Silk Central 21.1

API Help

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
http://www.microfocus.com

© Copyright 2004-2022 Micro Focus or one of its affiliates.

MICRO FOCUS, the Micro Focus logo and Silk Central are trademarks or registered trademarks of Micro Focus or one of its affiliates.

All other marks are the property of their respective owners.

2022-01-27

Contents

Silk Central API Help 4
Creating Plug-Ins 4
Code Coverage Integration 5
Creating Your Own Code Coverage Plugin 5
Installing the Code Analysis Framework on a Linux AUT Environment 10
Source Control Integration 11
Source Control Integration Interface 11
Source Control Integration Conventions 12
Issue Tracking Integration 12
Java Interface 12
Requirements Management Integration 13
Java Interfaces 13
Third-Party Test Type Integration 13
Plug-In Implementation 14
Structure of the API 15
Sample Code 15
Configuration XML File 18
Custom Icons 20
Deployment 20
Indicating Start and Finish for Video Capturing 20
Cloud Integration 21
Silk Central Web Services 22
Web Services Quick Start 22
Web-Service Authentication 25
Available Web Services 26
Services Exchange 27
Web Service Demo Client 53
Triggering Silk Central from a CI Server 53

Silk Central API Help

This guide provides the information you need to create and deploy plug-ins to integrate third-party tools into Silk Central, as well as information on managing results of external execution plan runs. For the available SOAP-based Web Services, this guide contains specifications and API descriptions and it explains how to integrate third-party plug-ins into Silk Central.

Note: This guide assumes you are familiar with the implementation and usage of Web Services.

Overview

Silk Central offers SOAP-based Web Services for the integration of third-party applications, as well as a REST API for managing results of external execution plan runs.

With the SOAP-based Silk Central Web Services, you can integrate your existing source-control, issue-tracking, and requirements-management tools by configuring Silk Central plug-ins. A number of sample plug-ins ship with Silk Central.

With the REST API for managing results of external execution plan runs, you can upload external results, which were generated by execution plan runs that were not executed by Silk Central execution servers, to Silk Central for further test management. You can also specify such external execution plans, which will be executed on an external execution environment instead of the Silk Central execution servers.

Documentation for the SOAP-based Web Services

Refer to the *Javadoc* for full details regarding available Java classes and methods. If the link does not work, click **Help** > **Documentation** > **Silk Central API Specification** in the Silk Central menu to open the Javadoc..

Documentation for the REST API

If Silk Central is installed on your system, you can access the interactive documentation of the REST API from *here*.

Silk Central Integration Plug-ins

The Silk Central plug-ins are provided "as is" and with all faults and without any warranty whatsoever. Micro Focus disclaims all warranties and conditions, whether express, implied, or statutory, as to any matter whatsoever, including but not limited to, any warranties, duties, or conditions or related to merchantability, fitness for a particular purpose, lack of viruses, accuracy or completeness, title, quiet enjoyment, quiet possession and noninfringement.

Your use of the plug-ins is at your own risk. In no event shall Micro Focus be liable for any direct, indirect, special, incidental or consequential damages of any kind, including, but not limited to, lost profits, incurred or arising out of your use of the plug-ins.

Creating Plug-Ins

Overview

This section describes how to create plug-ins for Silk Central. Only the tasks that are common to all plug-in types are discussed here.

Plug-In Species

Silk Central provides several plug-in APIs. Each API is considered a *species.*

Compilation

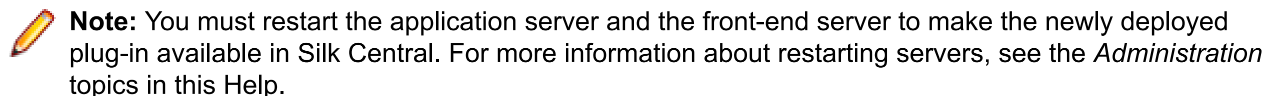
For developing and compiling plug-ins, see Silk Central Release Notes for the appropriate Java version. This is important for compatibility with the Silk Central Java runtime environment. Silk Central is using the AdoptOpenJDK.

Deployment

After you create your plug-in classes and implement a species API, you can create a plug-in package (a JAR or ZIP file).

- If your plug-in has no further dependencies (or depends on libraries that are already part of Silk Central), simply create a JAR file that contains your classes.
- If your plug-in depends on additional libraries, put the libraries into the subdirectory lib and pack all of the libraries together into a ZIP archive.

Place the created file into the plug-ins directory located at <application server installation directory>\plugins \.



Note: You must restart the application server and the front-end server to make the newly deployed plug-in available in Silk Central. For more information about restarting servers, see the *Administration* topics in this Help.

Distribution

As the plug-in species types are known by Silk Central, it is also known which servers (execution, application, and front-end) require which species. Each plug-in can be installed on the application server. Silk Central automatically distributes the correct plug-ins to each server.

Code Coverage Integration

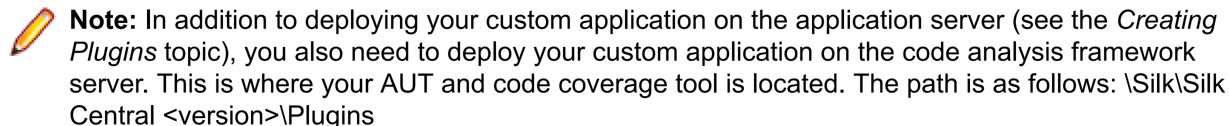
The Java API interface discussed in this chapter is required for creating plug-ins for Silk Central that enable the integration of a third-party (external) code coverage tool.

Code coverage tools enable you to deliver information about what code is covered by tests. Silk Central delivers the following code coverage tools out-of-the-box:

- Silk Central Java Code Analysis (Java Code Analysis Agent)
- DevPartner Studio .NET Code Analysis (Windows Code Analysis Framework)

Note: If your application under test runs on Linux, see the topic *Installing the Code Analysis Framework on a Linux AUT Environment*.

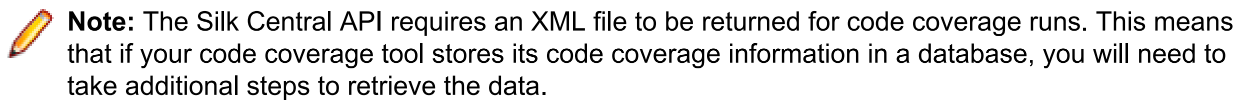
If the preceding two tools are not sufficient, you can create and deploy your own code coverage integration. See Creating Your Own Code Coverage Plugin.



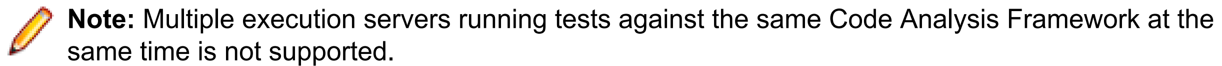
Note: In addition to deploying your custom application on the application server (see the *Creating Plugins* topic), you also need to deploy your custom application on the code analysis framework server. This is where your AUT and code coverage tool is located. The path is as follows: \Silk\Silk Central <version>\Plugins

Creating Your Own Code Coverage Plugin

This topic describes how to create a code coverage plugin. You should be familiar with the Silk Central baseline concept. In Silk Central, a baseline is required before each run. A baseline includes all namespaces/packages/classes/methods in the test application.



Note: The Silk Central API requires an XML file to be returned for code coverage runs. This means that if your code coverage tool stores its code coverage information in a database, you will need to take additional steps to retrieve the data.

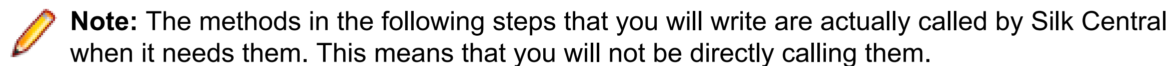


Note: Multiple execution servers running tests against the same Code Analysis Framework at the same time is not supported.

1. Add the library scc.jar to your classpath as it contains the interfaces that must be extended. The JAR file can be found in the lib directory of the Silk Central installation directory.
2. Add the following two import statements:

```
import com.segue.scc.published.api.codeanalysis.CodeAnalysisProfile;
import com.segue.scc.published.api.codeanalysis.CodeAnalysisProfileException;
import com.segue.scc.published.api.codeanalysis.CodeAnalysisResult;
```

3. Create a class that implements CodeAnalysisProfile.
4. Add all the required methods from the code coverage interface which are listed in the following steps. You can either refer to the Sample Interface Class for its definition and manually implement the methods or you can copy and paste the topic *Sample Profile Class* which has the imports and method definitions in it for you.

   Note: The methods in the following steps that you will write are actually called by Silk Central when it needs them. This means that you will not be directly calling them.
5. Code getBaseline. This method should return an XML file that contains all the namespaces/packages/classes/methods in the application. See the *Sample XML Data* topic file for the format of the file. You should validate the XML using the sample XSD file. See the *Code Coverage XSD* topic for the XSD.

   This function is called before starting coverage and is triggered by the Silk Central execution server starting a test run to start code analysis and return all of the objects to be covered. The output needs to be converted into XML using the format specified in the XML schema included in the CA-Framework installation folder.
6. Code startCoverage. This call should tell your code coverage tool to start collecting data. Return true if it starts.

   This is called by the Silk Central Code Coverage Framework after the getBaseLine() method is complete. This is where you should start your code coverage tool collecting code coverage data.
7. Code stopCoverage. This call should tell your code coverage tool to stop collecting data. Return true for success.

   This is called after startCoverage and is triggered by the Silk Central execution server finishing a test run to stop code analysis.
8. Code getCoverage. This returns an XML file with collected data from between the startCoverage and stopCoverage methods. See the *Sample XML Data* topic for the format of the file. You should validate the XML using the sample XSD file. See the *Code Coverage XSD* topic for the XSD.

   This function is called after stopCoverage() and returns all the coverage data collected. The output needs to be converted into XML using the specified in the XML schema.
9. Code GetName. This should provide the name that will be used to refer to the code coverage tool. For example, this value will be used as one of the values in the **Code Analysis Profile** list on the **Edit Code Analysis Settings** dialog box.

   This is called first by the Silk Central Code Coverage framework. The name of the plug-in is displayed in the code coverage list in Silk Central.
10. Build the plug-in into a jar and put the jar into a zip file.
11. Deploy your plug-in to the following locations:

    - In the Plugins directory of Silk Central's installation folder.
    - In the Plugins directory of the CA-Framework installation.

Sample Profile Class

This sample file outlines all the required methods, imports, and implements for a code coverage plugin.

```
//Add the library scc.jar to your classpath as it contains the interfaces that
//must be extended. The JAR file can be found in the lib directory of the Test
//Manager installation directory.
//
//make sure to include these imports after adding the scc.jar external reference
import com.segue.scc.published.api.codeanalysis.CodeAnalysisProfile;
import com.segue.scc.published.api.codeanalysis.CodeAnalysisProfileException;
import com.segue.scc.published.api.codeanalysis.CodeAnalysisResult;

public class myProfileClass implements CodeAnalysisProfile{

  // This function is called first by the Silk Central Code Coverage framework
  // The name of the plug-in is displayed in the code coverage drop down in Silk Central
  @Override
  public String getName() {
    // The name of the plugin cannot be an empty string
    return "my plugin name";
  }

  // This function is called before starting coverage,
  // this should return all of the objects to be covered and needs to be
  // converted into xml using the format specified in the XML schema
  // CodeCoverage.xsd included in the CA-Framework installation folder.
  // This is triggered by the Silk Central Execution Server starting a test run
  // to start code analysis.
  @Override
  public CodeAnalysisResult getBaseline() throws CodeAnalysisProfileException {
    CodeAnalysisResult result = new CodeAnalysisResult();
    try{
      String baselineData = MyCodeCoverageTool.getAllCoveredObjectsData();
      String xmlString = xmltransformXML(baselineData);
      result.Xml(xmlString);
      String myCustomLogMessage = "Code Coverage baseline successfully retrieved.";
      result.AddLogMsg(myCustomLogMessage);
    }catch(Exception e){
      throw new CodeAnalysisProfileException(e);
    }
    return result;
  }

  //This function is called by the Silk Central Code Coverage Framework after the getBaseLine() method is
complete
  //this is where you should start my code coverage tool
  //collecting code coverage data


  @Override
  public boolean startCoverage() throws CodeAnalysisProfileException {
    try{
      MyCodeCoverageTool.StartCollectingCoverageData();
    }catch(Exception e){
      throw new CodeAnalysisProfileException(e);
    }
  }
  //This function is called after startCoverage,
  //This is triggered by the Silk Central Execution Server finishing a test run
  //to stop code analysis
  //Call to my code coverage tool to stop collecting data here.
  @Override
```

```
 public boolean stopCoverage() throws CodeAnalysisProfileException {
   try{
     MyCodeCoverageTool.StopCollectingCoverageData();
   }catch(Exception e){
     throw new CodeAnalysisProfileException(e);
   }
 }
 // This function is called after stopCoverage(),
 // and should return all the coverage data collected and needs to be
 // converted into xml using the format specified in the XML schema
 // CCoverage.xsd included in the CA-Framework installation folder

 @Override
 public CodeAnalysisResult getCoverage() throws CodeAnalysisProfileException {
   CodeAnalysisResult result = new CodeAnalysisResult();
   try{
     String coverageData = MyCodeCoverageTool.getActualCoverageData();
     String xmlString = xmltransformXML(coverageData);
     result.Xml(xmlString);
     String myCustomLogMessage = "Code Coverage successfully retrieved.";
     result.AddLogMsg(myCustomLogMessage);
   }catch(Exception e){
    throw new CodeAnalysisProfileException(e);
   }

   return result;
 }

 private String transformXML(String myData){
   //code to transform from my data to the Silk CentralM needed xml
   ...
   return xmlString;
 }

}
```

Code Coverage XSD

The following is the code coverage XSD which you should use to validate the XML generated by your code coverage tool. This document is available in the following location: <CA Framework installation> \CodeAnalysis\CodeCoverage.xsd.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="data" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="coverage">
   <xs:complexType>
     <!--type will be method when defined as a child to class or line when defined as a child to method-->
     <xs:attribute name="type" type="xs:string" />
     <!--hits for the definition file will be 0, the update file will define the hits count-->
     <xs:attribute name="hits" type="xs:string" />
     <!--the total count will be sent with both the definition and update file, both counts will match-->
     <xs:attribute name="total" type="xs:string" />
     <!--this will be an empty string for the definition file, the line numbers will be sent in the update file
delimited by a colon-->
     <xs:attribute name="lines" type="xs:string" />
   </xs:complexType>
 </xs:element>
 <xs:element name="data">
   <xs:complexType>
     <xs:choice minOccurs="0" maxOccurs="unbounded">
       <xs:element ref="coverage" />
       <xs:element name="class">
         <xs:complexType>
           <xs:sequence>
```

```
        <xs:element name="sourcefile" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <!--full path to the code file-->
            <xs:attribute name="name" type="xs:string" />
          </xs:complexType>
        </xs:element>
        <xs:element ref="coverage" minOccurs="0" maxOccurs="unbounded" />
        <xs:element name="method" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element ref="coverage" minOccurs="0" maxOccurs="unbounded" />
            </xs:sequence>
            <!--
                <field_signature> ::= <field_type>
                <field_type>      ::= <base_type>|<object_type>|<array_type>
                <base_type>       ::= B|C|D|F|I|J|S|Z
                <object_type>     ::= L<fullclassname>;
                <array_type>      ::= [<field_type>

                The meaning of the base types is as follows:
                B byte signed byte
                C char character
                D double double precision IEEE float
                F float single precision IEEE float
                I int integer
                J long long integer
                L<fullclassname>; ... an object of the given class
                S short signed short
                Z boolean true or false
                [<field sig> ... array

                example signature for a java method 'doctypeDecl' with 3 string params and a return type
void

                    doctypeDecl : (Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)V

                refer to org.apache.bcel.classfile.Utility for more information on signatureToString
            -->
            <xs:attribute name="name" type="xs:string" />
            <!--method invokation count, this will be 0 for the definition file-->
            <xs:attribute name="inv" type="xs:string" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" />
    </xs:complexType>
  </xs:element>
  </xs:choice>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Sample XML Data

The code coverage API expects XML in the following format.

You can use the sample XSD document provided to validate your XML as well.

```
<?xml version="1.0" encoding="UTF-8"?><!--  Generated by 'MyPluginTool'at '2010-11-05T16:11:09'
 -->
<data>
  <class name="ProjectA.ClassA1">
    <sourcefile name="C:\Users\TestApp\ProjectA\ClassA1.cs"/>
    <coverage hits="8" total="8" type="method"/>
```

```
    <coverage hits="30" total="30" type="line"/>
    <method inv="2" name="ClassA1  : ()V">
      <coverage hits="3" lines="11:2,12:2,14:2" total="3" type="line"/>
    </method>
    <method inv="2" name="BoolByteMethod  : (LSystem/Boolean;LSystem/SByte;)V">
      <coverage hits="3" lines="17:2,18:2,19:2" total="3" type="line"/>
    </method>
    <method inv="1" name="CharStringMethod  : (LSystem/Char;LSystem/String;)V">
      <coverage hits="3" lines="38:1,39:1,40:1" total="3" type="line"/>
    </method>
    <method inv="2" name="DateMethod  : (LSystem/DateTime;)V">
      <coverage hits="3" lines="22:2,23:2,24:2" total="3" type="line"/>
    </method>
    <method inv="1" name="DecimalMethod  : (LSystem/Decimal;LSystem/Single;LSystem/Double;)V">
      <coverage hits="4" lines="27:1,28:1,29:1,30:1" total="4" type="line"/>
    </method>
    <method inv="1" name="IntMethod  : (LSystem/Int32;LSystem/Int64;LSystem/Int16;)V">
      <coverage hits="3" lines="33:1,34:1,35:1" total="3" type="line"/>
    </method>
    <method inv="1" name="passMeArrays  : (LSystem/Int32[];LSystem/Decimal[];)V">
      <coverage hits="6" lines="51:1,52:1,53:1,55:1,56:1,58:1" total="6" type="line"/>
    </method>
    <method inv="1" name="passMeObjects  : (LSystem/Object;LSystem/Object/ClassA1;)V">
      <coverage hits="5" lines="43:1,44:1,45:1,46:1,48:1" total="5" type="line"/>
    </method>
  </class>
  <class name="TestApp.Form1">
    <sourcefile name="C:\Users\TestApp\Form1.Designer.cs"/>
    <coverage hits="2" total="10" type="method"/>
    <coverage hits="24" total="110" type="line"/>
    <method inv="1" name="btnClassA_Click  : (LSystem/Object;LSystem/Object/EventArgs;)V">
      <coverage hits="3" lines="25:1,26:1,27:1" total="3" type="line"/>
    </method>
    <method inv="1" name="CallAllClassAMethods  : ()V">
      <coverage hits="21"
lines="35:1,36:1,37:1,38:1,39:1,40:1,41:1,42:1,43:1,44:1,45:1,46:1,48:1,49:1,50:1,51:1,52:1,53:1,54:1,55:1
,56:1" total="21" type="line"/>
    </method>
  </class>
</data>
```

Installing the Code Analysis Framework on a Linux AUT Environment

The following steps should be performed if the code coverage plugin that you create will interact with your .NET application under test on the Linux operating system.

1. The code analysis framework for Linux is available from **Help** > **Tools** > **Linux Code Analysis Framework**. Download and copy this over to root or any other folder on the Linux machine.
2. Ensure that the latest version of Java Runtime Environment 1.8 is installed on the application under test machine.
3. Extract CA-Framework.tar.gz.
4. Place the code analysis plug-in into the <install dir>/Silk Central 21.1/Plugins folder.
5. Change directory to <install dir>/Silk Central 21.1/Code Analysis.
6. Find startCodeAnalysisFramework.sh shell script which will run the CA-Framework process.
7. Run the following command to convert the file to Unix format: dos2unix startCodeAnalysisFramework.sh.
8. Run the following command to set required permissions to execute the shell script: chmod 775 startCodeAnalysisFramework.sh.

9. Run the following shell script to run the CAFramework process: ./startCodeAnalysisFramework.sh.

The code analysis framework is ready to be used from Silk Central.

Source Control Integration

Source control profiles enable Silk Central to integrate with external source control systems.

Once deployed, custom source control plug-ins can be configured in Silk Central, allowing you to define where Silk Central execution servers should retrieve program sources for test execution.

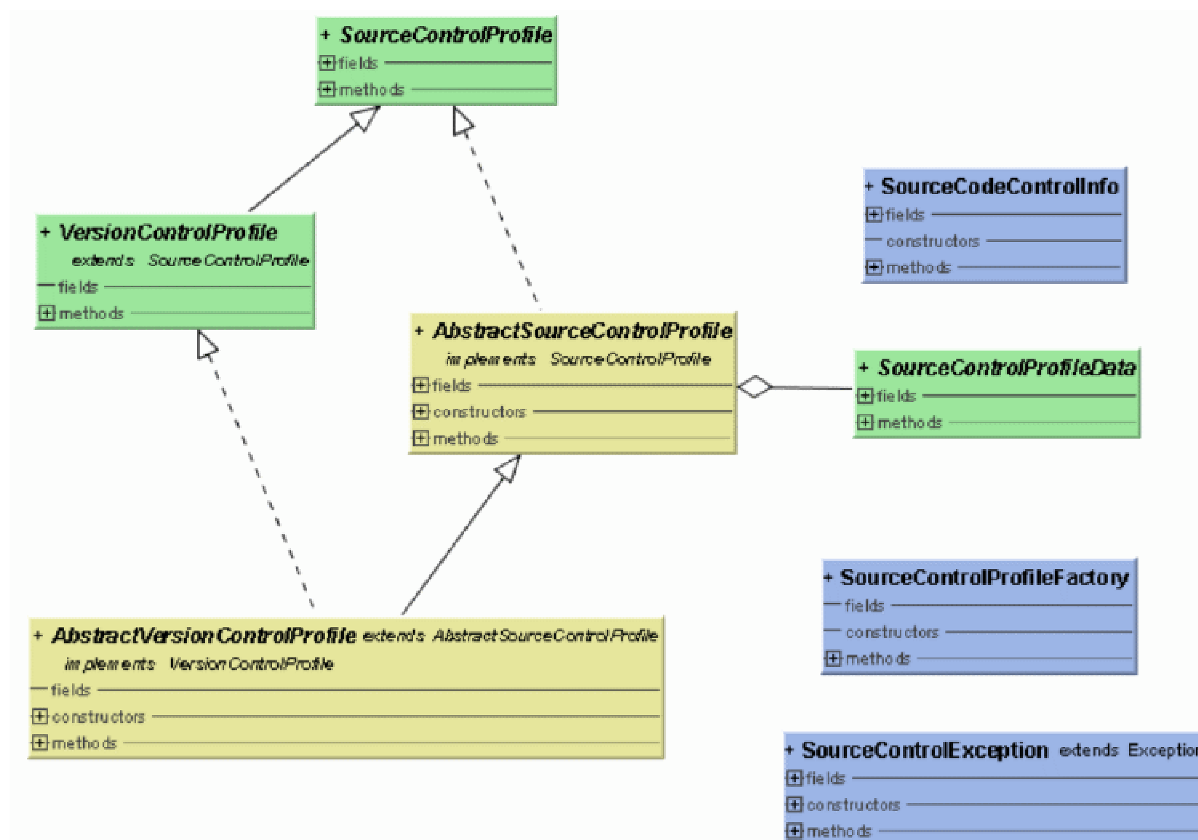
See the sources of the package com.segue.scc.vcs.subversion at C:\Program Files (x86)\Silk\Silk Central 21.1\instance_<instance number>_<instance name>\Plugins\subversion.zip to see how these elements fit together.

Source Control Integration Interface

Silk Central distinguishes between SourceControlProfile and VersionControlProfile. The difference is that SourceControlProfile is non-versioned and VersionControlProfile is versioned.

Following are the Silk Central interfaces that are used for source control integration:

- SourceControlProfile
- VersionControlProfile
- SourceControlProfileData
- SourceControlException
- SourceControlInfo



Refer to the *Javadoc* for full details regarding available Java classes and methods. If the link does not work, click **Help** > **Documentation** > **Silk Central API Specification** in the Silk Central menu to open the Javadoc.

Source Control Integration Conventions

Each implementation must provide a default constructor and, optionally, a constructor with a SourceControlProfileData as a parameter. If this constructor is not provided, a bean setter for SourceControlProfileData must be provided.

As each interface method specifies the SourceControlException that is to be thrown, it is not allowed to throw a RuntimeException in any method used by the interface.

Issue Tracking Integration

The interface discussed in this chapter is required for creating plug-ins for Silk Central that enable the integration of a third-party (external) Issue Tracking System (ITS).

Defining issue tracking profiles allows you to link tests within the **Tests** area to issues in third-party issue tracking systems. Linked issue states are updated periodically from third-party issue tracking systems.

See the sources of the package com.segue.scc.issuetracking.bugzilla4 at C:\Program Files (x86)\Silk\Silk Central 21.1\instance_<instance number>_<instance name>\Plugins\IT-Bugzilla4.zip to see how these elements fit together.

Java Interface

Refer to the *Javadoc* for full details regarding available Java classes and methods. If the link does not work, click **Help** > **Documentation** > **Silk Central API Specification** in the Silk Central menu to open the Javadoc.

Build environment

Add the library scc.jar to your classpath, as it contains the interfaces that must be extended. The JAR file can be found in the lib directory of the Silk Central installation directory.

You must extend two interfaces/classes:

- com.segue.scc.published.api.issuetracking82.IssueTrackingProfile
- com.segue.scc.published.api.issuetracking.Issue

Classes/Interfaces

- IssueTrackingProfile
- IssueTrackingData
- Issue
- IssueTrackingField
- IssueTrackingProfileException

Requirements Management Integration

Silk Central can be integrated with third-party Requirements Management System (RMS) tools to link and synchronize requirements.

This section describes a Java application programming interface that enables you to implement third-party plug-ins for synchronization of requirements in Silk Central with the requirements of a third-party Requirements Management System. This section describes the interfaces that identify a requirement plug-in and its deployment.

A JAR or ZIP file is provided following the standard Silk Central plug-in concept, which is automatically deployed to all front-end servers, thereby enabling access to third-party tools for configuration and synchronization. This plug-in implements a specified interface that allows Silk Central to identify it as a requirements plug-in and provides the required login properties to log in to the third-party tool.

Refer to the *Javadoc* for full details regarding available Java classes and methods. If the link does not work, click **Help** > **Documentation** > **Silk Central API Specification** in the Silk Central menu to open the Javadoc.

For information about additional plug-ins, contact customer support.

Java Interfaces

Refer to the *Javadoc* for full details regarding available Java classes and methods. If the link does not work, click **Help** > **Documentation** > **Silk Central API Specification** in the Silk Central menu to open the Javadoc.

The basic interface to start with is RMPluginProfile (com.segue.tm.published.api.requirements.javaplugin). RMPluginProfile specifies the plug-in as a requirements plug-in.

The requirements Java plug-in API includes the following additional interfaces:

- RMAction
- RMAttachment
- RMDataProvider
- *Optional:* RMIconProvider
- RMNode
- RMNodeType
- RMPluginProfile
- RMProject
- RMTest
- RMTestParameter
- *Optional:* RMTestProvider
- RMTestStep

Third-Party Test Type Integration

Silk Central enables you to create custom plug-ins for test types beyond the standard set of available test types, which includes Silk Performer, Silk Test Classic, manual tests, NUnit, JUnit, and Windows Scripting Host (WSH). Upon creating a new test-type plug-in, your custom test type becomes available in the **Type**

list box on the **New Test** dialog box, alongside the standard test types that are available for creating new tests in Silk Central.

A plug-in specifies which properties are required for configuring a test and implementing execution of a test. Meta information about properties is defined through a *Configuration XML File*.

The goal of the plug-in approach is to support tests based on common testing frameworks such as JUnit and NUnit, or scripting languages (WSH), for easy customization of Silk Central to your specific testing environment. The well-defined Silk Central public API allows you to implement a proprietary solution that meets your automated test needs. Silk Central is open and extensible to any third-party tool that can be invoked from a Java implementation or via command line call.

Refer to the *Javadoc* for full details regarding available Java classes and methods. If the link does not work, click **Help** > **Documentation** > **Silk Central API Specification** in the Silk Central menu to open the Javadoc.

The classes that are described in the Javadoc are included in the file tm-testlaunchapi.jar.

For information about additional plug-ins, contact customer support.

This section includes a code sample that implements the process executor test type. The process executor can be used to launch any executable and extends the published process test launcher class. For additional information, download the *Test Launch Plug-In Sample* from **Help** > **Tools** and read the Readme.txt file.

Plug-In Implementation

The principles of the API are based on widely known Java Beans concepts. This allows developers to easily implement test launch plug-ins. To avoid putting textual information in Java code, meta information about properties is defined in an XML file.

The plug-in implementation is packaged in a zip archive and implements a callback interface so that it can be integrated. Other interfaces are in turn provided by the plug-in framework and allow the implementation to access information or return results.

Packaging

The plug-in is packaged in a zip archive that contains the Java codebase and the XML configuration file. The archive also contains some test launch plug-in implementations. The codebase may be contained in a Java archive file (.jar) or directly in .class files within folders representing the Java package structure.

The TestLaunchBean plug-in class follows the Bean standard and implements the TestLaunchBean interface. The XML configuration file in the .zip archive has the same name as this class. This allows you to package multiple plug-ins and XML files in a single archive.

Passing Parameters to the Plug-In

If a plugin is based on the ExtProcessTestLaunchBean class, each parameter will automatically be set as an environment variable in the process started by the plugin. This is also the case if the parameter name matches the name of a system variable, so that the value of the system variable will be replaced by the parameter value, except when the parameter value is an empty string.

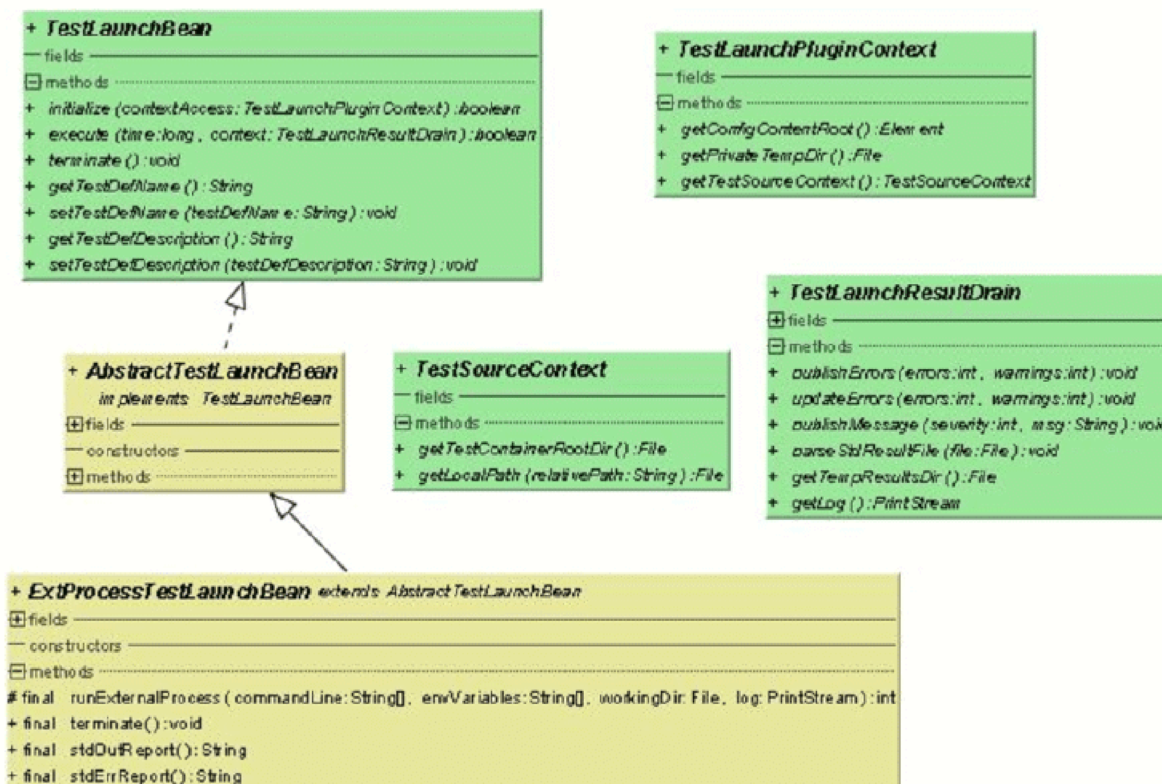
The plug-in interface provides full access to all custom parameters that have been defined in the Silk Central **Tests** area . Only custom parameters are supported for third-party test types. The plug-in cannot specify predefined parameters; the plug-in implementation determines if and how parameters are defined for specific tests.

Use the getParameterValueStrings() method in the TestLaunchPluginContext interface to obtain a container with mappings from parameter names (key) to their values represented as String.

For JUnit test types, any JUnit test class can access a custom parameter of the underlying test as a Java system property; the launcher passes these parameters to the executing virtual machine using the "-D" VM argument.

Structure of the API

This image details the structure of the API for third-party test type integration.



Refer to the *Javadoc* for full details regarding available Java classes and methods. If the link does not work, click **Help** > **Documentation** > **Silk Central API Specification** in the Silk Central menu to open the Javadoc.

Available Interfaces

- TestLaunchBean
- ExtProcessTestLaunchBean
- TestLaunchPluginContext
- TestSourceContext
- TestLaunchResultDrain

Sample Code

This sample code block implements the process executor test type, which can be used to launch any executable and extends the published ExtProcessTestLaunchBean class.

```
package com.borland.sctm.testlauncher;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
```

```
import com.segue.tm.published.api.testlaunch.ExtProcessTestLaunchBean;
import com.segue.tm.published.api.testlaunch.TestLaunchResultDrain;

/**
 * Implements an Silk Central test type that can be used to launch
 * any executables, for example from a command line.
 * Extends Silk Central published process test launcher class,
 * see Silk Central API Specification (in Help -> Documentation) for
 * further details.
 */
public class ProcessExecutor extends ExtProcessTestLaunchBean {

  // test properties that will be set by Silk Central using appropriate setter
  // methods (bean convention),
  // property names must conform to property tags used in the XML file

  /**
   * Represents property <command> defined in ProcessExecutor.xml.
   */
  private String command;

  /**
   * Represents property <arguments> defined in ProcessExecutor.xml.
   */
  private String arguments;

  /**
   * Represents property <workingfolder> defined in ProcessExecutor.xml.
   */
  private String workingfolder;

  /**
   * Java Bean compliant setter used by Silk Central to forward the command to be
   * executed.
   * Conforms to property <command> defined in ProcessExecutor.xml.
   */
  public void setCommand(String command) {
    this.command = command;
  }

  /**
   * Java Bean compliant setter used by Silk Central to forward the arguments
   * that will be passed to the command.
   * Conforms to property <arguments> defined in ProcessExecutor.xml.
   */
  public void setArguments(String arguments) {
    this.arguments = arguments;
  }

  /**
   * Java Bean compliant setter used by Silk Central to forward the working
   * folder where the command will be executed.
   * Conforms to property <workingfolder> defined in
   * ProcessExecutor.xml.
   */
  public void setWorkingfolder(String workingfolder) {
    this.workingfolder = workingfolder;
  }

  /**
   * Main plug-in method. See Silk Central API Specification
   * (in Help &gt; Documentation) for further details.
   */
```

```
@Override
public boolean execute(long time, TestLaunchResultDrain context)
  throws InterruptedException {
  try {
    String[] cmd = getCommandArgs(context);
    File workingDir = getWorkingFolderFile(context);
    String[] envVariables = getEnviromentVariables(context);

    int processExitCode = runExternalProcess(cmd, envVariables, workingDir,
      context.getLog());

    boolean outputXmlFound = handleOutputXmlIfExists(context);

    if (! outputXmlFound && processExitCode != 0) {
      // if no output.xml file was produced, the exit code indicates
      // success or failure
      context.publishMessage(TestLaunchResultDrain.SEVERITY_ERROR,
        "Process exited with return code "
        + String.valueOf(processExitCode));
      context.updateErrors(1, 0);
      // set error, test will get status 'failed'
    }
  } catch (IOException e) {
    // prints exception message to Messages tab in Test Run
    // Results
    context.publishMessage(TestLaunchResultDrain.SEVERITY_FATAL,
      e.getMessage());
    // prints exception stack trace to 'log.txt' that can be viewed in Files
    // tab
    e.printStackTrace(context.getLog());
    context.publishErrors(1, 0);
    return false; // set test status to 'not executed'
  }
  return true;
}

/**
 * Initializes environment variables to be set additionally to those
 * inherited from the system environment of the Execution Server.
 * @param context the test execution context
 * @return String array containing the set environment variables
 * @throws IOException
 */
private String[] getEnviromentVariables(TestLaunchResultDrain context)
  throws IOException {
  String[] envVariables = {
    "SCTM_EXEC_RESULTSFOLDER="
    + context.getTempResultsDir().getAbsolutePath(),
    "SCTM_EXEC_SOURCESFOLDER="
    + sourceAccess().getTestContainerRootDir().getAbsolutePath(),
  };
  return envVariables;
}

/**
 * Let Silk Central parse the standard report xml file (output.xml) if exists.
 * See also Silk Central Web Help - Creating a Test Package. A XSD file
 * can be found in Silk Central Help -&gt; Tools -&gt; Test Package XML Schema
 * Definition File
 * @param context the test execution context
 * @return true if output.xml exists
 * @throws IOException
 */
private boolean handleOutputXmlIfExists(TestLaunchResultDrain context)
```

```
    throws IOException {
    String outputFileName = context.getTempResultsDir().getAbsolutePath()
    + File.separator + TestLaunchResultDrain.OUTPUT_XML_RESULT_FILE;
    File outputfile = new File(outputFileName);
    boolean outputXmlExists = outputfile.exists();
    if (outputXmlExists) {
      context.parseStdResultFile(outputfile);
      context.publishMessage(TestLaunchResultDrain.SEVERITY_INFO,
        String.format("output.xml parsed from '%s'", outputFileName));
    }
    return outputXmlExists;
  }

  /**
   * Retrieves the working folder on the Execution Server. If not configured
   * the results directory is used as working folder.
   * @param context the test execution context
   * @return the working folder file object
   * @throws IOException
   */
  private File getWorkingFolderFile(TestLaunchResultDrain context)
    throws IOException {
    final File workingFolderFile;
    if (workingfolder != null) {
      workingFolderFile = new File(workingfolder);
    } else {
      workingFolderFile = context.getTempResultsDir();
    }
    context.publishMessage(TestLaunchResultDrain.SEVERITY_INFO,
      String.format("process is exectued in working folder '%s'",
        workingFolderFile.getAbsolutePath()));
    return workingFolderFile;
  }

  /**
   * Retrieves the command line arguments specified.
   * @param context the test execution context
   * @return an array of command line arguments
   */
  private String[] getCommandArgs(TestLaunchResultDrain context) {
    final ArrayList<String> cmdList = new ArrayList<String>();
    final StringBuilder cmd = new StringBuilder();
    cmdList.add(command);
    cmd.append(command);
    if (arguments != null) {
      String[] lines = arguments.split("[\\r\\n]+");
      for (String line : lines) {
        cmdList.add(line);
        cmd.append(" ").append(line);
      }
    }
    context.publishMessage(TestLaunchResultDrain.SEVERITY_INFO,
      String.format("executed command '%s'", cmd.toString()));
    context.getLog().printf("start '%s'%n", cmd.toString());
    return (String[]) cmdList.toArray(new String[cmdList.size()]);
  }
}
```

Configuration XML File

The configuration XML file contains meta information about the third-party test type plug-in.

Plug-In Meta Information

Plug-in meta information generally provides information about the plug-in and test type. Available information types are listed below.

Type	Description
id	An internal string that is unique among all installed plug-ins identifying this type.
label	The GUI text displayed in the choice list and edit dialogs for this type.
description	Additional textual information describing this test type.
version	Distinguishes different versions of a type.

General Property Meta Information

For editable properties, there is general information that is the same for each property type in addition to the type-specific information. The name of a property must match those defined in the code by get<<propertyname>> methods, with the first character in lowercase.

Type	Description
label	The text displayed in the GUI.
description	Additional textual information describing the property.
isOptional	A value of true indicates that user input is not required.
default	The default value for the property when creating a new test. This value must match the property type.

String Property Meta Information

Here are the types of string-property meta information that are available in the plug-in.

Type	Description
maxLength	Denotes the maximum length the user can enter.
editMultiLine	Denotes if the edit field should have multiple rows.
isPassword	A value of true hides the user input as ***.

File Property Meta Information

Here are the types of file-property meta information that are available in the plug-in.

For specified file values, non version-controlled files with absolute paths on the execution server are distinguished from version-controlled files. Files under source control are always relative to the root directory of the test container and are typically used to specify a test source to be executed. Absolute paths need to exist on the execution server, typically specifying a tool or resource used to invoke the test on the execution server.

Type	Description
openBrowser	A value of true indicates that a browser should be opened to select the file from the files in a test container.

Type	Description
isSourceControl	A value of true indicates that the file originates from a source control system.
fileNameSpec	Denotes a restriction for allowed file names as known from the standard Windows file browse dialog.

Custom Icons

You can design custom icons for your test type to make the test type identifiable. To define these icons for the plug-in, for which you have defined the identifier PluginId in the configuration XML file, you have to place the following four icons into the root directory of the plug-in ZIP container.

Name	Description
<PluginId>.gif	The default icon for your tests type. For example, ProcessExecutor.gif.
<PluginId>_package.gif	The icon used for the test package root and the suite nodes, in case you convert a test of the specified test type into a test package. For example, ProcessExecutor_package.gif.
<PluginId>_linked.gif	The icon used if the parent folder of the test is a linked test container. For example, ProcessExecutor_linked.gif.
<PluginId>_incomplete.gif	The icon used if the product or the source control profile of the parent test container of the test is not defined.

When you design a new icon for a test type, apply the following rules:

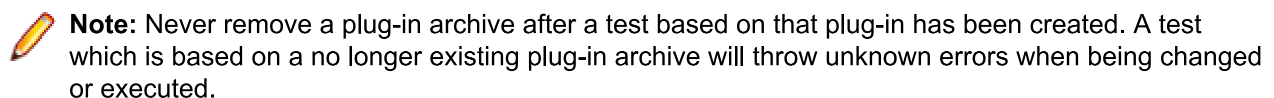
- Use only icons of type GIF. The file extension is case sensitive and must always be lowercase (.gif).
- Remove old or invalid icons from <Silk Central deploy folder>\wwwroot\silkroot\img\PluginIcons, because otherwise the icons will not be updated with the new icons in the root directory of the plug-in ZIP container.
- The size of the icon is 16x16 pixels.
- The maximal number of colors allowed for an icon is 256.
- The icon includes a 1 bit transparency.

Deployment

The plug-in ZIP archive must reside in the Plugins subdirectory within the Silk Central installation directory. To integrate plug-ins residing in this directory, restart your application server and front-end server through Silk Central Service Manager.

Note: Hot deployment is not supported.

Each time an archive is modified, these two servers must be restarted. The archive is uploaded automatically to execution servers.



Note: Never remove a plug-in archive after a test based on that plug-in has been created. A test which is based on a no longer existing plug-in archive will throw unknown errors when being changed or executed.

Indicating Start and Finish for Video Capturing

Refer to the *Javadoc* for full details regarding available Java classes and methods. If the link does not work, click **Help** > **Documentation** > **Silk Central API Specification** in the Silk Central menu to open the Javadoc.

When you create a new third-party test plug-in for Silk Central, with a third-party test type that supports processing multiple test cases in a single test execution, and you want to associate captured videos to specific test cases, you can proceed in two ways.

Third-Party Tests that are Running in the Plug-In

For these tests, we recommend you to use the indicateTestStart and indicateTestStop methods of the TestLaunchResultDrain class.

Third-Party Tests that are Running in an External Process

For these tests, you can use a TCP/IP-based service to send START and FINISH messages to the port of the Silk Central execution server. The port number to be used can be queried from *ExecutionContextInfo.ExecProperty#PORT_TESTCASE_START_FINISH* in the plugin. The port is also available as an environment variable called *#sctm_portTestCaseStartFinish* in the test process, if the plugin extends ExtProcessTestLaunchBean. These message types inform the execution server that a test case in the test has started or respectively finished. The messages must be encoded in Unicode (UTF8) or ASCII format.

Message Type	Format
START	START <Test Name>, <Test ID> <LF>, where LF has ASCII code 10.
FINISH	FINISH <Test Name>, <Test ID>, <Passed> LF, where LF has ASCII code 10. Passed can be True or False. If video capturing is set to be performed On Error, the video is only saved to the results if Passed is set to False.

The execution server responds with the message OK, if the request was recognized, or otherwise the execution server responds with an error message. Always wait for the response of the execution server before you execute the next test case, because otherwise the recorded video might not match the actual test case.

If the external process, where the test is executed, is based on a Java environment, we recommend you to use the indicateTestStart and indicateTestStop methods of the TestCaseStartFinishSocketClient class, which is included in the file tm-testlaunchapi.jar.

Cloud Integration

Silk Central enables you to integrate with providers of public or private cloud services by configuring cloud provider profiles. Cloud profiles are based on a plug-in concept, which enables you to write your own plug-in for a specific cloud provider. A cloud provider plug-in deploys a virtual environment before each automated test run.

Note: The cloud API is subject to change in upcoming versions of Silk Central. If you use this API, upgrading to a future version of Silk Central may require you to make updates to your implementation.

Refer to the *Javadoc* for full details regarding available Java classes and methods. If the link does not work, click **Help** > **Documentation** > **Silk Central API Specification** in the Silk Central menu to open the Javadoc.

The basic interface to start with is CloudProviderProfile (com.segue.scc.published.api.cloud). CloudProviderProfile specifies the access and control of an external cloud system. A cloud provider plug-in implementation needs to accomplish the following:

- Expose which properties have to be configured in a cloud provider profile to remotely access a provider of this type
- Validate profile properties and check the connection to the cloud provider
- Retrieve a list of available image templates from the cloud provider
- Deploy a virtual environment based on a selected image template and expose externally accessible host addresses

- Check if a virtual environment is deployed and running
- Delete a specific virtual environment

Silk Central Web Services

The SOAP-based Web Services for Silk Central do not require setup, they are enabled by default on each front-end server. For example, if http://www.yourFrontend.com/login is the URL that you use to access Silk Central, then http://www.yourFrontend.com/Services1.0/jaxws/system (legacy services at http://www.yourFrontend.com/Services1.0/services) and http://www.yourFrontend.com/AlmServices1.0/services are the base URLs you use to access available Web Services.

When you access the base URL using your browser, you are presented with a simple HTML list of all available Web Services. This list is provided by JAX-WS. The SOAP stack that Silk Central uses is *https://jax-ws.java.net/*.

The base URL provides links to WSDL (Web Service Definition Language) standardized XML files, where each file describes the interface of a single Web Service. These files are not human readable. For this reason, SOAP-enabled clients (for example, Silk Performer Java Explorer) read WSDL files and thereby retrieve information required for invoking methods on corresponding Web Services.

Refer to the *Javadoc* for full details regarding available Java classes and methods. If the link does not work, click **Help** > **Documentation** > **Silk Central API Specification** in the Silk Central menu to open the Javadoc.

For information on how to manage execution plan runs, which are executed on external execution environments instead of on the Silk Central execution servers, and their results, Silk Central offers a REST API documentation. If Silk Central is installed on your system, you can access the interactive documentation of the REST API from *here*.

Web Services Quick Start

This section includes prerequisites, a use case example, and other topics related to Web Services integration.

Prerequisites

The following prerequisites must be considered before attempting to develop a Web Services client:

- Basic knowledge of Object Oriented Programming (OOP). Java experience would be helpful, since examples will be given in that language. A developer with no Java but C++, C#, Python, or Perl experience should still be able to easily follow the examples. Experience in manipulating collections such as HashMaps and Lists is desirable.
- JUnit tests are briefly mentioned. This Java testing framework is not required, but is helpful to know.
- Introductory exposure to Web Services technology. This help does not offer an introductory course in Web Services or SOAP. At a minimum you should have experience coding and successfully running a "Hello World!" Web Services client.
- Knowledge of the Silk Central Web Services architecture.

Getting Started with Web Services

Refer to the *Silk Central Release Notes* to ensure that you have the appropriate SDK version installed and in your PATH.

It is helpful to have the JUnit jar in your classpath. You can download the JUnit.jar file from *http://www.junit.org/index.htm*.

1. Ensure that the bin directory of the Java SDK is in your PATH.

2. Run the following command, pointing to the WSDL of the desired Web Service.

```
wsimport -s <path where to save the generated files> -Xnocompile
-p <package structure you want to use for your client yourWebService>
http://<URL to your service>/yourWebService?wsdl
```

3. Look for the automatically generated Java class YourWebService.

   This class is ready for you to use all of the methods that YourWebService provides.

Web Services Client Overview

Web services typically use a SOAP over HTTP protocol. In this scenario, SOAP envelopes are sent out. When collections and other complex objects are bundled in SOAP envelopes, the ASCII data structures can become difficult to read and edit. Novice developers should not try to build web services clients by direct manipulation of SOAP envelopes. Experienced developers typically do not build web services clients at the SOAP envelope level. Doing so is tedious and error prone. For this reason, all major programming languages provide web services development kits. In Silk Central, the *Java API for XML Web Services (JAX-WS)* is used for building web services and clients that communicate using SOAP messages.

Regardless of the implementation language (Java, C++, C#, Perl, or Python), building web service clients generally follows the same pattern:

1. Point a development kit tool to the web services WSDL.
2. Get a client stub returned.
3. Edit the client stub generated in step 2 to obtain a full-blown client.

JAX-WS follows this pattern. Here, the wsimport tool (bundled with the JDK) is used to build the client stubs from the WSDL. Detailed information on how to use wsimport can be found in the *JDK Tools and Utilities documentation*. A brief description of the switches used in the above summary is as follows:

- -s: The output directory of the client stubs
- -p: The target package. Deploy to this package structure

Example: wsimport -s <location of generated stubs> -p <target package> <WSDL>

The wsimport tool generates several classes to support a client to the web service. If YourWebService is the name of the service, then one can expect the following classes to be output:

- YourWebService: An interface representing YourWebService.
- YourWebServiceService: A generated class representing the YourWebService Locator, for example to get a list of available ports (service endpoint interfaces)
- WSFaultException: An exception class mapped from wsdl:fault
- WsFaultBean: Asynchronous response bean derived from response wsdl:message
- Serializeable Objects: Client-side objects corresponding to the objects YourWebService uses.

To generate a JAX-WS client for accessing Silk Central web services, create a new a Java class called YourWebServiceClient. You have to bind to the web service through a port; a port is a local object that acts as a proxy for the remote service. Note that the port is created by executing the wsimport tool in the step above. To retrieve this proxy, invoke the getRequirementsServicePort method on the service:

```
// Bind to the Requirements service
 RequirementsServiceService port = new RequirementsServiceService
   (new URL("http", mHost, mPort, "/Services1.0/jaxws/requirements?wsdl"));
 RequirementsService requirementsService = port.getRequirementsServicePort();
```

To authenticate with the web services, generate a web-service token in the **User Settings** page of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**.

You can also invoke the service's logonUser method by passing username and password to obtain a session ID:

```
// Login to Silk Central and get session ID
 String sessionId = requirementsService.logonUser(mUsername, mPassword);
```

Example Use Case: Adding a Requirement

Following up on previous steps detailed in this section, this topic completes a use case for adding a requirement to Silk Central.

Before proceeding you must have satisfied the following pre-requisites:

- You have completed the steps detailed for the requirements web service.
- A working POJO or JUnit class with binding and login methods has been created.
- You have familiarized yourself with the other Silk Central API help topics.

1. Generate a web-service token in the **User Settings** page.
   a) Click on the user name in the Silk Central menu. The **User Settings** page opens.
   b) In the **Web-Service Token** section of the page, click **Generate Token**.
2. Construct a requirements object that contains the desired data.
3. Call the updateRequirement method by using the web-service token, project ID and the requirements object you have generated.
4. Save the requirement ID that is returned by the updateRequirement method.
5. Create a PropertyValue array of the requirement properties.
6. Call the updateProperties method using the previously created array.

The wsimport will create the above mentioned Web Service objects:

- Requirement
- PropertyValue

You can now use the OOP methods of the above objects to consume the Web Service. No SOAP envelope construction is required. Below are excerpts of the code that is required to complete this use case.

```
/** project ID of Silk Central project */
private static final int PROJECT_ID = 0;

/** propertyID for requirement risk */
public static final String PROPERTY_RISK = "Risk";

/** propertyID for requirement reviewed */
public static final String PROPERTY_REVIEWED = "Reviewed";

/** propertyID for requirement priority */
public static final String PROPERTY_PRIORITY = "Priority";

/** propertyID for requirement obsolete property */
public static final String PROPERTY_OBSOLETE = "Obsolete";


// Get the Requirements service
RequirementsService service = getRequirementsService();

// The web-service token that you have generated in the UI. Required to authenticate when using
// a web service.
String webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";

// Construct Top Level Requirement
Requirement topLevelRequirement = new Requirement();
topLevelRequirement.setName("tmReqMgt TopLevelReq");
topLevelRequirement.setDescription("tmReqMgt TopLevel Desc");

PropertyValue propRisk = new PropertyValue();
propRisk.setPropertyId(PROPERTY_RISK);
propRisk.setValue("2");
PropertyValue propPriority = new PropertyValue();
```

```
propPriority.setPropertyId(PROPERTY_PRIORITY);
propPriority.setValue("3");
PropertyValue[] properties = new PropertyValue[] {propRisk, propPriority};

/*
* First add requirement skeleton, get its ID
* service is a binding stub, see above getRequirementsService()
*/
int requirementID = service.updateRequirement(webServiceToken, PROJECT_ID, topLevelRequirement,
-1);

// Now loop through and set properties
for (PropertyValue propValue : properties) {
propValue.setRequirementId(requirementID);
service.updateProperty(webServiceToken, requirementID, propValue);
}

// Add Child Requirement
Requirement childRequirement = new Requirement();
childRequirement.setName("tmReqMgt ChildReq");
childRequirement.setDescription("tmReqMgt ChildLevel Desc");
childRequirement.setParentId(requirementID);
propRisk = new PropertyValue();
propRisk.setPropertyId(PROPERTY_RISK);
propRisk.setValue("1");
propPriority = new PropertyValue();
propPriority.setPropertyId(PROPERTY_PRIORITY);
propPriority.setValue("1");
properties = new PropertyValue[] {propRisk, propPriority};

int childReqID = service.updateRequirement(webServiceToken, PROJECT_ID, childRequirement, -1);

// Now loop through and set properties
for (PropertyValue propValue : properties) {
propValue.setRequirementId(requirementID);
service.updateProperty(webServiceToken, childReqID, propValue);
}

// Print Results
System.out.println("Login Successful with web-service token: " + webServiceToken);
System.out.println("Top Level Requirement ID: " + requirementID);
System.out.println("Child Requirement ID: " + childReqID);
```

Note: This sample code is also available in the Web Service Demo Client's class com.microfocus.silkcentral.democlient.samples.AddingRequirement.

Web-Service Authentication

Silk Central data is protected against unauthorized access. Login credentials must be provided before data access is granted. This is true not only when working with the HTML front-end, but also for communication with Silk Central through SOAP or REST API calls.

The first step in querying data, or applying configuration changes for Silk Central, is authentication. When authentication is successful, a user session is created that allows execution of subsequent operations in the context of that user login.

When accessing Silk Central through a web browser, session information is not visible to the user. The browser handles session information with the use of cookies. In contrast to using Silk Central through HTML, SOAP calls must handle the session information manually.

Micro Focus recommends authentication through a web-service token. You can generate such a web-service token in the **User Settings** page of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**.

You can also use the logonUser SOAP call or the login REST API call for authentication. The method call returns a session identifier that references the session created on the server and at the same time is used as a key to access Silk Central in the context of that session.

Each subsequent API call that requires authentication takes such a web-service token or session identifier as one of its parameters, checks its validity, and executes in the context of the corresponding session.

A Silk Central session created through web services can not be ended explicitly. Instead, sessions end automatically when they are not used for a period of time. As soon as a session times out on a server, the subsequent SOAP calls that attempt to use the session throw exceptions.

A demonstration client is available for download in Silk Central at **Help** > **Tools** > **Web Services Demo Client**. This demo project utilizes the Silk Central tests web service, helping you to get familiar with the web service interface.

Examples

If you have generated a web-service token in Silk Central UI, the following Java code sample demonstrates access to Silk Central through web services and the use of the web-service token:

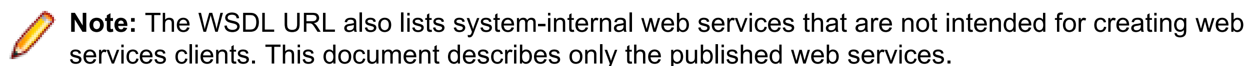
```
string webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";
Project[] projects = sccentities.getProjects(webServiceToken);
```

The following Java code sample demonstrates the same access to Silk Central through web services and the use of the session identifier:

```
long sessionID = systemService.logonUser("admin", "admin");
Project[] projects = sccentities.getProjects(sessionID);
```

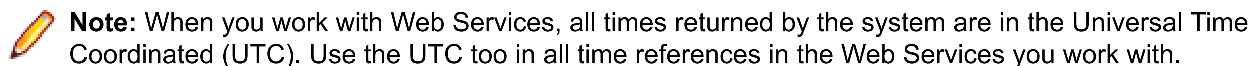
Available Web Services

The following table lists the available Silk Central web services. For information on accessing Silk Central through HTTP-based interfaces, see *Services Exchange*. For information on the REST API-based web services that enable you to schedule and execute execution plan runs on external execution environments, refer to the interactive REST API documentation that is available from *host:port[/inst]/Services1.0/swagger-ui.html*, for example http://localhost:19120/Services1.0/swagger-ui.html.



Note: The WSDL URL also lists system-internal web services that are not intended for creating web services clients. This document describes only the published web services.

Refer to the *Javadoc* for full details regarding available Java classes and methods. If the link does not work, click **Help** > **Documentation** > **Silk Central API Specification** in the Silk Central menu to open the Javadoc.



Note: When you work with Web Services, all times returned by the system are in the Universal Time Coordinated (UTC). Use the UTC too in all time references in the Web Services you work with.

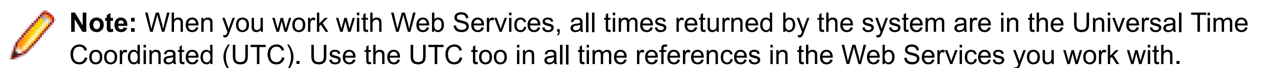
WS Name (Interface)	WSDL URL	Description
system (SystemService)	/Services1.0/jaxws/system?wsdl	This is the root service that provides authentication and simple utility methods.
administration (AdministrationService)	/Services1.0/jaxws/administration?wsdl	This service provides access to the *Project* and *Product* entities of Silk Central.

WS Name (Interface)	WSDL URL	Description
requirements (RequirementsService)	/Services1.0/jaxws/requirements?wsdl	This service provides access to the **Requirements** area of Silk Central.
tests (TestsService)	/Services1.0/jaxws/tests?wsdl	This service provides access to the **Tests** area of Silk Central.
executionplanning (ExecutionPlanningService)	/Services1.0/jaxws/executionplanning?wsdl	This service provides access to the **Execution Planning** area of Silk Central.
filter (FilterService)	/Services1.0/jaxws/filter?wsdl	This service allows you to create, read, update, and delete filters.
issuemanager (IssueManagerService)	/Services1.0/jaxws/issuemanager?wsdl	This service provides access to Issue Manager.

Services Exchange

This section describes the HTTP-based interfaces that are used to handle reports, attachments, test plans, executions, and libraries in Services Exchange.

You can also access Silk Central through Web Services. See *Available Web Services* for detailed information.



Note: When you work with Web Services, all times returned by the system are in the Universal Time Coordinated (UTC). Use the UTC too in all time references in the Web Services you work with.

reportData Interface

The reportData interface is used to request the data of a report. The following table shows the parameters of the reportData interface:

Interface URL	Parameters	Descriptions
http://<front-end URL>/servicesExchange?hid=reportData	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the session identifier by invoking the logonUser method of one of the *Available Web Services*.
	reportFilterID	Report filter ID
	type	Response body format: (csv or xml)
	includeHeaders	Include report headers or not. (true or false)
	projectID	ID of the project

Example: http://<front-end URL>/servicesExchange?hid=reportData&reportFilterID=<id>&type=<csv or xml>&includeHeaders=<true or false>&sid=<webServiceToken>&projectID=<id>

reportData Interface Example

```
String reportID = "<id>";
String webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";
String host = "<any_host>";

URL report = new URL("http", host, 19120,
   "/servicesExchange?hid=reportData" +
   "&type=xml" + // or csv
   "&sid=" + webServiceToken +
   "&reportFilterID=" + reportID +
   "&includeHeaders=true" +
   "&rp_execNode_Id_0=1" +
   "&projectID=27);

BufferedReader in = new BufferedReader(new
InputStreamReader(report.openStream(), "UTF-8"));

StringBuilder builder = new StringBuilder();
String line = "";

while ((line = in.readLine()) != null) {
   builder.append(line + "\n");
}

String text = builder.toString();
System.out.println(text);
```

If the report requires parameters, you need to add the following code to the report URL for each parameter:

```
"&rp_parametername=parametervalue"
```

In the example, the parameter rp_execNode_Id_0 is set to the value 1.

Note: The names of parameters that are passed to the reportData service have to be prefixed with rp_. Example: /servicesExchange?hid=reportData&type=xml&sid=<...>&reportFilterID=<...>&projectID=<...>&rp_TestID=<...>

TMAttach Interface

The TMAttach interface is used to upload attachments to a test or requirement. The following table shows the parameters of the TMAttach interface.

Interface URL	Parameter	Descriptions
http://<front-end URL>/servicesExchange?hid=TMAttach	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the session identifier by invoking the logonUser method of one of the *Available Web Services*.
	entityType	Target entity type: (test, requirement, or TestStepParent)
	entityID	Target entity ID:

Interface URL	Parameter	Descriptions
		(test ID, requirement ID, or manual test ID)
	description	Description of the attachment: URL encoded text, used to describe the attachment.
	isURL	If true, the attachment is a URL. If false, the attachment is a file.
	URL	Optional - URL to be attached.
	stepPosition	Optional - Test step order. Identifies the step of a manual test (for example, the order of the first step is 1). The order is mandatory if the entityType is TestStepParent.

Example: http://<front-end URL>/servicesExchange?hid=TMAttach&entityType=<test, requirement, or TestStepParent>&entityID=<id>&description=<text>&isURL=<true or false>&URL=<URL>&stepPosition=<number>&sid=<webServiceToken>

TMAttach Web Service Example

The following code uses Apache HtmlClient to get a convenient HTTP-POST API for uploading a binary attachment. Only one attachment can be uploaded per request.

Only one attachment can be uploaded per request. To download Apache HttpComponents, visit *http://hc.apache.org/downloads.cgi*. Refer to the documentation of the component for the required libraries.

```
import org.apache.commons.httpclient.*; // Apache HttpClient

   String webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5"; // Token
generated in the UI
   String testNodeID = null; // receiving test
   File fileToUpload = null; // attachment
   String AttachmentDescription = ""; // descriptive text

   HttpClient client = new HttpClient();
   String formURL = "http://localhost:19120/
      servicesExchange?hid=TMAttach" +
      "&sid=" + webServiceToken +
      "&entityID=" + testNodeID +
      "&entityType=Test" +
      "&isURL=false";
   PostMethod filePost = new PostMethod(formURL);
   Part[] parts = {
      new StringPart("description", attachmentDescription),
      new FilePart(fileToUpload.getName(), fileToUpload)
   };
   filePost.setRequestEntity(new MultipartRequestEntity(parts,
      filePost.getParams()));
   client.getHttpConnectionManager().
      getParams().setConnectionTimeout(60000);
   // Execute and check for success
   int status = client.executeMethod(filePost);
   // verify http return code...
   // if(status == httpStatus.SC_OK) ...
```

createTestPlan Interface

The createTestPlan interface is used to create new tests. The HTTP response of the call contains the XML structure of the changed tests. You can obtain the identifiers of the new nodes from the updated XML test structure.

The following table shows the parameters of the createTestPlan interface.

Interface URL	Parameter	Descriptions
http://<front-end URL>/servicesExchange?hid=createTestPlan	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the session identifier by invoking the logonUser method of one of the *Available Web Services*.
	parentNodeID	ID of the container to which the new test is added in the **Tests** tree

Example: http://<front-end URL>/servicesExchange?hid=createTestPlan&parentNodeID=<id>&sid=<webServiceToken>

The XML schema definition file that is used to validate test plans can be downloaded by using the front-end server URL http://<front-end URL>/silkroot/xsl/testplan.xsd or copied from the front-end server installation folder <Silk Central installation folder>/wwwroot/silkroot/xsl/testplan.xsd.

createTestPlan Web Service Example

The following code uses Apache HttpClient to create tests.

```
import org.apache.commons.httpclient.*; // Apache HttpClient

string webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5"; // The token
that you have generated in the UI

URL service = new URL("http", mWebServiceHelper.getHost(),
  mWebServiceHelper.getPort(),
  String.format("/servicesExchange?hid=%s&sid=%s&parentNodeID=%d",
  "createTestPlan", webServiceToken,
  PARENT_NODE_ID));

HttpClient client = new HttpClient();
PostMethod filePost = new PostMethod(service.toExternalForm());
String xmlFile = loadTestPlanUtf8("testPlan.xml");
StringPart xmlFileItem = new StringPart("testPlan", xmlFile,
  "UTF-8");
xmlFileItem.setContentType("text/xml");
Part[] parts = {xmlFileItem};

filePost.setRequestEntity(new MultipartRequestEntity(parts,
  filePost.getParams()));
client.getHttpConnectionManager().getParams().setConnectionTimeout(60000);
int status = client.executeMethod(filePost);
System.out.println(filePost.getStatusLine());
```

Only one attachment can be uploaded per request. To download Apache HttpComponents, visit *http://hc.apache.org/downloads.cgi*. Refer to the documentation of the component for the required libraries.

Test Example

The following code shows an example test that can be uploaded to Silk Central by using the createTestPlan and updateTestPlan service.

```
<?xml version="1.0" encoding="UTF-8"?>
 <TestPlan xmlns="http://www.borland.com/TestPlanSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://<front-end URL>/silkroot/xsl/testplan.xsd">

 <Folder name="Folder1" id="5438">
   <Description>Description of the folder</Description>
   <Property name="property1">
     <propertyValue>value1</propertyValue>
   </Property>
   <Test name="TestDef1" type="plugin.SilkTest">
     <Description>Description of the test</Description>
     <Property name="property2">
       <propertyValue>value2</propertyValue>
     </Property>
     <Property name="property3">
          <propertyValue>value3</propertyValue>
          <propertyValue>value4</propertyValue>
     </Property>
     <Parameter name="param1" type="string">string1</Parameter>
     <Parameter name="param2" type="boolean">true</Parameter>
                    <Parameter name="paramDate" type="date">01.01.2001</Parameter>
     <Parameter name="paramInherited" type="string"
       inherited="true">
       inheritedValue1
     </Parameter>
     <Step id="1" name="StepA">
       <ActionDescription>do it</ActionDescription>
       <ExpectedResult>everything</ExpectedResult>
     </Step>
     <Step id="2" name="StepB">
       <ActionDescription>and</ActionDescription>
       <ExpectedResult>everything should come</ExpectedResult>
     </Step>
   </Test>
   <Test name="ManualTest1" id="5441" type="_ManualTestType"
     plannedTime="03:45">
     <Description>Description of the manual test</Description>
     <Step id="1" name="StepA">
       <ActionDescription>do it</ActionDescription>
       <ExpectedResult>everything</ExpectedResult>
     </Step>
     <Step id="2" name="StepB">
       <ActionDescription>and</ActionDescription>
       <ExpectedResult>everything should come</ExpectedResult>
     </Step>
     <Step id="3" name="StepC">
       <ActionDescription>do it now"</ActionDescription>
       <ExpectedResult>
         everything should come as you wish
       </ExpectedResult>
     </Step>
   </Test>
   <Folder name="Folder2" id="5439">
```

```
    <Description>Description of the folder</Description>
    <Property name="property4">
      <propertyValue>value5</propertyValue>
    </Property>
    <Parameter name="param3" type="number">123</Parameter>
    <Folder name="Folder2_1" id="5442">
      <Description>Description of the folder</Description>
      <Test name="TestDef2" type="plugin.SilkPerformer">
        <Description>Description of the test</Description>
        <Property name="_sp_Project File">
          <propertyValue>ApplicationAccess.ltp</propertyValue>
        </Property>
        <Property name="_sp_Workload">
          <propertyValue>Workload1</propertyValue>
        </Property>
      </Test>
      <Test name="TestDef3" type="JUnitTestType"
        externalId="com.borland.MyTest">
        <Description>Description of the test</Description>
        <Property name="_junit_ClassFile">
          <propertyValue>com.borland.MyTest</propertyValue>
        </Property>
        <Property name="_junit_TestMethod">
          <propertyValue>testMethod</propertyValue>
        </Property>
        <Step id="1" name="StepA">
          <ActionDescription>do it</ActionDescription>
          <ExpectedResult>everything</ExpectedResult>
        </Step>
        <Step id="2" name="StepB">
          <ActionDescription>and</ActionDescription>
          <ExpectedResult>everything should come</ExpectedResult>
        </Step>
      </Test>
    </Folder>
  </Folder>
 </Folder>
</TestPlan>
```

exportTestPlan Interface

The exportTestPlan interface is used to export test plans as XML files. The following table shows the parameters of the exportTestPlan interface.

Interface URL	Parameter	Descriptions
http://<front-end URL>/servicesExchange?hid=exportTestPlan	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the session identifier by invoking the logonUser method of one of the *Available Web Services*.
	nodeID	The node with this ID and, recursively, all of the node's children nodes are exported

Example: http://<front-end URL>/servicesExchange?
hid=exportTestPlan&nodeID=<id>&sid=<webServiceToken>

exportTestPlan Web Service Example

The following code uses Apache HttpClient to export tests.

```
import org.apache.commons.httpclient.*; // Apache HttpClient

string webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";

URL service = new URL("http", mWebServiceHelper.getHost(),
  mWebServiceHelper.getPort(),
  String.format("/servicesExchange?hid=%s&sid=%s&nodeID=%d",
  "exportTestPlan", webServiceToken,
  PARENT_NODE_ID));

HttpClient client = new HttpClient();
client.getHttpConnectionManager().getParams().setConnectionTimeout(60000);
HttpMethod fileGet = new GetMethod(service.toExternalForm());
int status = client.executeMethod(fileGet);
System.out.println(fileGet.getStatusLine());
String exportedTestPlanResponse = fileGet.getResponseBodyAsString();
System.out.println(exportedTestPlanResponse);
```

To download Apache HttpComponents, visit *http://hc.apache.org/downloads.cgi*. Refer to the documentation of the component for the required libraries.

updateTestPlan Interface

The updateTestPlan interface is used to update tests with existing root nodes from XML files. The HTTP response of the call contains the XML structure of the changed tests. You can obtain the identifiers of the new nodes from the updated XML test structure.

The following table shows the parameters of the updateTestPlan interface.

Interface URL	Parameter	Descriptions
http://<front-end URL>/servicesExchange?hid=updateTestPlan	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the session identifier by invoking the logonUser method of one of the *Available Web Services*.

Example: http://<front-end URL>/servicesExchange?hid=updateTestPlan&sid=<webServiceToken>

The XML schema definition file that is used to validate test plans can be downloaded by using the front-end server URL http://<front-end URL>/silkroot/xsl/testplan.xsd or copied from the front-end server installation folder <Silk Central installation folder>/wwwroot/silkroot/xsl/testplan.xsd.

updateTestPlan Web Service Example

The following code uses Apache HttpClient to update tests.

```
import org.apache.commons.httpclient.*; // Apache HttpClient

String webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";
```

```
String xml = loadTestPlanUtf8(DEMO_TEST_PLAN_XML);
HttpClient client = new HttpClient();

URL webServiceUrl = new URL("http", mWebServiceHelper.getHost(),
  mWebServiceHelper.getPort(),
  String.format("/servicesExchange?hid=%s&sid=%s",
    "updateTestPlan",
    webServiceToken));
StringPart testPlanXml = new StringPart(DEMO_TEST_PLAN_XML, xml,
  "UTF-8");
testPlanXml.setContentType("text/xml");
Part[] parts = {testPlanXml};
PostMethod filePost = new PostMethod(webServiceUrl.toExternalForm());
filePost.setRequestEntity(new MultipartRequestEntity(parts,
  filePost.getParams()));
client.getHttpConnectionManager().getParams().setConnectionTimeout(60000);
int status = client.executeMethod(filePost);
System.out.println(filePost.getStatusLine());

String responseXml = filePost.getResponseBodyAsString();
```

Only one attachment can be uploaded per request. To download Apache HttpComponents, visit *http://hc.apache.org/downloads.cgi*. Refer to the documentation of the component for the required libraries.

Test Example

The following code shows an example test that can be uploaded to Silk Central by using the createTestPlan and updateTestPlan service.

```
<?xml version="1.0" encoding="UTF-8"?>
  <TestPlan xmlns="http://www.borland.com/TestPlanSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://<front-end URL>/silkroot/xsl/testplan.xsd">

  <Folder name="Folder1" id="5438">
    <Description>Description of the folder</Description>
    <Property name="property1">
      <propertyValue>value1</propertyValue>
    </Property>
    <Test name="TestDef1" type="plugin.SilkTest">
      <Description>Description of the test</Description>
      <Property name="property2">
        <propertyValue>value2</propertyValue>
      </Property>
      <Property name="property3">
           <propertyValue>value3</propertyValue>
           <propertyValue>value4</propertyValue>
      </Property>
      <Parameter name="param1" type="string">string1</Parameter>
      <Parameter name="param2" type="boolean">true</Parameter>
                  <Parameter name="paramDate" type="date">01.01.2001</Parameter>
      <Parameter name="paramInherited" type="string"
        inherited="true">
        inheritedValue1
      </Parameter>
      <Step id="1" name="StepA">
        <ActionDescription>do it</ActionDescription>
        <ExpectedResult>everything</ExpectedResult>
      </Step>
      <Step id="2" name="StepB">
        <ActionDescription>and</ActionDescription>
        <ExpectedResult>everything should come</ExpectedResult>
```

```
      </Step>
    </Test>
    <Test name="ManualTest1" id="5441" type="_ManualTestType"
      plannedTime="03:45">
      <Description>Description of the manual test</Description>
      <Step id="1" name="StepA">
        <ActionDescription>do it</ActionDescription>
        <ExpectedResult>everything</ExpectedResult>
      </Step>
      <Step id="2" name="StepB">
        <ActionDescription>and</ActionDescription>
        <ExpectedResult>everything should come</ExpectedResult>
      </Step>
      <Step id="3" name="StepC">
        <ActionDescription>do it now"</ActionDescription>
        <ExpectedResult>
          everything should come as you wish
        </ExpectedResult>
      </Step>
    </Test>
    <Folder name="Folder2" id="5439">
      <Description>Description of the folder</Description>
      <Property name="property4">
        <propertyValue>value5</propertyValue>
      </Property>
      <Parameter name="param3" type="number">123</Parameter>
      <Folder name="Folder2_1" id="5442">
        <Description>Description of the folder</Description>
        <Test name="TestDef2" type="plugin.SilkPerformer">
          <Description>Description of the test</Description>
          <Property name="_sp_Project File">
            <propertyValue>ApplicationAccess.ltp</propertyValue>
          </Property>
          <Property name="_sp_Workload">
            <propertyValue>Workload1</propertyValue>
          </Property>
        </Test>
        <Test name="TestDef3" type="JUnitTestType"
          externalId="com.borland.MyTest">
          <Description>Description of the test</Description>
          <Property name="_junit_ClassFile">
            <propertyValue>com.borland.MyTest</propertyValue>
          </Property>
          <Property name="_junit_TestMethod">
            <propertyValue>testMethod</propertyValue>
          </Property>
          <Step id="1" name="StepA">
            <ActionDescription>do it</ActionDescription>
            <ExpectedResult>everything</ExpectedResult>
          </Step>
          <Step id="2" name="StepB">
            <ActionDescription>and</ActionDescription>
            <ExpectedResult>everything should come</ExpectedResult>
          </Step>
        </Test>
      </Folder>
    </Folder>
  </Folder>
</TestPlan>
```

createRequirements Interface

The createRequirements interface is used to create new requirements. The HTTP response of the call contains the XML structure of the changed requirements. You can obtain the identifiers of the new nodes from the updated XML requirement structure.

The following table shows the parameters of the createRequirements interface.

Interface URL	Parameter	Descriptions
http://<front-end URL>/servicesExchange?hid=createRequirements	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the session identifier by invoking the logonUser method of one of the *Available Web Services*.
	parentNodeID	ID of the container to which the new requirement is added in the requirements tree

Example: http://<front-end URL>/servicesExchange?hid=createRequirements&parentNodeID=<id>&sid=<webServiceToken>

The XML schema definition file that is used to validate requirements can be downloaded by using the front-end server URL http://<front-end URL>/silkroot/xsl/requirements.xsd or copied from the front-end server installation folder <Silk Central installation folder>/wwwroot/silkroot/xsl/requirements.xsd.

createRequirements Web Service Example

The following code uses Apache HttpClient to create requirements.

```
import org.apache.commons.httpclient.*; // Apache HttpClient

String webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";

URL service = new URL("http", mWebServiceHelper.getHost(),
  mWebServiceHelper.getPort(),
  String.format("/servicesExchange?hid=%s&sid=%s&parentNodeID=%d",
  "createRequirements", webServiceToken,
  PARENT_NODE_ID));

HttpClient client = new HttpClient();
PostMethod filePost = new PostMethod(service.toExternalForm());
String xmlFile = loadRequirementsUtf8("requirements.xml");
StringPart xmlFileItem = new StringPart("requirements", xmlFile,
  "UTF-8");
xmlFileItem.setContentType("text/xml");
Part[] parts = {xmlFileItem};

filePost.setRequestEntity(new MultipartRequestEntity(parts,
  filePost.getParams()));
client.getHttpConnectionManager().getParams().setConnectionTimeout(60000);
int status = client.executeMethod(filePost);
System.out.println(filePost.getStatusLine());
```

Only one attachment can be uploaded per request. To download Apache HttpComponents, visit *http://hc.apache.org/downloads.cgi*. Refer to the documentation of the component for the required libraries.

Requirements Example

The following code shows an example requirement that can be uploaded to Silk Central by using the createRequirements, updateRequirements and updateRequirementsByExtId service.

```
<?xml version="1.0" encoding="UTF-8"?>
<Requirement id="0" name="name" xmlns="http://www.borland.com/
RequirementsSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://<front-end URL>/silkroot/xsl/requirements.xsd">
  <ExternalId>myExtId1</ExternalId>
  <Description>Description</Description>
  <Priority value="Low" inherited="false"/>
  <Risk value="Critical" inherited="false"/>
  <Reviewed value="true" inherited="false"/>
  <Property inherited="false" name="Document" type="string">MyDocument1.doc</
Property>
  <Requirement id="1" name="name" />
  <Requirement id="2" name="name1">
    <Requirement id="3" name="name" />
    <Requirement id="4" name="name1">
      <Requirement id="5" name="name" />
   <Requirement id="6" name="name1">
    <ExternalId>myExtId2</ExternalId>
    <Description>Another Description</Description>
    <Priority value="Medium" inherited="false"/>
    <Risk value="Critical" inherited="false"/>
    <Reviewed value="true" inherited="false"/>
    <Property inherited="false" name="Document" type="string">MyDocument2.doc</
Property>
      </Requirement>
    </Requirement>
  </Requirement>
</Requirement>
```

exportRequirements Interface

The exportRequirements interface is used to export requirements as XML files. The following table shows the parameters of the exportRequirements interface.

Interface URL	Parameter	Descriptions
http://<front-end URL>/servicesExchange?hid=exportRequirements	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the session identifier by invoking the logonUser method of one of the *Available Web Services*.

Interface URL	Parameter	Descriptions
	nodeID	The node with this ID and, recursively, all of the node's children nodes are exported
	includeObsolete	*Optional:* Specify true or false. Defaults to true if omitted. Specify false to exclude obsolete requirements.

Example: http://<front-end URL>/servicesExchange?hid=exportRequirements&nodeID=<id>&sid=<webServiceToken>

exportRequirements Web Service Example

The following code uses Apache HttpClient to export requirements.

```
import org.apache.commons.httpclient.*; // Apache HttpClient

String webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";

URL service = new URL("http", mWebServiceHelper.getHost(),
  mWebServiceHelper.getPort(),
  String.format("/servicesExchange?hid=%s&sid=%s&nodeID=%d",
  "exportRequirements", webServiceToken,
  PARENT_NODE_ID));

HttpClient client = new HttpClient();
client.getHttpConnectionManager().getParams().setConnectionTimeout(60000);
HttpMethod fileGet = new GetMethod(service.toExternalForm());
int status = client.executeMethod(fileGet);
System.out.println(fileGet.getStatusLine());
String exportedRequirementResponse = fileGet.getResponseBodyAsString();
System.out.println(exportedRequirementResponse);
```

To download Apache HttpComponents, visit *http://hc.apache.org/downloads.cgi*. Refer to the documentation of the component for the required libraries.

updateRequirements Interface

The updateRequirements interface is used to update requirements with existing root nodes from XML files. The requirements are identified by their internal Silk Central node ID in the requirements tree. The requirement tree node and all of the node's children are updated. New nodes are added, missing nodes are set to obsolete, and moved nodes are similarly moved in Silk Central. The HTTP response of the call contains the XML structure of the changed requirements. You can obtain the identifiers of the new nodes from the updated XML requirement structure.

The following table shows the parameters of the updateRequirements interface.

Interface URL	Parameter	Descriptions
http://<front-end URL>/servicesExchange?hid=updateRequirements	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the

Interface URL	Parameter	Descriptions
		session identifier by invoking the logonUser method of one of the *Available Web Services*.

Example: http://<front-end URL>/servicesExchange?hid=updateRequirements&sid=<webServiceToken>

The XML schema definition file that is used to validate requirements can be downloaded by using the front-end server URL http://<front-end URL>/silkroot/xsl/requirements.xsd or copied from the front-end server installation folder <Silk Central installation folder>/wwwroot/silkroot/xsl/requirements.xsd.

updateRequirements Web Service Example

The following code uses Apache HttpClient to update requirements.

```
import org.apache.commons.httpclient.*; // Apache HttpClient

String webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";
URL service = new URL("http", mWebServiceHelper.getHost(),
  mWebServiceHelper.getPort(),
  String.format("/servicesExchange?hid=%s&sid=%s",
    "updateRequirements", webServiceToken));

HttpClient client = new HttpClient();
PostMethod filePost = new PostMethod(service.toExternalForm());
string xmlFile = loadRequirementsUtf8(fileName);
StringPart xmlFileItem = new StringPart("requirements", xmlFile,
  "UTF-8");
xmlFileItem.setContentType("text/xml");
Part[] parts = {xmlFileItem};

filePost.setRequestEntity(new MultipartRequestEntity(parts,
  filePost.getParams()));
client.getHttpConnectionManager().getParams().setConnectionTimeout(60000);
int status = client.executeMethod(filePost);
System.out.println(filePost.getStatusLine());

String responseXml = filePost.getResponseBodyAsString();
```

Only one attachment can be uploaded per request. To download Apache HttpComponents, visit *http://hc.apache.org/downloads.cgi*. Refer to the documentation of the component for the required libraries.

Requirements Example

The following code shows an example requirement that can be uploaded to Silk Central by using the createRequirements, updateRequirements and updateRequirementsByExtId service.

```
<?xml version="1.0" encoding="UTF-8"?>
<Requirement id="0" name="name" xmlns="http://www.borland.com/
RequirementsSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://<front-end URL>/silkroot/xsl/requirements.xsd">
   <ExternalId>myExtId1</ExternalId>
   <Description>Description</Description>
   <Priority value="Low" inherited="false"/>
   <Risk value="Critical" inherited="false"/>
   <Reviewed value="true" inherited="false"/>
   <Property inherited="false" name="Document" type="string">MyDocument1.doc</
Property>
   <Requirement id="1" name="name" />
```

```
  <Requirement id="2" name="name1">
    <Requirement id="3" name="name" />
    <Requirement id="4" name="name1">
      <Requirement id="5" name="name" />
   <Requirement id="6" name="name1">
    <ExternalId>myExtId2</ExternalId>
    <Description>Another Description</Description>
    <Priority value="Medium" inherited="false"/>
    <Risk value="Critical" inherited="false"/>
    <Reviewed value="true" inherited="false"/>
    <Property inherited="false" name="Document" type="string">MyDocument2.doc</
Property>
      </Requirement>
    </Requirement>
  </Requirement>
</Requirement>
```

updateRequirementsByExtId Interface

The updateRequirementsByExtId interface is used to update requirements with existing root nodes from XML files. The requirements are identified by external IDs. The requirement tree node and all of the node's children are updated. New nodes are added, missing nodes are set to obsolete, and moved nodes are similarly moved in Silk Central. The HTTP response of the call contains the XML structure of the changed requirements. You can obtain the identifiers of the new nodes from the updated XML requirement structure.

The following table shows the parameters of the updateRequirementsByExtId interface.

Interface URL	Parameter	Descriptions
http://<front-end URL>/servicesExchange?hid=updateRequirementsByExtId	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the session identifier by invoking the logonUser method of one of the *Available Web Services*.
	nodeID	ID of the node in the requirement tree to be updated.

Example: http://<front-end URL>/servicesExchange?hid=updateRequirementsByExtId&nodeID=<id>&sid=<webServiceToken>

The XML schema definition file that is used to validate requirements can be downloaded by using the front-end server URL http://<front-end URL>/silkroot/xsl/requirements.xsd or copied from the front-end server installation folder <Silk Central installation folder>/wwwroot/silkroot/xsl/requirements.xsd.

updateRequirementsByExtId Web Service Example

The following code uses Apache HttpClient to update requirements.

```
import org.apache.commons.httpclient.*; // Apache HttpClient

String webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";
URL service = new URL("http", mWebServiceHelper.getHost(),
 mWebServiceHelper.getPort(),
 String.format("/servicesExchange?hid=%s&sid=%s",&nodeID=%s",
  "updateRequirementsByExtId",
```

```
    webServiceToken, rootNodeId));

HttpClient client = new HttpClient();
PostMethod filePost = new PostMethod(service.toExternalForm());
string xmlFile = loadRequirementsUtf8(fileName);
StringPart xmlFileItem = new StringPart("requirements", xmlFile,
  "UTF-8");
xmlFileItem.setContentType("text/xml");
Part[] parts = {xmlFileItem};

filePost.setRequestEntity(new MultipartRequestEntity(parts,
  filePost.getParams()));
client.getHttpConnectionManager().getParams().setConnectionTimeout(60000);
int status = client.executeMethod(filePost);
System.out.println(filePost.getStatusLine());

String responseXml = filePost.getResponseBodyAsString();
```

Only one attachment can be uploaded per request. To download Apache HttpComponents, visit *http://hc.apache.org/downloads.cgi*. Refer to the documentation of the component for the required libraries.

Requirements Example

The following code shows an example requirement that can be uploaded to Silk Central by using the createRequirements, updateRequirements and updateRequirementsByExtId service.

```
<?xml version="1.0" encoding="UTF-8"?>
<Requirement id="0" name="name" xmlns="http://www.borland.com/
RequirementsSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://<front-end URL>/silkroot/xsl/requirements.xsd">
    <ExternalId>myExtId1</ExternalId>
    <Description>Description</Description>
    <Priority value="Low" inherited="false"/>
    <Risk value="Critical" inherited="false"/>
    <Reviewed value="true" inherited="false"/>
    <Property inherited="false" name="Document" type="string">MyDocument1.doc</
Property>
    <Requirement id="1" name="name" />
    <Requirement id="2" name="name1">
        <Requirement id="3" name="name" />
        <Requirement id="4" name="name1">
            <Requirement id="5" name="name" />
      <Requirement id="6" name="name1">
        <ExternalId>myExtId2</ExternalId>
        <Description>Another Description</Description>
        <Priority value="Medium" inherited="false"/>
        <Risk value="Critical" inherited="false"/>
        <Reviewed value="true" inherited="false"/>
        <Property inherited="false" name="Document" type="string">MyDocument2.doc</
Property>
            </Requirement>
        </Requirement>
    </Requirement>
</Requirement>
```

createExecutionDefinitions Interface

The createExecutionDefinitions interface is used to create new execution plans. The HTTP response of the call contains the XML structure of the changed execution plans. You can obtain the identifiers of the new nodes from the updated XML execution plan structure.

The following table shows the parameters of the createExecutionDefinitions interface.

Interface URL	Parameter	Descriptions
http://<front-end URL>/servicesExchange?hid=createExecutionDefinitions	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the session identifier by invoking the logonUser method of one of the *Available Web Services*.
	parentNodeID	ID of the node to which the new execution plan is added in the execution tree

Example: http://<front-end URL>/servicesExchange?hid=createExecutionDefinitions&parentNodeID=<id>&sid=<webServiceToken>

The XML schema definition file that is used to validate executions can be downloaded by using the front-end server URL http://<front-end URL>/silkroot/xsl/executionplan.xsd or copied from the front-end server installation folder <Silk Central installation folder>/wwwroot/silkroot/xsl/executionplan.xsd.

createExecutionDefinitions Web Service Example

The following code uses Apache HttpClient to create execution plans.

```
import org.apache.commons.httpclient.*; // Apache HttpClient

String webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";

URL service = new URL("http", mWebServiceHelper.getHost(),
  mWebServiceHelper.getPort(),
  String.format("/servicesExchange?hid=%s&sid=%s&parentNodeID=%d",
  "createExecutionDefinitions", webServiceToken,
  PARENT_NODE_ID));

HttpClient client = new HttpClient();
PostMethod filePost = new PostMethod(service.toExternalForm());
String xmlFile = loadExecutionDefinitionsUtf8("executionplan.xml");
StringPart xmlFileItem = new StringPart("executionplan", xmlFile,
  "UTF-8");
xmlFileItem.setContentType("text/xml");
Part[] parts = {xmlFileItem};

filePost.setRequestEntity(new MultipartRequestEntity(parts,
  filePost.getParams()));
client.getHttpConnectionManager().getParams().setConnectionTimeout(60000);
int status = client.executeMethod(filePost);
System.out.println(filePost.getStatusLine());
```

Only one attachment can be uploaded per request. To download Apache HttpComponents, visit *http://hc.apache.org/downloads.cgi*. Refer to the documentation of the component for the required libraries.

Execution Plan Example

The following code shows an example execution plan that can be uploaded to Silk Central by using the createExecutionDefinitions and updateExecutionDefinitions service. The example creates a custom schedule for one of the execution definitions and assigns tests to an execution plan, both through a manual assignment and a filter. The example also creates a configuration suite with configurations.

```
<?xml version="1.0" encoding="UTF-8"?>
<ExecutionPlan xmlns="http://www.borland.com/ExecPlanSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://<front-end URL>/silkroot/xsl/executionplan.xsd">

 <Folder name="Folder1">
   <Description>Description of the folder</Description>
   <ExecDef name="ExecutionDefinition1" TestContainerId="1">
     <Description>Description1</Description>
     <CustomSchedule>
       <start>2009-11-26T21:32:52</start>
       <end>
         <forever>true</forever>
       </end>
       <Interval day="1" hour="2" minute="3"></Interval>
       <adjustDaylightSaving>false</adjustDaylightSaving>
       <exclusions>
         <days>Monday</days>
         <days>Wednesday</days>
         <from>21:32:52</from>
         <to>22:32:52</to>
       </exclusions>
       <definiteRun>2009-11-27T21:35:12</definiteRun>
     </CustomSchedule>
     <ReadFromBuildInfoFile>true</ReadFromBuildInfoFile>
     <Priority>High</Priority>
     <SetupTestDefinition>73</SetupTestDefinition>
     <CleanupTestDefinition>65</CleanupTestDefinition>
     <AssignedTestDefinitions>
       <ManualAssignment useTestPlanOrder="true">
         <TestId>6</TestId>
         <TestId>5</TestId>
       </ManualAssignment>
     </AssignedTestDefinitions>
   </ExecDef>
   <ExecDef name="ExecutionDefinition2" TestContainerId="1">
     <Description>Description2</Description>
     <Build>1</Build>
     <Version>1</Version>
     <Priority>Low</Priority>
     <SourceControlLabel>Label1</SourceControlLabel>
     <DependentExecDef id="65">
       <Condition>Passed</Condition>
       <Deployment>
         <Specific>
           <Execution type="Server" id="1"/>
           <Execution type="Tester" id="0"/>
         </Specific>
       </Deployment>
     </DependentExecDef>
```

```
  <DependentExecDef id="70">
    <Condition>Failed</Condition>
    <Deployment>
      <Specific>
        <Execution type="Tester" id="0"/>
      </Specific>
    </Deployment>
  </DependentExecDef>
  <DependentExecDef id="68">
    <Condition>Any</Condition>
    <Deployment>
      <UseFromCurrentExedDef>true</UseFromCurrentExedDef>
    </Deployment>
  </DependentExecDef>
</ExecDef>

<ConfigSuite name="ConfigSuite1" TestContainerId="1">
  <Description>ConfigSuite1 desc</Description>
  <CustomSchedule>
    <start>2009-11-26T21:32:52</start>
    <end>
      <times>1</times>
    </end>
    <Interval day="1" hour="2" minute="3"/>
    <adjustDaylightSaving>false</adjustDaylightSaving>
    <exclusions>
      <days>Monday</days>
      <days>Wednesday</days>
      <from>21:32:52</from>
      <to>22:32:52</to>
    </exclusions>
    <definiteRun>2009-11-27T21:35:12</definiteRun>
  </CustomSchedule>

  <ConfigExecDef name="Config1">
    <Description>Config1 desc</Description>
    <Priority>Medium</Priority>
  </ConfigExecDef>

  <ConfigExecDef name="Config2">
    <Priority>Medium</Priority>
    <DependentExecDef id="69">
      <Condition>Any</Condition>
      <Deployment>
        <UseFromCurrentExedDef>true</UseFromCurrentExedDef>
      </Deployment>
    </DependentExecDef>
  </ConfigExecDef>

  <Build>8</Build>
  <Version>2</Version>
  <SourceControlLabel>ConfigSuite1 label</SourceControlLabel>
  <SetupTestDefinition>73</SetupTestDefinition>
  <CleanupTestDefinition>65</CleanupTestDefinition>
  <AssignedTestDefinitions>
    <ManualAssignment useTestPlanOrder="true">
      <TestId>6</TestId>
      <TestId>5</TestId>
    </ManualAssignment>
  </AssignedTestDefinitions>
</ConfigSuite>
```

```
  </Folder>
</ExecutionPlan>
```

exportExecutionDefinitions Interface

The exportExecutionDefinitions interface is used to export execution plans as XML files. The following table shows the parameters of the exportExecutionDefinitions interface.

Interface URL	Parameter	Descriptions
http://<front-end URL>/servicesExchange?hid=exportExecutionDefinitions	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the session identifier by invoking the logonUser method of one of the *Available Web Services*.
	nodeID	The node with this ID and, recursively, all of the node's children nodes are exported

Example: http://<front-end URL>/servicesExchange?hid=exportExecutionDefinitions&nodeID=<id>&sid=<webServiceToken>

exportExecutionDefinitions Web Service Example

The following code uses Apache HttpClient to export execution plans.

```
import org.apache.commons.httpclient.*; // Apache HttpClient

String webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";

URL service = new URL("http", mWebServiceHelper.getHost(),
  mWebServiceHelper.getPort(),
  String.format("/servicesExchange?hid=%s&sid=%s&nodeID=%d",
  "exportExecutionDefinitions", webServiceToken,
  NODE_ID));

HttpClient client = new HttpClient();
client.getHttpConnectionManager().getParams().setConnectionTimeout(60000);
HttpMethod fileGet = new GetMethod(service.toExternalForm());
int status = client.executeMethod(fileGet);
System.out.println(fileGet.getStatusLine());
String exportedExecutionPlanResponse = fileGet.getResponseBodyAsString();
System.out.println(exportedExecutionPlanResponse);
```

To download Apache HttpComponents, visit *http://hc.apache.org/downloads.cgi*. Refer to the documentation of the component for the required libraries.

updateExecutionDefinitions Interface

The updateExecutionDefinitions interface is used to update execution plans from XML files. The HTTP response of the call contains the XML structure of the changed execution plans. You can obtain the identifiers of the new nodes from the updated XML execution plan structure.

The following table shows the parameters of the updateExecutionDefinitions interface.

Interface URL	Parameter	Descriptions
http://<front-end URL>/servicesExchange?hid=updateExecutionDefinitions	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the session identifier by invoking the logonUser method of one of the *Available Web Services*.

Example: http://<front-end URL>/servicesExchange?hid=updateExecutionDefinitions&sid=<webServiceToken>

The XML schema definition file that is used to validate executions can be downloaded by using the front-end server URL http://<front-end URL>/silkroot/xsl/executionplan.xsd or copied from the front-end server installation folder <Silk Central installation folder>/wwwroot/silkroot/xsl/executionplan.xsd.

updateExecutionDefinitions Web Service Example

The following code uses Apache HttpClient to update execution plans.

```
import org.apache.commons.httpclient.*; // Apache HttpClient

String webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";
string xml = loadExecutionPlanUtf8(DEMO_EXECUTION_PLAN_XML);
HttpClient client = new HttpClient();

URL webServiceUrl = new URL("http", mWebServiceHelper.getHost(),
  mWebServiceHelper.getPort(),
  String.format("/servicesExchange?hid=%s&sid=%s",
    "updateExecutionDefinitions",
    webServiceToken));
StringPart ExecutionPlanXml = new StringPart(DEMO_EXECUTION_PLAN_XML, xml,
  "UTF-8");
ExecutionPlanXml.setContentType("text/xml");
Part[] parts = {ExecutionPlanXml};
PostMethod filePost = new PostMethod(webServiceUrl.toExternalForm());
filePost.setRequestEntity(new MultipartRequestEntity(parts,
  filePost.getParams()));
client.getHttpConnectionManager().getParams().setConnectionTimeout(60000);
int status = client.executeMethod(filePost);
System.out.println(filePost.getStatusLine());

String responseXml = filePost.getResponseBodyAsString();
```

Only one attachment can be uploaded per request. To download Apache HttpComponents, visit *http://hc.apache.org/downloads.cgi*. Refer to the documentation of the component for the required libraries.

Execution Plan Example

The following code shows an example execution plan that can be uploaded to Silk Central by using the createExecutionDefinitions and updateExecutionDefinitions service. The example creates a custom schedule for one of the execution definitions

and assigns tests to an execution plan, both through a manual assignment and a filter. The example also creates a configuration suite with configurations.

```
<?xml version="1.0" encoding="UTF-8"?>
<ExecutionPlan xmlns="http://www.borland.com/ExecPlanSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://<front-end URL>/silkroot/xsl/executionplan.xsd">

 <Folder name="Folder1">
  <Description>Description of the folder</Description>
  <ExecDef name="ExecutionDefinition1" TestContainerId="1">
   <Description>Description1</Description>
   <CustomSchedule>
    <start>2009-11-26T21:32:52</start>
    <end>
     <forever>true</forever>
    </end>
    <Interval day="1" hour="2" minute="3"></Interval>
    <adjustDaylightSaving>false</adjustDaylightSaving>
    <exclusions>
     <days>Monday</days>
     <days>Wednesday</days>
     <from>21:32:52</from>
     <to>22:32:52</to>
    </exclusions>
    <definiteRun>2009-11-27T21:35:12</definiteRun>
   </CustomSchedule>
   <ReadFromBuildInfoFile>true</ReadFromBuildInfoFile>
   <Priority>High</Priority>
   <SetupTestDefinition>73</SetupTestDefinition>
   <CleanupTestDefinition>65</CleanupTestDefinition>
   <AssignedTestDefinitions>
    <ManualAssignment useTestPlanOrder="true">
     <TestId>6</TestId>
     <TestId>5</TestId>
    </ManualAssignment>
   </AssignedTestDefinitions>
  </ExecDef>
  <ExecDef name="ExecutionDefinition2" TestContainerId="1">
   <Description>Description2</Description>
   <Build>1</Build>
   <Version>1</Version>
   <Priority>Low</Priority>
   <SourceControlLabel>Label1</SourceControlLabel>
   <DependentExecDef id="65">
    <Condition>Passed</Condition>
    <Deployment>
     <Specific>
      <Execution type="Server" id="1"/>
      <Execution type="Tester" id="0"/>
     </Specific>
    </Deployment>
   </DependentExecDef>
   <DependentExecDef id="70">
    <Condition>Failed</Condition>
    <Deployment>
     <Specific>
      <Execution type="Tester" id="0"/>
     </Specific>
    </Deployment>
   </DependentExecDef>
   <DependentExecDef id="68">
    <Condition>Any</Condition>
    <Deployment>
```

```
      <UseFromCurrentExedDef>true</UseFromCurrentExedDef>
    </Deployment>
  </DependentExecDef>
</ExecDef>

<ConfigSuite name="ConfigSuite1" TestContainerId="1">
  <Description>ConfigSuite1 desc</Description>
  <CustomSchedule>
    <start>2009-11-26T21:32:52</start>
    <end>
      <times>1</times>
    </end>
    <Interval day="1" hour="2" minute="3"/>
    <adjustDaylightSaving>false</adjustDaylightSaving>
    <exclusions>
      <days>Monday</days>
      <days>Wednesday</days>
      <from>21:32:52</from>
      <to>22:32:52</to>
    </exclusions>
    <definiteRun>2009-11-27T21:35:12</definiteRun>
  </CustomSchedule>

  <ConfigExecDef name="Config1">
    <Description>Config1 desc</Description>
    <Priority>Medium</Priority>
  </ConfigExecDef>

  <ConfigExecDef name="Config2">
    <Priority>Medium</Priority>
    <DependentExecDef id="69">
      <Condition>Any</Condition>
      <Deployment>
        <UseFromCurrentExedDef>true</UseFromCurrentExedDef>
      </Deployment>
    </DependentExecDef>
  </ConfigExecDef>

  <Build>8</Build>
  <Version>2</Version>
  <SourceControlLabel>ConfigSuite1 label</SourceControlLabel>
  <SetupTestDefinition>73</SetupTestDefinition>
  <CleanupTestDefinition>65</CleanupTestDefinition>
  <AssignedTestDefinitions>
    <ManualAssignment useTestPlanOrder="true">
      <TestId>6</TestId>
      <TestId>5</TestId>
    </ManualAssignment>
  </AssignedTestDefinitions>
</ConfigSuite>
</Folder>
</ExecutionPlan>
```

createLibraries Interface

The createLibraries interface is used to create new libraries. The HTTP response of the call contains the XML structure of the changed libraries. You can obtain the identifiers of the new nodes from the updated XML library structure.

The following table shows the parameters of the createLibraries interface.

Interface URL	Parameter	Descriptions
http://<front-end URL>/servicesExchange?hid=createLibraries	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the session identifier by invoking the logonUser method of one of the *Available Web Services*.

Example: http://<front-end URL>/servicesExchange?hid=createLibraries&sid=<webServiceToken>

The XML schema definition file that is used to validate libraries can be downloaded by using the front-end server URL http://<front-end URL>/silkroot/xsl/libraries.xsd or copied from the front-end server installation folder <Silk Central installation folder>/wwwroot/silkroot/xsl/libraries.xsd.

createLibraries Web Service Example

The following code uses Apache HttpClient to create libraries.

```
import org.apache.commons.httpclient.*; // Apache HttpClient

String webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";

URL service = new URL("http", mWebServiceHelper.getHost(),
  mWebServiceHelper.getPort(), String.format("/servicesExchange?hid=%s&sid=%s",
  "createLibraries", webServiceToken));

HttpClient client = new HttpClient();
PostMethod filePost = new PostMethod(service.toExternalForm());
String xmlFile = loadTestPlanUtf8("libraries.xml");
StringPart xmlFileItem = new StringPart("libraries", xmlFile, "UTF-8");
xmlFileItem.setContentType("text/xml");
Part[] parts = {xmlFileItem};

filePost.setRequestEntity(new MultipartRequestEntity(parts, filePost.getParams()));
client.getHttpConnectionManager().getParams().setConnectionTimeout(60000);
int status = client.executeMethod(filePost);
System.out.println(filePost.getStatusLine());
```

To download Apache HttpComponents, visit *http://hc.apache.org/downloads.cgi*. Refer to the documentation of the component for the required libraries.

Libraries Example

The following code shows an example library that can be uploaded to Silk Central by using the createLibraries service. A new library is not restricted to be used in certain projects, unless one or more projects are defined in the GrantedProjects Section.

```
<?xml version="1.0" encoding="UTF-8"?>
<LibraryStructure xmlns="http://www.borland.com/TestPlanSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://<front-end URL>/silkroot/xsl/libraries.xsd">

  <Library name="Library 1">
    <Folder name="Folder 1">
      <Folder name="Folder 1.1">
        <SharedSteps name="Basic create user steps">
```

```
        <Step name="Login">
          <ActionDescription>
            Login with user admin.
          </ActionDescription>
          <ExpectedResult>Succesful login.</ExpectedResult>
          <CustomStepProperty name="Step Property 1">
                  <propertyValue>Step Property Value</propertyValue>
          </CustomStepProperty>
        </Step>
        <Step name="Create User">
          <ActionDescription>Create user tester</ActionDescription>
          <ExpectedResult>User created</ExpectedResult>
          <CustomStepProperty name="Step Property 1">
                  <propertyValue>Step Property Value</propertyValue>
          </CustomStepProperty>
        </Step>
        <Step name="Logout">
          <ActionDescription>
            Logout using start menu
          </ActionDescription>
          <ExpectedResult>Logged out.</ExpectedResult>
          <CustomStepProperty name="Step Property 1">
                  <propertyValue>Step Property Value</propertyValue>
          </CustomStepProperty>
        </Step>
      </SharedSteps>
    </Folder>
  </Folder>
  <GrantedProjects>
    <ProjectId>0</ProjectId>
    <ProjectId>1</ProjectId>
  </GrantedProjects>
 </Library>
</LibraryStructure>
```

exportLibraryStructure Interface

The exportLibraryStructure interface is used to export libraries, folders, and shared steps objects as XML files. The following table shows the parameters of the exportLibraryStructure interface.

Interface URL	Parameter	Descriptions
http://<front-end URL>/servicesExchange?hid=exportLibraryStructure	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the session identifier by invoking the logonUser method of one of the *Available Web Services*.
	nodeID	The library node or folder in the libraries tree that is to be exported. IDs of shared steps nodes are not allowed.

Example: http://<front-end URL>/servicesExchange?hid=exportLibraryStructure&sid=<webServiceToken>&nodeID=<id>

exportLibraryStructure Web Service Example

The following code uses Apache HttpClient to export libraries.

```
import org.apache.commons.httpclient.*; // Apache HttpClient

String webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";

URL service = new URL("http", mWebServiceHelper.getHost(),
  mWebServiceHelper.getPort(), String.format("/servicesExchange?hid=%s&sid=
%s&nodeID=%d",
  "exportLibraryStructure", webServiceToken, NODE_ID));

HttpClient client = new HttpClient();
client.getHttpConnectionManager().getParams().setConnectionTimeout(60000);
HttpMethod fileGet = new GetMethod(service.toExternalForm());
int status = client.executeMethod(fileGet);
System.out.println(fileGet.getStatusLine());
String exportedTestPlanResponse = fileGet.getResponseBodyAsString();
System.out.println(exportedTestPlanResponse);
```

To download Apache HttpComponents, visit *http://hc.apache.org/downloads.cgi*. Refer to the documentation of the component for the required libraries.

exportLibraryStructureWithoutSteps Interface

The exportLibraryStructureWithoutSteps interface is used to export libraries, folders, and shared steps objects as XML files. The steps included in the shared steps objects are not exported. The following table shows the parameters of the exportLibraryStructureWithoutSteps interface.

Interface URL	Parameter	Descriptions
http://<front-end URL>/servicesExchange?hid=exportLibraryStructureWithoutSteps	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the session identifier by invoking the logonUser method of one of the *Available Web Services*.
	nodeID	The library node or folder in the libraries tree that is to be exported. IDs of shared steps nodes are not allowed.

Example: http://<front-end URL>/servicesExchange?hid=exportLibraryStructureWithoutSteps&sid=<webServiceToken>&nodeID=<id>

exportLibraryStructureWithoutSteps Web Service Example

The following code uses Apache HttpClient to export libraries.

```
import org.apache.commons.httpclient.*; // Apache HttpClient

String webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";

URL service = new URL("http", mWebServiceHelper.getHost(),
  mWebServiceHelper.getPort(), String.format("/servicesExchange?hid=%s&sid=
```

```
%s&nodeID=%d",
  "exportLibraryStructureWithoutSteps", webServiceToken, NODE_ID));

HttpClient client = new HttpClient();
client.getHttpConnectionManager().getParams().setConnectionTimeout(60000);
HttpMethod fileGet = new GetMethod(service.toExternalForm());
int status = client.executeMethod(fileGet);
System.out.println(fileGet.getStatusLine());
String exportedTestPlanResponse = fileGet.getResponseBodyAsString();
System.out.println(exportedTestPlanResponse);
```

To download Apache HttpComponents, visit *http://hc.apache.org/downloads.cgi*. Refer to the documentation of the component for the required libraries.

getLibraryInfoByName Interface

The getLibraryInfoByName interface returns the ID, name, and description of all libraries with a specified name. The interface returns only the properties of libraries, not their structure. The following table shows the parameters of the getLibraryInfoByName interface.

Interface URL	Parameter	Descriptions
http://<front-end URL>/servicesExchange?hid=getLibraryInfoByName	sid	Web-service token or session identifier for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. You can retrieve the session identifier by invoking the logonUser method of one of the *Available Web Services*.
	libraryName	The name of the library

Example: http://<front-end URL>/servicesExchange?hid=getLibraryInfoByName&sid=<webServicesToken>&libraryName=<name>

getLibraryInfoByName Web Service Example

The following code uses Apache HttpClient to get library information.

```
import org.apache.commons.httpclient.*; // Apache HttpClient

String webServiceToken = "e39a0b5b-45db-42db-84b2-b85028d954d5";

URL service = new URL("http", mWebServiceHelper.getHost(),
  mWebServiceHelper.getPort(), String.format("/servicesExchange?hid=%s&sid=%s",
  "getLibraryInfoByName", webServiceToken, LIBRARY_NAME));

HttpClient client = new HttpClient();
client.getHttpConnectionManager().getParams().setConnectionTimeout(60000);
HttpMethod fileGet = new GetMethod(service.toExternalForm());
int status = client.executeMethod(fileGet);
System.out.println(fileGet.getStatusLine());
String response = fileGet.getResponseBodyAsString();
System.out.println(response);
```

To download Apache HttpComponents, visit *http://hc.apache.org/downloads.cgi*. Refer to the documentation of the component for the required libraries.

Web Service Demo Client

The Web Service Demo Client is a tool that demonstrates how to use the Silk Central Web Services. Download the client from **Help** > **Tools**.

The Web Service Demo Client shows all the attributes that are available in **Tests** > **Manage Test Attributes** for each test, and all properties for each available test type.

Attention: The Web Service Demo Client is designed to demonstrate usage of the Web Services. Do not use the demo client in a production environment.

Triggering Silk Central from a CI Server

This section describes how you can better integrate Silk Central into your continuous integration (CI) processes by triggering executions on Silk Central from your CI server using a Gradle script.

Additionally, this section describes how to get results from Silk Central and how to use these results in your build process.

To trigger executions on Silk Central from a CI server and to collect the execution results from Silk Central, you need to add a Gradle script with the appropriate commands to your source control. You can download the silkcentral.gradle file from the Silk Central UI. Navigate to **Help** > **Tools** and click **Gradle script for CI server integration**.

You can configure the following properties in the Gradle script:

Property	Description
sc_executionNodeIds	The comma-separated list of execution plans to be started. This list should not contain folders. For example 22431,22432,22433.
sc_host	The Silk Central host. For example http://[sc_server_name]:19120.
sc_token	The web-service token for user authentication. You can generate the web-service token in the **Settings Page** of the Silk Central UI. To access this page, hover the mouse cursor over the user name in the Silk Central menu and select **User Settings**. For example 80827e02-cfda-4d2d-b0aa-2d5205eb6eq9.
sc_sourceControlBranch	*Optional:* Specify this property to check out a specific branch. If no branch is specified, the setting of the execution plan is used.
sc_ buildName	*Optional:* The build for the run. If no build is specified, the setting of the execution plan is used. You can only set this property for execution plan nodes, not for folders.
sc_StartOption	*Optional:* The tests that should be executed. If this is not specified, all assigned tests will be executed. The allowed values are: • ALL • FAILED • NOT_EXECUTED • NOT_EXECUTED_SINCE_BUILD • FAILED_NOTEXECUTED_SINCE_BUILD • HAVING_FIXED_ISSUES The default value is ALL.
sc_sinceBuild	*Optional:* The name of the build since which tests have not been executed. Specify this property if the sc_StartOption property is set to FAILED_NOTEXECUTED_SINCE_BUILD or NOT_EXECUTED_SINCE_BUILD.
sc_collectResults	*Optional:* If true, the script will wait until the Silk Central execution is finished and will write results files in JUnit format. If false, the script will trigger the execution and will finish without waiting for the results. The files will be stored in the subfolder sc_results. The default value is true. Boolean.

Property	Description
sc_collectResultFiles	*Optional:* If true and sc_collectResults is true, the script will download the result files generated by the tests. The files will be stored in the subfolder sc_results. The default value is false. Boolean.
sc_startDelay	*Optional:* The start delay in seconds. Can be specified if you have more than one execution plan to execute. The execution plans will be started sequentially with the specified delay between starts. This can be helpful if you need to minimize the workload on the test environment at startup, for example when starting a virtual machine or when installing the application under test. The default value is 0.

You can either specify the properties directly in the script or pass them when triggering the script.

All additional project properties that are specified when triggering the script will be passed as parameters to Silk Central and are used for the execution. This enables you to parameterize the executions in Silk Central with values from the build server.

For example, if your build starts a test server in Docker, you can pass the URL to this server by specifying the property in the command line:

```
-PmyServerUrl=http://docker:1234
```

Command Line Example

The following command launches the script from the command line, starting the execution tree nodes 22431,22432, and 22433 on localhost and using the web-service token 80827e02-cfda-4d2d-b0aa-2d5205eb6ea9 for authentication:

```
gradle -b silkcentral.gradle
:silkCentralLaunch -Psc_executionNodeIds='22431,22432,22433'
-Psc_host='http://localhost:19120'
-Psc_token='80827e02-cfda-4d2d-b0aa-2d5205eb6ea9'
```

For specific information on triggering executions on Silk Central from Jenkins, see *Triggering Executions from Jenkins*. For specific information on triggering executions on Silk Central from TeamCity, see *Triggering Executions from TeamCity*.

Triggering Executions from Jenkins

If your build process is not already using Gradle, ensure Jenkins can execute Gradle scripts.

To trigger executions in Silk Central from Jenkins:

1. Install Gradle in Jenkins under **Manage Jenkins** > **Global Tool Configuration**.
2. In your Jenkins project, add a build step **Invoke Gradle script**.

   Depending on where you have stored the Gradle script, you need to adapt the **Build File** property. Configure the step like in the following screenshot:

a) As shown in the screenshot, you can use variables available in Jenkins, like *$BUILD_NUMBER*, to configure the script.
b) If your Jenkins project is parameterized, you can pass all parameters directly to Silk Central by checking **Pass all job parameters as Project properties**.

3. To show test results in Jenkins, add a post-build action **Publish JUnit test result report** to the Jenkins project.
4. Specify the location, to which the script writes the files to, in the **Test report XMLs** field.
   For example sc_results/junit*.xml.

5. *Alternative:* You can also use a pipeline script to configure Jenkins and to trigger executions in Silk Central.
   The following sample pipeline script triggers two executions in Silk Central and collects the results. The Gradle installation has the name Gradle5.4.

```
node () {
  stage("Trigger Silk Central Executions") {
    def path = tool name: 'Gradle5.4', type: 'gradle'
    def scFile = new File(pwd(), "silkcentral.gradle")
    scFile.delete()
```

```
    scFile.getParentFile().mkdirs()
    writeFile([file: scFile.getAbsolutePath(), text: new URL ("http://scHost:19120/silkroot/tools/
silkcentral.gradle").getText()])
    def scTriggerInfo = '-Psc_executionNodeIds=6164,6123 -Psc_host=http://scHost:19120 -
Psc_token=d28930f4-9c77-4fc7-bc1d-aac4cd235d33'
    if (isUnix()) {
      sh "${path}/bin/gradle :silkCentralLaunch -b ${scFile} " + scTriggerInfo
    } else {
      bat "${path}/bin/gradle.bat :silkCentralLaunch -b ${scFile} " + scTriggerInfo
    }
    junit 'sc_results/junit*.xml'
  }
}
```

Triggering Executions from TeamCity

To trigger executions in Silk Central from TeamCity:

1. Add a build step to the build in TeamCity:
   a) Select **Gradle** as the **Runner Type**.
   b) Specify silkCentralLaunch in the **Gradle task** field.
   c) Browse to and select the silkcentral.gradle file in the **Gradle build file** field.
   d) Specify any additional Gradle command-line parameters in the **Additional Gradle command line parameters** field.

   Additional Gradle command line parameters:

   ```
   -Psc_executionNodeIds=22431,22432,22433
   -Psc_host=http://sc_server:19120
   -Psc_token=80827e02-cfda-4d2d-b0aa-2d5205eb6ea9
   -Psc_buildName=%env.BUILD_NUMBER%
   -PmyParamFromJenkins=testvalue
   ```

   Additional parameters will be added to the 'Gradle' command line.

2. To process the test results from Silk Central in TeamCity, add the build feature **XML report processing** to the build in TeamCity.
3. Configure the **XML report processing** build feature.
   a) Select the **Report type**.
   b) Specify the location , to which the script writes the files to, in the **Monitoring rules** field.

      For example sc_results/*.xml.

   Report type: * Ant JUnit
   Choose a report type.

   Monitoring rules: *
   Type report monitoring rules:

   ```
   sc_results/*.xml
   ```

   Newline- or comma-separated set of rules in the form of
   +|-:path.
   Ant-style wildcards supported, e.g. **dir/**/*.xml**

Index

A

Apache Axis 22
API
    overview 4
API structure
    third-party test type plug-in 15
APIs
    code coverage integration 5
authentication
    web services 25

C

CI servers
    triggering executions, Gradle 53
classes 12
cloud integration 21
cloud plug-in 21
code coverage
    APIs 5
compiling plug-ins 4
createExecutionDefinitions
    examples 42
    interfaces 42
createLibraries
    examples 48
    interfaces 48
createRequirements
    examples 36
    interfaces 36
createTestPlan
    examples 30
    interfaces 30
custom icons
    third-party test type plug-in 20

D

demo client
    Web Services interface 53
deployment
    plug-ins 4
    third-party test type plug-in 20
distribution
    plug-ins 4

E

exportExecutionDefinitions
    examples 45
    interfaces 45
exportLibraryStructure
    examples 50
    interfaces 50
exportLibraryStructureWithoutSteps
    examples 51
    interfaces 51
exportRequirements
    examples 37
    interfaces 37
exportTestPlan
    examples 32
    interfaces 32
external results
    uploading 4

F

file property meta information
    third-party test type plug-in 19

G

general property meta information
    third-party test type plug-in 19
getLibraryInfoByName
    examples 52
    interfaces 52
Gradle
    collecting results 53
    executions, triggering on CI servers 53

I

icons
    custom 20
implementation
    third-party test type plug-in 14
integration
    third-party test type plug-in 13
interfaces
    createExecutionDefinitions 42
    createLibraries 48
    createRequirements 36
    createTestPlan 30
    exportExecutionDefinitions 45
    exportLibraryStructure 50
    exportLibraryStructureWithoutSteps 51
    exportRequirements 37
    exportTestPlan 32
    getLibraryInfoByName 52
    Java interface 12
    reportData 27
    source control 11
    TMAttach 28
    updateExecutionDefinitions 45
    updateRequirements 38
    updateRequirementsByExtId 40
    updateTestPlan 33
issue tracking
    plug-ins 12
issue-tracking integration
    overview 12

J

Java interface 12
Jenkins
  executions, triggering 54
  triggering executions, Gradle 53

M

meta information
  third-party test type plug-ins 19

P

packaging
  third-party test type plug-ins 14
plug-ins
  cloud 21
  compilations 4
  deployment 4
  distribution 4
  issue tracking 12
  overview 4
  requirements 13
  requirements management 13
  source control 11
  species 4
predefined parameters
  passing to third-party test type plug-ins 14
process executor
  sample code 15

R

reportData
  example 27
  interface 27
requirements plug-in 13
requirements-management integration 13
REST API
  documentation 4

S

sample code
  third-party test type plug-ins 15
services exchange 27
sessions
  authenticating 25
SOAP
  envelopes 23
  stack 22
source control
  integration 11
  interface conventions 12
  interfaces 11
  plug-ins 11
species
  plug-ins 4
string-property meta information
  third-party test type plug-in 19
synchronization
  requirements 13

T

TeamCity
  executions, triggering 56
third-party test type plug-in
  API structure 15
  configuration XML file 18, 20
  custom icons 20
  file property meta information 19
  general property meta information 19
  implementation 14
  integration 13
  string-property meta information 19
third-party test type plug-ins
  meta information 19
  packaging 14
  passing predefined parameters 14
  sample code 15
TMAttach
  example 28
  interface 28
triggering executions
  CI servers 53
  Jenkins 54
  TeamCity 56

U

updateExecutionDefinitions
  examples 45
  interfaces 45
updateRequirements
  examples 38
  interfaces 38
updateRequirementsByExtId
  examples 40
  interfaces 40
updateTestPlan
  examples 33
  interfaces 33
uploading
  external results 4

V

video capturing
  indicating start 20
  indicating stop 20

W

Web Service
  cloud 21
  prerequisites 22
  requirements management 13
Web Service Demo Client 53
Web Service interface
  quick start 22
  tutorial 22
web services
  available 26

example use case 24
login, credentials 25
Web Services
documentation 4
overview 22
REST API 22