check for success.
        CommandStatus commandStatus = null;
        if (0 < batchStatus.getCommandCount())
        {
            commandStatus = batchStatus.getCommandStatus(0);
        }
        //check to see if there is an error
        if (batchStatus.isError()
                || batchStatus.isWarning()
                || commandStatus == null
                || commandStatus.isError())
        {
            System.out.println("report batch error.");
            return null;
        }

        //batch success without error, retrieve the result
        //this is a list of CRS Requests
        rs = (RecordSearchResults)commandStatus.getData();
        return rs;
    }
```

Step 5 Pause CRS. When CRS is paused, execution of CRS requests is paused.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
final Map<String, Object> map = new HashMap<String, Object>();
//Pause CRS using SERVER_CRS_PAUSED
map.put(ServerDefaultsKeys.SERVER_CRS_PAUSED,  Boolean.TRUE);
batch.changeRDUDefaults(map, null);
BatchStatus bStatus = null;
//Post the batch to RDU Server
```

```
try
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```

Step 6 Delete the CRS requests.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
//Delete CRS using deleteCRSRequests
List crsRequestIdList = new ArrayList();
crsRequestIdList.add("325f44454641554c545f434c4153535f4f465f534552564943455f444f4353495
34d4f44454d3030");
  crsRequestIdList.add("3273616d706c652d676f6c642d646f637369733230");
batch.deleteCRSRequests(crsRequestIdList);


BatchStatus bStatus = null;
//Post the batch to RDU Server
try
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```

Step 7 Resume CRS. When CRS is resumed, the execution of CRS requests is resumed.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
final Map<String, Object> map = new HashMap<String, Object>();
//Resume CRS using SERVER_CRS_PAUSED
map.put(ServerDefaultsKeys.SERVER_CRS_PAUSED,  Boolean.FALSE);
batch.changeRDUDefaults(map, null);
BatchStatus bStatus = null;
//Post the batch to RDU Server
try
```

```
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```

Step 8 Disable CRS. When CRS is disabled the entire CRS service is stopped and existing requests in the queue are cleared automatically.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
final Map<String, Object> map = new HashMap<String, Object>();
//Disable CRS using SERVER_CRS_ENABLE
map.put(ServerDefaultsKeys.SERVER_CRS_ENABLE, Boolean.FALSE);
batch.changeRDUDefaults(map, null);
BatchStatus bStatus = null;
//Post the batch to RDU Server
try
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```

Step 9 Set the policies for CRS pause on exceeding failure threshold.

When SERVER_CRS_PAUSE_ON_FAILURE_THRESHOLD is set to true and once the percentage of failed devices exceeds SERVER_CRS_FAILURE_THRESHOLD_PERCENTAGE, CRS automatically pauses. By default SERVER_CRS_PAUSE_ON_FAILURE_THRESHOLD is set to false.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
final Map<String, Object> map = new HashMap<String, Object>();
//Set the policies for CRS using SERVER_CRS_PAUSE_ON_FAILURE_THRESHOLD and
SERVER_CRS_FAILURE_THRESHOLD_PERCENTAGE
map.put(ServerDefaultsKeys.SERVER_CRS_PAUSE_ON_FAILURE_THRESHOLD, Boolean.FALSE);
map.put(ServerDefaultsKeys.SERVER_CRS_FAILURE_THRESHOLD_PERCENTAGE,
Float.parseFloat("10"));
batch.changeRDUDefaults(map, null);
```

```
BatchStatus bStatus = null;
//Post the batch to RDU Server
try
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```

Provisioning Web Services (PWS)

This chapter provides an overview of PWS, and describes the PWS data types, PWS operations, and some sample use cases of PWS usage in Prime Cable Provisioning.

Overview

The Provisioning Web Services (PWS) component provides a RESTful based web interface that supports provisioning operation. The provisioning services include functionalities such as: adding, retrieving, updating, and removing objects necessary to support the provisioning and configuration generation of CPEs. The PWS objects include devices, classes of service, DHCP criteria, groups, and files.

The PWS component manages the following activities:

- Exposes a provisioning web service that provides functionality similar to the current API client.
- Supports stateless interactions.
- Supports both synchronous and asynchronous requests.
- Supports singular and plural operations.
- Supports both stop on failure and ignore on failure.
- Supports request that can operate on multiple objects.
- Supports WS-I Basic Profile v1.1.
- Supports both HTTP and HTTPS transport.
- Supports RESTful.
- Supports Swagger.

The web service is hosted on a Tomcat container, and it is recommended that you install it on a separate server.

For more information on PWS, see the *Cisco Prime Cable Provisioning 7.4 User Guide*.

PWS Concepts

The PWS component provides the following functionalities for running an operation on the device:

Session Management

The session management functionality enables you to establish a session between the WS client and the RDU to process multiple requests from WS client. A WS client can establish connection with multiple RDUs simultaneously. The client runs a **createSession** operation, and in response receives a context object on successful authentication. The context object identifies the session between the client and the RDU, and can be used as an authentication token to run all the requests in the session.

The client closes the session after processing all the requests. The session also gets closed automatically after a configured period of idle time. The default idle timeout is configured using a command line script for all the sessions. If the session gets closed and the client tries to reestablish the session, the session gets reestablished with the same session id.

Transactionality

The PWS operations can be classified into two categories:

- Single device operations - One operation for a single device. For single device operations, an operation is a single transaction in which all the changes are made or no change is made to the device.
- Multiple device operations - One operation for multiple devices. For multiple device operations, the transaction scope can be set to include all devices, or have each device change in its individual transaction.

You can set the execution option, transactionPerItem, to true to retrieve the transaction data for each batch. The OperationStatus returned will contain the status of the entire operation as well as the individual status if multiple transactions are used.

PWS Data Types

The following data types are supported by PWS:

- DeviceType - Attribute that defines the type of the device. For example, Computer; DOCSISModem; PacketCableMTA; CableHomeWanMan; CableHomeWanData; STB; eRouter; RPD.
- ClassOfService (COS) - Attribute that defines the class of service.
- DHCPCriteria - Attribute that defines the DHCP criteria.
- File - Attribute the defines the static configuration file, groovy scripts or dynamic configuration files, jar files, MIB files, template files, firmware image, and other generic files.
- Group - Attribute that defines a group. The devices are assigned to a group based on a specific logical criterion.
- Context - Attribute that defines the authentication data for a client or a client's session.
- Options - Attribute that helps you to customize the WS client’s request either at the PWS level or the RDU level. The Options data type is categorized into the following two types:

– Execution Options - Used to customize the execution process of the WS client's request. For example, you can run the client's request in reliable mode. In case of reliable mode, when the RDU receives a reliable request, the RDU persists the batch until it executes successfully. If the RDU restarts or the WS client restarts, a record of the Batch is retained. The RDU stores the last 1000 reliable batch responses.

Table 8-1 describes the execution options for PWS client's request.

Table 8-1 Execution Options for PWS Client's Request

Execution Options	Value
ActivationMode	AUTOMATIC- This mode specifies that the RDU will perform all the necessary steps to activate the devices in a batch. The batch processing in this mode involves performing batch writes to the database, generating a new device, configuration of the device the batch affects, downloading the new configuration to the appropriate DPE(s), and disrupting the device NO_ACTIVATION - Disable the AUTOMATIC activation mode.
ConfirmationMode	NO_CONFIRMATION - This mode specifies that the RDU will allow the batch to proceed, even if there is an error or warning during device disruption. CUSTOM_CONFIRMATION - This mode is currently ignored by the RDU and is equivalent to NO_CONFIRMATION.
PublishingMode	NO_PUBLISHING - This mode specifies that the RDU will not publish changes from this batch. PUBLISHING_CONFIRMATION - This mode specifies that the RDU will publish the changes from this batch, but if the publishing fails, the RDU will fail the batch. RDU will roll back all the publishing changes and any changes made by the batch. PUBLISHING_NO_CONFIRMATION- This mode specifies the RDU will publish the changes from this batch, but if publishing fails, the RDU will not fail the batch. RDU will roll back the publishing for this batch.
asynchronous	true - Activate asynchronous mode. The status of the web service operation will represent whether the RDU interaction was successful false - Deactivate asynchronous mode. The web service operation is blocked until the RDU interaction completes. The default value is false.
reliableMode	true - Activate reliable mode. false - Deactivate reliable mode. The default value is false.
timeout	Used to specify the timeout value in milliseconds.

Table 8-1 Execution Options for PWS Client's Request (continued)

Execution Options	Value
stopOnFailure	true - RDU will not execute the consecutive batch if the current batch processing fails. false - RDU will run all the batches even if any batch fails. The default value is true.
transactionPerItem	true - RDU will return the result of each individual transaction. false - No individual transaction result is available. The default value is false.

  - Operation Options - Used to customize the operation process of WS client's request. For example, to retrieve the lease data from a getDevice() request, add the flag **includeLeaseInfo** in the getDevice() request and set its value to **True**. The individual properties of the lease data are retrieved in the response. If you want to delete the devices behind the managed device, add the flag **deletedevicesbehind** in the deleteDevice() or deleteDevices() request and set its value to **True**.

- DeviceId - Attribute that allocates a unique identity to the device. The supported unique identities are types of device ids such as MACAddressType, DUIDType and FQDNType.
- Properties - Attribute that defines the provisioning criteria for the devices. The provisioning criteria is defined in terms of key-value list, where the key and value are both strings.
- SearchResult - Attribute that is used to retrieve the search results.
- Search - Attribute used to search devices, files, and groups based on a specific logical criterion. The QueryType attribute is used to define the logical criterion for the search operation. Table 8-2 describes the supported logical criteria for the search operation.

Table 8-2 QueryType for Search Operation

QueryType	Description
DeviceSearchByCOS	Used to search devices that are associated with a specific class of service.
DeviceSearchByDHCPCriteria	Used to search devices that are associated with a specific DHCP criteria.
DeviceSearchByDefaultCOS	Used to search devices that are associated with the default class of service of a specific device type.
DeviceSearchByDefaultDHCP Criteria	Used to search devices that are associated with the default DHCP criteria of a specific device type.
DeviceSearchByDeviceType	Used to search devices that are associated with a specific device type.
DeviceSearchByGroupName	Used to search devices that are associated with a specific group.
DeviceSearchByProvGroupNa me	Used to search devices that are associated with a specific provisioning group
DeviceSearchByDeviceIdPatte rn	Used to search devices that are associated with a specific device identifier pattern. For example, you can use this attribute to search all the devices whose MAC address starts with "1,6,".

Table 8-2 QueryType for Search Operation (continued)

QueryType	Description
DeviceSearchByOwnerId	Used to search devices that are associated with a specific owner identifier. For a search request with this QueryType, no paging support is available from RDU and all the devices associated with this specific owner id are retrieved in a single instance.
FileSearchByFileNamePattern	Used to search files that are associated with a specific file name pattern.
FileSearchByFileType	Used to search files that are associated with a specific file type.
GroupSearchByGroupType	Used to search groups that are associated with a specific group type.
GroupSearchByRelatedGroup	Used to search groups that are related to specific group.
GroupSearchByGroupNamePattern	Used to search groups that are associated with a specific group name pattern.
CosSearchByDeviceType	Used to search class of services that are associated with a particular device type. For a search request with this QueryType, no paging support is available from RDU and all the class of services associated with this specific DeviceType are retrieved in a single instance.
DHCPCriteriaSearch	Used to search the DHCP Criteria. For a search request with this QueryType, no paging support is available from RDU and all DHCP criteria are retrieved in a single instance.

- PropertyFilter - Attribute that specifies the properties that you want to include in the response. If the PropertyFilter data type is set, only the listed properties will be returned.
- OperationStatus - Attribute that displays the response of a web service operation.
- DeviceOperationStatus - Attribute that displays the response of a web service operation on devices that involves returning operation result for multiple devices. In addition to the attributes of OperationStatus data type, this object also includes a device object. On success a device object with a SUCCESS code is returned. On failure, the operation status is returned with the description about the issue/failure.
- DevicesBehindOperationStatus - Attribute that displays the response of a web service operation on list of devices behind the specified devices.
- DeviceTypeOperationStatus - Attribute that displays the response of a web service operation on a specific device type, for example; Computer, DOCSISModem, PacketCableMTA, CableHomeWanMan, CableHomeWanData, STB, and eRouter. In addition to the attributes of OperationStatus data type, this object also includes a DeviceType object. On success, a DeviceType object with a SUCCESS code is returned. On failure, the operation status is returned with the description about the issue/failure.
- LeaseResultsOperationStatus - Attribute that displays the response of a web service operation for DHCP lease information query. In addition to the attributes of OperationStatus data type, this object also includes a DHCPLeaseinfo object. On success a DHCPLeaseinfo object with a SUCCESS code is returned. On failure, the operation status is returned with the description about the issue/failure.
- ClassOfServiceOperationStatus - Attribute that displays the response of a web service operation for a class of service query. In addition to the attributes of OperationStatus data type, this object also includes a ClassOfService object. On success a ClassOfService object with a SUCCESS code is returned. On failure, the operation status is returned with the description about the issue/failure.

- DHCPCriteriaOperationStatus - Attribute that displays the response of a web service operation for a DHCP criteria query. In addition to the attributes of OperationStatus data type, this object also includes a DHCPCriteria object. On success a DHCPCriteria object with a SUCCESS code is returned. On failure, the operation status is returned with the description about the issue/failure.
- FileOperationStatus - Attribute that displays the response of a web service operation for a File query. In addition to the attributes of OperationStatus data type, this object also includes a File object. On success, a File object with a SUCCESS code is returned. On failure, the operation status is returned with the description about the issue/failure.
- GroupOperationStatus - Attribute that displays the response of a web service operation for a group query. In addition to the attributes of OperationStatus data type, this object also includes a Group object. On success, a Group object with a SUCCESS code is returned. On failure, the operation status is returned with the description about the issue/failure.

PWS Operations

A Java like syntax is used to denote the operations of the Provisioning Web Service.

This section includes the following PWS operations:

- Session Operations, page 8-6
- Device Provisioning Operations, page 8-7
- Device Type Operations, page 8-17
- Generic Device Operation, page 8-19
- Class Of Service Operations, page 8-20
- DHCP Criteria Operations, page 8-22
- File Operations, page 8-25
- Group Operations, page 8-28
- Custom Property Operations, page 8-30
- Poll Operation Status, page 8-31
- Search Operation, page 8-32
- REST PWS Operations, page 8-33
- Server Operations, page 8-36

Session Operations

Table 8-3 Creating a Session

Operation	Context createSession (String rduhost, int rduport, String username, String password)
Description	Authenticates a client and establishes a session between PWS client and the PWS
Pre-Condition	All parameters are required.
Post-Condition	A successful call will result in a Context object returned that can be used for all future requests.

Table 8-3 Creating a Session (continued)

Operation	Context createSession (String rduhost, int rduport, String username, String password)
Execution Options	None
Operation Options	None
Usage Restrictions	Authorization - The user must have a valid account in either the RDU or an external AAA. Concurrent Access - There is no limitation on the number of concurrent access of this interface. Concurrent calls results in concurrent RDU requests.
Error Handling	AccessDeniedException - User does not have proper privileges or credentials are wrong. ProvServiceException - The specified RDU is not reachable. A ProvServiceException is a generic fault that reflects the WS has trouble satisfying the operation request. If this error is displayed it suggests that no sessions could be established with an RDU for example, Connection info is wrong, RDU not reachable, or some internal error. The fault should contain some info to allow the client to take corrective action e.g. Fix the distribution.

Table 8-4 Closing a Session

Operation	OperationStatus closeSession(Context context)
Description	Releases the session between the user and the web service including closing and connection with Prime CP components.
Pre-Condition	A valid context must be provided
Post-Condition	A successful call will result in the context being released. No further requests using this context will be allowed.
Execution Options	None
Operation Options	None
Usage Restrictions	Authorization - The user must have a valid context. Concurrent Access - There is no limitation on the number of concurrent access of this interface.
Error Handling	AccessDenied - User does not have proper privileges or credentials are wrong. ProvServiceException - The context is not valid.

Device Provisioning Operations

Table 8-5 Adding a New Device

Operation	OperationStatus addDevice (Context context, Device device, Options options)
Description	Submit a request for the addition of a new device.

Table 8-5 Adding a New Device (continued)

Pre-Condition	Context must be valid. Device argument must not be null. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in device being added to Prime Cable Provisioning database. A failure will result in no change to the database.
Operation Options	None
Execution Options	See Table 8-1.
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to add the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The device objects contains invalid or missing data.

Table 8-6 Adding Multiple New Devices

Operation	OperationStatus addDevices (Context context, List<Device> devices, Options options)
Description	Submit a request to add multiple new devices. Using the execution options, each device can be wrapped in a separate transaction (batch) or a single transaction.
Pre-Condition	Context must be valid. List of device argument must not be null. Options argument is optional. If null, default execution options will be used.
Post-Condition	A OperationStatus object is returned. A successful call will result in all device being added to Prime Cable Provisioning database.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to add the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The device objects contains invalid or missing data.

Table 8-7 Retrieving Data of a Specific Device

Operation	DeviceOperationStatus getDevice (Context context, DeviceId deviceId, PropertyFilter filter, Options options)
Description	Retrieve data associated with a specific device.
Pre-Condition	Context must be valid. A valid device ID must be provided. PropertyFilter argument is optional. If null, all properties are returned. Otherwise all properties specified will be returned. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in device data being returned otherwise a fault will be returned.
Execution Options	See Table 8-1.
Operation Options	includeLeaseInfo - Include DHCP lease info in both IPv4 and IPv6 in the result. DHCP lease information will be retrieved from provisioning group DHCP server(s). If no DHCP server responds, an error condition is returned. If a DHCP server does respond, but there is no active lease, no lease information will be included in the result and there will be no error returned.
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The device id contains invalid or missing data or uniqueness constraints were violated. Device does not exist.

Table 8-8 Retrieving Data of Multiple Devices

Operation	List<DeviceOperationStatus> getDevices(Context context, List<DeviceId> deviceId, PropertyFilter filter, Options options)
Description	Retrieve data associated with the specific devices.
Pre-Condition	Context must be valid. A list of valid device ID must be provided. List must contain at least one entry. PropertyFilter argument is optional. If null, all properties are returned. Otherwise all properties specified will be returned. Is present, will apply to all devices. Options argument is optional. If null, default execution options will be used. Options present will apply to all devices.
Post-Condition	A successful call will result in list of DeviceOperationStatus being returned.
Execution Options	See Table 8-1.

Table 8-8 Retrieving Data of Multiple Devices (continued)

Operation Options	includeLeaseInfo - Include DHCP lease info in both IPv4 and IPv6 in the result. DHCP lease information will be retrieved from provisioning group DHCP server(s). If no DHCP server responds, an error condition is returned. If a DHCP server does respond, but there is no active lease, no lease information will be included in the result and there will be no error returned.
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access – There is no limitation on the number of concurrent access of this operation.
Error Handling	ProvServiceException – The device id contains invalid or missing data or uniqueness constraints were violated. Device does not exist.

Table 8-9 Retrieving List of Devices Downstream of a specific Device

Operation	DevicesBehindOperationStatus getDevicesBehindDevice(Context context, DeviceIdSet deviceId, PropertyFilter propertyFilter, Options options)
Description	Retrieve the list of devices downstream of a specific device. Either the device ids or the entire device object is returned.
Pre-Condition	Context must be valid. A valid device ID must be provided. PropertyFilter argument is optional. If null, all properties are returned. Otherwise all properties specified will be returned. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in the DevicesBehindOperationStatus object which in turn includes the list of device objects. This operation will only return device ids or device objects that are behind this device.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation. The operation will fail if the specified device has more than 100 devices behind it, and the attribute **stopOnFailure** is set to **True.**
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The device id contains invalid or missing data or device does not exist

Table 8-10 Retrieving List of Devices Downstream of Specified Devices

Operation	List<DevicesBehindOperationStatus> getDevicesBehindDevices(Context context, List<DeviceIdSet> deviceId, PropertyFilter propertyFilter, Options options)
Description	Retrieve the list of devices downstream of the specified devices. Either the device IDs or the entire device object is returned.
Pre-Condition	Context must be valid. A valid list of device IDs must be provided containing at least one device id. PropertyFilter argument is optional. If null, all properties are returned. Otherwise all properties specified will be returned. Is present, will apply to all devices. Options argument is optional. If null, default execution options will be used. Options present will apply to all devices.
Post-Condition	A successful call will result in the list of DevicesBehindOperationStatus which in turn holds the list of target devices. This operation will only return device ids or device objects that are behind the devices.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation. For 5.0 only, supports transactionPerItem set to true. The operation will fail if the specified device has more than 100 devices behind it, and the attribute **stopOnFailure** is set to **True.**
Error Handling	ProvServiceException – The device id contains invalid or missing data or device does not exist.

Table 8-11 Updating Properties of a Device

Operation	OperationStatus updateDevice(Context context, DeviceId deviceId, Device device, PropertiesToDelete propertiesToDelete, GroupsToUnassign groupsToUnassign, Options options)
Description	Updates the properties of the specified device.
Pre-Condition	Context must be valid. A valid Device ID must be provided. A valid Device must be provided to serve as a template of changes to make. Any non-null value will be used to update that respective field. All property and their values found in Properties will be either added if the property does not exist or updated if it does exist. PropertiesToDelete specifies those properties to be removed from the Device object. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in device being updated.

Table 8-11 Updating Properties of a Device (continued)

Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to update the object and its properties. Concurrent Access - There is no limitation on the number of concurrent access of this operation. However, if multiple calls to update the same object are made, the last update will win.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The Device contains invalid data or specifies non-existent device.

Table 8-12 Updating Properties of Multiple Devices

Operation	OperationStatus updateDevices(Context context, List<DeviceId> deviceId, Device device, PropertiesToDelete propertiesToDelete, GroupsToUnassign groupsToUnassign, Options options)
Description	This operation applies the data associated with a specific device object to all the devices specified in the list of device IDs. The properties that are excluded for this operation are deviceIds, hostName, fqdn, and embeddedDevices.
Pre-Condition	Context must be valid. A valid list of DeviceId must be provided containing at least one device id. A valid Device must be provided to serve as a template of changes to make. Any non-null value will be used to update that respective field. The changes will be applied to all specified devices. All property found in Properties will be either added if the property does not exist or updated if it does exist. PropertiesToDelete specifies those properties to be removed from the Device object. Options argument is optional. If null, default execution options will be used. Options present will apply to all devices. If present in the device object passed into this operation, the changing device id is ignored in this operation. Change of deviceIsd is not supported.
Post-Condition	A list of OperationStatus objects will be returned. A successful call will result in all devices being updated.
Execution Options	See Table 8-1.
Operation Options	None

Table 8-12 Updating Properties of Multiple Devices (continued)

Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to update the object and its properties. Concurrent Access - There is no limitation on the number of concurrent access of this operation. However, if multiple calls to update the same object are made, the last update will win.
Error Handling	ProvServiceException – The Device contains invalid data or specifies non-existent device.

Table 8-13 Deleting a Device

Operation	OperationStatus deleteDevice(Context context, DeviceId deviceid, Options options)
Description	Deletes the specified device.
Pre-Condition	Context must be valid. A valid DeviceId must be provided. Options argument is optional. If null, default execution options will be used.
Post-Condition	An OperationStatus object will be returned. A successful call will result in device being deleted.
Execution Options	See Table 8-1.
Operation Options	deleteDevicesBehind– A boolean flag to determine whether to delete the devices (if any) behind the specified IP device. If deleteDevicesBehind flag is set to “true”, all devices behind the specified device will be deleted. If deleteDevicesBehind flag is set to “false”, unregistered devices behind the specified device will be deleted. Registered devices will have their discovered DHCP data and provisioning group relationship delete leaving only their registered data. The default value is false.
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to delete the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Missing required arguments or DeviceId is missing or specifies non-existent device.

Table 8-14 Deleting Multiple Devices

Operation	OperationStatus deleteDevices (Context context, List<DeviceId> deviceid, Options options)
Description	Deletes the specified devices.

Table 8-14 Deleting Multiple Devices (continued)

Pre-Condition	Context must be valid. A valid list of DeviceIds must be provided. The list must contain at least one deviceid. Options argument is optional. If null, default execution options will be used. Options present will apply to all devices.
Post-Condition	An OperationStatus object will be returned. A successful call will result in all devices being deleted.
Execution Options	See Table 8-1.
Operation Options	deleteDevicesBehind– A boolean flag to determine whether to delete the devices (if any) behind the specified IP device. If deleteDevicesBehind flag is set to “true”, all devices behind the specified device will be deleted. If deleteDevicesBehind flag is set to “false”, unregistered devices behind the specified device will be deleted. Registered devices will have their discovered DHCP data and provisioning group relationship delete leaving only their registered data. The default value is false.
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to delete the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	ProvServiceException – Missing required arguments or DeviceId is missing or specifies non-existent device.

Table 8-15 Unregistering a Device

Operation	OperationStatus unregisterDevice(Context context, DeviceId deviceid, Options options)
Description	Unregister a device.
Pre-Condition	Context must be valid. A valid DeviceId must be provided. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in device being unregistered. If the device is registered, transition the device to the unregistered state. If the device is unregistered, it will be deleted from the database.
Execution Options	See Table 8-1.
Operation Options	None

Table 8-15 Unregistering a Device (continued)

Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to delete the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Missing required arguments or DeviceId is missing or specifies non-existent device.

Table 8-16 Unregistering Multiple Devices

Operation	OperationStatus unregisterDevices(Context context, DeviceId deviceid, Options options)
Description	Unregister devices.
Pre-Condition	Context must be valid. A valid list of DeviceIds must be provided containing at least one device. Options argument is optional. If null, default execution options will be used. Options are applied to the transaction as a whole.
Post-Condition	A successful call will result in all specified device being unregistered. If a device is registered, transition the device to the unregistered state. If a device is unregistered, it will be deleted from the database.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to delete the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	ProvServiceException – Missing required arguments or DeviceId is missing or specifies non-existent device.

Table 8-17 Retrieving DHCP Lease Information of a IP Address

Operation	LeaseResultsOperationStatus getDHCPLeaseInfo(Context context, IPAddress ipAddress, List<String> provGroups, Options options)
Description	Retrieves all known DHCP lease information about the specified IP Address.
Pre-Condition	Context must be valid. A valid device ID must be provided. IPAddress must be provided and can be either in IPv4 or IPv6 format. An optional list of provisioning groups to search. If null, DHCP servers in all provisioning groups are queried. Options argument is optional. If null, default execution options will be used.

Table 8-17 Retrieving DHCP Lease Information of a IP Address (continued)

Post-Condition	A successful call will result in DHCP both IPv4 and IPv6 lease query information returned. DHCP lease information will be retrieved from provisioning group DHCP server(s). If no DHCP server responds, an error condition is returned. If a DHCP server does respond, but there is no active lease, no lease information will be included in the result and there will be no error returned.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The IPAddress is missing or contains invalid data. This fault will also be raised if the DHCP server could not be contacted.

Table 8-18 Regenerating Configurations for the Devices that Match the Search Criteria

Operation	OperationStatus regenConfigs(Context context, Search deviceSearch, Options options)
Description	Submit a request to the RDU's Configuration Regeneration Service to regenerate configurations for the set of devices that match the specified search criteria.
Pre-Condition	Context must be valid. A valid Search must be provided. Search.query and Search.maxResults are the only required fields. Options argument is optional. If null, default execution options will be used.
Post-Condition	This operation returns immediately. A success signifies that it has been queued by the RDU for processing. The returned OperationStatus will contain the batch id that can be used to monitor the status of the regeneration.
Execution Options	See Table 8-1.
Operation Options	None

Table 8-18 Regenerating Configurations for the Devices that Match the Search Criteria

Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to update the devices. Concurrent Access – Duplicate requests of this operation will be not be allowed.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong.

Table 8-19 Rebooting a Device

Operation	OperationStatus rebootDevice(Context context, DeviceId deviceid, Options options)
Description	Reboot a device.
Pre-Condition	Context must be valid. A valid DeviceId must be provided. Under Execution options, the activationMode must be AUTOMATIC.
Post-Condition	A successful call will result in device being rebooted.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to update the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Missing required arguments or DeviceId is missing or specifies non-existent device.

Device Type Operations

Table 8-20 Adding a New Device Type

Operation	OperationStatus addDeviceType (Context context, String deviceType, Options options)
Description	Submit a request for the addition of a new device type.
Pre-Condition	Context must be valid. DeviceType argument must not be null. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in a new device type being added to Prime CP' database.

Table 8-20 Adding a New Device Type (continued)

Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to add the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The device objects contains invalid or missing data.

Table 8-21 Retrieving Data of a Device Type Definition

Operation	DeviceTypeOperationStatus getDeviceTypes(Context context, Options options)
Description	Retrieves all the device types available in the database.
Pre-Condition	Context must be valid. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in all the device types being returned.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The device id contains invalid or missing data or uniqueness constraints were violated.

Table 8-22 Deleting a Device Type

Operation	OperationStatus deleteDeviceType(Context context, String deviceType, Options options)
Description	Deletes the specified device type.
Pre-Condition	Context must be valid. A valid DeviceType name must be provided. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in the device type being deleted.
Execution Options	See Table 8-1.

Table 8-22 Deleting a Device Type (continued)

Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to delete the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Missing required DeviceType name or non-existent DeviceType.

Table 8-23 Updating a Device Type

Operation	OperationStatus updateDeviceType(Context context, String deviceTypeName, Map<String, String> propertiesToUpdate, PropertiesToDelete propertiesToDelete,Options options)
Description	Updates the specified device type.
Pre-Condition	Context must be valid. A valid DeviceType name must be provided. The properties to be updated or deleted must be provided. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in the device type being updated.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to update the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Missing required DeviceType name or non-existent DeviceType.

Generic Device Operation

Table 8-24 Generic Operation

Operation	OperationStatus deviceOperation(Context context, List<DeviceId> deviceIds, DeviceCommand command,Options options)
Description	A generic operation that sends an opaque command and parameters to all specified devices.

Table 8-24 Generic Operation (continued)

	Context must be valid. A valid list of one or more DeviceIds must be provided. A valid DeviceCommand specifying the device specific operation must exist. The command and its arguments are opaque to the service. Example of operations includes generation device's configuration, resetting the device, and enabling SNMP v3 access to the device. Each DeviceCommand includes an optional set of parameters that are specific to the type of command being executed.
Pre-Condition	Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in a list of asynchronous batch status being returned. The asynchronous batch status can be used for further query in pollOperation.
Execution Options	Only asynchronous mode is supported. See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to perform device operations. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	ProvServiceException – Missing required arguments or DeviceId is missing or specifies non-existent device.

Class Of Service Operations

Table 8-25 Adding a New Class of Service

Operation	OperationStatus addClassOfService(Context context, ClassOfService cos, Options options)
Description	Submit a request for the addition of a new class of service.
Pre-Condition	Context must be valid. ClassOfService argument must not be null. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in a new COS being added to Prime CP' database.
Execution Options	See Table 8-1.
Operation Options	None

Table 8-25 Adding a New Class of Service (continued)

Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to add the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The ClassOfService object contains invalid or missing required data.

Table 8-26 Retrieving Data of a Class of Service

Operation	ClassOfServiceOperationStatus getClassOfService(Context context, String cosName, Options options)
Description	Retrieve data associated with the specified COS.
Pre-Condition	Context must be valid. A valid COS name must be provided. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in COS data being returned.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The COS name is missing.

Table 8-27 Updating Properties of a COS Object

Operation	OperationStatus updateClassOfService(Context context, String cosName, ClassOfService cos, PropertiesToDelete propertiesToDelete, Options options)
Description	Updates the properties of the specified COS object.
Pre-Condition	Context must be valid. A valid COS name must be provided. A valid ClassOfService attribute must be provided. Any non-null value will be used to update that respective field. All the properties and their values are either added if the properties does not exist, or updated if the properties exist. In 5.0, renaming a COS will not be supported. PropertiesToDelete specifies those properties to be removed from the ClassOfService object. Options argument is optional. If null, default execution options will be used.

Table 8-27 Updating Properties of a COS Object (continued)

Post-Condition	A successful call will result in ClassOfService being updated.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to update the object and its properties. Concurrent Access - There is no limitation on the number of concurrent access of this operation. However, if multiple calls to update the same object are made, the last update will win.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – A ClassOfService by the name passed does not exist or the ClassOfService contains invalid or missing data.

Table 8-28 Deleting a Class of Service

Operation	OperationStatus deleteClassOfService(Context context, String cosName, Options options)
Description	Deletes the specified ClassOfService.
Pre-Condition	Context must be valid. A valid ClassOfService name must be provided. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in the ClassOfService being deleted.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to delete the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Missing required ClassOfService name or non-existent ClassOfService.

DHCP Criteria Operations

Table 8-29 Adding a New DHCP Criteria

Operation	OperationStatus addDHCPCriteria(Context context, DHCPCriteria dhcpCriteria, Options options)
Description	Submit a request for the addition of a new DHCPCriteria.

Table 8-29 Adding a New DHCP Criteria (continued)

Pre-Condition	Context must be valid. DHCPCriteria argument must not be null. It must contain a valid name and at least one of the following: clientClass, includeSectionTags, and/or excludeSelectionTags. Properties are optional. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in a new DHCPCriteria being added to Prime CP' database.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to add the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The DHCPCriteria object contains invalid or missing required data.

Table 8-30 Retrieving Data of a DHCP Criteria

Operation	DHCPCriteriaOperationStatus getDHCPCriteria(Context context, String dhcpCriteriaName, Options options)
Description	Retrieve data associated with the specified DHCPCriteria.
Pre-Condition	Context must be valid. A valid DHCPCriteria name must be provided. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in DHCPCriteria data being returned.
Execution Options	See Table 8-1.
Operation Options	None

Table 8-30 Retrieving Data of a DHCP Criteria (continued)

Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The DHCPCriteria name is missing or invalid.

Table 8-31 Updating Properties of a DHCP Criteria Object

Operation	OperationStatus updateDHCPCriteria(Context context, String dhcpCriteriaName, DHCPCriteria dhcpCriteria, PropertiesToDelete propertiesToDelete, Options options)
Description	Updates the properties of the specified DHCPCriteria object.
Pre-Condition	Context must be valid. A valid DHCPCriteria name must be provided. A valid DHCPCriteria must be provided. Any non-null value will be used to update that respective field. All property and their values found in Properties will be either added if the property does not exist or updated if it does exist. In 5.0, renaming a DHCPCriteria will not be supported. PropertiesToDelete specifies those properties to be removed from the DHCPCriteria object. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in DHCPCriteria being updated.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to update the object and its properties. Concurrent Access - There is no limitation on the number of concurrent access of this operation. However, if multiple calls to update the same object are made, the last update will win.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – A DHCPCriteria by the name passed does not exist or the DHCPCriteria contains invalid data.

Table 8-32 Deleting a DHCP Criteria

Operation	OperationStatus deleteDHCPCriteria(Context context, String dhcpCriteriaName, Options options)
Description	Deletes the specified DHCPCriteria.

Table 8-32 Deleting a DHCP Criteria (continued)

Pre-Condition	Context must be valid. A valid DHCPCriteria name must be provided. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in the DHCPCriteria being deleted.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to delete the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Missing required DHCPCriteria name or non-existent DHCPCriteria.

File Operations

Table 8-33 Adding a New File

Operation	OperationStatus addFile(Context context, File file, Options options)
Description	Submit a request for the addition of a new File.
Pre-Condition	Context must be valid. File argument must not be null. It must contain a valid filename, filetype. Properties are optional. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in a new File being added to Prime Cable Provisioning database.
Execution Options	See Table 8-1.
Operation Options	None

Table 8-33 Adding a New File (continued)

Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to add the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The File object contains invalid or missing required data.

Table 8-34 Retrieving Data of a File

Operation	FileOperationStatus getFile(Context context, String fileName, boolean retrieveFileData, PropertyFilter propertyFilter, Options options)
Description	Retrieve data associated with the specified File.
Pre-Condition	Context must be valid. A valid filename name must be provided. PropertyFilter is optional. If null, only the file's properties will be returned and not the data. If a PropertyFilter is provided with the property fileData, all the properties including data will be returned through MTOM. Also, if retrieveFileData is set to true, all properties including the data will be returned. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in specified File properties or data being returned.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The file name is missing or invalid or the file does not exist.

Table 8-35 Updating Properties of a File Object

Operation	OperationStatus updateFile(Context context, String fileName, File file, PropertiesToDelete propertiesToDelete, Options options)
Description	Updates the properties or data of the specified File object.

Table 8-35 Updating Properties of a File Object (continued)

	Context must be valid. A valid file name must be provided. A valid File must be provided. Any non-null value will be used to update that respective field. All property and their values found in Properties will be either added if the property does not exist or updated if it does exist. In 5.0, renaming a file is not supported. PropertiesToDelete specifies those properties to be removed from the File object.
Pre-Condition	Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in the File being updated.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to update the object and its properties. Concurrent Access - There is no limitation on the number of concurrent access of this operation. However, if multiple calls to update the same object are made, the last update will win.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The file name passed, doesn't exist.

Table 8-36 Deleting a File

Operation	OperationStatus deleteFile(Context context, String fileName, Options options)
Description	Deletes the specified File.
Pre-Condition	Context must be valid. A valid File name must be provided. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in the File being deleted.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to delete the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Missing required File name or non-existent File.

Group Operations

Table 8-37 Adding a New Group

Operation	OperationStatus addGroup(Context context, Group group, Options options)
Description	Add a new group.
Pre-Condition	Context must be valid. Group argument must not be null. It must contain a valid name and group type. Properties are optional. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in a new Group being added to Prime Cable Provisioning database.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to add the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The Group object contains invalid or missing required data.

Table 8-38 Retrieving Data of a Group

Operation	GroupOperationStatus getGroup(Context context, String groupName, Options options)
Description	Retrieve data associated with the specified Group name.
Pre-Condition	Context must be valid. A valid Group name must be provided. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in Group data being returned.
Execution Options	See Table 8-1.
Operation Options	None

Table 8-38 Retrieving Data of a Group (continued)

Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation. In 5.0, the type of the group will not be returned.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The Group name is missing or invalid, or Group does not exist.

Table 8-39 Updating Properties of a Group

Operation	OperationStatus updateGroup(Context context, String groupName, Group group, PropertiesToDelete propertiesToDelete, Options options)
Description	Updates the properties of the specified Group.
Pre-Condition	Context must be valid. A valid Group name must be provided. A valid Group must be provided. Any non-null value will be used to update that respective field. All property and their values found in Properties will be either added if the property does not exist or updated if it does exist. PropertiesToDelete specifies those properties to be removed from the Group object. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in Group being updated.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to update the object and its properties. Concurrent Access - There is no limitation on the number of concurrent access of this operation. However, if multiple calls to update the same object are made, the last update will win.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – The group name passed, doesn't exist.

Table 8-40 Deleting a Group

Operation	OperationStatus deleteGroup(Context context, String groupName, Options options)
Description	Deletes the specified Group.

Table 8-40 Deleting a Group (continued)

Pre-Condition	Context must be valid. A valid Group name must be provided. Options argument is optional. If null, default execution options will be used.
Post-Condition	A successful call will result in the Group being deleted. Any member object will be automatically unassigned.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to delete the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Missing required Group name or non-existent Group.

Custom Property Operations

Table 8-41 Adding a New Custom Property

Operation	OperationStatus addCustomProperty(Context context, CustomProperty customProperty, boolean encryptedProperty).
Description	Submit a request for the addition of a new custom property.
Pre-Condition	Context must be valid. CustomProperty argument must not be null. encryptedProperty value can be any boolean value.
Post-Condition	A successful call will result in a new custom property being added to Prime CP' database.
Usage Restrictions	Authorization - Can only be called by authenticated users with valid context and proper privileges to add the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException - User does not have proper privileges or credentials are wrong. ProvServiceException - The CustomProperty object contains invalid or missing required data.

Table 8-42 Retrieving Data of a Custom Property

Operation	CustomPropertyOperationStatus getCustomProperty(Context context)
Description	Retrieve data associated with the specified Custom Property.
Pre-Condition	Context must be valid.

Table 8-42 Retrieving Data of a Custom Property (continued)

Operation	CustomPropertyOperationStatus getCustomProperty(Context context)
Post-Condition	A successful call will result in Custom property data being returned.
Usage Restrictions	Authorization - Can only be called by authenticated users with valid context and proper privileges to add the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException - User does not have proper privileges or credentials are wrong. ProvServiceException - The CustomProperty object contains invalid or missing required data.

Table 8-43 Deleting a Custom Property

Operation	OperationStatus deleteCustomProperty(Context context, String customProperty)
Description	Deletes the specified Custom property.
Pre-Condition	Context must be valid. A valid Custom property name must be provided.
Post-Condition	A successful call will result in Custom property data being deleted.
Usage Restrictions	Authorization - Can only be called by authenticated users with valid context and proper privileges to add the object. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException - User does not have proper privileges or credentials are wrong. ProvServiceException - The CustomProperty object contains invalid or missing required data.

Poll Operation Status

Table 8-44 Querying Status of the RDU Asynchronous Requests

Operation	OperationStatus pollOperationStatus(Context context, String txId, Options options)
Description	Queries for the status of RDU asynchronous requests specified by the transaction ID.
Pre-Condition	Context must be valid. A non-null txId must be provided. Options argument is optional. If null, default execution options will be used.

Table 8-44 Querying Status of the RDU Asynchronous Requests (continued)

Post-Condition	A successful call will result in the OperationStatus of the request being return. For non-reliable asynchronous RDU requests that had completed, a request not found will be returned. The support is also available for the following API requests where the OperationStatus is returned: • getDeviceTypes • getDHCPCriteria • getDHCPLeaseInfo • getClassOfService • getDevice
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	ProvServiceException – Missing required data such as request id.

Search Operation

Table 8-45 Search Operation

Operation	SearchResult search(Context context, Search search, Options options)
Description	This is a generic operation. Retrieves devices or classes of service or files or DHCP Criterion or groups based on the search criterion defined in search object.
Pre-Condition	Context must be valid. A valid Search must be provided. Search.query is the only required field. Options argument is optional, if null, default execution options will be used.
Post-Condition	A successful call will result in devices or classes of service or files or DHCP Criterion or groups objects being returned. In 5.0, for device search based on owner id, Class of Service search and DHCP Criterion search, RDU doesn't support paging while retrieving the matching results from the database. Hence for these types of search operations, all matching objects will be retrieved at one shot.
Execution Options	See Table 8-1.
Operation Options	None

Table 8-45 Search Operation

Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges. Concurrent Access - There is no limitation on the number of concurrent access of this operation.
Error Handling	ProvServiceException – The Search object contains invalid or missing data.

REST PWS Operations

Table 8-46 Get All RDU Operation

Operation	ResponseEntity<?> getAllRDU(@RequestHeader(value = HttpHeaders.AUTHORIZATION, required = false) String basicAuthEncryptValue)
Description	Retrieve a list of all the RDUs registered with REST PWS
Pre-Condition	Authorization type needs to be BASIC AUTH with valid Username and Password (i,e Username and Password provided at the time of REST PWS installation).
Post-Condition	A successful call returns the list of all the Regional Distribution Unit servers.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization - Can only be called by Authorization type as BASIC AUTH with valid Username and Password.
Error Handling	UNAUTHORIZED - When Authorization type is not BASIC AUTH, Username and Password are wrong.
Exception	UNAUTHENTICATED to REST PWS will be returned if Authorization type is not BASIC AUTH, Username and Password are wrong.

Table 8-47 Add RDU Operation

Operation	ResponseEntity<?> addRDUDetails (@RequestBody(required = false) RDUContextWrapper rduContextObject, @RequestHeader(value = HttpHeaders.AUTHORIZATION, required = false) String basicAuthEncryptValue)
Description	Submit a request for the addition of a RDU to REST PWS
Pre-Condition	Authorization type needs to be BASIC AUTH with valid Username and Password (i,e Username and Password provided at the time of REST PWS installation).
Post-Condition	A successful call returns the Regional Distribution Unit server.
Execution Options	See Table 8-1.
Operation Options	None

Table 8-47 Add RDU Operation

Usage Restrictions	Authorization - Can only be called by Authorization type as BASIC AUTH with valid Username and Password.
Error Handling	UNAUTHORIZED - When Authorization type is not BASIC AUTH, Username and Password are wrong.
Exception	UNAUTHENTICATED to REST PWS will be returned if Authorization type is not BASIC AUTH, Username and Password are wrong. RDU details already exist for the host will be returned if you are trying to add RDU which already exists.

Table 8-48 Get RDU Operation

Operation	ResponseEntity<?> getRDU (@PathVariable String rduHost, @RequestHeader(value = HttpHeaders.AUTHORIZATION, required = false) String basicAuthEncryptValue)
Description	Retrieve the data associated with the specified RDU
Pre-Condition	Authorization type needs to be BASIC AUTH with valid Username and Password (i,e Username and Password provided at the time of REST PWS installation).
Post-Condition	A successful call returns the Regional Distribution Unit server.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization - Can only be called by Authorization type as BASIC AUTH with valid Username and Password.
Error Handling	UNAUTHORIZED - When Authorization type is not BASIC AUTH, Username and Password are wrong.
Exception	UNAUTHENTICATED to REST PWS will be returned if Authorization type is not BASIC AUTH, Username and Password are wrong. RDU details does not exist for the host will be returned if you specify invalid rduHost.

Table 8-49 Update RDU Operation

Operation	ResponseEntity<?> updateRDUDetails (@PathVariable String rduHost, @RequestBody(required = false) PWSUpdateRDUWrapper pwsContextObject, @RequestHeader(value = HttpHeaders.AUTHORIZATION, required = false) String basicAuthEncryptValue)
Description	Updates the properties or data of the specified RDU

Table 8-49 Update RDU Operation

Pre-Condition	Authorization type needs to be BASIC AUTH with valid Username and Password (i,e Username and Password provided at the time of REST PWS installation).
Post-Condition	A successful call returns no content and updates the properties of Regional Distribution Unit server.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization - Can only be called by Authorization type as BASIC AUTH with valid Username and Password.
Error Handling	UNAUTHORIZED - When Authorization type is not BASIC AUTH, Username and Password are wrong.
Exception	UNAUTHENTICATED to REST PWS will be returned if Authorization type is not BASIC AUTH, Username and Password are wrong. RDU details does not exist for the host will be returned if you specify invalid rduHost.

Table 8-50 Delete RDU Operation

Operation	ResponseEntity<?> removeRDUDetails (@PathVariable String rduHost, @RequestHeader(value = HttpHeaders.AUTHORIZATION, required = false) String basicAuthEncryptValue)
Description	Deletes the specified RDU
Pre-Condition	Authorization type needs to be BASIC AUTH with valid Username and Password (i,e Username and Password provided at the time of REST PWS installation).
Post-Condition	A successful call returns no content and removes the Regional Distribution Unit server.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization - Can only be called by Authorization type as BASIC AUTH with valid Username and Password.
Error Handling	UNAUTHORIZED - When Authorization type is not BASIC AUTH, Username and Password are wrong.
Exception	UNAUTHENTICATED to REST PWS will be returned if Authorization type is not BASIC AUTH, Username and Password are wrong. RDU details does not exist for the host will be returned if you specify invalid rduHost.

Server Operations

Table 8-51 Retrieving List of CNR Servers Registered with the System

Operation	GetAllCNRsResult getAllCNRs (Context context)
Description	Retrieve a list of all the Network Registrar servers currently registered with the system.
Pre-Condition	Context must be valid.
Post-Condition	A successful call will result in list of all the Network Registrar servers being returned.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access – There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Device Type can't be null. Please enter a valid Device Type.

Table 8-52 Retrieving List of DPEs Registered with the System

Operation	GetAllDPEsResult getAllDPEs(Context context)
Description	Retrieve a list of all the Device provisioning engines currently registered with the system.
Pre-Condition	Context must be valid.
Post-Condition	A successful call will result in list of all the DPEs
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access – There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong.

Table 8-53 Retrieving All Provisioning Groups for the Specified Parameters

Operation	ProvGroupsResult getAllProvGroups(Context context)
Description	Retrieve a list of all the provisioning groups.
Pre-Condition	Context must be valid.

Table 8-53 Retrieving All Provisioning Groups for the Specified Parameters (continued)

Post-Condition	A successful call will result in list of all the provisioning groups.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access – There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong.

Table 8-54 Retrieving CNR Details of a Hostname

Operation	CNRDetailsResult getCNRDetails (Context context, HostAddress hostAddress)
Description	Retrieve the details of CNR server.
Pre-Condition	Context must be valid. A valid host name must be provided.
Post-Condition	A successful call will result in details of CNR server data being returned.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access – There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Hostname can't be empty. Please enter a valid Hostname.

Table 8-55 Retrieving Default Details of a Specified Device Type

Operation	DefaultsResult getDefaults(Context context, DeviceType deviceType)
Description	Retrieves the default property values for the specified DeviceType.
Pre-Condition	Context must be valid. A valid DeviceType must be provided.
Post-Condition	A successful call will result in Defaults data being returned.
Execution Options	See Table 8-1.
Operation Options	None

Table 8-55 Retrieving Default Details of a Specified Device Type (continued)

Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access – There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Device Type can't be null. Please enter a valid Device Type.

Table 8-56 Retrieving DPE Defaults of a Hostname

Operation	DPEDefaultsResult getDPEDefaults(Context context, HostAddress hostAddress)
Description	Retrieve the defaults of DPE server.
Pre-Condition	Context must be valid. A valid host name must be provided.
Post-Condition	A successful call will result in details of DPE server data being returned.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access – There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Hostname can't be empty. Please enter a valid Hostname.

Table 8-57 Retrieving DPE Details of a Specified Device Type

Operation	DPEDetailsResult getDPEDetails(Context context, HostAddress hostAddress)
Description	Retrieve the details of DPE server.
Pre-Condition	Context must be valid. A valid host name must be provided.
Post-Condition	A successful call will result in details of DPE server data being returned.
Execution Options	See Table 8-1.
Operation Options	None

Table 8-57 Retrieving DPE Details of a Specified Device Type (continued)

Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access – There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Hostname can't be empty. Please enter a valid Hostname.

Table 8-58 Retrieving Extension Point Details for the Specified Parameters

Operation	ExtensionPointSettingsResult getExtensionPointSettings(Context context)
Description	Retrieves the Network Registrar extension point settings (the ones related to PCP) in a map.
Pre-Condition	Context must be valid.
Post-Condition	A successful call will result in Network Registrar extension point settings data being returned.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access – There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong.

Table 8-59 Retrieving License Key Details for the Specified Parameters

Operation	LicenseKeyDataResult getLicenseKeyData(Context context)
Description	Retrieve the details of License Key from RDU server.
Pre-Condition	Context must be valid. A valid hostname must be provided.
Post-Condition	A successful call will result in details of License Key Data for RDU server data being returned.
Execution Options	See Table 8-1.
Operation Options	None

Table 8-59 Retrieving License Key Details for the Specified Parameters (continued)

Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access – There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – Could not authenticate with RDU. ProvServiceException – Hostname can't be empty. Please enter a valid Hostname. The specified user name [abc] or password is not valid. Failed to establish a PACEConnection to [admin@hostname:port]; hostname: Name or service not known.

Table 8-60 Retrieving Details of a Specified Provisioning Group

Operation	ProvGroupDetailsResult getProvGroupDetails (Context context, ProvGroupType provGroupType)
Description	Retrieve the details of Provisioning Group.
Pre-Condition	Context must be valid. A valid Provisioning Group name must be provided.
Post-Condition	A successful call will result in details of Provisioning Group data being returned.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access – There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Provisioning Group Name can't be empty. Please enter a valid Provisioning Group Name.

Table 8-61 Retrieving RDU Defaults of a Hostname

Operation	RDUDefaultsResult getRDUDefaults(Context context)
Description	Retrieve the details of RDU server Default details.
Pre-Condition	Context must be valid. A valid host name must be provided.
Post-Condition	A successful call will result in details of RDU server Default details data being returned.
Execution Options	See Table 8-1.
Operation Options	None

Table 8-61 Retrieving RDU Defaults of a Hostname (continued)

Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access – There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong.

Table 8-62 Retrieving RDU Details of a Specified Hostname

Operation	RDUDetailsOperationStatus getRDUDetails (Context context, HostAddress hostAddress)
Description	Retrieve the details of RDU server.
Pre-Condition	Context must be valid. A valid host name must be provided.
Post-Condition	A successful call will result in details of RDU server data being returned.
Execution Options	See Table 8-1.
Operation Options	None
Usage Restrictions	Authorization – Can only be called by authenticated users with valid context and proper privileges to read the object. Concurrent Access – There is no limitation on the number of concurrent access of this operation.
Error Handling	AccessDeniedException – User does not have proper privileges or credentials are wrong. ProvServiceException – Hostname can't be empty. Please enter a valid Hostname.

PWS Samples

A URL and for parameters, a JSON like syntax is used to post the RESTful Provisioning Web Service.

This section includes the samples for PWS RESTful operations:

- Create Session
- Add Operation
- Update Operation
- Change Operation
- Get Operation
- Delete Operation
- Status, Reboot, Unregister, Search Operations
- Close Session

Create Session

Table 8-63 Create Session

Operation	URL	Request Body	Method
Create Session	http://bac-lnx-xxx:9101/cp-ws-rest-prov/createSession	{"rduHost": "bac-lnx-xxx","rduPort": "49187","username": "username","password":"password"}	POST

Note Session id value will be of the following format:

- When the load balancer is enabled.

```
xkCMebDaZqk1xdXoIMuWYN9giGrOjXWGJHjO0kawKXXzVqoCFUI3d7rVS-R6IGMMmlRzpUg7TjBwuWQe_cNLe1
mda7-1DJbMOeDlucF7Op_oQulo09qb0ZTGcv_b-s8Tr5N4zVZJGYhJpzKdiTRwGqvD_LuIlKSbRq1CMS-e15KO
QQ6lcRZOV4YYhf8GTMkURV5JF6wDDEdUMAuOZhMnqg==
```

Note Close session operation will not have any effect on the instances behind load balancer.

- When the load balancer is disabled:

```
B886BDD61590296429B530379AF5469CDAF78CF5
```

Add Operation

Table 8-64 Add Operation

Operation	URL	Request Body	Method
ADD COS	http://bac-lnx-xxx:9101/cp-ws-rest-prov/addCOS	{"deviceType":"DocsisModem" ,"sessionId":"D7BD8FADF897EB80365C5092030F187001FCA0F0" ,"name":"TestCOSRestPWS" ,"activationMode":"NO_ACTIVATION" ,"confirmationMode":"NO_CONFIRMATION" ,"asynchronous":"false" ,"reliableMode":"false" ,"timeout":"30000" ,"stopOnFailure":"true" ,"transactionPerItem":"false" ,"publishingMode":"NO_PUBLISHING" ,"properties":{"/cos/docsis/file":"gold.cm"} }	POST

ADD Device	http://bac-lnx-xxx:9101/cp-ws-rest-prov/addDevice	{
		"deviceType":"DOCSISModem",
		"sessionId":"42C76DAEAA76B1E90DC1F440D9E7E2F8D179C04F",
		"activationMode":"NO_ACTIVATION"
		,"confirmationMode":"NO_CONFIRMATION"
		,"asynchronous":"false"
		,"reliableMode":"false"
		,"timeout":"30000"
		,"stopOnFailure":"true"
		,"transactionPerItem":"false"
		,"publishingMode":"NO_PUBLISHING"
		,"cos":"unprovisioned-docsis"
		,"hostName":"TestHost"
		,"domainName":"Newtestdomain.com"
		,"properties":{"/fqdn/domain":"domain.com"},
		"deviceIdSet":{"macAddress":"1,6,00:00:00:00:12:02"}
		}
			POST

Table 8-64 Add Operation (continued)

Add Device Type	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/addDeviceType	{
		"rduHost": "pcp-lnx-xxx",
		"rduPort": 49187,
		"username": "username",
		"password": "password",
		"sessionId": "ABCDEF0123456789ABCDEF0123456789ABCDEF01",
		"deviceTypeName": "CustomDeviceType",
		"baseSelectionType": "custom/DOCSISModem",
		"properties": {
		"/extensionPoint/configurationGeneration": "com.cisco.provisioning.cpe.extensions.builtin.generation.DocsisModemGeneration",
		"/extensionPoint/serviceLevelSelection": "com.cisco.provisioning.cpe.extensions.builtin.selection.DocsisModemSelection",
		"/extensionPoint/activation": "com.cisco.provisioning.cpe.extensions.builtin.disruption.DocsisModemDisruption"
		},
		"behindDevice": true/false,
		"activationMode": "NO_ACTIVATION",
		"confirmationMode": "NO_CONFIRMATION",
		"publishingMode": "NO_PUBLISHING",
		"asynchronous": false,
		"reliableMode": false,
		"timeout": 30000,
		"stopOnFailure": true,
		"transactionPerItem": false
		}
			POST

Table 8-64 Add Operation (continued)

ADD Devices	http://bac-lnx-xxx:9101/cp-ws-rest-prov/addDevices	{
		"sessionId":"F61D528E364BC96F6956FF9A73FFB1ABA7C2371E",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING",
		"multiDeviceList":
		[
		{ "deviceIdSet": {"macAddress":"1,6,00:00:00:00:12:b2"},"deviceType":"DOCSISModem"},
		{ "deviceIdSet": {"macAddress":"1,6,00:00:00:00:12:b1"},"deviceType":"Computer"}
		
		]
		}
			POST

ADD DHCP Criteria	http://bac-lnx-xxx:9101/cp-ws-rest-prov/addDHCPCriteria	{
		"name":"RESTDHCP",
		"sessionId":"B6CFC958C3A445A0968754FFBDB9AF670EBE0762",
		"properties":{"/fqdn/domain":"cisco167.com"},
		"clientClass":"TestCC",
		"excludeSelectionTags":"Testest",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING"
		}
			POST

Table 8-64 Add Operation (continued)

ADD File	http://bac-lnx-xxx:9101/cp-ws-rest-prov/addFile	{"sessionId":"A37481F3AD753387D969ECDBB466D7BE0AD2D269","fileType":"GENERIC","fileName":"Test.txt","stream":"Testing Scenario"}	POST
	http://bac-lnx-xxx:9101/cp-ws-rest-prov/addFile	{ "sessionId":"A37481F3AD753387D969ECDBB466D7BE0AD2D269", "fileType": "GENERIC", "fileName": "test.txt", "fileDataBytes": [35, 33, 47, 98, 105, 110, 47, 115]}	POST

ADD Group	http://bac-lnx-xxx:9101/cp-ws-rest-prov/addGroup	{
		"name":"Mytestgroup",
		"sessionId":"B6CFC958C3A445A0968754FFBDB9AF670EBE0762",
		"groupType":"system",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING"
		}
			POST

Add Custom Property without encryption property	http://bac-lnx-xxx:9101/cp-ws-rest-prov/addCustomProperty	{"sessionId": "E5BE0A6F4F465CACBF2A35ACC1F799930B9972B1", "customProperty":"testCus","dataType":"INTEGER"}	POST

Table 8-64 Add Operation (continued)

Add Custom Property with encryption property	http://bac-lnx-xxx:9101/cp-ws-rest-prov/addCustomProperty	{"sessionId": "E5BE0A6F4F465CACBF2A35ACC1F799930B9972B1", "customProperty":"testCus","dataType":"INTEGER", "encryptedProperty":"true"}	POST

Add RDU
 	
Authorization - BASIC AUTH
	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/RDU	{
		"username": "username",
		"password": "password",
		"rduHost": "pcp-lnx-xxx.cisco.com",
		"rduPort": 49187
		}
			POST

Update Operation

Table 8-65 Update Operation

Operation	URL	Request Body	Method

Update COS	http://bac-lnx-xxx:9101/cp-ws-rest-prov/updateCOS	{
		"sessionId":"7C47EBD1E3436F41869DFBC4B6CA7B7F29A63D26",
		"resourceName":"TestCOSRestPWS",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING",
		"properties":{"/cos/docsis/file":"silver.cm"},
		"list[0]":"/fqdn/domain"
		}
			PUT

Table 8-65 Update Operation (continued)

Update Device	http://bac-lnx-xxx:9101/cp-ws-rest-prov/updateDevice	{
		"macAddress":"1,6,00:00:00:00:12:02",
		"deviceType":"DOCSISModem",
		"sessionId":"7C47EBD1E3436F41869DFBC4B6CA7B7F29A63D26",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING",
		"cos":"TestCOSRestPWS",
		"hostName":"test",
		"domainName":"cisco.com",
		"properties[/fqdn/domain]":"techmahindra.com"
		}
			PUT

Table 8-65 Update Operation (continued)

Update Devices	http://bac-lnx-xxx:9101/cp-ws-rest-prov/updateDevices	{
		"sessionId":"33A145758AF8D8377CF1370F1AF2CBD8CB4FD2ED",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING",
		"domainName":"test231.com",
		"properties":{"/fqdn/domain":"domain.com"},
		"multiDeviceIdList":
		[
		{"macAddress":"1,6,00:00:00:00:12:22","deviceType":"DOCSISModem"},
		
		{"macAddress":"1,6,00:00:00:00:12:29","deviceType":"Computer"}
		
		]
		}
			PUT

Table 8-65 Update Operation (continued)

Update DHCP Criteria	http://bac-lnx-xxx:9101/cp-ws-rest-prov/updateDHCPCriteria	{
		"resourceName":"RESTDHCP",
		"sessionId":"7C47EBD1E3436F41869DFBC4B6CA7B7F29A63D26",
		"properties":{"/fqdn/domain":"cisco197.com"},
		"clientClass":"CiscoTech",
		"excludeSelectionTags":"CiscoTest",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING"
		}
			PUT

Update File	http://bac-lnx-xxx:9101/cp-ws-rest-prov/updateFile	{
		"sessionId":"3A8AE27D1D2F5F7CB58C10E25B7F95E8D5754A77",
		"resourceName":"Test.txt",
		"stream":"Testing Scenario Updated"
		}
			PUT

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/updateFile	{ "sessionId":"3A8AE27D1D2F5F7CB58C10E25B7F95E8D5754A77", "fileName": "cisco.txt", "fileDataBytes": [35, 33, 47, 98, 105, 110, 47]	PUT

Table 8-65 Update Operation (continued)

Update Group	http://bac-lnx-xxx:9101/cp-ws-rest-prov/updateGroup	{
		"resourceName":"Mytestgroup",
		"sessionId":"7C47EBD1E3436F41869DFBC4B6CA7B7F29A63D26",
		"groupType":"mysystem",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING"
		}
			PUT

Table 8-65 Update Operation (continued)

Update Device Type	http://bac-lnx-xxx:9101/cp-ws-rest-prov/updateDeviceType	{
		"rduHost": "pcp-lnx-xxx",
		"rduPort": 49187,
		"username": "admin",
		"password": "admin123",
		"sessionId": "ABCDEF0123456789ABCDEF0123456789ABCDEF01",
		"deviceType": "sampleDeviceTypeName",
		"updateproperties": {
		"FQDNDomain": "domain.com"
		},
		"propertiesToDelete": [
		"FQDNDomain"
		],
		"activationMode": "NO_ACTIVATION",
		"confirmationMode": "NO_CONFIRMATION",
		"publishingMode": "NO_PUBLISHING",
		"asynchronous": false,
		"reliableMode": false,
		"timeout": 30000,
		"stopOnFailure": true,
		"transactionPerItem": false
		}
			PUT

Update RDU
 	
Authorization - BASIC AUTH
	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/RDU/pcp-lnx-xxx.cisco.com	{
		"username": "username",
		"password": "password",
		}
			PUT

Change Operation

Table 8-66 Change Operation

Operation	URL	Request Body	Method

Change Defaults	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/changeDefaults	{
		"rduHost": "pcp-lnx-27",
		"rduPort": 49187,
		"username": "admin",
		"password": "cisco123",
		"sessionId": "ABCDEF0123456789ABCDEF0123456789ABCDEF01",
		"deviceTypeName": "cdttest",
		"propertiesToAdd": {
		"cdttest": "martintesting1"
		},
		"propertiesToDelete": [
		"cdttesting"
		]
		}
			PUT

Table 8-66 Change Operation (continued)

Change RDU Defaults	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/changeRDUDefaults	{
		"sessionId":"3A8AE27D1D2F5F7CB58C10E25B7F95E8D5754A77",
		
		"properties": {
		"/server/rdu/deviceDetection/extensionPoint": "com.cisco.provisioning.cpe.extensions.builtin.detection.DeviceDetector",
		"/rdu/cdt/dhcpFingerPrint/expression": "",
		"/extensionPoint/publishing": "",
		"/extensionPoint/jarFiles": "",
		"/server/crs/failureThresholdPercentage": "0.1",
		"/server/crs/pauseOnFailureThreshold": "false",
		"/server/offline/dpe/delete": "false",
		"/server/scriptCompile": "false",
		"/rdu/cdt/enable": "true",
		"/rdu/cdt/dhcpFingerPrint/filename": "groovyexpression.groovy",
		"/auth/number/sessions/user": "10",
		"/rdu/auth/mode": "RADIUS",
		"/rdu/auth/extension/RADIUS/conf/primary/host" : "10.65.200.149",
		"/rdu/auth/extension/RADIUS/conf/primary/port" : "1812",
		"/rdu/auth/extension/RADIUS/conf/primary/sharedSecret" : "testing123",
		"/rdu/auth/extension/RADIUS/conf/secondary/host" : "",
		"/rdu/auth/extension/RADIUS/conf/secondary/port" : "0",
		"/rdu/auth/extension/RADIUS/conf/secondary/sharedSecret" : "",
		"/rdu/auth/extension/RADIUS/conf/timeout" : "1000",
		"/rdu/auth/extension/RADIUS/conf/retries" : "1"
		},
		"propertiesToDelete": [
		"/server/rdu/configuration/extensionPoint"
		],
		
		"activationMode": "NO_ACTIVATION",
		"confirmationMode": "NO_CONFIRMATION",
		"publishingMode": "NO_PUBLISHING",
		"asynchronous": false,
		"reliableMode": false,
		"timeout": 30000,
		"stopOnFailure": true,
		"transactionPerItem": false
		}
			PUT

Get Operation

Table 8-67 Get Operation

Operation	URL	Request Body	Method

Get COS	http://bac-lnx-xxx:9101/cp-ws-rest-prov/getClassOfService	{
		"sessionId":"3A8AE27D1D2F5F7CB58C10E25B7F95E8D5754A77",
		"resourceName":"sample-gold-docsis",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING"
		}
			POST

Get Device	http://bac-lnx-xxx:9101/cp-ws-rest-prov/getDevice	{
		"sessionId":"3A8AE27D1D2F5F7CB58C10E25B7F95E8D5754A77",
		"macAddress":"1,6,00:00:00:00:12:02",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING"
		}
			POST

Table 8-67 Get Operation (continued)

Get Devices	http://bac-lnx-xxx:9101/cp-ws-rest-prov/getDevices	{
		"sessionId":"57F67427C06F3C3E1DDA8186472ED39C3E9741E1",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"propertyMap":{"includeleaseinfo":"true"},
		"multiDeviceIdList":
		[
		{"macAddress":"1,6,00:00:00:00:12:22","deviceType":"DOCSISModem"},
		
		{"macAddress":"1,6,00:00:00:00:12:29","deviceType":"Computer"}
		]
		}
			POST

Get Device Type	http://bac-lnx-xxx:9101/cp-ws-rest-prov/getDeviceType	{
		"rduHost": "pcp-lnx-xxx",
		"rduPort": 49187,
		"username": "admin",
		"password": "admin123",
		"sessionId": "ABCDEF0123456789ABCDEF0123456789ABCDEF01",
		"deviceType": "sampleDeviceTypeName",
		"activationMode": "NO_ACTIVATION",
		"confirmationMode": "NO_CONFIRMATION",
		"publishingMode": "NO_PUBLISHING",
		"asynchronous": false,
		"reliableMode": false,
		"timeout": 30000,
		"stopOnFailure": true,
		"transactionPerItem": false
		}
			POST

Table 8-67 Get Operation (continued)

Get All Device Types	http://bac-lnx-xxx:9101/cp-ws-rest-prov/getAllDeviceTypes	{
		"rduHost": "pcp-lnx-xxx",
		"rduPort": 49187,
		"username": "admin",
		"password": "admin123",
		"sessionId": "ABCDEF0123456789ABCDEF0123456789ABCDEF01",
		"activationMode": "NO_ACTIVATION",
		"confirmationMode": "NO_CONFIRMATION",
		"publishingMode": "NO_PUBLISHING",
		"asynchronous": false,
		"reliableMode": false,
		"timeout": 30000,
		"stopOnFailure": true,
		"transactionPerItem": false
		}
			POST

Get Behind Device	http://bac-lnx-xxx:9101/cp-ws-rest-prov/getBehindDevice	{
		"sessionId":"3A8AE27D1D2F5F7CB58C10E25B7F95E8D5754A77",
		"macAddress":"1,6,00:00:00:00:12:02",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING"
		}
			POST

Table 8-67 Get Operation (continued)

Get Behind Devices	http://bac-lnx-xxx:9101/cp-ws-rest-prov/getBehindDevices	{
		"sessionId":"1EAFD21803275D48CDEE9243EE31A25D0CC8A9DE",
		"multiDeviceIdList":
		[
		{"macAddress":"1,6,22:00:00:00:00:09"},
		{"macAddress":"1,6,00:00:00:00:12:29"}
		]
		}
			POST

Get DHCP Criteria	http://bac-lnx-xxx:9101/cp-ws-rest-prov/getDHCPCriteria/	{
		"sessionId":"3A8AE27D1D2F5F7CB58C10E25B7F95E8D5754A77",
		"resourceName":"RESTDHCP",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING"
		}
			POST

Table 8-67 Get Operation (continued)

Get DHCP LeaseInfo	http://bac-lnx-xxx:9101/cp-ws-rest-prov/getDHCPLeaseInfo	{
		"sessionId":"3A8AE27D1D2F5F7CB58C10E25B7F95E8D5754A77",
		"resourceName":"10.0.0.17",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING"
		}
			POST

Get File	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/getFile	{
		"sessionId": "68A2AB172E6DFC8AC1F85F33F75CE01B0AC0463D",
		"resourceName": "cisco.txt",
		"retrievedata": "false"
		}
			POST

Get Group	http://bac-lnx-xxx:9101/cp-ws-rest-prov/getGroup	{
		"sessionId":"3A8AE27D1D2F5F7CB58C10E25B7F95E8D5754A77",
		"resourceName":"system-diagnostics",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING"
		}
			POST

Table 8-67 Get Operation (continued)

Get Custom Property	http://bac-lnx-xxx:9101/cp-ws-rest-prov/getCustomProperty	{"sessionId": "E5BE0A6F4F465CACBF2A35ACC1F799930B9972B1"}	POST
Get RDU Details	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/getRDUDetails	{ "sessionId" : "2F2F6AAFEB9B7578CBFF6DE20F3072B680C10094", "hostName" : "pcp-lnx-xxx" }	POST
Get DPE Details	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/getDPEDetails	{ "sessionId" : "DEAE2903AFE16C5B14CB4FDEC382887F944CAC7A", "hostName" : "pcp-lnx-xxx" }	POST
Get RDU Defaults	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/getRDUDefaults	{ "sessionId": "DB7DF171B5D20AC2FB3B1F51FA434ED30B53D9CF" }	POST
Get DPE Defaults	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/getDPEDefaults	{ "sessionId": "DEAE2903AFE16C5B14CB4FDEC382887F944CAC7A", "hostName" : "pcp-lnx-xxx" }	POST
Get Defaults	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/getDefaults	{ "rduHost" : "pcp-lnx-xxx", "rduPort" : "49187", "username" : "username", "password" : "password", "deviceTypeName" : "Erouter" }	POST

Table 8-67 Get Operation (continued)

Get CNR Details	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/getCNRDetails	{
		"sessionId": "DB7DF171B5D20AC2FB3B1F51FA434ED30B53D9CF",
		"hostName": "pcp-lnx-xxx"
		}
			POST

Get Provisioning Group Details	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/getProvGroupDetails	{
		"rduHost" : "pcp-lnx-xxx",
		"rduPort" : "49187",
		"username" : "username",
		"password" : "password",
		"provGroupName": "default"
		}
			POST

Get License Key Data	http://pcp-lnx-xx:9101/cp-ws-rest-prov/getLicenseKeyData	{
		"sessionId": "DB7DF171B5D20AC2FB3B1F51FA434ED30B53D9CF"
		}
			POST

Get Extension Point Settings	http://pcp-lnx-xx:9101/cp-ws-rest-prov/getExtensionPointSettings	{
		"sessionId": "DB7DF171B5D20AC2FB3B1F51FA434ED30B53D9CF"
		}
			POST

Get RDU **Authorization** - BASIC AUTH	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/RDU/pcp-lnx-xxx.cisco.com		GET
Get All RDUs **Authorization** - BASIC AUTH	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/RDU		GET

Table 8-67 Get Operation (continued)

Get All CNRs	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/getAllCNRs	{
		"rduHost": "pcp-lnx-xxx",
		"rduPort": 49187,
		"username": "username",
		"password": "password",
		}
			POST

Get All DPEs	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/getAllDPEs	{
		"rduHost": "pcp-lnx-xxx",
		"rduPort": 49187,
		"username": "username",
		"password": "password",
		"sessionId": "ABCDEF0123456789ABCDEF0123456789ABCDEF01",
		}
			POST

Get All Provisioning Groups	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/getAllProvGroups	{
		"rduHost": "pcp-lnx-xxx",
		"rduPort": 49187,
		"username": "username",
		"password": "password",
		}
			POST

Table 8-68 Delete Operation

Operation	URL	Request Body	Method

Delete COS	http://bac-lnx-xxx:9101/cp-ws-rest-prov/deleteCOS	{
		"resourceName":"TestCOSCM",
		"sessionId":"42C76DAEAA76B1E90DC1F440D9E7E2F8D179C04F"
		}
			Delete

Delete Device	http://bac-lnx-xxx:9101/cp-ws-rest-prov/deleteDevice	{
		"sessionId":"42C76DAEAA76B1E90DC1F440D9E7E2F8D179C04F",
		"macAddress":"1,6,aa:00:00:00:00:01"
		}
			Delete

Table 8-68 Delete Operation (continued)

Delete Device Type	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/deleteDeviceType	{
		"rduHost": "pcp-lnx-xxx",
		"rduPort": 49187,
		"username": "username",
		"password": "password",
		"sessionId": "ABCDEF0123456789ABCDEF0123456789ABCDEF01",
		"resourceName": "CustomDeviceType",
		"activationMode": "NO_ACTIVATION",
		"confirmationMode": "NO_CONFIRMATION",
		"publishingMode": "NO_PUBLISHING",
		"asynchronous": false,
		"reliableMode": false,
		"timeout": 30000,
		"stopOnFailure": true,
		"transactionPerItem": false
		}
			Delete

Delete Devices	http://bac-lnx-xxx:9101/cp-ws-rest-prov/deleteDevices	{
		"sessionId":"55736B06B8877C4E864F37CDA2923558A3CF7A9A",
		"multiDeviceIdList":
		[
		{"macAddress":"1,6,b0:00:00:00:00:02"},
		{"macAddress":"1,6,b0:00:00:00:00:03"}
		]
		}
			Delete

Delete DHCP Criteria	http://bac-lnx-xxx:9101/cp-ws-rest-prov/deleteDHCPCriteria	{
		"resourceName":"RESTDHCP",
		"sessionId":"42C76DAEAA76B1E90DC1F440D9E7E2F8D179C04F"
		}
			Delete

Table 8-68 Delete Operation (continued)

Delete File	http://bac-lnx-xxx:9101/cp-ws-rest-prov/deleteFile	{
		"resourceName":"test.txt",
		"sessionId":"42C76DAEAA76B1E90DC1F440D9E7E2F8D179C04F"
		}
			Deleter

Delete Group	http://bac-lnx-xxx:9101/cp-ws-rest-prov/deleteGroup	{
		"resourceName":"Mytestgroup",
		"sessionId":"42C76DAEAA76B1E90DC1F440D9E7E2F8D179C04F"
		}
			Delete

Delete Custom Property	http://bac-lnx-xxx:9101/cp-ws-rest-prov/deleteCustomProperty	{"sessionId": "E5BE0A6F4F465CACBF2A35ACC1F799930B9972B1", "customProperty":"testCus"}	Delete
Delete RDU **Authorization** - BASIC AUTH	http://pcp-lnx-xxx:9101/cp-ws-rest-prov/RDU/pcp-lnx-xxx.cisco.com		Delete

Table 8-69 Status, Reboot, Unregister, Search Operations

Operation	URL	Request Body	Method

Status Test	http://bac-lnx-xxx:9101/cp-ws-rest-prov/statusTest	{
		"statusTest":"Good"
		}
			POST

Table 8-69 Status, Reboot, Unregister, Search Operations (continued)

Reboot Device	http://bac-lnx-xxx:9101/cp-ws-rest-prov/rebootDevice	{
		"sessionId":"975B4C17880CF31FB606F4BDF8CBCBC11BC4F68A",
		"macAddress":"1,6,b0:00:00:00:00:02",
		"activationMode":"AUTOMATIC",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"true",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING"
		}
			POST

Regen Configs	http://bac-lnx-xxx:9101/cp-ws-rest-prov/regenConfigs	{
		"sessionId":"975B4C17880CF31FB606F4BDF8CBCBC11BC4F68A",
		"maxResults":"10",
		"query":"DeviceSearchByDefaultDHCPCriteria",
		"name":"DOCSISModem",
		"associationType":"REGISTERED"
		}
			POST

Table 8-69 Status, Reboot, Unregister, Search Operations (continued)

Device Operation	http://bac-lnx-xxx:9101/cp-ws-rest-prov/deviceOperation	{
		"sessionId":"2B9F8FE59CBD0F61F043F4F89D140598475E6BB2",
		"multiDeviceIdList":
		[
		{"macAddress":"1,6,00:00:00:00:00:12:22"},
		{"macAddress":"1,6,00:00:00:00:00:12:29"}
		
		],
		"asynchronous":"true",
		"command":"RESET",
		"activationMode":"AUTOMATIC"
		}
			POST

Unregister Device	http://bac-lnx-xxx:9101/cp-ws-rest-prov/unregisterDevice	{
		"sessionId":"5851A4A10D8DDFA65B5111757F8C6647F16E6170",
		"macAddress":"1,6,b0:00:00:00:00:00:02",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING"
		}
			PUT

Table 8-69 Status, Reboot, Unregister, Search Operations (continued)

Unregister Devices	http://bac-lnx-xxx:9101/cp-ws-rest-prov/unregisterDevices	{
		"sessionId":"5851A4A10D8DDFA65B5111757F8C6647F16E6170",
		"multiDeviceIdList":
		[
		{"macAddress":"1,6,00:00:00:00:12:22"},
		{"macAddress":"1,6,00:00:00:00:12:29"}
		
		],
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING"
		}
			PUT

Poll Operation Status	http://bac-lnx-xxx:9101/cp-ws-rest-prov/pollOperationStatus	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"requestID":"Batch:bac-lnx-xxx/xx.xx.xxx.xx:b8c61a6:165404e05bc:8000009b",
		"activationMode":"NO_ACTIVATION",
		"confirmationMode":"NO_CONFIRMATION",
		"asynchronous":"false",
		"reliableMode":"false",
		"timeout":"30000",
		"stopOnFailure":"true",
		"transactionPerItem":"false",
		"publishingMode":"NO_PUBLISHING"
		}
			POST

Table 8-69 Status, Reboot, Unregister, Search Operations (continued)

Search Device	http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"DeviceSearchByDeviceIdPattern",
		"returnParameters":"ALL",
		"maxResults":"10",
		"name":"1,6,22:*",
		"deviceIDPattern":"macaddresspattern"
		}
			POST

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"DeviceSearchByCOS",
		"returnParameters":"ALL",
		"maxResults":"1",
		"associationType":"SELECTED",
		"name":"TestDocsis"
		}
			POST

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"DeviceSearchByDefaultCOS",
		"returnParameters":"ALL",
		"maxResults":"1",
		"associationType":"SELECTED",
		"name":"DOSCISModem"
		}
			POST

Table 8-69 Status, Reboot, Unregister, Search Operations (continued)

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"DeviceSearchByDHCPCriteria",
		"returnParameters":"ALL",
		"maxResults":"1",
		"associationType":"SELECTED",
		"name":"unprovisioned-docsis"
		}
			POST

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"DeviceSearchByDefaultDHCPCriteria",
		"returnParameters":"ALL",
		"maxResults":"1",
		"associationType":"SELECTED",
		"name":"DOSCISModem"
		}
			POST

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"DeviceSearchByDeviceType",
		"returnParameters":"ALL",
		"maxResults":"1",
		"name":"DOSCISModem"
		}
			POST

Table 8-69 Status, Reboot, Unregister, Search Operations (continued)

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"DeviceSearchByGroupName",
		"returnParameters":"ALL",
		"maxResults":"1",
		"associationType":"SELECTED",
		"name":"system-diagnostics"
		}
			POST

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"DeviceSearchByProvGroupName",
		"returnParameters":"ALL",
		"maxResults":"1",
		"associationType":"SELECTED",
		"name":"default"
		}
			POST

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"DeviceSearchByOwnerId",
		"maxResults":"1",
		"name":"cisco"
		}
			POST

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/cosSearch	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"CosSearchByDeviceType",
		"maxResults":"1",
		"deviceType":"DOCSISModem"
		}
			POST

Table 8-69 Status, Reboot, Unregister, Search Operations (continued)

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/dhcpSearch	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"DHCPCriteriaSearch",
		"maxResults":"1",
		"deviceType":"DOCSISModem"
		}
			POST

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/fileSearch	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"FileSearchByFileNamePattern",
		"maxResults":"1",
		"name":"*.cm"
		}
			POST

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/fileSearch	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"FileSearchByFileType",
		"maxResults":"1",
		"fileType":"MIB"
		}
			POST

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/groupSearch	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"GroupSearchByGroupNamePattern",
		"maxResults":"1",
		"groupNamePattern":"*"
		}
			POST

Table 8-69 Status, Reboot, Unregister, Search Operations (continued)

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/groupSearch	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"GroupSearchByRelatedGroup",
		"maxResults":"1",
		"name":"system-diagnostics"
		}
			POST

	http://bac-lnx-xxx:9101/cp-ws-rest-prov/groupSearch	{
		"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
		"query":"GroupSearchByGroupType",
		"maxResults":"1",
		"name":"system"
		}
			POST

Close Session

Table 8-70 Close Session

Operation	URL	Request Body	Method
Close Session	http://bac-lnx-xxx:9101/cp-ws-rest-prov/closeSession	{"sessionId":"9E70A267079DB31B402BCB87941BED9031A3060F"}	POST

PWS Use Cases

This section illustrates some of the common device related operations and RESTful request/response messages. For conciseness, only the request and response messages are depicted.

Note PWS RESTful can communicate only with PCP 6.2 RDU version or above. It is not compatible with RDUs of earlier releases of Prime Cable Provisioning.

This section includes the following use cases:

- Registering a New Device, page 8-73

- Unregistering a Device, page 8-77
- Getting DHCP Lease Information of a Device, page 8-79
- Updating Device Details, page 8-82
- Searching a Device, page 8-87
- Supported Query Elements, page 8-90
- Deleting a Device, page 8-93
- Multiple Devices Operations in a Single Request, page 8-95
- Reboot of Device or Devices, page 8-101

Registering a New Device

For a device to be operational in the home network and to get required access, a subscriber must connect the device to the network and register it. The device could be of type DOCSISModem, PacketCableMTA, CableHomeWanMan, CableHomeWanData, Set Top Box (STB), RPD, eRouter, or computer.

Desired Outcome

Use this workflow to register a device and to bring the device online with the appropriate level of service.

Step 1 Create a connection with the respective RDU by sending a create session RESTful request using POST method. The session RESTful request must contain user and RDU details as shown below:

```
http://pcp-lnx-x:9101/cp-ws-rest-prov/createSession
```

Request Body:

```
{"rduHost": "xx.xx.xxx.xx","rduPort": "49187","username":
"username","password":"password"}
```

RESTful response:

```
sessionId:B886BDD61590296429B530379AF5469CDAF78CF5
```

Step 2 Create class of service required for registering the device using a RESTful request. The supported device types are DOCSISModem, Set Top Box (STB), PacketCableMTA, RPD, eRouter, computer, CableHomeWanMan, and CableHomeWanData.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/addCOS
```

Request Body:

```
{"deviceType":"DocsisModem"
,"sessionId":"D7BD8FADF897EB80365C5092030F187001FCA0F0"
,"name":"TestCOSRestPWS"
,"activationMode":"NO_ACTIVATION"
,"confirmationMode":"NO_CONFIRMATION"
,"asynchronous":"false"
,"reliableMode":"false"
,"timeout":"30000"
,"stopOnFailure":"true"
,"transactionPerItem":"false"
,"publishingMode":"NO_PUBLISHING"
,"properties":{"/cos/docsis/file":"gold.cm"}
}
```

RESTful response:


```
{
    "operationId": "9e61c86e-507e-4b31-b728-0f15302ebaff",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:4c43dbf3:164388870cb:80000112",
                "cmdCodes": [
                    {
                        "index": 0,
                        "code": "CMD_OK"
                    }
                ],
                "code": "CMD_OK",
                "batchCode": "BATCH_COMPLETED"
            }
        ]
    }
}
```


Step 3 Create DHCP criteria required for registering the device using RESTful request.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/addDHCPCriteria
```

Request body:


```
{
"name":"RESTDHCP",
"sessionId":"B6CFC958C3A445A0968754FFBDB9AF670EBE0762",
"properties":{"/fqdn/domain":"cisco167.com"},
"clientClass":"TestCC",
"excludeSelectionTags":"Testest",
"activationMode":"NO_ACTIVATION",
"confirmationMode":"NO_CONFIRMATION",
"asynchronous":"false",
"reliableMode":"false",
"timeout":"30000",
"stopOnFailure":"true",
"transactionPerItem":"false",
"publishingMode":"NO_PUBLISHING"
}
```


RESTful response:


```
{
    "operationId": "665aa915-d637-4b69-beca-9784ee281072",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:4c43dbf3:164388870cb:8000014c",
                "cmdCodes": [
                    {
                        "index": 0,
                        "code": "CMD_OK"
                    }
                ],
```

```
                "code": "CMD_OK",
                "batchCode": "BATCH_COMPLETED"
            }
        ]
    }
}
```

Step 4 Add group using RESTful request.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/addGroup
```

Request body:

```
{
"name":"Mytestgroup",
"sessionId":"B6CFC958C3A445A0968754FFBDB9AF670EBE0762",
"groupType":"system",
"activationMode":"NO_ACTIVATION",
"confirmationMode":"NO_CONFIRMATION",
"asynchronous":"false",
"reliableMode":"false",
"timeout":"30000",
"stopOnFailure":"true",
"transactionPerItem":"false",
"publishingMode":"NO_PUBLISHING"
}
```

RESTful response:

```
{
    "operationId": "f41d073f-6d94-45f4-a7c4-c9dd15efbc0d",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:4c43dbf3:164388870cb:80000150",
                "cmdCodes": [
                    {
                        "index": 0,
                        "code": "CMD_OK"
                    }
                ],
                "code": "CMD_OK",
                "batchCode": "BATCH_COMPLETED"
            }
        ]
    }
}
```

Step 5 Add a Custom property using RESTful request.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/addCustomProperty
```

Request body:

```
{"sessionId":"E5BE0A6F4F465CACBF2A35ACC1F799930B9972B1","customProperty":"testcus","dataTy
pe":"INTEGER","encryptedProperty":"true"}
```

RESTful response:

```
{
    "operationId": "1d8bfe0e-1080-46d9-bca4-ab2eddb89f9e",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:bac-lnx-xxx/127.0.0.1:4a68f274:1669a553ad7:8000000c",
                "cmdCodes": [
                    {
                        "index": 0,
                        "code": "CMD_OK",
                        "message": null
                    }
                ],
                "code": "CMD_OK",
                "batchCode": "BATCH_COMPLETED",
                "message": null
            }
        ]
    }
}
```


Step 6 Use the device ID such as MAC address or DUID or FQDN and the device type for adding a new device to the RESTful request. You can assign a group to the new device.

The add device RESTful request also passes the subscriber's information, Class of Service, DHCP Criteria, Hostname and domain to Prime Cable Provisioning, which then registers the subscriber's device such as modem and computer.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/addDevice
```

Request body:

```
{
"deviceType":"DOCSISModem",
"sessionId":"42C76DAEAA76B1E90DC1F440D9E7E2F8D179C04F",
"activationMode":"NO_ACTIVATION"
,"confirmationMode":"NO_CONFIRMATION"
,"asynchronous":"false"
,"reliableMode":"false"
,"timeout":"30000"
,"stopOnFailure":"true"
,"transactionPerItem":"false"
,"publishingMode":"NO_PUBLISHING"
,"cos":"unprovisioned-docsis"
,"hostName":"TestHost"
,"domainName":"Newtestdomain.com"
,"properties":{"/fqdn/domain":"domain.com"},
"deviceIdSet":{"macAddress":"1,6,00:00:00:00:12:02"}
}
```

RESTful response


```
{
    "operationId": "6b7385b8-fb4f-4da7-98f3-6c2da1bb03c8",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:4c43dbf3:164388870cb:8000011e",
                "cmdCodes": [
```

```
                    {
                        "index": 0,
                        "code": "CMD_OK"
                    }
                ],
                "code": "CMD_OK",
                "batchCode": "BATCH_COMPLETED"
            }
        ]
    }
}
```

Step 7 Provision the device by connecting it to the network. Prime Cable Provisioning provides the device its registered service level.

The device is now a provisioned device with access to the appropriate level of service.

Step 8 Close the open sessions.

Note Idle sessions are automatically closed after 15 minutes.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/closeSession
```

Request body:

```
{"sessionId":"9E70A267079DB31B402BCB87941BED9031A3060F"}
```

RESTful response:

```
{
    "operationId": "07591855-50bd-4e8d-9345-3202eae89331",
    "code": "SUCCESS",
    "message": null,
    "subStatus": null
}
```

Unregistering a Device

A device can be unregistered from Prime Cable Provisioning using the unregister RESTful request. This deletes the records from the Prime Cable Provisioning and subscriber gets the appropriate service level.

Desired Outcome

Unregistering a device removes the device details from Prime Cable Provisioning and gets the unprovisioned level of service.

Use this workflow to unregister a device and to bring the device online with the appropriate level of service.

Step 1 Create a connection with the respective RDU by sending a create session RESTful request using POST method. The session RESTful request must contain user and RDU details as shown below.

```
http://pcp-lnx-x:9101/cp-ws-rest-prov/createSession
```

Request Body:

```
{"rduHost": "xx.xx.xxx.xx","rduPort": "49187","username":
"username","password":"password"}
```

RESTful response:

```
sessionId:B886BDD61590296429B530379AF5469CDAF78CF5
```

Step 2 Use the device ID such as MAC address or DUID for unregistering a device. Unregister the device using the following RESTful request.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/unregisterDevice
```

Request body:

```
{
"sessionId":"5851A4A10D8DDFA65B5111757F8C6647F16E6170",
"macAddress":"1,6,b0:00:00:00:00:02",
"activationMode":"NO_ACTIVATION",
"confirmationMode":"NO_CONFIRMATION",
"asynchronous":"false",
"reliableMode":"false",
"timeout":"30000",
"stopOnFailure":"true",
"transactionPerItem":"false",
"publishingMode":"NO_PUBLISHING"
}
```

RESTful response

```
{
    "operationId": "c4f37622-c1d7-4cfa-b024-82c1f49a982a",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:705306be:1645c87ed08:8000005e",
                "cmdCodes": [
                    {
                        "index": 0,
                        "code": "CMD_OK"
                    }
                ],
                "code": "CMD_OK",
                "batchCode": "BATCH_COMPLETED"
            }
        ]
    }
}
```

The device is now a unregistered and unprovisioned device with access to the appropriate level of service.

Step 3 Close the open sessions.

Note Idle sessions are automatically closed after 15 minutes.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/closeSession
```

Request body:

```
{"sessionId":"9E70A267079DB31B402BCB87941BED9031A3060F"}
```

RESTful response:

```
{
    "operationId": "07591855-50bd-4e8d-9345-3202eae89331",
    "code": "SUCCESS",
    "message": null,
    "subStatus": null
}
```

Getting DHCP Lease Information of a Device

A registered or unregistered device is assigned an IP address when it is provisioned. This IP address is used for collecting lease information such as, network register server state, and list of provisioning group in this lease.

Desired Outcome

Lists the details of the lease information from the Prime Cable Provisioning for the provisioned devices.

Use this workflow to collect the lease information of a device.

Step 1 Use an existing session or create a connection with the respective RDU by sending a create session RESTful request using POST method. The session RESTful request must contain user and RDU details as shown below.

```
http://pcp-lnx-x:9101/cp-ws-rest-prov/createSession
```

Request Body:

```
{"rduHost": "xx.xx.xxx.xx","rduPort": "49187","username":
"username","password":"password"}
```

RESTful response:

```
sessionId:B886BDD61590296429B530379AF5469CDAF78CF5
```

Step 2 Use the IP address along with its provisioning group name or just the IP address of the provisioned devices in the RESTful request.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/getDHCPLeaseInfo
```

Request body:

```
{
"sessionId":"3A8AE27D1D2F5F7CB58C10E25B7F95E8D5754A77",
"resourceName":"10.0.0.17",
"activationMode":"NO_ACTIVATION",
"confirmationMode":"NO_CONFIRMATION",
"asynchronous":"false",
"reliableMode":"false",
"timeout":"30000",
"stopOnFailure":"true",
```

```
"transactionPerItem":"false",
"publishingMode":"NO_PUBLISHING"
}
```

RESTful response:

```
{
    "operationStatus": {
        "operationId": "048db9a1-34d4-463d-ac46-26bb4730003c",
        "code": "SUCCESS",
        "message": "Operation successful",
        "subStatus": {
            "statusList": [
                {
                    "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:705306be:1645c87ed08:80000037",
                    "cmdCodes": [
                        {
                            "index": 0,
                            "code": "CMD_OK"
                        }
                    ],
                    "code": "CMD_OK",
                    "batchCode": "BATCH_COMPLETED"
                }
            ]
        }
    },
    "leaseResults": [
        {
            "device": {
                "id": null,
                "deviceType": "DOCSISModem",
                "subscriberId": null,
                "cos": null,
                "dhcpCriteria": null,
                "hostName": null,
                "domainName": null,
                "properties": {
                    "/provisioning/classOfService/selected": "unprovisioned-docsis",
                    "/provisioning/properties/selected": "{/docsis/version=3.1}",
                    "/provisioning/access": "DEFAULT",
                    "/provisioning/domain": "RootDomain",
                    "/provisioning/explanation": "Because the device is not registered",
                    "/generic/oidRevisionNumber": "1008806320825958504-1530560966000",
                    "/provisioning/provGroup": "default",
                    "/provisioning/isRegistered": "false",
                    "/provisioning/dhcpCriteria/selected": "unprovisioned-docsis",
                    "/provisioning/isInRequiredProvGroup": "true",
                    "/provisioning/reason": "NOT_REGISTERED",
                    "/node": "[]",
                    "/discoveredData/raw/dhcpv4": "{REQUEST={dhcp-class-identifier=AIC
Echo,docsis3.1:, chaddr=b0:00:00:00:00:01, relay-agent-circuit-id=01:04:80:01:03:ef,
vendor-encapsulated-options=04:0c:62:30:30:30:30:30:30:30:30:30:31:05:06:76:33:2e:32:2e
:31:06:06:76:31:2e:30:2e:32:07:07:42:4f:4f:54:31:2e:30:08:06:62:30:30:30:30:30:09:07:58:6d
:61:6e:32:30:30:0a:09:43:49:53:43:4f:20:62:61:63, dhcp-message-type=01,
v-i-vendor-opts=00:00:11:8b:84:01:01:02:05:7f:05:7d:01:01:01:02:01:02:03:01:01:04:01:01:05
:01:01:06:01:01:07:01:0f:08:01:10:09:01:00:0a:01:01:0b:01:18:0c:01:01:0d:02:01:00:0e:02:01
:00:0f:01:01:10:04:00:00:00:01:14:01:00:1d:01:03:1e:01:20:1f:01:10:20:01:10:21:01:01:22:01
:01:24:01:00:25:01:00:26:01:00:27:01:01:2f:04:00:00:00:00:30:02:00:00:31:01:00:32:01:00:33
:01:00:34:02:00:00:35:02:00:00:36:01:01:37:01:00:38:01:00:39:01:00, giaddr=0a:00:00:01,
relay-agent-remote-id=02:06:b0:00:00:00:00:01,
client-id-created-from-mac-address=00:00:00:00, htype=01,
```

```
relay-agent-info=01:04:80:01:03:ef:02:06:b0:00:00:00:00:01:09:0b:00:00:11:8b:06:01:04:01:0
2:03:01, client-id=ff:00:00:00:00:00:03:00:01:b0:00:00:00:00:01, hlen=06,
dhcp-parameter-request-list={01,03,06,07,0c,0f,33,36,04,02,43,42,7d}}}",
                    "/network/docsisVersion": "3.1",
                    "/provisioning/properties/detected": null,
                    "/provisioning/isBehindRequiredDevice": "true"
                },
                "discoveredData": {
                    "dhcpv4RequestData": {
                        "dhcp-class-identifier": "AIC Echo,docsis3.1:",
                        "chaddr": "b0:00:00:00:00:01",
                        "relay-agent-circuit-id": "01:04:80:01:03:ef",
                        "vendor-encapsulated-options": "(device-serial-number 4
b00000000001), (hardware-version-number 5 v3.2.1), (software-version-number 6 v1.0.2),
(boot-rom-version 7 BOOT1.0), (vendor-oui 8 b00000), (model-number 9 Xman200),
(vendor-name 10 CISCO bac)",
                        "dhcp-message-type": "1",
                        "v-i-vendor-opts": "enterprise-id 4491, (oro 1 2),
(modem-capabilities 5
05:7d:01:01:01:02:01:02:03:01:01:04:01:01:05:01:01:06:01:01:07:01:0f:08:01:10:09:01:00:0a:
01:01:0b:01:18:0c:01:01:0d:02:01:00:0e:02:01:00:0f:01:01:10:04:00:00:00:01:14:01:00:1d:01:
03:1e:01:20:1f:01:10:20:01:10:21:01:01:22:01:01:24:01:00:25:01:00:26:01:00:27:01:01:2f:04:
00:00:00:00:30:02:00:00:31:01:00:32:01:00:33:01:00:34:02:00:00:35:02:00:00:36:01:01:37:01:
00:38:01:00:39:01:00)",
                        "giaddr": "10.0.0.1",
                        "relay-agent-remote-id": "02:06:b0:00:00:00:00:01",
                        "client-id-created-from-mac-address": "0",
                        "htype": "01",
                        "relay-agent-info": "(circuit-id 1 80:01:03:ef), (remote-id 2
b0:00:00:00:00:01), (v-i-vendor-opts 9 enterprise-id 4491, (cmts-capabilities 1
(docsis-version 1 03:01)))",
                        "client-id": "ff:00:00:00:00:00:03:00:01:b0:00:00:00:00:01",
                        "hlen": "06",
                        "dhcp-parameter-request-list":
"{1,3,6,7,12,15,51,54,4,2,67,66,125}"
                    },
                    "dhcpv6RequestData": null,
                    "dhcpv6RelayRequestData": null
                },
                "leaseData": {
                    "dhcpv4LeaseQueryData": {
                        "relay-agent-circuit-id": "80:01:03:ef",
                        "client-last-transaction-time": "5214",
                        "relay-agent-remote-id": "b0:00:00:00:00:01",
                        "client-mac-address": "1,6,b0:00:00:00:00:01",
                        "client-ipaddress": "10.0.0.12",
                        "dhcp-lease-time": "599586",
                        "subnet-mask": "255.255.255.0",
                        "giaddr": "xx.xx.xxx.xx",
                        "client-id": "ff:00:00:00:00:00:03:00:01:b0:00:00:00:00:01",
                        "routers": "10.0.0.1",
                        "dhcp-server-identifier": "xx.xx.xxx.xx"
                    },
                    "dhcpv6LeaseQueryData": null
                },
                "groups": null,
                "deviceIdSet": {
                    "ids": [
                        {
                            "type": null,
                            "id": "1,6,b0:00:00:00:00:01"
                        }
                    ],
                    "macAddress": {
```

```
                        "type": null,
                        "id": "1,6,b0:00:00:00:00:01"
                    },
                    "duid": null,
                    "fqdn": null,
                    "deviceIDList": [
                        {
                            "keyType": "MAC_ADDRESS",
                            "deviceId": "1,6,b0:00:00:00:00:01"
                        },
                        {
                            "keyType": "MAC_ADDRESS",
                            "deviceId": "1,6,b0:00:00:00:00:01"
                        }
                    ],
                    "firstDeviceID": {
                        "keyType": "MAC_ADDRESS",
                        "deviceId": "1,6,b0:00:00:00:00:01"
                    }
                }
            },
            "v4": true,
            "v6": false,
            "active": true,
            "partialAnswer": false
        }
    ]
}
```


Step 3 Close the open sessions.

Note Idle sessions are automatically closed after 15 minutes.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/closeSession
```

Request body:


```
{"sessionId":"9E70A267079DB31B402BCB87941BED9031A3060F"}
```


RESTful response:


```
{
    "operationId": "07591855-50bd-4e8d-9345-3202eae89331",
    "code": "SUCCESS",
    "message": null,
    "subStatus": null
}
```


Updating Device Details

You can update the device's class of service, DHCP criteria, assign or unassign a group, and change or add properties. You can update any or all of this information as part of single updateDevice operation.

Embedded devices are not supported in Prime Cable Provisioning. The following workflow updates a single device. Before updating the device a new class of service and new dhcp criteria are added.

Desired Outcome

Prime Cable Provisioning regenerates the configuration file after updating the device with the required changes. The device then gets re-provisioned with updated level of service.

Use this workflow to update a device's details and provision it with the updated level of service.

Step 1 Create a connection with the respective RDU by sending a session RESTful request using POST method. The session RESTful request must contain user and RDU details as shown below.

```
http://pcp-lnx-x:9101/cp-ws-rest-prov/createSession
```

Request Body:

```
{"rduHost": "xx.xx.xxx.xx","rduPort": "49187","username":
"username","password":"password"}
```

RESTful response:

```
sessionId:B886BDD61590296429B530379AF5469CDAF78CF5
```

Step 2 Add a new class of service for the device using the following RESTful request:

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/addCOS
```

Request Body:

```
{"deviceType":"DocsisModem"
,"sessionId":"D7BD8FADF897EB80365C5092030F187001FCA0F0"
,"name":"TestCOSRestPWS"
,"activationMode":"NO_ACTIVATION"
,"confirmationMode":"NO_CONFIRMATION"
,"asynchronous":"false"
,"reliableMode":"false"
,"timeout":"30000"
,"stopOnFailure":"true"
,"transactionPerItem":"false"
,"publishingMode":"NO_PUBLISHING"
,"properties":{"/cos/docsis/file":"gold.cm"}
}
```

RESTful response:

```
{
    "operationId": "9e61c86e-507e-4b31-b728-0f15302ebaff",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:4c43dbf3:164388870cb:80000112",
                "cmdCodes": [
                    {
                        "index": 0,
                        "code": "CMD_OK"
                    }
```

```
                ],
                "code": "CMD_OK",
                "batchCode": "BATCH_COMPLETED"
            }
        ]
    }
}
```

Step 3 Add a new DHCP criteria for the device using the following RESTful request:

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/addDHCPCriteria
```

Request body:

```
{
"name":"RESTDHCP",
"sessionId":"B6CFC958C3A445A0968754FFBDB9AF670EBE0762",
"properties":{"/fqdn/domain":"cisco167.com"},
"clientClass":"TestCC",
"excludeSelectionTags":"Testest",
"activationMode":"NO_ACTIVATION",
"confirmationMode":"NO_CONFIRMATION",
"asynchronous":"false",
"reliableMode":"false",
"timeout":"30000",
"stopOnFailure":"true",
"transactionPerItem":"false",
"publishingMode":"NO_PUBLISHING"
}
```

RESTful response:

```
{
    "operationId": "665aa915-d637-4b69-beca-9784ee281072",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:4c43dbf3:164388870cb:8000014c",
                "cmdCodes": [
                    {
                        "index": 0,
                        "code": "CMD_OK"
                    }
                ],
                "code": "CMD_OK",
                "batchCode": "BATCH_COMPLETED"
            }
        ]
    }
}
```

Step 4 If you want to assign a group that is not yet created, create it using the following RESTful request.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/addGroup
```

Request body:

```
{
```

```
"name":"Mytestgroup",
"sessionId":"B6CFC958C3A445A0968754FFBDB9AF670EBE0762",
"groupType":"system",
"activationMode":"NO_ACTIVATION",
"confirmationMode":"NO_CONFIRMATION",
"asynchronous":"false",
"reliableMode":"false",
"timeout":"30000",
"stopOnFailure":"true",
"transactionPerItem":"false",
"publishingMode":"NO_PUBLISHING"
}
```

RESTful response:

```
{
    "operationId": "f41d073f-6d94-45f4-a7c4-c9dd15efbc0d",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:4c43dbf3:164388870cb:80000150",
                "cmdCodes": [
                    {
                        "index": 0,
                        "code": "CMD_OK"
                    }
                ],
                "code": "CMD_OK",
                "batchCode": "BATCH_COMPLETED"
            }
        ]
    }
}
```

Step 5 Use the device ID such as MAC address or DUID or FQDN and the device type to update the device details. You can update the subscriber's details as well as assign or unassign the device to a group by using the following RESTful request.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/updateDevice
```

Request body:

```
{
    "macAddress":"1,6,00:00:00:00:12:02",
"deviceType":"DOCSISModem",
"sessionId":"7C47EBD1E3436F41869DFBC4B6CA7B7F29A63D26",
"activationMode":"NO_ACTIVATION",
"confirmationMode":"NO_CONFIRMATION",
"asynchronous":"false",
"reliableMode":"false",
"timeout":"30000",
"stopOnFailure":"true",
"transactionPerItem":"false",
"publishingMode":"NO_PUBLISHING",
"cos":"TestCOSRestPWS",
"hostName":"test",
"domainName":"cisco.com",
"properties[/fqdn/domain]":"techmahindra.com"
}
```

The device gets a regenerated configuration and is provisioned with updated service level.

RESTful response:


```
{
    "operationId": "c0692ec3-53fd-4ff9-acd2-c9b2dab37a20",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:738933c7:16457b49b5e:80000017",
                "cmdCodes": [
                    {
                        "index": 0,
                        "code": "CMD_OK"
                    },
                    {
                        "index": 1,
                        "code": "CMD_OK"
                    },
                    {
                        "index": 2,
                        "code": "CMD_OK"
                    },
                    {
                        "index": 3,
                        "code": "CMD_OK"
                    }
                ],
                "code": "BATCH_COMPLETED",
                "batchCode": "BATCH_COMPLETED"
            }
        ]
    }
}
```


Step 6 Close the open sessions.

Note Idle sessions are automatically closed after 15 minutes.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/closeSession
```

Request body:


```
{"sessionId":"9E70A267079DB31B402BCB87941BED9031A3060F"}
```


RESTful response:


```
{
    "operationId": "07591855-50bd-4e8d-9345-3202eae89331",
    "code": "SUCCESS",
    "message": null,
    "subStatus": null
}
```

Searching a Device

The subscriber has devices such as DOCSIS, MTA, DPoE and computer in Prime Cable Provisioning. These devices can be searched from the Prime Cable Provisioning and get the results as a list in RESTful response.

Desired Outcome

Searches device(s) from Prime Cable Provisioning based on specific search criteria.

Use this workflow to search a device and get the list that is generated based on the search criteria.

Step 1 Create a connection with the respective RDU by sending a session RESTful request using POST method. The session RESTful request must contain user and RDU details as shown below.

```
http://pcp-lnx-x:9101/cp-ws-rest-prov/createSession
```

Request Body:

```
{"rduHost": "xx.xx.xxx.xx","rduPort": "49187","username":
"username","password":"password"}
```

RESTful response:

```
sessionId:B886BDD61590296429B530379AF5469CDAF78CF5
```

Step 2 Use the query to create a search criteria.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices
```

Request body:

```
{
"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
"query":"DeviceSearchByDeviceIdPattern",
"returnParameters":"ALL",
"maxResults":"10",
"name":"1,6,22:*",
"deviceIDPattern":"macaddresspattern"
}
```

Note For the first query, the start element must not be included.

For all the supported search requests Search.query and Search.maxResults elements are required. Search.start is used to support paging across large search results. The first search operation must not include Search.start element. Its absence indicates that a new search is being initiated. A response to a search operation contains a SearchResult. SearchResult contains a *next* element which itself is a Search object. The entire content of the next element (query, start, size) should be used to page to the next batch of search results. The value of the start element is used by RDU's internal paging mechanism. Hence, from second request onwards, Search.start is required to page over the results. At the end of the search process, an empty SearchResult where SearchResult.size equals zero will be returned.

Step 3 Ensure that the response contains the next query request.

```
{
```

```
    "operationStatus": {
        "operationId": "4f98e5a2-c420-4c7a-8c62-73647e3557dc",
        "code": "SUCCESS",
        "message": "Operation successful",
        "subStatus": {
            "statusList": [
                {
                    "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:3715dcc8:1647b882056:8000001a",
                    "cmdCodes": [
                        {
                            "index": 0,
                            "code": "CMD_OK"
                        }
                    ],
                    "code": "CMD_OK",
                    "batchCode": "BATCH_COMPLETED"
                }
            ]
        }
    },
    "results": [
        {
            "id": null,
            "deviceType": "DOCSISModem",
            "subscriberId": null,
            "cos": null,
            "dhcpCriteria": null,
            "hostName": null,
            "domainName": null,
            "properties": {
                "/provisioning/classOfService/selected": "unprovisioned-docsis",
                "/provisioning/properties/selected": "{/docsis/version=3.1}",
                "/provisioning/access": "DEFAULT",
                "/provisioning/domain": "RootDomain",
                "/provisioning/explanation": "Because the device is not registered",
                "/generic/oidRevisionNumber": "1008806320825958501-1531081017000",
                "/provisioning/provGroup": "default",
                "/provisioning/isRegistered": "false",
                "/provisioning/dhcpCriteria/selected": "unprovisioned-docsis",
                "/provisioning/isInRequiredProvGroup": "true",
                "/provisioning/reason": "NOT_REGISTERED",
                "/node": "[]",
                "/discoveredData/raw/dhcpv4": "{REQUEST={dhcp-class-identifier=AIC
Echo,docsis3.1:, chaddr=ab:00:00:00:00:01, relay-agent-circuit-id=01:04:80:01:03:ef,
vendor-encapsulated-options=04:0c:61:62:30:30:30:30:30:30:30:30:31:05:06:76:33:2e:32:2e
:31:06:06:76:31:2e:30:2e:32:07:07:42:4f:4f:54:31:2e:30:08:06:61:62:30:30:30:30:09:07:58:6d
:61:6e:32:30:30:0a:09:43:49:53:43:4f:20:62:61:63, dhcp-message-type=01,
v-i-vendor-opts=00:00:11:8b:84:01:01:02:05:7f:05:7d:01:01:01:02:01:02:03:01:01:04:01:01:05
:01:01:06:01:01:07:01:0f:08:01:10:09:01:00:0a:01:01:0b:01:18:0c:01:01:0d:02:01:00:0e:02:01
:00:0f:01:01:10:04:00:00:00:01:14:01:00:1d:01:03:1e:01:20:1f:01:10:20:01:10:21:01:01:22:01
:01:24:01:00:25:01:00:26:01:00:27:01:01:2f:04:00:00:00:00:30:02:00:00:31:01:00:32:01:00:33
:01:00:34:02:00:00:35:02:00:00:36:01:01:37:01:00:38:01:00:39:01:00, giaddr=0a:00:00:01,
relay-agent-remote-id=02:06:ab:00:00:00:00:01,
client-id-created-from-mac-address=00:00:00:00, htype=01,
relay-agent-info=01:04:80:01:03:ef:02:06:ab:00:00:00:00:01:09:0b:00:00:11:8b:06:01:04:01:0
2:03:01, client-id=ff:00:00:00:00:00:03:00:01:ab:00:00:00:00:01, hlen=06,
dhcp-parameter-request-list={01,03,06,07,0c,0f,33,36,04,02,43,42,7d}}}",
                "/network/docsisVersion": "3.1",
                "/provisioning/properties/detected": null,
                "/provisioning/isBehindRequiredDevice": "true"
            },
            "discoveredData": {
                "dhcpv4RequestData": {
                    "dhcp-class-identifier": "AIC Echo,docsis3.1:",
```

```
                    "chaddr": "ab:00:00:00:00:01",
                    "relay-agent-circuit-id": "01:04:80:01:03:ef",
                   "vendor-encapsulated-options": "(device-serial-number 4 ab0000000001),
(hardware-version-number 5 v3.2.1), (software-version-number 6 v1.0.2), (boot-rom-version
7 BOOT1.0), (vendor-oui 8 ab0000), (model-number 9 Xman200), (vendor-name 10 CISCO bac)",
                    "dhcp-message-type": "1",
                   "v-i-vendor-opts": "enterprise-id 4491, (oro 1 2), (modem-capabilities
5
05:7d:01:01:01:02:01:02:03:01:01:04:01:01:05:01:01:06:01:01:07:01:0f:08:01:10:09:01:00:0a:
01:01:0b:01:18:0c:01:01:0d:02:01:00:0e:02:01:00:0f:01:01:10:04:00:00:00:01:14:01:00:1d:01:
03:1e:01:20:1f:01:10:20:01:10:21:01:01:22:01:01:24:01:00:25:01:00:26:01:00:27:01:01:2f:04:
00:00:00:00:30:02:00:00:31:01:00:32:01:00:33:01:00:34:02:00:00:35:02:00:00:36:01:01:37:01:
00:38:01:00:39:01:00)",
                    "giaddr": "10.0.0.1",
                    "relay-agent-remote-id": "02:06:ab:00:00:00:00:01",
                    "client-id-created-from-mac-address": "0",
                    "htype": "01",
                    "relay-agent-info": "(circuit-id 1 80:01:03:ef), (remote-id 2
ab:00:00:00:00:01), (v-i-vendor-opts 9 enterprise-id 4491, (cmts-capabilities 1
(docsis-version 1 03:01)))",
                    "client-id": "ff:00:00:00:00:00:03:00:01:ab:00:00:00:00:01",
                    "hlen": "06",
                    "dhcp-parameter-request-list": "{1,3,6,7,12,15,51,54,4,2,67,66,125}"
                },
                "dhcpv6RequestData": null,
                "dhcpv6RelayRequestData": null
            },
            "leaseData": null,
            "groups": null,
            "deviceIdSet": {
                "ids": [
                    {
                        "type": null,
                        "id": "1,6,ab:00:00:00:00:01"
                    }
                ],
                "macAddress": {
                    "type": null,
                    "id": "1,6,ab:00:00:00:00:01"
                },
                "duid": null,
                "fqdn": null,
                "deviceIDList": [
                    {
                        "keyType": "MAC_ADDRESS",
                        "deviceId": "1,6,ab:00:00:00:00:01"
                    },
                    {
                        "keyType": "MAC_ADDRESS",
                        "deviceId": "1,6,ab:00:00:00:00:01"
                    }
                ],
                "firstDeviceID": {
                    "keyType": "MAC_ADDRESS",
                    "deviceId": "1,6,ab:00:00:00:00:01"
                }
            }
        }
    ],
    "size": 1,
    "next": {
        "query": null,
        "start": "1,6,ab:00:00:00:00:01",
        "QueryValue": null,
```

```
        "maxResults": 10,
        "propertyFilter": null
    }
}
```

Step 4 Use the next element contents for the next search query criteria.

Step 5 Repeat step 4 and step 5 till you receive the search response size as zero.

All the devices are available based on the search request.

Step 6 Close the open sessions.

Note Idle sessions are automatically closed after 15 minutes.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/closeSession
```

Request body:

```
{"sessionId":"9E70A267079DB31B402BCB87941BED9031A3060F"}
```

RESTful response:

```
{
    "operationId": "07591855-50bd-4e8d-9345-3202eae89331",
    "code": "SUCCESS",
    "message": null,
    "subStatus": null
}
```

Note You can also run search query based on different object types such as, device, Class of Service, DHCP Criteria, file and group. A sample search query pattern with these object types is shown below:

Supported Query Elements

The supported search elements in Prime Cable Provisioning 7.2.1 are:

Searching basic device details of a registered device with class of service:

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices
```

Request body:

```
{
"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
"query":"DeviceSearchByCOS",
"returnParameters":"ALL",
"maxResults":"1",
"associationType":"SELECTED",
"name":"TestDocsis"
}
```

Searching basic device details of a registered device with DHCP criteria:

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices
```

Request body:

```
{
"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
"query":"DeviceSearchByDHCPCriteria",
"returnParameters":"ALL",
"maxResults":"1",
"associationType":"SELECTED",
"name":"unprovisioned-docsis"
}
```

Searching basic device details of a registered device with default CoS device type:

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices
```

Request body:

```
{
"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
"query":"CosSearchByDeviceType",
"maxResults":"1",
"deviceType":"DOCSISModem"
}
```

Searching basic device details of a registered device with default DHCP criteria device type:

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices
```

Request body:

```
{
"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
"query":"DeviceSearchByDefaultDHCPCriteria",
"returnParameters":"ALL",
"maxResults":"1",
"associationType":"SELECTED",
"name":"DOSCISModem"
}
```

Searching all the device details of a device with device type:

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices
```

Request body:

```
{
"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
"query":"DeviceSearchByDeviceType",
"returnParameters":"ALL",
"maxResults":"1",
"name":"DOSCISModem"
}
```

Searching all the device details of a device with group name:

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices
```

Request body:

```
{
"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
"query":"DeviceSearchByGroupName",
"returnParameters":"ALL",
"maxResults":"1",
"associationType":"SELECTED",
"name":"system-diagnostics"
}
```

Search the entire device by ownerId:

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices
```

Request body:

```
{
"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
"query":"DeviceSearchByOwnerId",
"maxResults":"1",
"name":"cisco"
}
```

Searching all class of service with device type:

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices
```

Request body:

```
{
"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
"query":"CosSearchByDeviceType",
"maxResults":"1",
"deviceType":"DOCSISModem"
}
```

Searching all DHCP criteria:

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices
```

Request body:

```
{
"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
"query":"DHCPCriteriaSearch",
"maxResults":"1",
"deviceType":"DOCSISModem"
}
```

Searching all files with file type:

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices
```

Request body:

```
{
"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
```

```
"query":"FileSearchByFileType",
"maxResults":"1",
"fileType":"MIB"
}
```

Searching all files with a file name pattern:

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices
```

Request body:

```
{
"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
"query":"FileSearchByFileNamePattern",
"maxResults":"1",
"name":"*.cm"
}
```

Searching all groups with group type:

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices
```

Request body:

```
{
"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
"query":"GroupSearchByGroupType",
"maxResults":"1",
"name":"system"
}
```

Searching all groups with a group name pattern:

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/searchDevices
```

Request body:

```
{
"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
"query":"GroupSearchByGroupNamePattern",
"maxResults":"1",
"groupNamePattern":"*"
}
```

Deleting a Device

The existing devices in Prime Cable Provisioning can be deleted using the delete RESTful request.

Desired Outcome

Deleting a device removes the device details from the Prime Cable Provisioning.

Use this workflow to delete a device.

Step 1 Create a connection with the respective RDU by sending a session RESTful request using POST method. The session RESTful request must contain user and RDU details as shown below.

```
http://pcp-lnx-x:9101/cp-ws-rest-prov/createSession
```

Request Body:

```
{"rduHost": "xx.xx.xxx.xx","rduPort": "49187","username":
"username","password":"password"}
```

RESTful response:

```
sessionId:B886BDD61590296429B530379AF5469CDAF78CF5
```

Use the device ID such as MAC address or DUID to delete the device.

Step 2 Delete the device using the following RESTful request.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/deleteDevice
```

Request body:

```
{
    "sessionId":"42C76DAEAA76B1E90DC1F440D9E7E2F8D179C04F",
    "macAddress":"1,6,aa:00:00:00:00:01"
}
```

RESTful response:

```
{
    "operationId": "b6160005-ba83-4b81-bf07-eee3f8586efe",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:4c43dbf3:164388870cb:800000b9",
                "cmdCodes": [
                    {
                        "index": 0,
                        "code": "CMD_OK"
                    }
                ],
                "code": "CMD_OK",
                "batchCode": "BATCH_COMPLETED"
            }
        ]
    }
}
```

Step 3 Close the open sessions.

Note Idle sessions are automatically closed after 15 minutes.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/closeSession
```

Request body:

```
{"sessionId":"9E70A267079DB31B402BCB87941BED9031A3060F"}
```

RESTful response:

```
{
```

```
    "operationId": "07591855-50bd-4e8d-9345-3202eae89331",
    "code": "SUCCESS",
    "message": null,
    "subStatus": null
}
```

The device record is deleted from Prime Cable Provisioning.

Multiple Devices Operations in a Single Request

Multiple devices in Prime Cable Provisioning can be added, retrieved, updated, or deleted using a single RESTful request.

Desired Outcome

Use this workflow to add, retreive, update, and delete multiple devices.

Step 1 Create a connection with the respective RDU by sending a session RESTful request using POST method. The session RESTful request must contain user and RDU details as shown below.

```
http://pcp-lnx-x:9101/cp-ws-rest-prov/createSession
```

Request Body:

```
{"rduHost": "xx.xx.xxx.xx","rduPort": "49187","username":
"username","password":"password"}
```

RESTful response:

```
sessionId:B886BDD61590296429B530379AF5469CDAF78CF5
```

Step 2 Add five devices using a single RESTful request.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/addDevices
```

Request body:

```
{
"sessionId":"F61D528E364BC96F6956FF9A73FFB1ABA7C2371E",
"activationMode":"NO_ACTIVATION",
"confirmationMode":"NO_CONFIRMATION",
"asynchronous":"false",
"reliableMode":"false",
"timeout":"30000",
"stopOnFailure":"true",
"transactionPerItem":"false",
"publishingMode":"NO_PUBLISHING",
"multiDeviceList":
[
    { "deviceIdSet": {"macAddress":"1,6,00:00:00:00:12:b2"},"deviceType":"DOCSISModem"},
    { "deviceIdSet": {"macAddress":"1,6,00:00:00:00:12:b1"},"deviceType":"Computer"}

]
 }
```

RESTful response:

```
{
    "operationId": "7a02904c-38a6-4c80-9b04-16ee37056d71",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:4c43dbf3:164388870cb:80000144",
                "cmdCodes": [
                    {
                        "index": 0,
                        "code": "CMD_OK"
                    },
                    {
                        "index": 1,
                        "code": "CMD_OK"
                    }
                ],
                "code": "BATCH_COMPLETED",
                "batchCode": "BATCH_COMPLETED"
            }
        ]
    }
}
```

Step 3 Using a single RESTful request, you can get more than one device.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/getDevices
```

Request body:

```
{
"sessionId":"57F67427C06F3C3E1DDA8186472ED39C3E9741E1",
"activationMode":"NO_ACTIVATION",
"confirmationMode":"NO_CONFIRMATION",
"asynchronous":"false",
"reliableMode":"false",
"timeout":"30000",
"stopOnFailure":"true",
"propertyMap":{"includeleaseinfo":"true"},
"multiDeviceIdList":
[
{"macAddress":"1,6,00:00:00:00:12:22","deviceType":"DOCSISModem"},

{"macAddress":"1,6,00:00:00:00:12:29","deviceType":"Computer"}


]
}
```

RESTful response:

```
[
    {
        "devices": [
            {
                "id": null,
                "deviceType": null,
                "subscriberId": null,
                "cos": null,
                "dhcpCriteria": null,
```

```
        "hostName": null,
        "domainName": null,
        "properties": {
            "/provisioning/provGroup": "default"
        },
        "discoveredData": null,
        "leaseData": {
            "dhcpv4LeaseQueryData": {
                "relay-agent-circuit-id": "80:01:03:ef",
                "client-last-transaction-time": "2542",
                "relay-agent-remote-id": "b0:00:00:00:00:01",
                "client-mac-address": "1,6,b0:00:00:00:00:01",
                "client-ipaddress": "10.0.0.12",
                "dhcp-lease-time": "602258",
                "subnet-mask": "255.255.255.0",
                "giaddr": "xx.xx.xxx.xx",
                "client-id": "ff:00:00:00:00:00:03:00:01:b0:00:00:00:00:01",
                "routers": "10.0.0.1",
                "dhcp-server-identifier": "xx.xx.xxx.xx"
            },
            "dhcpv6LeaseQueryData": null
        },
        "groups": null,
        "deviceIdSet": {
            "ids": [
                {
                    "type": null,
                    "id": "1,6,b0:00:00:00:00:01"
                }
            ],
            "macAddress": {
                "type": null,
                "id": "1,6,b0:00:00:00:00:01"
            },
            "duid": null,
            "fqdn": null,
            "deviceIDList": [
                {
                    "keyType": "MAC_ADDRESS",
                    "deviceId": "1,6,b0:00:00:00:00:01"
                },
                {
                    "keyType": "MAC_ADDRESS",
                    "deviceId": "1,6,b0:00:00:00:00:01"
                }
            ],
            "firstDeviceID": {
                "keyType": "MAC_ADDRESS",
                "deviceId": "1,6,b0:00:00:00:00:01"
            }
        }
    },
    {
        "id": null,
        "deviceType": null,
        "subscriberId": null,
        "cos": null,
        "dhcpCriteria": null,
        "hostName": null,
        "domainName": null,
        "properties": {
            "/provisioning/provGroup": "default"
        },
        "discoveredData": null,
```

```
                    "leaseData": {
                        "dhcpv4LeaseQueryData": {
                            "relay-agent-circuit-id": "80:01:03:ef",
                            "client-last-transaction-time": "2577",
                            "relay-agent-remote-id": "a0:00:00:00:00:01",
                            "client-mac-address": "1,6,aa:00:00:00:00:01",
                            "client-ipaddress": "10.0.0.6",
                            "dhcp-lease-time": "602223",
                            "subnet-mask": "255.255.255.0",
                            "giaddr": "xx.xx.xxx.xx",
                            "client-id": "ff:00:00:00:00:00:03:00:01:a0:00:00:00:00:00:01",
                            "routers": "10.0.0.1",
                            "dhcp-server-identifier": "xx.xx.xxx.xx"
                        },
                        "dhcpv6LeaseQueryData": null
                    },
                    "groups": null,
                    "deviceIdSet": {
                        "ids": [
                            {
                                "type": null,
                                "id": "1,6,aa:00:00:00:00:01"
                            }
                        ],
                        "macAddress": {
                            "type": null,
                            "id": "1,6,aa:00:00:00:00:01"
                        },
                        "duid": null,
                        "fqdn": null,
                        "deviceIDList": [
                            {
                                "keyType": "MAC_ADDRESS",
                                "deviceId": "1,6,aa:00:00:00:00:01"
                            },
                            {
                                "keyType": "MAC_ADDRESS",
                                "deviceId": "1,6,aa:00:00:00:00:01"
                            }
                        ],
                        "firstDeviceID": {
                            "keyType": "MAC_ADDRESS",
                            "deviceId": "1,6,aa:00:00:00:00:01"
                        }
                    }
                }
            ],
            "operationStatus": {
                "operationId": "de0b1eba-3cea-43e0-a25a-af53a72e1eb0",
                "code": "SUCCESS",
                "message": "Operation successful",
                "subStatus": {
                    "statusList": [
                        {
                            "txId":
    "Batch:pcp-lnx-x/xx.xx.xxx.xx:705306be:1645c87ed08:8000000f",
                            "cmdCodes": [
                                {
                                    "index": 0,
                                    "code": "CMD_OK"
                                },
                                {
                                    "index": 1,
                                    "code": "CMD_OK"
```

```
                            }
                        ],
                        "code": "BATCH_COMPLETED",
                        "batchCode": "BATCH_COMPLETED"
                    }
                ]
            }
        }
    }
]
```

Step 4 Using a single RESTful request, you can update more than one device.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/updateDevices
```

Request body:

```
{
"sessionId":"33A145758AF8D8377CF1370F1AF2CBD8CB4FD2ED",
"activationMode":"NO_ACTIVATION",
"confirmationMode":"NO_CONFIRMATION",
"asynchronous":"false",
"reliableMode":"false",
"timeout":"30000",
"stopOnFailure":"true",
"transactionPerItem":"false",
"publishingMode":"NO_PUBLISHING",
"domainName":"test231.com",
"properties":{"/fqdn/domain":"domain.com"},
"multiDeviceIdList":
[
{"macAddress":"1,6,00:00:00:00:12:22","deviceType":"DOCSISModem"},

{"macAddress":"1,6,00:00:00:00:12:29","deviceType":"Computer"}


]
}
```

RESTful response:

```
{
    "operationId": "07b1ae46-714f-40e6-a1c4-250001709312",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:3a0658b2:16459e4d58b:8000001a",
                "cmdCodes": [
                    {
                        "index": 0,
                        "code": "CMD_OK"
                    },
                    {
                        "index": 1,
                        "code": "CMD_OK"
                    },
                    {
                        "index": 2,
                        "code": "CMD_OK"
                    },
                    {
                        "index": 3,
```

```
                        "code": "CMD_OK"
                    },
                    {
                        "index": 4,
                        "code": "CMD_OK"
                    },
                    {
                        "index": 5,
                        "code": "CMD_OK"
                    }
                ],
                "code": "BATCH_COMPLETED",
                "batchCode": "BATCH_COMPLETED"
            }
        ]
    }
}
```

Step 5 Using a single RESTful request, you can delete more than one device.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/deleteDevice
```

Request body:

```
{
    "sessionId":"42C76DAEAA76B1E90DC1F440D9E7E2F8D179C04F",
    "macAddress":"1,6,aa:00:00:00:00:01"
}
```

RESTful response:

```
{
    "operationId": "9627c062-24d5-451a-9b3a-bb96bc83298c",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:4c43dbf3:164388870cb:800000da",
                "cmdCodes": [
                    {
                        "index": 0,
                        "code": "CMD_OK"
                    },
                    {
                        "index": 1,
                        "code": "CMD_OK"
                    }
                ],
                "code": "BATCH_COMPLETED",
                "batchCode": "BATCH_COMPLETED"
            }
        ]
    }
}
```

Step 6 Close the open session.

Note Idle sessions are automatically closed after 15 minutes.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/closeSession
```

Request body:

```
{"sessionId":"9E70A267079DB31B402BCB87941BED9031A3060F"}
```

RESTful response:

```
{
    "operationId": "07591855-50bd-4e8d-9345-3202eae89331",
    "code": "SUCCESS",
    "message": null,
    "subStatus": null
}
```

Reboot of Device or Devices

The devices in Prime Cable Provisioning can be rebooted using the reboot RESTful request.

Desired Outcome

Use this workflow to reboot a device or multiple devices.

Step 1 Create a connection with the respective RDU by sending a session RESTful request using POST method. The session RESTful request must contain user and RDU details as shown below.

```
http://pcp-lnx-x:9101/cp-ws-rest-prov/createSession
```

Request Body:

```
{"rduHost": "xx.xx.xxx.xx","rduPort": "49187","username":
"username","password":"password"}
```

RESTful response:

```
sessionId:B886BDD61590296429B530379AF5469CDAF78CF5
```

Use the device ID such as MAC address or DUID to reboot the device.

Step 2 Reboot the device using the following RESTful request.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/rebootDevice
```

Request body:

```
{
    "sessionId":"975B4C17880CF31FB606F4BDF8CBCBC11BC4F68A",
"macAddress":"1,6,b0:00:00:00:00:02",
"activationMode":"AUTOMATIC",
"confirmationMode":"NO_CONFIRMATION",
"asynchronous":"true",
"reliableMode":"false",
"timeout":"30000",
"stopOnFailure":"true",
```

```
"transactionPerItem":"false",
"publishingMode":"NO_PUBLISHING"
}
```

RESTful response:

```
{
    "operationId": "dcbe2462-7811-4581-8b79-4560fb0ac95d",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:705306be:1645c87ed08:80000057",
                "cmdCodes": [],
                "code": null,
                "batchCode": null
            }
        ]
    }
}
```

The device record is initiated for reboot operation in Prime Cable Provisioning.

Step 3 Use the transaction ID (txId) to retrieve the operation status from the Prime Cable Provisioning.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/pollOperationStatus
```

Request body:

```
{
"sessionId":"B34F984E600B25357CD0615BB0A8A27619E5E783",
"requestID":"Batch:bac-lnx-xxx/xx.xx.xxx.xx:b8c61a6:165404e05bc:8000009b",
"activationMode":"NO_ACTIVATION",
"confirmationMode":"NO_CONFIRMATION",
"asynchronous":"false",
"reliableMode":"false",
"timeout":"30000",
"stopOnFailure":"true",
"transactionPerItem":"false",
"publishingMode":"NO_PUBLISHING"
}
```

RESTful response:

```
{
    "operationId": "2e647a75-b5f3-4604-ac08-7f972422974a",
    "code": "SUCCESS",
    "message": "Operation successful",
    "subStatus": {
        "statusList": [
            {
                "txId": "Batch:pcp-lnx-x/xx.xx.xxx.xx:705306be:1645c87ed08:80000057",
                "cmdCodes": [
                    {
                        "index": 0,
                        "code": "CMD_OK"
                    }
                ],
                "code": "CMD_OK",
                "batchCode": "BATCH_COMPLETED"
            }
```

```
        ]
    }
}
```

Step 4 Close the open session.

Note Idle sessions are automatically closed after 15 minutes.

```
http://bac-lnx-xxx:9101/cp-ws-rest-prov/closeSession
```

Request body:

```
{"sessionId":"9E70A267079DB31B402BCB87941BED9031A3060F"}
```

RESTful response:

```
{
    "operationId": "07591855-50bd-4e8d-9345-3202eae89331",
    "code": "SUCCESS",
    "message": null,
    "subStatus": null
}
```

A

alert — A syslog or SNMP message notifying an operator or administrator of a problem.

API — Application programming interface. Specification of function-call conventions that defines an interface to a service.

audit logs — A log file containing a summary of the major changes in the RDU database. This includes the changes to system defaults, technology defaults, and classes of service.

auto configuration server (ACS) — A server that provisions a device or a collection of devices. In Prime Cable Provisioning, ACS refers to the BAC server, and in some instances, the DPE.

B

broadband — Transmission system that multiplexes multiple independent signals onto one cable. In Telecommunications terminology; any channel having a bandwidth greater than a voice-grade channel (4 kHz). In LAN terminology; a co-axial cable on which analog signaling is used.

Cisco Prime Cable Provisioning — An integrated solution for managing and provisioning broadband home networks. Prime Cable Provisioning is a scalable product capable of supporting millions of devices.

Business Support Systems(BSS) — Components that service providers use to run business operations. The roles of a BSS in a service provider network include managing products, customers, revenue, and orders.

C

caching — Form of replication in which information learned during a previous transaction is used to process later transactions.

cipher suites — A set of cryptographic algorithms that the SSL module requires to perform key exchange, authentication, and Message Authentication Code.

customer premises equipment (CPE) — Terminating equipments, such as telephones, computers, and modems, supplied and installed at a customer location.

D

device provisioning engine (DPE) Distributed servers that cache device instructions. DPEs automatically synchronize with the RDU to obtain the latest instructions, and provide Prime Cable Provisioning scalability and redundancy.

F

fully qualified domain name (FQDN) FQDN is the full name of a system, rather than just its hostname. For example, cisco is a hostname and www.cisco.com is an FQDN.

H

HTTPS See Secure Sockets Layer and Transport Layer Security.

I

instruction generation service (IGS) The process of generating instructions at the RDU, for devices defined by a search criteria, and distributing these instructions to the DPE, which then caches the instructions. The instructions inform the DPE the actions to be performed on the CPE, which may include configuration, firmware upgrade, or other operations.

IP address An IP address is a 32-bit number that identifies each sender or receiver of information that is sent in packets across the Internet.

N

network address translation (NAT) Mechanism for reducing the need for globally unique IP addresses. NAT allows an organization with addresses that are not globally unique to connect to the Internet by translating those addresses into globally routeable address space.

network administrator Person responsible for operation, maintenance, and management of a network. See also network operator.

network operator Person who routinely monitors and controls a network, performing such tasks as reviewing and responding to alarms, monitoring throughput, configuring new circuits, and resolving problems. See also network administrator.

Network Time Protocol (NTP) A protocol designed to synchronize server clocks over a network.

O

Operations Support Systems(OSS) Computer systems used by telecommunication providers, dealing with telecom network, customers and support processes.

P

provisioning API A series of Prime Cable Provisioning functions that programs can use to make the operating system perform various functions.

provisioning groups Groupings of devices with a defined set of associated DPE servers, based on either network topology or geography.

publishing Publishing provides provisioning information to an external datastore in real time. Publishing plug-ins must be developed to write data to a datastore.

PACE Provisioning API Command Engine.

R

redundancy In internetworking, the duplication of devices, services, or connections so that, in the event of a failure, the redundant devices, services, or connections can perform the work of those that failed.

regional distribution unit (RDU) The RDU is the primary server in the Prime Cable Provisioning provisioning system. It manages generation of device instructions, processes all API requests, and manages the Prime Cable Provisioning system.

S

Secure Sockets Layer (SSL) A protocol for transmitting private documents via the Internet. SSL uses a cryptographic system that uses two keys to encrypt data: a public key known to everyone and a private or secret key known only to the recipient of the message. URLs that require an SSL connection start with https: instead of http:. Prime Cable Provisioning supports SSLv3.

See Transport Layer Security.

shared secret A character string used to provide secure communication between two servers or devices.

T

template files XML files that contain configuration or firmware rules for devices.

Transport Layer Security (TLS) A protocol that guarantees privacy and data integrity between client/server applications communicating over the Internet. Prime Cable Provisioning supports TLSv1.

See Secure Sockets Layer.

V

Voice over IP (VoIP) Mechanism to make telephone calls and send faxes over IP-based data networks with a suitable quality of service (QoS) and superior cost savings.

W

watchdog agent A watchdog agent is a daemon process that is used to monitor, stop, start, and restart Prime Cable Provisioning component processes such as the RDU, JRun, and SNMP agent.

INDEX

A

ACS
    definition 9-1
Adding 7-31
alert messages
    alerts, definition 9-1
API 2-3, 3-1, 4-1
API, definition 9-1
API use cases
    See use cases
atomic 4-2
audit logs, definition 9-1
autoconfiguration server
    See ACS

C

caching, definition 9-1
cipher suites
    definition 9-1
Class Of Service Operations 8-20
Compile 6-2
connection 3-2
container 4-11
customer premises equipment, definition 9-1

D

deadlock 5-3
Deleting 8-53
Device 1-2
Device Provisioning Operations 8-7
DeviceType Operations 8-17
DHCP Criteria Operations 8-22
DOCSIS 8-48
documentation
    related documents i-vii
DPE (Device Provisioning Engine)
    definition 9-2
DPoE 8-48

E

Error Handling 8-2
event reliability 5-3
executes 4-2
Execution Options 8-3

F

File Operations 8-24
FQDN
    definition 9-2

G

Generic 8-19
Generic Device Operation 8-19
Getting 8-38
Group Operations 8-27

I

instructions
    generation, definition 9-2

IP address
definition 9-2

L

lost events 5-3

M

MTA 8-48
Multiple 8-55

N

NAT, definition 9-2
network address translation
See NAT
Network Time Protocol, definition 9-2
non 3-2

O

Options
execution 8-3

P

pollOperation Status 8-29
Provisioning 1-2, 8-1
provisioning 4-1
provisioning API, definition 9-3
provisioning groups
definition 9-3
provisioning use cases, API 7-3
Provisioning Web Services 8-1
ProvServiceException 8-19
publishing, definition 9-3
PWS 8-1, 8-2
Concepts 8-2
Data Types 8-2
data types 8-2
execution options 8-3
multiple device operation 8-2
Operations 8-6
Overview 8-1
QueryType 8-4
Session Management 8-2
single device operation 8-2
Tomcat 8-1
Transactionality 8-2
use cases 8-30
PWS Operation
Class of Service 8-20
DeviceType 8-17
DHCP Criteria 8-22
File 8-24
Generic 8-19
Group 8-27
pollOperation 8-29
Search 8-30
PWS Operations 8-6
device provisioning 8-7
session 8-6
PWS Use Cases 8-30

Q

QueryType 8-4

R

RDU (Regional Distribution Unit)
definition 9-3
reconnects 3-2
redundancy, definition 9-3
Registering 8-32

S

Searching 8-48
Search Operation 8-30
Secure Sockets Layer
  See SSL
Session Management 8-2
Session Operations 8-6
shared secret
  definition 9-3
SNMP
  cloning on RDU, DPE
    PacketCable eMTA (use case) 7-44
SSL
  definition 9-3
Supported 8-51
System 1-2

T

TCP 3-2
template files, developing
  template file definition 9-4
TLS
  definition 9-4
  See SSL
Transactionality 8-2
Transport Layer Security
  See TLS

U

Unregistering 8-36
Updating 8-43
use cases
  about 7-1
  adding
    new computer behind a modem with NAT 7-32
    new computer in fixed standard mode 7-10
    second computer in promiscuous mode 7-31
  bulk provisioning modems in promiscuous mode 7-25
  CableHome with firewall configuration 7-54
  creating API client 7-3
  disabling subscriber 7-13
  getting detailed device information 7-35
  incremental provisioning of PacketCable eMTA 7-45
  logging
    batch completions using events 7-35
    device deletions using events 7-33
  modifying an existing modem 7-17
  monitoring RDU connection using events 7-34
  moving device to another DHCP scope 7-32
  optimistic locking 7-49
  preprovisioning
    CableHome WAN-MAN 7-53
    DOCSIS modems with configuration files 7-48
    first-time activation in promiscuous mode 7-27
    modems and self-provisioned computers 7-15
    PacketCable eMTA 7-42
  replacing existing modem 7-29
  retrieving capabilities for CableHome WAN-MAN 7-56
  searching for devices using class of service 7-41
  searching for devices using vendor prefix 7-41
  searching using default class of service 7-40
  self-provisioning
    CableHome WAN-MAN 7-57
    first-time activation in promiscuous mode 7-22
    first-time activation with NAT 7-31
    modem, computer in fixed standard mode 7-7
  SNMP cloning on PacketCable eMTA 7-44
  subscriber bandwidth, temporarily throttling 7-51
  unregistering, deleting subscriber device 7-18

V

VoIP, definition 9-4

W

watchdog process
agent, definition 9-4