krugle

Krugle Enterprise SCMI Integration Guide

© 2008 Krugle, Inc. All Rights Reserved

V 2.0.1

Contents

Introduction

This document describes the Krugle SCM Integration (SCMI) interface. An SCMI application manages the continuous integration between Krugle Enterprise and a code / data repository. This document summarizes the SCMI architecture as well as the application programming interface of the SCMI software. This document is designed for anyone evaluating Krugle SCMI integration as well as for developers involved in the SCMI integration effort.

SCMI Concepts

Krugle Enterprise Operation Overview

Krugle Enterprise software runs on a secure appliance inside of your organization's firewall. Krugle Enterprise maintains a comprehensive index of your organization's source code, code metadata and related documentation. To keep this information current, Krugle Enterprise periodically "crawls" repositories to collect the most current files and data.

During the "crawl" processes for a specified project, Krugle Enterprise must access:

- All files that have been created or modified since the last indexing process.
- The list of files for the specified project that have been deleted since the last indexing process History comments (i.e. checkin comments).

There are two mechanisms that Krugle Enterprise uses to get access to code, code metadata and other content:

- Direct SCM support - for directly supported SCMs, the connection between Krugle Enterprise and each SCM is configured and managed completely within the Krugle Enterprise appliance. See the Krugle Enterprise Administration Guide for more information about using direct SCM support.
- SCMI integration - SCMI application software runs outside of the Krugle Enterprise appliance - on your organization's own hardware. If you have special access or management requirements for your SCM or data source, or if you want to integrate with a data source or SCM not directly supported by Krugle Enterprise, SCMI integration will meet your needs.

This document describes SCMI software integration mechanism.

SCMI Operation Overview

The SCMI application that you create will allow Krugle Enterprise to access code files and history comments from a code repository of your choice. SCMI software handles file/data access requests from Krugle Enterprise and helps ensure that the correct information is uploaded to Krugle Enterprise.

The following example shows how Krugle Enterprise uses SCMI software to access code files and repository comments. This is a common task performed by Krugle Enterprise when it needs to bring new and updated information (code files, comments) about a project into its index. The process begins when Krugle Enterprise issues a Files request for the latest files to the SCMI application. In subsequent steps, Krugle Enterprise and SCMI exchange information and files in an orderly process. See Figure 1 for a diagram of messaging and file exchange between Krugle Enterprise and the SCMI application.Krugle Enterprise uses the **History** request to access the SCM comments posted for check - in activity that occurred between the two most recent checkpoint markers.

1. Krugle Enterprise sends a **Files** request to the SCMI application. Note: the protocol, server, port, address and any necessary credentials used to envoke the SCMI application must first be specified in Krugle's Project definition.
2. SCMI maps the Krugle Enterprise request into a form that can be handled by the particular SCM / file system being accessed and issues the file checkout request to the SCM / file system. The SCMI passes a response to the **Files** command that includes the result file list and a checkpoint token identifying the most version of files accessed. Krugle stores this token to define the "starting point" for the next update.
3. Krugle Enterprise issues the **FilesRetrievalComplete** command to notify the SCMI that the Files request was successfully completed.

The calling and return xml for the commands used in the preceding example are included in the SCMI Command Reference section below.

The SCMI architecture is designed to provide flexibility in several key areas. Options include:

Repository types - repositories can include SCM systems, document systems, file systems and databases. If a repository can be configured to return file lists or data as xml, then that repository can be integrated with Krugle Enterprise through SCMI.

Repository configuration data - Information needed to access the files or records in a particular system (i.e. network address, path, credentials, access options, etc.) can be stored and managed from the Krugle Enterprise Administration Console - or from the SCMI application. For the former option, any necessary options are managed through the “SCMI project parameter” field that is specified, stored and managed by in the Krugle Enterprise project definition. Alternatively, some or all of this information can be stored/accessed on the SCMI host in a manner that is suitable for your organization.

File access - Files to be accessed by KE can be cached by the SCMI or accessed directly from the file / record repository. The technique that you choose must be reflected in the URLs that are returned from the SCMI in the file list. In the example above, the files are cached on the SCMI host (and the URLs that are returned to Krugle Enterprise refer to the file locations on the SCMI host). Another approach would be to specify that files / records be accessed directly from the SCM system, in this case the SCMI is configured to return URLs that point directly at the files on the SCM system. Both approaches have advantages, the caching method is generally more versatile, but it also can be more complicated; the SCMI must maintain some state information to know what resources keep available for download by the KE appliance. You can choose the approach that best suits your requirements.

Note: When using an SSH connection type or the SCP file transfer method on a unix based SCMI host, create a new user account that: a) is free of startup scripts and b) uses bash as the default shell.

A unique instance of SCMI software is required for each repository type. Multiple instances of a single SCMI script can be accessed by one or more Krugle Enterprise appliances. Multiple SCMI scripts can be deployed on a single SCMI host system. The SCMI modules available from Krugle target specific source code management systems (such as SVN and the file system) and are written in languages that are easily integrated through http(s) and ssh. Other languages can be used to implement the SCMI functionality.

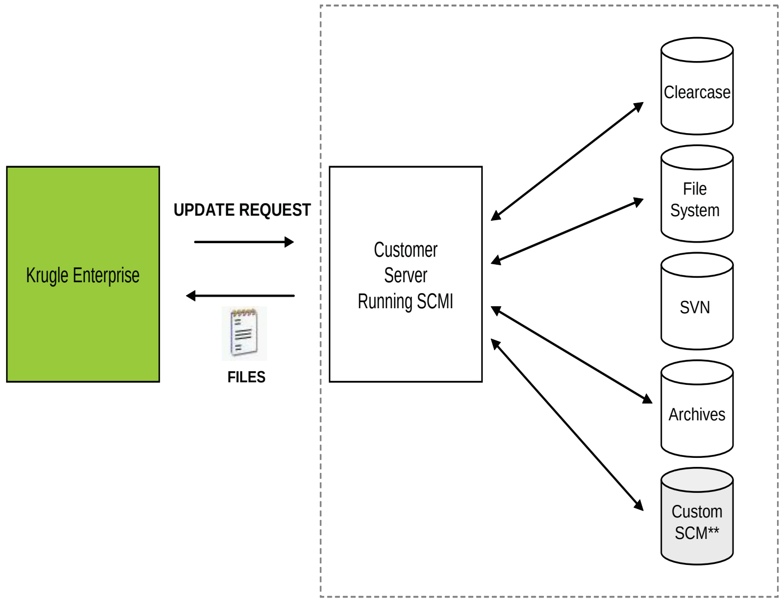

Figure 1

SCMI Deployment and Configuration - Implementation

The SCMI installation and configuration process:

1. Provision the SCMI host with: (i) a supported operating system of your choice, (ii) the scripting tools (i.e. Python, .net) that will run the SCMI script and (iii) any SCM client software that will be required to access information from the target SCM.
2. Transfer the SCMI script onto the SCMI host. The SCMI scripts are located at http://www.krugle.com/products/enterprise_documentation.html.
3. Modify the SCMI script to call the target data source / SCM system. Write down the location of the script for use in steps 4 & 5.
4. Test the SCMI application (see step 4 in the following example). First, verify that the SCMI script can be invoked on the SCMI host. Next verify that the SCMI script can be invoked over a network, using any required credentials.
5. In the Krugle Enterprise Administration Console: create a Project that uses this SCMI application as an SCM repository (See the Krugle Administration Guide Chapter 1 to learn about specifying a project). NOTE: if the SCMI host (the computer that will run the SCMI script) requires credentialed access, a username/password pair that will provide access required to run the SCMI script will need to be specified in the Krugle Enterprise definition of the SCM repository.

6. Check the Project Summary page in the Krugle Enterprise Administration Console to verify that the Project from step 5 has crawled correctly, and execute a Krugle client search in the “Code” channel that should produce a result file from the SCMI accessed data.

SCMI Deployment and Configuration Requirements

- A machine designated as the “SCMI Host” (NOTE: It is not required that it is dedicated to Krugle)
- Specification for SCMI Host:
  - OS – Windows, Linux, Unix, Mac OSX.
  - RAM – 2G, with dual core processor.
  - Disk space – Minimum 30% greater than size of total code base to be crawled, if caching required (caching is not required for Clearcase dynamic views and Synergy).
- Python v2.3 or greater installation on SCMI Host (assuming SCMI scripts are used as is).
- Network Bandwidth – SCMI Host connected via minimum 100 MB ethernet to Krugle appliance and SCM server.
- Access via ssh or http(s) from Krugle appliance to SCMI Host.
- Username/password on SCMI Host that is required by Krugle appliance to run SCMI scripts.
- SCM client installation on SCMI Host, if applicable (required for Clearcase).
- Username/password required by SCM client to access SCM server and corresponding SCM user license, if applicable.

SCMI Example - Linux Filesystem

This section provides a detailed, step-by-step example of how SCMI is used to integrate with Krugle Enterprise through SSH. This example uses Python to implement an SCMI integration to the Linux file system. This script allows Krugle Enterprise to crawl and index all files at (and beneath) a specified root directory. The script to perform this integration is available from Krugle, and as the instructions below show, this sample script requires only several edits and configuration steps to be operational.

NOTE: SCMI can be used to integrate Krugle Enterprise with other SCM systems, on other operating systems (Windows), using different scripting languages (Perl, .net, etc.) and different communication protocols. This example illustrates just one possible configuration option.

1. **Verify SCMI host configuration** (example: linux).

1a. Verify that Python 2.3 (or later) is configured on the SCMI host. To test this, execute a shell command that will confirm the presence of python - e.g. :

```
> python -V
```

1c. Verify that SSH is available on the SCMI host. Execute a test command such as the following:

```
> ps ax |grep sshd
```

2. **Access the SCMI template script.**

2a. Navigate to http://www.krugle.com/products/enterprise_documentation.html

2b. Click the SCMI download link; this will download a compressed folder archive Krugle - SCMI - Examples to your computer.

2c. Unpack the Krugle-SCMI-Examples archive and place the contents of the python folder in a directory; note the name for use in subsequent steps, e.g.

/home/krugle_scmi/scripts

2d. Make sure that the SCMI script is executable on the SCMI host; the following command should return to the shell without error:

> chmod u+x gateway-filesystem.py

3. **Edit gateway-filesystem.py**

3a. Enter the host name of the SCMI host (italicized in the sample line below). If the SCMI host has a static IP address, you can enter the address or a domain name that can be resolved to an IP address on the network that the SCMI host is connected to.

“host” : “*your_host_name*”

3b. Enter the root location of the SCMI working directory. Using the locations from 2c above
“rootScmiPath” : */home/krugle_scmi*

3c. Save changes to gateway-filesystem.py

4. Verify proper access and operation of the SCMI script.
On the SCMI host machine, in the scripts directory (from step 2c above) open test-files-req.xml and type in:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<files-request version=”1”>
   <project>
           <serverUid>default</serverUid>
           <projectUid>krugle-fs-test</projectUid>
           <location>/bin/</location>
           <params></params>
   </project>
</files-request>
```

Save the edits to the file and test the SCMI script from the shell:

```
> cat test-files-req.xml | ./gateway-filesystem.py
```

The stdout response should be XML formatted in a files-response.

Once it is verified that the SCMI script operates properly from the SCMI host, test the script over the network. First copy the test-files-req.xml over to another machine, and then from that machine:

```
> cat test-files-req.xml | ssh <user>@<host> /absolute/path/to/executable
```

5. **Identify the root directory of code files that you want crawled by Krugle Enterprise**
For this example we assume that the code is located:
/code_files

6. **Setup a Project in Krugle Enterprise to crawl and index the code files**

6a. Sign in to the Krugle Enterprise Hub.

6b. Create a new project. Click the projects tab. Click “Add New Project”. Add required project metadata.

6c. Under the “Add SCM location to project definition” section, select create new repository from the dropdown list.

6d. Choose SCMI from the SCM type and enter the host name that was specified in step 3a above. Click Next.

6e. On the SCMI detail screen, provide a name for the Repository; specify the path to the SCMI working directory - for this example, this is the path specified in step 3b above. Specify the connection type as SSH. Click Save.

6f. In the “Add SCM location to project definition” section, specify the path location from step 4 above. Click Add.

This completes the setup. Krugle Enterprise will crawl and index the code located at the path specified in step 4. Within several minutes, the code will be available for client searching.

SCMI Command Reference

Files command

This request is sent by the Krugle Enterprise to synchronize itself with the latest files available on the SCMI. Depending on if **lastFilesCheckpoint** is present in the request, this comment can represent either a request for all the files, or just the changes since the given **lastFilesCheckpoint** token. In SVN terms, sending a request without a lastFilesCheckpoint is asking for a “checkout”, while including the lastFilesCheckpoint is asking for an “update” from that point.

A checkpoint is an arbitrary string chosen by the SCMI to represent a point in time. The SCMI returns a checkpoint with the Files response, expecting the KE to store it and send it back with the next Files request (as well as next History request, see next section). For example in SVN you could use the repository revision for the checkpoint, but in CVS you would want to use the date, because in CVS revisions are file specific. Using the lastFilesCheckpoint field correctly will make the files update process efficient because generally less files will have to be retrieved. Although the field is required in the files response, it can be set to anything and ignored, if the SCMI author wishes.

Use UTF-8 encoding for all XML communications between Krugle Enterprise and the SCMI script.
File URLs sent from the SCMI to Krugle (http, https or scp) must be URL encoded.

Call

XML	Comments

`<?xml version=”1.0” encoding=”UTF-8”?>`
`<files-request version=”1”>`
`    <project>`
`        <serverUid>Server1</serverUid>`
`        <projectUid>Project1</projectUid>`
`        <location>svn://svn.host.net/repo</location>`
`        <params>u$3rn4m3 p4$$w0rd</params>`
`    </project>`
`    <lastFilesCheckpoint>2311</lastFilesCheckpoint>`
`</files-request>`
	The “serverUid” element will contain an identifier unique to each KE Appliance.
	The “projectUid” element will contain a string which identifies this project uniquely in the scope of the KE Appliance.
	The “location” is the user defined text provided as part of the KE Appliances’s project configuration. It should give a path to the resources the SCMI will process.
	The “params” is another string which is defined by the user in the project configuration. It should specify any information needed in addition to the location, like credentials.
	The “lastFilesCheckpoint” is the token that was returned as part of the last update response. If this token is omitted the call, the response will include the full list of current files.

Response

XML	Comments

`<?xml version="1.0" encoding="UTF-8"?>`
`<files-response version="1">`
`    <project>`
`        <serverUid>Server1</serverUid>`
`        <projectUid>Project1</projectUid>`
`        <location>svn://svn.host.net/repo</location>`
`        <params>u$3rn4m3 p4$$w0rd</params>`
`    </project>`
`    <files>`
`        <file>`
`            <action>ADDED</action>`
`            <name>/path/to/file.txt</name>`
`            <url>http://server:port/root/path/to/file.txt</url>`
`            <md5>d41d8cd98f00b204e9800998ecf8427e</md5>`
`            <revision>345</revision>`
`        </file>`
`        <file>`
`           ...`
`        </file>`
`    </files>`
`    <filesCheckpoint>2601</filesCheckpoint>`
`</files-response>`
	Zero or one "files" elements may be present.
	Zero or more "file" elements may be present in each "files" elements.
	The "action" must be "Added", "Removed" or "Updated".
	The "name" element contains the path of the file relative to the project root.
	The "URL" element contains the full URL that can be used to retrieve the file / record. Not present for files that are being REMOVED. Examples of well formed URLs: "http://server1:80/java/com/krugle/helper/save.txt" or "ftp://public:password@ftp1:21/java/com/krugle/helper/save.txt" or "SSH://server:port/command java/com/krugle/helper/save.txt"
	OPTIONAL: If present, the "revision" element contains a revision string which will be added to the index.
	OPTIONAL: If present, the "md5" element contains the MD5 hash of the file contents, but the md5 element can be omitted.
	The "filesCheckpoint" is the token which the KE will store and send back to the SCMI in the next files-request in the lastFilesCheckpoint.

HISTORY command

The history request asks for the SCM history, both the total file changes and any comments associated with these changes, from the SCMI. The request always includes a lastFilesCheckpoint and may also include a lastHistoryCheckpoint. These checkpoints specify the bounds of the request. If the lastHistoryCheckpoint element is missing, the response should be composed of all the SCM history from the start of the project up to and including the lastFilesCheckpoint. If the lastHistoryCheckpoint is present, the history response will include the history from just after the lastHistoryCheckpoint up to and including the lastFilesCheckpoint.

The lastFilesCheckpoint is the token returned to Krugle Enterprise during the last files response, the lastHistoryCheckpoint is the checkpoint token returned to Krugle Enterprise in the last history response, if one has been made for this project.

The response provides a list of changeSet objects, where a changeSet represents a batch of file changes. Each changeSet has an id field which can be any string to uniquely identifies the changeSet in the context of the project in Krugle Enterprise. In SVN this would be a revision. Each changeSet has a date and author field, which would be the date and author of the revision used for the id field. Inside of each changeSet is a list of file elements which will enumerate every file change that occurred in the revision specified by the id element. Things are more complicated for a SCM such as CVS which does not have a concept of changeSets, but instead gives every file its own revision count. For CVS you could either create a changeSet for every single file change, or better yet group comments together that were made at the same time by the same author. The id of the changeSet in this case should be the commit number, which isn't tracked by CVS but could be inferred during this comment grouping process.

Use UTF-8 encoding for all XML communications between Krugle Enterprise and the SCMI script.

Call

XML	Comments

`<?xml version=”1.0” encoding=”UTF-8”?>`
`<history-request version=”1”>`
`    <project>`
`        <serverUid>Server1</serverUid>`
`        <projectUid>Project1</projectUid>`
`        <location>svn://svn.host.net/repo</location>`
`        <params>u$3rn4m3 p4$$w0rd</params>`
`    </project>`
`    <lastHistoryCheckpoint>2311</lastHistoryCheckpoint>`
`    <lastFilesCheckpoint>2601</lastFilesCheckpoint>`
`</history-request>`
	The “lastHistoryCheckpoint” is equal to the historyCheckpoint returned from the SCMI on the last history response for this project, if one has been sent yet.
	
	The “lastFilesCheckpoint” is equal to the filesCheckpoint token returned from the SCMI on the last files response from this project.

Response

XML	Comments

`<?xml version=”1.0” encoding=”UTF-8”?>`
`<history-response version=”1”>`
`    <project>`
`        <serverUid>Server1</serverUid>`
`        <projectUid>Project1</projectUid>`
`        <location>svn://svn.host.net/repo</location>`
`        <params>u$3rn4m3 p4$$w0rd</params>`
`    </project>`
`    <changeSets>`
`        <changeSet>`
`            <id>2312</id>`
`            <date>2001-10-26T21:32:52+02:00</date>`
`            <comment>check-in comment</comment>`
`            <author>The change author</author>`
`            <files>`
`                <file>`
`                    <action>ADDED</action>`
`                    <name>/path/to/file.txt</type>`
`                </file>`
`            </files>`
`        </changeSet>`
`        <changeSet>`
`             ...`
`        </changeSet>`
`    </changeSets>`
`    <complete>false</complete>`
`    <historyCheckpoint>new _TOKEN_</historyCheckpoint>`
`</history-response>`
	The “historyCheckpoint” element contains a string indicating the latest checkpoint that this response has information about.
	
	The “complete” element may contain either “true” or “false”. If “false” the appliance will process the response and then immediately send a history request with the new historyCheckpoint. This should be used to breakup huge histories.
	
	Zero or one “changeSets” elements may be present.
	
	Zero or more “changeSet” elements may be present.
	
	The “id” is unique text (in the scope of a project) that identifies this history event.
	
	The “date” element contains a xsd:dateTime formated date.
	
	The “comment” element contains text describing the change, typically a check-in comment.
	
	The “author” element identifies the author of the change.
	
	Zero or one “files” elements may be present within each “changeSet” element.
	
	Zero or more “file” elements may be present within the “files” element.
	
	The “action” must be “Added”, “Removed” or “Updated”.
	
	The “name” element contains the path to the file relative to the project root.

FILE RETRIEVAL COMPLETE notification

Sent by the SCMI application to Krugle Enterprise after all the files from the last files request have been downloaded. The SCMI must keep any file resources available until the file retrieval complete, update, or delete notification comes. It is important to assume that there will never be 2 Krugle Enterprise appliances asking for the same project at once. Some SCMI implementations will involve locks on resources, so this notification can be used as a trigger to unlock various ones.

Use UTF-8 encoding for all XML communications between Krugle Enterprise and the SCMI script.

Call

XML	Comments

```
<?xml version=”1.0” encoding=”UTF-8”?>
<fileRetrievalComplete-notification version=”1”>
    <project>
        <serverUid>Server1</serverUid>
        <projectUid>Project1</projectUid>
        <location>svn://svn.host.net/repo</location>
        <params>u$3rn4m3 p4$$w0rd</params>
    </project>
</fileRetrievalComplete-notification>
```

Response

XML	Comments

```
<?xml version=”1.0” encoding=”UTF-8”?>
<fileRetrievalComplete-response version=”1”>
    <project>
        <serverUid>Server1</serverUid>
        <projectUid>Project1</projectUid>
        <location>svn://svn.host.net/repo</location>
        <params>u$3rn4m3 p4$$w0rd</params>
    </project>
</fileRetrievalComplete-response>
```

DELETE notification

This is a courtesy notification from Krugle Enterprise to the SCMI application. This command is issued when Krugle Enterprise has successfully completed the file / record crawl process. In a robust SCMI implementation, projects can be deleted at any time to free up space, and they will be dynamically recreated upon a files or history request However - it is still most efficient to keep a project on the SCMI for every project which exists on a connected Krugle Enterprise appliance, and this notification type facilitates this synchronization.

Call

XML	Comments
`<?xml version=”1.0” encoding=”UTF-8”?>` `<delete-notification version=”1”>` `<project>` `<serverUid>Server1</serverUid>` `<projectUid>Project1</projectUid>` `<location>svn://svn.host.net/repo</location>` `<params>u$3rn4m3 p4$$w0rd</params>` `</project>` `</delete-notification>`	

```
<?xml version=”1.0” encoding=”UTF-8”?>
<delete-notification version=”1”>
    <project>
        <serverUid>Server1</serverUid>
        <projectUid>Project1</projectUid>
        <location>svn://svn.host.net/repo</location>
        <params>u$3rn4m3 p4$$w0rd</params>
    </project>
</delete-notification>
```

Response

XML	Comments
`<?xml version=”1.0” encoding=”UTF-8”?>` `<delete-response version=”1”>` `<project>` `<serverUid>Server1</serverUid>` `<projectUid>Project1</projectUid>` `<location>svn://svn.host.net/repo</location>` `<params>u$3rn4m3 p4$$w0rd</params>` `</project>` `</delete-response>`	

```
<?xml version=”1.0” encoding=”UTF-8”?>
<delete-response version=”1”>
    <project>
        <serverUid>Server1</serverUid>
        <projectUid>Project1</projectUid>
        <location>svn://svn.host.net/repo</location>
        <params>u$3rn4m3 p4$$w0rd</params>
    </project>
</delete-response>
```

SCMI - XML Schema

```
<xsd:schema xmlns:xsd=”http://www.w3.org/2001/XMLSchema” xmlns:jxb=”http://java.sun.com/xml/ns/
jaxb” jxb:version=”1.0”>
  <xsd:annotation>
    <xsd:documentation>
      blah blah blah
    </xsd:documentation>
    <xsd:appinfo>
    </xsd:appinfo>
  </xsd:annotation>
  <!-- Base Types -->
  <xsd:complexType name=”Project”>
    <xsd:sequence>
      <xsd:element name=”serverUid” type=”xsd:string” minOccurs=”1” maxOccurs=”1” />
      <xsd:element name=”projectUid” type=”xsd:string” minOccurs=”1” maxOccurs=”1” />
      <xsd:element name=”location” type=”xsd:string” minOccurs=”1” maxOccurs=”1” />
      <xsd:element name=”params” type=”xsd:string” minOccurs=”1” maxOccurs=”1” />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name=”EventType”>
    <xsd:restriction base=”xsd:string”>
      <xsd:enumeration value=”Added” />
      <xsd:enumeration value=”Updated” />
      <xsd:enumeration value=”Removed” />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name=”FileURL”>
    <xsd:restriction base=”xsd:string”>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name=”MD5”>
    <xsd:restriction base=”xsd:string”>
      <xsd:length value=”32” />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name=”ChangeSets”>
    <xsd:sequence>
      <xsd:element name=”changeSet” type=”ChangeSet” minOccurs=”0” maxOccurs=”unbounded” />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name=”ChangeSet”>
<xsd:sequence>
      <xsd:element name=”id” type=”xsd:string” />
      <xsd:element name=”date” type=”xsd:dateTime” minOccurs=”1” maxOccurs=”1” />
      <xsd:element name=”comment” type=”xsd:string” minOccurs=”1” maxOccurs=”1” />
      <xsd:element name=”author” type=”xsd:string” minOccurs=”1” maxOccurs=”1” />
      <xsd:element name=”files” type=”Files” minOccurs=”0” maxOccurs=”1” />
    </xsd:sequence>
```

```
</xsd:complexType>
<xsd:complexType name="Files">
  <xsd:sequence>
    <xsd:element name="file" type="File" minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="File">
    <xsd:sequence>
      <xsd:element name="action" type="EventType" minOccurs="1" maxOccurs="1" />
      <xsd:element name="name" type="xsd:string" minOccurs="1" maxOccurs="1" />
      <xsd:element name="url" type="FileURL" minOccurs="0" maxOccurs="1" />
      <xsd:element name="md5" type="MD5" minOccurs="0" maxOccurs="1" />
      <xsd:element name="revision" type="xsd:string" minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="ErrorType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="generalError" />
      <xsd:enumeration value="internalError" />
      <xsd:enumeration value="invalidConfiguration" />
      <xsd:enumeration value="invalidFilesCheckpoint" />
      <xsd:enumeration value="invalidHistoryCheckpoint" />
      <xsd:enumeration value="notReady" />
      <xsd:enumeration value="outOfMemoryError" />
      <xsd:enumeration value="protocolError" />
      <xsd:enumeration value="protocolVersionError" />
      <xsd:enumeration value="rebuildProject" />
      <xsd:enumeration value="scmAuthenticationError" />
      <xsd:enumeration value="scmConnectionError" />
      <xsd:enumeration value="volumeFullError" />
    </xsd:restriction>
  </xsd:simpleType>
  <!-- Commands -->
  <xsd:element name="files-request">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="project" type="Project" minOccurs="1" maxOccurs="1" />
        <xsd:element name="lastFilesCheckpoint" type="xsd:string" minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
      <xsd:attribute name="version" type="xsd:int" use="required" fixed="1" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="history-request">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="project" type="Project" minOccurs="1" maxOccurs="1" />
        <xsd:element name="lastHistoryCheckpoint" type="xsd:string" minOccurs="0" maxOccurs="1" />
        <xsd:element name="lastFilesCheckpoint" type="xsd:string" minOccurs="1" maxOccurs="1" />
      </xsd:sequence>
      <xsd:attribute name="version" type="xsd:int" use="required" fixed="1" />
    </xsd:complexType>
  </xsd:element>
```

```
<!-- Notifications -->
  <xsd:element name=”fileRetrievalComplete-notification”>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name=”project” type=”Project” minOccurs=”1” maxOccurs=”1” />
      </xsd:sequence>
      <xsd:attribute name=”version” type=”xsd:int” use=”required” fixed=”1” />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name=”delete-notification”>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name=”project” type=”Project” minOccurs=”1” maxOccurs=”1” />
      </xsd:sequence>
      <xsd:attribute name=”version” type=”xsd:int” use=”required” fixed=”1” />
    </xsd:complexType>
  </xsd:element>
  <!-- Commands Responses -->
  <xsd:element name=”history-response”>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name=”project” type=”Project” minOccurs=”1” maxOccurs=”1” />
        <xsd:element name=”changeSets” type=”ChangeSets” minOccurs=”0” maxOccurs=”1” />
        <xsd:element name=”complete” type=”xsd:boolean” minOccurs=”1” maxOccurs=”1” />
        <xsd:element name=”historyCheckpoint” type=”xsd:string” minOccurs=”1” maxOccurs=”1” />
      </xsd:sequence>
      <xsd:attribute name=”version” type=”xsd:int” use=”required” fixed=”1” />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name=”files-response”>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name=”project” type=”Project” minOccurs=”1” maxOccurs=”1” />
        <xsd:element name=”files” type=”Files” minOccurs=”0” maxOccurs=”1” />
        <xsd:element name=”filesCheckpoint” type=”xsd:string” minOccurs=”1” maxOccurs=”1” />
      </xsd:sequence>
      <xsd:attribute name=”version” type=”xsd:int” use=”required” fixed=”1” />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name=”error-response”>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name=”errorType” type=”ErrorType” minOccurs=”1” maxOccurs=”1” />
        <xsd:element name=”description” type=”xsd:string” minOccurs=”1” maxOccurs=”1” />
      </xsd:sequence>
     <xsd:attribute name=”version” type=”xsd:int” use=”required” fixed=”1” />
    </xsd:complexType>
  </xsd:element>
```

```
<!-- Notifications Responses -->
  <xsd:element name=”fileRetrievalComplete-response”>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name=”project” type=”Project” minOccurs=”1” maxOccurs=”1” />
      </xsd:sequence>
      <xsd:attribute name=”version” type=”xsd:int” use=”required” fixed=”1” />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name=”delete-response”>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name=”project” type=”Project” minOccurs=”1” maxOccurs=”1” />
      </xsd:sequence>
      <xsd:attribute name=”version” type=”xsd:int” use=”required” fixed=”1” />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

SCMI Error Handling

The SCMI module can return an error response to any request made by Krugle Enterprise. Several common error conditions are already defined, and some may trigger different behavior from Krugle Enterprise. This error handling helps ensure that Krugle Enterprise logging, error reporting and administrator notification swiftly and accurately dispatch the proper resources to remedy network, authentication and SCMI host processing issues that will adversely affect Krugle Enterprise operation.

Error handling

The following table summarizes the Krugle XML Schema ErrorType. Each error specified in the SCMI application must identify the error as one of the following types:

Error	Description	Krugle Enterprise Behavior
generalError	A general error type.	As a response to a files request, it puts the project in an error state. As a response to a history request or notification, it is merely logged.
internalError	Any error that occurs due to the SCMI application or any tool external to the Krugle Enterprise Appliance.	Same as generalError.
invalidConfiguration	The contents of the “location” or “params” element, as provide in the project configuration in the Appliance, is not valid or otherwise does not match what the SCMI is expecting.	Same as generalError.
invalidFilesCheckpoint	The filesCheckpoint given in a files request or history request is invalid.	Same as generalError.

Error handling table continued on next page

Error handling table continued from previous page

Error	Description	
invalidHistoryCheckpoint	The historyCheckpoint given in a history request is invalid.	Same as generalError.
notReady	The SCMI is busy doing something, and isn't ready to handle a command. This will NOT cause an error condition, but the crawl will be canceled for this project.	The crawl for this project is canceled but no error is registered.
outOfMemoryError	If the SCMI box runs out of memory trying to fulfill the request.	Same as generalError.
protocolVersionError	Unexpected protocol version.	Same as generalError, as there is only one protocol version right now.
protocolError	Anything else unexpected in the protocol. This would be considered a critical error as retrying the command would not be expected to have a different result. For example, invalid XML, unknown XML entities.	Same as generalError.
rebuildProject	Thrown if the SCMI wants KE to rebuild its files index by requesting all the files and history again.	The project is scheduled to be rebuilt as soon as the current snapshot finishes.
scmAuthenticationError	Thrown if the SCMI cannot authenticate to a remote SCM.	Same as generalError.
scmConnectionError	Thrown if the SCMI cannot connect to a remote SCM.	Same as generalError.
volumeFullError	Thrown if the SCMI box runs out of disk space trying to fulfill the request.	Same as generalError.

Error Response

<table>
<tr><th>XML</th><th>Comments</th></tr>
<tr><td><pre>&lt;?xml version=”1.0” encoding=”UTF-8”?>
<error-response version=”1”>
    <errorType>Internal Error</errorType>
    <description>
         This would be some text like: Out-of-disk-space
retrieving files. Or a Java exception trace
    </description>
</error-response&gt;</pre></td><td>The “errorType” must be one of the values specified in Krugle XML Schema ErrorType.

The “description” is arbitrary text provided by the SCMI.</td></tr>
</table>