Technical white paper

Creating a Simple HP CDA Artifact Provider Plugin

Table of contents

What is an Artifact Provider? 3
A Look at a Simple Artifact Provider Plugin 3
    The Server File System Artifact Provider Plugin is a *Configurable* Plugin 3
    The Server File System Artifact Provider Plugin is *Testable* 4
    A Look at the Server File System Artifact Provider Plugin in Use 5
    Conclusion 7
The Server File System Artifact Provider Plugin Zip File 8
Creating a Simple HP CDA Plugin 8
    Designing in Configurability 8
    Designing in Testability 9
    Registering the Plugin Class as a Bean 9
    Building the Simple Plugin Jar File 10
    Adding Plugin Dependencies 11
    Signing the Plugin 12
    Uploading the Plugin to HP CDA 12
    Testing the Plugin's Functionality 14
Creating an Artifact Provider plugin 17
Working with Data Access Objects (DAOs) 17
Creating the Implementation 18
Completing the plugin IExternalArtifactProvider 20
For more information 22

Introduction

The HP CDA Plugin environment provides a mechanism to load and maintain connection and configuration envrionments within the HP CDA product. This enables the HP CDA administrator to load more than one version of a plugin to talk to different versions of remote servers that do not support the same API. It enables swapping out plugin implementations on the fly. A patch can be applied to a CDA plugin on the fly, and that patch will start to be used. It also allows a new version of a plugin to be loaded and tested before making the plugin live across the entire environment.

HP CDA provides several different types of plugins to allow HP CDA to interface with various artifact providers, platform provisioners, and deployers. HP CDA also offers an extendable Software Development Kit (SDK) that allows for the addition and extension of the plugin concept to other providers. This whitepaper explains how to design and implement a simple plugin for an artifact provider.

Part 1: An Artifact Provider Plugin Explained

What is an Artifact Provider?

An *artifact provider* is a source tool that stores source artifacts and configuration files that can be used in an application deployment. HP CDA collects artifacts used in deployments from an implementation of one or more artifact providers via artifact provider plugins. An artifact provider can be a disk, build system, web site or other entity that supports the downloading of artifacts. For example, Concurrent Versions System (CVS) and Subversion (SVN) are artifact providers that allow access to artifacts within a version repository. Once collected, these artifacts can be stored within HP CDA's Definitive Software Library (DSL) so that later deployments can utilize the same artifacts.

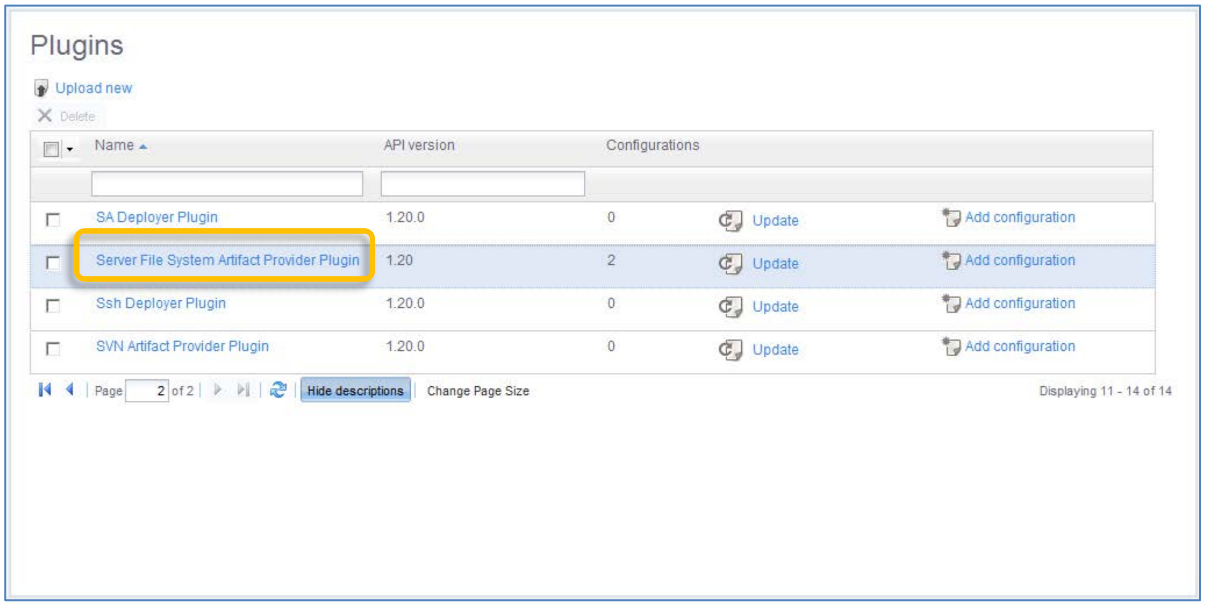
A Look at a Simple Artifact Provider Plugin

An artifact provider plugin's job is to adapt the browsing of the artifact provider, collect artifacts from the source, and capture the trace data so that the location of the source can later be identified. It accomplishes this by adapting an external artifact provider's APIs into HP CDA Interface Implementations. These implementations provide, for example, details on how to browse a repository and provide filters that can be established. They provide the interface to define the artifact that needs to be retrieved, along with the implementations of how to retrieve artifacts from a remote server. These interfaces are wrapped into a plugin implementation that can be loaded into HP CDA on the fly.

Plugins are uploaded into HP CDA as Jar files. Once loaded, they appear in the list of plugins in the HP CDA interface.

This section explains the features of a simple artifact provider plugin named "Server File System Artifact Provider Plugin." This plugin is shown in the Plugins screen of the HP CDA interface in Figure 1.

Figure 1. Plugin List

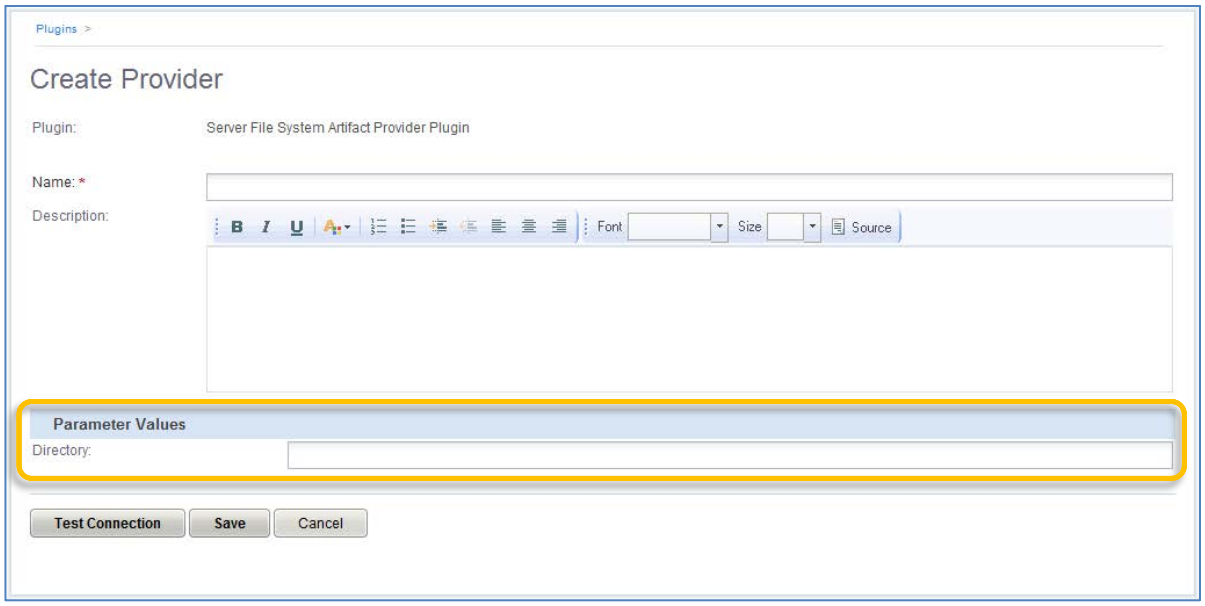


Name	API version	Configurations		
SA Deployer Plugin	1.20.0	0	Update	Add configuration
Server File System Artifact Provider Plugin	1.20	2	Update	Add configuration
Ssh Deployer Plugin	1.20.0	0	Update	Add configuration
SVN Artifact Provider Plugin	1.20.0	0	Update	Add configuration

The Server File System Artifact Provider Plugin is a *Configurable* Plugin

"Server File System Artifact Provider Plugin" is designed to browse the directory structure of a storage device for artifact files. The plugin has been designed as a configurable plugin which takes a single parameter; the root of the directory tree to be browsed. Figure 2 shows the Create Provider screen of the HP CDA user interface where the user enters this parameter.

Figure 2. Create Provider Screen



Create Provider

Plugin: Server File System Artifact Provider Plugin

Name: *

Description:

Parameter Values

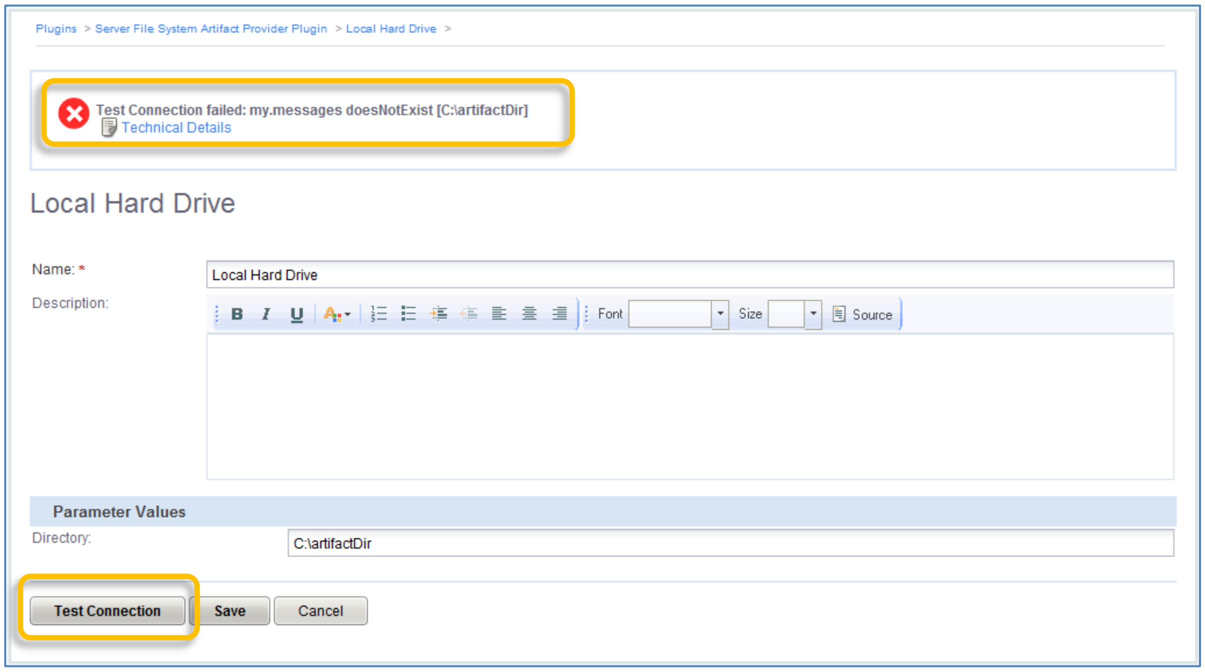
Directory:

The section entitled Designing in Configurability in the second part of this whitepaper explains how to create a configurable plugin and explains how to define a *parameter definition group*, which presents configuration parameters to the user and allows for user input.

The Server File System Artifact Provider Plugin is *Testable*

The example plugin has been designed as testable, so that the entered parameter can be checked for validity. When a plugin has been designed to be testable, a Test Connection button will be placed in the user interface, and the plugin designer can program the conditions to be tested and any messages to be returned in the event of a failure. Figure 3 shows an instantiation of the example plugin named "Local Hard Drive" where the connection test has failed, and Figure 4 shows the result of a successful connection test.

Figure 3. Test Connection Failure


Plugins > Server File System Artifact Provider Plugin > Local Hard Drive >
Technical Details
Test Connection
Save
Cancel

Test Connection failed: my.messages doesNotExist [C:\artifactDir]

Local Hard Drive

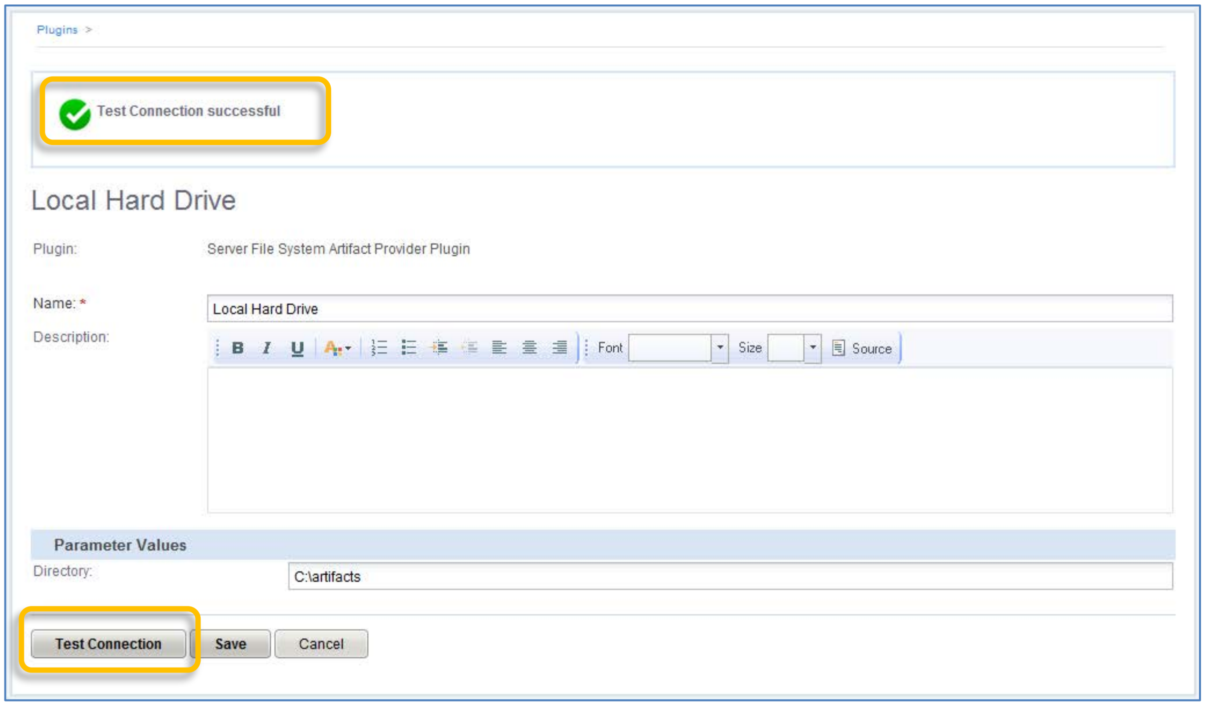
Name: * Local Hard Drive

Description:

Parameter Values

Directory: C:\artifactDir

Figure 4. Test Connection Success



Local Hard Drive

Plugin: Server File System Artifact Provider Plugin

Name: * Local Hard Drive

Description:

Parameter Values

Directory: C:\artifacts

In addition to the **Test Connection** button being added to the plugin screen, such plugins are added to the list of plugins that are sequentially tested by accessing the **Check Connections** link on the **Administration** tab of the HP CDA user interface.

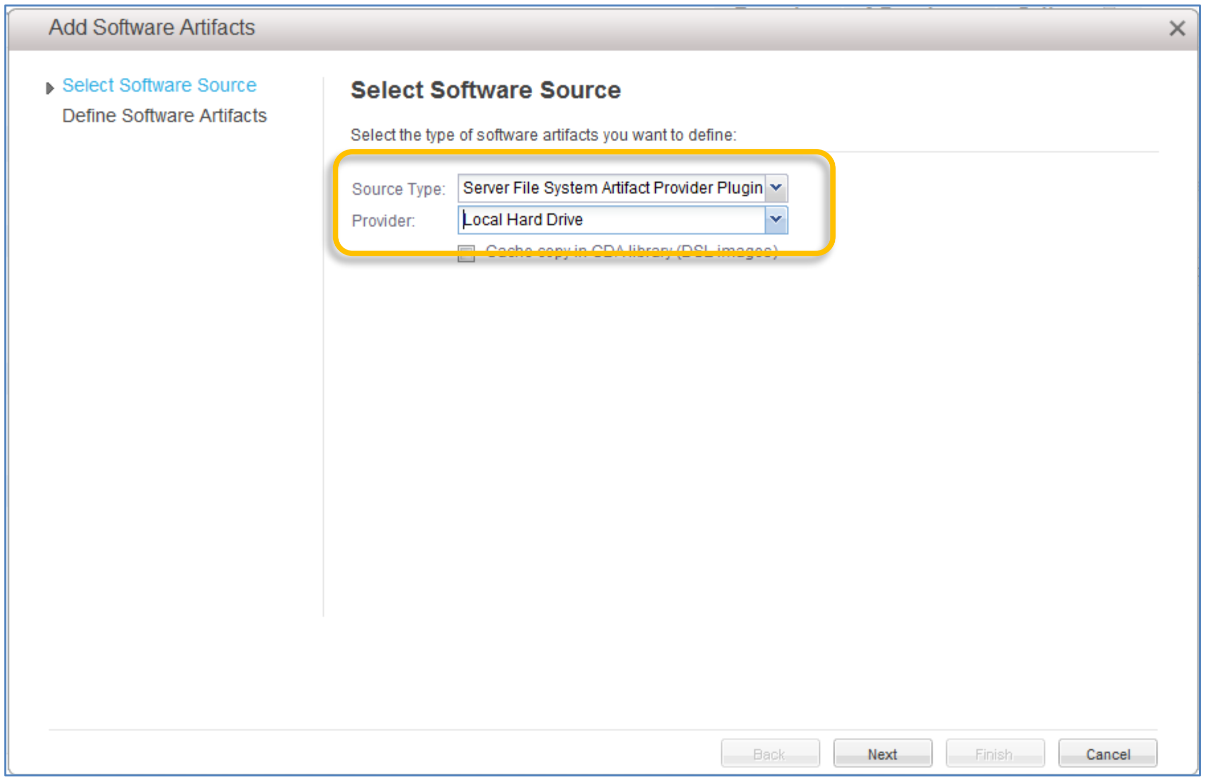
The section entitled Designing in Testability In the second part of this whitepaper explains how to design testability into a plugin and how to specify the conditions to be tested.

A Look at the Server File System Artifact Provider Plugin in Use

This section explores the instantiated artifact provider plugin named "Local Hard Drive" in use.

Artifacts are added to bundles on the **Software Artifacts** tab for a given version of an application in the HP CDA interface. On that tab, you click the **Add Software Artifacts** button to invoke the "Add Software Artifacts" wizard. In the first screen of the wizard, named "Select Software Source," you choose the type of plugin in the "Source Type" field and the desired plugin created from that type in the "Provider" field. Figure 5 shows the "Local Hard Drive" artifact provider plugin chosen in the screen.

Figure 5. Select Software Source Screen



Select Software Source

Select the type of software artifacts you want to define:

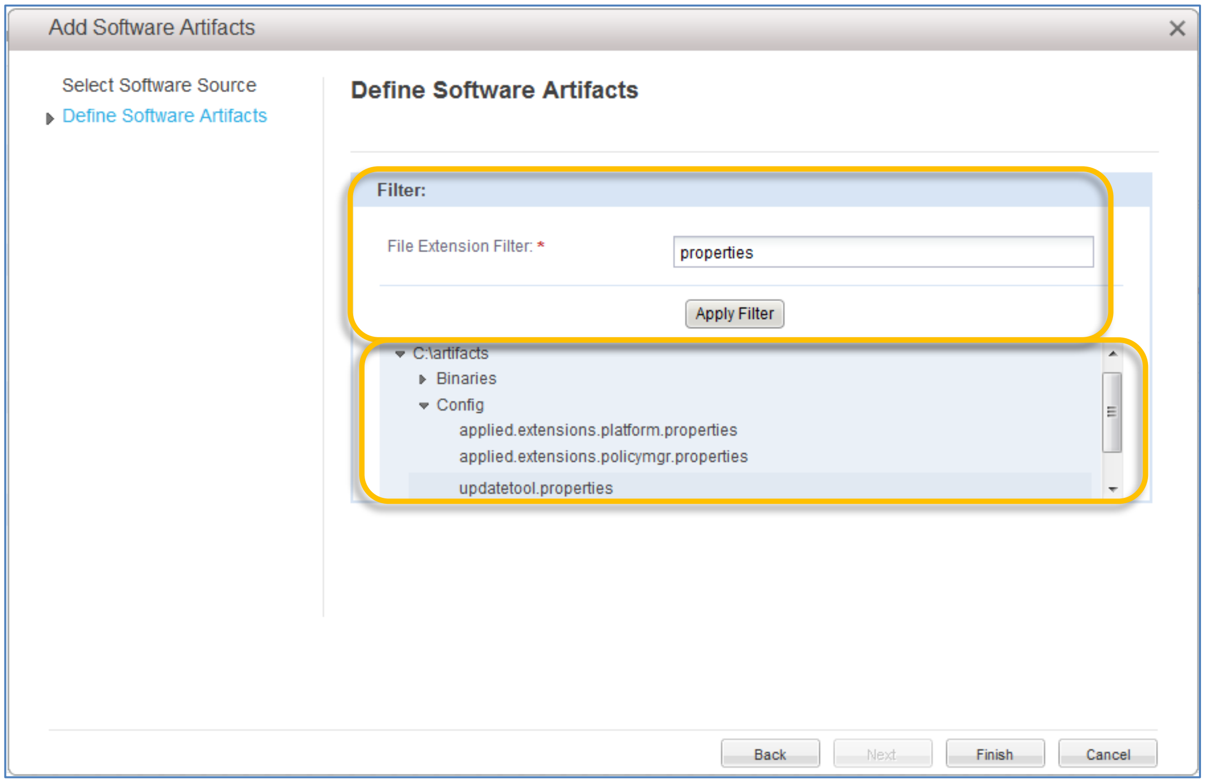
Source Type: Server File System Artifact Provider Plugin

Provider: Local Hard Drive

Cache copy in CDA library (DSL images)

The "Define Software Artifacts" screen, shown in Figure 6, appears upon clicking the **Next** button in the wizard.

Figure 6. Define Software Artifacts Screen



Define Software Artifacts

Filter:

File Extension Filter: * properties

```
C:\artifacts
  Binaries
  Config
    applied.extensions.platform.properties
    applied.extensions.policymgr.properties
    updatetool.properties
```

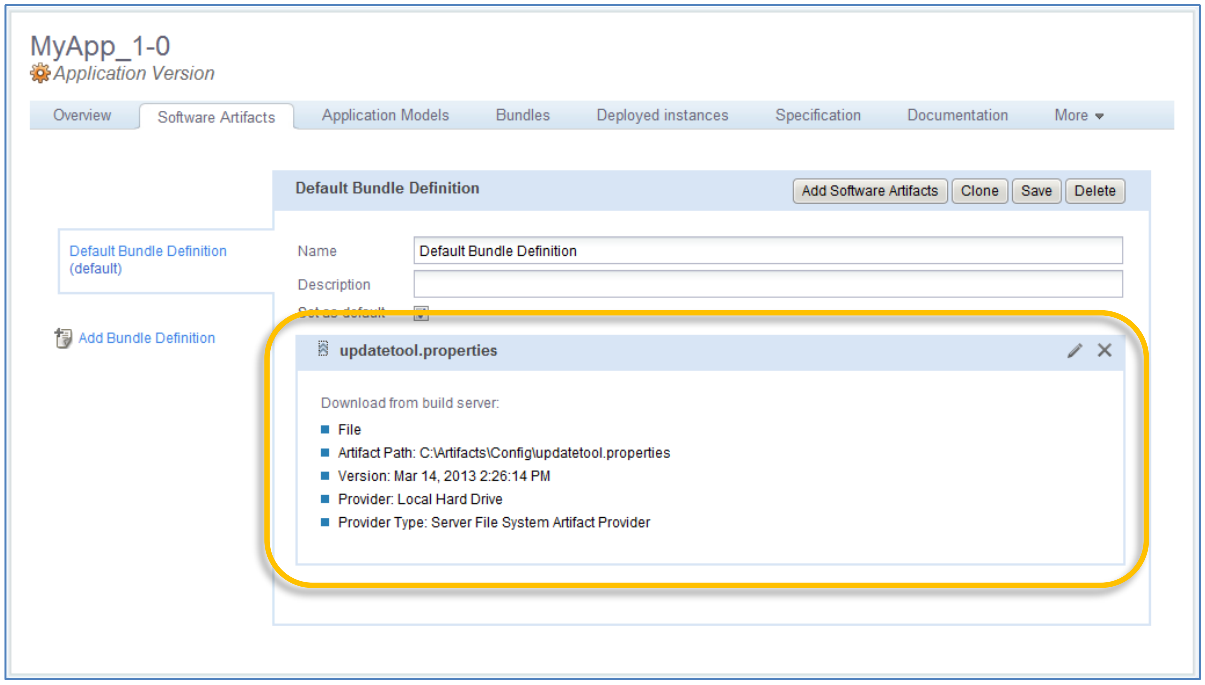
Figure 6 shows some more designed-in features of “Server File System Artifact Provider Plugin:”

- The plugin’s ability to browse a directory structure on the artifact provider for artifacts and display the directory tree. Note that the root of the directory tree is C:\artifacts, which was entered as a configurable parameter prior to instantiating the plugin.
- The plugin’s ability to filter and display only the artifacts in the directory tree having a certain file extension.

The section entitled Creating the Implementation in the second part of this whitepaper explains how to design browseability and filterability into a plugin.

When the **Finish** button is clicked, the plugin downloads and displays the metadata for the selected artifact. Figure 7 shows the metadata that has been returned for an artifact named “updatetool.properties.”

Figure 7. Metadata for Artifact “Updatetool.properties”



Name: Default Bundle Definition

Description:

Download from build server:

- File
- Artifact Path: C:\Artifacts\Config\updatetool.properties
- Version: Mar 14, 2013 2:26:14 PM
- Provider: Local Hard Drive
- Provider Type: Server File System Artifact Provider

Conclusion

In the first part of this whitepaper, we took a look at the following features of the simple plugin named “Server File System Artifact Provider Plugin:”

- *Configurability*: the ability to present parameters to a user; accept parameter entries from the user, and display the parameters in a configuration screen.
- *Testability*: the ability to test entered parameters for validity based on a set of designed in conditions, and return the test results to the interface.
- *Browseability*: the ability to obtain and display the directory structure of the artifact source.
- *Filterability*: the ability to display artifacts only of specific types, in our case, artifacts having a certain file extension.
- *Metadata Capture*: the ability to collect, store, and display metadata associated with a chosen artifact.

The second part of this whitepaper explains how to use HP CDA’s SDK to create a simple artifact provider plugin.

Part 2: Creating a Simple Artifact Provider Plugin

The Server File System Artifact Provider Plugin Zip File

A Zip file named “ServerFileSystemArtifactProviderPluginSample.zip,” which is available on HP Live Network (HPLN), contains the source code files and other files associated with the sample artifact provider plugin explained in this whitepaper. You can download a copy of the Zip fie from the **Downloads** tab at the following link:

https://hpln.hp.com/node/10233/contentfiles

After downloading the Zip file, save it locally and then extract it.

Creating a Simple HP CDA Plugin

Designing in Configurability

The plugin configuration screen in the HP CDA interface reads IArtifactProvider.getParameterDefinitionGroup() from the plugin main class, and presents those parameters in the configuration screen. The plugin developer needs only to define the parameter definition group so that the user will be prompted to enter the parameters. The HP CDA user interface will collect the parameters from the user and persist them.

When the plugin is used, the plugin manager will instantiate and configure the plugin. An instance of the class required will be retrieved from the plugin. If the instance created is a configurable plugin, then the plugin will be configured by calling IConfigurablePlugin.configure(). The class instance can inspect and use the configuration parameter values.

The following example code, from the artifact provider class, shows how parameter definitions are returned from the parameter definition group class, and the call to the configure() method to configure the plugin. Similar, more complete code is included in the “ServerFileSystemArtifactProvider.java” source code file:

```
public class ServerFileSystemArtifactProvider implements IParameterizedPlugin, IConfigurableProvider,
...{

        private final Logger LOG = LoggerFactory.getLogger(this.getClass());
        private IPluginConfiguration pluginConfiguration;
        .
        .
        public ServerFileSystemArtifactProvider() {
                .
                .
        }

        .
        .
        public ParameterDefinitionGroup getParameterDefinitionGroup() {
                return ServerFileSystemParameters.getParameterDefinitionGroup();
        }

        public void configure(IPluginConfiguration provider) {
                pluginConfiguration = provider;
                if (LOG.isDebugEnabled() && pluginConfiguration != null) {
                        ParameterValues vals = pluginConfiguration.getParameterValues();
                        LOG.debug(vals.toString());
                }
        }
```

The following class defines the parameter definition group. Similar code is included in the “ServerFileSystemParameters.java” source code file:

```
public class ServerFileSystemParameters {
        public static final ParameterDefinition HOST = new ParameterDefinition("my.host",
"Host","Name of host to connect.", false, true, true, new ParameterConstraintString("", 256, 1, null,
".*[^/]$"));
        public static final ParameterDefinition USER = new ParameterDefinition("my.username", "User",
"Remote User name on host", false, true, false, ParameterConstraintString.buildDefault());
```

```
        public static final ParameterDefinition PASSWORD = new ParameterDefinition("my.password", 
"Password", "Password for user on remote host", true, true, false, 
ParameterConstraintString.buildDefault());
        public static final ParameterDefinition DIRECTORY = new ParameterDefinition("my.basedir", 
"Directory", "Base directory to begin browsing from", false, true, false, 
ParameterConstraintString.buildDefault());


        private static ParameterDefinitionGroup connectionParameterGroup;

        static {
                connectionParameterGroup = new ParameterDefinitionGroup();

        connectionParameterGroup.addAllParameters(ParameterDefinitionUtils.getStaticParameterDefiniti
ons(ServerFileSystemParameters.class));
        }

        public static ParameterDefinitionGroup getParameterDefinitionGroup() {
                return connectionParameterGroup;
        }
}
```

Designing in Testability

If we define the plugin as ITestable, we can then implement a test method to validate the parameters and/or the connection. For the case where a server directory is entered, we can validate that the given directory exists. If there is a problem with the configuration, we will throw a message with the resource bundle and the message key. In this case, we pass an additional parameter for the directory.

The following example code, from the artifact provider class, shows how the “DIRECTORY” parameter is tested for various conditions. Similar code is included in the “ServerFileSystemArtifactProvider.java” source code file:

```
        public class ServerFileSystemArtifactProvider implements  … ITestable {
        …
        public void test(IPluginConfiguration config) throws LocalizableException {
                ParameterValues vals = config.getParameterValues();
                String dir = vals.get(ServerFileSystemParameters.DIRECTORY);

                File f = new File(dir);
                if (!f.exists()) {
                        throw new LocalizableException("my.messages", "doesNotExist", dir);
                }
                if (!f.isDirectory()) {
                        throw new LocalizableException("my.messages", "notDirectory", dir);
                }
                if (!f.canRead()) {
                        throw new LocalizableException("my.messages", "notReadable", dir);
                }
        }
```

Registering the Plugin Class as a Bean

Once you have completed the simple plugin, you must register it as a Bean. This is done through a file ending in Context.xml that is placed in the META-INF directory. The following is the contents of the file named “myContext.xml” which is included in the sample.zip file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd
http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee.xsd
```

```
http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang.xsd
http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd
http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd
http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-
context-3.0.xsd">

    <bean class="com.hp.samples.artifactprovider.ServerFileSystemArtifactProvider"
name="serverFileSystemProvider.artifactprovider"/>

</beans>
```

Building the Simple Plugin Jar File

We now have a simple plugin that presents a configuration to a user and allows the parameters to be tested. We need to now configure the build for the plugin. This involves setting the appropriate data into the manifest. In the maven project, set the title, ID, and main class appropriately for your plugin.

Note
Each unique version of a plugin should have its own ID. UUID generators are available on the Web and can be used to generate such IDs.

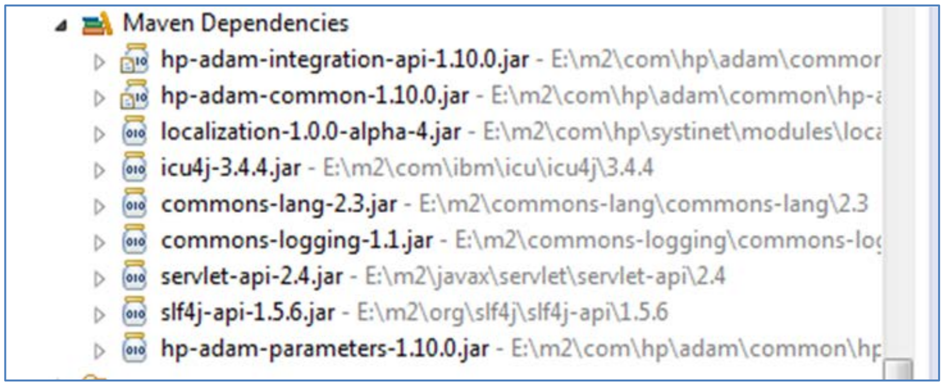
The following is the contents of the file named “pom.xml” which is included in the sample.zip file:

```
        <project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
        <modelVersion>4.0.0</modelVersion>

        <groupId>com.hp.cda.samples</groupId>
        <version>1.0.0-SNAPSHOT</version>
        <artifactId>hp-serverfs-artifactprovider</artifactId>

        <name>HP Sample CDA Artifact Provider</name>
        <description>HP Sample CDA Artifact Provider</description>
        <build>
                <plugins>
                <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-jar-plugin</artifactId>
                        <configuration>
                        <archive>
                                <manifest>
                                        <addClasspath>true</addClasspath>
                                        <useUniqueVersions>false</useUniqueVersions>
                                </manifest>
                                        <manifestSections>
                                        <manifestSection>
                                                <name>com.hp.adam.plugin</name>
                                                <manifestEntries>
                                                <apiVersion>1.20.0</apiVersion>
                                                <!-- generate a unique ID to be bound for the
plugin. -->
                                                <id>826bb6a0-f436-40ba-827d-b7d9e3c3e16e</id>
                                                <title>Server File System Artifact Provider
                                                 Plugin</title>
                                                <providedTypes>IArtifactProvider,
IExternalArtifactCatalog</providedTypes>

        <mainClass>com.hp.samples.artifactprovider.ServerFileSystemArtifactProvider</mainClass>
                                                        </manifestEntries>
                                                </manifestSection>
                                        </manifestSections>
                                </archive>
                        </configuration>
                </plugin>
        </plugins>
        </build>
        <dependencies>
        <dependency>
```

```
                <groupId>com.hp.adam.common</groupId>
                <artifactId>hp-adam-integration-api</artifactId>
                <version>1.10.0</version>
                <scope>provided</scope>
        </dependency>
        <dependency>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-jcl</artifactId>
                <version>1.7.2</version>
                <scope>test</scope>
        </dependency>
        <dependency>
                <groupId>com.hp.adam.common</groupId>
                <artifactId>hp-adam-integration-impl</artifactId>
                <version>1.10.0</version>
                <scope>provided</scope>
        </dependency>
        <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>4.11</version>
                <scope>test</scope>
        </dependency>
        </dependencies>
</project>
```

Now that we have our pom file configured, we need to get the appropriate files into our maven repository for the build to work.

Figure 8. Maven Dependencies



```
Maven Dependencies
  hp-adam-integration-api-1.10.0.jar - E:\m2\com\hp\adam\commor
  hp-adam-common-1.10.0.jar - E:\m2\com\hp\adam\common\hp-a
  localization-1.0.0-alpha-4.jar - E:\m2\com\hp\systinet\modules\loca
  icu4j-3.4.4.jar - E:\m2\com\ibm\icu\icu4j\3.4.4
  commons-lang-2.3.jar - E:\m2\commons-lang\commons-lang\2.3
  commons-logging-1.1.jar - E:\m2\commons-logging\commons-lo
  servlet-api-2.4.jar - E:\m2\javax\servlet\servlet-api\2.4
  slf4j-api-1.5.6.jar - E:\m2\org\slf4j\slf4j-api\1.5.6
  hp-adam-parameters-1.10.0.jar - E:\m2\com\hp\adam\common\hp
```

From your HP CDA installation, you can add the hp-adam-integration-api-xxx.jar, hp-adam-common-xxx.jar, hp-adam-parameters-xxx.jar, and localization-xxx.jar from the <cda_installation_dir>/lib directory into your Maven repository. Other jar files can be loaded but may be sourced from external sources.

Now we are ready to build the simple plugin and test it by running `mvn install` to build and create the plugin jar file. Once the jar file is created we can load the plugin into HP CDA.

Adding Plugin Dependencies

HP CDA plugins also allow for the inclusion of dependent Jar files within the plugin. Any Jar files located in the /lib directory of the plugin will be added to the classpath at runtime.

Note
To keep the plugin small, do not load the Jar files already included in the HP CDA Ear file. These files will already be included in the classpath.

To add these Jar files, modify the maven pom.xml file with the following plugin configuration in the build/Plugins section, to cause the dependencies to be packaged into the Jar files /lib directory. Be sure to update the includeArtifacts entry to include the artifactIds of the Jar files to be included in the /lib directory.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <version>2.3</version>
    <executions>
       <execution>
            <id>copy</id>
            <phase>process-resources</phase>
            <goals>
                <<goal>copy-dependencies</goal>
            </goals>
            <configuration>
                <includeArtifactIds>myartifact,myartifact2</includeArtifactIds>
                <outputDirectory>${project.build.directory}/classes/lib</outputDirectory>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Signing the Plugin

For security, signing the plugin Jar file is recommended. See the maven-jarsigner-plugin for configuration details.

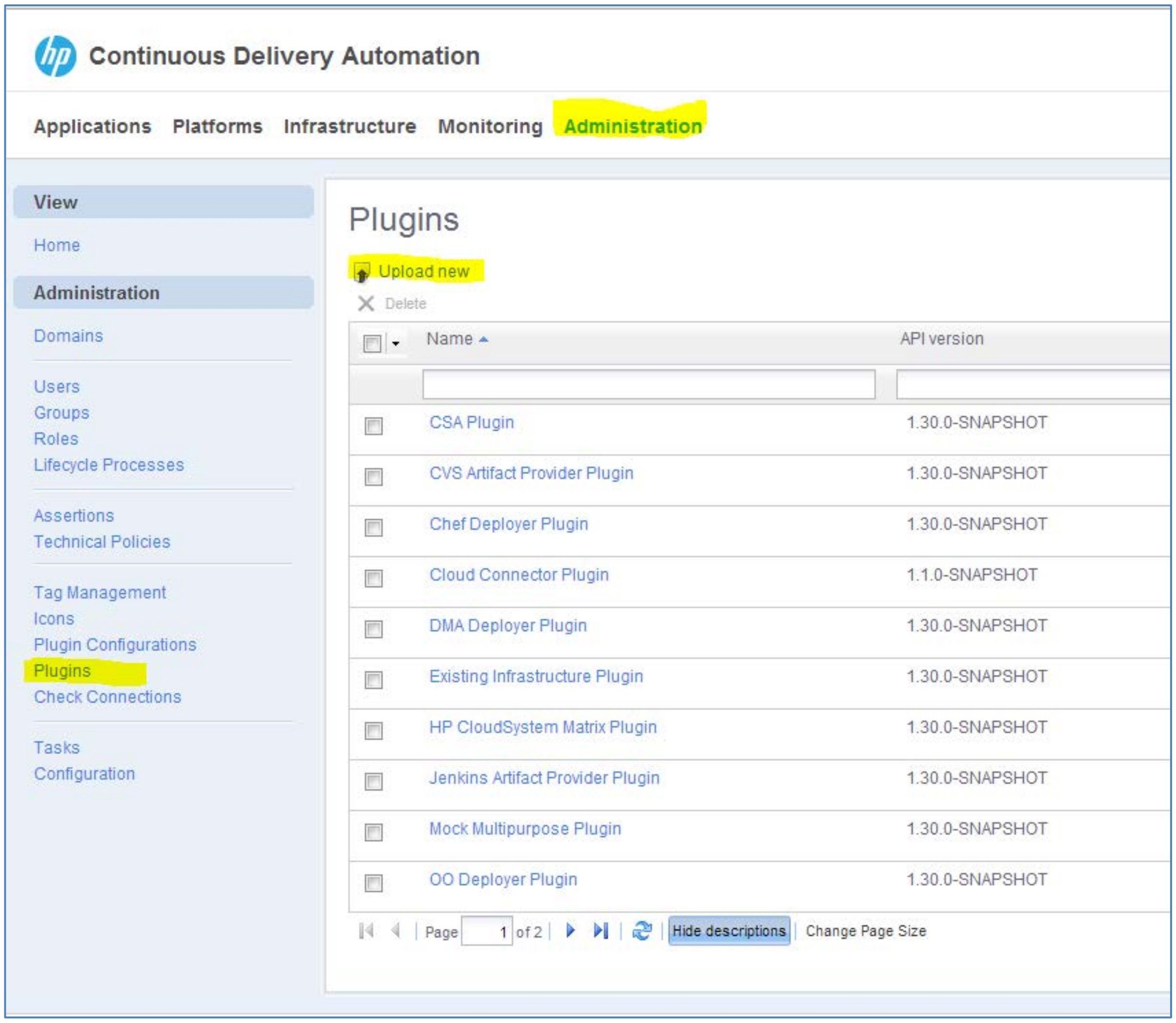
Uploading the Plugin to HP CDA

Note
This section explains how to initially upload and install a plugin. Once installed, a different procedure must be followed to update an existing plugin. Refer to the HP CDA online help for details.

Now that the jar file for the plugin has been created, we can upload and install it in HP CDA. Copy the plugin jar file to the machine from which you will access the HP CDA interface, and then perform the following steps:

1. In the HP CDA user interface, click the Administration tab, and then click Plugins in the left hand navigation pane. This will present a list of plugins that have already been loaded.

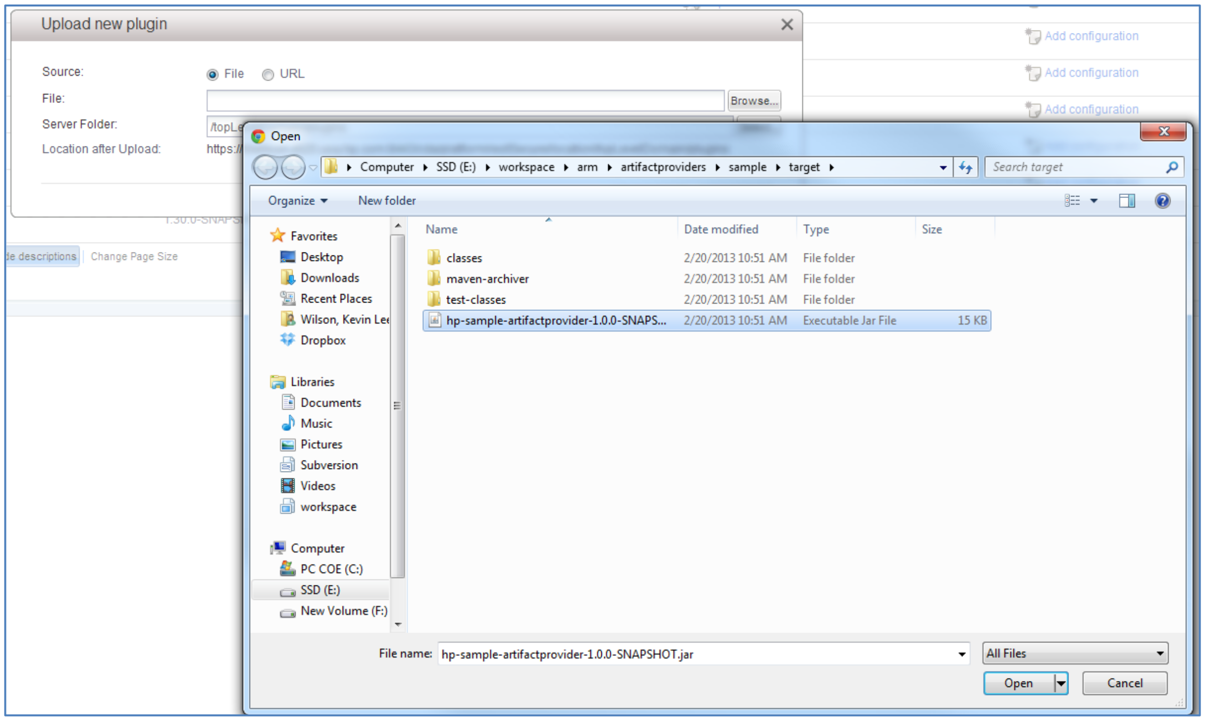
Figure 9. List of Plugins



Name	API version
CSA Plugin	1.30.0-SNAPSHOT
CVS Artifact Provider Plugin	1.30.0-SNAPSHOT
Chef Deployer Plugin	1.30.0-SNAPSHOT
Cloud Connector Plugin	1.1.0-SNAPSHOT
DMA Deployer Plugin	1.30.0-SNAPSHOT
Existing Infrastructure Plugin	1.30.0-SNAPSHOT
HP CloudSystem Matrix Plugin	1.30.0-SNAPSHOT
Jenkins Artifact Provider Plugin	1.30.0-SNAPSHOT
Mock Multipurpose Plugin	1.30.0-SNAPSHOT
OO Deployer Plugin	1.30.0-SNAPSHOT

2. Click the **Upload new** button at the top of the list to begin the jar file upload process. When the **Upload new** button is clicked, the "Upload new plugin" dialog box will appear.

Figure 10. Uploading the Plugin Jar File



Name	Date modified	Type	Size
classes	2/20/2013 10:51 AM	File folder	
maven-archiver	2/20/2013 10:51 AM	File folder	
test-classes	2/20/2013 10:51 AM	File folder	
hp-sample-artifactprovider-1.0.0-SNAPS...	2/20/2013 10:51 AM	Executable Jar File	15 KB

File name: hp-sample-artifactprovider-1.0.0-SNAPSHOT.jar

3. Click the Browse button in the “Upload new plugin” dialog box to open a file browser, select the jar file in the file browser and click Open, and then click the Upload button in the “Upload new plugin” dialog box.

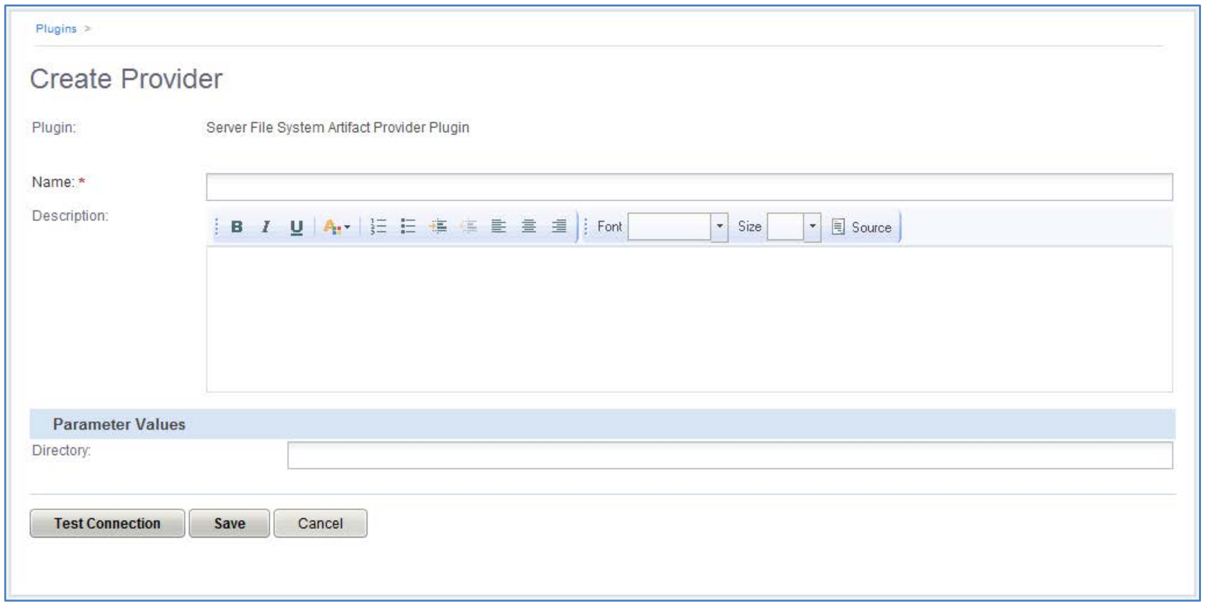
If the plugin uploads and installs correctly, it will appear in the list of plugins in the HP CDA interface (see Figure 9). If a problem was encountered during the upload/install process, an error message might appear in the display and logging will be available in the log file. Some typical issues that might cause problems include the following:

– Incorrect main class definition is specified in the Maven pom.xml file.
– The context definition is incorrect.
– The UUID is not updated in pom.xml file.

Testing the Plugin’s Functionality

Now that the plugin jar file has been uploaded to HP CDA and installed, we can test its configuration parameter input and test capabilities by browsing to the plugin by name in the list of plugins, and clicking the Add Configuration button. A screen similar to that shown in Figure 11 should appear.

Figure 11. Plugin Configuration Screen


Plugins >

Create Provider

Plugin: Server File System Artifact Provider Plugin

Name: *

Description:

Parameter Values

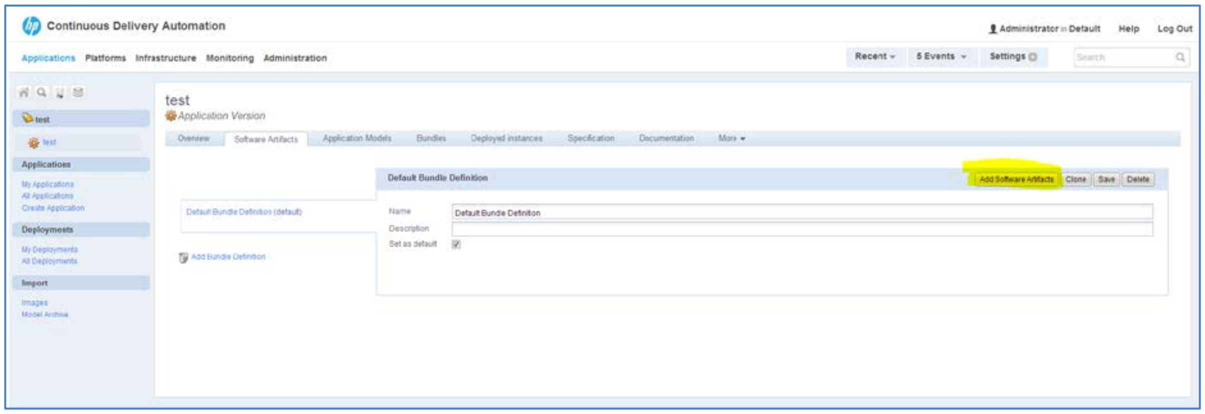
Directory:

Enter a name for the provider, enter an appropriate value into the "Parameter Values" section of the screen and then click the **Test Connection** button. Note that if you enter a directory in the "Directory" field that does not exist, then the test will fail based on the previous coding. Save the configuration by clicking the **Save** button and we will continue our exploration of the artifact provider.

Now click the **Applications** tab in the HP CDA user interface, and create an application that you can use to test the plugin, if one does not already exist. Once you have the test application, open it, and in the "Application Version" screen, click the **Software Artifacts** tab.

Click the **Add Software Artifacts** button (see Figure 12) to open the "Add Software Artifacts" wizard, and then choose the plugin you have configured in the first screen of the wizard (see Figure 13).

Figure 12. Adding a Software Artifact

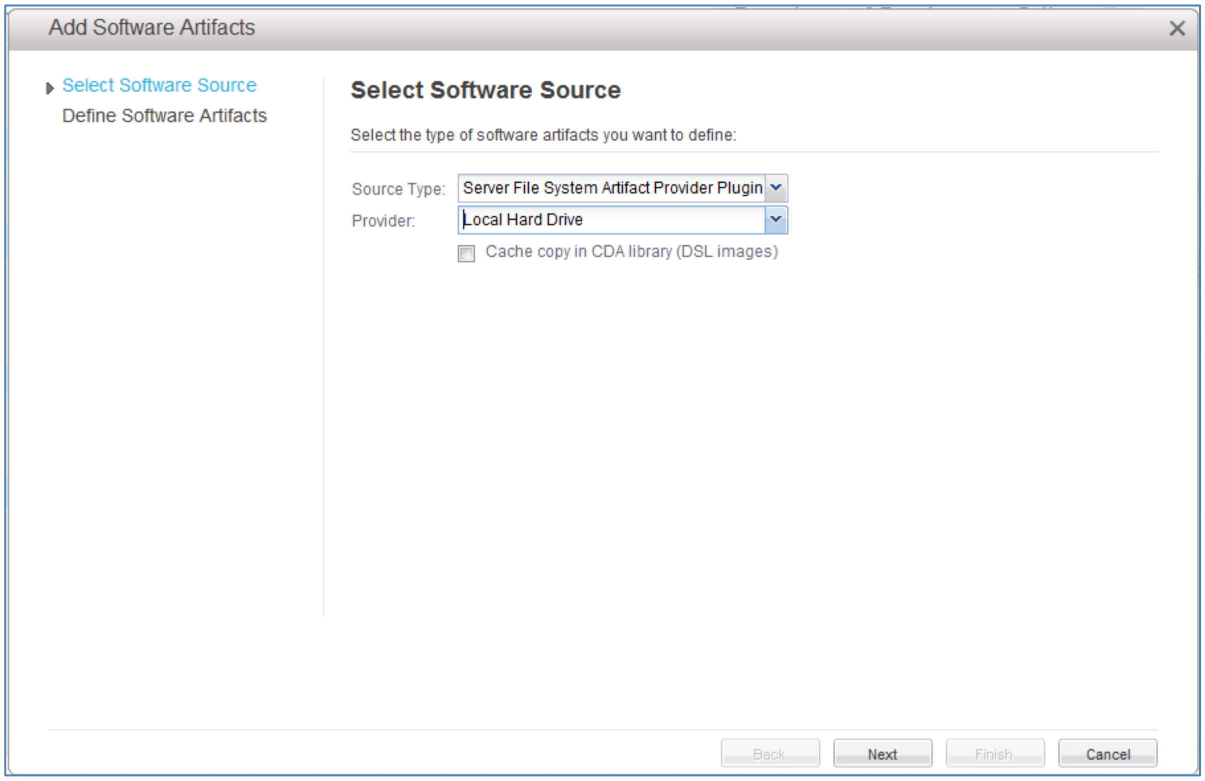


Name: Default Bundle Definiton

Description:

Set as default: ✓

Figure 13. Choosing the Software ArtifactSource



Select Software Source

Select the type of software artifacts you want to define:

Source Type: Server File System Artifact Provider Plugin

Provider: Local Hard Drive

Cache copy in CDA library (DSL images)

Choosing the plugin configuration as demonstrated above will succeed because that part does not yet use the API, but further testing will fail because we have not yet completed the entire API. Once you navigate to the next screen of the wizard, a failure will occur when the roots are obtained from the plugin. If you receive an error, simply use the refresh button on the browser to reload the page.

In this section we were able to create, build, load, and test a simple plugin that gathers basic configuration information. These tasks can be attributed to any HP CDA plugin development.

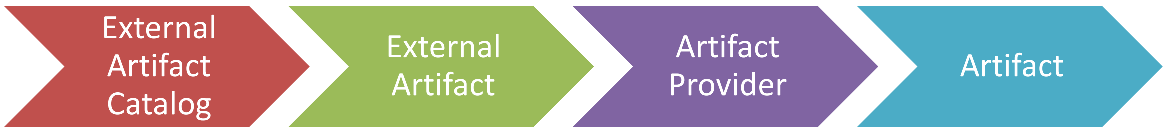
In the background, the HP CDA plugin environment is at work keeping the plugin jar file in its own classpath so that many different versions of the same plugin can be loaded into HP CDA simultaneously. Each call to a plugin will setup the classpath appropriately to run the plugin class independently from the rest of the system. This also allows for hot swapping and loading of plugins during runtime. You do not need to start/stop the server to reload a plugin.

Creating an Artifact Provider plugin

You have created, configured, and partially tested a simple plugin. In this section, you will learn how to extend that plugin into an artifact provider.

The plugin interface for artifact providers is in two phases:

- The first phase includes browsing artifacts with the implementation of IExternalArtifactCatalog. This results in metadata regarding an artifact's source location to be collected and persisted in the application model.
- The second phase includes using the metadata collected to later obtain the actual bits the metadata is referencing. This gives HP CDA the ability to go back out to the source location and check for new bits.



We will first take a look at the IExternalArtifactCatalog class and add this to our plugin. Through this section we will look at the various methods to be implemented and walk through an example.

The browsing session of IExternalArtifactCatalog is session based. The session with the catalog must be opened and closed. The caller of the session maintains the session and returns it in the calls. The catalog however, is responsible for creating the session object when a session is opened with the catalog. Thus the session may be implemented by an http session or some other browsing object that keeps track of the user context.

The following example code is used to create and close a session object. Similar code is included in the "ServerFileSystemArtifactProvider.java" source code file:

```
public Object createSession() throws ExternalArtifactException {
        String session = "" + (nextCtx++);
        sessionMap.put(session, new ServerFileSystemContext());
        return session;
}

public boolean closeSession(Object session) {
        sessionMap.remove(session);
        return true;
}
```

The next piece to consider includes the filters or additional data the user can provide to narrow the artifact search results. These parameter types are returned by the following API and are present in the interface before the tree is browsed.

```
public ParameterDefinitionGroup getFilters() {
        return ServerFileSystemFilters.getParameterDefinitionGroup();
}
```

The ServerFileSystemFilters are created identically to theServerFileSystemParameters. For the sample application, we will use a simple regular expression matching filter. The code for ServerFileSystemFilters is included in the "ServerFileSystemFilters.java" source code file.

Working with Data Access Objects (DAOs)

We will now consider the data storage objects that are used. The system uses a Data Access Object (DAO) to create and persist the data. The implementation at runtime will supply the appropriate DAO implementations, and create the appropriate objects. However, for testing we do not want the overhead of the entire interface, hence this can be replaced with a class to provide mock objects for testing.

For the Artifact Provider the IExternalArtifactFileSetDescriptorDao and IExternalArtifactDao are the two interfaces used to create and store objects in the database. Methods in the IExternalArtifactCatalog and IExternalArtifactProvider create and return these instances.

To retrieve the instance of the DAO object at runtime, you should reference the PluginDaoProxy class from the hp-adam-integration-impl.jar file. Thus, this jar will need to be added from your HP CDA installation to the project as a provided jar.

The implementations of the DAO can be constructed and passed in for doing quick unit testing without the need for the entire DB/JBoss environments.

Note
Spring Autowiring is not supported in the plugin architecture.

```
IExternalArtifactDao externalArtifactDao = PluginDaoProxy.getInstance().getExternalArtifactDao();
IExternalArtifactFileSetDescriptorDao externalArtifactFileSetDescriptorDao =
PluginDaoProxy.getInstance().getExternalArtifactFileSetDescriptorDao();
```

Plugin implementations only use the create() method in the DAO interfaces:

```
IExternalArtifact externalArtifact = dao.create();
```

The caller of the plugin will be responsible for calling the update() method to persist the data object.

Note
We keep the business logic out of this layer of the code. This is meant only to act as an interface to the persistence of the object.

Creating the Implementation

In this section, we will begin the implementation of the methods to provide the artifacts. We will explore each one in succession.

We will start with the getRootNodes() method. This method establishes the root context of the tree to be browsed for artifacts. The IExternalArtifactTree implementation can be a local implementation and is not a persisted artifact, so DAO creation is not required. In this method we are also passed the filtering and we simply store the active filter in the context to be used when loading.

```
        public List<IExternalArtifactCatalogTree> getRootNodes(Object session,
                        ParameterValues filters) throws ExternalArtifactException {

                ArrayList<IExternalArtifactCatalogTree>  roots = new
ArrayList<IExternalArtifactCatalogTree>();
                IExternalArtifactCatalogTree root = this.createCatalogNode(new File(getDirectory()),
filters, true);
                root.setName(getDirectory());
                roots.add(root);
                return roots ;
        }
```

The next method to discuss is the load() implementation. Here we implement the business logic to load the node. It is called when the node is expanded, allowing the next level of the tree to be explored. This can be called with a reload flag which should be observed.

In our implementation, we are simply going to use the filter to build a FilenameFilter object to control the list. We will also browse the local OS for sub directories and files to be returned. Once we have built the list of artifacts and trees, we will apply those to the node being loaded with node.setSubItem(). Note the file filter was setup to observe our filter property defined on the tree.

```
        public boolean load(Object session, IExternalArtifactCatalogTree node,
                        boolean reload) throws ExternalArtifactException {

                // If the node was loaded then exit.
                if (node.isLoaded() && !reload) {
                        return false;
                }

                final String ext;
                ParameterValues filters = node.getExternalArtifactFileSetDescriptor().getFilters();
                if (filters != null && filters.contains(ServerFileSystemFilters.EXT)) {
                        ext = filters.get(ServerFileSystemFilters.EXT);
                } else {
                        ext = "";
                }

                FilenameFilter fFilter = new FilenameFilter() {
                        public boolean accept(File dir, String name) {
                                if (name.endsWith(ext)) {
                                        return true;
                                }
                                File f = new File (dir, name);
                                if (f.isDirectory()) {
                                        return true;
                                }
                                return false;
                        }
                };

                List<IExternalArtifactCatalogTree> subfolders = new
ArrayList<IExternalArtifactCatalogTree>();
                List<IExternalArtifact> artifacts = new ArrayList<IExternalArtifact>();

                for (File file : new File(node.getPath()).listFiles(fFilter)) {
                        if (file.isDirectory() && !file.isHidden()) {
                                subfolders.add(createCatalogNode(file, filters, true));
                        } else {
                                if (!file.isHidden()) {
                                        IExternalArtifact artifact = createArtifact(file, filters);
                                        if (artifact != null) {
                                                artifacts.add(artifact);
                                        }
                                }
                        }
                }
                node.setSubItems(subfolders, artifacts);

                return true;
        }
```

We'll now finish up the implementation by creating the private implementations for createArtifact() and createCatalogNode(). When creating the artifact, we will preserve the filter.

Note
It is required that ServerFileSystemTreeNode implement both getName() and getId() appropriately, so that nodes can be displayed in the tree. If this is missed, an exception will be reported. ServerFileSystemTreeNode is included in the "ServerFileSystemTreeNode.java" source code file.

```
        private IExternalArtifact createArtifact(File file, ParameterValues filters) {
                LOG.debug("Adding File: " + file.getName());
                try {
                        IExternalArtifact artifact = artifactDao.create();
                        artifact.setName(file.getName());
                        artifact.setDescription(file.getCanonicalPath());

                        String path = file.getCanonicalPath();
                        if (path.startsWith(getDirectory())) {
                                path = path.substring(getDirectory().length());
                        }

                        artifact.setRelativeUri(path);
                        artifact.setVersion(toVersion(file));
                        artifact.setFilters(filters);
                        return artifact;
                } catch (IOException ioe) {
                        LOG.error("Could not add artifact, trouble calculating the Cannonical path of 
the artifact:" + file.getPath()+ File.separator + file.getName());
                }

                return null;
        }

        private IExternalArtifactCatalogTree createCatalogNode(File file, ParameterValues filters, 
boolean selectable) {
                LOG.debug("Adding directory:" + file.getName());

                ServerFileSystemTreeNode treeNode = new ServerFileSystemTreeNode (file);
                treeNode.setSelectable(true);
                return treeNode;
        }
```

The relative URI should be stored in a generic form with separators translated to "/". These should be translated on the way in to create the artifact. On the way out, the values should be translated for the appropriate OS on the usage of the URI.

Also note that if a tree node is selectable, it must also supply the IExternalFileSetDescriptor.

Note
The search() method, although defined in the API, is not yet utilized.

We are now half way through creating an operational artifact provider plugin. You can test the above code by loading the plugin into HP CDA and browsing the software artifacts again as you tried previously. This should now display the tree interface and allow the selection of artifacts.

Completing the plugin IExternalArtifactProvider

This section will work with the Packaging interface, and will be responsible for taking the definition and returning the artifacts. In this example we simply use the date/time as a representation of the version.

```
        public InputStream getArtifactContent(IExternalArtifact artifact)
                        throws ExternalArtifactException {
                String path = artifact.getName();
                File f = new File(path);
                try {
                        return new FileInputStream(f);
                } catch (FileNotFoundException e) {
                        throw new LocalizableException(getBundle(), "FileNotFound");
                }
        }
```

The above method returns the input stream for the artifact.

```
        public ArtifactStatus getArtifactStatus(IExternalArtifact artifact)
                        throws ExternalArtifactException {

                if (artifact == null || artifact.getVersion() == null || 
artifact.getVersion().trim().length() == 0) {
                        return ArtifactStatus.NEW;
                }
                String version = artifact.getVersion();
                String path = getDirectory() + File.separator + artifact.getRelativeUri();
                path = path.replace("/", File.separator);

                File f = new File(path);
                if (f.exists()) {
                        String newVersion = toVersion(f);
                        if (version.equals(newVersion)) {
                                return ArtifactStatus.UNCHANGED;
```

```
				} else {
					return ArtifactStatus.UPDATED;
				}
			} else {
				return ArtifactStatus.DELETED;
			}
		}
```

The above method retrieves the status of the artifact. It normalizes and assembles the relativeURI with the base directory. It then assembles a new version string based on the date/time of the artifact in the file system and compares the date to the stored artifact.

```
		public IExternalArtifact refreshArtifact(IExternalArtifact artifact)
				throws ExternalArtifactException {
			String path = getDirectory() + File.separator + artifact.getRelativeUri();
			path = path.replace("/", File.separator);

			return  createArtifact(new File(path), artifact.getFilters());
		}
```

The above API provides an updated artifact.

Note
There is a defect in getFileSetContent in that a ZipInputStream is a required output. Check the HP CDA release notes for a resolution to this issue. Until the issue is resolved, the more specific ZipInputStream can be used.

```
		public InputStream getFileSetContent(
				IExternalArtifactFileSetDescriptor descriptor)
				throws ExternalArtifactException {

			String path = getDirectory() + File.separator + descriptor.getRelativeUri();
			path = path.replace("/", File.separator);
			File f = new File(path);
			InputStream result = null;
			try {
				File zipFile = File.createTempFile("serverfs.", ".zip");
				ZipUtils.zip(zipFile, true, true, true, f);
				result = new ZipInputStream(new FileInputStream(zipFile));

			} catch (FileNotFoundException e) {
				throw new ExternalArtifactException("Failed to get content for " +
descriptor.getRelativeUri());
			} catch (IOException e) {
				throw new ExternalArtifactException("Failed to create zip archive of content
at " + descriptor.getRelativeUri());
			}
			return result;
		}
```

For more information

HP software product manuals and documentation for HP CDA can be found at http://h20230.www2.hp.com/selfsolve/manuals. You will need an HP Passport to sign in and gain access.

Note
General-access documentation requires that you register for an HP Passport and sign in. In some cases, access to the documentation is restricted and requires that you have an active HP support agreement ID (SAID) and an HP Passport sign-in.

Table 1. Document Revision History

Date or Version	Changes
December 2013	Initial release of document.

Sign up for updates
hp.com/go/getupdated