



---

# Cisco Smart Update SDK Programmer's Guide

Version 1.0

Fox Smart, Inc.  
© 2002-2008 Fox Smart, Inc.  
All rights reserved.

This document may only be used for the purpose of understanding, evaluating or using the Cisco Smart Update toolkit. The document may be reproduced, stored, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, only for this purpose. The document must be maintained in whole and it must contain the Fox Smart copyright notice. This document must not be used for any other purpose without prior written permission of Fox Smart, Inc.

The Fox Smart logo is a trademark of Fox Smart, Inc.

Use of the Fox Smart logo for commercial purposes without the prior written consent of Fox Smart may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Fox Smart retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications that use the Cisco Smart Update toolkit.

Every effort has been made to ensure that the information in

this document is accurate. However, this document could include technical inaccuracies or typographical errors. Fox Smart is not responsible for these inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. Fox Smart may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any reference in this document to Web sites other than those provided by Fox Smart are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this Fox Smart product and use of those Web sites is at your own risk.

Fox Smart, Inc.  
20006 Hickman Way  
Poolesville, MD 20837

Fox Smart, Cisco Smart Update, and their respective logos are trademarks of Fox Smart, Inc. in the United States and other countries.

Cisco is a registered trademark of Cisco Systems, Inc.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc.

Even though Fox Smart has reviewed this document, FOX SMART MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL FOX SMART BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Fox Smart dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

# Contents

<b>1.</b>	<b>Introduction</b>	<b>7</b>
<hr/>		
1.1	Purpose	7
1.2	Audience	7
1.3	Getting Help	7
1.4	How To Use This Guide	8
<b>2.</b>	<b>Installation and Configuration</b>	<b>9</b>
<hr/>		
2.1	Dependencies	9
2.1.1	JDK Dependencies	9
2.1.2	Platform Dependencies	9
2.1.3	Third-party Library Dependencies	9
2.1.4	SSH Dependencies	10
2.2	Distribution Packaging	10
2.2.1	Toolkit Distribution	10
2.2.2	Toolkit Licenses	11
2.3	Installation	11
2.3.1	Extract the CSU SDK Distribution	11
2.3.2	Installing the Java Runtime Environment (JRE)	11
2.3.3	Adding the JRE to the system PATH	12
2.3.4	Setting up the CLASSPATH	12
2.3.5	Applying the License File	12
2.3.6	Logging Options	13
2.3.7	Running an Application Program	14

2.3.8	Running CSU SDK Behind a Firewall	15
2.3.9	Common Exceptions	15
2.3.10	runCsuApp.bat	16

### **3. CSU SDK Example Applications 18**

---

3.1	DisplayLicenseInfo	18
3.2	VerifyLicenseForHostname	19
3.3	VerifyRouterAuthentication	19
3.4	GetRouterPrompt	20
3.5	GetRouterTime	20
3.6	GetRouterVersionInfo	20
3.7	ReadRouterOriginalConfiguration	21
3.8	ReadRouterGeneratedConfiguration	22
3.9	VerifyRouterConfigs	23
3.10	BackupRouterConfigs	24
3.11	SetRouterTime	24
3.12	SendRouterCommand	25
3.13	UpdateRouterBanners	25
3.14	UpdateStaticRoute	26

### **4. CiscoRouter Class 27**

---

4.1	Instantiating a CiscoRouter Object	27
4.2	CiscoRouter Communication Options	27
4.2.1	protocolHandlers	28
4.2.2	defaultNoDataReadDelay	28
4.2.3	defaultProtocolTimeout	28
4.2.4	updateNvRamConfigFlag	29
4.2.5	stopUpdatesOnConfigErrorFlag	29
4.2.6	rollbackOnConfigErrorFlag	29
4.3	Logging In and Logging Out	29

4.3.1	Logging In .....	30
4.3.2	Logging Out .....	31
4.4	Sending Commands .....	31
4.4.1	Sending Generic Commands .....	31
4.4.2	Sending Built In Commands .....	31
4.5	Reading and Updating Configurations .....	32
4.5.1	Reading a Running Configuration .....	32
4.5.2	Writing a Configuration .....	32
<b>5.</b>	<b>Working with Configurations</b> .....	<b>34</b>
<hr/>		
5.1	CiscoRouterConfig Class .....	34
5.2	Configuration Components .....	34
5.2.1	Manipulating Components .....	35
5.2.2	Component Interfaces .....	36
<b>6.</b>	<b>Update Static Route Example Explained</b> .....	<b>37</b>
<hr/>		
6.1	Working With Command Line Parameters .....	37
6.2	Reading the Running Configuration .....	38
6.3	Get or Create an IpRouteComponent .....	38
6.4	Working with IpRouteEntry Objects .....	39
6.5	Writing the Configuration Back to the Router .....	40
<b>7.</b>	<b>Parsing, Validating, and Generating IOS</b> .....	<b>41</b>
<hr/>		
7.1	CiscoRouterConfigComponent Abstract Methods .....	41
7.2	Parsing IOS Configuration Lines .....	42
7.3	Working With Lines .....	42
7.4	Working With Tokens .....	42
7.5	Converting Tokens .....	43
7.6	Token Validation .....	44

7.7	<b>InvalidConfigurationException Generation</b>	<b>44</b>
<b>8.</b>	<b>Adding New Components</b>	<b>45</b>
<hr/>		
8.1	<b>Create A New Component</b>	<b>45</b>
8.2	<b>Add Component Properties</b>	<b>45</b>
8.3	<b>Add Required Component Methods</b>	<b>46</b>
8.4	<b>Integrate Component into the SDK</b>	<b>46</b>
8.4.1	<b>Package Integration</b>	<b>46</b>
8.4.2	<b>Class Integration</b>	<b>47</b>
8.5	<b>DatabitsComponent Example</b>	<b>48</b>
8.6	<b>Implementing CompositeComponent</b>	<b>50</b>
8.7	<b>Implementing MultiInstanceComponent</b>	<b>51</b>
8.8	<b>Implementing MultiLineComponent</b>	<b>52</b>
8.9	<b>Implementing IndelibleComponent</b>	<b>52</b>
8.10	<b>Advanced Methods To Override</b>	<b>52</b>

# 1. Introduction

---

Welcome to the Cisco Smart Update Software Development Kit (SDK) Programmer's Guide. This chapter provides an introduction to the Cisco Smart Update SDK including its purpose and the document audience.

## 1.1 Purpose

---

Cisco Smart Update (CSU) Software Development Kit (SDK) is a Java application programmer interface (API) that enables a software application to communicate directly with a Cisco Router to issue commands, read the configuration, make changes to the configuration, and apply those changes back to the router. The router configuration is represented in an object oriented class hierarchy that directly mirrors the router configuration hierarchy. The purpose of this document is to explain how to use the CSU SDK API.

## 1.2 Audience

---

The primary audience of this guide is a Java application programmer who wishes to interface directly with a Cisco router using a programmatic interface. This guide assumes the reader is familiar with the Java programming language, router concepts, and the Cisco router Internetwork Operating System (IOS) configuration hierarchy.

## 1.3 Getting Help

---

Although this document intends to provide the reader with all the necessary knowledge about how to use the CSU SDK, additional help can be obtained from the Fox Smart web site (<http://www.FoxSmart.com>) or by contacting Fox Smart directly at:

Fox Smart, Inc.  
20006 Hickman Way  
Poolesville, MD 20837  
[support@foxsmart.com](mailto:support@foxsmart.com) (E-mail)

## 1.4 How To Use This Guide

---

This document is divided into chapters that describe the various aspects of the SDK. The following list briefly describes each chapter:

- Chapter 1 (this chapter) provides an overview of this guide.
- Chapter 2 explains how to install and configure the SDK.
- Chapter 3 describes the sample applications that come with the SDK.
- Chapter 4 discusses the `CiscoRouter` class and its main API's.
- Chapter 5 discusses the `CiscoRouterConfig` class and how to work with a router configuration.
- Chapter 6 discusses how the Update Static Router example program works.
- Chapter 7 discusses how to parse, validate, and generate IOS configurations.
- Chapter 8 describes how to extend the SDK by adding custom components.

## 2. Installation and Configuration

---

This chapter provides information on how to install and setup the Cisco Smart Update SDK. It also covers any product dependencies.

### 2.1 Dependencies

---

This section covers any required dependencies to use the Cisco Smart Update SDK.

#### 2.1.1 JDK Dependencies

---

The Cisco Smart Update SDK requires a minimum of *JDK 1.5* Standard Edition. The JDK can be downloaded from the Sun Microsystems web site at <http://java.sun.com/javase/downloads>.

#### 2.1.2 Platform Dependencies

---

The Cisco Smart Update SDK is written in 100% pure Java and does not use any platform specific Java Native Interface (JNI) code. As such, it should run properly on any platform that supports the Java Runtime environment. Nonetheless, it is recommended that CSU SDK and your application be properly tested on the desired targeted platform to ensure no issues are found.

Note that SSH support is available using either a Java library or a platform specific implementation. See “SSH Dependencies” below for more information.

#### 2.1.3 Third-party Library Dependencies

---

The Cisco Smart Update SDK requires the following third-party libraries to be in the classpath:

- commons-lang.jar (<http://commons.apache.org/lang>)
- commons-collections.jar (<http://commons.apache.org/collections>)
- commons-logging.jar (<http://commons.apache.org/logging>)
- commons-codec.jar (<http://commons.apache.org/codecs>)
- dom4j.jar (<http://www.dom4j.org>)

Customers may choose to download these third-party libraries themselves or use the ones that are packaged with the product.

## 2.1.4 SSH Dependencies

---

CSU SDK provides SSH support by using one of two possible options:

1. **plink 0.60**: plink is a free Putty platform specific command line SSH implementation. plink is automatically distributed with the CSU SDK for the Linux, MacOS, Solaris, and Windows operating systems. It is the default SSH implementation and is automatically used by CSU SDK. For more information on Putty and plink, please consult their web site at <http://www.chiark.greenend.org.uk/~sgtatham/putty/>.
2. **MindTerm 3.2**: CSU SDK comes with a MindTerm protocol handler that enables it to use the 100% pure Java MindTerm SSH implementation. Due to licensing restrictions, the MindTerm application JAR is not distributed with CSU SDK. Nonetheless, it can be furnished by the end user if a pure Java solution is desired. MindTerm is available free of charge for personal and limited commercial use for up to 25 users. For more information on MindTerm, please consult their web site at <http://www.apagate.com/index/products/mindterm>.

## 2.2 Distribution Packaging

---

This section covers how the Cisco Smart Update SDK and requires licenses are packaged and distributed.

### 2.2.1 Toolkit Distribution

---

Cisco Smart Update SDK customers should have already received a copy of the CSU product. If you do not already have a copy of the SDK and wish to receive a copy for evaluation purposes, please contact Fox Smart to receive an evaluation copy of the product.

The Cisco Smart Update SDK comes in four possible distribution packages:

- **csu-1.0.jar** is the main Cisco Smart Update SDK JAR (Java Archive) distribution. It contains the core SDK functionality. Customers who use this distribution will be required to furnish their own dependent third-party JAR libraries as listed previously.
- **csu-1.0-with-dependencies.zip** contains the main Cisco Smart Update SDK JAR distribution as well as required dependent third-party libraries. The following are the third-party library versions that are included in this distribution: **commons-lang.jar 2.3**, **commons-collections.jar 3.2**, **commons-logging.jar 1.1**, **commons-codec.jar 1.3**, **dom4j.jar 1.6.1**, and **plink 0.60**. Other versions of these libraries may also work if needed, however, customers should check with Fox Smart if a specific version of a library is supported.
- **csu-1.0-docs.zip** contains the Cisco Smart Update SDK documentation (e.g. Javadoc). No libraries are included in this distribution.
- **csu-1.0-all.zip** contains the entire Cisco Smart Update SDK distribution including everything in the other distributions (i.e. the toolkit library, third-party libraries, and documentation).

No license files are included in any of the CSU SDK distributions.

## 2.2.2 Toolkit Licenses

---

The Cisco Smart Update SDK requires a separate valid license file to activate the product. This license file is not included in any of the CSU SDK distributions and the product will not function without a valid license file.

Cisco Smart Update SDK customers should have already received a permanent license file. If you do not have a license file and wish to evaluate the product, a temporary 30 day license file can be obtained by contacting Fox Smart.

## 2.3 Installation

---

This section describes how to install the CSU SDK and run an example program that reads the IOS configuration from a router and displays a generated version of the configuration to the standard output.

The following sub-sections provide the details in setting up the CSU SDK and running an example application. They can all be followed in order to get the toolkit up and running. However, an alternate approach for Windows users is to follow the instructions in the sub-section called "runCsuApp.bat".

### 2.3.1 Extract the CSU SDK Distribution

---

Once the Cisco Smart Update SDK distribution has been obtained, the first step is to extract the contents of the distribution archive into a directory. The following describes how this can be done for the various CSU SDK distributions:

- **csu-1.0.jar**: This is the main SDK JAR file that needs to be placed in the application classpath. If this distribution is used, all dependent third-party JAR files must be obtained separately and also placed in the classpath.
- **csu-1.0-with-dependencies.zip**: Extract the ZIP file into a directory. The archive contains a top level directory called "*csu-1.0*" where all distribution files will be located. The main SDK and an examples archive are located in the "*dist*" sub-directory. The dependent third-party libraries are located in the "*lib*" sub-directory. All JAR files must be individually placed in the application classpath. Environment specific binaries are located in the "*bin*" sub-directory.
- **csu-1.0-docs.zip**: This distribution contains documentation only and does not contain the SDK itself.
- **csu-1.0-all.zip**: This distribution can be installed in the same manner as "csu-1.0-with-dependencies.zip" above.

As an example, extract the contents of **csu-1.0-all.zip** into the `C:\` directory on a Windows platform.

### 2.3.2 Installing the Java Runtime Environment (JRE)

---

To run an application program (e.g. the CSU SDK Examples) that uses the CSU SDK, a Java Runtime Environment must be installed on the target platform. If the JRE has not yet been installed, it first needs to be downloaded from Sun Microsystems using the following URL:

<http://java.sun.com/javase/downloads/>

Note that *JRE 1.5* or later must be selected. Once the JRE has been downloaded, install it into any directory on your target machine. For example, if *JRE 1.6* is downloaded, it can be installed into the following directory:

```
C:\Program Files\Java\jre1.6.0_02
```

### 2.3.3 Adding the JRE to the system PATH

---

Once the JRE is installed on the target system, the JRE's bin directory needs to be added to the system PATH so the JRE executables can be found. The following is an example of how the JRE can be added to the system PATH on a Windows machine:

```
SET PATH=%PATH%;C:\Program Files\Java\jre1.6.0_02\bin
```

### 2.3.4 Setting up the CLASSPATH

---

To run any application that uses the CSU SDK, the classpath must be setup to point to the CSU distribution, all the third party libraries, and optionally, the CSU SDK examples archive. Setting the application classpath is different for each target environment so all options will not be discussed here. However, the following is an example of how to set the classpath from a Windows command line assuming the *csu-1.0-all.zip* archive was extracted into the C:\ directory:

```
SET CLASSPATH=C:\csu-1.0\lib\jakarta-commons\commons-lang.jar;C:\csu-1.0\lib\jakarta-commons\commons-collections.jar;C:\csu-1.0\lib\jakarta-commons\commons-logging.jar;C:\csu-1.0\lib\jakarta-commons\commons-codec.jar;C:\csu-1.0\lib\log4j\log4j.jar;C:\csu-1.0\lib\dom4j\dom4j.jar;C:\csu-1.0\dist\csu-1.0.jar;C:\csu-1.0\dist\csu-1.0-examples.jar;
```

### 2.3.5 Applying the License File

---

Once a CSU application instantiates the `CiscoRouter` class, the SDK will activate itself by attempting to locate and validate a Fox Smart license file.

By default, the SDK will look in the JVM startup directory for a file called `license.foxsmart`. However, the license file can be renamed and/or placed in an alternate location if desired. If this is the case, the SDK must be informed of the location and name of the license file using the JVM system property startup parameter `FOXSMART_LICENSE_URL`. This parameter must specify a valid URL to the location of the file. For example, the following startup parameter would specify that the file is located in the `C:\csu-1.0` directory and has the name `license.foxsmart`:

```
-DFOXSMART_LICENSE_URL="file:///C:\csu-1.0\license.foxsmart"
```

*The license file contains customer specific information including what version of the product can be used, when the license was created, when the license expires (for evaluation customers), the IP address of the machine the SDK is permitted to run on, and the list of devices (i.e. routers and switches) the SDK is permitted to communicate with. The license file should not be modified by the customer in any way or the product will no longer function.*

As an example, copy the Fox Smart license file into the following location:

```
C:\csu-1.0\license.foxsmart
```

*Special Notes For Evaluation Customers Only:*

- An evaluation license will instruct the SDK to validate itself by contacting the Fox Smart web site at [www.FoxSmart.com](http://www.FoxSmart.com). As such, the Fox Smart web site must be network accessible or the toolkit will not function. This validation process occurs periodically (approximately once a day).
- Evaluation licenses are valid for 30 days. Once the 30 day evaluation period expires, the SDK will no longer function.

## 2.3.6 Logging Options

---

Cisco Smart Update SDK uses the **commons-logging** API to provide useful runtime logging information. The commons-logging API provides a generic logging interface that can be used by itself or with another 3rd party logging toolkit. Customers are free to use any commons-logging implementation they wish although Fox Smart recommends using **log4j** which is included in the CSU SDK distribution.

CSU uses the following logging levels:

- **WARN** displays information when problems are encountered that are not severe enough to throw an exception.
- **INFO** displays useful runtime information such as router interaction, runtime parameters, configuration information, etc. that can be useful to monitor the toolkit's main functionality. Note that this log level (or any more detailed log level) will display router commands including sent passwords. As such, it is highly recommended not to use this or any more detailed logging level in a production environment.
- **DEBUG** displays extra information that helps monitor the toolkit flow (e.g. key method entry and exit points, adding and removing components from the configuration, etc.).
- **TRACE** displays detailed debugging information that includes raw data read from the router. This log level will produce a lot of output so it should typically only be used when debugging low level router communication issues.

The following is an example log4j.xml configuration file that can be used to log useful information to the console:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/" debug="false">
  <!-- Console Appender -->
  <appender name="console.appender" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern">
```

```

        value="%d{MMM dd yyyy HH:mm:ss} [%C{1}.%M] %m%n"/>
    </layout>
</appender>

<!-- Fox Smart Logging Configuration -->
<logger name="com.foxsmart" additivity="false">
    <level value="info"/>
    <appender-ref ref="console.appender"/>
</logger>

<!-- Root Configuration -->
<root>
    <level value="warn"/>
    <appender-ref ref="console.appender"/>
</root>
</log4j:configuration>

```

## 2.3.7 Running an Application Program

Once the SDK is fully setup, an application program can be run using the CSU SDK. One example program that comes with the CSU distribution is called `ReadRouterGeneratedConfiguration` which reads the router IOS configuration from the router and displays a generated version of the configuration to the standard output. The following demonstrates how this example program can be run from the Windows command line. You will need to substitute your router's unique hostname, username, password, and enable password for the parameters listed. If a parameter is not applicable based on how a user is authenticated, it does not need to be entered.

```

C:\>java -DFOXSMART_LICENSE_URL="file:///C:\csu-1.0\license.foxsmart"
com.foxsmart.csu.example.ReadRouterGeneratedConfiguration -h <router hostname> -u
<username> -p <password> -e <enable password>

```

The following is an example of what the program output should look like:

```

Mar 02 2008 17:08:30 [License.validateLicense] License does not require validation.
Mar 02 2008 17:08:30 [CommonsTelnetProtocolHandler.connect] Connecting to router "<router>" on port 23
using Telnet protocol...
Mar 02 2008 17:08:30 [CommonsTelnetProtocolHandler.connect] A username prompt was found.
Mar 02 2008 17:08:30 [CommonsTelnetProtocolHandler.connect] Sending username to the router...
Mar 02 2008 17:08:30 [AbstractProtocolHandler.sendCommand] Command: <username>
Mar 02 2008 17:08:30 [CommonsTelnetProtocolHandler.connect] Reading data after username sent...
Mar 02 2008 17:08:31 [CommonsTelnetProtocolHandler.connect] A password prompt was found.
Mar 02 2008 17:08:31 [CommonsTelnetProtocolHandler.connect] Sending standard password to the router...
Mar 02 2008 17:08:31 [AbstractProtocolHandler.sendCommand] Command: <password>
Mar 02 2008 17:08:31 [CommonsTelnetProtocolHandler.connect] Reading data after password sent...
Mar 02 2008 17:08:31 [CiscoRouter.login] Connected successfully.
Mar 02 2008 17:08:31 [CiscoRouter.login] Reading data after initial router connection has been made...
Mar 02 2008 17:08:31 [CiscoRouter.login] Sending "terminal length 0" command to router...
Mar 02 2008 17:08:31 [AbstractProtocolHandler.sendCommand] Command: terminal length 0
Mar 02 2008 17:08:31 [CiscoRouter.login] Reading data after command sent...
Mar 02 2008 17:08:32 [CiscoRouter.login] Logged in successfully.
Mar 02 2008 17:08:32 [CiscoRouter.login] Sending "terminal width 0" command to router...
Mar 02 2008 17:08:32 [AbstractProtocolHandler.sendCommand] Command: terminal width 0
Mar 02 2008 17:08:32 [CiscoRouter.login] Sending "enable" command to gain additional router access...
Mar 02 2008 17:08:32 [AbstractProtocolHandler.sendCommand] Command: enable
Mar 02 2008 17:08:32 [CiscoRouter.login] Reading data after enable command sent...
Mar 02 2008 17:08:32 [CiscoRouter.login] Looking for the enable password prompt...
Mar 02 2008 17:08:32 [CiscoRouter.login] Sending enable password to the router...
Mar 02 2008 17:08:32 [AbstractProtocolHandler.sendCommand] Command: <enable password>
Mar 02 2008 17:08:32 [CiscoRouter.login] Reading data after password sent...
Mar 02 2008 17:08:32 [CiscoRouter.login] Enable access obtained successfully.
Mar 02 2008 17:08:32 [CiscoRouter.login] Determining the router prompt...
Mar 02 2008 17:08:32 [CiscoRouter.login] Router prompt determined to be "Router#".
Mar 02 2008 17:08:32 [AbstractProtocolHandler.sendCommand] Command: show running-config
Mar 02 2008 17:08:33 [CiscoRouter.logout] Disconnected successfully.

```

Generated Router configuration:

```

version 12.2
!
hostname <router>
!
logging rate-limit console 10 except errors
enable secret 5 <enable password>
!
memory-size iomem 40
ip subnet-zero
no ip finger
ip classless
no ip http server
!
call rsvp-sync
!
interface Serial0
no ip address
no keepalive
shutdown
!
interface FastEthernet0
ip address 192.168.0.1 255.255.255.0
no keepalive
speed auto
!
line con 0
line aux 0
line vty 0 4
exec-timeout 0 0
password 7 <password>
login
!
end

```

## 2.3.8 Running CSU SDK Behind a Firewall

---

As mentioned previously, Fox Smart evaluation licenses are validated on the Fox Smart web site approximately once per day. This requires the SDK to make an HTTP connection to `www.foxsmart.com`. If the example program produces a timeout exception because it can't connect to the Fox Smart server and the reason is because a firewall is in place that requires all outgoing HTTP connections to go through an HTTP proxy, the proxy information must be entered on the startup command line using startup properties. For example, the following will run the same example program using a proxy host of "`proxy.domain.com`" and port "`80`":

```

C:\>java -DproxySet="true" -DproxyHost="proxy.domain.com" -DproxyPort="80" -
DFOXSMART_LICENSE_URL="file:///C:\csu-1.0\license.foxsmart"
com.foxsmart.csu.example.ReadRouterGeneratedConfiguration -h <router hostname> -u
<username> -p <password> -e <enable password>

```

Substitute your own proxy host and port into the above command line as appropriate.

## 2.3.9 Common Exceptions

---

When an application program attempts to login to the router, it will connect with and attempt to authenticate with the router. When CSU SDK is used for the first time, there are many reasons why an exception might be thrown when initially attempting to login to the router. The following are some common reasons:

- **Invalid Hostname:** When an IP address can't be determined for the specified hostname, a `java.net.UnknownHostException` will typically be thrown. When this happens, ensure an IP address can be determined for the specified hostname. A quick way to test

if the hostname is recognized is to ping the hostname on your local machine. If the hostname can't be resolved, contact your network administrator.

- **Connection Problem:** If an IP address is specified or if an IP address can be successfully determined from the specified hostname, it is possible that a connection to the router still can't be established. This will typically result in a `java.io.IOException`. Possible reasons for this type of exception could be that the IP address is not a Cisco router or the device with the specified IP address is down or does not exist. Another possible reason could be a firewall that is blocking the connection. CSU SDK uses the SSH and telnet protocols to connect to and communicate with the router so a good test is to manually SSH or telnet to the router to ensure connectivity can be established.
- **Invalid Username or Password:** If the router requires a username or password for standard access and/or an enable password for "enable" access, these passwords must be passed to the `CiscoRouter` class. If the standard password is incorrect, a `com.foxsmart.csu.AuthenticationFailedException` will be thrown. If the enable password is incorrect, a `com.foxsmart.csu.InvalidEnablePasswordException` will be thrown. When this happens, ensure the username and passwords are correct and try again. This can be verified by connecting directly to the router using SSH or telnet, logging in, and gaining enable access with the specified username and passwords.
- **Invalid Configuration:** When Cisco Smart Update SDK obtains a running configuration from the router, it downloads the configuration, parses it, and creates a `CiscoRouterConfig` class hierarchy that represents the entire router configuration. Given the large number of Cisco router hardware types and IOS versions that exist in a network, it is possible that an unanticipated IOS configuration can be encountered which will typically result in a `com.foxsmart.csu.config.InvalidConfigurationException`. If this happens, it is possible that a CSU SDK software patch will be required. If so, Fox Smart will typically require the following information to help diagnose the problem and provide a patch: 1) the Cisco router module and hardware configuration, 2) the output of the "*show version*" command on the router, and 3) the output of the "*show run*" command on the router.

## 2.3.10 runCsuApp.bat

---

For Windows users, CSU SDK comes with a batch file called `runCsuApp.bat` that provides an easier way to run the example applications. To use it, you must first follow the previous sections:

1. Extract the CSU SDK Distribution
2. Installing the Java Runtime Environment (JRE)
3. Adding the JRE to the system PATH
4. Applying the License File

At this point, the full CSU SDK distribution should be extracted into a directory (e.g. `C:\csu-1.0`), the JRE should be installed, the JRE package should be added to the system path, and the license file should be placed in the root directory (e.g. `C:\csu-1.0`).

The `runCsuApp.bat` batch file is located in the "bin\windows" subdirectory of the installation. It can be run by changing into this directory or ensuring this directory is part of the path. The following describes the various parameters that it supports:

```
Usage: runCsuApp.bat [-help] [-proxyHost <proxyHost>] [-proxyPort <proxyPort>] [-licensePath <licensePath>] [-classPath <classPath>] -appName <appName> [<appParam1> <appParam2> ...]
```

## CHAPTER 2

help: Display this usage.  
proxyHost: The hostname of the HTTP proxy (e.g. proxy.domain.com).  
proxyPort: The port number of the HTTP proxy (e.g. 80).  
licensePath: The path to the license file  
(e.g. C:\csu\license.foxsmart).  
Default is CSU root directory.  
classPath: Additional classpath entries  
(e.g. userLibrary1.jar@userLibrary2.jar).  
CSU classpath is automatically included.  
Use '@' instead of ';' to separate entries.  
appName: The case sensitive name of the CSU application  
(e.g. ReadRouterOriginalConfiguration).  
CSU examples package will be used if no package is present.  
appParams: The list of application specific parameters  
(e.g. -h myhost ...).

The *proxyHost* and *proxyPort* parameters are only needed if connecting via a firewall. See "Running CSU SDK Behind a Firewall" for more details.

The *licensePath* parameter is not needed if the Fox Smart license file is called `license.foxsmart` and is located in the root installation directory.

The *classPath* parameter is only needed if adding additional JAR's to the program execution. The CSU SDK required JAR's are automatically added.

The *appName* is required and specifies the name of the example application program.

The *appParams* are only needed if the example application takes additional parameters.

For example, the following is how the *GetRouterVersionInfo* example program can be executed:

```
runCsuApp.bat -appName GetRouterVersionInfo -h <host> -u <username> -p <password>
```

## 3. CSU SDK Example Applications

---

The CSU SDK distribution contains several example applications contained in the `csu-examples.jar` file that demonstrate some of its basic functionality. Although almost any imaginable functionality can be programmed using the CSU SDK, the provided examples are meant to give a brief idea of what can be done with the product. The source code for each example is also included in the examples JAR file.

This chapter describes the individual example programs, their usage, and sample output. The sample output listed is just an example of what the output might look like. A customer's specific output will vary depending on their specific environment.

The `csu-examples.jar` file contains a sample `log4j.xml` logging configuration file that is configured to display "info" level logging for the SDK. This means that informational logging will be displayed along with the program's output. This is meant to display useful information that demonstrates what the SDK is doing behind the scenes. The logging level can be modified by editing the `log4j.xml` file in the `csu-examples.jar` file and setting the "`com.foxsmart`" logger to a "`level`" other than "`info`".

Each example program can be run as discussed in the "Installation and Configuration" chapter, however, the name of the program and its options will be different as described.

### 3.1 DisplayLicenseInfo

---

The `DisplayLicenseInfo` program reads a Fox Smart license and display information about the license.

Usage: `DisplayLicenseInfo`

Sample program output:

```
Mar 02 2008 07:39:44 [License.validateLicense] License does not require validation.
Licensee: Fox Smart, Inc.
Licensee Id: <license_id>
Component: Cisco Smart Update
Component version: 1.0
License version: any
License version valid: true
License created: Oct-18-2007
License expiration: <expiration date>
Current date: Mar-02-2008
License expired: false
License created before current day: true
License expires soon (within 14 days): false
```

```

Number of days until license expires: <number of days>
Validation required: false
License file found: true
License found: true
License structure valid: true
License validated: true
License valid: true
Signature: <signature>
Signature valid: true
Fox Smart URL: www.foxsmart.com
Localhost address: 192.168.0.2
Localhost authorized: true
Authorized IP: 192.168.0.2
Authorized Devices: 192.168.0.1

```

## 3.2 VerifyLicenseForHostname

---

The *VerifyLicenseForHostname* program verifies a Cisco Smart Update license by creating a *CiscoRouter* object for a specific hostname.

```

Usage: VerifyLicenseForHostname <-h <hostname>> [-u <username>] [-p <password>] [-e <enable password>]

-h = The hostname or IP address of the device
-u = The authentication username
-p = The authentication password
-e = The authentication enable password

```

Sample program output:

```

Mar 02 2008 07:42:48 [License.validateLicense] License does not require validation.
License successfully verified.

```

## 3.3 VerifyRouterAuthentication

---

The *VerifyRouterAuthentication* program verifies that a router can be logged into and the access that is obtained.

```

Usage: VerifyRouterAuthentication <-h <hostname>> [-u <username>] [-p <password>] [-e <enable password>]

-h = The hostname or IP address of the device
-u = The authentication username
-p = The authentication password
-e = The authentication enable password

```

Sample program output:

```

Mar 02 2008 14:25:58 [License.validateLicense] License does not require validation.
Mar 02 2008 14:25:58 [CommonsTelnetProtocolHandler.connect] Connecting to router "<router>" on port 23
using Telnet protocol...
Mar 02 2008 14:25:59 [CommonsTelnetProtocolHandler.connect] A username prompt was found.
Mar 02 2008 14:25:59 [CommonsTelnetProtocolHandler.connect] Sending username to the router...
Mar 02 2008 14:25:59 [AbstractProtocolHandler.sendCommand] Command: <username>
Mar 02 2008 14:25:59 [CommonsTelnetProtocolHandler.connect] Reading data after username sent...
Mar 02 2008 14:25:59 [CommonsTelnetProtocolHandler.connect] A password prompt was found.
Mar 02 2008 14:25:59 [CommonsTelnetProtocolHandler.connect] Sending standard password to the router...
Mar 02 2008 14:25:59 [AbstractProtocolHandler.sendCommand] Command: <password>
Mar 02 2008 14:25:59 [CommonsTelnetProtocolHandler.connect] Reading data after password sent...
Mar 02 2008 14:25:59 [CiscoRouter.login] Connected successfully.
Mar 02 2008 14:25:59 [CiscoRouter.login] Reading data after initial router connection has been made...
Mar 02 2008 14:25:59 [CiscoRouter.login] Sending "terminal length 0" command to router...
Mar 02 2008 14:25:59 [AbstractProtocolHandler.sendCommand] Command: terminal length 0
Mar 02 2008 14:25:59 [CiscoRouter.login] Reading data after command sent...
Mar 02 2008 14:26:00 [CiscoRouter.login] Logged in successfully.
Mar 02 2008 14:26:00 [CiscoRouter.login] Sending "terminal width 0" command to router...
Mar 02 2008 14:26:00 [AbstractProtocolHandler.sendCommand] Command: terminal width 0

```

```

Mar 02 2008 14:26:00 [CiscoRouter.login] Sending "enable" command to gain additional router access...
Mar 02 2008 14:26:00 [AbstractProtocolHandler.sendCommand] Command: enable
Mar 02 2008 14:26:00 [CiscoRouter.login] Reading data after enable command sent...
Mar 02 2008 14:26:00 [CiscoRouter.login] Looking for the enable password prompt...
Mar 02 2008 14:26:00 [CiscoRouter.login] Sending enable password to the router...
Mar 02 2008 14:26:00 [AbstractProtocolHandler.sendCommand] Command: <enable password>
Mar 02 2008 14:26:00 [CiscoRouter.login] Reading data after password sent...
Mar 02 2008 14:26:00 [CiscoRouter.login] Enable access obtained successfully.
Mar 02 2008 14:26:00 [CiscoRouter.login] Determining the router prompt...
Mar 02 2008 14:26:00 [CiscoRouter.login] Router prompt determined to be "Router#".
Mar 02 2008 14:26:00 [CiscoRouter.logout] Disconnected successfully.
Router hostname:                <router>
Router username:                <username>
Router password:                <password>
Router enable password:        <enable password>
Username used for authentication: false
Password used for authentication: true
Enable password used for authentication: true
Enable access obtained:        true

```

## 3.4 GetRouterPrompt

---

The *GetRouterPrompt* program gets and displays the router prompt.

```

Usage: GetRouterPrompt <-h <hostname>> [-u <username>] [-p <password>] [-e <enable password>]

-h = The hostname or IP address of the device
-u = The authentication username
-p = The authentication password
-e = The authentication enable password

```

Sample program output:

```
Router prompt: Router#
```

## 3.5 GetRouterTime

---

The *GetRouterTime* program reads and displays the router time (i.e. the "show clock" command).

```

Usage: GetRouterTime <-h <hostname>> [-u <username>] [-p <password>] [-e <enable password>]

-h = The hostname or IP address of the device
-u = The authentication username
-p = The authentication password
-e = The authentication enable password

```

Sample program output:

```
Router time: Mar 02, 2008 8:58:18 AM
```

## 3.6 GetRouterVersionInfo

---

The *GetRouterVersionInfo* program reads and displays the router version information (i.e. the "show version" command).

```

Usage: GetRouterVersionInfo <-h <hostname>> [-u <username>] [-p <password>] [-e <enable password>]

-h = The hostname or IP address of the device
-u = The authentication username
-p = The authentication password

```

-e = The authentication enable password

### Sample program output:

```
Router version: Cisco Internetwork Operating System Software
IOS (tm) C1700 Software (C1700-K9O3SY7-M), Version 12.2(11)T10,  RELEASE SOFTWARE (fc3)
TAC Support: http://www.cisco.com/tac
Copyright (c) 1986-2003 by cisco Systems, Inc.
Compiled Thu 23-Oct-03 11:43 by eaarmas
Image text-base: 0x80008124, data-base: 0x80D52B4C

ROM: System Bootstrap, Version 12.0(3)T, RELEASE SOFTWARE (fc1)
ROM: C1700 Software (C1700-K9O3SY7-M), Version 12.2(11)T10,  RELEASE SOFTWARE (fc3)

Router uptime is 1 week, 5 days, 22 hours, 24 minutes
System returned to ROM by power-on
System image file is "flash:c1700-k9o3sy7-mz.122-11.T10.bin"

cisco 1720 (MPC860T) processor (revision 0x601) with 29492K/3276K bytes of memory.
Processor board ID JAD05190RWV (3501632589), with hardware revision 0000
MPC860T processor: part number 0, mask 32
Bridging software.
X.25 software, Version 3.0.0.
1 FastEthernet/IEEE 802.3 interface(s)
1 Serial(sync/async) network interface(s)
32K bytes of non-volatile configuration memory.
8192K bytes of processor board System flash (Read/Write)

Configuration register is 0x10
```

## 3.7 ReadRouterOriginalConfiguration

---

The *ReadRouterOriginalConfiguration* program connects to a router, reads the IOS configuration (i.e. the "show running-config" command), parses the IOS configuration, and displays the original configuration.

```
Usage: ReadRouterOriginalConfiguration <-h <hostname>> [-u <username>] [-p <password>] [-e <enable password>]
```

```
-h = The hostname or IP address of the device
-u = The authentication username
-p = The authentication password
-e = The authentication enable password
```

### Sample program output:

```
Original Router configuration:
!
! Last configuration change at 07:02:24 UTC Fri Mar 2 2008
!
version 12.2
no service single-slot-reload-enable
no service timestamps debug uptime
no service timestamps log uptime
no service password-encryption
!
hostname <router>
!
logging rate-limit console 10 except errors
enable secret 5 <enable password>
!
memory-size iomem 40
ip subnet-zero
no ip finger
!
call rsvp-sync
!
interface FastEthernet0
 ip address 192.168.0.1 255.255.255.0
 no keepalive
 speed auto
```

```

!
interface Serial0
  no ip address
  no keepalive
  shutdown
!
ip classless
no ip http server
!
line con 0
line aux 0
line vty 0 4
  exec-timeout 0 0
  password 7 <password>
  login
!
no scheduler allocate
end

```

## 3.8 ReadRouterGeneratedConfiguration

---

The *ReadRouterGeneratedConfiguration* program connects to a router, reads the IOS configuration (i.e. the "show running-config" command), parses the IOS configuration, and displays the toolkit generated configuration.

Usage: *ReadRouterGeneratedConfiguration* <-h <hostname>> [-u <username>] [-p <password>] [-e <enable password>]

```

-h = The hostname or IP address of the device
-u = The authentication username
-p = The authentication password
-e = The authentication enable password

```

Sample program output:

```

Generated Router configuration:
version 12.2
!
hostname <router>
!
logging rate-limit console 10 except errors
enable secret 5 <enable password>
!
memory-size iomem 40
ip subnet-zero
no ip finger
ip classless
no ip http server
!
call rsvp-sync
!
interface Serial0
  no ip address
  no keepalive
  shutdown
!
interface FastEthernet0
  ip address 192.168.0.1 255.255.255.0
  no keepalive
  speed auto
!
line con 0
line aux 0
line vty 0 4
  exec-timeout 0 0
  password 7 <password>
  login
!
end

```

## 3.9 VerifyRouterConfigs

---

The *VerifyRouterConfigs* program reads router configurations from a list of files and/or directories and verifies that the configurations can be parsed successfully. To run this program, router configurations need to be stored as individual text files on the local file system.

Usage: VerifyRouterConfigs <-l <file list>>

-l = comma separated list of filenames and/or directories that contain IOS configurations to verify (e.g. "file.txt, dir")

Sample program output:

```

=====
Processing File: "C:\csu\config.txt".
Reading contents of file...Successful.
Parsing IOS configuration...Successful.
Generating IOS configuration...Successful.
Original IOS Configuration:
-----
!
! Last configuration change at 07:02:24 UTC Fri Mar 2 2008
!
version 12.2
no service single-slot-reload-enable
no service timestamps debug uptime
no service timestamps log uptime
no service password-encryption
!
hostname <router>
!
logging rate-limit console 10 except errors
enable secret 5 <enable password>
!
memory-size iomem 40
ip subnet-zero
no ip finger
!
call rsvp-sync
!
interface FastEthernet0
 ip address 192.168.0.1 255.255.255.0
 no keepalive
 speed auto
!
interface Serial0
 no ip address
 no keepalive
 shutdown
!
ip classless
no ip http server
!
line con 0
line aux 0
line vty 0 4
 exec-timeout 0 0
 password 7 <password>
 login
!
no scheduler allocate
end

Generated IOS Configuration:
-----
version 12.2
!
hostname <router>
!
logging rate-limit console 10 except errors
enable secret 5 <enable password>
!
memory-size iomem 40
ip subnet-zero

```

```

no ip finger
ip classless
no ip http server
!
call rsvp-sync
!
interface Serial0
no ip address
no keepalive
shutdown
!
interface FastEthernet0
ip address 192.168.0.1 255.255.255.0
no keepalive
speed auto
!
line con 0
line aux 0
line vty 0 4
exec-timeout 0 0
password 7 <password>
login
!
end

```

## 3.10 BackupRouterConfigs

---

The *BackupRouterConfigs* program reads router configurations from a list of routers and writes those configurations to an optional user supplied output directory. This program uses the same username, password, and enable password for all routers.

```
Usage: BackupRouterConfigs [-o] [-u <username>] [-p <password>] [-e <enable password>] [-f <input file>]
[-d <output directory>] [-l <device list>]
```

```

-o = The backup file will be overridden if one already exists
-u = The authentication username
-p = The authentication password
-e = The authentication enable password
-f = Input file containing the list of devices to backup (one per line)
-d = The output directory where the backed up configurations should be stored
-l = comma separated list of devices to backup (e.g. "host1, host2, host3")

```

Sample program output:

```
File '<router>_backup_20080302.txt' created successfully.
All backups are completed.
```

## 3.11 SetRouterTime

---

The *SetRouterTime* program sets the router time with the user supplied parameter (i.e. the "clock set" command). The time is then re-read from the router and displayed to the user.

```
Usage: SetRouterTime <-h <hostname>> [-u <username>] [-p <password>] [-e <enable password>] <time>
```

```

-h = The hostname or IP address of the device
-u = The authentication username
-p = The authentication password
-e = The authentication enable password
time = the time to set in the format of MM/dd/yyyy HH:mm:ss (e.g. 10/24/2007 13:30:05)

```

Sample program output:

```
Router time set successfully.
Time as entered by the user:   Mar 02, 2008 9:36:30 AM
Time read back from the router: Mar 02, 2008 9:36:30 AM
```

## 3.12 SendRouterCommand

---

The *SendRouterCommand* program sends a user supplied command to the router and displays the results. The example below assumes the “show ip route” command was sent.

```
Usage: SendRouterCommand <-h <hostname>> [-u <username>] [-p <password>] [-e <enable password>]
<command>
```

```
-h = The hostname or IP address of the device
-u = The authentication username
-p = The authentication password
-e = The authentication enable password
command = the command to send to the router (e.g. show version)
```

Sample program output:

```
Command sent successfully to the router.
Command response: Codes: C - connected, S - static, I - IGRP, R - RIP, M - mobile, B - BGP
D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
E1 - OSPF external type 1, E2 - OSPF external type 2, E - EGP
i - IS-IS, L1 - IS-IS level-1, L2 - IS-IS level-2, ia - IS-IS inter area
* - candidate default, U - per-user static route, o - ODR
P - periodic downloaded static route

Gateway of last resort is not set

C    192.168.0.0/24 is directly connected, FastEthernet0
```

## 3.13 UpdateRouterBanners

---

The *UpdateRouterBanners* program updates the various router banners (i.e. the “*banner*” command). The example below sets the “incoming” banner to “Incoming Banner”. This example program keeps track of how long it takes to update the configuration and displays the total time and average time per command. It also displays details about the update which are displayed using the *toString* method on the returned *ConfigUpdateResult* object.

```
Usage: UpdateRouterBanners <-h <hostname>> [-u <username>] [-p <password>] [-e <enable password>] [-exec
<exec banner>] [-incoming <incoming banner>] [-login <login banner>] [-motd <motd banner>] [-prompt-
timeout <prompt timeout banner>] [-slip-ppp <slip ppp banner>]
```

```
-h = The hostname or IP address of the device
-u = The authentication username
-p = The authentication password
-e = The authentication enable password
-exec = The exec banner text or "delete" to remove the banner
-incoming = The incoming banner text or "delete" to remove the banner
-login = The login banner text or "delete" to remove the banner
-motd = The motd banner text or "delete" to remove the banner
-prompt-timeout = The prompt timeout banner text or "delete" to remove the banner
-slip-ppp = The slip/ppp banner text or "delete" to remove the banner
```

Sample program output:

```
Configuration Update Result:
Config Errors Present: false
Rollback Error:         false
NVRAM Update Error:    false
Configlet:
[banner incoming ^Incoming Banner^]

Update Session:
[Router#configure terminal
Enter configuration commands, one per line. End with CNTL/Z.
Router(config)#banner incoming ^Incoming Banner^
Router(config)#end
```

```

Router#]

Commands And Responses:
Command 0: [configure terminal]
Response 0: [Enter configuration commands, one per line. End with CNTL/Z.]
Command 1: [banner incoming ^Incoming Banner^]
Response 1: []
Command 2: [end]
Response 2: []

Issued 3 command(s) in 1 Second, 94 Milliseconds.
Average time per command: 364 Milliseconds.

```

## 3.14 UpdateStaticRoute

---

The *UpdateStaticRoute* program updates adds/updates/delete a single IP route (i.e. the "ip route" command). If the "-delete" flag is used, the route with the destination prefix and mask will be deleted. Otherwise, the route will be added if it doesn't exist or update if it does exist. The example below adds a route with a destination prefix of "10.10.10.10", a destination mask of "255.255.255.255", a next hop interface of "Serial0", a next hop address of "10.10.10.11", a distance metric of 2, and makes the route permanent.

```

Usage: UpdateStaticRoute [-permanent] [-delete] <-h <hostname>> [-u <username>] [-p <password>] [-e
<enable password>] <-destPrefix <destination prefix>> <-destMask <destination mask>> [-nextHopInterface
<next hop interface>] [-nextHopAddress <next hop address>] [-distance <distance>]

```

```

-permanent = The route will be permanent
-delete = Delete the route
-h = The hostname or IP address of the device
-u = The authentication username
-p = The authentication password
-e = The authentication enable password
-destPrefix = The destination prefix for the route
-destMask = The destination mask for the route
-nextHopInterface = The next hop interface
-nextHopAddress = The next hop address
-distance = The route distance metric

```

### Sample program output:

```

Configuration Update Result:
Config Errors Present: false
Rollback Error:         false
NVRAM Update Error:    false
Configlet:
[ip route 10.10.10.10 255.255.255.255 Serial0 10.10.10.11 permanent 2]

Update Session:
[Router#configure terminal
Enter configuration commands, one per line. End with CNTL/Z.
Router(config)#ip route 10.10.10.10 255.255.255.255 Serial0 10.10.10.11 permanent 2
Router(config)#end
Router#]

Commands And Responses:
Command 0: [configure terminal]
Response 0: [Enter configuration commands, one per line. End with CNTL/Z.]
Command 1: [ip route 10.10.10.10 255.255.255.255 Serial0 10.10.10.11 permanent 2]
Response 1: []
Command 2: [end]
Response 2: []

```

## 4. CiscoRouter Class

---

The `CiscoRouter` class is the core class that represents a single Cisco router or switch. It handles all low level communication with the actual device and is responsible for sending and receiving commands to and from the router. This chapter discusses how to use this class, including the core `CiscoRouter` API's.

### 4.1 Instantiating a CiscoRouter Object

---

The `CiscoRouter` class has only one constructor that takes the following parameters:

- *hostname*: the hostname or IP address of the device. This parameter is required.
- *username*: the optional username that is used for device authentication. Null should be passed in if no username is required for authentication.
- *password*: the optional password that is used for device authentication. Null should be passed in if no password is required for authentication.
- *enablePassword*: the optional enable password that is required when enable access is needed to issue commands or work with configurations. Null should be passed in if enable access is not required.

The following is an example of how the `CiscoRouter` object can be instantiated:

```
CiscoRouter router = new CiscoRouter("hostname", "username", "password",  
"enablePassword");
```

Once the `CiscoRouter` class is instantiated, the SDK will verify the Fox Smart license and verify that the hostname is authorized. If any license problems are encountered, an exception will be thrown.

### 4.2 CiscoRouter Communication Options

---

The `CiscoRouter` class has a few communication options that can be configured to determine how the SDK will communicate with the device.

## 4.2.1 protocolHandlers

---

The `protocolHandlers` property determines which protocols will be used to communicate with the router. The two SDK currently supported protocols are SSH and Telnet. Both SSH 1.5 and SSH 2.0 protocols are supported.

The `setProtocolSupport` method is the preferred way to specify which protocols should be used when connecting to the router. This method takes three parameters: 1) a boolean to specify whether the SSH protocol should be used, 2) a boolean to specify whether the Telnet protocol should be used, and 3) the preferred protocol to use (i.e. SSH or Telnet). This method will automatically determine which protocol handlers are available and use the appropriate ones as necessary. Note that the *plink* SSH implementation takes precedence over the *MindTerm* implementation. The following example specifies that both the SSH and Telnet protocols should be used and the SSH protocol is the preferred one. It is also the default protocol implementation for the `CiscoRouter` class.

```
router.setProtocolSupport(true, true, ProtocolHandler.Protocol.SSH);
```

The `setProtocolHandlers` method is an alternate way to specify individual protocol handlers to use when connecting to the router. The three protocol handlers that come with the SDK are *PlinkSshProtocolHandler*, *MindTermSshProtocolHandler*, and *CommonsTelnetProtocolHandler*. The following example specifies that the *MindTermSshProtocolHandler* should be attempted first followed by the *CommonsTelnetProtocolHandler*:

```
List<ProtocolHandler> handlers = new ArrayList<ProtocolHandler>();
handlers.add(new MindTermSshProtocolHandler());
handlers.add(new CommonsTelnetProtocolHandler());
router.setProtocolHandlers(handlers);
```

## 4.2.2 defaultNoDataReadDelay

---

The `defaultNoDataReadDelay` property determines how many 1/10<sup>th</sup> of a second increments the SDK will wait while receiving no data until it considers a data read complete. In other words, in cases where the SDK doesn't know exactly what data is coming back after issuing a command, it will keep reading data until no more data is being returned from the router. This property determines how long that wait will be. The default value for this property is "20" or 2 seconds which is pretty conservative. This value can typically be lowered to as low as "1" on networks with little latency.

The following is an example of how this property can be set to 1:

```
router.setDefaultNoDataReadDelay(1);
```

## 4.2.3 defaultProtocolTimeout

---

The `defaultProtocolTimeout` property determines how many seconds the SDK will wait to receive an expected response from the device before it times out and throws an exception. The default value for this property is "60" seconds. This value is typically sufficient since most commands receive expected responses well before 60 seconds pass.

The following is an example of how this property can be set to 30 seconds:

```
router.setDefaultProtocolTimeout(30);
```

## 4.2.4 updateNvRamConfigFlag

---

The `updateNvRamConfigFlag` determines if any configuration updates should be written to NVRAM memory if no configuration errors are present during the update. Setting this flag to true has the effect of sending the *"write memory"* command to the router after the configuration commands have been successfully sent. If the flag is set to false, only the operating configuration will be updated, but the NVRAM configuration will be left alone. The default value for this flag is *"false"*.

The following is an example of how this flag can be set to true:

```
router.setUpdateNvRamConfigFlag(true);
```

## 4.2.5 stopUpdatesOnConfigErrorFlag

---

The `stopUpdatesOnConfigErrorFlag` determines whether updates to the operating configuration will continue if an individual command produces an error. If this flag is set to true, no further configuration updates will be sent to the device once an error is encountered. If this flag is set to false, all configuration commands will be issued to the device, even if an error is encountered. The default value for this flag is *"true"*.

The following is an example of how this flag can be set to false:

```
router.setStopUpdatesOnConfigErrorFlag(false);
```

## 4.2.6 rollbackOnConfigErrorFlag

---

The `rollbackOnConfigErrorFlag` determines whether a configuration will be rolled back if an error is encountered while sending configuration update commands. Note that rollback refers to writing the contents of NVRAM on top of (i.e. in addition to) what is currently in the operating configuration. As such, any settings on the operating configuration which are not explicitly undone in the NVRAM configuration will be left alone. If true, the currently saved configuration in NVRAM will be restored if a configuration error is encountered during a configuration update. If false, the configuration will be left the way it is at the time the configuration error occurred. The default value for this flag is *"true"*.

The following is an example of how this flag can be set to false:

```
router.setRollbackOnConfigErrorFlag(false);
```

## 4.3 Logging In and Logging Out

---

Once a `CiscoRouter` class has been instantiated for a particular host and set of authentication credentials, any communication with the router can only be done once the SDK is logged into the router. Likewise, when communication with the router is completed, the SDK should log out of the router.

### 4.3.1 Logging In

---

To login to the router, the *"login"* method should be called. The login method will establish communications with the router and authenticate to the highest extent possible based on the authentication parameters used when constructing the `CiscoRouter` object.

The following describes the high level protocol that is used when logging into the router:

- If the SDK is already logged into the router and the login method is called again, the SDK will disconnect from the router first before attempting to re-authenticate.
- The SDK will attempt to connect to the router using the list of configured protocol handlers in order. By default, SSH will be attempted first followed by Telnet. Each protocol handler attempts to connect to the router using the supplied username and password as appropriate.
- When SSH is used, the username and password are both needed to make a connection to the router and a connection can only be made when the username and password are both valid. If not, an `AuthenticationFailedException` will be thrown.
- When Telnet is used, the SDK will see if the router responds with a *"Username:"* prompt. If a *"Username:"* prompt is found, the specified username will be sent to the router. If a username was not supplied when the `CiscoRouter` object was instantiated, an exception will be thrown. If the router did not respond with a *"Username:"* prompt, the SDK will assume that a username is not required.

The SDK will see if the router responds with a *"Password:"* prompt. If a *"Password:"* prompt is found, the specified password will be sent to the router. If a password was not supplied when the `CiscoRouter` object was instantiated, an exception will be thrown. If the router did not respond with a *"Password:"* prompt, the SDK will assume that a password is not required.

The SDK will send an empty command (i.e. a carriage return) and check if the router responds with another username or password prompt. If one is found, an exception is thrown.

- The SDK will send a *"terminal length 0"* command to the router.
- The SDK will send a *"terminal width 0"* command to the router.
- If an enable password was supplied when the `CiscoRouter` object was instantiated, the *"enable"* command along with the *"enable password"* will be sent to the router to gain enable access. If the router does not have an enable password set or if the enable password is invalid, an `InvalidEnablePasswordException` will be thrown.
- If no enable password was supplied when the `CiscoRouter` object was instantiated, then enable access will not be attempted.

The following methods can be used to determine the state of the login session once the login method returns successfully:

- *getUsername()* - Returns the *"username"* that was specified when the `CiscoRouter` object was constructed.

- *getPassword()* - Returns the "password" that was specified when the `CiscoRouter` object was constructed.
- *getEnablePassword()* - Returns the "enable password" that was specified when the `CiscoRouter` object was constructed.
- *getUsernameUsed()* - Returns whether the "username" was used during the login process.
- *getPasswordUsed()* - Returns whether the "password" was used during the login process.
- *getEnableUsed()* - Returns whether the "enable password" was used during the login process.
- *getHasEnableAccess()* - Returns whether enable access was obtained during the login process.
- *getPrompt()* - Returns the router prompt that is used once the login process is complete.

### 4.3.2 Logging Out

---

To logout of the router, the "logout" method should be called. The logout method will disconnect any previously establish communications with the router. If the logout method is called and the SDK is not logged into the router, no action will be taken.

## 4.4 Sending Commands

---

Once the SDK has logged into the router, commands can be sent to the router. Any generic command can be sent to the router using one of the *sendCommandWithResults* methods although `CiscoRouter` has some built in helper methods that issue various commands automatically and perform any necessary type conversion.

### 4.4.1 Sending Generic Commands

---

Generic commands can be sent to the router using the following two methods:

- *sendCommandWithResults(String command)* - This method will send the specified command to the router and the results of the command will be returned. The SDK will attempt to determine if the command is valid or not before issuing the command itself.
- *sendCommandWithResults(String command, boolean checkForRecognizedCommand)* - This method does the same thing as the above command, but takes an additional parameter that determines whether the command should be checked to see if it is valid before issuing it.

A command is considered recognized if it can be entered on the router given the currently level of authentication access. If a command is not recognized either because it is an invalid command or because the appropriate level of access has not been established during the authentication process, an `UnrecognizedCommandException` will be thrown.

### 4.4.2 Sending Built In Commands

---

The `CiscoRouter` class comes with some built in methods that automatically issue certain command and return their results:

- *getTime()* - This method will return the router's time in the form of a `Calendar` object. The "*show clock*" command is the actual command that is sent to the router to obtain the time.
- *getVersionInfo()* - This method will return the router's version information as a `String`. The "*show version*" command is the actual command that is sent to the router to obtain the version information.
- *setTime(Calendar calendar)* - This method will set the router's time to the specified `Calendar` value. The "*clock set*" command is the actual command that is sent to the router to set the time.

## 4.5 Reading and Updating Configurations

---

The core strength of the Cisco Smart Update SDK is reading and writing IOS configurations. Configurations are first read in from the router, changes to the configurations are made using the SDK API's, and the configuration changes are written back to the router. The basic reading and writing of the configuration is done using the `CiscoRouter` class.

### 4.5.1 Reading a Running Configuration

---

The *getRunningConfiguration* method reads a running configuration from the router, parses it, and returns a `CiscoRouterConfig` object with all the configuration data present in the appropriate class hierarchy. This API should be used to simply examine the current running configuration or to get the current configuration with the intent to make changes to it.

The running configuration is obtained by issuing the "*show running-config*" command on the router.

### 4.5.2 Writing a Configuration

---

Once a running configuration is obtained from the router and changes are made to it in memory, the updated configuration can be written back to the router using the *writeConfiguration(CiscoRouterConfig newRouterConfig)* method. This method will take the new router configuration, re-read the current running configuration from the router, determine what has changed, determine the commands that need to be issued to make those changes, and issue those commands to the router.

The *writeConfiguration* method returns a `ConfigUpdateResult` object which contains information about the update session including the following properties:

- *configlet* - The configlet is the part of the IOS configuration that was written back to the router to make the desired changes. This may contain many lines if multiple IOS statements were written to perform the update.
- *updateSession* - This is the raw update session including the commands and responses as seen on the router.
- *commandsAndResponses* - This is a list of `CommandAndResponse` objects which each contain a `Command` and a `String` response. This list provides The `Command` object itself contains a list of `String` lines that in unison make up the full command. The list of *commandsAndResponses* provide a granular way to see each command that was issued

along with its own unique response. This gives the caller the ability to see if any warnings or errors were produced for each individual command.

- *configErrorsPresent* - This flag indicates whether any configuration errors were present when updating the router with the new configuration.
- *rollbackError* - This flag indicates whether a rollback error occurred if one was attempted.
- *nvRamUpdateError* - This flag indicates whether there was an error updating NVRAM if the NVRAM update was attempted.

In addition to the *writeConfiguration* method described above, the following API's are also available, but not recommended given their noted limitations:

- *writeConfigurationAsString(String routerConfig)* - This method takes the passed in full IOS configuration as a *String* and attempts to update the router based on the parsed version of the configuration. In other words, the passed in *routerConfig* object will be parsed into a *CiscoRouterConfig* object and passed into the main *writeConfiguration* method.

*Note that the passed in routerConfig must contain the entire IOS configuration and not just a configlet since it will be parsed and compared to the current running configuration. Any supported lines that are not present in the passed in routerConfig will be considered removed and deleted from the router.*

- *writeConfiglet(Commands commands)* - This method writes the specified commands to the router. No syntactical or logic checking is performed on the configlet commands so they are simply sent as is. This method will not perform a rollback if any errors occur during the writing of the configlet. If rollback functionality is required, the main *writeConfiguration* method should be used.
- *writeConfiglet(String configlet)* - This method takes the passed in configlet as a *String*, separates the *String* into multiple lines based on the carriage return character, creates a list of *Commands* with each command containing an individual line, and delegates to the *writeConfiglet(Commands commands)* API to perform the update. This method assumes that each line contains its own single line command so it will not work if a command spans more than one line. If that is the case, the *writeConfiglet(Commands commands)* API should be used.

## 5. Working with Configurations

---

The previous chapter discussed the `CiscoRouter` class and its core API's. The preferred `CiscoRouter` API's that read and write router configurations both work with a `CiscoRouterConfig` class which is the root configuration class. This chapter discusses the structure of the configuration and how it can be analyzed and manipulated to make configuration changes.

### 5.1 CiscoRouterConfig Class

---

The `CiscoRouter` *getRunningConfiguration* API reads the router's running IOS configuration and returns a `CiscoRouterConfig` object that contains the entire configuration in a fully populated class hierarchy. Likewise, the *writeConfiguration* API takes an updated version of the `CiscoRouterConfig` object that represents the desired changes to the router configuration and writes those changes back to the router. The user's job is to manipulate the read router configuration to the point where it represents the desired updated configuration.

Although the `CiscoRouterConfig` class isn't that large on its own, like all the configuration components, it extends `CiscoRouterConfigComponent` which provides much of the classes' power. The `CiscoRouterConfigComponent` class is described in detail in the upcoming chapters and sections.

The only unique API on the `CiscoRouterConfig` class is the *getOriginalIos()* API which returns the original unparsed IOS configuration as it was read from the router.

### 5.2 Configuration Components

---

As discussed in the previous section, the root router configuration is stored in a class called `CiscoRouterConfig`. The `CiscoRouterConfig` class as well as all the other configuration classes extend `CiscoRouterConfigComponent` which means that all the router configuration elements are stored as separate individual "components".

The components themselves are stored in a parent/child hierarchy which in totality make up the entire IOS router configuration. This entire router component hierarchy mirrors the same configuration hierarchy that is used by the router's IOS configuration. The `CiscoRouterConfig` component is the top most parent component whose children make up

the entire router configuration. Each component has configuration attributes of its own that can be updated as needed and each component may optionally be a parent (i.e. contains children) of other configuration components.

*Note that each component contains its own set of children so it is important when examining or manipulating configurations to search for and add child components to the correct parent.*

Let's look at the following IOS configuration piece as an example:

```
line vty 0 4
  exec-timeout 0 0
  password 7 <password>
  login
```

This example shows a VTY line configuration along with 3 indented children configuration lines that make up the complete configuration for that specific line. In Cisco Smart Update SDK, each line of the configuration is represented by its own component class where the parent component would be a "line" component and the 3 indented components would be children of the "line" component. The following shows what the component class hierarchy would look like:

```
+ CiscoRouterConfig
  + LineComponent
    - ExecTimeoutComponent
    - PasswordComponent
    - LoginComponent
```

## 5.2.1 Manipulating Components

---

When processing a router configuration, searching for and manipulating components is often required to either examine parts of a configuration or make changes to the configuration when updates are desired. For example, if you want to add an "exec-timeout" to an existing line within the configuration, you would need to find the `LineComponent`, create the `ExecTimeoutComponent` and add it to the `LineComponent`. Note that components are grouped together based on their class so various methods that take "Class" as a parameter should be called with the desired component's class (e.g. `LineComponent.class`). The following is a list of methods that are available for finding and manipulating components:

- `addComponent(CiscoRouterConfigComponent component)` - Adds a new child component to the current component. Multiple children with the same class type can be added as long as the child is an instance of `MultiInstanceComponent` and it shares the same characteristics as the other children that were added with the same class type. Otherwise, an `InvalidConfigurationException` will be thrown.
- `getAllComponents()` - Gets a list of all children components of the current component regardless of type.
- `getFirstComponent(Class key)` - Gets the first child component of the specified class. If more than one child exists for the specified class, then the first one that was originally added will be returned. This method is useful when a child of a specific class is desired and the caller knows that only one child will exist for that class (usually because the children are not an instance of `MultiInstanceComponent`).
- `getComponents(Class key)` - Gets the list of components of the specified class. If no components of the specified class exist, then an empty list will be returned.
- `getParentComponent()` - Gets the parent component of the current component. Null will be returned when no parent exists (usually because the current component is the root `CiscoRouterConfig` component).

- *getCiscoRouterConfig(CiscoRouterConfigComponent component)* - Static method that gets the parent `CiscoRouterConfig` component of this component. This is a useful way to get the root `CiscoRouterConfig` component when traversing configurations.
- *getTotalComponents(Class key)* - Gets the number of child components with the specified class of the current component. For example, if this method were called on the `CiscoRouterConfig` object by passing in `LineComponent.class` and three lines existed within the configuration, then this method would return 3.
- *removeAllComponents()* - Removes all children for all class types from the current component.
- *removeAllComponents(Class key)* - Removes all children for the specified class type from the current component.
- *removeComponent(CiscoRouterConfigComponent component)* - Removes a specific child component from the current component. If the child couldn't be found, no component is removed.

## 5.2.2 Component Interfaces

---

Although all router configuration elements are components because they extend from `CiscoRouterConfigComponent`, certain components implement special interfaces to give them extra meaning. The following is a list of these interfaces and what they represent:

- *CompositeComponent* - This interface signifies that a component may have children. A `LineComponent` is an example of a `CompositeComponent` because its children specify the configuration elements of that line.
- *MultiInstanceComponent* - This interface signifies that multiple instances of a component may exist under the same parent. A `LineComponent` is an example of a `MultiInstanceComponent` because `CiscoRouterConfig` may have multiple lines configured.
- *MultiLineComponent* - This interface signifies that the component may contain multiple lines that make up its configuration. For example, a `BannerComponent` is a `MultiLineComponent` because one `BannerComponent` contains all the configuration attributes for all the banners combined. As such, a single `BannerComponent` can generate multiple lines of IOS configuration - each one representing a single banner line.
- *IndelibleComponent* - This interface signifies that the component can not be deleted from the configuration. A `LineComponent` is an example of an `IndelibleComponent` because a line can not be deleted from the router.

## 6. Update Static Route Example Explained

In Section 3.14, we discussed an example program called `UpdateStaticRoute`. This program provides the ability to add, update, or delete a static route on the router. This chapter explains the details behind how this sample program was created. While some parts of the example program are not discussed here, the salient parts are. The complete original source code for the example is contained in the `csu-examples.jar`. The program contains the following usage:

```
Usage: UpdateStaticRoute [-permanent] [-delete] <-h <hostname>> [-u <username>] [-p <password>] [-e
<enable password>] <-destPrefix <destination prefix>> <-destMask <destination mask>> [-nextHopInterface
<next hop interface>] [-nextHopAddress <next hop address>] -distance <distance>
```

```
-permanent = The route will be permanent
-delete = Delete the route
-h = The hostname or IP address of the device
-u = The authentication username
-p = The authentication password
-e = The authentication enable password
-destPrefix = The destination prefix for the route
-destMask = The destination mask for the route
-nextHopInterface = The next hop interface
-nextHopAddress = The next hop address
-distance = The route distance metric
```

### 6.1 Working With Command Line Parameters

Like all the Cisco Smart Update SDK example programs, the first step is to parse the command line arguments and obtain them in a usable form. This is done using a CSU helper class called `CommandLineParams`. This class lets the application specify various program information including options and flags and handles the parsing of the command line arguments. Various constants are used for the various options and flags. Once the arguments are parsed, they are read and converted to the various formats needed by the SDK:

```
// Parameter keywords.
public static final String DEST_PREFIX = "destPrefix";
public static final String DEST_MASK = "destMask";
public static final String NEXT_HOP_INTERFACE = "nextHopInterface";
public static final String NEXT_HOP_ADDRESS = "nextHopAddress";
public static final String DISTANCE = "distance";
public static final String PERMANENT = "permanent";
public static final String DELETE = "delete";

// Build the command line arguments.
CommandLineParams params = new CommandLineParams();
params.setProgramName("UpdateStaticRoute");
params.addOption("h", "hostname", "The hostname or IP address of the device", true);
params.addOption("u", "username", "The authentication username", false);
params.addOption("p", "password", "The authentication password", false);
params.addOption("e", "enable password", "The authentication enable password", false);
params.addOption(DEST_PREFIX, "destination prefix", "The destination prefix for the route", true);
```

```

params.addOption(DEST_MASK, "destination mask", "The destination mask for the route", true);
params.addOption(NEXT_HOP_INTERFACE, "next hop interface", "The next hop interface", false);
params.addOption(NEXT_HOP_ADDRESS, "next hop address", "The next hop address", false);
params.addOption(DISTANCE, "distance", "The route distance metric", false);
params.addFlag(PERMANENT, "permanent", "The route will be permanent");
params.addFlag(DELETE, "delete", "Delete the route");

// Check the parameters.
try
{
    params.parse(args);
}
catch (Exception ex)
{
    // There was a problem parsing the arguments so display the error and the program usage and then
    exit.
    System.out.println(ex.getMessage());
    System.out.println();
    System.out.println(params.getUsage());
    System.exit(-1);
}

// Read and convert the parameters.
IpAddress destPrefix = new IpAddress(params.getOption(DEST_PREFIX));
IpAddress destMask = new IpAddress(params.getOption(DEST_MASK));
IpAddress nextHopAddress = (params.containsOption(NEXT_HOP_ADDRESS)) ?
    new IpAddress(params.getOption(NEXT_HOP_ADDRESS)) : null;
Integer distance = (params.containsOption(DISTANCE)) ?
    new Integer(params.getOption(DISTANCE)) : null;
boolean deleteRoute = params.containsFlag(DELETE);
boolean permRoute = params.containsFlag(PERMANENT);

InterfaceComponent nextHopInterface = null;
if (params.containsOption(NEXT_HOP_INTERFACE))
{
    // Create a next hop interface by getting the interface component and building it based
    // on the specified command line parameter.
    String nextHopInterfaceKeyword = params.getOption(NEXT_HOP_INTERFACE);
    nextHopInterface = InterfaceComponent.getComponent(nextHopInterfaceKeyword);
    nextHopInterface.buildInterfaceAndSubInterfaceNames(nextHopInterfaceKeyword,
        nextHopInterfaceKeyword);
}

```

## 6.2 Reading the Running Configuration

---

To read the device's running configuration, a `CiscoRouter` object is first instantiated with the hostname, username, password, and enable password. Once instantiated, we log into the router using the `login` method, and obtain the running configuration using the `getRunningConfiguration` method:

```

// Create the Cisco router object with the user supplied parameters.
CiscoRouter router = new CiscoRouter(params.getOption("h"), params.getOption("u"),
    params.getOption("p"), params.getOption("e"));

// Connect to the router.
router.login();

// Get the router configuration.
CiscoRouterConfig routerConfig = router.getRunningConfiguration();

```

## 6.3 Get or Create an IpRouteComponent

---

Once the running configuration is obtained, the specific component or components that represent the part of the configuration we need to work with must be obtained or created. In this example, we want to work with the `IpRouteComponent` which is the configuration component that deals with all IP routes. Since only one `IpRouteComponent` contains all routes

within it, we can call the *getFirstComponent* method with the component class we wish to obtain. If more than one component existed for the configuration you are working with, then the *getComponents* method could be used which returns a list of components. If the obtained component doesn't yet exist, a new one will be created and added to the configuration:

```
// Get the IP Route component.
IpRouteComponent ipRouteComponent =
    (IpRouteComponent)routerConfig.getFirstComponent(IpRouteComponent.class);

// If an IP Route component doesn't exist, then create a new one.
if (ipRouteComponent == null)
{
    ipRouteComponent = new IpRouteComponent();
    routerConfig.addComponent(ipRouteComponent);
}
```

## 6.4 Working with IpRouteEntry Objects

---

The *IpRouteComponent* contains a list of routes and stores them by destination prefix and mask. Each unique IP route is contained within an *IpRouteEntry* class. The list of routes for the specified destination prefix and mask can be obtained using the *getRoutes* method. Since this example program only knows how to work with one route, if the total number of routes found within the route configuration is greater than one, the program will exit.

If only one route exists, it will first be deleted by calling the *removeRoute* method. Then, if the user isn't deleting the route, a new one will be created and added to the configuration using the *addRoute* method. The new route will have all its information set on it using various "set" methods (e.g. *setDestinationPrefix*). Although we are handling an update using delete and add methods, an alternate approach would be to obtain the original route entry and update it directly with the user supplied parameters.

```
// Get the list of routes for the destination prefix and mask.
List routes = ipRouteComponent.getRoutes(destPrefix, destMask);

// If more than 1 route exists for the destination prefix and mask, then exit.
if (routes.size() > 1)
{
    System.out.println(routes.size() + " routes are configured for the destination prefix \"" +
        destPrefix + "\" and destination mask \"" + destMask +
        "\". Test driver only works with 1 route so no updates will be performed.");
    System.exit(-1);
}

// If only 1 route exists, get it from the list of routes and delete it.
if (routes.size() == 1)
{
    IpRouteEntry routeEntry = (IpRouteEntry)routes.get(0);
    ipRouteComponent.removeRoute(routeEntry);
}

// If the user isn't deleting a route, then add a new one.
if (!deleteRoute)
{
    IpRouteEntry routeEntry = new IpRouteEntry();
    routeEntry.setDestinationPrefix(destPrefix);
    routeEntry.setDestinationPrefixMask(destMask);
    routeEntry.setNextHopInterface(nextHopInterface);
    routeEntry.setNextHopAddress(nextHopAddress);
    routeEntry.setDistanceMetric(distance);
    routeEntry.setPermanentRouteFlag(permRoute);
    ipRouteComponent.addRoute(routeEntry);
}
```

## 6.5 Writing the Configuration Back to the Router

---

Once the `CiscoRouterConfig` has been updated with the routing changes, the updated configuration can be written back to the router using the `writeConfiguration` method which returns a `ConfigUpdateResult` object. The `ConfigUpdateResult` object contains information about the device update session that can be interrogated or displayed using its `toString` method. Once the configuration is written, we will logout of the router using the `logout` method:

```
// Update the configuration on the router.
ConfigUpdateResult result = router.writeConfiguration(routerConfig);

// Disconnect from the router.
router.logout();

// Display the configuration update results.
System.out.println("Configuration Update Result:\n" + result);
```

## 7. Parsing, Validating, and Generating IOS

---

All the chapters to this point have covered topics that are required to work with the Cisco Smart Update SDK out of the box. The remaining chapters cover advanced topics that describe how the SDK works behind the scenes and covers information required to extend the toolkit to add custom functionality.

The previous chapter discussed how the `CiscoRouterConfigComponent` class represents a single portion of the router's configuration and how the components can be manipulated to make configuration changes. But how did the configuration components get created to begin with when the router configuration was read? By parsing the originally read configuration.

### 7.1 CiscoRouterConfigComponent Abstract Methods

---

When a router configuration is read using the `getRunningConfiguration` method on `CiscoRouter`, the SDK does a lot of work behind the scenes to create the class hierarchy of components. Specifically, it parses the IOS configuration lines and "builds" the configuration information. Each component is responsible for building its own configuration from the IOS configuration lines that make up that component. In addition, each component is also responsible for validating its configuration and generating the IOS commands that make up its configuration. To enforce this functionality, the following abstract methods exist on `CiscoRouterConfigComponent` that each component must implement:

- `buildFromIos(List configLines)` - This method takes the passed in list of IOS configuration lines (each one a separate String) and builds its own component configuration based on the parsed information. It returns an updated list of the IOS configuration lines with its own lines removed. That way, the IOS configuration lines will get fewer and fewer as each component parses its own portion of the configuration until no lines are left and the entire configuration is parsed and built. If any portion of the configuration can't be parsed or if any unexpected information is found, an `InvalidConfigurationException` is thrown.
- `validate()` - This method is responsible for validating that all the configuration information contained within the component is valid. This helps protect against trying to work with or generate an IOS configuration which would be invalid. If any part of the configuration is deemed to be invalid, an `InvalidConfigurationException` is thrown. Validations occur when generating IOS statements which typically happen when attempting to write configuration changes back to the router.
- `generateIos()` - This method is responsible for generating and returning the IOS commands that make up the configuration for this component. This is the basis of

viewing a configuration and generating the necessary update statements to modify an existing configuration. This method will typically call *validate()* before generating IOS statements to ensure the necessary information needed to generate the IOS commands is valid.

## 7.2 Parsing IOS Configuration Lines

---

As mentioned above, each component must implement the *buildFromIos(List configLines)* method to build its configuration by parsing the passed in IOS configuration lines. To make this job easier, all components that extend *CiscoRouterConfigComponent* in turn extend another helper class called *CiscoRouterConfigParser*. *CiscoRouterConfigParser* provides a series of helper methods that aid in configuration parsing.

## 7.3 Working With Lines

---

The *buildFromIos* method is passed a list of lines. The following methods help working with the list of lines:

- *generateCommands(String ios)* - Generates a *Commands* object from a multi-line string. Each line in the string that is separated by a carriage return is a separate command.
- *getLine(List lines, int count)* - Gets a specified line, that is to say, an element, from the passed in List of lines. No bounds checking is performed to ensure that the List contains enough elements to retrieve count's element. If a carriage return is the last character on a line, it is removed.
- *getTotalLines(List configLines)* - Gets the total number of lines that are a part of this configuration component. The first line is assumed to be a part of the component and all additional lines are considered part of the component if they are indented more than the first line.
- *moreIndentedLines(List lines, int originalIndentPos)* - Determines if the next line in the list of lines is indented beyond the originally indented position for a component.
- *moreLines(List lines, int index)* - Determines if there are more lines to read.
- *removeLines(List lines, int count)* - Removes a specified number of lines (List elements) from the beginning of the passed in lines list. If count is greater than the number of elements remaining, all the elements will be removed and an empty list will be returned.

## 7.4 Working With Tokens

---

Once an individual line is obtained from the passed in list of lines, it can be divided into a list of tokens where each token is a *String* separated by at least one space. Individual methods also exist to manipulate a token index that can be used to keep track of which token is being parsed in the list of tokens. The following methods help working with the list of tokens:

- *checkLineComplete(String line, List tokens)* - Checks that there are no more tokens left to read on the current line.

- *checkTokenCount(String line, List tokens, Integer count)* - Ensures that the passed in list of tokens contains the count specified. This has the same functionality as calling *checkTokenCount(line, tokens, count, count)*.
- *checkTokenCount(String line, List tokens, Integer lowRange, Integer highRange)* - Ensures that the passed in list of tokens contains the correct count.
- *decrementTokenIndex()* - Decrements the current token index.
- *getNextOptionalToken(String line, List tokens)* - Returns the next token in the list of tokens if one exists. Otherwise, null is returned.
- *getNextToken(String line, List tokens)* - Returns the next token in the list of tokens.
- *getToken(List tokens, int count)* - Returns the string token at the specified position. It is assumed that the List of tokens contains a string at the specified position.
- *getTokenIndex()* - Gets the current token index.
- *getTokens(String line)* - Gets a list of tokens from the specified line. Each set of characters separated by a space character is a unique token. If line is null or is empty, an empty List is returned.
- *incrementTokenIndex()* - Sets the current token index.
- *moreTokens(List tokens)* - Determines if there are more tokens to read.
- *resetTokenIndex()* - Resets the current token index. This is the same as calling *setTokenIndex(0)*.
- *setTokenIndex(int index)* - Sets the current token index.
- *tokensLeft(List tokens)* - Returns the number of tokens left to read.

## 7.5 Converting Tokens

---

Once an individual token is obtained, the following methods help check and convert tokens to a specific type:

- *checkTokenMatch(String line, List tokens, int count, String match)* - Checks that the token number in the array of tokens matches the passed in string.
- *checkTokenMatch(String line, String token, String match)* - Checks that a token matches the passed in match string. Note that this check is case-insensitive.
- *getBigDecimalNumberFromString(String line, String token)* - Checks that a token is a BigDecimal number and returns it.
- *getDoubleNumberFromString(String line, String token)* - Checks that a token is a double number and returns it.
- *getEnum(Class enumClass, String enumStringValue, String line)* - Instantiates an enum of the specified type with the value specified. First, the string version of the value is attempted. If that fails, then the Integer version of the value is attempted. If that fails, null is returned.
- *getHexadecimalNumberFromString(String line, String token)* - Checks that a token is a hexadecimal number and returns it.
- *getIntegerNumberFromString(String line, String token)* - Checks that a token is an integer and returns it.
- *getIpAddressFromString(String line, String token)* - Checks that a token is an IP Address and returns it.
- *getLongNumberFromString(String line, String token)* - Checks that a token is a long number and returns it.
- *getMacAddressFromString(String line, String token)* - Checks that a token is a 48-bit mac address and returns it.

## 7.6 Token Validation

---

Once a token is obtained, the following methods help validate the token:

- *checkExists(Object object, String description)* - Checks to see if an object exists or not. If the object doesn't exist, an `InvalidConfigurationException` is thrown using the passed in description.
- *checkNotEmpty(String value, String description)* - Checks to see if the value is not empty. If the object is empty, an `InvalidConfigurationException` is thrown using the passed in description.
- *checkRange(Number value, Number minRange, Number maxRange, boolean required, String description)* - Checks that the value is possibly not null and within the specified range.
- *isEmpty(String string)* - Determines if the passed in string is empty.

## 7.7 InvalidConfigurationException Generation

---

The following methods help generate `InvalidConfigurationException`'s:

- *invalidConfiguration(String errorMessage, String line)* - Throws an `InvalidConfigurationException` with a specified error message. The IOS configuration line will be appended automatically.
- *invalidTokenFound(String line, String token)* - Throws an `InvalidConfigurationException` because an invalid token was found.

## 8. Adding New Components

---

The Cisco Smart Update SDK contains enough built-in functionality to meet most users' needs. However, there may be times when users will want to extend the toolkit to add custom functionality. The most common reason to extend the SDK is to add additional components that are not currently supported by the SDK. This chapter discusses the steps required to add new components.

### 8.1 Create A New Component

---

To add a new component, create a class that extends `CiscoRouterConfigComponent`. This should typically be placed within a customer specific domain package as opposed to a "com.foxsmart" package. The name of the class must match the first key word of IOS configuration name followed by the word "Component" (e.g. "line con 0" has "line" as the first key word so the component should be called "LineComponent"). If a hyphen is present in the IOS configuration key word, camel notation should be used for each part of the hyphen (e.g. "exec-timeout" should be called "ExecTimeoutComponent").

One key word is usually sufficient to create unique components that aren't too big or cumbersome in their configuration. However, there are times when the first key word has too many configuration pieces to make it useful. In these situations, the first two key words can be used. For example, in the global configuration, "ip" is the first key word for dozens of configuration items. This would be too much for one component to handle so two key words should be used (e.g. "ip alias" should be called "IpAliasComponent" and "ip multicast-routing" should be called "IpMulticastRoutingComponent").

Depending on the component's functionality, it should implement the appropriate component interfaces as needed. See "Section 5.2.2 - Component Interfaces" for more information on which interfaces need to be implemented.

Once the component exists in the appropriate package, the SDK will automatically try to instantiate the component when an IOS configuration line is parsed that has the matching key words. It will also automatically call the appropriate methods on the class to perform various functions.

### 8.2 Add Component Properties

---

The configuration properties that are specific to the component should be added next. In addition to the properties, appropriate getter and setter methods should be added to get and set the custom properties. Since the required properties are unique to each component, it will be up to the class implementer to decide the best way to work with custom properties. Refer to existing SDK components for various example of how properties can be specified and configured.

## 8.3 Add Required Component Methods

---

At a minimum, the core abstract methods on `CiscoRouterConfigComponent` must be added. For a description of these methods, refer to “Section 7.1 - CiscoRouterConfigComponent Abstract Methods”.

The *buildFromIos* method needs to parse the list of IOS configuration lines that apply to that component, populate the component properties with values from the configuration, and remove the lines from the list that it processed. Use base class methods in `CiscoRouterConfigParser` to help parse the IOS configuration lines (see “Chapter 7 - Parsing, Validating, and Generating IOS” for more information on the parser helper methods). If any errors or unexpected configuration elements are found, `InvalidConfigurationException` should be thrown using the *invalidConfiguration* method in the base class.

The *validate* method needs to validate the properties of the component to ensure they are all valid. Validation will typically consist of the following types of checks:

- Required fields are present.
- Numbers fall within their allowed ranges.
- Dependent fields are present when necessary.
- Other parts of the router configuration are consistent with this component’s configuration. *getParentComponent* and *getCiscoRouterConfig* methods may be useful for navigating to other parts of the configuration hierarchy.

Refer to Section “7.6 - Token Validation” and Section “7.7 - InvalidConfigurationException Generation” for helper methods to validate and throw exceptions as appropriate. If the component requires no validation, this method can have no implementation.

The *generateIos* method should return the list of commands needed to generate the IOS configuration associated with the component.

## 8.4 Integrate Component into the SDK

---

There are two main integration points when adding a new component into the SDK: 1) “Package Integration” which informs the SDK which packages to try when instantiating components from a read router configuration, and 2) “Class Integration” which informs the SDK which component classes to look for when displaying configurations and when performing updates.

### 8.4.1 Package Integration

---

`CiscoRouterConfigComponent`'s that implement `CompositeComponent` are the parent components that are responsible for instantiating their children components when reading a configuration. All the `CompositeComponent`'s call the *instantiateComponent* methods on `CiscoRouterConfigComponent` when instantiating new components while building the configuration class hierarchy. This method takes a list of Java packages to use when instantiating the components. Each `CompositeComponent` within the SDK provides its own list of packages to use and typically consists of the core configuration package (i.e. "com.foxsmart.csu.config") and the package where its children are located (e.g. `LineComponent` uses "com.foxsmart.csu.config.global.line").

Although customers may choose to create new components in the same CSU SDK packages, this is not recommended given that these packages are reserved for Fox Smart's use and may change over time. Instead, there are two options to inform the SDK about the packages to check: 1) Add a static global list of packages to the `CiscoRouter` class or 2) Add the package to the specific parent `CompositeComponent` for the new child component. Only one approach is needed.

1) Global packages are tried first in the order they are listed before the `CompositeComponent`'s specified packages are attempted. They provide a convenient way to add a list of packages where all newly added components reside. However, this adds extra overhead to the SDK since the list of global packages are always tried first - even when instantiating core SDK components. Assuming the new component is located in a package called "com.customerdomain.csu", the following is how to add the package to the existing list of global packages:

```
List globalPackages = new ArrayList(CiscoRouter.getGlobalPackagesStatic());
globalPackages.add("com.customerdomain.csu");
CiscoRouter.setGlobalPackagesStatic(globalPackages);
```

2) To add the package to the parent `CompositeComponent`, the package needs to be added to the existing list of component specific packages. This means the existing list should be obtained, the new package added to the list, and the new list set on the component. Assuming the new component is located in a package called "com.customerdomain.csu" and it is being added to the `Interface` composite component, the following is how the package would be added:

```
List interfacePackages = new ArrayList(InterfaceComponent.getPackagesStatic());
interfacePackages.add("com.customerdomain.csu");
InterfaceComponent.setPackagesStatic(interfacePackages);
```

## 8.4.2 Class Integration

---

When displaying a router configuration from a class hierarchy and when determining which commands to issue while performing updates, a `CompositeComponent` maintains two lists of keys which are returned using the *getChildComponentKeys* and *getChildComponentUpdateKeys* respectively. Both lists are typically the same although two lists are maintained in case the update order processing varies from the display order processing. Also, keys are a combination of "component classes" for supported classes and "token strings" for unsupported components. Assuming the new component is in a class called `TestComponent`, and it is being added to both lists within the `Interface` composite component, the following is how the classes would be added:

```
// Add class for displaying configurations
List componentKeys = new ArrayList(InterfaceComponent.getChildComponentKeysStatic());
componentKeys.add(TestComponent.class);
InterfaceComponent.setChildComponentKeysStatic(componentKeys);

// Add class for performing updates
List componentUpdateKeys =
```

```

    new ArrayList(InterfaceComponent.getChildComponentUpdateKeysStatic());
    componentUpdateKeys.add(TestComponent.class);
    InterfaceComponent.setChildComponentUpdateKeysStatic(componentUpdateKeys);

```

## 8.5 DatabitsComponent Example

---

This section shows the `DatabitsComponent` of the line portion of the IOS configuration. This component is part of the Cisco Smart Update SDK and demonstrates what a simple component class looks like.

The IOS configuration for this component looks like the following:

```
databits <number of data bits>
```

This component has only one property (`numBits`) which must have a value between 5 and 8. The following is the component class definition:

```

// Package
package com.foxsmart.csu.config.global.line;

// Imports
import java.util.*;
import com.foxsmart.csu.config.*;

/**
 * The data bits class sets the number of data bits per character.
 */
public class DatabitsComponent extends CiscoRouterConfigComponent
{
    // The number of bits
    private Integer numBits = null;

    // Configuration Constants
    public static final String DATABITS = "databits";

    /**
     * The minimum number of data bits.
     */
    public final static Integer MIN_BITS = new Integer(5);

    /**
     * The maximum number of data bits.
     */
    public final static Integer MAX_BITS = new Integer(8);

    /**
     * Constructs a data bits object.
     */
    public DatabitsComponent()
    {
    }

    /**
     * Sets the number of bits. Valid values are between 5 and 8.
     *
     * @param numBits The number of bits.
     */
    public void setNumBits(Integer numBits)
    {
        this.numBits = numBits;
    }
}

```

```

    * Gets the number of bits.
    *
    * @return the number of bits.
    */
public Integer getNumBits()
{
    return numBits;
}

/**
 * @see com.foxsmart.csu.config.CiscoRouterConfigComponent
 */
public List buildFromIos(List configLines) throws InvalidConfigurationException
{
    // Get one line.
    String line = getLine(configLines, 0);

    // Split the line into tokens.
    List tokens = getTokens(line);

    // Get the first token.
    String token = getNextToken(line, tokens);

    // Check that the first token is our component keyword.
    checkTokenMatch(line, token, DATABITS);

    // Get the next token and parse it as an Integer.
    token = getNextToken(line, tokens);
    Integer numBits = getIntegerNumberFromString(line, token);

    // Check to make sure there are no more tokens.
    checkLineComplete(line, tokens);

    // Store the information.
    setNumBits(numBits);

    // Remove the first line from the list of lines.
    configLines = removeLines(configLines, 1);

    // Return the remaining list of lines.
    return configLines;
}

/**
 * @see com.foxsmart.csu.config.CiscoRouterConfigComponent
 */
public void validate() throws InvalidConfigurationException
{
    checkRange(numBits, MIN_BITS, MAX_BITS, true, "number of data bits");
}

/**
 * @see com.foxsmart.csu.config.CiscoRouterConfigComponent
 */
public Commands generateIos() throws InvalidConfigurationException
{
    // Ensure the object is valid.
    validate();

    // Initialize the IOS commands
    Commands commands = new Commands();

    // Add the command.
    // Output is only generated if the bits are not equal to 8.
    if (!(numBits.equals(MAX_BITS)))
    {
        commands.add(new Command(" " + DATABITS + " " + numBits));
    }

    // Return the IOS commands
    return commands;
}

```

```

    }
}

```

## 8.6 Implementing CompositeComponent

---

If the new component is a parent component that contains children, it must implement the `CompositeComponent` interface. Implementing this interface requires the addition of the following methods:

- *generateHeaderlos()* - This method should return the IOS String for the header portion of the component. This header is the configuration that makes up the component itself without its children. For example, the `LineComponent` would return the line itself (e.g. `line con 0`) without the children.
- *getPackages()* - This method should return the list of packages that are used to instantiate this component's children. This can return an empty List if all the children are placed in the global packages specified previously.
- *getChildComponentKeys()* - This method should return a list of the child component keys in the order they should be displayed. The individual keys should be a list of the component classes (e.g. `DatabitsComponent.class`) and/or Strings (e.g. "ip") if a particular class isn't supported.
- *getChildComponentUpdateKeys()* - This method should return a list of the child component keys in the order they should be processed during an update. In most cases, this method can simply delegate to the *getChildComponentKeys()* method.

In addition, the following base class methods may be useful:

- *validateChildren()* - This method can be called by a component's `validate` to validate its children after it has finished validating its own data. The default implementation calls the *validate()* method on all the component's children.
- *generateCompositelos(CiscoRouterConfigComponent component)* - When the standard *generatelos()* method is called on a `CompositeComponent`, it should return the entire IOS configuration for this component (i.e. the IOS configuration for the header and for all its children). `CompositeComponent`'s should typically just delegate to this method by passing in "this" to perform this processing automatically.
- *generateUpdateCompositelos(CiscoRouterConfigComponent component)* - Like the above method, this method provides default processing for the *generateUpdatelos()* method.

The *buildFromlos(List configLines)* method of a `CompositeComponent` needs to build its own configuration, but also the configuration of its children. This is typically done by first processing the first line of the configuration for the component itself, and then using the *moreIndentedLines(List lines, int originalIndentPos)* method to process the remaining lines. The following is an example of the code that processes the remaining lines.

```

// Get the index of the first non-space character of the first line
// to determine the original indent position
int originalIndentPos = StringUtils.indexOfNonSpace(line);

// Process the indented lines
while (moreIndentedLines(configLines, originalIndentPos))
{
    // Get one line
    line = getLine(configLines, 0);

    // Split the line into tokens
    tokens = getTokens(line);

    // Reset the token index

```

```

resetTokenIndex();

// Get the first token
token = getNextToken(line, tokens);

// If first token is the "no" keyword, use the 2nd token
if (token.equals(NO))
{
    // Get the next token
    token = getNextToken(line, tokens);
}

// Get the first and possibly second tokens to determine which component to use.
String firstConfigKey = token;
String secondConfigKey = "";
if (moreTokens(tokens))
{
    // Get the next token
    secondConfigKey = getNextToken(line, tokens);
}

// Create a component based on the token
List instantiateTokens = new ArrayList();
instantiateTokens.add(firstConfigKey);
instantiateTokens.add(secondConfigKey);
CiscoRouterConfigComponent component = instantiateComponent(instantiateTokens, getPackages());

if (component == null)
{
    // The component couldn't be instantiated so create an UnknownComponent.
    component = new UnknownComponent();
}

// Build this component
configLines = component.buildFromIos(configLines);

// Store the configuration component
if (component instanceof UnknownComponent)
{
    addComponent(firstConfigKey, component);
}
else
{
    addComponent(component);
}
}

```

## 8.7 Implementing MultiInstanceComponent

---

If the new component can have more than instance of itself, it must implement the `MultiInstanceComponent` interface. Note that each instance of a component must have identical properties to the next for it to be considered a `MultiInstanceComponent` (e.g. lines, interfaces, etc.). If a component contains multiple lines, but each line has different types of information, it should be configured as a `MultiLineComponent` and not a `MultiInstanceComponent`. There might be some cases where a component that contains multiple lines with each line containing the same properties elects not to be a `MultiInstanceComponent` (e.g. access lists). This would done to provide more granular API's than are available in `CiscoRouterConfigComponent` and to more easily validate and generate IOS configurations given that one instance of the component has easy access to the entire set of data for that component. Implementing this interface requires the addition of the following methods:

- equalsComponentInstance(CiscoRouterConfigComponent component)*** - This method should return whether the passed in component of the same type is the same instance of the component itself. An "instance" in this case refers to the same configuration item in the IOS configuration (e.g. a component would be the same instance if this component

and the passed in component are both for "line con 0". If one component was for "line aux 1" and the other was for "line con 0", then false would be returned).

- *requiresCommentSeparator()* - This method should return whether a comma separator should be generated in between instances of the component. This is used by the core SDK classes as a way to generate a line with a single "!" character to separate out individual instances of the component. This is for aesthetics only and does not have any functional impact.

## 8.8 Implementing MultiLineComponent

---

If the new component can have more than one that makes up a single instance of the component, it must implement the `MultiLineComponent` interface. For example, the `ServiceComponent` is a `MultiLineComponent` because it has many individual lines that make up the entire component's configuration where each line individually contains different types of data. Implementing this interface requires the addition of the following methods:

- *generateDeletelos(CiscoRouterConfig updatedCiscoRouterConfig)* - This method should return the IOS commands necessary to delete all parts of the component's configuration. Unlike a standard component, this is needed since a simple "no" followed by the *generatelos()* output won't delete the entire configuration given that the component's configuration contains multiple lines.

## 8.9 Implementing IndelibleComponent

---

If the new component is one that can't be deleted, it must implement the `IndelibleComponent` interface. This interface does not require any additional methods to be implemented, but its existence instructs the core SDK methods to not attempt to delete the component.

## 8.10 Advanced Methods To Override

---

When a configuration is read and written, Cisco Smart Update SDK performs all the core processing and uses the component's individual methods (described in the previous sections) to handle the portions of processing that are specific to each component. Although this is sufficient for most components, there are times when more granular control is needed by a component. The following methods exist in the base `CiscoRouterConfigComponent` and can be overridden as needed for individual components to obtain this control:

- *getUpdateConfiglet(CiscoRouterConfigComponent origComponent)* - This is the method that is initially called for a component to return the necessary update commands needed to update the configuration to its desired state. The original component's information is passed in as a way for the component to compare the original configuration with the new one. This method provides default processing for all types of components and delegates to other methods for the individual actions needed so it is unlikely that this method will need to be overridden, but it is mentioned here just in case.
- *generateChildrenUpdateConfiglet(CiscoRouterConfigComponent origCompositeComponent)* - This method is called to process updates for the children of

`CompositeComponent`'s. Like the `getUpdateConfiglet()` method above, this method handles all types of components and delegates to other methods for the individual actions needed so it is unlikely that this method will need to be overridden.

- `checkComponentType(CiscoRouterConfigComponent component, Class compareClass)` - This method checks to see if the passed in component is of the specified type. If the component is not the correct type, an `InvalidConfigurationException` is thrown. If the type is correct, the component is simply returned where the caller can safely type cast it to the desired type.
- `generateDeletelos(CiscoRouterConfig updatedCiscoRouterConfig)` - Although a `MultiLineComponent` requires this method to be implemented, any component may choose to override this method to perform custom deletion functionality. By default, this command will place a "no" keyword in front of the component's header IOS or standard IOS. Although this functionality may be correct for most components, individual components that require special deletion processing should override this method. Note that the default implementation also sets the "deferred" flag on the returned commands based on the result of the `getDeferredDelete()` method.
- `generateUpdatelos()` - By default, this method delegates to the `generatelos()` method. In other words, the list of commands that view the IOS configuration would be the same as the commands needed to update the IOS configuration. Components that want to handle views different from updates should override this method to perform special processing.
- `generateMultiLineUpdateConfiglet(CiscoRouterConfigComponent origComponent)` - For `MultiLineComponent`'s, this method is called to return the update commands necessary to update the component. The default version of this method calls `generateDeletelos()` first to delete the original configuration followed by `generateUpdatelos()` to generate the update statements to add the new configuration. Note that the `generateDeletelos()` won't be called when the component implements `IndelibleComponent` or when the `deleteOldConfigWhenUpdating()` method returns true (default). Although this processing may work for most `MultiLineComponent`'s, individual components may choose to override this method with custom processing.
- `deleteOldConfigWhenUpdating()` - This method determines whether delete commands will be issued before the update commands when processing this component. The default implementation of this method returns true. Components may choose to override this method to return false in special situations. For example, if deleting this component will cause the router to implicitly delete other configuration entries that are not desired (e.g. `EncapsulationComponent`), then this method should return false. This flag has no effect when a component implements `IndelibleComponent`. Note that this flag only applies to how deletes are handled when a component still exists and needs to be updated. When a component is solely being deleted (i.e. it is being removed from the configuration), the `generateDeletelos()` method will be called regardless.
- `getDeferredDelete()` - This method can be overridden to have the delete commands associated with a component deferred until all other commands have been issued. This can be useful when deleting a component might negatively affect another component until the other component has a chance to have been processed first. The default implementation of this method returns false.
- `instantiateComponent(List tokens, List packages)` - This method takes a list of tokens and a list of packages and attempts to instantiate the correct component to process that part of the configuration. Packages are attempted in the order they are listed and multiple tokens are concatenated together to find a matching class.