vCO Plug-in Development Tools

Index

vCO Annotations API

Developing plug-ins for a vCO always includes writing scripting/inventory objects in Java and describing them in vso.xml.

For example, a Java class from a plug-in may look like:

```
public class VLan extends BaseInventoryObject {

        private String name;

        public VLan(String id, String dn) {
                super(id, dn);
        }

        public void setName(String name) {
                this.name = name;
        }

        public String getName() {
                return name;
        }

        public BaseInventoryObject create(String range) {
                // some logic
        }
}
```

And the description of the object inside the vso.xml file may look like :

```
<module xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:nonamespaceschemalocation="http://www.vmware.com/support/orchestrator/plugin-
4-0.xsd" name="UCSM" version="${project.version}" build-number="${build.number}"
image="images/cisco_16x16.png">
         <finders>
                 <finder script-object="UCSMVLan" java-
class="com.vmware.o11n.vmo.plugin.ucsm.model.VLan" datasource="ucsm-datasource"
type="VLan" hidden="false">
                         <id accessor="getDn()" />
                         <properties>
                                 <property show-in-description="false" show-in-column="false"
bean-property="name" display-name="name" name="name" hidden="false"></property>
                                 <property show-in-description="false" show-in-column="false"
bean-property="dn" display-name="dn" name="dn" hidden="false"></property>
                         </properties>
                 </finder>
        </finders>
```

```
	<scripting-objects>
		<object strict="false" java-
class="com.vmware.o11n.vmo.plugin.ucsm.model.VLan" script-name="UCSMVLan">
			<description>vLAN</description>
			<constructors>
				<constructor><parameters><parameter name="id"
type="java.lang.String"><parameter name="dn" type="java.lang.String"></parameter></
parameter></parameters>
				 </constructor>
			</constructors>
			<attributes>
				<attribute show-in-api="true" read-only="false" return-
type="java.lang.String" java-name="name" script-name="name"></attribute>
				<attribute show-in-api="true" read-only="true" return-
type="java.lang.String" java-name="dn" script-name="dn"></attribute>
			</attributes>
			<methods>
				<method show-in-api="true" return-
type="com.vmware.o11n.vmo.plugin.ucsm.model.BaseInventoryObject" java-name="create"
script-name="create"><parameters><parameter name="range" type="java.lang.String"></
parameter>
					</parameters>
				</method>
			</methods>
		</object>
	</scripting-objects>
</module>
```

The main issues with this approach are that you have to remember always to change the vso.xml every time you change your Java code, that writing XML is error prone and that usually you have to describe your domain class in two places in vso.xml, i.e. as a Finder object and as a Scripting object.

On the other hand, centralized XML artifact provide a “blueprint” view of entire plug-in. Using XML you can configure object relation which are not backed by a Java object.

To keep the benefits of XML “blueprint” and simplify developers life, the SDK introduces a set of classes to enable annotation based configuration.

The main idea is to use Java annotations and use as much as possible from the code meta, e.g. class name, fields names and types, method names and parameters. But still generate the vso.xml file from this meta information.

To show an example, here is the previous domain object using annotations:

```
@VsoObject(description="vLAN")
@VsoFinder(name="VLan",idAccessor="getDn()")
public class VLan extends BaseInventoryObject {
	@VsoProperty(readOnly=false)
	private String name;

	@VsoConstructor
	public VLan(String id, String dn) {
		super(id, dn);
	}

	@VsoMethod
	public BaseInventoryObject create(String range) {
		// some logic
	}

	public void setName(String name) {
		this.name = name;
	}
	public String getName() {
		return name;
	}
}
```

And that is all you need to provide to generate the previous vso.xml file content.

Steps to enable the Annotation-based configuration

To enable Annotation based configuration you have to do the following steps:

Add the proper library dependencies to the build path of the plug-in

The vCO annotations are included inside the o11n-plugin-sdk-tools.jar file, but this library requires from other libraries to run. Using Ant that means to add the following libraries to the Ant build.classpath path variable:

```
<path id="build.classpath">
	...
	<pathelement location="${maven.repo.local}/o11n-sdkapi.jar"/>
	<pathelement location="${maven.repo.local}/o11n-model.jar"/>
	<pathelement location="${maven.repo.local}/o11n-util.jar"/>
	<pathelement location="${maven.repo.local}/o11n-plugin-sdk-tools.jar"/>
	<pathelement location="${maven.repo.local}/o11n-plugin-sdk-plugen.jar"/>
	<pathelement location="${maven.repo.local}/commons-cli-1.2.jar"/>
	<pathelement location="${maven.repo.local}/commons-collections-3.2.1.jar"/>
	<pathelement location="${maven.repo.local}/commons-lang-2.6.jar" />
	<pathelement location="${maven.repo.local}/commons-logging-1.0.4.jar" />
	<pathelement location="${maven.repo.local}/spring-asm-3.1.0.RELEASE.jar"/>
	<pathelement location="${maven.repo.local}/spring-beans-3.1.0.RELEASE.jar"/>
	<pathelement location="${maven.repo.local}/spring-context-3.1.0.RELEASE.jar"/>
	<pathelement location="${maven.repo.local}/spring-core-3.1.0.RELEASE.jar"/>
```

```
        <pathelement location="${maven.repo.local}/spring-oxm-3.1.0.RELEASE.jar"/>
</path>
```

Nevertheless the vso.xml file generation process occurs at build time of the plug-in package so those auxiliar libraries don't need to be packaged inside the plug-in.

Enable the vso.xml file generation

Using Ant, to enable the generation of the vso.xml file, you can add a new target (to be executed during the package target) that invokes the VsoGenerator class. This class receives as a parameter the plug-in's class that extends the ModuleBuilder class which, as you'll see later, is the responsible of composing the parts of the vso.xml file that are not defined inside the domain classes.

As example of this Ant target we have:

```
<target name="package" depends="compile,test" description="Package the application">
        <antcall target="generate-vso" />
        ...
</target>

<target name="generate-vso">
        <java fork="true" failonerror="yes"
classname="com.vmware.o11n.plugin.sdk.plugen.vso.VsoGenerator"
classpathref="build.classpath">
                <arg line="-name ${plugin.build.name}" />
                <arg line="-vsoDirectory ${maven.build.darDir}/VSO-INF" />
                <arg line="-moduleBuilder com.vmware.o11n.plugin.PowerShellModuleBuilder"
/>
        </java>
</target>
```

Annotating objects

To mark a domain class as a vCO scripting object you use the **@VsoObject** annotation. By default, the simple class name prefixed with you plugin (module's name attribute) name will be used as a scripting object name. You can override this behavior by specifying explicitly the name attribute within the annotation.

The current implementation does not export object's properties as a scripting attributes.
To enable given property as a scripting attribute, you annotate it with the **@VsoProperty** annotation. By default the field name is used as a scripting attribute name. You can use @VsoProperty, on a field or on a getter method level.

To generate the finder definition for a given scripting object, you annotate your class with the **@VsoFinder** annotation.

Fragments of the vso.xml file not connected with Java classes (Java-based configuration)

There are some fragments of the vso.xml file which are not connected with the Java classes (scripting objects) annotated previously, these are for example the “configuration”, ”installation”, “inventory” and ”finder-datasources” elements.

To address this you have to create your own ModuleBuilder class which extends from the com.vmware.o11n.plugin.sdk.module.ModuleBuilder class, and to implement its configure method. For example this piece of code will generate the same as the above vso.xml:

```
...
public class CiscoModuleBuilder extends ModuleBuilder {

    private static final String UCSM_DATASOURCE = "ucsm-datasource";

    @Override
    public void configure() {
        module("UCSM")
                .withDescription("Cisco UCSM Plug-in.")
                .withImage("images/cisco_16x16.png")
                .basePackages("com.vmware.o11n.vmo.plugin.ucsm.model");

        configuration(CiscoUCSMConfigurationAdaptor.class, "images/cisco_16x16.png")
                .configurationWar("o11nplugin-ucsm-config.war").validatable();

        installation(InstallationMode.BUILD)
                .action(ActionType.INSTALL_PACKAGE,
                        "packages/${artifactId}-package-${project.version}.package");

        inventory("System");

        finderDatasource(CiscoUCSMPluginAdaptor.class,
                UCSM_DATASOURCE).anonymousLogin(LoginMode.INTERNAL);
    }
}
```

And to enable it, as it’s described in the previous section “enable the vso.xml file generation” you have to use the Ant target generate-vso and configure it with your ModuleBuilder.

vCO Spring-based plug-in API

The Spring-based plug-in API allows plug-in developers to develop plug-ins following the Spring philosophy based on the IoC (Inversion of Control) idea and DI (Dependency Injection) pattern. This API simplifies the plug-in development process by saving developers to write a lot of boilerplate code in many cases and offering “for free” additional features that otherwise should be implemented in the same way for each plug-in. Some of these features are dependency injection, scripting object lifecycle management, basic resource management, easy way for a plug-in to refer to the environment, etc.

Why another abstraction layer?

- Current API is to low level and generic
- Current API has its quirks
- Current API is too restrictive, implementing advanced features forces you to go certain ways
- Current API does not speak the domain language of vCO
- Don’t need to reinvent the wheel in every plug-in

Why integration with Spring

- Just a class library is not enough
- Need lifecycle management
- Dependency injection promotes good design
- Spring has a lot of added value already

Basic configuration

To start developing a plug-in based on the Spring API there are some plug-in typical components that need to be implemented in the Spring way by extending some of the classes offered by the Spring-based API.

The plug-in adapter implementation

```
public final class DemoPluginAdaptor extends AbstractSpringPluginAdaptor {
	private static final String DEFAULT_CONFIG = "com/vmware/o11n/plugin/demo/
pluginConfig.xml";
	@Override
	protected ApplicationContext createApplicationContext(ApplicationContext
defaultParent) {
		ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext(
		new String[] { DEFAULT_CONFIG }, defaultParent);
		return applicationContext;
	}
}
```

The plug-in application context definition

This is defined inside the pluginConfig.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi=“http://
www.w3.org/2001/XMLSchema-instance”>
```

```
	<context:component-scan base-package="com.vmware.o11n.plugin.demo" scope-
resolver="com.vmware.o11n.plugin.sdk.spring.VsoAnnotationsScopeResolver">
	<context:include-filter type="annotation"
expression="ch.dunes.vso.sdk.annotation.VsoFinder"/>
	<context:include-filter type="annotation"
expression="ch.dunes.vso.sdk.annotation.VsoObject"/>
	</context:component-scan>
	<bean class="com.vmware.o11n.plugin.demo.DemoPluginFactory" id="pluginFactory"
autowire-candidate="false" scope="prototype" />
</beans>
```

The plug-in factory is just another spring-bean!

The plug-in factory implementation

```
public final class DemoPluginFactory extends AbstractSpringPluginFactory {
	@Override
	public Object find(InventoryRef ref) {
	}
	@Override
	public QueryResult findAll(String type, String query) {
	}
	@Override
	public List<?> findChildrenInRootRelation(String type, String relationName) {
	}
	@Override
	public List<?> findChildrenInRelation(InventoryRef parent, String relationName) {
	}
}
```

The plug-in factory is only responsible to find objects, that means that no longer you'll need to implement these methods:

- public void registerEventPublisher(String type, String id, IPluginEventPublisher pluginEventPublisher)
- public void unregisterEventPublisher(String type, String id, IPluginEventPublisher pluginEventPublisher)
- public Object find(String type, String id)
- public List<?> findRelation(String type, String id, String relationName)

New features

InvetoryRef Value Object

Type + Id = InvetoryRef

```
//A long time ago, in a SolarSystem far far away
public void addPolicyElement(String sdkType, String id, IPluginEventPublisher publisher) {
	String key = sdkType + "' / '" + id;
	log.info("Registering element to watch : '" + key + "'");
	policyElements.put(key, publisher);
}
```

Need to hook up to the IPluginFactory lifecycle?

- Let your scripting object implement PluginFactoryLifecycleAware
- The pluginFactory stores all objects it created and calls beforeFactoryUninstall just before the factory is destroyed
- Follow the well known Spring convention for *Aware interfaces
- Just in case you need to close sockets, release locks, close JDBC connections, JMS sessions, etc.

```
public interface PluginFactoryLifecycleAware {
	void beforeFactoryUninstall();
}
```

Need the pluginFactory in your scripting object?

- Let your scripting object implement PluginFactoryAware
- setPluginFactory() will be called for you by the Spring SDK

```
public interface PluginFactoryAware {
	void setPluginFactory(AbstractSpringPluginFactory factory);
	AbstractSpringPluginFactory getPluginFactory();
}
```

Need The current UserToken?

- Get it from you pluginFactory
- AbstractSpringPluginFactory stores it for you

```
public final UserToken getUserToken() {
	return userToken;
}
```

Need a IVSOFactoryClient?

- Get it from your AbstractSpringPluginFactory
- You already know how to get hold of it
- The client will be disconnected when the factory is destroyed

```
public final IVSOFactoryClient getVsoFactoryClient() throws RemoteException {
        if (userToken == null) {
                throw new IllegalStateException("cannot use IVSOFactoryClient without
UserToken");
        }
        if (vsoFactoryClient == null) {
                VSOFactoryClient local = new VSOFactoryClient(false);
                local.setNotificationEnabled(false);
                local.connect(vcoAddress + ":" + port, getUserToken());
                vsoFactoryClient = local;
        }
        return vsoFactoryClient;
}
```

What if you need the pluginFactory in your Service?

- Do you pass it in every call to your domain objects?
    - …when you need it in the implementation?
- Then use CurrentFactory: it resolves to the plugin factory that was used to create the scripting object
- Implemented using AOP

```
@Autowired
private CurrentFactory current;
private AmqpBrokerConfigPersister makePersister() throws Exception {
        AbstractSpringPluginFactory factory = current.get();
        return new AmqpBrokerConfigPersister(factory.getVsoFactoryClient());
}
```

Spring SDK brings to vCO plug-in development

- Dependency Injection
- Lightweight annotated model
- Ready solutions for everyday plug-in development
- Code that you need to write anyway is already written, tested and tried
- Offers basic resource management
- It tries to make the right things easy and the wrong things hard
- Compensates for certain features of the platfom
- Encapsulates the communication with the vCO server
- Offers easy way for a plug-in to refer to the environment
- Does not force a specific plug-in design, just promotes one
- Leaves plenty of room for extension
- Dependency Injection

Emerging patterns

- All scripting objects are stateless

- All state is in singleton-scoped beans
- A factory is used only to find objects
- Pattern for testing

vCO Workflow generation API

One possible way to extend the vCO functionality is by creating Workflows and Actions. Usually this is achieved using the swing Client. In certain cases it can be use full to create new Workflows/Actions based on some external definition. There are helper classes distributed with the SDK allowing basic workflow/action generation at runtime.

Generating Actions - ScriptActionGenerator

ScriptActionGenerator can be used for generating Actions. Main attributes of Action are input parameters, return type and script to be executed.

Steps to follow for generating actions :

```
// Cretate instance of ScriptmoduleBuilder and set required action name
ScriptModuleBuilder builder = new ScriptModuleBuilder().setName(actionName);

// Set type of returned type (Optional)
builder .setResultType("string");

// Add input params if any
builder.addParameter("SessionId", "string");

// set script to be executed
builder.setScript("var a = 1");

// Persist generated Action in vCO
builder.insert(categoryName, factory);
        }
```

Generating Workflows

Main attributes of vCO Workflows are input/output parameters, attributes and workflow item tasks. To have complete workflow we also need to define links between workflow items and optionally enhance the workflow presentation.

One way to achieve it is using WorkflowBuilderExt.

Steps to follow for generating workflows:

```
// Cretate instance of WorkflowBuilderExt
WorkflowBuilderExt wb = new WorkflowBuilderExt();
```

```
// Set generated workflow name
wb.setName(workflowName);

// Create wf output param
wb.addInParameter("someParam", “string”);

//Create needed workflow items and specify their location.
wb.createEndItem(“endItem”, 50, 100);;
ScriptingBoxItem item = wb.createScriptingBoxItem(“item1”, “var a = someParam”)
                .setLocation(50,50;
// In/Out parameters can be added if needed
item.addInParameter(“someParam”, “someParam”, "string");
wb.bindItemInParameter(“item1”, “someParam”,“someParam”);


//Connect items to create real workflow
wb.connectItem(“item1”, “endItem”;

//Set workflow start item
wb.setRootItemName(“item1”);

//Persist Workflow into vCO
wb.insertWorkflow(factory, targetFolder);
```

vCO SSL configuration API

Why SSL?

We can find on Internet lots of answers to this question, and perhaps one of the simplest and also most complete is this one from the Java Secure Socket Extension Reference Guide:

Transferring sensitive information over a network can be risky due to the following three issues:

- *You cannot always be sure that the entity with whom you are communicating is really who you think it is.*
- *Network data can be intercepted, so it is possible that it can be read by an unauthorized third party, sometimes known as an attacker.*
- *If an attacker can intercept the data, the attacker may be able to modify the data before sending it on to the receiver.*

SSL addresses each of these issues. It addresses the first issue by optionally allowing each of two communicating parties to ensure the identity of the other party in a process called authentication. Once the parties are authenticated, SSL provides an encrypted connection between the two parties for secure message transmission. Encrypting the communication between the two parties provides privacy and therefore addresses the second issue. The encryption algorithms used with SSL include a secure hash function, which is similar to a checksum. This ensures that data is not modified in transit. The secure hash function addresses the third issue of data integrity.

So, basically you can see that vCO supports and uses SSL to avoid the 3 described issues when establishing connections with 3rd party hosts or services.

The Java standard way

The Java standard way to use SSL is clear, and it's precisely described on the Java Secure Socket Extension (**JSSE**) Reference Guide mentioned before:

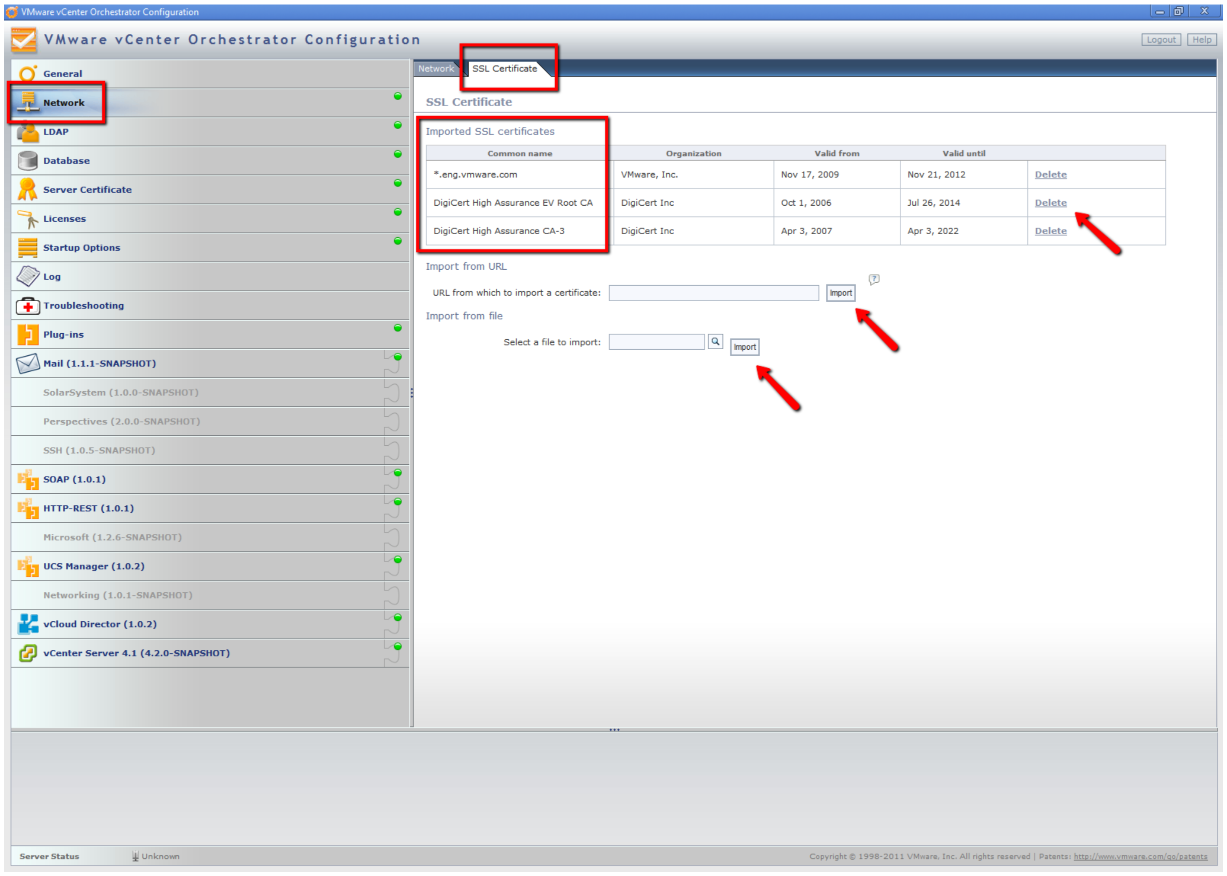
http://download.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html

The main idea, from a client point of view, is that you just have to **maintain your own keystore** (with the certificates of the servers and services that you trust), and to **implement and use** few **Java standard interfaces and classes** which, by having access to your keystore, will be able to establish SSL connections.

So, as a vCO plug-in developer, it's possible to follow this approach and to manage your own plug-in keystore and to implement your own classes to establish SSL connections between your plug-in and your configured hosts or services.

The vCO way

The vCO way follows the Java standard one, of course, and also simplifies it. The main is that you don't need to maintain your own keystore since vCO comes with its own **internal keystore** that it's accessible through the **Web Configurator**.


VMware vCenter Orchestrator Configuration
VMware vCenter Orchestrator Configuration
Logout
Help
General
Network
LDAP
Database
Server Certificate
Licenses
Startup Options
Log
Troubleshooting
Plug-ins
Mail (1.1.1-SNAPSHOT)
SolarSystem (1.0.0-SNAPSHOT)
Perspectives (2.0.0-SNAPSHOT)
SSH (1.0.5-SNAPSHOT)
SOAP (1.0.1)
HTTP-REST (1.0.1)
Microsoft (1.2.6-SNAPSHOT)
UCS Manager (1.0.2)
Networking (1.0.1-SNAPSHOT)
vCloud Director (1.0.2)
vCenter Server 4.1 (4.2.0-SNAPSHOT)
Network
SSL Certificate
Import
Import
Server Status
Unknown
Copyright © 1998-2011 VMware, Inc. All rights reserved | Patents: http://www.vmware.com/go/patents

SSL Certificate

Imported SSL certificates

Common name	Organization	Valid from	Valid until	
*.eng.vmware.com	VMware, Inc.	Nov 17, 2009	Nov 21, 2012	Delete
DigiCert High Assurance EV Root CA	DigiCert Inc	Oct 1, 2006	Jul 26, 2014	Delete
DigiCert High Assurance CA-3	DigiCert Inc	Apr 3, 2007	Apr 3, 2022	Delete

Import from URL

URL from which to import a certificate:

Import from file

Select a file to import:

This keystore contains the trusted certificates that will be used, when necessary, to establish SSL connections (mostly HTTPS) with 3rd party services. Usually these SSL connections will be established between plug-ins and remote hosts or services that they have been configured with.

From that web interfaces it’s possible to add and remove certificates from the vCO keystore, but remember that **after any update you must restart the vCO server for the changes to take effect**.

Some plug-ins, like the HTTP-REST plug-in or the SOAP plug-in, may offer to add new certificates to the vCO keystore through some specific workflows, but the standard way to manage trusted certificates is still the Web Configurator interface. You can find more information on the vCO documentation with the example of importing vCenter Server SSL certificates:

http://pubs.vmware.com/vsphere-50/index.jsp?topic=/com.vmware.vsphere.vco_install_config.doc_42/GUID39C552FA-0225-4D29-A4AC-7680252A5EC8.html

And, if you were just a simple vCO user, basically that would be everything you should know. If your SSL certificate is included inside the vCO keystore, any official plug-in will be able to use it to establish SSL connections with it.

But, as a vCO plug-in developer, **how is it possible to use the vCO keystore to establish SSL connections with the trusted certificates included there?**

The Plug-in SDK approach

To follow this approach you need to use the library **o11n-plugin-sdk-tools.jar**.
With this dependency you will have out-of-the-box some alternatives to easily configure SSL connections depending on what library or mechanism you want to use to create secure connections:

- Plain Java URLConnection class
- Plain Java SSLSocketFactory class
- Apache's HttpClient version 3
- Apache's HttpClient version 4

Plain Java URLConnection

If you want to use the Java standard URLConnection class you have 2 options:
To configure a couple of properties through the HttpsURLConnection class. In this case the SDK provides the proper implementations for the standard Java interfaces SSLSocketFactory and also HostnameVerifier. After that you can create underlying secure connections from any URL as usual.

```
import java.net.URLConnection;
import javax.net.ssl.HttpsURLConnection;
import com.vmware.o11n.plugin.sdk.ssl.factory.PluginSSLSocketFactory;
import com.vmware.o11n.plugin.sdk.ssl.verifier.PluginHostnameVerifier;

...

// Initialization
HttpsURLConnection.setDefaultSSLSocketFactory(PluginSSLSocketFactory.getDefault());
// Optionally
HttpsURLConnection.setDefaultHostnameVerifier(new PluginHostnameVerifier());

...

URLConnection conn = new URL("https://...").openConnection();
...
```

To create underlying secure connections directly from the PluginSSLSocketFactory class with the PluginHostnameVerifier class configured by default.

```
import java.net.URLConnection;
import com.vmware.o11n.plugin.sdk.ssl.factory.PluginSSLSocketFactory;

...

URLConnection conn = PluginSSLSocketFactory.getConnection("https://...");
...
```

Plain Java SSLSocketFactory

If you want to use directly the Java standard SSLSocketFactory class, the SDK provides the proper implementation to create underlying secure sockets..

```
import java.net.Socket;
import javax.net.ssl.SSLSocketFactory;
import com.vmware.o11n.plugin.sdk.ssl.factory.PluginSSLSocketFactory;

...

SSLSocketFactory factory = PluginSSLSocketFactory.getDefault();
Socket s = factory.createSocket(...);
...
```

Apache’s HttpClient 3

Again, with the Apache’s HttpClient version 3 you need to register (at some point and only once) the secure protocol that you want to use (e.g. “https”) with your own implementation of their ProtocolSocketFactory interface. In this case the SDK provides the vCO implementation of that interface for you and 2 different ways to register the protocol:

The long way: You need to register your protocol (e.g. “https”) manually.

```
import com.vmware.o11n.plugin.sdk.ssl.factory.HttpClient3PluginSSLSocketFactory;
import org.apache.commons.httpclient.protocol.Protocol;
import org.apache.commons.httpclient.protocol.ProtocolSocketFactory;

...

Protocol https = new Protocol("https", (ProtocolSocketFactory)
 HttpClient3PluginSSLSocketFactory.getDefault(), 443);
Protocol.registerProtocol("https", https);
```

The short way: The HttpClient3PluginSSLSocketFactory can register the “https” protocol for you. (For other protocols or specific ports you should use the long way.)

```
import com.vmware.o11n.plugin.sdk.ssl.factory.HttpClient3PluginSSLSocketFactory;

...

HttpClient3PluginSSLSocketFactory.registerHttpsProtocol();
```

Apache’s HttpClient 4

With the Apache's HttpClient version 4, you need to configure the ClientConnectionManager that you are going to use (e.g. ThreadSafeClientConnectionManager) with the secure scheme that you want (e.g. "https") and your own implementation of their SSLSocketFactory interface. In this case the SDK provides again the vCO implementation of that interface for you:

```
import com.vmware.o11n.plugin.sdk.ssl.factory.HttpClient4PluginSSLSocketFactory;
import org.apache.http.conn.scheme.Scheme;
import org.apache.http.conn.scheme.SchemeRegistry;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.impl.conn.SchemeRegistryFactory;
import org.apache.http.impl.conn.tsccm.ThreadSafeClientConnManager;

...

SchemeRegistry registry = SchemeRegistryFactory.createDefault();

Scheme https = new Scheme("https", 443, HttpClient4PluginSSLSocketFactory.getDefault());
registry.register(https);

ThreadSafeClientConnManager manager = new ThreadSafeClientConnManager(registry);
DefaultHttpClient client = new DefaultHttpClient(manager);
```

The HostValidator helper class

Independently of the SSL configuration itself the SDK provides also a helper class that can be used by the plug-ins to retrieve, from a HTTPS URL, the certificate (and its chain of certificates) used to establish the connection. In this way the plug-in can show back to the user the information about the certificate(s), in order the user to accept it or not, typically through a user interaction. And in case the user accepts, this helper class is able to install the certificate within the vCO keystore.

This class is the HostValidator and the features described are available via:

- The constructor HostValidator(url), the URL is the HTTPS URL.
- The method getCertificateInfo(), which returns a map with the properties available for that HTTPS connection and its cerfiticate(s).
- The method installCertificates(), which installs the certificate(s) within the vCO keystore.

Conclusion

Using SSL is pretty much a must within enterprise level software. vCO enables plug-in developers to create secure communication channels with 3rd party systems. The process is quite simple right now, but our goal is to provide a more straightforward and flexible way within the next releases and updates.