HP Service Activator

Developing Plug-Ins and Compound Tasks

Edition: V62-1A

for Microsoft Windows® Server 2008 R2, HP-UX i v3, and Red Hat Enterprise Linux 6.4 operating systems

Manufacturing Part Number: None

October 14, 2013

© Copyright 2001-2013 Hewlett-Packard Development Company, L.P.

Legal Notices

Warranty.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

A copy of the specific warranty terms applicable to your Hewlett-Packard product can be obtained from your local Sales and Service Office.

Restricted Rights Legend.

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013.

Hewlett-Packard Company
United States of America

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19(c)(1,2).

Copyright Notices.

©Copyright 2001-2013 Hewlett-Packard Development Company, L.P., all rights reserved.

No part of this document may be copied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this material is subject to change without notice.

Trademark Notices.

Java™ is a registered trademark of Oracle and/or its affiliates.

Linux is a U.S. registered trademark of Linus Torvalds.

Microsoft® is a U.S. registered trademark of Microsoft Corporation.

Red Hat® Enterprise Linux® is a registered trademark of Red Hat, Inc.

EnterpriseDB® is a registered trademark of EnterpriseDB.

Postgres Plus® Advanced Server is a registered trademark of EnterpriseDB.

Oracle® is a registered trademark of Oracle and/or its affiliates.

UNIX® is a registered trademark of the Open Group.

Windows® and MS Windows® are U.S. registered trademarks of Microsoft Corporation.

All other product names are the property of their respective trademark or service mark holders and are hereby acknowledged.

Document id: p158-pd001408

Contents

1. Understanding and Using Service Activator Plug-ins and Compound Tasks

Understanding Plug-ins. 14
Atomic Tasks 14
Packaging a Plug-In 15
Understanding the Resource Manager 15
Understanding the Plug-in Life-Cycle 22
Understanding Plug-in Archives 24
Understanding the Plug-in Context 26
Understanding Compound Tasks 27
Using the Plug-in Library 28
Accessing Plug-in Documentation 28
Using Plug-in Classes 28

2. Understanding and Using Service Builder

Using Service Builder to Create Plug-ins 32
Starting Service Builder 32
Creating a Project 32
Creating a Plug-in 32
Adding Atomic Tasks 37
Using Common Source Files in Multiple Plug-ins 39
Adding Java Classes (Source Files) 40
Adding Scripts 42
Adding Files 43
Including Preprovisioning Scripts and Files 44
Adding a Library 45
Viewing General Properties 46
Configuration Parameters 48
Compiling a Plug-in 49
Deploying a Plug-in 51
Testing an Atomic Task 52
Generating Plug-in Documentation 53
Using Service Builder to Create Compound Tasks 56
Creating Compound Tasks 56
Importing and Exporting a Compound Task 62
Deploying a Compound Task 62
Testing a Compound Task 63
Documenting a Compound Task 63
Maintaining Consistency Between Deployed Tasks 65
Configuring Authentication or Authorization 66
Using Service Builder to Manage Plug-in Archives and Compound Tasks 67
Setting Service Builder Configuration 69
Using Service Builder from the Command Line 70

3. Creating Customized Plug-ins and Compound Tasks

Creating Plug-ins: Advanced Tips 74
Plug-in Java Class 74

Contents

Executing Scripts and Commands . . . 78
Saving Data in the Database . . . 79
Reading Data from the Database . . . 81
Understanding the Plug-in Deployment Descriptor (Manifest) . . . 82
Packaging and Deploying a Plug-in . . . 83
The Difference Between PAR Deployment and Script Deployment . . . 83
Creating Compound Tasks Manually: Advanced Tips . . . 85

A. Generic CLI
Generic CLI Plug-in . . . 88

B. NNM Liaison
NNMLiaison Plug-in . . . 92

C. Generic LDAP
GenericLDAP Plug-in . . . 94

D. Generic HTTP Plugin
Generic HTTP Plug-in . . . 96

In This Guide

This guide describes the process of developing plug-ins and compound tasks for HP Service Activator.

Audience

The audience for this guide is the Systems Integrator (SI). The SI has a combination of some or all of the following:

- Understands and has a solid working knowledge of:
  - UNIX® commands
  - Windows® system administration
- Understands networking concepts and language
- Understands database programming and management
- Ability to program in Java™ and XML
- Understands security issues

Conventions

The following typographical conventions are used in this guide.

Font	What the Font Represents	Example
Italic	Book or manual titles, and manpage names	Refer to the *HP Service Activator — Workflows and the Micro-Workflow Manager* and the *Javadocs* manpage for more information.
	Provides emphasis	You *must* follow these steps.
	Specifies a variable that you must supply when entering a command	Run the command: `InventoryBuilder` <*sourceFiles*>
	Parameters to a method	The *assigned_criteria* parameter returns an ACSE response.
Bold	New terms	The **distinguishing attribute** of this class...
`Computer`	Text and items on the computer screen	The system replies: `Press Enter`
	Command names	Use the `InventoryBuilder` command ...
	Method names	The `get_all_replies()` method does the following...
	File and directory names	Edit the file `$ACTIVATOR_ETC/config/mwfm.xml`
	Process names	Check to see if `mwfm` is running.
	Window/dialog box names	In the `Test and Track` dialog...
	XML tag references	Use the `<DBTable>` tag to...
`Computer Bold`	Text that you must type	At the prompt, type: **`ls -l`**
Keycap	Keyboard keys	Press **Return**.
`[Button]`	Buttons on the user interface	Click `[Delete]`. Click the `[Apply]` button.

Font	What the Font Represents	Example
`Menu Items`	A menu name followed by a colon (:) means that you select the menu, then the item. When the item is followed by an arrow (->), a cascading menu follows.	Select `Locate:Objects->by Comment`

Install Location Descriptors

The following names are used throughout this guide to define install locations.

Descriptor	What the Descriptor Represents
$ACTIVATOR *$ACTIVATOR_OPT*	The base install location of Service Activator. The UNIX location is `/opt/OV/ServiceActivator` The Windows location is *<install drive>*`:\HP\OpenView\ServiceActivator\`
$ACTIVATOR_ETC	The install location of specific Service Activator files. The UNIX location is `/etc/opt/OV/ServiceActivator` The Windows location is *<install drive>*`:\HP\OpenView\ServiceActivator\etc\`
$ACTIVATOR_VAR	The install location of specific Service Activator files. The UNIX location is `/var/opt/OV/ServiceActivator` The Windows location is *<install drive>*`:\HP\OpenView\ServiceActivator\var\`
$ACTIVATOR_BIN	The install location of specific Service Activator files. The UNIX location is `/opt/OV/ServiceActivator/bin` The Windows location is *<install drive>*`:\HP\OpenView\ServiceActivator\bin\`
$JBOSS_HOME	The install location for JBoss. The UNIX location is `/opt/HP/jboss` The Windows location is *<install drive>*`:\HP\jboss`
$JBOSS_DEPLOY	The install location of the Service Activator J2EE components. The UNIX location is `/opt/HP/jboss/standalone/deployments` The Windows location is *<install drive>*`:\HP\jboss\standalone\deployments`
$JBOSS_EAR_LIB	The location for libraries (Java *.jar files) to be executed by the HPSA engine (workflow manager and resource manager). The UNIX location is `/opt/HP/jboss/standalone/deployments/hpsa.ear/lib` The Windows location is *<install drive>*`:\HP\jboss\standalone\deployments`\hpsa.ear\lib
$ACTIVATOR_DB_USER	The database user name you define. Suggestion: `ovactivator`
$ACTIVATOR_SSH_USER	The Secure Shell user name you define. Suggestion: `ovactusr`

1 Understanding and Using Service Activator Plug-ins and Compound Tasks

This chapter contains information about constructing plug-ins and compound tasks, and their behavior in Service Activator.

Understanding Plug-ins

A **plug-in** is a Java class that contains methods to perform configuration tasks related to a specific type of software or hardware component. A plug-in should be able to perform its operations on any component of that type as long as the target is reachable from the Service Activator server and has the necessary prerequisites to enable the communication from Service Activator. Typically this communication is via Secure Shell, but other communication mechanisms are possible.

For information about where plug-ins fit in the overall Service Activator solution, see in *HP Service Activator—System Integrator's Overview*.

Service Activator comes complete with a number of plug-ins. You can find more details about these plug-ins in the *$ACTIVATOR*/docs/plugins directory.

You can write your own plug-ins. They are created manually or using the Service Builder tool. The following chapters in this book describe how to develop new plug-ins.

While nothing prevents you from creating a plug-in operating on multiple types of components, there is, generally, no benefit from such grouping. The plug-in shipped with Service Activator operates on a single type of target.

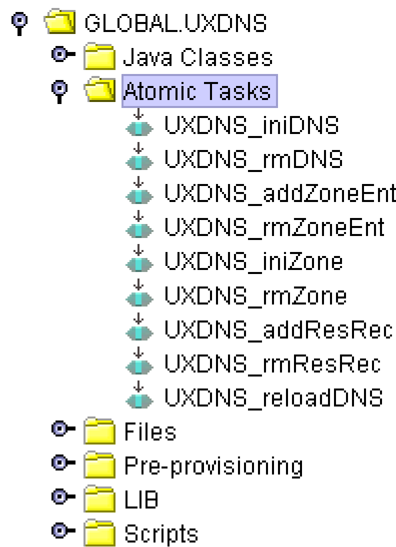
Atomic Tasks

The methods in a plug-in are called **atomic tasks**. There can be any number of atomic tasks inside a plug-in. Each atomic task performs one individual configuration change on a target. The semantics of an atomic task could be quite complex but generally are simple changes.

One hallmark of an atomic task is that it should be reversible. That is, it should be possible to reset the state of the target to the way it was before the task was performed. There are cases where an atomic task is not reversible, but these are rare and should be avoided if possible.

Figure 1-1 shows a plug-in used to configure DNS. Each of the tasks contained in the plug-in (UXDNS_iniDNS, UXDNS_rmDNS, and so on) is called an atomic task.

Figure 1-1 The DNS Plug-in



Atomic Task Parameters

Atomic tasks always take a list of parameters. These parameters tell the task what target to operate on and, optionally, other values to specify the exact nature of the change to make. The parameters are directly reflected in the parameters of the Java method: each parameter to the atomic task is a parameter of the method. All parameters must be of type String.

As noted above, each atomic task should be reversible. If the Resource Manager determines that an activation transaction needs to be rolled back, then any atomic tasks in a transaction that have been completed will be invoked again and told to undo their changes. Thus, the atomic task methods take an initial parameter indicating whether this is the `DO_AND_CHECK` or the `UNDO_AND_CHECK` invocation of the task.

Packaging a Plug-In

In many cases, a plug-in consists of more than just the Java class. Depending on the target devices, the plug-in may also depend on scripts (in any scripting language supported by the targets), libraries, and other non-executable files. All of these files, along with a manifest that contains additional configuration data, comprise the plug-in.

All of the components of a plug-in are packaged and delivered in a **Plug-in Archive (PAR)** that obeys a specific directory structure. For more details on this structure, see “Understanding Plug-in Archives” on page 24. A PAR is delivered as a single file that uses the Java Archive Format (this is the same as a ZIP format).

Understanding the Resource Manager

The Service Activator Resource Manager is XA compliant and participates in transactions coordinated by a Transaction Manager. The Resource Manager shields the plug-ins from the complexities of the XA protocol.

NOTE

XA is a standardized interface which is used to pass and coordinate transaction identifiers. For additional information, see *X/Open CAE Specification—Distributed Transaction Processing: The XA Specification*, ISBN 1-872630-24-3, published by X/Open Company Limited, U.K., and currently available for download at the following URL: `http://www.opengroup.org/products/publications/catalog/c193.htm`

The Resource Manager reliably maintains the state of each transaction in order to recover transactions that were interrupted by a system failure. The default behavior of Resource Manager is as follows:

- Each transaction initiated by a workflow passes through the Resource Manager.
- The Resource Manager assigns each transaction a unique ID (UID) based on the transaction ID (XID). This UID is approximately 13 digits. It maps to the XID and the transaction is saved in the database with the 13 digit name. The mapping between the UID and the transaction ID is printed in the Resource Manager log file found in:

  `$ACTIVATOR_VAR/log/resmgr_active_log.xml`

  The mapping is also maintained in the database.
- When the activation is complete, Resource Manager either deletes the transaction from the DB or leave it in there based on the setting of *`SaveOldTransactions`* configuration parameter in the `resmgr.xml`. See the comments in the `resmgr.xml` for more information.
- When the Resource Manager starts up, it processes all the ongoing transactions present in the database. Typically, there will be no transactions to recover unless the Resource Manager had crashed during the execution of the transaction.

NOTE

To view the state on an individual transaction, run the script:

```
ViewTransactionState[.bat] <options> [ all ] -dbUser <name> -dbPassword <password>
```

Append the parameter `all` to see the entire list of saved states for this transaction. If you do not use the parameter `all`, you will see only the last state of the transaction.

To delete the completed transactions from the database in case the *`SaveOldTransactions`* is set to true, run the script:

```
DeleteCompletedTransactions[.bat] -dbUser <name> -dbPassword <password>
```

This script only deletes the Db transactions which have been completed.

Understanding Locking

Locking exists to prevent two or more related atomic tasks from interfering with one another. For example, your plug-in might contain a task named `addUser()` that you don't want to run at the same time as `removeUser()` on the same target machine. In its typical usage, locking will prevent two atomic tasks from the same plug-in from executing at the same time on a single target. It is also possible to confiugre no locking arguments or a count, which indicate how many atomic tasks can be running in parallel with the same values for the locking arguments. Locking is managed by the Resource Manager and is based on the plug-in being used. An atomic task from one plug-in will never block an atomic task from a different plug-in.

NOTE The locking applies across different cluster nodes in a cluster environment.

When you create a plug-in using Service Builder, you specify the locking arguments or no lock arguments for the plug-in. These locking arguments should be given as a space-separated list of numbers in ascending order. Typically, a locking argument of 1 is sufficient. However, you can set the locking arguments to include as many arguments as you wish, for example, `1 3 5`. The locking arguments must be space-separated, and they must be in ascending order.

Typically, it is sufficient to lock on the machine argument of an atomic task. This is usually the first string argument of an atomic task. However, more elaborate locking is possible. You must specify at least one locking argument for a plug-in. The more locking arguments you specify, the more fine-grained the locking will be. Since locking arguments apply to an entire plug-in, you must ensure that every atomic task in a plug-in has as many arguments as the highest locking argument number for that plug-in.

When the Resource Manager is about to run an atomic task, it checks the locking arguments for that plug-in. If there is another atomic task in the same plug-in that is currently running with the same values for the locking arguments, the Resource Manager will block the new atomic task from running until the first atomic task is complete. When the first atomic task completes, the new atomic task can begin.

Example 1-1 **Locking Example**

Assume you have a plug-in named `GLOBAL.testplugin` with two atomic tasks:

```
public ExecutionDescriptor task_task1(int op, String machine, String name,
                                      String directory)

public ExecutionDescriptor task_task2(int op, String machine, String name)
```

The locking argument for this plug-in is set to 1. This specifies the “`machine`” argument as the locking argument. Locking arguments start with the first `String` argument to your atomic task (not with the `int op` argument). Also, there is no requirement that the first argument to every atomic task in this plug-in have the same name. A locking argument of 1 specifies that the first `String` argument is the locking argument, regardless of the parameter name used in the code.

The Resource Manager is currently running `task_task1(DO_AND_CHECK, "machine.domain.com", "joe", "/home")` of the `GLOBAL.testplugin`. While this atomic task is running, a request comes in to invoke `task_task2(DO_AND_CHECK, "machine.domain.com", "bill")` of the `GLOBAL.testplugin`. When the second atomic task request comes in, the Resource Manager checks a locking table to see if another atomic task is currently running with a locking argument of `machine.domain.com` in the same plug-in. Since the second atomic task request belongs to the same plug-in as the first atomic task call, and since the locking argument (`machine.domain.com`) is identical in both cases, the Resource Manager will block the second atomic task call until the first has completed. If the second atomic task call had a parameter of `machine2.domain.com`, there would not be any blocking, since the locking arguments do not match.

If this plug-in specified the locking argument as `1 2`, blocking would not take place. All locking arguments must match for the Resource Manager to block an atomic task, so in the above case blocking would not take place because “`joe`” does not equal “`bill`.”

Configuring the Resource Manager

You can change various aspects of the Resource Manager behavior. This configuration is specified in the `resmgr.xml file`. For information about these parameters, see the comments in the `$ACTIVATOR_ETC/config/resmgr.xml` file.

Table 1-1 ResourceManager Parameters

Parameter	Required	Description	Reconfigurable	Default
Port	Yes	The port to which the Resource Manager is bound.	No	None
SaveOldTransactions	No	If *SaveOldTransactions* is set to '`true`', the DB row for the transaction state in the database is not deleted when the activation completes , instead the completion status is update to completed and the transaction time is set. The completed transaction can then be viewed with `ViewTransactionState[.bat]`. If the parameter is set to '`false`', the DB rows are deleted when transactions are completed.	No	No
PluginMonitorPollInterval	Yes	The interval in milliseconds between attempts to retrieve and cache last modified time of deployed plugins.	No	None
LockMonitorThreadSleepInterval	Yes	The lock monitor thread handles retry of releasing locks to other atomic tasks. Retry happens e.g. if one cluster node goes down and the work is taken over by another cluster node or if a cluster node is suspended.	No	None
VARDirectory	Yes	A directory where the Resource Manager repository is.	No	None
EnableRemoteDeployment	No	Creates a deployer deploying plug-in scripts, files, etc.	No	No
EnableVersioning	No	Enables versioning of PAR into account when executing plug-in scripts.	No	No

Table 1-1 ResourceManager Parameters (Continued)

Parameter	Required	Description	Reconfigurable	Default
CacheExpiration	Yes	Controls how long (in milliseconds) plug-in objects will be in cache.	No	None
PluginLogs/SpecificPluin				
name	Yes	Name of plugin whose log should be written to a separate log file.	No	None
namespace	No	Name space of plugin.	No	GLOBAL
log_level	No	Debug level of the log information for this Plug-in.	No	INFORMAT IVE
log_directory	Yes	A directory where the log files will be placed.	Yes	None
log_max_entries	No	The maximum number of log entries written to a log file before the log file is closed and a new one is created.	Yes	Will use the Logger value
Logger				
Logger ClassName	Yes		No	None
Logger log_directory	Yes	A directory where the log files will be placed.	Yes	None
Logger log_level	Yes	The following *log_level* parameters set the type of information logged: ERROR - to record only error messages logged. WARNING - to record errors and warnings. INFORMATIVE - to record errors, warnings and some additional information. DEBUG - to get additional debugging information. DEBUG2 - to get even more detailed debugging information.	Yes	None
Logger log_max_entries	Yes	The maximum number of log entries written to a log file before the log file is closed and a new one is created.	Yes	None
Deployer				

Table 1-1 ResourceManager Parameters (Continued)

Parameter	Required	Description	Reconfigurable	Default
Deployer ClassName	Yes	The configurable parameters listed are applicable only to the default SSHScriptDeployerFactory which provides a deployer that uses Secure Shell for secure communications.	SSHScriptDeployerFactory	None
Deployer Param	Yes	Parameter name and value pair for deployer. This depends on specified deployer in classname. You can repeat this parameter with different values as many times as you need.	No	None
DisabledtaskDelay	No	By default, if a task is disabled, then the Resource Manager instantly skips over the task, acting as if the task succeeded w/o any stdout/stderr/description. You can specify a finite delay if desired. The *DisabledtaskDelay* is specified in seconds and is global for all disabled tasks.	Yes	0
DisableAtomicTask	No	You can declare some atomics to be disabled. This will allow an existing solution to operate in an environment where the give atomics cannot actually run. Thus, a complete solution can be demonstrated in the absence of various target hardware or software. You can disable one or more atomics and can use some rudimentary wild-carding... the star (*) will work at the end of the task name (not in the middle). Repeat the tag to disable multiple atomics. You can repeat this parameter with different values as many times as you need.	Yes	None
Database (*ResMgrOracleDatabaseConnectionManager*) The recommended one				
datasource_name	Yes	The Datasource file name. The datasource file must be found in the directory $JBOSS_DEPLOY	No	None

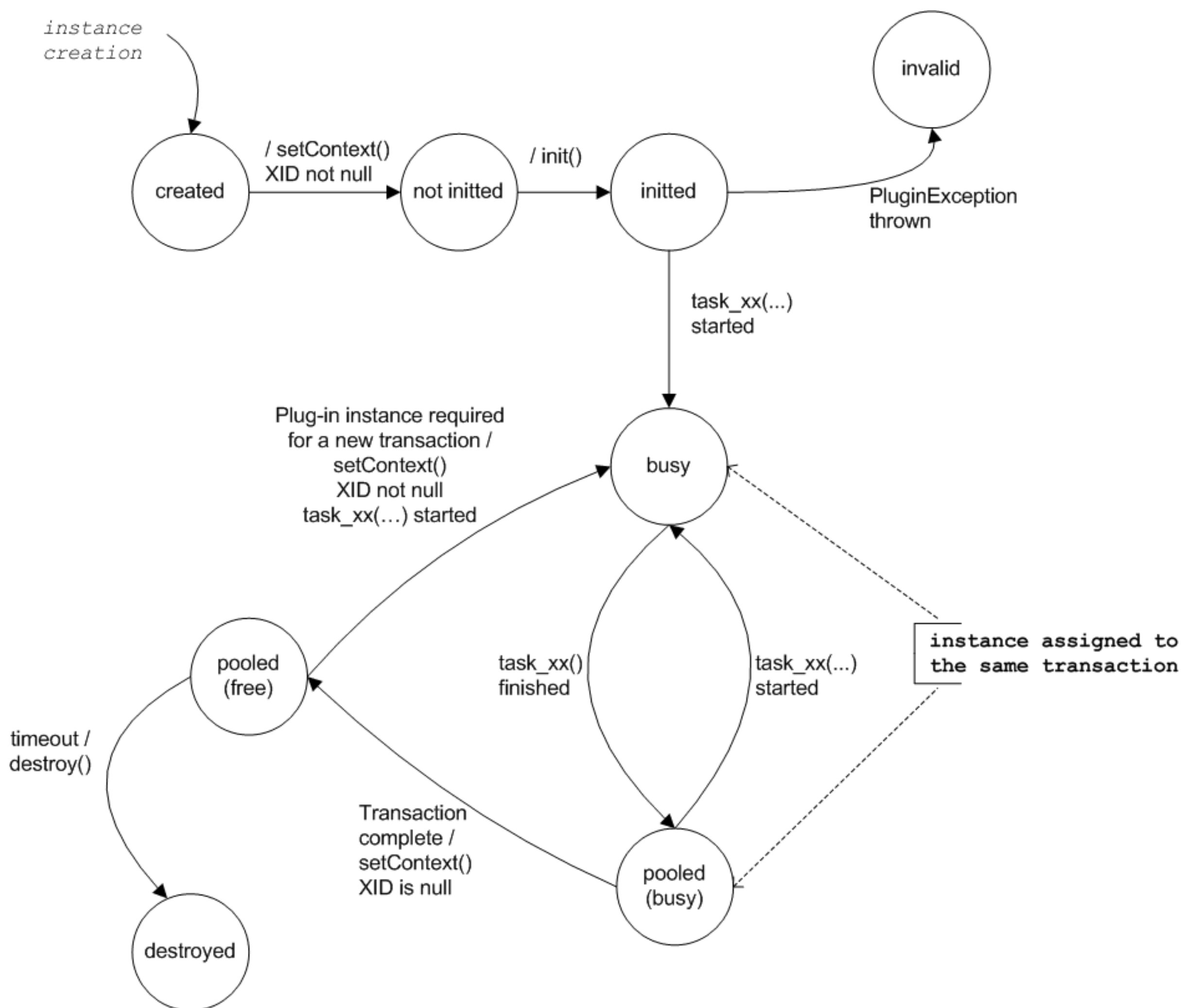
Table 1-1 ResourceManager Parameters (Continued)

Parameter	Required	Description	Reconfigurable	Default
Database (*ResMgrSimpleDatabaseConnectionManager*)				
driver	Yes	The name of the Java class that implements the JDBC driver to be used to access the database	No	None
Server_name	Yes	The Database server name or IP address.	No	None
user	Yes	The user name that is required to make connection to the database.	No	None
password	Yes	The password for connecting as the specified user.	No	None
database_name	Yes	The name of the database to be connected to.	No	None
port	Yes	The port on which the DB connections are to be made.	No	None
connections	Yes	The number of simultaneous connections that the module can maintain to the database. This allows multiple threads to work against the database concurrently.	Yes	None
max_usages	No	The maximum number of times that a connection is reused before that connection is released and another is established. A value of 0 means that connection will not be released.	Yes	10000
timeout	No	Indicates how long a connection can operate before it is forced to close. The default value is 45 seconds. 0 indicates that there is no time-out	Yes	45000

Understanding the Plug-in Life-Cycle

A plug-in moves into and out of several valid states during its life cycle, as shown in Figure 1-2.

Figure 1-2 Plug-in Life Cycle Diagram



The plug-in moves from one valid state to another using valid transitions. These transitions are established when the Resource Manager invokes a method.

The different states include:

created

The first time a workflow calls a plug-in for a transaction (and this plug-in is not in the pool), a new instance of the plug-in is created by the Resource Manager. This initiates a call to the `setContext` method (that the plug-in might have overridden). The Resource Manager invokes this method before `init()` in order to set the context of the plug-in. The context contains information about the current transaction (see `getXID()` from the plug-in context).

not initted

This is the state the plug-in is in right after the `setContext()` method finishes running. The Resource Manager runs the `init()` method, which will transition the plug-in to the initted state. This method is invoked only once with a list of attributes and values. If a `PluginException` is thrown in this method, the plug-in is considered invalid, because it was unable to initialize itself. The developer should throw this kind of exception in the event of a misconfiguration.

busy

A plug-in instance is busy when it is being used in a transaction (for example, there is a call to an atomic task)

task_xx (atomic task runtime)

Each time an atomic task invocation for this plug-in is requested, a call to the actual method will occur. If the call happens during the normal running of the task, the first parameter to this method (`op` parameter) is `DO_AND_CHECK`. If the request is within a rollback, the value is `UNDO_AND_CHECK`.

pooled (busy)

After an atomic task is finished, the plug-in instance returns to the pool state (with *busy* substate). The plug-in instance remains in this state until the current transaction ends, such that the plug-in instance remains linked to the XID until the end of the transaction.

If the current transaction requires another atomic task within the same plug-in, the plug-in instance is reused; it leaves the pool state (with *busy* substate) and returns to the busy state. When the transaction ends, the plug-in instance goes to the pool state (with *free* substate).

pooled (free)

When the transaction finishes, all the plug-in instances involved in this transaction are marked as *free* in the pool so they can be used again by another transaction. Before that, there is a call to the method `transactionComplete()`. This method is invoked so that the plug-in can take any additional measures when it is disassociated from a transaction (for example, releasing resources).

If another transaction requires a plug-in that has an instance as pooled and *free*, this instance is used and there is a call to `setContext()` with information about the new XID.

destroyed

After some time without being used by any transaction, the Resource Manager reaps unused instances. Those plug-in instances in the pool marked as *free* are removed. Before destroying the instances, there is a call to the `destroy()` method.

destroy( )

This is the method invoked when the plug-in instance is destroyed in order to free resources.

The Resource Manager can choose to create several plug-in instances, depending on the number of simultaneous activations. An instance will remain linked to a transaction from the first time it is used until the transaction ends.

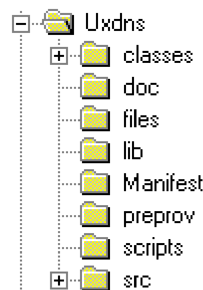
If there is a failure in the Resource Manager, and the Resource Manager goes down, new instances are created when the running transactions are recovered (there is no serialization of the plug-in instances).

NOTE You should *not* use member variables in the plug-in instance to store things like the previous state of a resource if you expect to have that information available in a rollback. There are special methods in the `PARContext` available for storing persistent state across the life of a transaction. For additional information on these methods, see the *Javadoc* for `ParContext`.

Understanding Plug-in Archives

Every plug-in archive has a specific file system layout. This directory structure is automatically generated for you when you create a plug-in using Service Builder. The parent directory typically has the same name as the plug-in. The directory structure for the DNS plug-in archive, for example, is shown in Figure 1-3.

Figure 1-3 Plug-in Archive File Layout



A Plug-in Archive (PAR) is a file that contains all of the necessary elements to properly run the plug-in's atomic tasks on a target machine. These elements include classes, files, scripts, preprovisioning scripts or files, and libraries.

The following paragraphs describe the purpose and contents of each directory. The `src` and `doc` sections can be excluded when deploying a PAR because they are not needed at runtime. Service Builder gives you the option of excluding them when deploying the plug-in archive.

classes

This directory contains the Java class or classes for the plug-in. The Java compiler generates this class or classes when you compile your code.

doc

This directory is created and populated when the documentation generation is run for a plug-in. This process generates HTML files that are stored in this directory. The information is gathered from the *Javadoc* in the Java class that is implementing the plug-in for the given task, and from the **deployment descriptor**.

files

This directory contains files (not scripts) that are used to configure the plug-in for a particular environment. The plug-in copies these files to the target machine so that the information required by the atomic task is available to it locally. Using files also simplifies future changes to a plug-in.

For example, the Oracle® PAR uses an XML definition file stored in this directory to write the SQL sentences used in each atomic task. An operator can change the references in the file to adapt the plug-in to new releases of the database without changing the source code of the plug-in.

lib

This directory contains a set of JAR and ZIP files. These files represent a set of tools that the plug-in Java class or classes may need. These files are not copied to the target machine. All of these files must be in this directory and not in subdirectories. These JARs are available to be linked into the Resource Manager if your plug-in needs additional functionality at run-time. An example of a file that you might include in the `lib` directory is a JDBC driver for accessing a database from within your plug-in.

MANIFEST

Contains the deployment descriptor in the file `par.xml` that defines the characteristics of the plug-in, including atomic tasks and scripts. You can modify this file for an existing plug-in to create your own plug-in. Also, when you use Service Builder to create a plug-in, Service Builder creates and populates `par.xml` automatically with the information that you type.

preprov

This directory contains preprovisioning files that are copied to the target machine the first time an atomic task from this plug-in is executed for a given target. The files contain preprovisioning information that the atomic tasks use to set up the target environment for the plug-in.

Each preprovisioning file may be marked as executable. The Resource Manager attempts to execute each preprovisioning file when both of the following conditions apply:

- It is the first time the plug-in is used on the target machine.
- The `executable` flag for that file is set to `true`. In the GUI, this is represented as a check-box.

Additionally, an interpreter may be specified that will be used to execute the file. This is important if the file is a script.

scripts

This directory contains scripts that are called directly by atomic tasks. All the scripts are copied to the target machine the first time an atomic task from this plug-in is executed for a given target. An atomic task can use more than one script.

Additionally, an interpreter may be specified that will be used to run the script.

src

This directory contains the Java source code that makes up the plug-in. Often there will be only one Java source file that comprises the plug-in, but you can include as many source files as you wish. The main plug-in class must do the following:

- Extend the class `PARPlugin`, which is provided as part of Service Activator
- Define one or more atomic task methods in the plug-in class

Additionally, the plug-in class may do the following:

- Provide an `init()` method
- Provide a `destroy()` method that cleans up a class instance before the instance is destroyed
- Provide a `transactionComplete()` method that releases resources at the end of a transaction involving this plug-in.

NOTE Scripts and files are copied to the remote machine only on first activation. Later this step is skipped. If these files are removed from the target machine, you will have to re-deploy the plug-in to Service Activator. This will force to upload scripts and files on the remote machine once more.

For more information about building the source code for a plug-in, see "Creating Plug-ins: Advanced Tips" on page 74.

Understanding the Plug-in Context

In most of the steps described in the plug-in life cycle, the plug-ins will have a context available where they are able to log messages, run scripts remotely, retrieve plug-in properties, and so on.

For a list of the available `PARContext` methods, see the *Javadocs* for `ParContext`.

Understanding Compound Tasks

A compound task is an ordered list of atomic tasks (or other compound tasks), along with a mapping from the input parameters of the compound task to the parameters of each called task. When a compound task is invoked, each of the atomic tasks within the compound task are invoked in the order they were specified in the compound task. The parameters that were passed to the compound task are passed to each called task according to the parameter mapping specified in the compound task definition. For additional information on parameter mapping, see “Changing Parameter Mappings” on page 60 and “Creating Compound Tasks Manually: Advanced Tips” on page 85

If one of the atomic tasks in the compound task fails, each of the previously executed tasks in the compound task is invoked again and told to undo the work it just completed. This is how the transactionality of compound tasks is implemented.

Compound tasks can be created graphically within Service Builder. It is also possible to specify the definition of a compound task using XML. In either case, just like plug-ins, compound tasks must be deployed into Service Activator before they can be invoked from the workflow engine.

Using the Plug-in Library

Follow the instructions below to create and use the plug-in library.

Accessing Plug-in Documentation

Each plug-in normally has an associated *Javadoc*. Service Builder allows you to open each plug-in and generate the plug-in documentation in HTML format. Use the following steps:

1. Create a new project in Service Builder by selecting `New Project` from the `File` menu.
2. Type a name for the project and, if you choose, a description field.
3. Select `[Finish]`.
4. In the view of the directory structure, open the folder for the project you just created and right-click the `Plug-in Archives` folder.
5. Select `Add Plug-in Archive` from the pop-up menu.
6. Use the file browser dialog box, locate the plug-in archive that you want to read about, and then select `Open`.
7. From the `Tools` menu, select `Generate Documentation` to generate an `index.html` file for this plug-in.
8. Browse to the `doc` directory for the project and open the `index.html` file to view the documentation for that plug-in.

Using Plug-in Classes

Each plug-in can access certain plug-in classes within the library provided with Service Activator. These include:

- `AttributeTable` – a standard Java class used to query for parameter values and retrieve them.
- `PARPlugin` – a Service Activator class that all plug-ins are required to extend, which provides `init` and `destroy` methods that a plug-in can override.
- `PARContext` – a Service Activator built-in variable of the `PARPlugin` that provides several capabilities including the ability to execute scripts remotely, save transaction state, and log messages to the Resource Manager log file. The `PARContext` interface also includes the following 5 parent interfaces:
  - `PARLogger`
  - `AtomicTaskStateSaver`
  - `TransactionStateSaver`
  - `StateSavingConstants`
  - `DoneRequester`
  - `DataUploader`

For more information about each of the classes discusses here, see the *Javadocs* for that particular class.

2 Understanding and Using Service Builder

This chapter provides information about the Service Builder tool supplied with Service Activator and how to use it to develop and manage plug-ins and compound tasks.

Using Service Builder to Create Plug-ins

The Service Builder tool provided with Service Activator is an Integrated Development Environment (IDE) you can use to create plug-ins and the associated atomic tasks. This is especially convenient when you are creating large, complex plug-ins, as Service Builder automatically generates atomic task method stubs for you and updates the deployment descriptor.

Since plug-ins are simply a collection of files that obey a well defined directory structure, plug-ins can be created without the use of Service Builder. Many developers find a hybrid approach to be the most useful: they use the Service Builder GUI to get the structure right but do most of the actual writing of the plug-in code using their favorite editor.

Service Builder also provides a GUI tool to create compound tasks built-up from atomic tasks. Again, these can be created in any text editor, but generally the GUI tool is easier for this activity.

Service Builder is also the conduit for deploying plug-ins into the Service Activator environment and for obtaining information about the currently deployed plug-ins and compound tasks. This functionality is available from the command line or the GUI mode.

Starting Service Builder

To start the Service Builder tool in the GUI mode, run:

***$ACTIVATOR_BIN*/servicebuilder**

Creating a Project

In GUI mode, all of your work is organized by projects. Think of a project as a container to hold plug-ins and compound tasks during your development phase. Once your plug-ins are complete and a PAR has been generated, the plug-ins are no longer tied to the project (for example, the PAR can then be used by a different project).

To begin, you must first create a project or open a previous project:

- To create a new project, select `File` from the menu, and then select `New Project`.

  Specify the name and location of your project. You can name your project anything you like. The project file name uses the project name by default, but it can be modified.

- To open an existing project, select `File` from the menu, then select `Open Project`.

Creating a Plug-in

In the GUI you can create a new plug-in in two ways:

- To create a new plug-in from scratch in your project, right-click the `Plug-in Archives` folder, and select `New Plug-in`.

NOTE If the `Plug-in Archives` folder is not visible, click the horizontal magnifying glass icon to the left of your project folder.

- Alternatively you may start from an existing plug-in and modify it. In your project, right-click the Plug-in Archives folder, and select `Add Plug-in Archive`. Browse to the location of some existing PAR files. Once you add a plug-in to a project, you have a complete copy of that plug-in in your project. Changes you make to the plug-in are *not* reflected in the original.

Creating a New Plug-in from Scratch

When you create a new plug-in from scratch, the wizard displays the following dialog box:

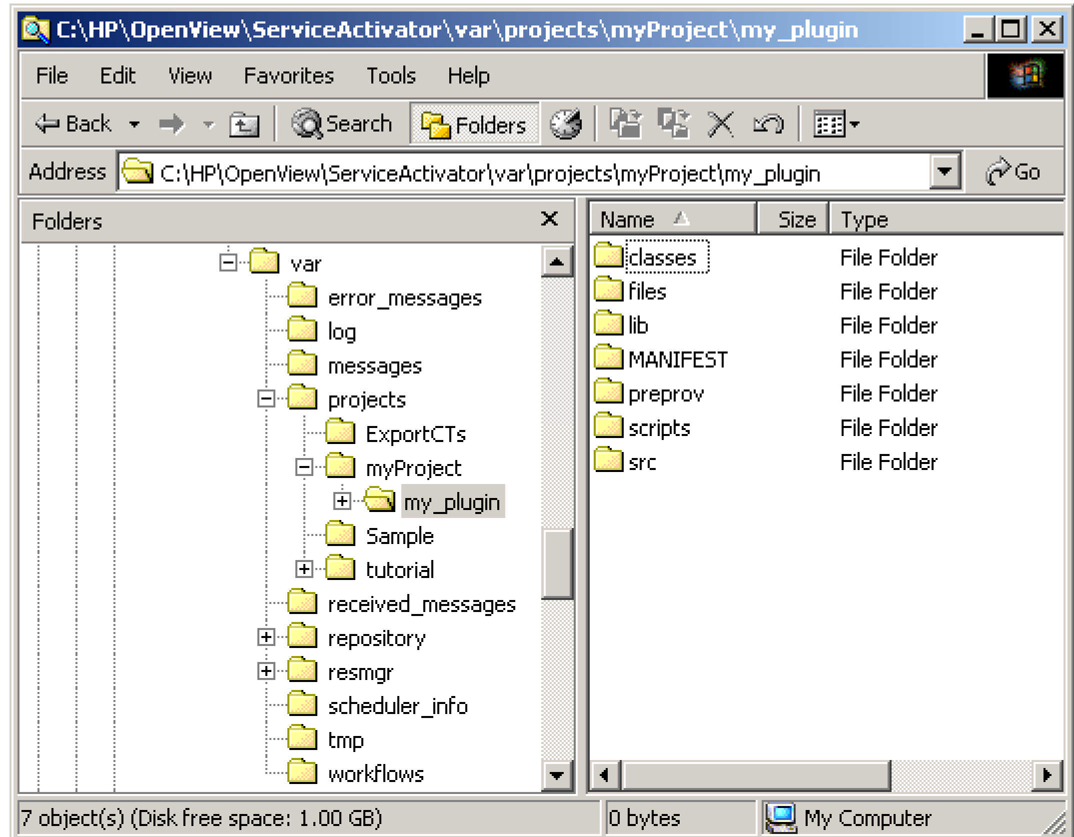
Figure 2-1 Service Builder Plug-in Properties Dialog

1. Name your plug-in. This is also the name of the Java class; thus, it must be a valid Java identifier.

2. Type a brief description of your plug-in. This description is shown in the *Javadoc* and the documentation of your plug-in.
3. Type the package name. A package name is not required, but it is encouraged. The built-in plug-ins use the following package "`com.hp.ov.activator.plugins`". You should use a different package name.
4. Type the version number. The three fields correspond to major, minor, and revision. The major field is required.

   The system does not support multiple versions of the same plug-in active in the system at the same time. The version information is currently only useful for being able to determine which version is loaded. Future releases may support multiple versions of a plug-in being deployed at the same time.
5. Choose the name space for your plug-in. A name space is used when referring to the atomic tasks within a plug-in to perform activations with Service Activator (the Java class and package name is not used from within Service Activator). Consider the following requirements when choosing the name space for your plug-in:
   - The combination of name space and atomic task name must be unique within the set of tasks deployed on a server. Thus, if you have two tasks with the same name in different plug-ins, the two plug-ins must have different name spaces.
   - By convention, all plug-ins provided with Service Activator have a `GLOBAL` name space, and all atomic task names are prepended by the plug-in name (for example, `UXOS_addDir`).
6. Choose the deployment mode. The deployment mode tells the Resource Manager whether it should copy files to the target resources or not:
   - ON-DEMAND - copies scripts, files, and preprovisioning files to the target systems prior to running the first atomic task on that target
   - NO DEPLOYMENT - The Resource Manager will not copy these files to the target systems. It assumes that the scripts, files, and preprovisioning files are already available on the target system. In this case, the fact that these files are packaged into the PAR is a simply a matter of keeping track of the versions that are expected to be used together. The proper files must be manually provisioned onto the target systems.
7. Specify which arguments of the atomic tasks will be used as locking arguments. If no arguments should be used select None. Typically, plug-ins use only the first argument as the locking arguments, so you would only specify "1". See "Understanding Locking" on page 16 for more information about locking arguments. Set the Count value to the number of atomic tasks which can be running in parallel.
8. Click `[Finish]`:
   - This creates the basic Java code for the plug-in. There are no atomic tasks yet; just the basic Java shell.
   - The Service Builder message section at the bottom of the window indicates a successful addition of the plug-in.
   - Service Builder creates a new PAR directory structure for your project that is similar the one shown in Figure 2-2.

Figure 2-2 Example of PAR Directory Structure

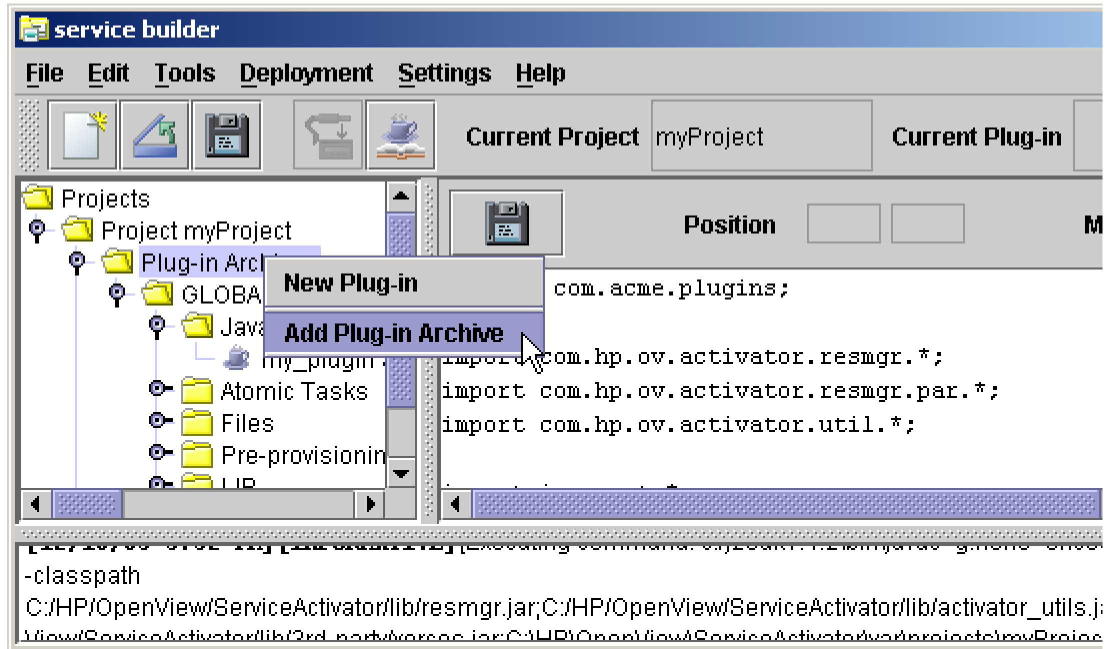


NOTE Figure 2-2 shows the PAR directory structure as displayed in Windows Explorer.

Adding an Existing Plug-in

1. In Service Builder, right-click the `Plug-in Archives` folder of the project tree, and select the `Add Plug-in Archive` option, as shown in Figure 2-3.

Figure 2-3 Add Plug-in Archive



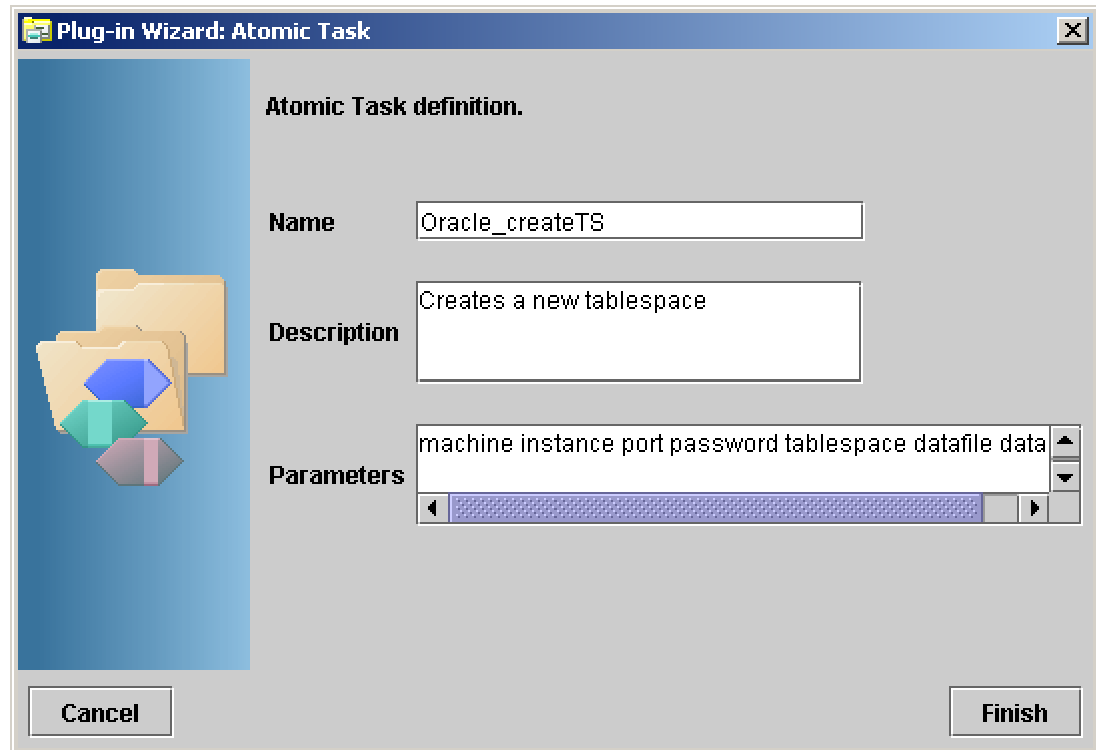
2. Browse to the location of the PAR file you want to add to your project.
3. Select the PAR, and then click `[Open]`. A successful operation produces messages similar to the following:

```
[12/16/03 4:01 PM] [INFORMATIVE] [Importing Plug-in Archive in
C:\HP\OpenView\ServiceActivator\SPI\web_hosting\MSSQL.par]
[12/16/03 4:01 PM] [INFORMATIVE] [Plug-in Archive successfully added: MSSQL]
```

Adding Atomic Tasks

1. Right-click the `Atomic Tasks` folder in the plug-in directory, and select `New Element`.
2. In the `Atomic Task` dialog box (shown below), type the name of the atomic task you are creating. If your plug-in is using the GLOBAL namespace, a good convention is to prepend the name of the plug-in to the name of your atomic task. For example, the `addDir` atomic task in the W2K plug-in is actually called `W2K_addDir`.

Figure 2-4 Service Builder Atomic Task Definition Dialog



3. Type a short description of the task. This description is kept in the manifest for the plug-in and is accessible by Service Builder when it displays the list of all tasks that are currently deployed to the Resource Manager.

NOTE When the task is first created, this description will be transferred to the *Javadoc* comments above the newly generated atomic task. Any subsequent changes to the description will not be transferred to the task header. Likewise, any edits to the task header will not be transferred back to the task description.

4. Specify the names of the parameters, separating them with commas or spaces. Usually, the parameters will begin with *`machine`*, as shown in the following example:

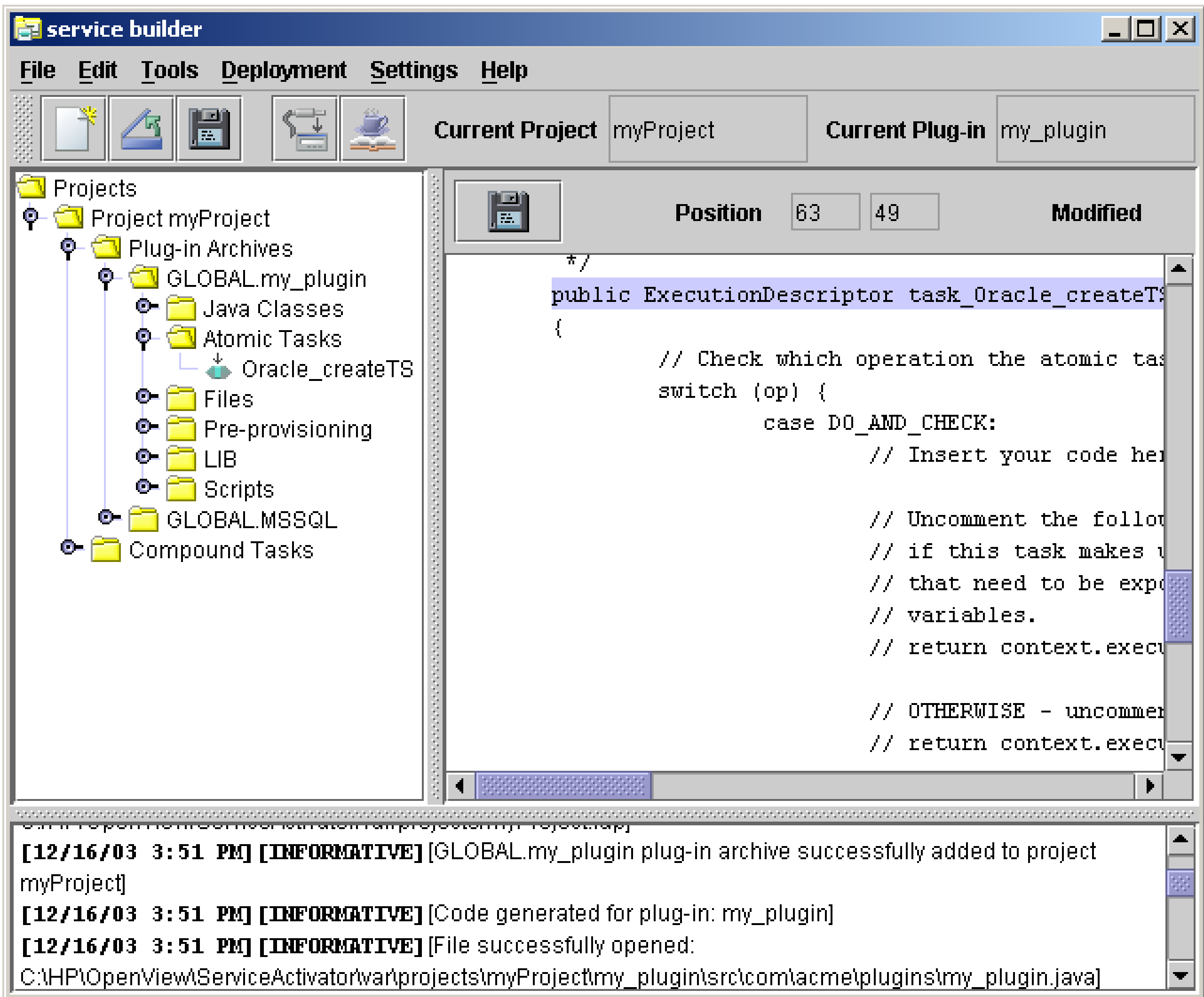
```
machine instance port password tablespace datafile data
```

5. Click `[Finish]`. The wizard will automatically create the code for the atomic task and add it to the Java class.

6. To view the Java code that was just generated, select the atomic task in the left-hand pane of Service Builder. The Java code will appear in the right-hand pane, as shown in Figure 2-5.

NOTE You can change the parameter list for an atomic task by editing the Java code directly. To remove an atomic task from a plug-in, you must delete the method from the Java code. For additional information, see “Creating Plug-ins: Advanced Tips” on page 74

Figure 2-5 Viewing the Java Code for a New Atomic Task



Using Common Source Files in Multiple Plug-ins

Many plug-ins have files, scripts, and Java code that are unique to that individual plug-in. Sometimes, however, there are files that can be used in multiple plug-ins. Typically, in this case, you want to have any changes to any common files reflected in all of the PARs that include them.

When you create a PAR and include these various files in it, the PAR actually contains a copy of these files. However, the PAR also keeps a reference to the original location from which the files were copied. This way, if a common file gets updated at a later point in time, you can tell Service Builder to update the PAR with fresh copies of any of its files that have been updated.

NOTE If you want to change the local (PAR) copy of a file *without* changing the original file, you will need to break the link between the two files. You can use the following process to do this:

1. Within the PAR directory structure but outside of Service Builder, make a copy of the file (for example, copy `myscript.bat` to `myscript.copy`).
2. Remove the file from your plug-in by right-clicking on it in the Service Builder GUI and selecting `Remove From Plug-in`.
3. Rename the copy you created in Step 1 back to its original name (for example, rename `myscript.copy` to `myscript.bat`).
4. Add this file back into the plug-in by selecting the `New Element` option under the relevant folder (for example, "Files").

Updating the PAR File

The Update operation checks for PAR content (directory structure) changes to any original file (Java class, script, file, preprovisioning program, JAR, or ZIP) in the PAR. If it detects changes, Service Builder copies the changed files from their original location to the destination in the current PAR.

1. Right-click the PAR folder, and then select `Update`.

   The Service Builder message window shows a message similar to the following:

   `[12/13/03 2:48 PM][INFORMATIVE]` [Updating Plug-in files for 'my_plugin'...]

   `[12/13/03 2:48 PM][INFORMATIVE]`[Copying C:\project\scripts\my_script.sh...]

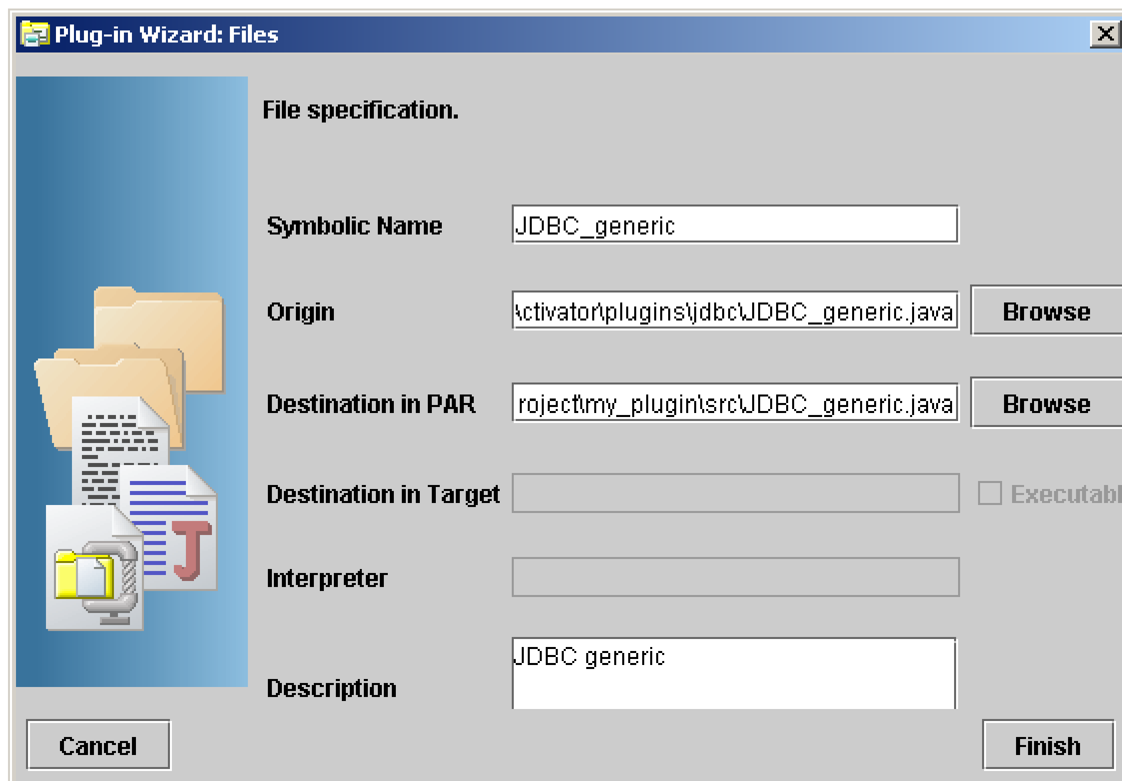
   `[12/13/03 2:48][INFORMATIVE]`[File 'C:\project\scripts\my_script.sh copied successfully to 'C:\project\tutorial\my_plugin\scripts\my_script.sh'.]

Adding Java Classes (Source Files)

Sometimes your plug-in implementation will involve multiple Java source files. Only one file can contain the atomic tasks, but other Java source files can be used in the plug-in. You can add additional Java source files to your plug-ins by following these steps:

1. Right-click the Java classes folder in the plug-in directory structure, and then choose `New Element`. The wizard displays the dialog box shown in Figure 2-6.

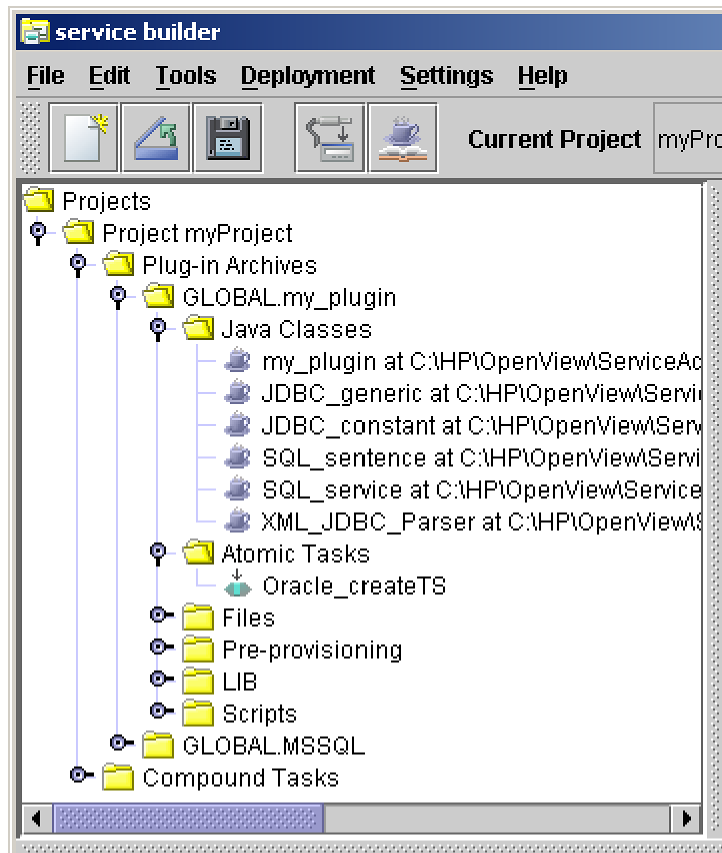
Figure 2-6 Service Builder Files Dialog - Adding a New Java Class



2. Specify the name of the class that you are adding.
3. Specify or browse to the location of the class you want to add.
4. Specify or browse to the location within the `PAR` where you want to place the new class.

   Make sure that the destination matches the package name
5. Type a description of the class and then click `[Finish]`.
6. Repeat steps 1 through 5 for each Java class that you want to add to the plug-in. When you have added your classes, the directory structure for the Java classes will look similar to the one shown in Figure 2-7.

Figure 2-7 Service Builder Project View - After Adding a New Java Class

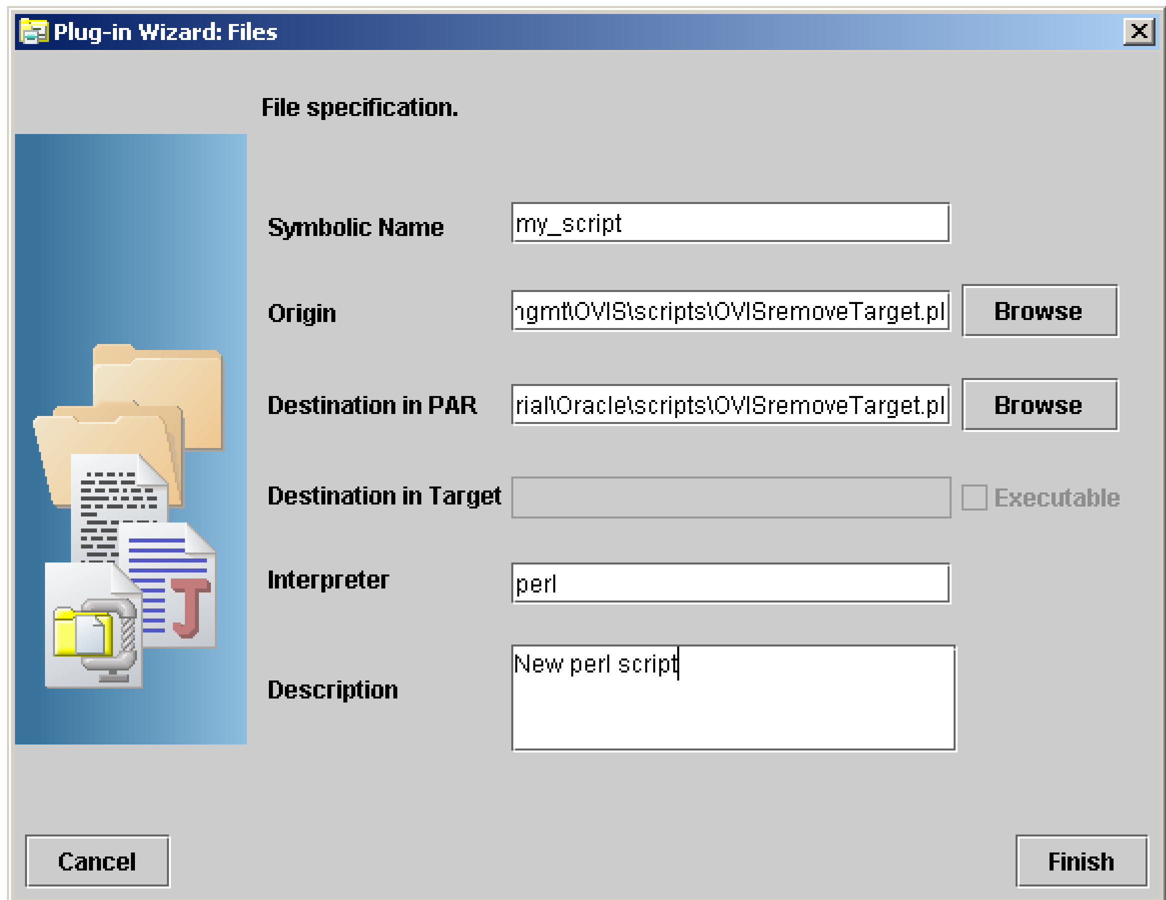


7. To see the configuration of a class, right-click the class in the directory structure under Java classes, and then select `Properties`.

Adding Scripts

1. Right-click the `Scripts` folder in the plug-in directory, and then select `New Element`. The wizard displays the dialog box in Figure 2-8:

Figure 2-8 Service Builder Files Dialog - Adding a Script



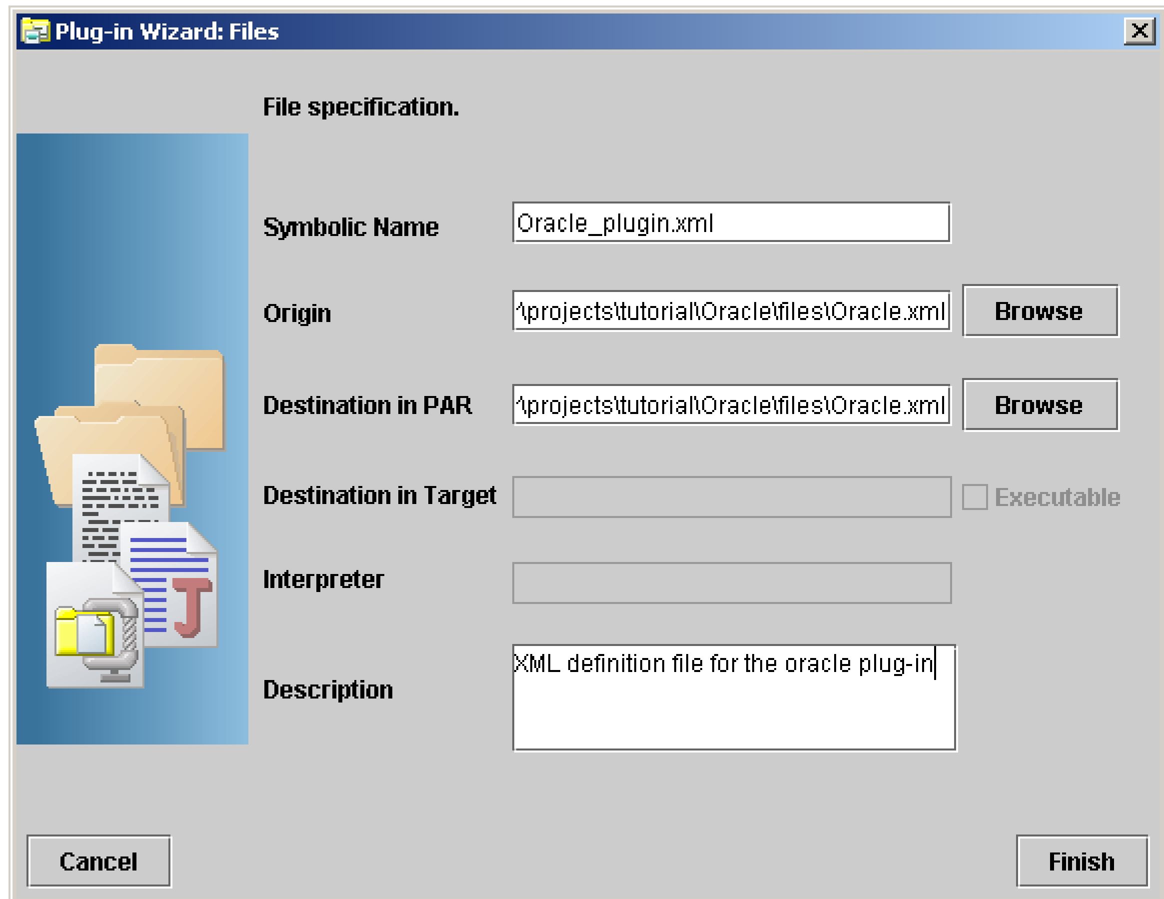
2. Type the symbolic name of the script. This is the name that is used when invoking the scripts from within the Java class.
3. Specify or browse to the location of the script.
4. Specify or browse to the location in the PAR directory where you will store the script. By default, the destination in the PAR is *`project directory`*/*`PAR_dir`*/`scripts`
5. Specify the interpreter that will run the script on the target machine, if required. You can either use the full path or simply specify the name of the interpreter. If you don't provide the full path, the interpreter must be accessible from the `$PATH` environment variable in the user account used in connecting to every target machine.
6. Type a brief description of the script and its purpose.
7. Click `[Finish]` to add the script to the PAR. Once you have added the script to the PAR, you can edit it from the Service Builder editor window:

- To edit the script from Service Builder, double-click the script name in the plug-in script directory.

Adding Files

1. Right-click the `Files` folder in the plug-in directory structure. The wizard displays the dialog box in Figure 2-9:

Figure 2-9 Service Builder Files Dialog - Adding Files

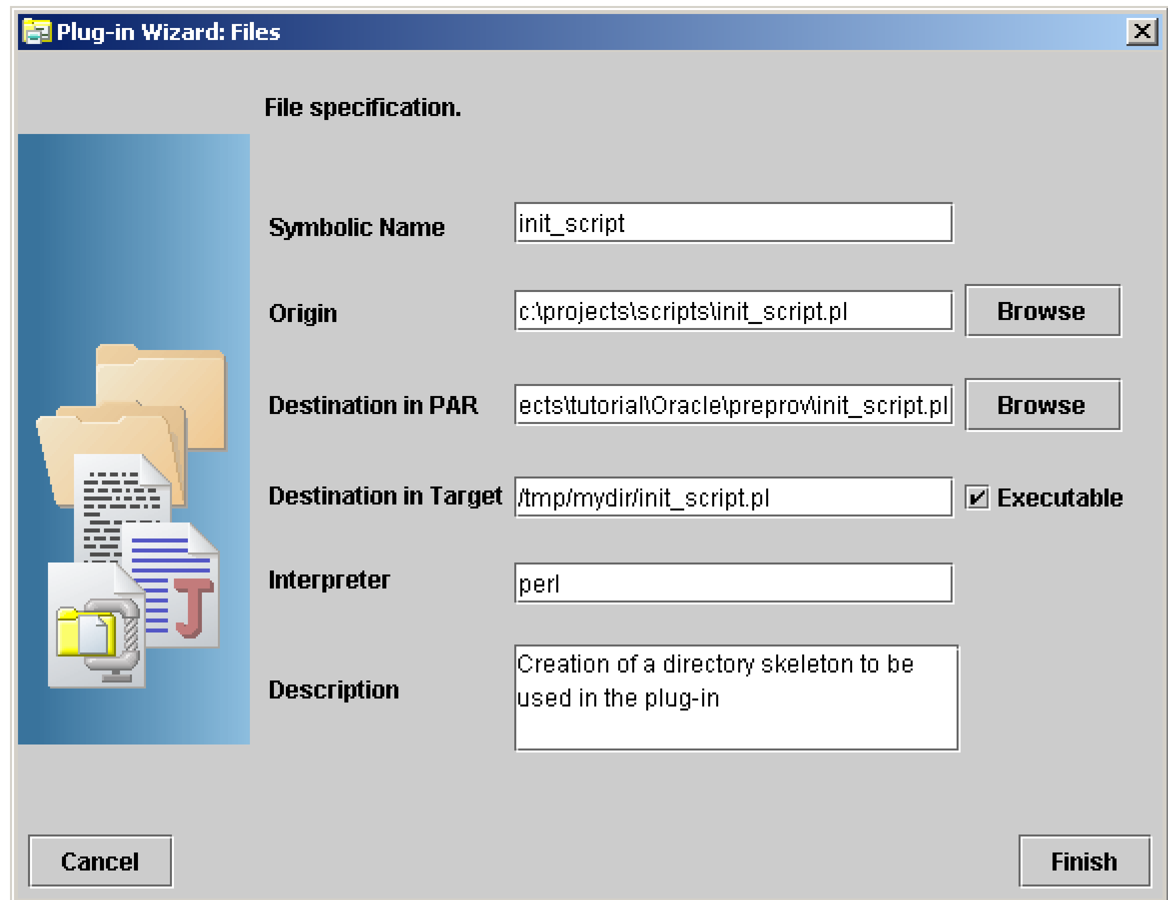


2. Type the symbolic name of the file. This is the name that is used when referring to the file from within the Java class.
3. Specify or browse to the location of the file.
4. Specify or browse to the location in the PAR directory where you will store the file. By default, the destination in the PAR is *project directory*/*PAR_dir*/`files`
5. Type a brief description of the file and its purpose.
6. Click `[Finish]` to add the file to the PAR. Once the file is added to the PAR, you are able to edit it from the Service Builder editor window:
   - To edit the file from Service Builder, double-click the file name in the plug-in `Files` directory.

Including Preprovisioning Scripts and Files

1. To add scripts or files that the plug-in uses to set up the environment on the target machine, right-click the `Preprovisioning` folder, and then select `New Element`.

Figure 2-10 Service Builder Files Dialog - Adding Preprovisioning Scripts and Files



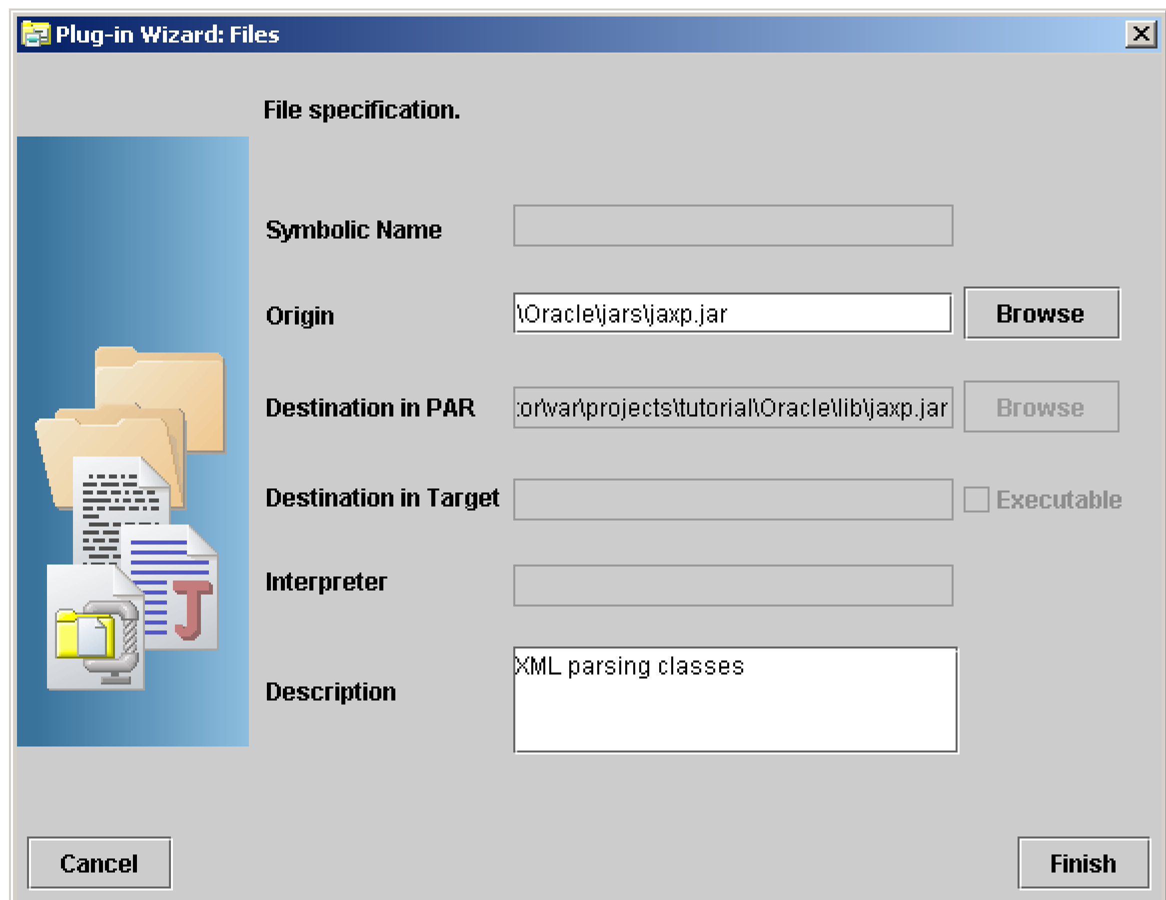
2. Type the symbolic name of the file. This is the name that is used when invoking the scripts from within the Java class.
3. Specify or browse to the location of the file.
4. Specify or browse to the location in the PAR directory where you will store the file. By default, the destination in the PAR is *`project directory`*`/`*`PAR_dir`*`/preprov`.
5. Specify the location to which this file should be copied on the activation target machine.
6. Specify the interpreter that runs the file on the target machine. You can either use the full path or simply specify the name of the interpreter. If you don't provide the full path, the interpreter must be accessible from the `$PATH` environment variable in the user account used in connecting to every target machine.
7. Type a brief description of the file and its purpose.
8. Click `[Finish]` to add the file to the PAR. Once the file is added to the PAR, you are able to edit it from the Service Builder editor window:

- To edit the file from Service Builder, double-click the file name in the plug-in `Pre-provisioning` directory.

Adding a Library

1. To add a library to the PAR, right-click `LIB`, and then select `New Element`.

Figure 2-11 Service Builder Files Dialog - Adding a Library



2. Specify or browse to the location of the original file on the local machine.
3. Describe the purpose of the library, and then select `[Finish]`

Viewing General Properties

1. To view the properties of a plug-in, right-click the plug-in shown in the PAR folder structure, and then select the `General` option. The wizard displays the dialog box in Figure 2-12:

Figure 2-12 Service Builder General Properties Dialog

Plug-in Wizard: General

General plug-in properties.

Name: Switch

Description: Dummy plug-in for us

Package Name: com.hp.ov.activator.plugins

Class Name: Switch

Version: 1 0 0

Namespace: GLOBAL / Private: Intro

Deployment: ON-DEMAND

Lock: None / Arguments: 1

Count: 1

Cancel / Finish

2. If necessary, you can edit the following fields:

- `Description`
- `Version`
- `Name Space`
- `Deployment`

- `Locking Arguments`

NOTE You cannot edit the `Name`, `Package Name`, and `Class Name` fields once you have created the plug-in.

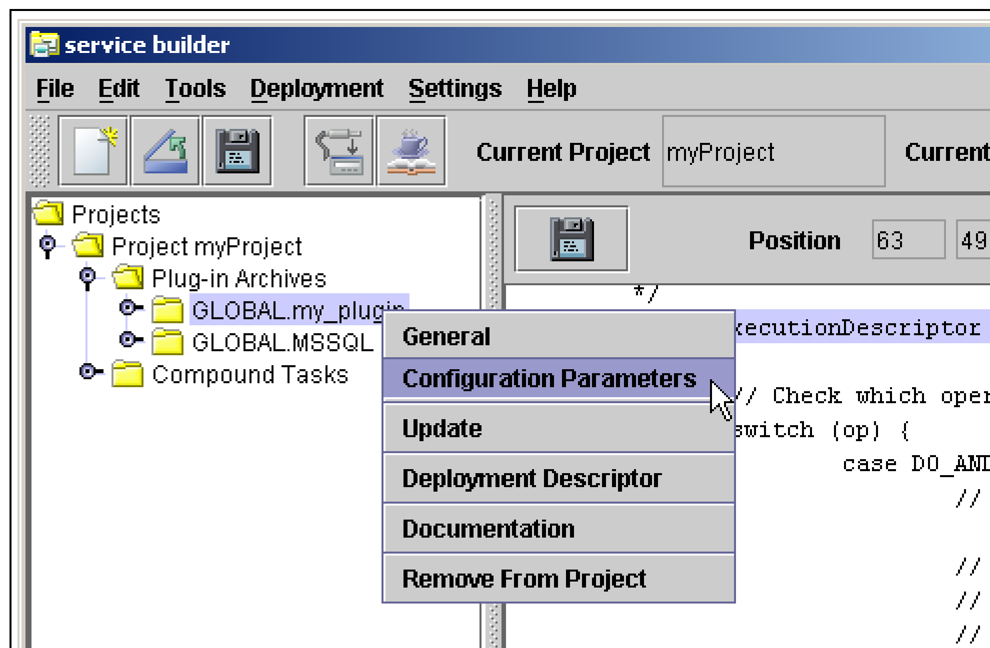
Configuration Parameters

Configuration parameters are a mechanism by which the behavior of a plug-in can be customized without modifying the Java code or the scripts. These parameters and their values are maintained in the PAR manifest. The plug-in Java code can use the PARContext API to ask for the value of these parameters at runtime. When scripts are executed on the target machine, these same parameters can be passed to the script as environment variables.

To create or modify the configuration parameters of a plug-in:

1. Right-click the plug-in you want to configure, and then choose `Configuration Parameters`.

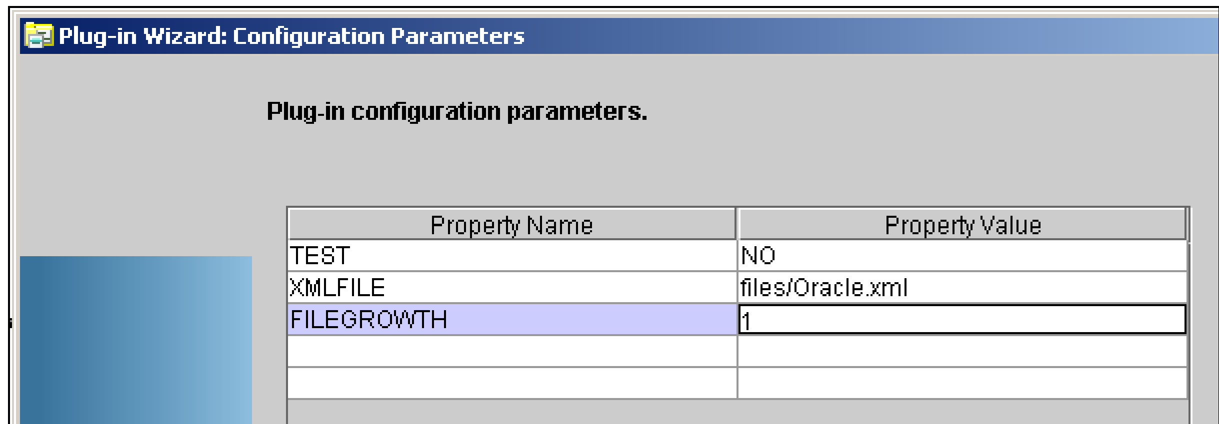
Figure 2-13 Service Builder UI - Configuration Parameters Menu Selection



Note that the two APIs that are used to execute commands on the target machine must specifically indicate that these configuration parameters are to be exported before executing the command. See the Javadoc specification for the APIs `PARContext.executeScript()` and `PARContext.executeCommand()`.

2. Define the plug-in parameters and values in the `Plug-in configuration parameters` dialog box.

Figure 2-14 Service Builder Plug-in Configuration Parameters Dialog

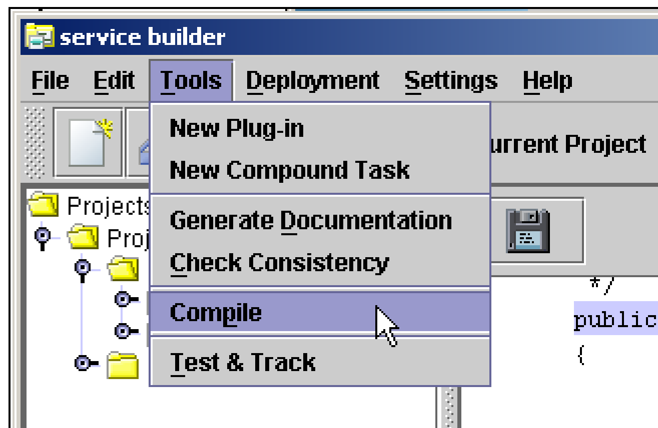


3. Click `[Add]` to add new lines to the table:
   - Type the name of the parameter in the `Attribute` column and the value of the parameter in the `Value` column.
   - Service Activator will prepend `ACTIVATOR_` to the `Attribute` name you define when it exports the parameters to the target machine environment.
   - A script running on the target can then use these environment variables.
   - To remove an attribute, select it, and then click `[Remove]`.
4. Click `[Finish]` when you have completed this task.

Compiling a Plug-in

1. Save all of the files that you intend to compile into your PAR.
2. Click the plug-in (in the plug-in directory structure) that you want to compile.
3. Select `Tools`, and then select `Compile`. This will compile all the Java source code files located in the `Java Classes` folder of your plug-in.

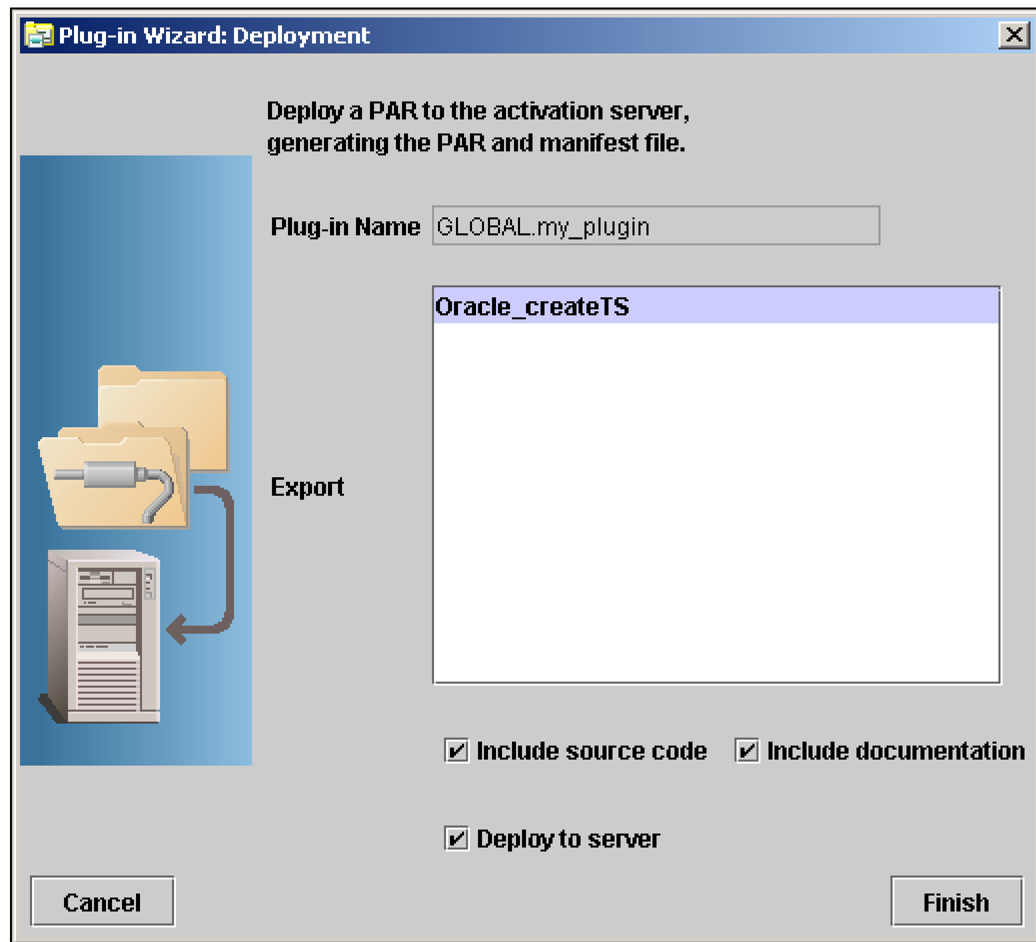
Figure 2-15 Service Builder Compile Menu Selection



Deploying a Plug-in

1. Click `Deployment`, and then select `Synchronize with Server`. This connects you to the activation server.
2. Select the plug-in archive that you want to deploy, and then select `Deployment->Deploy Plug-in Archive`.

Figure 2-16 Service Builder Deployment Dialog



3. In the `Deployment` dialog box, specify the atomic tasks you want to make available (publish) when you deploy the PAR.
4. Check the items that are appropriate for your deployment:
   - `Include source code` – includes the source code with the PAR file deployment.
   - `Include documentation` – includes the documentation with the PAR file deployment.
   - `Deploy to server` – sends the PAR to the server. Having the option to deploy or not deploy is useful when you are working without a connection to the server. If you do not deploy to the server, the operation performs a consistency check on the plug-in and creates the PAR file for the plug-in.

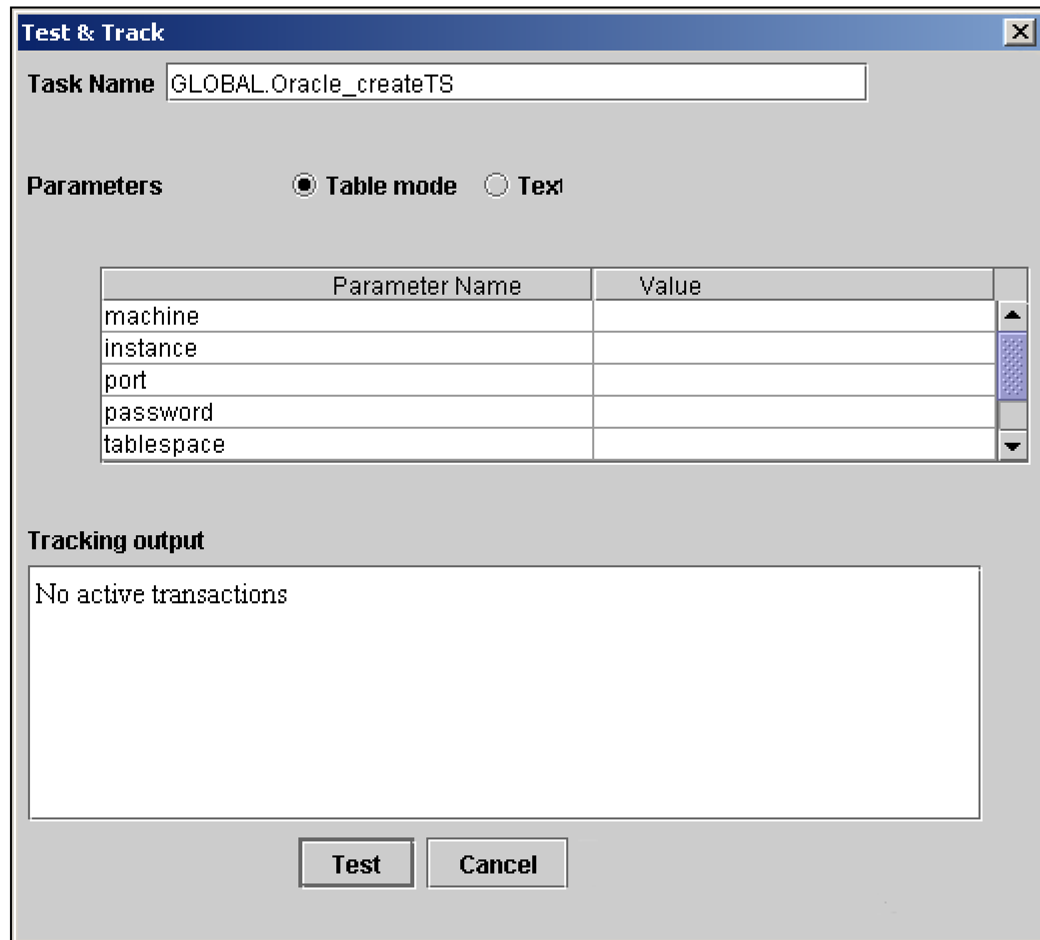
5. Click `[Finish]` to:
   a. Create the deployment descriptor (`par.xml`).
   b. Build the PAR file.
   c. Check the consistency of the PAR.
   d. Deploy the PAR file to the server.
   e. Make the atomic tasks of the plug-in available on the server.

Testing an Atomic Task

1. Right-click an atomic task in a PAR, and then select `[Test]`.
2. Enter the parameters to pass to the atomic task. There are two ways to specify them: table mode or text mode. Table mode makes it easy if you don't know the parameters for the task and want to edit the values for each parameter individually. Text mode is useful when you want to rerun a test using a list of parameters that you can paste from a cut-buffer.
   - In table mode, a value for each parameter in the table is passed to the atomic task. If the value cell for a parameter is empty or contains only white space, the value for the parameter will be passed as an empty string. If you need to pass a value that consists of only white space, enter a string with the desired white space characters surrounded by quotes (either single or double) in the value cell for that parameter. Quotes can be embedded in a parameter by surrounding the parameter with the alternate type of quote character.
   - In text mode you must enter a string with the correct number of parameters. Parameters are comma-separated or space-separated. You can embed white space within an individual parameter by enclosing it in quotes (either single or double). Quotes can be embedded in a parameter by surrounding the parameter with the alternate type of quote character (for example "`new user 'jack'`").

NOTE When a plugin is deployed, it can take up until the *`PluginMonitorPollInterval`* before the Resource Manager figure out that a new plug-in or an updated plug-in has been deployed.

Figure 2-17 Service Builder Test Dialog



3. Click `[Test]`, and monitor the activity in the tracking area.

You can switch between table mode and text mode. If you expect to test an atomic task repeatedly, you may find it helpful to switch to text mode, highlight the text of the parameters and place it into your text buffer (Ctrl-C). Then the next time you want to test the task with the same parameters you can paste the text back into the text mode editor.

Generating Plug-in Documentation

1. To document your plug-in archives, select a plug-in from the project list.
2. Click `Tools`, and then select `Generate Documentation`.
3. To view the documentation, use an external browser or the browser provided by Service Builder.

   The following example shows documentation generated by Service Builder and displayed in a Microsoft Internet Explorer browser.

NOTE If you want to fully document the entire project, select the project itself, and then generate the documentation.

Figure 2-18 Sample Plug-in Javadoc

Plug-in *my_plugin*

General | Atomic Tasks

General	
Name	my_plugin
Class Name	com.acme.plugins.my_plugin
Namespace	GLOBAL
Deployment	ON_DEMAND
Locking Arguments	1
Description	A new plug-in
General pre-provisioning tasks	*General pre-provisioning information*
Platform	*Platform here*
Version	1.0.0 (c) 2003 Copyright Hewlett-Packard Development Company, L.P.
Author	HP OpenView Service Activator ServiceBuilder

Top

Atomic Tasks	
Oracle_createTS	machine, instance, port, password, tablespace, datafile, datasize

Atomic Task *Oracle_createTS*

Top

Creates a new tablespace

Synopsis

```
Oracle_createTS (machine, instance, port, password, tablespace, datafile, datasize)
```

Using the Javadoc Tags for Documentation

The following sections describe the list of tags you can use in a plug-in *Javadoc*. Service Builder-generated code will already have these tags. All tags except `@platform`, `@author`, and `@preprov` are atomic task specific.

@platform

Use the `@platform` tag to denote the HW/OS platform of the target machine. Include any relevant OS versions. If the plug-in is supported on multiple HW/OS platforms, list them one per line by using the HTML `<br>` tag. For example:

```
HP-UX 11.11<br>
Sun Solaris 2.8
```

@author

Use the `@author` tag to specify the author and copyright notice.

@preprov

Use the `@preprov` tag to describe any preprovisioning requirements that are general to the plug-in. In addition, use this tag in your atomic task *Javadocs* to describe preprovisioning requirements that are specific to this atomic task and are not already covered in the general preprov section. Use a separate `@preprov` tag for each requirement. The generated *Javadoc* creates a bulleted list of these preprovisioning requirements.

@param

Use the `@param` tag to document ALL atomic task parameters. Again, be sufficiently detailed so that someone can use the atomic task without having to read/understand its internal implementation of code/scripts.

@do_and_check

Use the `@do_and_check` tag to describe the functional details of the atomic task.

@undo_and_check

Use the `@undo_and_check` tag to indicate whether rollback is possible or not, and what the results mean.

@warning

Use the `@warning` tag to describe any additional warnings/cautions. Use a separate `@warning` tag for each warning/caution. The generated *Javadoc* creates a bullet-list of these warnings.

@dependency

Use the `@dependency` tag to describe any dependencies that this atomic task has on other atomic tasks. For example, if atomic task `UXOS_createUser()` must be performed before this atomic task.

Using Service Builder to Create Compound Tasks

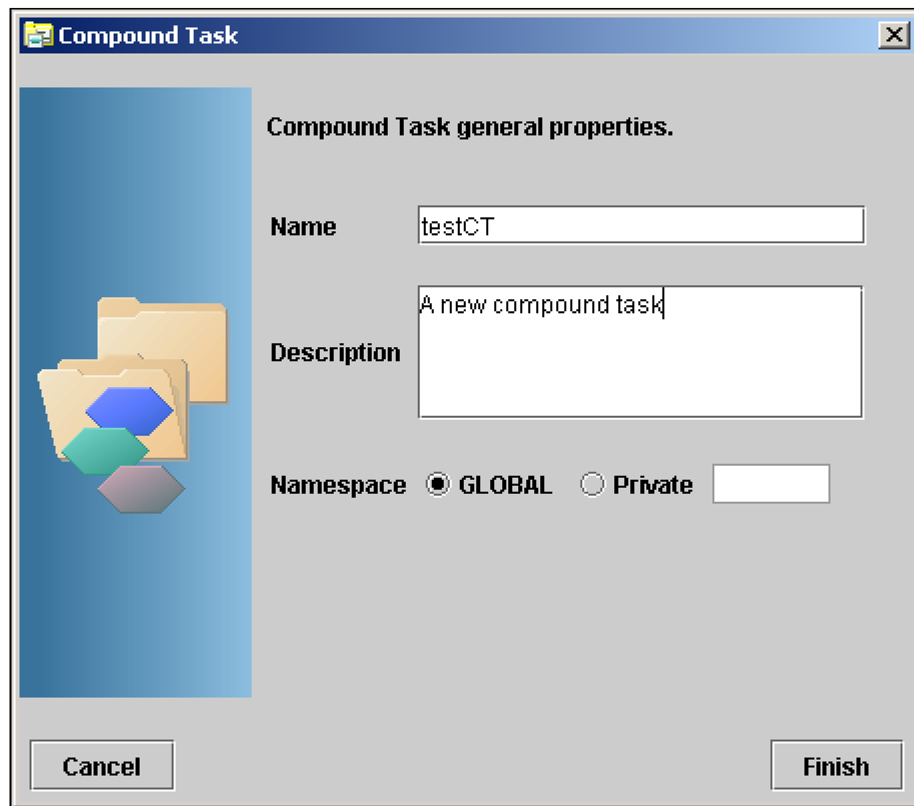
This section provides information about how to use the Service Builder tool supplied with Service Activator to construct, deploy, test, and document compound tasks. It also discusses methods you can use to maintain consistency between deployed tasks.

Creating Compound Tasks

This section provides instructions for creating, deploying, and testing compound tasks using Service Builder.

1. To create a new compound task, click `Tools`, and then select `New Compound Task`. This launches the wizard shown in Figure 2-19:

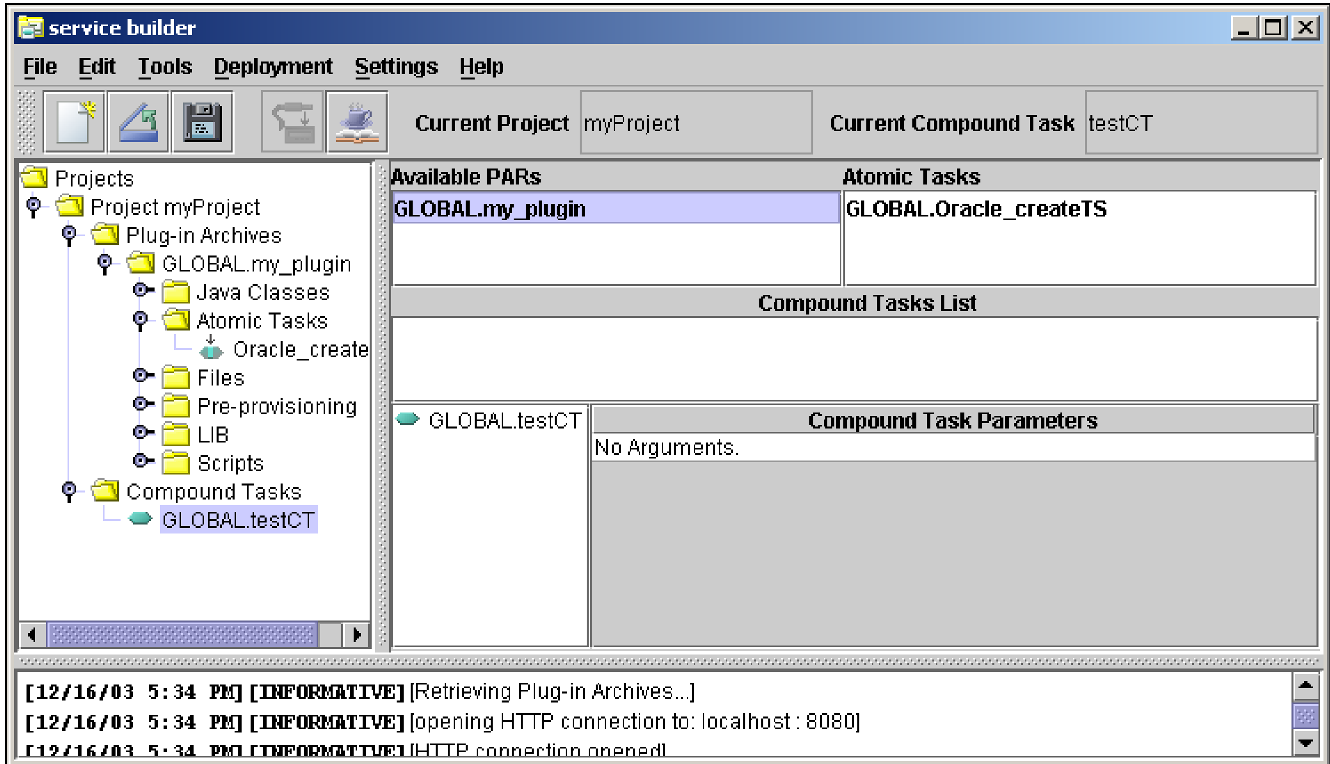
Figure 2-19 Service Builder New Compound Task Dialog



2. Type the name of the compound task.
3. Type a description of the compound task you are creating. This will appear in the *Javadocs* for this compound task. This description will also be visible in the Manage Tasks panel (under the File menu) after the compound task has been deployed.
4. Set the `Namespace` for the compound task. The combination of the `Namespace` and the compound task name must be unique across all tasks.

5. Click `[Finish]`. The wizard displays a compound task work area similar to the one shown in Figure 2-20.

Figure 2-20 Service Builder Compound Task Work Area



Adding Tasks to a Compound Task

Use the following steps to add atomic tasks to your compound task:

1. View the available PARs in the top left section. This always shows you the list of PARs that were deployed from the last time you synchronized with the server.

2. Click a PAR to see the list of its atomic tasks in the top right section.

3. Drag and drop an atomic task from the list onto your compound task. Drop it onto the *name* of the compound task (not onto the list below the compound task).

Your updated task will look similar to the one shown in Figure 2-21. When you save your project, your directory structure will look similar to the one shown in Windows Explorer in Figure 2-22.

Figure 2-21 Service Builder Compound Task Work Area After Adding Atomic Tasks

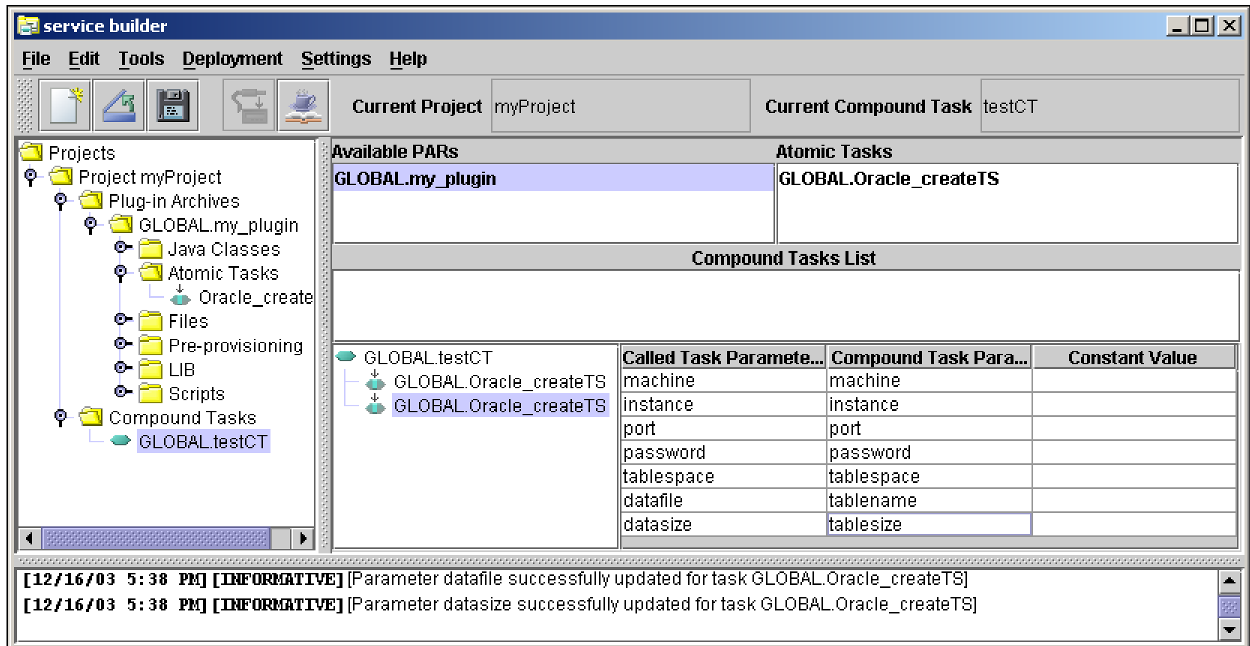
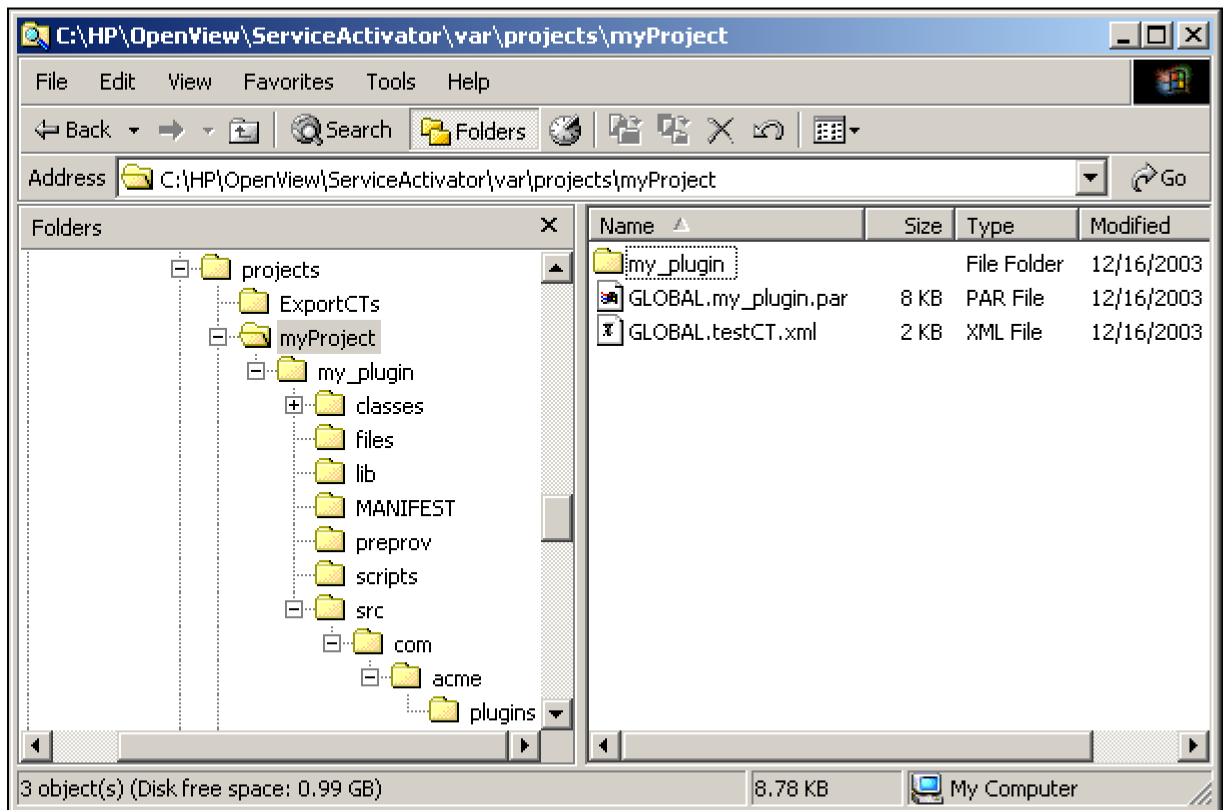


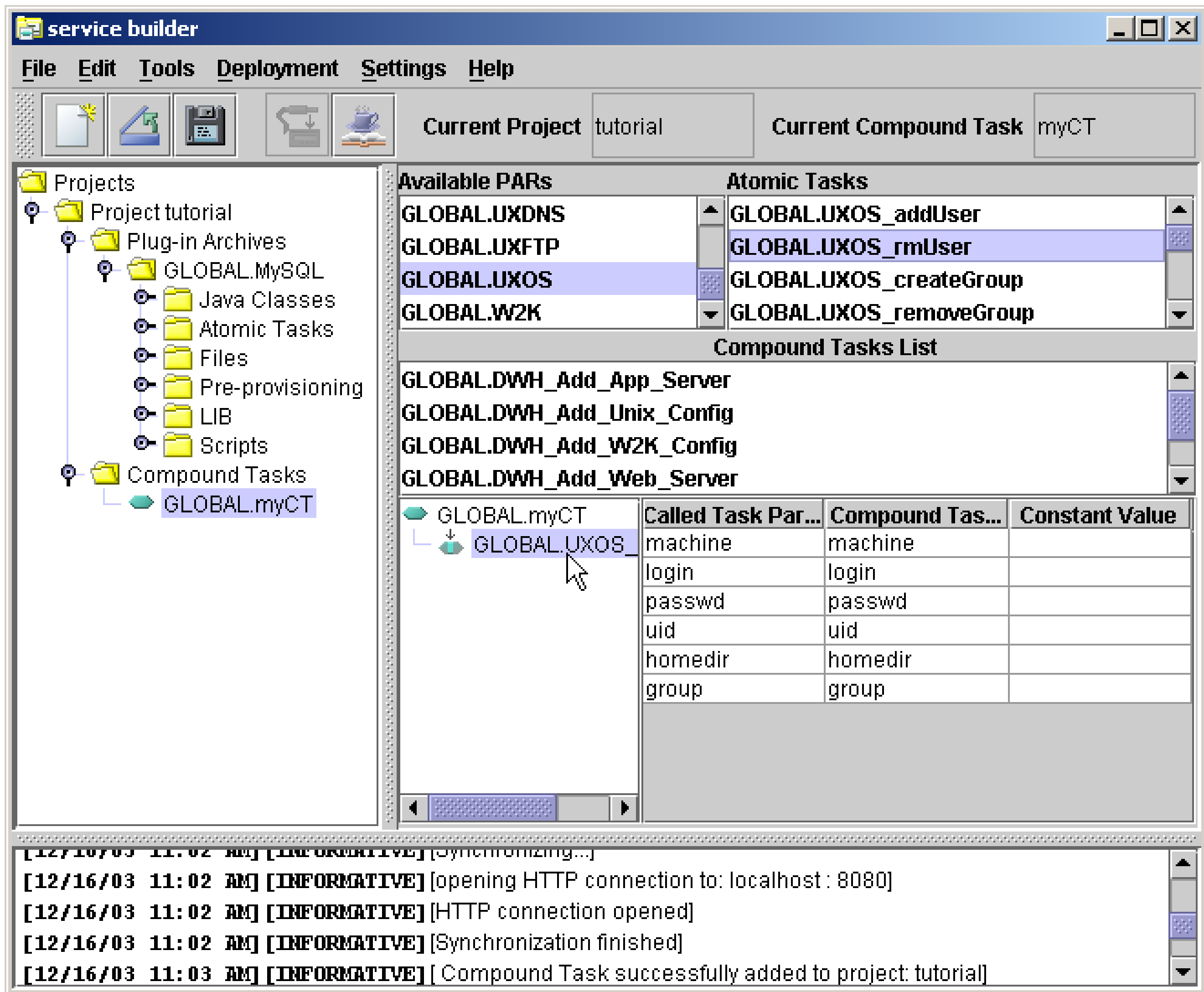
Figure 2-22 Project Directory Structure - After Adding a Compound Task



You can also add other compound tasks to your compound task, as shown in Figure 2-23. Use the following steps to do this.

1. View the available compound tasks in the Compound Tasks List. This always shows you the list of compound tasks that were deployed from the last time you synchronized with the server.
2. Drag and drop a compound task from the list onto your compound task.

Figure 2-23 Compound Task Work Area with Other Compound Tasks Available



Modifying Compound Tasks

In the following sections we will refer to the **calling task** and the **called tasks**. The calling task is the compound task that we are creating/editing. The called tasks are those atomic or compound tasks that are being invoked as part of the compound task we are editing.

Reordering Tasks in the Compound Task

The order that the tasks in the compound task appear is the order in which they will be called when the compound task is invoked. As tasks are added to the compound task, they are always placed at the end of the list. To reorder the tasks, drag a task and drop it again on the compound to have it placed at the end of the list.

Removing a Called Task

To remove a called task, right-click the task, then select `Remove Task`.

Changing Parameter Mappings

Each called task has a list of parameters that must be passed to it. The calling task also has a list of parameters which is the union of all parameters for the called tasks.

Frequently, one or more called tasks will have a parameter name that is the same (for example, *machine*). The Compound Task Editor automatically assumes that any called task parameters with the same name should come from the same parameter in the calling task. This may not be the desired behavior and can be overridden.

Conversely, some called task parameters may have different names, but should come from the same parameter from the calling task. This can also be specified.

1. Click one of the called tasks to see a list of the parameters this task expects. Notice there is a column for the name of the parameter in the called task. This column is not editable. The second column, which *is* editable, shows the name of the parameter in the calling task.
2. Click the calling task to see the unified list of parameters. Notice that any parameters from the called tasks with the same name appear only once in the calling task parameters.
3. If you want to make two called task parameters come from the same calling task parameter, give them the same name. If you want two called task parameters to come from different calling task parameters, give them different names. You should use descriptive names (e.g. `dnsServer` instead of `machine2`).

Setting Constant Values for Called Task Parameters

One significant value of compound tasks is the ability to set constant values for some of the parameters in the called tasks. For example, in the `W2K_addDir` atomic task, the third parameter specifies the name of a skeleton directory that will be used to populate the new directory being created. In many cases, you don't want to use this behavior, so you pass an empty string to this parameter. But rather than having to pass an empty string from the workflow engine, you can set the value of this parameter in the compound task to be an empty string.

1. Click the called task that contains the parameter you want to set to a constant value.
2. Enter the value for the parameter in the "Constant Value" column. Use two double quotes to set an empty value (or a value containing only white space).

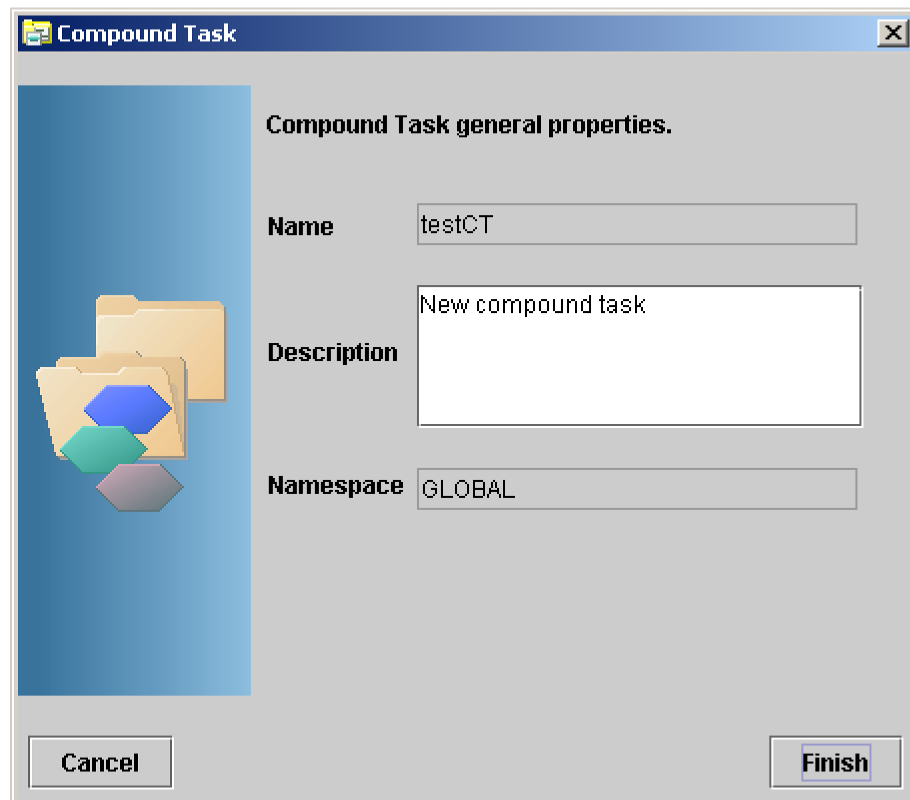
3. Click the calling task to see that the parameter no longer appears in the list of calling task parameters. Note that the parameter *will* appear if there is another called task that has a parameter of the same name. The constant value only applies to the one called task parameter, not all parameters with the same name.

NOTE You cannot set a constant value for *all* of the parameters of a compound task. The resulting compound task must take at least one non-constant parameter. You will get an error when you attempt to deploy a compound task that does not have any non-constant parameters.

Viewing the Properties of a Compound Task

1. To view the properties of a compound task, right-click the service in the `Compound Tasks` folder, and then select `Properties`.

Figure 2-24 Service Builder Compound Task Properties Dialog



2. You may change the description. You cannot change the name or the namespace of the compound task after it has been created.

Importing and Exporting a Compound Task

You can export a compound task to an XML file to facilitate sharing and to allow manual edits to the compound task (see “Creating Compound Tasks Manually: Advanced Tips” on page 85). Similarly, you can import a compound task from an XML file into your project.

To export a compound task, follow these steps:

1. Right-click the compound task in the `Compound Tasks` folder.
2. Select `Export to XML`.
3. Choose a destination directory for the file.

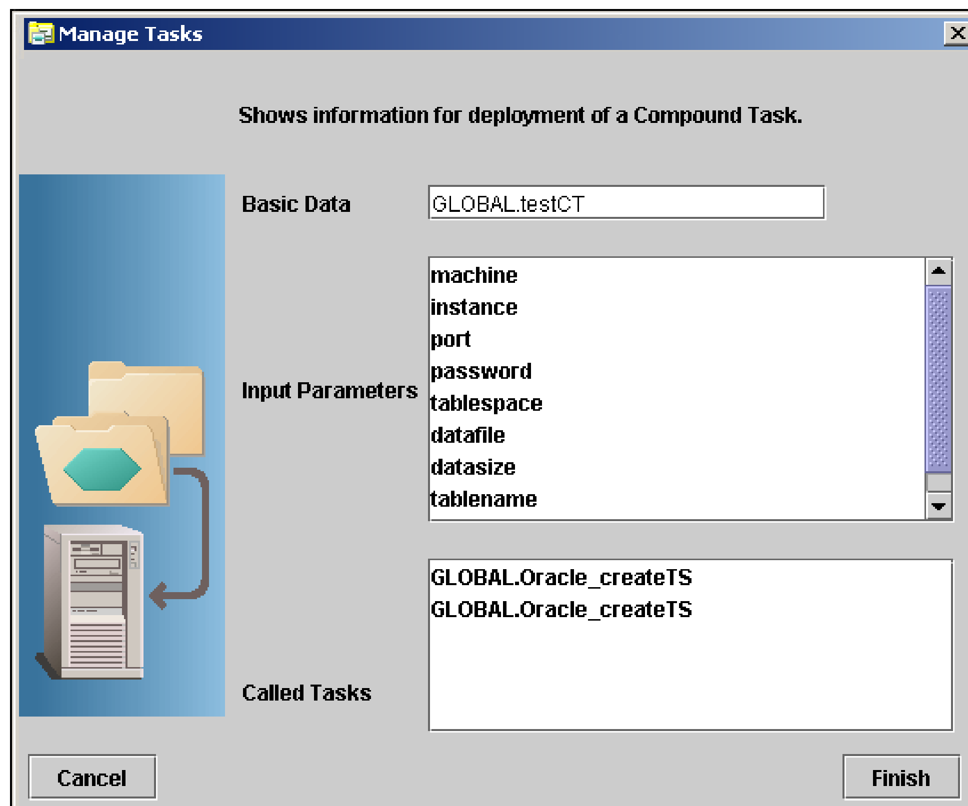
To import a compound task into an existing project, follow these steps:

1. Right-click the `Compound Tasks` folder.
2. Select `Add Compount Task`.
3. Browse to the XML file that you want to import, select it, and click `[Open]`. Check the message at the bottom of the window to see if the import operation was successful.

Deploying a Compound Task

1. Click `Deployment`, and then select `Synchronize with Server`. This connects you to the activation server.

Figure 2-25 Service Builder Compound Task Deployment Dialog



2. Click the compound task you want to deploy.
3. Click Deployment from the menu bar, and then select Deploy Compound Task. The Deployment window shows the list of parameters and called tasks for the compound task.
4. Verify that all of the values are what you expected:
   - Click [Finish] to deploy the task
   - Click [Cancel] to cancel the deployment and make changes to the task.

   If the connection settings are defined and the connection is available, clicking [Finish] deploys the compound task to the deployment engine and makes the task available on the server.

Testing a Compound Task

Testing a compound task is very similar to testing an atomic task.

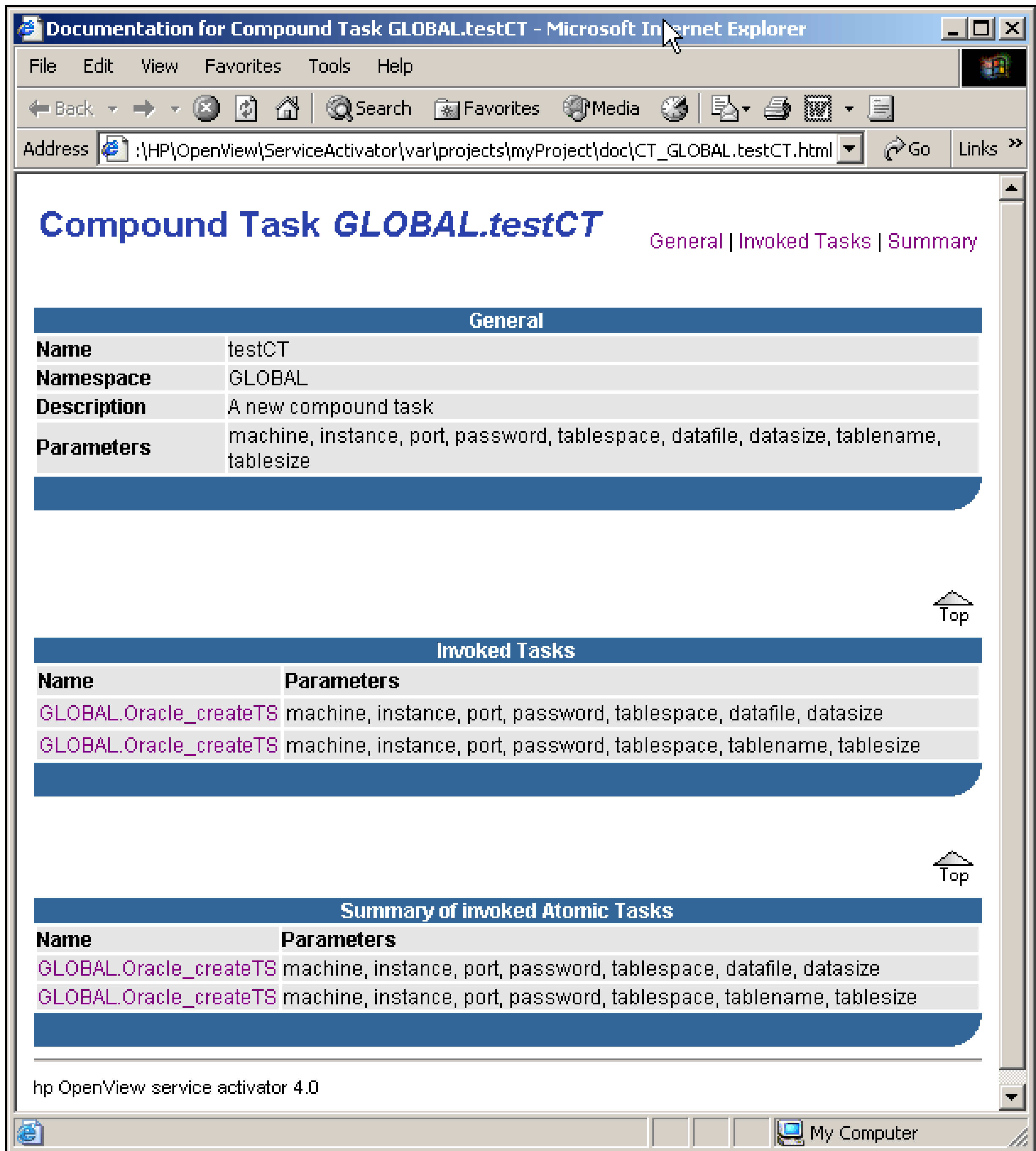
1. Right-click the compound task in the Compound Task folder in the project tree, and select Test.
2. Follow the same instructions as in “Testing an Atomic Task” on page 52.

Documenting a Compound Task

1. Select the compound task that you want to document.
2. Click Tools, and then select Generate Documentation. This generates the documentation for the compound task in HTML format.
3. Use a browser to view the documentation, or right-click the compound task, and then select Documentation.

Figure 2-26 shows an example of the documentation generated by Service Builder in a Microsoft Internet Explorer browser.

Figure 2-26 Sample Javadoc for a Compound Task



Maintaining Consistency Between Deployed Tasks

The process of developing atomic and compound tasks is typically iterative and involves developing an initial version, deploying it, making changes, redeploying, and so on. During this iterative process, you might make changes to an interface of an atomic task that is called by a previously deployed compound task. This would cause consistency issues for the compound task.

To avoid these consistency issues, when you change an atomic or compound task, Service Builder checks all previously deployed compound tasks that depend on that task to ensure that the task and all tasks that call it still have consistent interfaces.

If Service Builder identifies a consistency issue, it will move all dependent compound tasks to an `INVALID` state, meaning that you cannot use them until you resolve the consistency errors. Service Builder will notify you of the compound tasks that it has moved to an `INVALID` state. You can also identify which compound tasks are `INVALID` by bringing up the `Manage Tasks` dialog box, which lists all the compound tasks currently deployed on the activation server.

NOTE When you are notified that a task has been invalidated, you should download the original specification of the task in the Manage Tasks dialog box. The original specification of the invalidated task will be deleted when Service Activator is restarted.

After you fix the consistency issues, redeploy the tasks.

Configuring Authentication or Authorization

The default deployment engine configuration allows anyone to deploy PARs or Compound Tasks. You can configure the deployment engine to restrict deployment access. Use the following steps to enable authentication/authorization restrictions when deploying PARs and compound tasks:

1. Enable authentication/authorization in the workflow manager (see “Required and Typical Workflow Manager Modules” on page 340 in *HP Service Activator—Workflows and the Workflow Manager*).
2. Enable authorization in the deployer WAR file:
   a. Edit the file `$JBOSS_DEPLOY/hpsa.ear/deployer.war/WEB_INF/web.xml`
   b. Modify the value of the `authenticate` parameter, setting it to “`true`.” This reconfigures the Java servlet that services requests to deploy PARs and compound tasks.
   c. Restart Service Activator.
3. Ensure that any users who should be allowed to perform deployments exist in the “deployer” role. This may be accomplished in one of the following ways:
   - Assuming you are using one of the built-in authentication modules (HPUXAdvancedAuthModule, SolarisAdvancedAuthModule, WindowsAdvancedAuthModule, DatabaseAdvancedAuthModule), you may create a group in the OS called “deployer” and assign the appropriate users to that group.
   - Assuming that the authentication module you are using supports role-mapping (all of the built-in modules support this), then you can simply supply a role mapping from the “deployer” role to the roles/groups in your authentication domain. See the discussion of role mapping files in *Chapter Roles, Privileges and Authentication* of *HP Service Activator - System Integrator’s Overview*
4. Start Service Builder.
5. Click `Settings`, and then select `Activation Server`.
6. Specify the user and password information in the configuration parameters for the activation server.

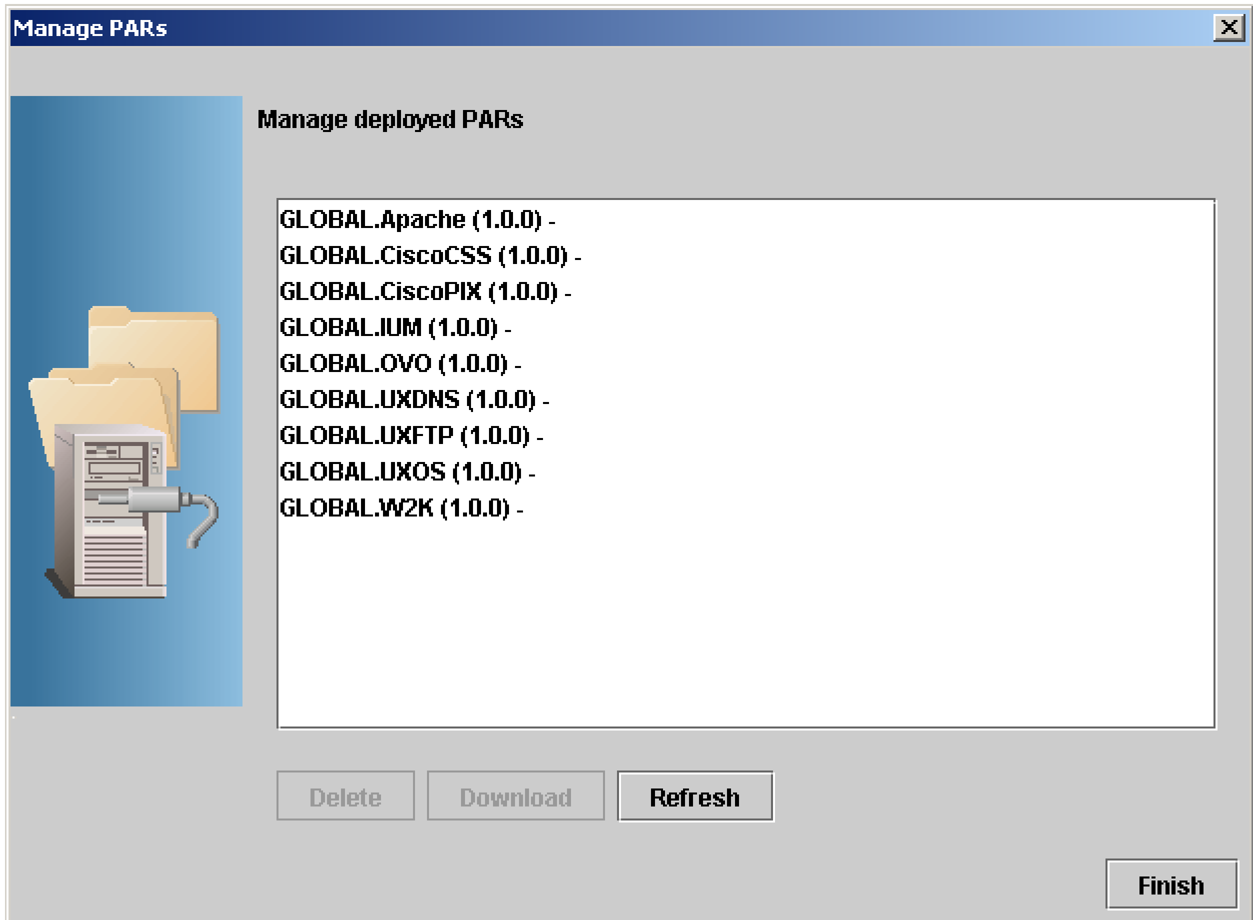
   The user information is stored persistently in the file `$ACTIVATOR_ETC/config/service_builder.xml`. Make sure to restrict read access to this file.

Once you have completed steps 1-3, then anyone wishing to deploy PARs or compound tasks must be a member of the “deployer” role and must configure Service Builder (as per steps 4-6) to supply the proper username and password.

Using Service Builder to Manage Plug-in Archives and Compound Tasks

To manage plug-in archives that are already deployed on the server, click `File`, then select `Manage Plug-in Archives`.

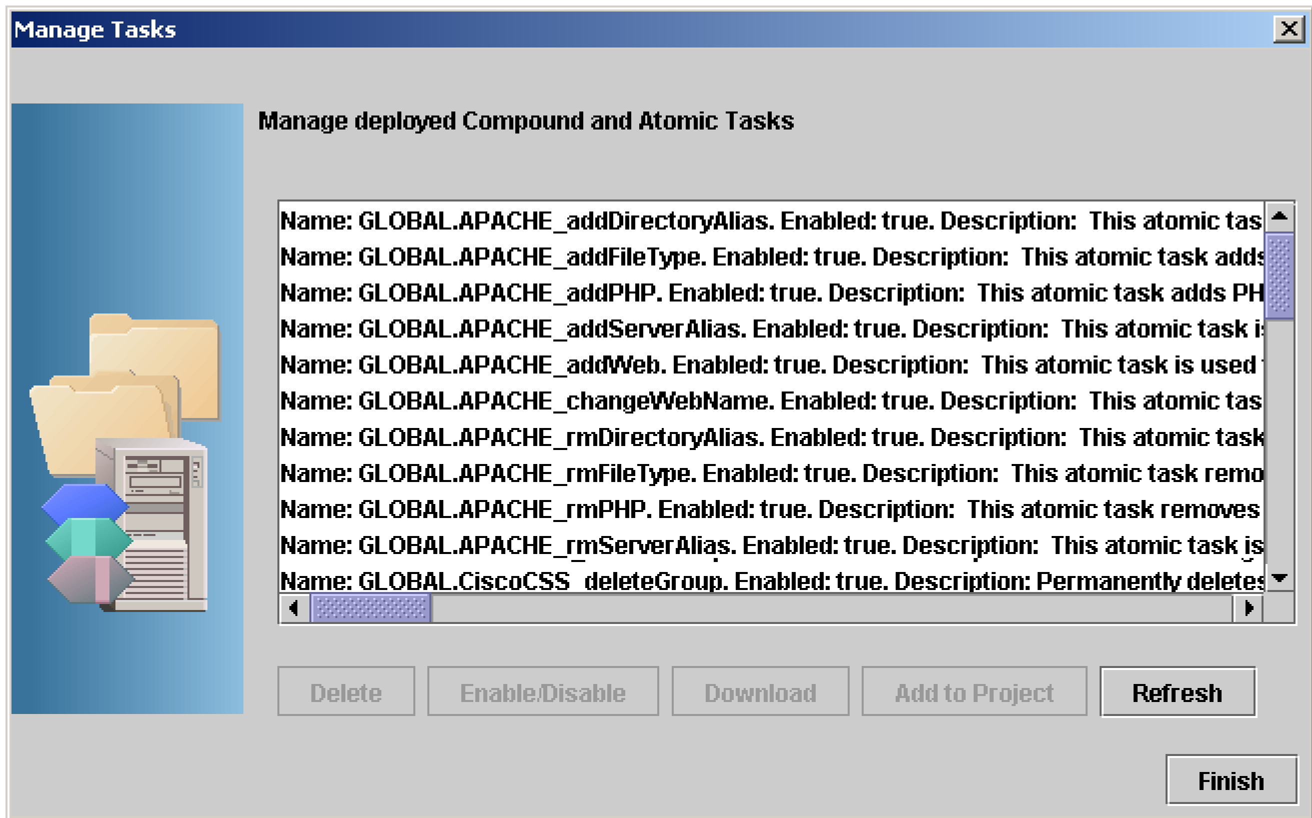
Figure 2-27 Service Builder Manage Plug-in Archives Dialog



From the `Manage PARs` dialog, you can download a plug-in archive or delete it. Downloading a plug-in archive allows you to edit and update it. Deleting a plug-in archive removes the PAR and all references to it. This means that the atomic tasks in this plug-in are no longer usable by any compound tasks. This will invalidate any compound tasks that call it.

The `Manage Tasks` dialog box allows you to delete existing compound or atomic tasks, enable and disable them, and export compound tasks to XML.

Figure 2-28 Service Builder Manage Tasks Dialog



Disabling a task means that it cannot be invoked. Attempting to invoke it will cause an activation error. If a compound task contains either a compound or atomic task that is disabled, then invoking the calling task will cause an activation error.

Setting Service Builder Configuration

You can change some aspects of Service Builder's behavior from the `Settings` menu.

From the `Service Builder` menu item you can set:

- The default home directory for new projects that you create (defaults to *$ACTIVATOR_VAR*/projects).
- The default directory where compound tasks are exported to (defaults to *$ACTIVATOR_VAR*/projects/ExportCTs).
- Whether to show plug-in and compound task documentation in an external browser window or in the right-hand pane of the Service Builder GUI.

From the `Activation Server` menu item, you can set the details necessary for Service Builder to contact the activation server. This includes:

- The host name and port where the activation server can be reached. This defaults to `localhost` and port `8080`. Note that the port is the HTTP port for the web server that accepts and forwards requests to the activation server.
- Optionally, you may specify a proxy host and port if it is necessary to go through a web proxy to reach the activation server from the machine where Service Builder is running. By default, no proxy information is configured.
- Optionally, you may specify a username and password for connecting to the activation server. By default, the activation server is not configured to perform any authorization regarding who may deploy new plug-ins or compound tasks. If you have configured the activation server to perform such authorization, then you will need to put an appropriate username and password here.

  Note that the username and password are stored in clear text (not encrypted) in the `service_builder.xml` file. See Appendix D, “Security Considerations,” on page 63 for additional information about protecting access to this file.

Using Service Builder from the Command Line

Some of the operations of Service Builder are available via its command line. Issue the **-help** option to see the list of invocation options that are supported. Here is the list of available options:

Table 2-1 Common Service Builder Command Line Options

Option Syntax	Purpose
-help	Displays a list of the valid options
-version	Displays the version information
-host ***<hostname>***	Sets the Service Activator server host for this invocation of Service Builder only
-port ***<port>***	Sets the Service Activator server port for this invocation of Service Builder only
-username ***<user>***	Sets the username with which to connect to the Service Activator server for this invocation of Service Builder only
-password ***<password>***	Sets the password with which to connect to the Service Activator server for this invocation of Service Builder only
-packPAR ***<dir> <file.par>***	Packs a directory into a plug-in archive
-verifyPAR ***<file.par>***	Verifies the contents of a plug-in archive to ensure that the actual contents match the PAR manifest
-listPAR	Retrieves the list of available plug-in archives from the repository
-deployPAR ***<file.par>***	Deploys a plug-in archive to the repository
-deletePAR ***<name> <version>***	Deletes a plug-in archive from the repository
-downloadPAR ***<name> <version> [<destination>]***	Downloads a plug-in archive from the repository to the given destination
-listPARVersion	Prints out the current version information of the Plug-in Archice file
-compilePAR ***<dir>***	Compiles the Java classes associated with the plug-in in the given directory
-compilePARdebug ***<dir>***	Compiles the Java classes associated with the plug-in in the given directory with debug turned on

Table 2-1 Common Service Builder Command Line Options (Continued)

Option Syntax	Purpose
-docPAR ***<dir>***	Generates the *Javadoc* documentation for the plug-in in the given directory
-listTasks	Retrieves the list of available tasks (atomic and compound) that are deployed in the repository
-addCT ***<task.xml>***	Deploys a compound task to the repository
-deleteCT ***<task>***	Deletes the given compound task from the repository
-enableTask ***<task>***	Enables the use of the given task that has already been deployed to the repository
-disableTask ***<task>***	Disables the use of the given task
-exportCT ***<task> [<destination file>]***	Creates an XML file that describes the given task that has been deployed in the repository
-activate ***<task> <params>***	Causes the given task to be invoked with the given parameters

3 Creating Customized Plug-ins and Compound Tasks

This chapter explains advanced tips about how to create your own customized plug-ins and compound tasks.

Creating Plug-ins: Advanced Tips

This section provides key information you need to generate, configure, and run a plug-in. Some of this information is specific to manually creating a plug-in without Service Builder, while other information is of general importance to help you create your own plug-in. You should have some experience creating plug-ins using Service Builder before you attempt to create one manually.

The code used in these examples was created for example purposes only and is not generally useful. The simplest way to create your own plug-in (without using Service Builder) is to modify the `par.xml` file of an existing plug-in:

1. Unpack the existing plug-in using jar:

   `jar xf` ***`Plugin`*** **`.par`**

   This creates the plug-in directory structure discussed in “Understanding Plug-in Archives” on page 24.

2. Use the information provided in the following sections to make modifications to the `par.xml` file.

Plug-in Java Class

You will find the plug-in Java class source code in the `src` directory. In this example, the plug-in name is Example, so the Java class source code is the file `Example.java`.

1. Import statements

   The plug-in source code needs to import the necessary classes into the local namespace. These include:

   - `import com.hp.ov.activator.resmgr.*;`
   - `import com.hp.ov.activator.resmgr.par.*;`
   - `import com.hp.ov.activator.util.*;`
   - `import java.net.*;`

2. Javadoc comments

   You can add *Javadoc* comments to your plug-in source code and use Service Builder to generate that code into a *Javadoc*. Following is an example of *Javadoc* comments nested within Java code:

```
/**
* Plug-in wizard automatically generated code (Mon Jan 13 10:22:52 MST 2003)
* <p>
* Example plug-in to demonstrate basic concepts. The atomic task
* presented here is for demonstration purposes only.
*
* @platform <i>HP-UX 11i,
*         Solaris 2.8</i>
*
* @author  HP Service Activator ServiceBuilder.
* @version 1.0.0
*/
```

3. Plug-in class declaration

The plug-in class declaration must extend the class `PARPlugin`. Your code entry should look like this:

```
public class Example extends PARPlugin
{
```

4. Plug-in init method

   The method `init` contains some user-added code to get the value of the configuration parameter `PASSWORD_FILE`. The `AttributeTable` object provides each of the configuration parameters defined in `par.xml`.

```
    /**
     * Method invoked prior to any atomic task call.
     *
     * @param config The plug-in configuration object.
     */
    public void init( AttributeTable config ) throws PluginException
    {
        // Store the config object for later use
        super.init( config );

        // Insert your code here
        String attribute = config.getAttribute("PASSWORD_FILE");
        if (attribute.length() != 0) {
            password_file = attribute;
        }
    }

    /**
     * Method invoked before destroying the plug-in.
     * <p>
     * Intended for resource clean-up.
     */
    public void destroy()
    {
        // Insert your code here
    }
}
```

5. Plug-in atomic task methods

   a. Name

      Each atomic task in the plug-in is defined by the methods that begin with `task_`. The example shown below contains the atomic task method called `task_Example_userExists`. The actual atomic task name is `Example_userExists`. The methods that define atomic tasks must return an `ExecutionDescriptor` object and throw the `PluginException`.

```
    public ExecutionDescriptor task_Example_userExists (int op, String machine,
                              String username) throws PluginException
```

   b. Operation and return codes

      The first parameter to every atomic task defined in the plug-in is the operation. The operation is either `DO_AND_CHECK` or `UNDO_AND_CHECK`. During the transaction that calls the atomic task (see plug-in life-cycle), the `DO_AND_CHECK` operation is used to run the atomic task. The `UNDO_AND_CHECK` operation is used to roll back the atomic task in the event of a failure during a compound task. This will leave the target machine in its preactivation state.

The ExecutionDescriptor returned from an atomic task provides information about the result of the atomic task call. An OK response indicates that an atomic task completed successfully. An ERROR response indicates an error, and will trigger a rollback call.

The precise definitions of the different ExecutionDescriptors that may be returned by an atomic task are listed in Table 3-1. When designing your own atomic task, you should follow these conventions.

Table 3-1 Atomic Task Return Codes

majorCode	minorCode	Condition when should be used
ExecutionDescriptor.OK	ExecutionDescriptor.NONE	OK/NONE will be returned by an atomic task when the requested operation (DO_AND_CHECK or UNDO_AND_CHECK) completed successfully.
ExecutionDescriptor.ERROR	ExecutionDescriptor.CONSISTENT	ERROR/CONSISTENT will be returned by an atomic task when the requested operation (DO_AND_CHECK or UNDO_AND_CHECK) was unable to complete successfully, but the target system was successfully restored to the state it was in before the requested atomic task operation was invoked. Note that for UNDO_AND_CHECK, an ERROR/CONSISTENT return implies that the net effect of the UNDO_AND_CHECK call involved no change to the target system - it does not imply that the system is restored to the state it was in before the DO_AND_CHECK call.
ExecutionDescriptor.ERROR	ExecutionDescriptor.INCONSISTENT	ERROR/INCONSISTENT will be returned by an atomic task when the requested operation (DO_AND_CHECK or UNDO_AND_CHECK) was unable to complete successfully, and the target system was not restored to the state it was in before the requested atomic task operation was invoked.

An ERROR response from an atomic task's DO_AND_CHECK call triggers a rollback of the compound task. This rollback involves an UNDO_AND_CHECK call to all previously executed atomic tasks in the compound task. The UNDO_AND_CHECK calls occur in reverse order; the first atomic task in the compound task will be rolled back last. Note that UNDO_AND_CHECK is not called for the atomic task that returned the ERROR that caused the rollback. Before returning the error, it is the responsibility of the DO_AND_CHECK call to attempt to restore the target system to

the state it was in before the DO_AND_CHECK call. During rollback, each atomic task's UNDO_AND_CHECK response will be logged, but an ERROR will not stop the rollback from continuing.

It is important to understand how the return code of an atomic task maps to the return code of the compound task. If all atomic task DO_AND_CHECK operations in a compound task return OK/NONE, the compound task will return OK/NONE. If an atomic task DO_AND_CHECK operation in a compound task returns an ERROR response, the compound task will return this ERROR response. Any errors during rollback are not included as part of the compound task return.

In general, an atomic task should not return OK if it detects that the specific operation has already been performed. By following this convention, you'll avoid the possibility of rolling back an operation that was performed as part of a separate compound task. For example, imagine an atomic task whose DO_AND_CHECK operation adds a user to a UNIX machine (its UNDO_AND_CHECK operation will remove the user from the UNIX machine). A previous compound task (executed months ago) may have added "activatorUser" to the UNIX machine "unix1." When a new compound task tries to add "activatorUser" to "unix1," it is important that this atomic task return ERROR—otherwise, the compound task would continue and could potentially rollback, causing the "activatorUser" to be removed from "unix1." This type of behavior would violate transactional semantics, so it is important that you follow these conventions.

Atomic tasks can also throw Java exceptions. Any exception thrown from an atomic task will be caught by the Resource Manager and logged. The Resource Manager considers a thrown atomic task exception to be equivalent to an ERROR/INCONSISTENT return. Each atomic task has a PluginException listed in its "throws" clause. You can throw a PluginException from your atomic task to signal an unrecoverable error. Note that it is your choice whether to throw a PluginException or return an ERROR ExecutionDescriptor when a failure condition has been detected in your atomic task. However, an ExecutionDescriptor can provide more information than a PluginException (that is, majorCode, minorCode, stdout, stderr, description), and is generally preferred.

In some cases, it is very difficult (if not impossible), to properly implement UNDO_AND_CHECK for an atomic task. This is often the case for "negative" atomic tasks. For example, if your atomic task removes a directory as part of its DO_AND_CHECK operation, it may not be possible to completely restore the deleted directory in the UNDO_AND_CHECK operation. As a general policy, when an atomic task cannot guarantee that an UNDO_AND_CHECK call will return the target system to its initial state, ERROR/CONSISTENT is returned, and an undo is not attempted.

```
// Check which operation the atomic task has to perform
switch (op) {
    case DO_AND_CHECK:
        // Insert your code here
        return context.executeScript( "do_Example_userExists_script",
                        machine, new String[]{username} );
        break;

    case UNDO_AND_CHECK:
        // Insert your code here
        return new ExecutionDescriptor( ExecutionDescriptor.ERROR,
                        ExecutionDescriptor.CONSISTENT, "", "",
                        "Cannot perform UNDO operation for this task.");
        break;

    default:
        throw new PluginException( "Operation not supported" );
}
```

Executing Scripts and Commands

Many atomic tasks need to execute a script or command on the target machine. The `context.executeScript()` and `context.executeCommand()` methods are provided to perform this function. These calls return an `ExecutionDescriptor` based on the exit code of the script or command. Table 3-2, Mapping Exit Codes to the Execution Descriptor, provides the mapping between exit codes and the `ExecutionDescriptor` returned by these methods. Follow these conventions when you write your own scripts or commands to be invoked in this manner.

Table 3-2 Mapping Exit Codes to the Execution Descriptor

Exit Code	**`ExecutionDescriptor` returned by `context.executeScript()`**
0	majorCode=`OK`, minorCode=`NONE`
1	majorCode=`ERROR`, minorCode=`CONSISTENT`
2 (or anything else)	majorCode=`ERROR`, minorCode=`INCONSISTENT`

The script is named in the deployment descriptor (see “MANIFEST” on page 25). Here is an example Perl script with file name `userExists.pl`.

```
# Perl script
#
# Checks if a user is defined in a password file
#
# ARGV[0] username
#
# Returns 0 if user is found
# Returns 1 if user is not found
#
# check no of arguments
if ($#ARGV != 0) {
    print "Usage: $0 <username>\n";
    exit 1;
}
```

```
# get arguments
$username = $ARGV[0];

# get value of passwordFile from configuration parameter if it exists
$passwordFile = "/etc/passwd";
if (defined $ENV{'ACTIVATOR_PASSWORD_FILE'}) {
    $passwordFile = $ENV{'ACTIVATOR_PASSWORD_FILE'};
}

# open passwordFile
if (! open passwordFile) {
    print "$passwordFile not found!\n";
    exit 1;
}

# search for username
while (<passwordFile>) {
    if (m/^$username/) {
        # username found
        close passwordFile;
        exit 0;
    }
}
# username not found
close passwordFile;
print "$username not found in $passwordFile!\n";
exit 1;
```

Capturing Output from Scripts and Commands

When you use `context.executeScript()` or `context.executeCommand()`, all of the output from the process is captured in the `ExecutionDescriptor` that is returned. The standard output and standard error from the process are captured in the `stdout` and `stderr` fields of the `ExecutionDescriptor`. You can either simply return this `ExecutionDescriptor` to the caller (the ResourceManager), or you can edit these fields in some way as desired.

Saving Data in the Database

The plug-in framework allows uploading of data back to the Workflow Manager. But, uploading large amount of data may lead to memory issues. Hence, the framework allows the plug-in to write data into the database during its execution, which can be read later during execution of rest of the workflow nodes. This is the way to pass large amount of data from a plugin to the Workflow Manager, in case Service Activator is running in a clustered environment due to that the data is possible to read from all cluster nodes which is not the case if the information is passed in a file.

The plug-in can invoke the following method on the `PARContext` to save the data in the database. The data is stored in the DATABASE_MESSAGE table.

```
String saveData(String messageUrl, long pos, byte[] data, int offset, int
len, String messageIdKey)
```

The message id is returned and has the value `db:<message id>`. The message id is also uploaded back to the Workflow Manager as a key value pair; the key being the value specified by the parameter `messageIdKey` and the value being `db:<message id>`.

Optionally, you can specify the message id with the syntax, `db:<message id>`, if data has to be updated. In this case, the position from where data is to be updated has to be specified.

In the Workflow Manager, the message id can be retrieved from the parameter "*uploaded_data_var*" in the Activate node. Since the data is uploaded as key value pair, use the `messageIdKey` to extract the message id.

The following exceptions must be handled:

PluginException thrown if incorrect message url is specified, or an exception occurs when storing the data.

InterruptedException thrown in case of a database connectivity error.

The below example shows how the data can be saved from an atomic task:

```
public ExecutionDescriptor task_TestSaveData_taskSaveAndUpdate (int op,
String pluginDataToSave, String pluginDataToUpdate) throws PluginException,
InterruptedException
{

// Check which operation the atomic task has to perform
switch (op) {
case DO_AND_CHECK:
int pos=0;
String messageId="db:0";
int offset=0;
Object obj=new String(pluginDataToSave);
byte[] data=null;
try{
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(baos);
oos.writeObject(obj);
oos.flush();
data=baos.toByteArray();
baos.close();
oos.close();
}catch(Exception e)
{
                   return new ExecutionDescriptor(ERROR_MAJOR,
ERROR_MINOR,null,                                                    null,
"taskSaveAndUpdate not executed");
}
//invoke saveData() on the PARContext
messageId=context.saveData(messageId,pos,data,offset,data.length,"message_id");
//print the message id
context.logInfo("the message message after saving
pluginDataToSave::"+messageId);
//To append the string contained in pluginDataToUpdate to the saved data
//invoke the saveData() with the newly created message id to update the data
pos=data.length+1;
obj=new String(pluginDataToUpdate);
try{
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(baos);
oos.writeObject(obj);
oos.flush();
data=baos.toByteArray();
baos.close();
```

```
oos.close();
}catch(Exception e){
return new ExecutionDescriptor(ERROR_MAJOR, ERROR_MINOR, null, null,
"taskSaveAndUpdate not executed");
}
//invoke saveData() again
messageId=context.saveData(messageId,pos,data,offset,data.length,"message_id");
//print the message id, which should be the same as earlier
context.logInfo("The message id after appending pluginData2 is::"+messageId);
return new ExecutionDescriptor(OK_MAJOR, OK_MINOR, null, null, "
taskSaveAndUpdate executed successfully");
case UNDO_AND_CHECK:
return new ExecutionDescriptor(OK_MAJOR, OK_MINOR, null, null, "No undo for
taskSaveAndUpdate ");
default:
throw new PluginException ("Operation not supported");
 }
 }
```

Reading Data from the Database

The plug-in framework allows you to retrieve data written into the database by the Workflow Manager.

The plug-in can invoke the following method on the `PARContext` to read the data from the database. The data is read from the DATABASE_MESSAGE table:

```
byte[] readData(String messageUrl, long dataPosition, int dataLength)
```

The message id is specified using the syntax `db:<message id>`. The data is returned as `byte[]`.

Optionally, the plugin can retrieve partial data by specifying the *`dataPosition`* and *`dataLength`* parameters. If both the data length and position are zero, then the complete data is read.

The following exceptions must be handled:

PluginException thrown if incorrect message url is specified, or an exception occurs when storing the data.

InterruptedException thrown in case of a database connectivity error.

The below example shows how complete data can be retrieved in an atomic task:

```
public ExecutionDescriptor task_TestReadData_completeData (int op, String
messageId) throws PluginException, InterruptedException

{
// Check which operation the atomic task has to perform
switch (op) {
case DO_AND_CHECK:
//the message id is in the format “db:<message id>”
Context.logInfo("The messageId is ::"+messageId);
byte[] data=context.readData(messageId, 0, 0);
try{
ObjectInputStream objectInputStream = new ObjectInputStream(new
ByteArrayInputStream(data));
Object mwfmData = objectInputStream.readObject();
context.logInfo("Activation successful, the data retrieved is ::"+mwfmData);
}catch(Exception e){
```

```
return new ExecutionDescriptor(ERROR_MAJOR, ERROR_MINOR, null, null,
"completeData not executed");
}
return new ExecutionDescriptor(OK_MAJOR, OK_MINOR, null, null, "
completeData       executed successfully");
case UNDO_AND_CHECK:
return new ExecutionDescriptor(OK_MAJOR, OK_MINOR, null, null, "No undo for
completeData ");
default:
throw new PluginException ("Operation not supported");
}
}
```

The below example shows how partial data can be retrieved:

```
public ExecutionDescriptor task_TestReadData_partialData (int op, String
messageId) throws PluginException, InterruptedException
{
// Check which operation the atomic task has to perform
switch (op) {
case DO_AND_CHECK:
context.logInfo("The messageId is ::"+messageId);
//the data length is 30 since “write this first string”.getBytes() is 30
byte[] data=context.readData(messageId, 1, 30);
try{
ObjectInputStream objectInputStream = new ObjectInputStream(new
ByteArrayInputStream(data));
Object mwfmData = objectInputStream.readObject();
context.logInfo("Activation successful, the first string is ::"+mwfmData);
//read the second string, data is read from 31st position
data=context.readData(messageId, data.length+1, 31);
objectInputStream = new ObjectInputStream(new ByteArrayInputStream(data));
mwfmData = objectInputStream.readObject();
context.logInfo("Activation successful, the second string is ::"+mwfmData);
}catch(Exception e){
return new ExecutionDescriptor(ERROR_MAJOR, ERROR_MINOR, null, null, "
partialData not executed");
}
return new ExecutionDescriptor(OK_MAJOR, OK_MINOR, null, null, " partialData
executed successfully");
case UNDO_AND_CHECK:
return new ExecutionDescriptor(OK_MAJOR, OK_MINOR, null, null, "No undo for
artialData ");
default:
throw new PluginException ("Operation not supported");
}
}
```

Understanding the Plug-in Deployment Descriptor (Manifest)

The `MANIFEST` directory contains the file `par.xml`, which is also known as the deployment descriptor. The `par.xml` file declares all of the components in the plug-in. These components include the plug-in Java class, the atomic tasks defined in the Java class, configuration parameters, scripts, files, preprovisioned scripts and files, and libraries.

Further, `par.xml` contains the definition of each configuration parameter in the plug-in. The configuration parameters can be accessed from the context object in the plug-in Java class, or through the environment in scripts. From the environment, the configuration parameter name begins with `ACTIVATOR_`. See the Perl example above for an example of gaining access to the configuration parameters from a script.

```
<Configuration>
    <Param name="PASSWORD_FILE" value="/etc/passwd" />
</Configuration>
```

1. Atomic tasks

   Each atomic task defined in the Java class must be declared in `par.xml`.

```
<AtomicTask>
    <Task exported="true" execution="ON_LINE">
        <Name>Example_userExists</Name>
        <Argument>machine</Argument>
        <Argument>username</Argument>
    </Task>
</AtomicTask>
```

2. Scripts

   Each script in the plug-in must be declared in `par.xml`. The attribute `interpreter` specifies the command that the plug-in uses to run the script. The interpreter must be in the path on the remote machine. The `Script` name is the same name used to identify the script in the method `executeScript` of the plug-in context.

```
<Scripts>
    <Script name="do_Example_userExists_script" file="userExists.pl"
            interpreter="perl">
        <Description>Perl script to check if user exists.</Description>
    </Script>
</Scripts>
```

Packaging and Deploying a Plug-in

You can use Service Builder from the command-line to pack a plug-in into a Plug-in Archive (PAR), as well as to deploy the plug-in.

1. Verify that your plug-in directory structure matches the one shown in "Understanding Plug-in Archives" on page 24.
2. To pack a plug-in into a PAR, run:

   servicebuilder -packPAR <*plug-in directory*> <*PAR name*>

3. To deploy the plug-in to the server, run:

   servicebuilder -deployPAR <*PAR name*>

   Your plug-in is now available on the server so that you can test it or invoke it from a workflow Activate node.

The Difference Between PAR Deployment and Script Deployment

It is important to understand the distinction between PAR deployment and script deployment. PAR deployment occurs when you use Service Builder to deploy a PAR. This deployment stores the PAR on the Service Activator server machine, making it accessible

by the Resource Manager. The only machines affected by PAR deployment are the Service Activator server machine and the machine hosting the Oracle database used by Service Activator.

Script deployment is very different from PAR deployment. Script deployment occurs when you invoke the `context.executeScript()` method of an atomic task, and the Resource Manager deploys all the scripts for that plug-in to the target machine.

The Resource Manager maintains a cache that records whether script deployment for a given plug-in has already occurred on a particular target machine. This allows the Resource Manager to make an optimization and avoid copying all the plug-in scripts to a target each time an atomic task is invoked. This cache is located in the `$ACTIVATOR_VAR/resmgr/cache` directory.

Creating Compound Tasks Manually: Advanced Tips

A compound task is represented by an XML file that defines the compound task, its called tasks, and its parameters.

There are two main advantages to editing the compound task XML file manually, rather than through the GUI. First, it is easier to reorder the tasks. Second, you have control over the order of the parameters in the compound task.

This is a case where you may find a hybrid approach easier than exclusively using the GUI or manually creating the XML. First, create the compound task in the GUI, by dragging and dropping the required atomic tasks. Then, if you need some fine control over the order of the parameters or the order in which the atomics are called, follow these steps:

1. Export the compound task to a file.
2. Edit the file manually.
3. Remove the compound task from the project.
4. Import the compound task from the file.

Example 3-1 **Compound Task Example**

Look at this example of a simple compound task that calls two atomic tasks.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- service builder 3.5 -->

<CompoundTask>
    <Name>AddUserDir</Name>
    <NameSpace type="GLOBAL"/>
    <CalledTask>
        <TaskName>MyCT</TaskName>
        <NameSpace type="GLOBAL"/>
        <Param name="machine"/>
        <Param name="dir"/>
        <Param name="username"/>
    </CalledTask>

    <CalledTask>
        <TaskName>W2K_addDir</TaskName>
        <NameSpace type="GLOBAL"/>
        <Param name="machine"/>
        <Param name="dir"/>
        <Param name="skeldir" value="&quot;&quot;"/>
    </CalledTask>

    <CalledTask>
        <TaskName>W2K_chgPerm</TaskName>
        <NameSpace type="GLOBAL"/>
        <Param name="machine"/>
        <Param name="username"/>
        <Param compoundTaskName="dir" name="path"/>
        <Param name="access" value="RWC"/>
    </CalledTask>
</CompoundTask>
```

First notice the list of three parameters for the compound task. The order they appear in the XML is the order they must be passed to the compound when it is called. You can easily change this order in the XML. A similar change cannot be made from the Service Builder GUI.

Next notice the list of called tasks. The order they appear in the XML is the order in which they will be invoked when the compound is called.

Finally, notice the parameters to the called tasks. The order of the parameters within a called task is important, and it depends on the actual order that the atomic task expects them. The use of the `compoundTaskName` and value attributes is also important. These are used to determine what parameter values from the compound task to pass to the called task parameters. When a called task is invoked, the system first checks whether the called task parameter has a value specified, in which case this constant value is passed. If no constant value is specified, then the system next uses the `compoundTaskName` attribute; if there is a matching parameter name in the compound task parameter list, then the value of that compound task parameter is passed to the called task. Finally, the system uses the name attribute; if there is a matching parameter name in the compound task parameter list, then the value of that compound task parameter is passed to the called task. If no matching compound task parameter is found, an error will be thrown.

A Generic CLI

This appendix describes the generic CLI plug-in.

Generic CLI Plug-in

Table A-1 Generic CLI plug-in

<table>
<tr><td>Description</td><td colspan="2">Activation commands are obtained from a specified XML formatted file. The XML syntax provides the description for a series of commands that are associated with a Connection sequence. These are the commands used to establish the initial state of the device (such as logging in). Should one of the commands fail, the connection is closed.

Once the connection has been made, the Activate sequence commands are executed. Each of these commands can have an associated UNDO_AND_CHECK command. Should a DO_AND_CHECK command fail, all UNDO_AND_CHECK commands up to this point are then executed (in reverse order) to place the system back into a consistent state.

Next, the Disconnect sequence of commands is specified. These commands are always executed regardless of the successful completion or activate sequences in order to close the connection completely.

The last series of commands in the file is the Rollback sequence. These commands define the commands to be executed should an UNDO_AND_CHECK operation be requested. You typically would want to invoke the activate/undo commands in reverse order (via the Rewind element), but additional or an entirely different set of commands may be specified.</td></tr>
<tr><td>Plug-in archive</td><td colspan="2"><code>GenericCLI.par</code></td></tr>
<tr><td>Platform</td><td colspan="2">Any Telnet-capable or Secure Shell-capable system or device. It is also possible to adopt custom application for establishing connection using any other protocol. All protocols must be able to communicate using command line interface (CLI).</td></tr>
<tr><td>Atomic Tasks</td><td>GenericCLI_activate</td><td>Execute CLI activation commands from a single XML file or from a connect and activation sequence file on the target host:

•Establish a connection to the host by using either telnet ssh or other protocol of your choice (defined within the XML connection sequence), and execute the Connect sequence in order. Should a command fail, skip all subsequent commands.

•Upon successfully completing the entire Connect sequence, execute the Activate sequence. Should a command fail, run all UNDO_AND_CHECK commands associated with previously run commands (in reverse order) to perform any required error recovery.

•Finally, run the Disconnect sequence, and terminate the connection.

•If an UNDO_AND_CHECK operation is requested, execute the Rollback sequence instead of the Activate sequence.</td></tr>
</table>

Table A-1 Generic CLI plug-in (Continued)

	GenericCLI_ActivatePool	Same as the atomic task GenericCLI_Activate. The only difference is that an extra parameter - poolName - which explicit define the connection pool which should be used when executing the atomic task. Only the Activation sequence in the XML document is used as the configuration of the connection establishment and later disconnect is configured though the pool configuration (done from the operator ui).
See also	For more information about this plug-in, see the Javadoc associated with the plug-in code. You can find more information about this plug-in in the *$ACTIVATOR*/docs/plugins directory. Windows users can access this information from the desktop by navigating to `Start-> All Programs-> HP OpenView-> Service Activator-> Docs-> Plug-ins Documentation`	

B NNM Liaison

This appendix describes the NNMLiaison plug-in.

NNMLiaison Plug-in

Table B-1 NNM Liaison Plug-in

Description	Plug-in for integration with HP Network Node Manager. The NNMLiaison plug-in is useful for: • Creating Interface Groups in NNMi's GUI	
Plug-in Archive	`NNMLiaison.par`	
Platform	Supported NNMi platforms: • all Supported NNM versions: • HP Network Node Manager version v9	
Atomic Tasks	NNM_createInterfaceGroup	The task creates an NNMi interface group and filter. This allows the NNMi operator to lauch interface group views with interfaces marked with Service Activator custom attributes. The views may also be cross launched from Service Activator inventory view.
See Also	For more information about this plug-in, see the the Javadoc associated with the plug-in code. You can find more information about this plug-in in the `$ACTIVATOR/docs/plugins` directory. Windows users can access this information from the desktop by navigating to `Start-> All Programs-> HP OpenView-> Service Activator-> Docs-> Plug-ins Documentation`	

C Generic LDAP

This appendix describes the GenericLDAP plug-in.

GenericLDAP Plug-in

Table C-1 GenericLDAP Plug-in

Description	The plug-in provides the capability to execute a create, delete, modify, and search operations against an LDAP server. The plug-in has support for full roll-back functionality.
Plug-in Archive	`GenericLDAP.par`
Platform	Any LDAP server which support LDAP v3.0.
Atomic Tasks	The plug-in has one atomic tasks: GenericLDAP_execute. All of the information needed by the plugin is passed in two parameters. The first parameter contains the hostname of the target LDAP server and the second parameter contains the rest of the information, including connection information and data to be updated in the LDAP server's directory(ies). The second parameter comes in the form of either an xml file or an xml string. Within this xml, you may specify one of several different operations on the LDAP server
See Also	For more information about this plug-in then see the Javadoc associated with the plug-in code. You can find more information about this plug-in in the *$ACTIVATOR*`/docs/plugins` directory. Windows users can access this information from the desktop by navigating to `Start-> All Programs-> HP OpenView-> Service Activator-> Docs-> Plug-ins Documentation`

D Generic HTTP Plugin

This appendix describes the Generic HTTP plug-in.

Generic HTTP Plug-in

Table D-1 Generic HTTP Plug-in

Description	The plug-in provides the capability to send HTTP(S) POST/GET request and receive the response. The plug-in also supports the following additional features. The plug-in has support for full roll-back functionality in case of a POST request and the parameter **undo** must be provided to make use of this capability. Results from HTTP(S) GET and POST requests are uploaded back to the workflow manager as key value pair, the key beeing `HttpGet` and `HTTPPost` respectively. The content of returned cookie, if any, is uploaded with the key `CookieValue`.	
Plug-in Archive	`GenericHTTP.par`	
Platform	Any HTTP-capable system or device.	
Atmic Tasks	HTTPGet_URL	Makes a HTTP(S) GET request to the specified target URL.
	HTTPGet_URLAnd Cookie	Makes a HTTP(S) GET request to the specified target URL. The cookie returned from an earlier request can also be sent as a request property
	HTTPGet_URLCook ieAndTimeout	Makes a HTTP(S) GET request to the specified target URL. A cookie can also be sent as a request property. A connection and read timeout can be associated with the http connection.
	HTTPGet_URLCook ieAndProxySettings	Makes a HTTP(S) GET request to the specified target URL. A cookie can also be sent as a request property. The proxy host and port necessary to connect to the target URL can also be specified
	HTTPGet_URLCook ieAndNetworkAuthe ntication	Makes a HTTP(S) GET request to the specified target URL. A cookie can also be sent as a request property. The username and password necessary to make a network authentication in order connect to the target URL can also be specified.
	HTTPGet_URLCook ieAndSecureConnec tion	Makes a HTTP(S) GET request to the specified target URL. A cookie can also be sent as a request property. A keystore containing a valid SSL certificate identifying the server and the necessary key pasword and store password in order make a secure connection to the target URL can also be specified.

Table D-1 Generic HTTP Plug-in (Continued)

	HTTPGet	Makes a HTTP(S) GET request to the specified target URL. It has the following collection of parameters: • **httpUrl** The target URL for the HTTP(S) connection. This is also the locking argument • **username** Username for network connection authentication • **password** Password for network connection authentication • **keystore** The location of the keystore file, necessary for client authentication • **storepass** The password of the keystore file, necessary for client authentication • **keypass** The password of the public certificate/private key pair • **proxyServer** Name of proxy server, if proxy is to be used • **proxyPort** Port of proxy server, if proxy is to be used • **cookie** Cookie of the HTTP(S) request • **connectTimeout** Connection timeout value, in milliseconds • **readTimeout** Read timeout value, in milliseconds • **contentType** The request content type. Default value is "text/xml".
	HTTPPost_URLAnd Request	Makes a HTTP(S) POST request to the specified target URL. The message to be posted in case of normal execution and during rollback should be specified.
	HTTPPost_URLCoo kieAndRequest	Makes a HTTP(S) POST request to the specified target URL. The request and undoRequest to be posted should be specified. The cookie returned from an earlier request can also be sent as a request property.
	HTTPPost_URLCoo kieTimeoutAndRequ est	Makes a HTTP(S) POST request to the specified target URL. The request and undoRequest to be posted along with a cookie can be specified. A connection and read timeout can also be associated with the http connection to the target URL.

Table D-1 Generic HTTP Plug-in (Continued)

	HTTPPost_URLCookieProxySettingsAndRequest	Makes a HTTP(S) POST request to the specified target URL. The request and undoRequest to be posted along with a cookie can be specified. The proxy host and port necessary to connect to the target URL can also be specified.
	HTTPPost_URLCookieNetworkAuthenticationAndRequest	Makes a HTTP(S) POST request to the specified target URL. The request and undoRequest to be posted along with a cookie can be specified. The username and password necessary to make a network authentication in order connect to the target URL can also be specified.
	HTTPPost_URLCookieSecureConnectionAndRequest	Makes a HTTP(S) POST request to the specified target URL. The request and undoRequest to be posted along with a cookie can be specified. A keystore containing a valid SSL certificate identifying the server and the necessary key pasword and store password in order make a secure connection to the target URL can also be specified.
	GenericHTTP_HTTPPost	Makes a HTTP(S) POST request to the specified target URL. Besides all the parameters for the HTTPGet atomic task, two additional parameters are needed. • **request** The message to be sent to the http(s) server • **undoRequest** The request sent to the http(s) server during roll-back Here both parameters can either be the message to be sent or a file URL for a file containing the message. For the second case, the URL must start with file:// The task returns an ExecutionDescriptor with major code OK and minor code NONE upon successful completion. The task fails if the server sends back a response code different than '200', In this case, the value of major code is ERROR and the value of minor code depends on whether the POST request has reached the target system. As from the HTTP plug-in's perspective, there is no way to tell whether the target system state is restored to the original state before the atomic task operation was invoked, the rule is to set the minor code to CONSISTENT if the error happens before the request has reached the target system, and set to INCONSISTENT otherwise
See Also	For more information about this plug-in, see the Javadoc associated with the plug-in code. You can find more information about this plug-in in the `$ACTIVATOR/docs/plugins` directory. Windows users can access this information from the desktop by navigating to `Start-> All Programs-> HP OpenView-> Service Activator-> Docs-> Plug-ins Documentation`	

Symbols

@author tag, PAR documentation, 55
@dependency tag, PAR documentation, 55
@do_and_check tag, PAR documentation, 55
@param tag, PAR documentation, 55
@preprov tag for PAR documentation, 55
@undo_and_check tag, PAR documentation, 55
@warning tag, PAR documentation, 55

A

advanced tips for creating plug-ins, 74
atomic task
  adding using Service Builder, 37
  declaring methods, 75
  definition, 14
  parameters, 15
  setting parameters, 37
  testing, 52
AttributeTable, 28
authorizing deployment, 66

C

classes
  available for plug-ins, 28
  PAR directory, 24
  provided in library, 28
command line parameters, Service Builder, 70
compiling PARs, 49
compound task
  adding services and changing parameters, 57
  constant values, 60
  creating, 56
  deploying, 62
  documenting, 63
  enabling and disabling, 67
  export to XML, 67
  managing using Service Builder, 67
  properties, 61
  removing or changing a called task, 60
  reordering tasks, 60
  testing, 63
configuration parameters, 48
configuring Service Builder, 69
context, execution of a plug-in, 26
conventions
  typographical, 9
created, plug-in state, 22
creating compound tasks using Service Builder, 56
creating plug-ins without service builder, 74

D

declaring
  plug-in class, 75
deployed tasks, maintaining consistency, 65
deploying compound tasks, 62
deploying PARs
  authentication, 66
  authorization, 66
  using Service Builder, 51
deployment authorization, 66
deployment descriptor, see MANIFEST
deployment modes
  NO DEPLOYABLE, 34
  ON-DEMAND, 34
destroy( ), plug-in method, 23
destroyed, plug-in state, 23
DO_AND_CHECK, 15, 75
doc directory, PAR, 24
documenting compound tasks, 63
documenting plug-ins, 53

E

execution context of a plug-in, 26
ExecutionDescriptor, 76, 78
exporting and importing PARs, 36

F

files directory, PAR, 25
files, adding using Service Builder, 43

I

importing and exporting PARs, 36
interpreter, 42
invalid
  compound tasks, 65

J

Java classes
  description, 74
  using Service Builder to add, 40
Javadoc tags for plug-in documentation, 54
Javadocs, accessing for plug-ins, 28

L

lib directory, PAR, 25
library
  adding to PAR, 45
life-cycle of a plug-in, 22
locking
  arguments, 17
  description, 16
  example, 17
locking, description, 17

M

MANIFEST directory, PAR, 25, 82
manifest, plug-in, 82
manually creating plug-ins, 74

N

NO DEPLOYMENT, 34
not initted, plug-in state, 23

O

ON-DEMAND, 34

P

PAR

adding atomic tasks, 37
classes directory, 24
compiling, 49
configuration parameters, 48
deploying using Service Builder, 51
deployment modes, 34
description, 15, 24
directory descriptions, 24
directory structure, 22, 24, 35
doc directory, 24
file layout, 24
files directory, 25
general properties, viewing, 46
generating documentation, 53
importing and exporting, 36
Javadoc tags for documentation, 54
lib directory, 25
MANIFEST directory, 25
scripts directory, 25
src directory, 26
testing using Service Builder, 52
updating, 39
PAR documentation
@author tag, 55
@do_and_check tag, 55
@param tag, 55
@preprov tag, 55
@undo_and_check tag, 55
@warning tag, 55
PAR documentation tag
@dependency, 55
par.xml, 82
parameters
atomic task, 15
PARContext, 26, 28
PARPlugin, 28
plug-in
archives, description, 24
busy, 23
class, declaring, 75
classes defined, 28
classes, description, 28
context, 26
creating in Service Builder, 33
creating manually, 74
definition, 14
deploying using Service Builder, 51
description, 14
destroyed, 23
documentation, 28
documentation tags, 54
enabling and disabling, 67
execution context, 26
generating documentation, 53
Javadocs, accessing, 28
life-cycle, 22
managing PARs using Service Builder, 67
method, declaring, 75
not initted, 23
properties, 46
service_xx, 23
states
busy, 23
created, 22
destroyed, 23
not initted, 23
pooled (busy), 23
pooled (free), 23
testing, 52
using Service Builder to manipulate, 32
valid states, 22
pooled (busy), plug-in state, 23
pooled (free), plug-in state, 23
preprovision scripts and files, including in a PAR, 44
preprovisioning
files, 25
include files in PAR, 44
project, Service Builder
creating, 32
properties, compound task, 61

R

reordering tasks in a compound task, 60
resmgr.xml, 18
Resource Manager
configuration, 18
resource manager
description, 15
resmgr.xml, 18
roles
, 66

S

scripts directory, PAR, 25
scripts, adding using Service Builder, 42
Service Builder
adding atomic tasks, 37
adding files, 43
adding information to the PAR library, 45
adding Java classes, 40
adding scripts, 42
adding services and changing parameters, 57, 59
command line, 70
compiling PARs, 49
configuration, 69
deploying a plug-in, 51
deploying compound tasks, 62
documenting compound tasks, 63
generating documentation for a PAR, 53
importing and exporting PARs, 36
including preprovisioning scripts and files, 44
maintaining consistency of deployed tasks, 65
managing plug-in archives and compound tasks, 67
removing or changing a called service, 60
starting, 32
testing a compound task, 63
testing a plug-in, 52
updating PARs, 39

  using to create compound tasks, 56
  viewing general properties, 46
  viewing properties of a compound task, 61
sharing source files between plug-ins, 39
src directory, PAR, 26
starting Service Builder, 32

T

tasks
  maintaining consistency of deployed, 65
testing atomic tasks, 52
transaction ID, 16

U

UNDO_AND_CHECK, 15, 75
updating a PAR file, 39
using Service Builder to test a compound task, 63

V

valid states of a plug-in, 22
viewing properties of a compound task, 61
ViewTransactionState command, 16

X

XA protocol, 15
XID transaction ID, 16