Sun N1 Service Provisioning System 5.1 Plug-in Development Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 819–1662–10
September 2005

050829@12762

Contents

Examples

Preface

The *Sun N1™ Service Provisioning System 5.1 Plug-In Development Guide* explains how to create plug-in solutions.

Who Should Use This Book

The audience for this book includes Sun™ internal developers, partners, and ISVs who need to develop solutions for applications to be provisioned through the Sun N1 Service Provisioning System (N1 SPS) software. These readers should be familiar with the following items:

- Networking and data center environments
- TheN1 SPS product
- Standard Unix® and Microsoft Windows commands and utilities, as appropriate for the plug-in being developed
- Java™ programming and standards
- XML and standard XML editors and parsers

Before You Read This Book

To become familiar with theN1 SPS product, read the following documentation:

- *Sun N1 Service Provisioning System 5.1 Installation Guide*
- *Sun N1 Service Provisioning System 5.1 System Administration Guide*
- *Sun N1 Service Provisioning System 5.1 Plan and Component Developer's Guide*

How This Book Is Organized

Chapter 1 introduces you to the concept of plug-ins for the N1 SPS product.

Chapter 2 describes the process and procedures that you use to create a plug-in.

Chapter 3 explains the Java-based APIs that you can use for your plug-in.

Appendix A provides sample XML and Java examples for a plug-in.

Documentation, Support, and Training

Sun Function	URL	Description
Documentation	`http://www.sun.com/documentation/`	Download PDF and HTML documents, and order printed documents
Support and Training	`http://www.sun.com/supportraining/`	Obtain technical support, download patches, and learn about Sun courses

Typographic Conventions

The following table describes the typographic changes that are used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`

TABLE P–1 Typographic Conventions *(Continued)*

Typeface or Symbol	Meaning	Example
AaBbCc123	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. Perform a *patch analysis*. Do *not* save the file. [Note that some emphasized items appear bold online.]

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Overview of N1 Service Provisioning System Plug-Ins

This chapter provides a brief introduction to the Sun N1 Service Provisioning System (N1 SPS) environment and explains how plug-ins fit into that environment. The chapter contains the following information:

Overview of Sun N1 Service Provisioning System

The N1 SPS product is an object oriented, XML-based, distributed environment to solve enterprise system configuration, service provisioning, and application deployment needs. The provisioning system provides an extensible framework and environment that at a minimum provides the following functionality:

- Common framework to build service provisioning automation
- Maintains an audit log of changes over time
- Compares the current state of target hosts with their expected state
- Simulates a change to identify configuration problems
- Implements a set of rules to govern automation execution
- Notifies system administrators of problems and actions
- Automatically manages version control

The N1 SPS software implements a distributed environment in which object-oriented components are authored in XML scripts and orchestrated to follow execution plans for distribution, provisioning, and installation needs. For more information about N1 SPS basic concepts and terminology, see Chapter 1, "N1 Service Provisioning System 5.1 Overview," in *Sun N1 Service Provisioning System 5.1 Installation Guide*.

Overview of the Solution Development Environment

You can use the provisioning system to build system configuration, service provisioning, and application deployment solutions. At a very high level, you follow this simple process:

1. Build a set of components. This step might involve any of the following sub-tasks:
   a. Defining application-specific component types
   b. Naming each component
   c. Assigning a component type to each component
   d. Identifying any source files and directories that a component needs
   e. Defining specific tasks for that component
2. Create a plan to direct the deployment of the components. Each plan includes the following information:
   a. A list of components
   b. A sequence in which the components are to run
   c. A list of any variables that the components need
   d. A set of target hosts to which the components should be deployed
3. Create a plug-in that enables others to use the components and plans that you developed for a given platform or application. This task involves four main sub-tasks:
   a. Installing the core plug-in development files
   b. Using the XML plug-in definition schema to provide an interface for the users
   c. Using the Java-based APIs to provide component browsers and exporters, and to define custom `execJava` steps.

   Note – If you need to use Java to develop your plug-in, use the NetBeans product. For more information, see `http://www.netbeans.org/`.

   d. Packaging the components, plans, resources, plug-in definition files, and APIs for delivery to other N1 SPS users

Introduction to Plug-Ins

In general usage, plug-in applications are programs that can easily be installed and used as part of your web browser. A plug-in application is recognized automatically by the browser and its function is integrated into the main HTML file that is being presented. Web browser plug-in applications generally play sound or motion video or perform some other functions.

In the N1 SPS environment, a plug-in differs only slightly in concept from the general usage. A plug-in for the N1 SPS product is a packaged solution that extends the provisioning capability of the product for a specific platform, application, or environment. For example, you might create a plug-in solution for a specific application, such as Oracle 8i, or for some feature of an operating system, such as Solaris Zones.

A plug–in includes all the relevant data needed to support a new custom application. The contents of the plug–in are described in the plug–in descriptor file. This file is located in a standard place within the plug–in packaging structure.

The plug-in descriptor contains meta-data about the plug-in including name, description, vendor, version number, previous version, and dependencies. In addition, the descriptor may contain a pointer to a `readme.txt` file. The descriptor also contains instructions for creating components, plans, folders, host types, host sets, host searches, resources, component types, and system services. The descriptor may optionally define a library of server-side plug-in code and a set of GUI extensions for the plug-in.

Objects defined in the plug-in are loaded in the order in which they are defined within the descriptor file. Objects defined in the plug-in may only reference objects defined earlier in the plug-in, or in a plug-in on which this plug-in directly depends. This dependency must be declared in the plug-in descriptor.

XML Schemas

In the N1 SPS environment, plans, components, and other parts of the solution are defined through XML. You can use several XML schemas to define your plug-in solution. The following schemas are provided in the `docs/xml` directory of the product media:

- `plugin.xsd` – Plug-in schema used to describe the parts of the plug-in through the plug-in descriptor file
- `pluginUI.xsd` – Plug-in user interface schema used to define an interface to the plug-in within the N1 SPS browser interface
- `component.xsd` – Component schema used to define components and component types

- `plan.xsd` – Plan schema used to define execution plans
- `planCompShared.xsd` – Schema that contains elements that are common to plans and components

This document includes examples that illustrate the XML schemas. For complete reference information about the elements and attributes used in the XML schemas, see *Sun N1 Service Provisioning System 5.1 XML Schema Reference Guide*.

Parts of a Plug-In

A plug-in solution includes all the relevant data needed to support a new custom application. This data includes first-class provisioning system objects:

- Components
- Component types
- Folders
- Host searches
- Host sets
- Host types
- Plans
- System services

In addition, the plug-in can also include auxiliary objects for use by the system, such as the following objects:

- Resources
- Server-side plug-in code (using Java-based APIs)
- Browser interface extensions

Plug-In Packaging

A plug-in is packaged as a Java Archive (JAR) file. The contents and instructions for interpreting the contents of the JAR file are contained in an optionally signed `plugin-descriptor.xml` file located in the top-level directory of the JAR file. The syntax of the plug-in descriptor is specified using XML Schema as per the May 2, 2001 W3C Recommendation (`http://www.w3.org/TR/2001/xmlschema-0-20010502/`). The schema can be used in conjunction with a validating parser to determine the syntactical validity of a plug-in descriptor file.

Recommended Naming

To avoid potential conflicts, you should use a Java package naming convention for plug-ins (`com.`*companyname*`.`*productname*, for example, `com.sun.solaris`). Any objects that can be in a folder should be placed in a folder directory structure that mirrors the plug-in name, such as `/com/sun/solaris`. The plug-in JAR file name should use the convention *pluginname_version*`.jar`, for example `com.sun.solaris_1.1.jar`.

Installation Considerations

To install a plug-in, service provisioning administrators load the plug-in JAR file.

Plug-In Upgrade Considerations

To upgrade from an existing version of a plug-in to a newer version of a plug-in, you provide a patch JAR file that contains only the contents needed for the patch . For example, if you only changed two component types between version 1.2 and version 1.3, then your upgrade patch would contain only those new component type XML files. Define your patch so that it can be applied in series to upgrade multiple versions . For example, to upgrade from version 1.0 to version 1.2, a user would first apply the upgrade from 1.0 to 1.1, then apply the upgrade from version 1.1 to version 1.2. Update patches are strictly additive with respect to the previously loaded version of the plug-in. You can also create a patch that would upgrade from a specific existing version (for example, 1.0) to a specific newer version (for example, 1.3). However, you cannot create a patch to upgrade from any arbitrary version to a higher version.

Uninstalling Plug-In Versions

You cannot uninstall an individual patch of a plug-in, and you cannot delete objects created by previous versions of a plug-in. To remove this content, you would need to uninstall the current version of the plug-in and reinstall the older version of the plug-in. Alternatively, you could create an *anti-patch* that would install the old plug-in version's code while creating new versions of the plug-in defined objects.

Component Versions and Dependencies

Objects that are defined by a plug-in are loaded at installation time in the order in which they are defined in the plug-in descriptor file. These objects may only reference other objects that were defined either earlier in the plug-in or in a plug-in on which the defining plug-in directly depends. Any dependencies must be declared in the plug-in descriptor file. If a plug-in attempts to create a versioned object that matches a

same typed and named object existing in the system, a new version of the object is created. The minor version of this object is incremented unless the plug-in definition explicitly defines the object as requiring a major version increment. If a plug-in attempts to create a non-versioned object that matches a same typed and named existing object, the plug-in object replaces the previous definition of the object. In both the versioned and unversioned cases, the existing object must have been created by a prior version of the same plug-in that is attempting to create the new version of the object.

Certificates

If the plug-in descriptor file is signed for one version of a plug-in, then the file must be signed for any subsequent versions of that plug-in. Use the standard `jarsigner` tool to sign the plug-in descriptor file. If the file is signed, the signature will be verified against the public certificate when the plug-in is installed. When upgrading a plug-in, the certificate used to sign the newer version is matched against the certificate used to sign the existing version in the system. The upgrade will not succeed if certificates have expired between plug-in versions.

You should sign all entries in the plug-in JAR (not just the plug-in descriptor file) with the same certificate. Only a single certificate may be attached to each entry.

Security Considerations

A plug-in does not include facilities for defining groups or permissions. This is because permission management depends highly on the environment into which the plug-in is loaded, and cannot be effectively modeled for all environments.

The administrator who adds the plug-in must decide what permissions are appropriate. The general expectation is that plug-ins are designed to be used by everyone. However, certain clients may wish to limit the use of a plug-in to a certain group. Plug-ins may also have certain folders that are meant to have different execution permissions.

Plug-In `readme.txt` File

You can provide a `readme.txt` file with your plug-in, if needed. The plug-in `readme.txt` file is intended as the holding place for instructions on configuring the system for a plug-in. In general, the `readme.txt` file should document the permissions, session variables, and other instance-specific settings required for the plug-in to function. Specifically, the `readme.txt` should contain instructions for setting permissions on plug-in created folders, as well as enumerating expected session variables, their descriptions and encryption methods.

CHAPTER 2

Creating a Plug-In

This chapter explains how to use the plug-in framework to create a provisioning solution for a specific application or platform. The chapter includes the following information:

Installing the Plug-In Development Environment

Most of the pieces that you need to create a plug-in solution are part of the standard Sun N1 Service Provisioning System software. However, you must install a few additional software ingredients to provide you with a complete development solution. These key pieces are contained in the `plugin-core.jar` file on the *Sun N1 Service Provisioning System 5.1 Supplement CD*.

Note – Once you place the `plugin-core.jar` file where you want, be sure to modify the `classpath` for your Java tools to find the file.

The `plugin-core.jar` contains three packages that provide file system-based component browse and export classes:

`com.sun.n1.sps.pluginimpl.system`
Includes several constants that identify supported platforms

`com.sun.n1.sps.pluginimpl.system.browse`
Includes five classes that you can use to support file system-based browsing functionality:

- `FileDisplay` – A display appropriate for file system files
- `FilesystemBrowser` – A hierarchy browser for files ystems
- `FilesystemBrowserFactory` – Factory to return types sufficient for browsing a file system as a hierarchy
- `FilesystemExtensionFilter` – A `FilesystemFilter` that filters based on the file extension suffix
- `FilesystemFilter` – Base class for all file system filters

`com.sun.n1.sps.pluginimpl.system.export`
Provides one class `FilesystemExporter` that you can use to export a simple filesystem object

Creating a Plug-In: Process Overview

Developing a plug-in solution can be simple or complex, depending on the needs of your environment and the application or platform to which the solution applies. A plug-in solution can involve any of the following segments of the Sun N1 Service Provisioning System environment:

- Working with variables and configuration templates
- Enabling users to browse through files and export those files to the master server
- Executing Java applications through the `execJava` feature
- Creating and modifying components and plans
- Packaging the plug-in and defining an interface for it through the plug-in XML

The general process that you follow includes the following steps:

1. Develop a general model for the platform or application.
2. Create plans and components to implement the model.

3. Define specific host types, host sets, and host searches to easily constrain the plug-in.
4. Define an interface for the application within Sun N1 Service Provisioning System.
5. Package the plans, components, resources, and interface definition into a Java Archive (JAR) file.
6. Test the plug-in.

Plug-In Directory Structure

As you develop your solution using the plug-in framework, you need to pay attention to where files are placed. Having an accurate record of the files is essential when you package your solution into a JAR file. The following list illustrates a recommended directory structure for plug-ins:

```
META-INF
components
plans
resources
gui
plugin-descriptor.xml
readme.txt
```

`META-INF` directory	Contains the mainfest of pieces of the plug-in.
`components` directory	Contains a series of subdirectories that contain component and component type XML definition files. Subdirectories follow the structure of the plug-in name. For example, if the plug-in name is `com.sun.solaris`, the `components` subdirectories would be `com`, then `sun` then `solaris`. For example, the actual component XML files would live inside the `components/com/sun/solaris` directory.

Note – You might want to wrap the components, plans, and resources directories into a larger directory structure for a given plug-in version. For example, to differentiate between versions 1.0 and 1.1 of a given plug-in, you might use directory structures such as `1.0/components/com/sun/solaris/Project.xml` and `1.1/components/com/sun/solaris/Project.xml`

`plans` directory	Contains a series of subdirectories that contain execution plan XML definition files. Subdirectories follow the structure of the plug-in name. For example, if the plug-in name is `com.sun.solaris`, the `plans` subdirectories would be `com`, then `sun` then `solaris`. For example, the actual execution plan XML files would live inside the `plans/com/sun/solaris` directory.
`resources` directory	Contains a series of subdirectories that contain resource files. Subdirectories follow the structure of the plug-in name. For example, if the plug-in name is `com.sun.solaris`, the `resource` subdirectories would be `com`, then `sun` then `solaris`. For example, the actual resource files would live inside the `resources/com/sun/solaris` directory.
`gui` directory	Contains the user interface descriptor file (`pluginUI.xml`) and files for any icons that need to be displayed in the user interface. See Chapter 7, "Plug-In User Interface Schema," in *Sun N1 Service Provisioning System 5.1 XML Schema Reference Guide* for more information about the elements in the user interface descriptor file.
`plugin-descriptor.xml` file	XML file that describes the plug-in. See Chapter 6, "Plug-In Description Schema," in *Sun N1 Service Provisioning System 5.1 XML Schema Reference Guide*for more information about the elements in the plug-in descriptor file.
`readme.txt` file	Text file that contains any instructions on configuring the system for the plug-in.

Developing a Model

Before you build your plug-in solution, you need to do some planning and modeling work. The following questions indicate some common areas to consider:

- What is the expected environment in which you want this solution to be used? For example, operating system requirements, application version requirements, and so on.
- Do you need to account for any variable values, such as path names, when provisioning this platform or application?
- What files need to be deployed to the provisionable hosts to enable this platform or application to function? For example, configuration files.
- Do you need to define any new component types for this solution, or can you use the existing component types? Many simple solutions can use existing component types, such as `system#file` and `system#directory`. If necessary, however, you can define your own component types that extend the existing component types.
- Will a user need to browse for and create instances of a component from a remote system?
- What is the flow of tasks you need your users to be able to perform?

The following illustrates one possible modelling flow, based on the flow for deploying Java™ 2 Platform, Enterprise Edition (J2EE) :

1. Deploy infrastructure.
   - Execute installer binaries to install infrastructure
   - Install targetable components
2. Capture all application objects as components, such as the following objects:
   - Java Archive (JAR) files, Enterprise Archive (EAR) files, Web Archive (WAR) files, Enterprise Java Beans (EJB) files
   - JDBC connection and data sources
3. Create an “environment” component that contains environment settings, such as the following:
   - Java Virtual Machine (JVM) settings
   - Session management settings
4. Configure application and environment components
5. Deploy components into targetable components

Creating Components and Plans

To be able to effectively reproduce a given solution across an enterprise, you need to define components, resources, and plans that identify common parts of the solution. In addition, you need to define a process for deploying them. For more information about plans, components, and how to manage them, see *Sun N1 Service Provisioning System 5.1 Plan and Component Developer's Guide*.

Building Components

A key piece in developing your solution is creating components. In the Sun N1 Service Provisioning System environment, components are deployable objects. Some examples of the objects you might have in components include the following:

- A collection of files and directories
- Archive files, such as JAR files or EAR files
- Complete applications, including all needed resources
- Specific application resources, such as configuration files or documentation

For information about creating components by using the Sun N1 Service Provisioning System browser interface, see "How to Create a Component" in *Sun N1 Service Provisioning System 5.1 Plan and Component Developer's Guide*.

Simple and Composite Components

Simple components contain a single physical resource, such as a file, directory, archive file, or application. Simple components do not reference other components.

Composite components only reference other simple or composite components. Composite components do not directly contain any physical resources.

EXAMPLE 2–1 XML for a Simple Component

The following XML example shows a simple component that extends the system component type `system#CR Simple Base` to contain a JAR file. For more information about the specific elements and attributes used to define a component, see Chapter 3, "Component Schema," in *Sun N1 Service Provisioning System 5.1 XML Schema Reference Guide*.

```
<?xml version="1.0" encoding="UTF-8"?>
<component xmlns='http://www.sun.com/schema/SPS' name='plugin-core.jar'
    version='5.1' description='Jar file implementation of core plugin services'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' author='system'
```

EXAMPLE 2–1 XML for a Simple Component *(Continued)*

```
    softwareVendor='Sun Microsystems' path='/system'
    xsi:schemaLocation='http://www.sun.com/schema/SPS component.xsd'>
  <extends>
    <type name='system#CR Simple Base'>
    </type>
  </extends>
  <resourceRef>
    <resource name='/system/plugin-core.jar' version='1.1'>
    </resource>
  </resourceRef>
</component>
```

Variables

When you create a component or plan, you can define variables to use when that component is deployed or the plan is executed. Many component types include common variables, such as *installPath*, which defines where to install the component. The value of the *installPath* variable is determined for a given host when the component is installed on that host.

Some common variables that you might see include the following:

- *installPath* – Path to where the component, plug-in, or other resource file is installed
- *installName* – Name of item being installed
- *installUser* – Login name of the user who installed the component, plug-in, or other resource file
- *pluginClasspath* – Path to where the classes that apply to a specific plug-in are installed

A variable can refer to another variable, such as the variable of a container component. For example, the value of the *installPath* variable for a simple component could be the value of the *installPath* variable for its parent container component.

When defined, each variable must have a name and a default value attribute. The default value can be obtained from several places:

- A literal string
- The host, using the *target* keyword
- Another component, using the *component* keyword
- The user's session, using the *session* keyword

For detailed information about using these attributes, see "Types of Variables Available for Substitution" in *Sun N1 Service Provisioning System 5.1 Plan and Component Developer's Guide*.

You can define a variable through the browser interface or directly in the XML file. Within the XML file, variables are defined using the `<var>` element and contained within a `<varList>` element.

EXAMPLE 2–2 Variable Definitions in XML

The following XML fragment shows several variable definitions.

```
<varList>
    <var name='installPath'
         default=':[target:sys.raDataDir]:[/]systemcomps'>
    </var>
    <var name='pluginClasspath'
         default=':[installPath]:[/]plugin-core.jar'>
    </var>
    <var name='fileBrowser'
          default='com.sun.n1.sps.pluginimpl.system.browse.FilesystemBrowserFactory'>
    </var>
    <var name='directoryBrowser'
          default='com.sun.n1.sps.pluginimpl.system.browse.FilesystemBrowserFactory'>
    </var>
    <var name='symlinkBrowser'
          default='com.sun.n1.sps.pluginimpl.system.browse.FilesystemBrowserFactory'>
    </var>
</varList>
```

Configuration Templates

A *configuration template* is a special type of file component. The configuration template enables you to do token substitution in a file that you are deploying. An example of this usage would be deploying the DNS `/etc/resolv.conf` file. The goal for deployment might be to have the file use a variable substitution and use a host type attribute to define the closest DNS server. The configuration template might look like the following example:

```
search :[search_path]
nameserver :[primary_dns]
nameserver :[secondary_dns]
```

In this case, the configuration template would automatically create component variables called *search_path*, *primary_dns*, *secondary_dns*. Then you could use variable substitutions in plans or component controls to provide appropriate values.

▼ How to Define a Configuration Template

Step

- **To designate a file component as a configuration template, select "configuration template" in the Options section of the Component Details page.**

Defining Component Types

Many basic component types are included with the Sun N1 Service Provisioning System product. Some of these basic component types include such items as files and directories. You can also define specific component types for use with a specific application or platform. For example, perhaps your application has some specific file types that would always exist for this application. You could then define a new component type for your application that is based on the `system#CR Simple Base` component type but extends that component type for your specific application.

The component type definition is stored in an XML file like any other component XML file. When you define your plug-in, you provide a path to the file for the *backing component* in the `<component>` element in the descriptor file. You use the `<componentType>` child element of the `<component>` element to provide additional information, such as its name, description, and so on. For more information, see Example 2–15.

▼ How to Create a Component Type

Steps

1. **Determine what component types you need to create.**
2. **From the Administrative section of the browser interface, click Component Types.**
3. **From the Component Types page, click Create in the Action column.**

Note – You can type a name for the component type before you choose the Create action or you can type or change the name in the Edit window. Once you check in the component type, you cannot change its name.

---

4. **Accept the name or change it.**

   A component type name has a maximum of 64 characters. The name must start with a letter or underscore, followed by any number of letters, digits, or special characters, such as underscore (_), period (.), plus (+), minus (-), and space ( ). Unicode letters and digits are permitted.

5. **(Optional) Type a menu group name.**

   Group names follow the same requirements as the component type name . In addition, a group can be declared as `hidden`, which prevents the type from showing up in the component type drop-down list on the component list page.

6. **Type an alpha-numeric string for the menu order.**

The menu order is a maximum of 18 characters. In addition to Unicode letters and digits, any character that you can type on an ASCII keyboard is permitted. The order should be sufficient to sequence all of the types that are defined within a particular plug-in.

7. **Type a numeric value between 0 and 10 for the indent level.**

   The indent level specifies how types should appear within the component type drop-down list on the component list page. Indents are used in conjunction with ordering to imply relationships between component types. For example, a `wep app` type might have related types `web app configuration` and `web app archive` that are indented more to highlight the relationship.

8. **(Optional) Type a description.**

   The description must be less than 1024 characters in length. In addition to Unicode letters and digits, any character that you can type on an ASCII keyboard is permitted.

9. **Select a backing component.**

   A backing component provides a template for the component type.

10. **Click Save.**

Creating Plans

A plan is a sequence of instructions that is used to manage one or more components on the specified hosts. For example, a plan might install three components and initiate the startup control of another component. To create most plans, you have to edit the XML. The one exception to this rule is an auto-generated plan. The Sun N1 Service Provisioning System software can automatically generate a plan consisting of direct run procedures. For example, you could auto-generate a plan that consists of installing a single component. You could then run this plan directly or save it for use as a template for authoring more complex plans.

Simple and Composite Plans

Simple plans contain a series of deployment instructions, or steps. Simple plans are executed on a single host or host set. Simple plans can call common procedures, such as install or uninstall, and can also use conditional programming constructs.

Composite plans contain calls to simple plans. Composite plans can apply some procedures to one host, while applying other procedures to a different host or host set.

EXAMPLE 2–3 XML for a Simple Plan

A simple plan might look like the following example. This plan provides an install block and an uninstall block. For more information about the specific elements and attributes used to define a plan, see Chapter 4, "Plan Schema," in *Sun N1 Service Provisioning System 5.1 XML Schema Reference Guide*.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated by N1 SPS -->
<executionPlan xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    name='plugin-core.jar-1096573592002' version='5.1'
    xsi:schemaLocation='http://www.sun.com/schema/SPS plan.xsd'
    xmlns='http://www.sun.com/schema/SPS' path='/system/autogen'>
    <simpleSteps>
        <install blockName='default'>
            <component name='plugin-core.jar' path='/system' version='1.1'>
            </component>
        </install>
        <uninstall blockName='default'>
            <installedComponent name='plugin-core.jar' versionOp='='
                 version='1.1' path='/system'>
            </installedComponent>
        </uninstall>
    </simpleSteps>
</executionPlan>
```

EXAMPLE 2–4 XML for a Composite Plan

A composite plan might look like the following example. This example calls three sub-plans.

```
<executionPlan
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  name="apache-tomcat-uninstall" version="4.0"
  xsi:schemaLocation="http://www.centerrun.com/schema/CR plan.xsd"
  xmlns="http://www.centerrun.com/schema/CR">
    <compositeSteps>
        <execSubplan planName="mod-jk-uninstall" />
        <execSubplan planName="apache-uninstall" />
        <execSubplan planName="tomcat-uninstall" />
    </compositeSteps>
</executionPlan>
```

EXAMPLE 2–5 XML for a More Sophisticated Plan

The following example shows a more complicated plan that determines what to execute based on some conditions.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated by CR -->
<executionPlan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    name="BAM_backout_new_version_NODE-A" version="4.0"
    xsi:schemaLocation="http://www.centerrun.com/schema/CR plan.xsd"
```

EXAMPLE 2–5 XML for a More Sophisticated Plan *(Continued)*

```
 xmlns="http://www.centerrun.com/schema/CR" path="/plans/uat">
<paramList>
   <param name="backout_type" prompt="Enter type of backout (all,ear,prop)"></param>
</paramList>
<varList>
   <var name="admin_server" default="wusx119"></var>
   <var name="node" default="wust3022"></var>
   <var name="wl_server_name" default="bamC"></var>
   <var name="apphome" default="/opt/uat/ceodomain"></var>
   <var name="prop_args" default="-s wust3022"></var>
   <var name="application_name" default="bam"></var>
   <var name="staging_base" default="/usr/local"></var>
   <var name="user" default="weblogic"></var>
</varList>
<simpleSteps limitToHostSet="uat-bam">
   <if>
      <condition>
         <or>
            <equals value2="all" value1=":[backout_type]"></equals>
            <equals value2="prop" value1=":[backout_type]"></equals>
            <equals value2="ear" value1=":[backout_type]"></equals>
         </or>
      </condition>
   <then>
      <call blockName="backout_application">
         <argList application_name=":[application_name]"
              staging_base=":[staging_base]"
              backout_type=":[backout_type]"
              user=":[user]">
         </argList>
         <installedComponent name="deploy_tools"
               path="/components/function_library">
          </installedComponent>
      </call>
      <call blockName="wl_stop">
         <argList wl_server_name=":[wl_server_name]"
               node=":[node]" apphome=":[apphome]" user=":[user]">
         </argList>
         <installedComponent name="deploy_tools"
               path="/components/function_library">
         </installedComponent>
      </call>
      <if>
         <condition>
           <equals value2="all" value1=":[backout_type]"></equals>
         </condition>
      <then>
            <call blockName="clusterdeploy">
               <argList application_name=":[application_name]"
                     staging_base=":[staging_base]" node=":[node]" user=":[user]">
               </argList>
               <installedComponent name="deploy_tools"
```

EXAMPLE 2–5 XML for a More Sophisticated Plan *(Continued)*

```
            path="/components/function_library">
         </installedComponent>
      </call>
      <call blockName="deploy_prop">
         <argList application_name=":[application_name]"
               prop_args=":[prop_args]" staging_base=":[staging_base]"
               user=":[user]">
         </argList>
         <installedComponent name="deploy_tools"
               path="/components/function_library">
            </installedComponent>
         </call>
         <call blockName="wl_startjsp">
            <argList application_name=":[application_name]"
               wl_server_name=":[wl_server_name]"
               node=":[node]" apphome=":[apphome]" user=":[user]">
            </argList>
            <installedComponent name="deploy_tools"
               path="/components/function_library">
            </installedComponent>
         </call>
      </then>
   </if>
   <if>
      <condition>
         <equals value2="ear" value1=":[backout_type]"></equals>
      </condition>
      <then>
         <call blockName="clusterdeploy">
            <argList application_name=":[application_name]"
               staging_base=":[staging_base]" node=":[node]" user=":[user]">
            </argList>
            <installedComponent name="deploy_tools"
               path="/components/function_library">
            </installedComponent>
         </call>
         <call blockName="wl_startjsp">
            <argList application_name=":[application_name]"
               wl_server_name=":[wl_server_name]"
               node=":[node]" apphome=":[apphome]" user=":[user]">
            </argList>
            <installedComponent name="deploy_tools"
               path="/components/function_library">
            </installedComponent>
         </call>
      </then>
   </if>
   <if>
      <condition>
         <equals value2="prop" value1=":[backout_type]"></equals>
      </condition>
      <then>
```

EXAMPLE 2–5 XML for a More Sophisticated Plan *(Continued)*

```
            <call blockName="deploy_prop">
               <argList application_name=":[application_name]"
                  prop_args=":[prop_args]"
                  staging_base=":[staging_base]" user=":[user]">
               </argList>
               <installedComponent name="deploy_tools"
                  path="/components/function_library">
               </installedComponent>
            </call>
            <call blockName="wl_start">
               <argList application_name=":[application_name]"
                  wl_server_name=":[wl_server_name]"
                  node=":[node]"
                  apphome=":[apphome]"
                  user=":[user]">
               </argList>
               <installedComponent name="deploy_tools"
                  path="/components/function_library">
               </installedComponent>
            </call>
         </then>
      </if>
   </then>
   <else>
      <raise message="Please enter a valid deployment type (all/ear/prop)"></raise>
   </else>
  </if>
 </simpleSteps>
</executionPlan>
```

▼ How to Generate a Plan

Steps

1. **Go to the Components page.**
2. **Select the component for which you want to generate the plan.**
3. **View the component's details.**
4. **If needed, scroll down the page until you see Component Procedures.**
5. **Select the procedures that you want to use in the plan.**
6. **Click Generate Plan with Checked Procedures.**

   The Plans editing page appears. From this page, you can modify the XML to include more complex steps, like those shown in Example 2–5.

Using Native Commands in Plans and Components (<execNative> Step)

The <execNative> XML step enables you to run native commands from within your plans and components. For example, if you need to verify that a process has started, you might use <execNative> to call the UNIX ps command. For more information about the <execNative> schema, attributes, and child elements, see "<execNative> Step" in *Sun N1 Service Provisioning System 5.1 XML Schema Reference Guide*.

Before <execNative> executes the specified command, the Sun N1 Service Provisioning System software verifies that the command exists and that the specified user has permission to run the command. If either of these checks fail, <execNative> exits with an error.

EXAMPLE 2–6 Using <execNative> to Invoke a Simple Command

The following <execNative> example performs the equivalent of the UNIX ps -ef command.

```
<execNative>
    <exec cmd="ps">
        <arg value="-ef" />
    </exec>
</execNative>
```

EXAMPLE 2–7 Using <execNative> to Start an Application

The following <execNative> example starts a web server instance.

```
<execNative
  dir="/opt/ns/https-admserv"                    Set working directory
  userToRunAs="webadmin"                         Equates to "su -webadmin"
  timeout="5">
    <inputText>
        start.sh                                 Input parameters to command
    </inputText>
    <exec cmd="sh />                             Command to run
    <successCriteria status="0" />               execNative succeeds only if exit code is
                                                 "0"
</execNative>
```

Calling Java-based Objects in Plans and Components (<execJava>)

The <execJava> mechanism enables agent-side, in-process execution of client-provided Java code within a plan or component definition. <execJava> is similar to <execNative>, but is specifically intended to enable execution of Java code.

The <execJava> feature is provided as an XML step and as a Java-based API. For information about the XML schema, attributes, and child elements, see "<execJava> Step" in *Sun N1 Service Provisioning System 5.1 XML Schema Reference Guide*. For more information about the execJava API, including examples, see "execJava API" on page 50.

The <execJava> XML step has one required and two optional attributes:

- *className* – A required attribute that provides the Java class to be executed on the target host.
- *classPath* – An optional attribute that provides the path to the class identified by the *className* attribute; If this attribute is not used, the system class path of the remote agent is used
- *timeout* – An optional attribute that specifies the number of seconds to wait for the Java class to execute before timing out

The <execJava> mechanism can pass arguments to the Java Executor using the <argList> child element.

EXAMPLE 2–8 Using <execJava> in Component XML

```
<varList>
   <var name="installPath" default="/opt/util"/>
</varList>
<resourceList defaultInstallPath=":[installPath]">
   <resource resourceName="util/propPrint.jar" installName="propPrint.jar"/>
</resourceList>
   ...
<controlList>
   <control name="showProp"/>
   <paramList>
       <param name="propName">
   </paramList>
   <execJava
       className="com.raplix.util.PropertyPrinterFactory"
       classPath="$[installPath]/propPrint.jar">
       <argList>
           <arg name="propertyName" value=":[propName]"/>
       </argList>
       <successCriteria outputMatches="<undefined>" inverse="true"/>
   </execJava>
```

EXAMPLE 2–9 Using <execJava> in Plan XML

```
<executionPlan xmlns="http://www.sun.com/schema/SPS"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.sun.com/schema/SPSplan.xsd"
   name="execJavaExample" version="5.1">
   <paramList>
       <param name="name"></param>
       <param name="value"></param>
   </paramList>
   <varList>
       <var name="classpath"
           default=":[target:sys.raDataDir]:[/]systemcomps:[/]plugin-com.sun.sample.jar"/>
   </varList>
   <simpleSteps>
       <execJava className="com.sun.n1.sps.pluginimpl.sample.executor.SampleExecutorFactory"
           classPath=":[classpath]">
           <argList nameParam=":[name]" valueParam=":[value]" />
       </execJava>
   </simpleSteps>
</executionPlan>
```

Conditional Elements

Within a plan or a component, you can use the <if> element to conditionally perform a block of steps. Similar to traditional programming if-then-else constructs, the statement within the <if> element is evaluated. If that statement is true, then the steps of the <then> element are performed. Otherwise, the steps of the <else> element are performed. If no <else> element exists, then no action is taken.

EXAMPLE 2–10 XML for <if> Element

The following example uses the <if> element to allow users to decide at deployment time whether to take a snapshot.

```
<if>
    <condition>
        <istrue value=:[createSnapshot]"></istrue>
    </condition>
    <then>
        <createSnapshot blockName="default"></createSnapshot>
    </then>
</if>
```

Error Handling

The XML schemas provides a set of elements for handling possible errors . The parent of this set of elements is the `<try>` element. You might use these elements for situations similar to the following examples:

- To suppress errors during deployment. For example, if installation of a component consists of deploying some files or other resources followed by a restart and the restart fails, then the installation itself does not fail.
- To control whether a step that depends on another step should be performed. For example, if you need to perform both `useradd` and `groupadd` functions, the `groupadd` should only be performed if the `useradd` is successful.
- To determine which install path to take. For example, if you are installing version 1.1 of an application, the install path might be different depending on whether version 1.0 of that application is on the target host.

The `<try>` element includes a block of steps that are executed in order until either all complete successfully or a step fails. If a step fails and a `<catch>` element exists, then the steps in the `<catch>` element are executed in order until they succeed or a step fails. If a `<finally>` element is defined, the steps in the `<finally>` element are executed in order until all steps complete or a step fails *regardless* of whether the `<try>` and `<catch>` elements succeeded. Typically, the `<finally>` element is used to perform clean-up functions or release resources.

The `<raise>` element is used to indicate a failure condition without having to create a step to do so. The `<raise>` step always fails. Although the `<raise>` element can be used by itself, it is often contained within a `<catch>` element block.

EXAMPLE 2–11 XML for `<try>` Element

The following XML example uses the `<try>` element to determine whether a fresh install or an upgrade install should be performed.

```
<installSteps blockName="default">
    <try>
        <block>
            <checkDependency>
                <installedComponent name="foo" version="1.0" />
            </checkDependency>
        </block>
        <catch>
        </catch>
    </try>
</installSteps>
```

Limiting Hosts for a Plug-In

The Sun N1 Service Provisioning System enables you to limit plug-in behavior to hosts that match certain criteria. There are three mechanisms that you can use to limit your hosts:

- Define a specific host type. The host type defines a base class of servers that is bound by a set of common attributes. For example, you might define a host type that identifies servers that are considered to be Solaris 10 global zones.
- Define a host set. The host set is a logical grouping of hosts that share one or more common attributes, such as physical location or functional group. Use a host set to quickly and easily update all hosts in the set. You can also use host sets to perform install-to-install comparisons.
- Define a host search. A host search queries the host database to provide a list of hosts whose attributes match those that the query specifies. You might use the host search to find all hosts that match a given host type or that run a certain application.

You define all three host limiters in the plug-in descriptor file, as shown in the following examples.

EXAMPLE 2–12 Host Type Definition in `plugin-descriptor.xml` File

The following example defines two host types for use with Solaris containers: one for a global zone and one for a local zone. The plug-in name is appended to the actual `hostType` name. When a user creates a host of type `com.sun.solaris#global_zone`, four attributes are provided, each attribute of which has a default value. The `com.sun.solaris#local_zone` host type, on the other hand, has no user-defined attributes associated with it.

```
<hostType name="global_zone"
    description="a physical host from which partitioned local zones can be created">
  <varList>
    <var name="local_zone_base_path" default="/export/zones"/>
    <var name="local_zone_connection_type" default="RAW"/>
    <var name="local_zone_port" default="1131"/>
    <var name="local_zone_advanced_params" default=" "/>
  </varList>
    </hostType>
    <hostType name="local_zone"
      description="a physical host that is created out of the larger global_zone"/>
```

EXAMPLE 2–13 Host Set Definition in `plugin-descriptor.xml` File

The following example defines a host set that contains global zones. The actual contents of the host set are provided when the referenced host search is performed.

EXAMPLE 2–13 Host Set Definition in `plugin-descriptor.xml` File *(Continued)*

```
<hostSet name="global_zones"
     description="Solaris global zones">
<hostSearchRef name="global_zones"/>
```

EXAMPLE 2–14 Host Search Definition in `plugin-descriptor.xml` File

The following example defines a host search to find all global zones. The search returns a result for any host that matches the following criteria:

- Is running the Solaris 10 operating system
- Has a host type of `com.sun.solaris#global_zone`
- Is running a remote agent
- Is a physical host, rather than a virtual host

```
<hostSearch name="global_zones" description="Solaris global zones">
  <criteriaList>
    <criteria name="sys.OS" pattern="SunOS"/>
    <criteria name="sys.OSVersion" pattern="5.10"/>
    <criteria name="sys.hostType" pattern="com.sun.solaris#global_zone"/>
  </criteriaList>
  <appTypeCriteria ra="true"/>
  <physicalCriteria physical="true"/>
</hostSearch>
```

Enabling Users to Browse and Export Files

The Sun N1 Service Provisioning System provides capabilities for you to enable users to include specific resources in their components. The browsing feature consists of two primary functions:

- Browse – Enables the user to traverse arbitrary, tree-like, filtered object hierarchies on the remote agent machines and to select an object in that tree.
- Export – Enables the user to check into the master server the selected object or collection of objects, possibly in a modified form.

For example, you could enable a user to traverse a file system, select a file, and check in the file through a component.

Browsing and exporting functionality are provided through the `com.sun.n1.sps.plugin.browse` and `com.sun.n1.sps.plugin.export` packages, as described in "Component APIs" on page 43.

Browsing and Exporting: Process Overview

From an external view, the browsing and exporting process is similar to the following sequence:

1. The user selects a component type to create a component. If the backing component of the selected type has *exporterClassName* defined, the browse and export user interface is launched.
2. The provisioning software obtains all the browser information in the `BrowserInfo` class. To obtain this information, the software calls the `getAvailableBrowsers` method of the `ComponentExporter` interface.
3. The provisioning software obtains the information about the `BrowserFactory` from `BrowserInfo` and instantiates it. From there, the provisioning software gets the `Browser` object.
4. From the `Browser` object, the software finds the `root` node by calling the `getNode()` method of `Browser`.
5. When the user selects a node and continues with the check-in process, the provisioning software calls into the `constructComponent` method of the `ComponentExporter` class which finally exports and checks-in the resource.

From the plug-in development perspective, a more detailed view of this process is similar to the following sequence:

1. The backing component of a component type defines a component variable named *exporterClassName*. The value of *exporterClassName* is the class that implements `com.sun.n1.sps.plugin.export.ComponentExporter`.
2. The `ComponentExporter` class method `getAvailableBrowsers` returns an array of `BrowserInfo` objects. These `BrowserInfo` objects have the following information about the browser:
   - Name of the system service
   - Variable *name* in the above system service. This variable will have the `BrowserFactory` class as its value
   - Variable *name* in the above system service. This variable will have the class path for the browser as its value.
   - The actual class path, if system service is not used for class path.
3. The `BrowserFactory` class has a method to get the browser which implements the `Browser` interface.
4. The `Browser` method `getNode(...)` finds the nodes of a tree. When used with a null argument, `getNode(...)` should give the `root` node.
5. The `ComponentExporter` class has another method to construct the component. This method is used once the actual browsing is done. The `constructComponent` method is passed a *ComponentMonitor* which is used to finally export and check-in the selected resource into the master server as part of the component.

Browse Function

`BrowserNode` is the class which implements the entire hierarchy tree functionality. This functionality is segmented into four key areas:

- Providing all the children of the node
- Providing the parent if the node
- Describing whether the node is a leaf node
- Providing other descriptions and properties related to the node

For more information about the classes and methods that you use to implement a browser for your plug-in, see "Browsing Function" on page 46.

Export Function

`ComponentExporter` is the class which enables a user to export a file to the master server, once he has browsed to it. For more information about the classes and methods that you use to implement the export feature for your plug-in, see "Exporting Function" on page 48.

Defining the Plug-In

To make the solution available for others to use, you wrap the plans, components, and component type definitions into a plug-in. To define the plug–in, you create an XML file that uses the `<plugin>` element and its children. For information about the `<plugin>` element, see Chapter 6, "Plug-In Description Schema," in *Sun N1 Service Provisioning System 5.1 XML Schema Reference Guide*.

EXAMPLE 2–15 Sample Plug-In Descriptor File

The following sample descriptor file is for the Solaris Zones plug-in.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin name="com.sun.solaris"
   description="Solaris plugin"   version="1.0"
   vendor="Sun Microsystems Inc"
   xmlns="http://www.sun.com/schema/SPS"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.sun.com/schema/SPS plugin.xsd"
   schemaVersion="5.1">
   <gui jarPath="gui/pluginUI.xml"/>
   <memberList>
       <folder name="/com/sun/solaris" description="Solaris plugin folder"/>
       <hostType name="global_zone"
```

EXAMPLE 2–15 Sample Plug-In Descriptor File *(Continued)*

```
        description="a physical host from which partitioned local zones can be created">
       <varList>
         <var name="local_zone_base_path" default="/export/zones"/>
         <var name="local_zone_connection_type" default="RAW"/>
         <var name="local_zone_port" default="1131"/>
         <var name="local_zone_advanced_params" default=" "/>
      </varList>       </hostType>
    <hostType name="local_zone"
         description="a physical host that is created out of the larger global_zone"/>
    <hostSearch name="global_zones" description="Solaris global zones">
      <criteriaList>
         <criteria name="sys.OS" pattern="SunOS"/>
         <criteria name="sys.OSVersion" pattern="5.10"/>
         <criteria name="sys.hostType" pattern="com.sun.solaris#global_zone"/>
      </criteriaList>
      <appTypeCriteria ra="true"/>
      <physicalCriteria physical="true"/>
    </hostSearch>
    <hostSet name="global_zones" description="Solaris global zones">
      <hostSearchRef name="global_zones"/>
    </hostSet>
    <component jarPath="fiji/components/com/sun/solaris/zone_util.tar.xml">
      <resource jarPath="fiji/resources/com/sun/solaris/zone_util.tar"
          name="/com/sun/solaris/zone_util.tar"/>
    </component>
    <component jarPath="fiji/components/com/sun/solaris/N1GridContainer.xml"
      majorVersion="true">
    </component>
    <component jarPath="fiji/components/com/sun/solaris/ZoneSS.xml">
      <systemService name="zoneSS"
        description="the Solaris zone system service"/>
    </component>   </memberList>
</plugin>
```

Defining an Interface to the Plug-In

One of the key activities in creating a solution that you can provide to others or distribute across your environment is defining an interface to your solution within the Sun N1 Service Provisioning System browser interface. To define the interface, you create an XML file that uses the `<plguinUI>` element and its children. For information about the `<pluginUI>` element, see Chapter 7, "Plug-In User Interface Schema," in *Sun N1 Service Provisioning System 5.1 XML Schema Reference Guide*.

EXAMPLE 2–16 Sample Plug-In Interface File

The following sample plug-in interface file `pluginUI.xml` is for the Solaris Zones plug-in.

```
<?xml version="1.0" encoding="UTF-8"?>
<pluginUI menuItem="Solaris"   xmlns="http://www.sun.com/schema/SPS"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.sun.com/schema/SPS pluginUI.xsd"
   schemaVersion="5.1">
   <icon jarPath="gui/solaris.gif"/>
   <customPage name="Solaris">
    <section title="Solaris specific tasks"
       description="create and manage Solaris specific components...">
       <entry title="Solaris Zones" description="create and manage zones">
         <action text="list" toolTip="list of installed zones">
           <compWhereInstalled path="/com/sun/solaris" name="N1GridContainer"/>
         </action>
         <action text="create and manage" toolTip="create and manage zones">
           <compDetails path="/com/sun/solaris" name="N1GridContainer" />
         </action>
       </entry>
     </section>
   </customPage>
</pluginUI>
```

Packaging the Solution

To enable others to use your solution or to make it available for easy distribution within your own environment, you package your solution in a Java Archive (JAR) file. The contents and instructions for interpreting the contents of the JAR file are contained in an optionally signed `plugin-descriptor.xml` file located in the top level directory of the JAR. The syntax of the plug-in descriptor is specified using XML Schema as per the May 2, 2001 W3C Recommendation (`http://www.w3.org/TR/2001/xmlschema-0-20010502/`). The schema can be used in conjunction with a validating parser to determine the syntactical validity of a plug-in. For information about the plug-in descriptor file, see "Defining the Plug-In" on page 38.

To create the JAR file, you use the JAR utility. The JAR utility uses similar options to the standard UNIX tar utility.

To create a JAR file, use the following command from the root directory that contains all the plug-in files: `jar cf` *jarfile inputfiles*

where:

- The `c` option creates a new archive named *jarfile* that contains the files and directories specified by *inputfiles*.
- The `f` *jarfile* option specifies the name of the file to be created.
- *inputfiles* identifies the files or directories to be included in the JAR file. You can provide a list of file and directory names separated by spaces, or you can use the asterisk (`*`) character to include all the files in the current directory. All directories are processed recursively.

EXAMPLE 2–17 Creating a JAR File That Contains Subdirectories

If you have subdirectories, you can combine them into a single JAR file, as shown in the following example command:

```
% jar cvf myplugin.jar *
added manifest
ignoring entry META-INF/
ignoring entry META-INF/MANIFEST.MF
adding: components/(in = 0) (out= 0)(stored 0%)
adding: components/com/(in = 0) (out= 0)(stored 0%)
adding: components/com/sun/(in = 0) (out= 0)(stored 0%)
adding: components/com/sun/myplugin/(in = 0) (out= 0)(stored 0%)
adding: components/com/sun/myplugin/mycomponent.xml(in = 6224) (out= 1182)(deflated 81%)
adding: components/com/sun/myplugin/myothercomponent.xml(in = 1291) (out= 507)(deflated 60%)
adding: components/com/sun/myplugin/mycomponenttype.xml(in = 940) (out= 470)(deflated 50%)
adding: resources/(in = 0) (out= 0)(stored 0%)
adding: resources/com/(in = 0) (out= 0)(stored 0%)
adding: resources/com/sun/(in = 0) (out= 0)(stored 0%)
adding: resources/com/sun/solaris/(in = 0) (out= 0)(stored 0%)
adding: resources/com/sun/solaris/zone_util.tar(in = 20480) (out= 4232)(deflated 79%)
adding: gui/(in = 0) (out= 0)(stored 0%)
adding: gui/pluginUI.xml(in = 861) (out= 407)(deflated 52%)
adding: gui/solaris.gif(in = 1622) (out= 1627)(deflated 0%)
adding: plugin-descriptor.xml(in = 1990) (out= 707)(deflated 64%)
%
```

To verify the files in the JAR file, use the following command:

```
% jar tf mypluin.jar
META-INF/MANIFEST.MF
fiji/
fiji/components/
fiji/components/com/
fiji/components/com/sun/
fiji/components/com/sun/solaris/
fiji/components/com/sun/solaris/N1GridContainer.xml
fiji/components/com/sun/solaris/ZoneSS.xml
fiji/components/com/sun/solaris/zone_util.tar.xml
fiji/resources/
fiji/resources/com/
fiji/resources/com/sun/
fiji/resources/com/sun/solaris/
fiji/resources/com/sun/solaris/zone_util.tar
```

EXAMPLE 2–17 Creating a JAR File That Contains Subdirectories *(Continued)*

```
gui/
gui/pluginUI.xml
gui/solaris.gif
plugin-descriptor.xml
```

Testing the Solution

Before you make your solution available across your environment or for others to use, you should test the solution. The following ideas might help you decide what to test:

- Use an XML parser to validate the `plugin-descriptor.xml` file against the `plugin.xsd` schema.
- Use an XML parser to validate the `pluginUI.xml` file against the `pluginUI.xsd` schema.
- Make sure that any Java code that you use builds cleanly and works as expected.
- Import your finished plug-in to the Sun N1 Service Provisioning System product. Check for errors and make sure that the customized user interface, if it exists, renders as expected, and that all links on the customized page work as expected.
- Verify that you can delete your plug-in after a successful import.

CHAPTER 3

Using the Application Programming Interface

The Sun N1 Service Provisioning System includes a Java-based application programming interface (API) that you can use to further extend the functionality of the system. This chapter explains how you can use the classes and methods of the API. Detailed syntax for each class and method is provided in the Javadoc™ information included with the provisioning system. The API provides several types of additional functionality:

- Ability to create component-specific features, such as browseability, as described in "Component APIs" on page 43
- Ability to execute Java code, as described in "`execJava` API" on page 50

Component APIs

The Java-based component APIs enable you to provide export and browse functionality for your plug-ins. You can enable users to be able to browse through directory structures and export files from within the Sun N1 Service Provisioning System browser interface.

`com.sun.n1.sps.componentdb`
This package provides two interfaces for working with the component database:

- `InstallMode` – A strongly typed enumeration of component install modes
- `InstallMode.Factory` – A factory interface for `InstallMode` enums

`com.sun.n1.sps.plugin`
This package contains one interface and three classes to support general plug-in related functionality:

- `AgentContext` – This interface publishes services available to the plug-in code on a remote agent.
- `Logger` – Use this high level wrapper class for logging in service provisioning projects.

- `PluginMessage` – Instances of this class are used to internationalize messages within the plug-in implementations.
- `PluginException` – Class representing any exception that uses a `PluginMessage` for its message resolution.

`com.sun.n1.sps.plugin.browse`
This package contains five interfaces and four classes that specify browse functionality:

- `Browser` – This interface defines the set of functionality that any resource handler that wants to support browsing must export.
- `BrowserDisplay` – This interface is used by the UI Browsing portion of the hierarchy manager to make the display more informative and correct.
- `BrowserFactory` – This interface provides the interface for the loader to use to obtain an actual instance of the appropriate browser.
- `BrowserFilter` – This interface describes how nodes can be filtered according to certain criteria.
- `BrowserNode` – This interface defines the functionality for a browsable hierarchy node.
- `BrowserContext` – This class provides a container for the client to set initial parameters for a browsing session.
- `BrowserInfo` – This class describes the browser that is appropriate for display in the user interface and retrieval of actual instance from within the system.
- `BrowserNodeBase` – This class provides a default implementation for the `BrowserNode` interface.
- `BrowserException` – This class identifies typed exceptions to be thrown from within browsing sessions.

More information and examples for the browsing functionality are provided in "Browsing Function" on page 46.

`com.sun.n1.sps.plugin.export`
This package contains seven interfaces and one exception class for specifying component definition and creation functionality:

- `ComponentExporter` – All plug-ins must implement this base interface to construct a component from a browse process.
- `ComponentMonitor` – Monitor created by the system that manages the component creation process for a given component.
- `ComponentToken` – The token to represent a component for purposes of adding a contained component to a `CompositeComponentMonitor`.
- `CompositeComponentMonitor` – The monitor for a component that contains other components.
- `ResourceProcessor` – Allows for introspection of a resource.
- `SimpleComponentMonitor` – Component monitor for components that contain a resource.
- `SystemData` – Gives access to variables defined by various persistent system objects related to the current export and browse operations.
- `ComponentExportException` – Strongly typed exception for use with errors related to component export.

More information and examples for the export functionality are provided in "Exporting Function" on page 48.

`com.sun.n1.sps.resource`

This package contains seven interfaces and one exception class for managing resources:

- `CheckInMode` – A strongly typed enumeration for representing check in modes
- `CheckInMode.Factory` – A factory interface for `CheckInMode` enumerations
- `ResourceEntry` – Represents an entry within a resource
- `ResourceEntryIterator` – An iterator for `ResourceEntry` objects
- `ResourceManifest` – A manifest that describes the resource
- `ResourceType` – A strongly typed enumeration for representing resource types
- `ResourceType.Factory` – A factory interface for `ResourceType` enumerations
- `ResourceException` – Typed exceptions thrown from error conditions related to resources

`com.sun.n1.util`

This package provides one interface and three additional packages for managing utilities:

- `RPCSerializable` – This interface marks objects that can be serialized by RPC.
- `com.sun.n1.util.enum` – This package contains two interfaces and one exception class:
  - `Enum` – An interface for strongly typed enumerations
  - `Enum.Factory` – Enables a client to look up all values defined for a particular `Enum` subclass, and also to look up a particular value by its integer or string value
  - `NoSuchEnumException` – Exception class indicating that an enumeration lookup failed
- `com.sun.n1.util.message` – This package contains two interfaces:
  - `Severity` – A strongly typed enumeration for representing severities associated with messages
  - `Severity.Factory` – A factory interface for Severity enumerations
- `com.sun.n1.util.vars` – This package contains three interfaces and three classes:
  - `DisplayMode` – A strongly-typed enumeration of display modes
  - `DisplayMode.Factory` – A factory interface for `DisplayMode` enumerations
  - `VariableSettingsSource` – Defines the interface for objects that can be used as a source of variable settings
  - `PromptParam` – A parameter that includes information about the structure of a prompt, including a textual prompt and a display mode
  - `PromptParamList` – A list of `PromptParam` objects

- `VariableSettingsHolder` – An implementation of the `VariableSettingsSource` interface that can be used to specify variable name-value pairs

Browsing Function

The `com.sun.n1.sps.plugin.browse` package contains five interfaces and four classes that specify browse functionality:

- `Browser` – Any resource handler must use this base interface to support browsing functionality.
- `BrowserDisplay` – This interface is used by the Browsing portion of the hierarchy manager to make the display more informative and correct.
- `BrowserFactory` – This interface provides the interface for the loader to use to obtain an actual instance of the appropriate browser.
- `BrowserFilter` – This interface describes how nodes can be filtered according to certain criteria.
- `BrowserNode` – This interface defines the functionality for a browsable hierarchy node.
- `BrowserContext` – This class provides a container for the client to set initial parameters for a browsing session.
- `BrowserInfo` – This class describes the browser that is appropriate for display in the user interface and retrieval of actual instance from within the system.
- `BrowserNodeBase` – This class provides a default implementation for the `BrowserNode` interface.
- `BrowserException` – This class identifies typed exceptions to be thrown from within browsing sessions.

Browser API Implementation

The Browser implementation includes the following key API segments:

`BrowserFilter[] getAvailableFilters()`
Returns the different filters this browser supports. Use the `BrowserFilter` interface to filter `BrowserNodes` based on particular criteria, for example, filter all files to show just `*.tmp` files.

`BrowserDisplay getDisplay()`
Gets the display properties object to be used with this browser.

`BrowserNode getNode(java.lang.String location)`
Returns a node in the hierarchy this browser represents.

`void setFilterName(java.lang.String name)`
Specifies the filter to be used while browsing.

`BrowserNode` Class

The `BrowserNode` class implements the entire hierarchy tree functionality. This functionality is segmented into four key areas:

- Providing all the children of the node
- Providing the parent if the node
- Describing whether the node is a leaf node
- Providing other descriptions and properties related to the node

`BrowserFactory` Interface

The `BrowserFactory` interface provides the interface for the `HierarchyBrowserLoader` to obtain an actual instance of the appropriate `HierarchyBrowser`.

To define a class which implements the `BrowserFactory` interface, use an API call similar to the following example:

`Browser getBrowser(BrowserContext` *bContext*`,AgentContext` *aContext*`)`

where:

- *bContext* is the context retrieved from the component exporter that specified this browser.
- *aContext* is the context supplied for the agent in case native libraries must be loaded

The `BrowserFactory` implementation defines a `getBrowser` method with the system-supplied `BrowserContext` object and `AgentContext` objects as parameters.

In the backing component of the component type, declare the fully qualified class name of the browser factory in the `browserClass` variable. The following code fragment defines two browser factories for a backing component:

```
<var
 access="PRIVATE"
 name="EJBFileSystemBrowser"
 default="com.raplix.rolloutexpress.plugins.weblogic.hierarchies.ejb.EJBFileBrowserFactory"
/>
<var
 access="PRIVATE"
 name="EJBDomainBrowser"
 default="com.raplix.rolloutexpress.plugins.weblogic.hierarchies.ejb.EJBDomainBrowserFactory"
/>
```

Sample Code for Browsing Function

EXAMPLE 3–1 Browser Filter

The following example filters all files of the name `*.tmp`:

EXAMPLE 3–1 Browser Filter *(Continued)*

```
public class TmpFilter implements BrowserFilter, ExampleFilter {

    public String getName() {
        return "tmpFilter";
    }
    public String getDescription() {
        return "show only *.tmp files";
    }
    public boolean filter(ExampleBrowserNode node) {
        return node.getLocalName().endsWith(".tmp");
    }

}
```

Exporting Function

The `com.sun.n1.sps.plugin.export` package contains seven interfaces and one exception class for specifying component definition and creation functionality:

- `ComponentExporter` – All plug-ins must implement this base interface to construct a component from a browse process.
- `ComponentMonitor` – Monitor created by the system that manages the component creation process for a given component.
- `ComponentToken` – The token to represent a component for purposes of adding a contained component to a `CompositeComponentMonitor` interface.
- `CompositeComponentMonitor` – The monitor for a component that contains other components.
- `ResourceProcessor` – Allows for introspection of a resource.
- `SimpleComponentMonitor` – Component monitor for components that contain a resource.
- `SystemData` – Gives access to variables defined by various persistent system objects related to the current export and browse operations.
- `ComponentExportException` – Strongly typed exception for use with errors related to component export.

`ComponentExporter` Process

To enable an export function, use a process similar to the following sequence:

1. In the backing component of the component type, declare the fully qualified class name of the `componentExporter` in the *exporterClass* variable.

```
<varList>
  <var name="exporterClassName"
    default="com.sun.n1.sps.pluginimpl.sample.export.StaticCompExporter"/>
```

```
</varList>
```

2. Define a class which implements the `ComponentExporter` interface.

   `ComponentExporter` calls the various methods on the `ComponentMonitor` input argument to build the component. These methods might include `addComponentVar`, `addSourceInfoParam`, `setComponentDescription`, and `setComponentLabel`.

   `ComponentExporter` can also call *get* routines to obtain information from the `ComponentMonitor`. These get routines might include `getPluginComponentVars`, `getPluginHostVars`, `getActiveBrowser`, `getSourceInfoParam`, and `getLocation`.

   `ComponentExporter` can also call `exportResource` to call into control blocks to execute component type-specific functionality for exporting the component.

3. After constructing the component, the `ComponentExporter` can call `setResource` to set a physical resource to be bundled in the component, completing the export process.

`ComponentExporter` Example

EXAMPLE 3–2 `ComponentExporter`

```
public class  implements ComponentExporter {

    public ExampleExporter() {

    }

    public BrowserContext getBrowserContext() {
        return new BrowserContext();
    }

    public BrowserInfo[] getAvailableBrowsers() {
        return new BrowserInfo[] {
            new BrowserInfo("example",             //relevant comp type
                            "Example Browser",     //browser ui display name
                            "example ss",          //relevant ss
                            null,                  //valid for all platforms
                            null,                  //no host set restriction
                            new PromptParamList()) //no checkin params
        };
    }

    public String getBrowserClassPath(BrowserInfo browser) {
        return null;
    }
public void constructComponent(ComponentMonitor mon)
        throws ComponentExportException {

        //It's the responsibility of the infrastructure to  identify the type
        //of component and construct the component with the appropriate monitor
```

EXAMPLE 3–2 `ComponentExporter` *(Continued)*

```
        SimpleComponentMonitor sMon = (SimpleComponentMonitor)mon;

      sMon.setComponentDescription("This is an example component");
        sMon.setComponentLabel("What the hell is a label for?");

        sMon.setResource(ResourceType.FILE,  //our sample type is a file
                         sMon.getLocation(), //get the location specified
                         false,              //do not use differential checkin
                         false,              //not a config template
                         false,              //file->symlinks meaningless
                         true,               //capture permissions
                         null,               //file->checkinmode meaningless
                         null);              //no special processing of rsrc
    }
}
```

execJava API

`execJava` functionality is provided through the XML schema for plans and components. Through the XML, you can execute a piece of Java code as needed. In addition, `execJava` also exists as an API.

Both preflight and actual behavior may be specified. The classes are typically deployed using a JAR resource of a component. For more information about the `execJava` classes, methods, and interfaces, see the JavaDoc software.

```
<execJava
className= classname of the executor factory class
class Path=...
>
```

The `execJava` API is contained in the `com.sun.n1.sps.plugin.execJava` package. The `execJava` API consists of five interfaces and two exception classes:

`ActualExecJavaContext`
: This interface publishes the services available to the `execJava` implementations when they are invoked during the deployment or actual phase of the execution.

`ExecJavaContext`
: This interface provides an execution context to an `execJava` implementation that is common to both the preflight and actual run levels.

`Executor`
: This interface is implemented by classes that need to execute code on the agent through `execJava`

`ExecutorFactory`
: This interface is part of the infrastructure to execute arbitrary code on the remote agent using `execJava` steps.

`PreflightExecJavaContext`
: This interface publishes the services available to the `execJava` implementations when they are invoked during the preflight phase of the execution.

`ExecutionException`
: Instances of `ExecutionException` are used to flag failure or warnings from `execJava` invocations.

`ExecutionTimeoutException`
: Instances of this exception are thrown when `execJava` execution is timed out.

`ExecutorFactory` Interface

The `ExecutorFactory` interface is used to obtain the preflight and actual executor instances for a particular step:

```
Executor getActualExecutor(AgentContext callContext)
Executor getPreflightExecutor(AgentContext callContext)
```

The call context passed between preflight and actual execution steps need not be the same.

`AgentContext` Method

The `AgentContext` method provides an invocation context on a particular remote agent.

```
VariableSettingsHolder getVariables()
    // Returns the variables passed to the execJava step using <argList>

PrintStream getStandardOutput()
PrintStream getStandardError()
InputStream getStandardInput()
File getWorkingDir()
```

`Executor` Interface

The `Executor` interface provides an entry point that is used to execute the step body:

```
void execute() throw ExecutionException
```

Execution output and error output are written into the stdout and stderr streams of the associated agent context. Input is read from the input stream of the associated agent context. Errors are reported by calling an instance of the `ExecutionException` class.

execJava Examples

EXAMPLE 3–3 execJava in Java Code

```
public class StopServerFactory extends WLFactoryBase {

    public static final String TARGET = "serverName";

    public Executor
        getActualExecutor(AgentContext inAgentContext, ActualExecJavaContext inContext)
        {

        VariableSettingsSource variableSettings = inContext.getVariableSettings();
        String target = variableSettings.getVarValue(TARGET);
        return new StopServerExecutor(getConnect(variableSettings), target);
    }

    public VariableSettingsSource getParams() {
        VariableSettingsHolder list = getWLParams();
        list.setVarValue(TARGET, null);
        return list;
    }

}
public class StopServerExecutor implements Executor {
    private WLConnect mConnect;
    private String mTarget;

    /**
     *
     **/
    public StopServerExecutor(WLConnect connect, String target) {
           mConnect = connect;
           mTarget = target;
    }

    /**
     *
     **/
    public void execute() throws ExecutionException {

  try {
    WLAdminServer server = new WLAdminServer(mConnect);
    server.stopServer(server.getServer(mTarget));
    }
catch (Exception e) {
    throw new ExecutionException
    (new PluginMessage(WLPluginHierarchyException.MSG_WEBLOGIC_ERROR), e);
    }
  }
}
```

EXAMPLE 3–4 Another execJava Code Sample

```
public class SampleExecutorFactory implements ExecutorFactory
{
    public Executor getActualExecutor(AgentContext inAgentContext,
```

EXAMPLE 3–4 Another `execJava` Code Sample *(Continued)*

```
            ActualExecJavaContext inActualExecJavaContext)
    {
        return new EnvParamSettingActualExecutor(inActualExecJavaContext);
    }

    public Executor getPreflightExecutor(AgentContext inAgentContext,
            PreflightExecJavaContext inPreflightExecJavaContext)
    {
        return new EnvParamSettingPreflightExecutor(inPreflightExecJavaContext);
    }

    public VariableSettingsSource getParams()
    {
        VariableSettingsHolder params = new VariableSettingsHolder();
        params.setVarValue(PARAM_NAME, "");
        params.setVarValue(PARAM_VALUE, "");
        return params;
    }

    public static final String PARAM_NAME = "nameParam";
    public static final String PARAM_VALUE = "valueParam";
}
public class EnvParamSettingPreflightExecutor implements Executor
{
    VariableSettingsSource mVars;
    public EnvParamSettingPreflightExecutor
           (PreflightExecJavaContext inPreflightExecJavaContext)
    {
        mVars = inPreflightExecJavaContext.getVariableSettings();
    }

    public void execute() throws ExecutionException
    {
        String propName = mVars.getVarValue(SampleExecutorFactory.PARAM_NAME);
        if("".equals(propName)) {
            throw new ExecutionException(new PluginMessage("sample.noNameParam"));
        }

        String propValue=System.getProperty(propName);
        if(!(propValue == null || "".equals(propValue))) {
            // property already set, error out
            throw new ExecutionException(new PluginMessage("sample.propAlreadySet",
                                       new String[]{propName, propValue}));
        }
    }
}

public class EnvParamSettingActualExecutor implements Executor
{
    VariableSettingsSource mVars;
    public EnvParamSettingActualExecutor(ActualExecJavaContext inCtx)
    {
        mVars  = inCtx.getVariableSettings();
```

EXAMPLE 3–4 Another `execJava` Code Sample *(Continued)*

```
    }

    public void execute() throws ExecutionException
    {
        String propName = mVars.getVarValue(SampleExecutorFactory.PARAM_NAME);
        String propValue = mVars.getVarValue(SampleExecutorFactory.PARAM_VALUE);
        System.setProperty(propName, propValue);
        if(Logger.isDebugEnabled(this)) {
            Logger.debug("Setting prop "+propName + " to " + propValue, this);
        }
        System.out.println("Setting prop "+propName + " to " + propValue);
    }
}
```

APPENDIX **A**

Example Plug-In

This appendix contains example code for a simple plug-in. This sample is based on the Linux plug-in, which is provided with the Sun N1 Service Provisioning System 5.1 software.

Description of the Sample Plug-In

The sample plug-in includes the following files and directories in the `com.sun.linux_1.1.jar` file:

```
META-INF/
META-INF/MANIFEST.MF
plugin-descriptor.xml
1.1/
1.1/components/
1.1/components/com/
1.1/components/com/sun/
1.1/components/com/sun/linux/
1.1/components/com/sun/linux/RPM CT.xml
1.1/resources/
1.1/resources/com/
1.1/resources/com/sun/
1.1/resources/com/sun/linux/
1.1/resources/com/sun/linux/plugin-linux.jar
1.1/plans/
```

The sample plug-in does not include a `pluginUI.xml` because the Linux plug-in does not provide a customized interface page. For an example of a plug-in interface file, see Example 2–16.

Plug-In Descriptor File

The `plugin-descriptor.xml` defines the sample plug-in. Look at the following items in the example below:

- Most attributes to the `<plugin>` element use standard values. The two exceptions are the *name* and *version* attributes.
- The `<dependencyList>` element tells you that the `system` plug-in, version 1.0 is required for the sample plug-in to work correctly. The `system` plug-in is a core part of the Sun N1 Service Provisioning System software and should always exist.
- The `<folder>` element creates a folder in which Linux objects can be stored.
- The `<component>` element defines a component type whose backing component is `1.0/components/com/sun/linux/RPM CT.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin xmlns="http://www.sun.com/schema/SPS"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.sun.com/schema/SPS ../plugin.xsd"
  name="com.sun.linux"
  vendor="Sun Microsystems"
  version="1.1"
  schemaVersion="5.1">

  <dependencyList>
    <pluginRef name="system" version="1.0"/>
  </dependencyList>

  <memberList>
    <!-- Folders -->

    <folder name="/com/sun/linux"
        description="contains linux plugin objects"/>

    <!-- Components -->

    <component jarPath="1.0/components/com/sun/linux/RPM CT.xml">
      <componentType name="rpm file"
         description="the active component type for rpm files"
         group="any UNIX"
         order="000700-000100-000200"
         indentLevel="1"/>
    </component>

  </memberList>
</plugin>
```

Components

The sample plug-in contains the `RPM CT.xml` file in the `components` directory. This file defines the backing component for the `rpm` component type, and is not expected to be used directly as a component itself. Look at the following items in the example below:

- The *path*, *name*, *description*, and *platform* attributes to the `<component>` element provide specific information about the component type.
- The `<extends>` element tells you that the `RPM` component type extends the features available in the `system#CR Simple Base` component type.
- The `<varlist>` element defines several variables that enable the user to customize components based on this component type.
- The `<installList>` element calls the `<exceNative>` step to run the Linux command to install the RPM files.
- The `<uninstallList>` element calls the `<exceNative>` step to run the Linux command to uninstall the RPM files.

```
<?xml version="1.0" encoding="UTF-8"?>
<component xmlns="http://www.sun.com/schema/SPS"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    version="5.1"
    xsi:schemaLocation="http://www.sun.com/schema/SPS
                    component.xsd"
    modifier="ABSTRACT"
    path="/com/sun/linux"
    name="RPM CT"
    description="RPM Installer"
    platform="system#Red Hat Linux - any version">

    <extends>
        <type name="system#CR Simple Base"/>
    </extends>

    <varList>
        <var name="filterName" default="rpmOnly" access="PRIVATE"/>
        <var name="filterDescription"
            default="show RPM file types only (.rpm)"
            access="PRIVATE"/>
        <var name="filterExtensions" default=".rpm" access="PRIVATE"/>
        <var name="rpmCmd" default="rpm"/>
        <var modifier="FINAL" name="installDeployMode" default="REPLACE"/>
        <var name="installDiffDeploy" default="TRUE"/>
        <var access="PRIVATE" name="exporterClassName"
             default="com.sun.n1.sps.pluginimpl.system.export.FilesystemExporter"/>
        <var access="PRIVATE" name="canBeConfigTemplate" default="FALSE"/>
    </varList>

    <installList>
```

```
        <installSteps name="default">
            <deployResource/>
            <execNative userToRunAs="root">
                <exec cmd=":[rpmCmd]">
                    <arg value="-i"></arg>
                    <arg value=":[sys.rsrcInstallPath]"></arg>
                </exec>
            </execNative>
        </installSteps>
    </installList>

    <uninstallList>
        <uninstallSteps name="default">
            <execNative userToRunAs="root">
                <shell cmd="sh -c">
                     <![CDATA[:[rpmCmd] -e `:[rpmCmd]
                        -qp :[sys.rsrcInstallPath]
                        -qf '%{NAME}' 2> /dev/null`]]>
                </shell>
            </execNative>
            <call blockName="deleteFile">
                <argList absPath=":[sys.rsrcInstallPath]"/>
                <systemService name="system#core services"/>
            </call>
        </uninstallSteps>
    </uninstallList>

</component>
```

Index

D

E

F

G

H

I

V

X