HP Service Activator

Inventory Subsystem

Edition: V62-1A

for Microsoft Windows® Server 2008 R2, HP-UX 11i v3, Red Hat Enterprise Linux 6.4

Manufacturing Part Number: None

October 15, 2013

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.

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-pd001310

Contents

Contents

Install Location Descriptors

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

Descriptor	What the Descriptor Represents
$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 JEE components. The UNIX location is `/opt/HP/jboss/server/standalone/deployments` The Windows location is `<install drive>:\HP\jboss\server\standalone\deployments`
$JBOSS_EAR_LIB	Location for libraries (Java *.jar files) to be executed by the HPSA engine (workflow manager and resource manager): *$JBOSS_DEPLOY*`/hpsa.ear/lib`
$JBOSS_ACTIVATOR	More specific location of Service Activator UI components deployed in JBoss: *$JBOSS_DEPLOY*`/hpsa.ear/activator.war`

In This Guide

This guide describes Service Activator's inventory subsystem from the perspective of customization.

Audience

The audience for this guide is:

- Systems Integrator, who will use it as a resource for customizing solutions.

1 Introduction

This chapter introduces the Service Activator inventory subsystem by presenting an overview of the subsystem and the related tools.

Overview of Service Activator Inventory

This manual is about the repository of solution data that is customized for a Service Activator solution, also known as the *inventory*. As system integrator you are free to define the data model to be stored in the inventory, depending on the requirements of the solution.

Packaged with the Service Activator core product is a data model that is suitable for a provider's network infrastructure, the Common Network Resource Model. If that model is suitable for your solution, possibly with modifications, you can use it as a starting point. For information about the Common Network Resource Model, see *HP Service Activator, System Integrator's Overview*.

The design process that precedes the definition of the model (understand the reality to be modelled in the inventory and the details that will be needed for activation, analyze and understand the relationships between the different entities that are modelled) is not taught here, as the knowledge needed for the analysis and design is of a general nature; it is not specific to a Service Activator solution. This manual covers the data model definitions that you must prepare, how to present the data model and provide access to it via the operator UI, and the principles for accessing inventory data from your Service Activator workflows.

NOTE

Since historically Service Activator inventory has been used primarily to manage resources in a service provider's network and service infrastructure and indeed has special features for reserving and releasing objects that represent reservable resources, a data object is - in the parlance of this manual - generally called a "resource"; it is defined in a "resource definition file". This does not imply that you are restricted to modelling objects that can reasonably be thought of as resources. In many applications, it is natural to divide the inventory in three portions: *resource inventory* including network resources, *service inventory* possibly including multiple classes of objects to represent services, and *solution configuration* data. In object-oriented terms, think of a resource as an object class. In database terms it is a table.

Service Activator's inventory subsystem stores instances of the defined resources in tables in the database which is used as part of the Service Activator platform (see *HP Service Activator, System Integrator's Overview* for more information about use of the database in general and configuration of data sources, i.e. pools of database connections, in particular). The inventory subsystem makes the data repository accessible from workflows for creation, query, updating and deletion of data instances, and it supports a powerful UI to view resources and perform operations on them; the UI is easily customized so the data is arranged in a way that is natural for the user of the activation solution.

The Service Activator core product includes these parts which are used in the inventory subsystem, some of them as tools during solution customization:

- A preinstalled underlying database is a prerequiste for Service Activator.

- A command line tool called Inventory Builder. This tool takes inventory object definitions written in a special XML syntax as input and generates three types of artifacts:
  - — Java beans that manage instances of your inventory resources and use JDBC to access the database.
  - — SQL table definitions that you can use to create inventory resource tables and indexes in the database.
  - — UI files supporting capabilities to manually view, search, create, update, and delete inventory resources from the inventory UI.

NOTE UI files are generated to fit into the Struts framework. These files comprise Java Server Pages (JSPs), Struts Java classes, Struts config files and property files.

- A workflow manager module called the database module that manages a pool of JDBC connections to the database to allow workflow jobs to access inventory data. Refer to *HP Service Activator, Workflows and the Workflow Manager* for more information about this module.
- A set of nodes in the workflow manager node library that interact with the Java beans generated by Inventory Builder to create, query, update, reserve, release and delete resources in inventory. There are some additional nodes in the library that can execute explicitly provided SQL statements to query and update data in database tables; these nodes do not use the generated Java beans. Refer to *HP Service Activator, Workflows and the Workflow Manager* for more information about these nodes.
- A separately launchable window in the operator UI that presents the inventory resources in hierarchical tree structures viewable from an Internet Explorer Web browser, providing search functions and allowing access to operations defined for each resource. Refer to the chapter “Inventory User Interface” in *HP Service Activator, User’s and Administrator’s Guide* for a description of the capabilities of the inventory UI. You can define several trees, each one for a different part or a different view of the inventory. If you choose to divide your inventory across multiple physical databases, each tree can only access resources that are stored within a single database. The inventory UI window must be configured with definitions of one or more presentation trees. See chapter 4 for full information about presentation trees.
- A tool called Inventory Tree Designer. This tool can be used in graphical interactive mode to define and edit inventory presentation trees and to manage their deployment. It can also perform tree deployment management in command-line mode Tree definitions must be deployed into Service Activator’s static repository to become active and control the behaviour of the inventory UI.

Inventory Solution Development Process

The list below gives an overview of the activities you will undertake to develop and deploy a Service Activator inventory solution. Not all the activities are needed for all solutions. Details for most of these activities are described in the following chapters.

- Analyze the desired solution to understand what resources will be stored in your inventory. This should include a list of all of the resources and their attributes. You might produce a document or a diagram in any convenient notation such as UML. From the analysis you can determine the degree to which the Common Network Resource Model will be applicable.
- Write resource definition files for the resources in your solution. There will be one file for each type of resource. The file encodes information gathered during your preceeding analysis. An XML syntax is defined for this purpose. The vocabulary and associated features for managing resources are described in detail in chapter 2.
- Process your resource definition files with the Inventory Builder tool supplied with Service Activator. Then deploy the output produced: compile the Java beans and move the generated

UI files to the proper location for use by the JEE engine. See chapter 3 for information about the Inventory Builder.

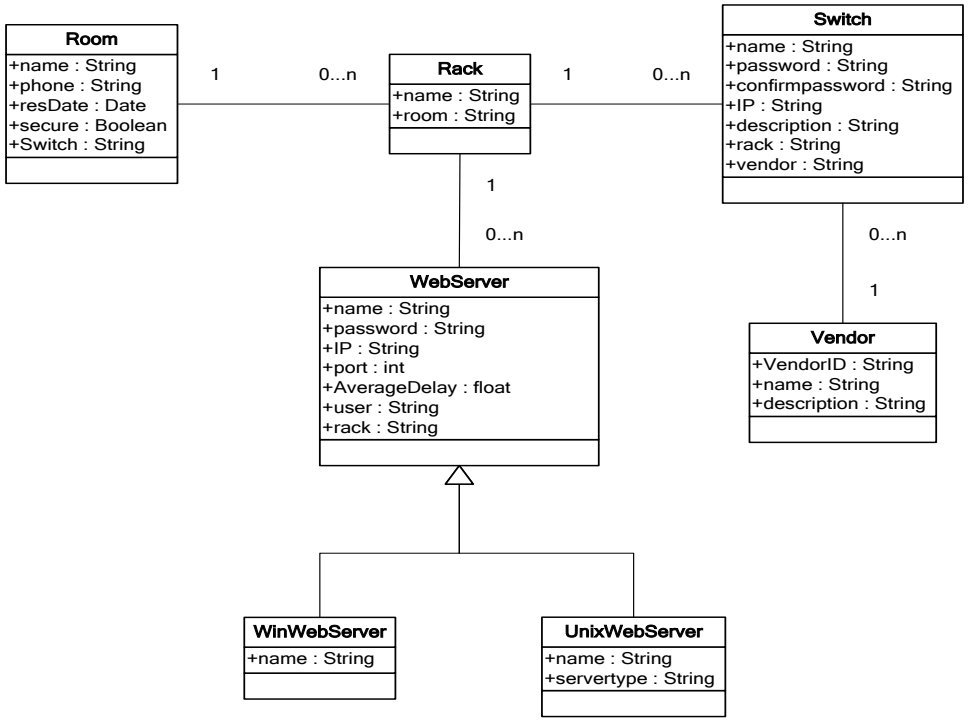
- Prepare one or more inventory tree definitions to specify the presentation of your inventory data. There is an XML syntax defined for this purpose and a graphical tool, the Inventory Tree Designer, to facilitate editing of the definition. With this tool you can also deploy your tree(s) into the static repository for testing. See chapter 4 for full information about presentation trees and how to define them.
- Design the physical implementation of your inventory. The default implementation is to place all tables in a single tablespace belonging to the same database user in the same database instance as the predefined tables in Service Activator's repositories (see description at the beginning of this chapter). If you want to use multiple tablespaces and/or multiple database instances, you must create them explicitly. The tables to hold the different resources you have defined are created by running the SQL command files generated by the Inventory Builder. If you use multiple databases you must configure the workflow manager with a database module to connect to each of them and prepare your worfklows to access each resource through the proper database module. Refer to *HP Service Activator, Workflows and the Workflow Manager* for information about configuring the workflow manager.
- Populate your inventory with appropriate resource instances to model the provider's network and service infrastructure and anything else you are modelling. For modest numbers of resource instances you can create them manually from the inventory UI page. Alternatively, you may use SQL tools to populate the database directly.

Example Data Model Used in this Manual (DocEx)

The following chapters use a small but meaningful data model to demonstrate the capabilities of the Service Activator Inventory subsystem. The example relates to the hardware resources that would be deployed by a provider offering web services. When Service Activator has been installed you can find the files for this example in *$ACTIVATOR_OPT*/examples/doc_example/DocEx.zip. You can use the Deployment Manager to install the example as a solution named DocEx. In the resource definition files you can find examples of most of the concepts described in chapter 2, and similarly the tree definition uses most of the features described in chapter 4.

Here is a UML class diagram showing the complete data model for the example.

Figure 1-1 Example Data Model



2 Resource Definitions

This chapter explains how to write the resource definition files to describe the resources to be managed in your inventory system.

A resource definition is an XML document. In overview its main parts are:

- A header with optional attributes, defining general properties of the resource
- Field specifications, defining the data members of the JavaBean which correspond to columns of the database table
- Key specifications, indicating the primary key and other search keys
- Operation declarations, indicating what operations may be performed on the resource

There is not a special graphical tool for creating resource definitions. Use your preferred text editor to prepare resource definition files.

The section "*XML Vocabulary Quick Reference*" provides a quick reference to the contents of a resource definition. It is followed by a longer section containing thorough explanations of the individual attributes and elements with examples.

The XML syntax for a resource definition is specified in file bean.dtd that is found in *$ACTIVATOR_THIRD_PARTY*/`inventory`. As you can see in the DocEx files, resource definition files must begin with a header including a <!DOCTYPE> element to reference bean.dtd. The syntax is also easily understood from Table 2-1 below.

Localization

In a resource definition you will define several names - for the resource, for its fields, etc. These names will be used for the Java beans that are generated by the Inventory Builder based on the resource definition, and must contain only ASCII characters (with further restrictions). For a localized solution using a language with a different character set, they may not be suitable for displaying to the user. You can then override them for presentation use. The general rule is that a resource definition must contain only ASCII characters, except for the following elements and attributes, which are the ones you need to localize; they can be specified in local language: <Beanlabel>, <Label>, <Description>, <Column>, <Default>, <DBTable>, <HistDBTable>, <Value>, show, message.

XML Vocabulary Quick Reference

Table 2-1 describes the XML vocabulary for resource definitions. For each element that can occur in a resource definition, starting with the root tag <Bean>, all its attributes and tagged child elements are listed and briefly explained in one row each, attributes before tagged child elements, tags enclosed in angle brackets <..>. When reading the descriptions, beware that many features pertain to the inventory UI. More thorough descriptions are given for most elements and attributes in the next section, "*Resource Definition Elements*". Explanation pertaining to generated SQL statements are gathered in the section "*Generated SQL*" at the end of this chapter.

Table 2-1 Resource Definition Quick Reference

Parent tag	Attribute name/ child element tag	Mandatory/ default value	Description
<Bean>	audit	false	activates generation of audit records
	maxCount		declares resource reservable with automatic resource counter
	hideCount	false	makes resource counter hidden on View and Update
	inheritsFrom		defines inheritance relationship
	inheritsSolution	*Solution*	solution to which the (parent) superclass resource belongs
	extAttributes		declares user may add fields to the resource at run-time
	history		declares history table shall exist for the resource
	<Name>	M	name of resource (bean name)
	<Solution>		name of solution to which resource belongs
	<Beanlabel>	(bean) *Name*	presentation name for resource
	<ConstraintName>	(bean) *Name*	name to use for bean when generating names of constraints on database tables; these names combine bean and field names
	<SequenceName>		name of database sequence to be used for the bean
	<Package>	M	package for generated Java bean
	<DBTable>	(bean) *Name*	database table name for the resource
	<DBAlias>	*Solution#Name*	overrides generated alias name for the resource table
	<HistDBTable>	Hist*DBTable*	history table name for the resource
	<Fields>	M	encloses field declarations, see entries below
	<DisplaySequence>		encloses definition of sequence in which fields will appear in forms on UI
	<ParentFields>		encloses <ParaentField> element
	<JoinBridges>		encloses global definitions of join bridges
	<Keys>	M	encloses key declarations, see entries below
	<Operations>		encloses operation declarations, see entries below

Parent tag	Attribute name/ child element tag	Mandatory/ default value	Description
	<Construction>		Java code that is added to the generated constructor for the bean class
	<Validation>		Java code that is added to the validate method in the bean, used to validate field values when a resource instance is created or updated
	<DBInitCustomCode>		Java code appended to field initialization code of methods to instantiate the bean.
	<GuiStorage>		Java code inserted in Struts action beans for store, update and delete forms.
	<GuiUpdate>		
	<GuiDeletion>		
	<FormValidations>		encloses form validation declarations, see entries below
<DBTable> <HistDBTable>	tablespace		specifies database tablespace where resource table is stored
	index_tablespace		specifies database tablespace where indexes for the resource are stored
<Fields>	<Field>		repeatable, defines one field, has attributes and inner tags
<Field>	mandatory	true	makes field value mandatory on Create and Update
	hiddenView	false	makes field hidden on View
	hiddenUpdate	false	makes field hidden on Update
	update	true	makes field editable on Update
	hiddenCreate	false	makes field hidden on Create
	create	true	makes field editable on Create
	sequence	false	generates field value from database sequence
	sequenceStart	1	starting value for database sequence
	searchable	true	makes field searchable
	maxCount	false	declares resource reservable with field as resource counter
	dateFormat		specifies format of date field
	integerFormat	true	for fields of type integer or long: apply integer formatting according to locale (for some locales thousands separator will be displayed)
	password	false	displays field as password

Parent tag	Attribute name/ child element tag	Mandatory/ default value	Description
	encrypt	false	password is stored in encrypted form in database
	store	true	creates a column to store field value in database
	<Name>	M	name of field
	<Column>	(field) *Name*	name of database column for the field
	<Label>	(field) *Name*	presentation name for field
	<Type>	M	type of field, one of: String, int, long, Clob, Blob, long, boolean, double, float, Date
	<Default>		default value for field
	<Description>		description for field, displayed on UI
	<ListOfValues>		defines selectable values for the field on Create/Update, either explicitly (list of <Value> elements), or to be found from associated foreign bean (<BeanName>, <Label>, <Method>, <Param>)
	<Loader>		for field that is not stored in database (attribute store=”false”), defines Java code to calculate value to display
	<ShowConditions>		encloses and combines one or more <ShowCondition> elements
<ShowConditions>	<ShowCondition>		repeatable, defines a condition for the field to be visible
	operator	and	defines operator to combine several boolean condition values (‘and’ or ‘or’)
<ShowCondition>	empty	false	if true, an empty condition, otherwise a pattern match condition
<ShowCondition>	<FieldName>	M	specifies the field the condition depends on
	<Pattern>		must be present unless empty is “true”, specifies the pattern that must match the value of the field (<FieldName>)
<ListOfValues>	withoutForeignKey	false	true: to use <BeanName> on field which is not foreign key
	<Value>		explicitly defines one possible value for the field, repeatable, see attributes below
	<BeanName>		fully qualified name of foreign bean (or just bean)

Parent tag	**Attribute name/ child element tag**	**Mandatory/ default value**	**Description**
	<Label>	primary key	(foreign) bean field to show in drop-down list
	<Method>	findAll	find method to select candidates
	<Param>		parameter for find method, repeatable, order is significant
<Value>	show		specifies string to show in drop-down list, allowing it to be different from the element value that is assigned to the field and stored in the database
	selected	false	identifies preselected value, only one value can be selected
<DisplaySequence>	<FieldName>		name of field in display sequence
<ParentFields>	<ParentField>		contains definitions to override properties of an inherited superclass
<ParentField>	hiddenView hiddenUpdate update hiddenCreate create searchable dateFormat integerFormat		overrides value of superclass (parent) attribute for the inheriting (child) class
<ParentField>	<Name> <Label> <Description> <ListOfValues> <Loader>		overrides value of superclass (parent) element for the inheriting (child) class
<JoinBridges>	<JoinBridge>		repeatable, defines a bridge two beans
<JoinBridge>	name		identifies a globally defined JoinBridge, either in the definition or where it is used
	origin		name of origin bean of bridge
	originSolution	*Solution*	solution to which the origin bean belongs
	destination		name of destination bean of bridge
	destinationSolution	*Solution*	solution to which the destination bean belongs
	jumpField		resolves ambiguity when origin bean has multiple foreign keys to first jump (or destination)
	inverseJumpField		resolves ambiguity when destination bean has multiple foreign keys to previous jump (or origin)

Parent tag	Attribute name/ child element tag	Mandatory/ default value	Description
	<Jump>		repeatable, identifies a bean as jump pillar of the bridge
<Jump>	jumpSolution	*Solution*	solution to which the jump bean belongs
	jumpField		resolves ambiguity when jump bean has multiple foreign keys to next jump or destination of the bridge
	inverseJumpField		resolves ambiguity when jump bean has multiple foreign keys to previous jump or origin of the bridge
<Keys>	<Key>		repeatable, defines one key, has attributes and inner tags
<Key>	pk	false	identifies primary key
	foreignBean		declares the key as foreign key and identifies the bean it will reference
	foreignSolution	*Solution*	solution to which the foreign bean belongs
	includeBean		requests a special bean method related to a foreign key
	inverseIncludeBean		requests a special bean method related to a foreign key
	unique	false	requests database to enforce uniqueness of the field value over all resources of same type (the column)
	distinct	false	requests database to enforce uniqueness of the field value over all resources of same type (the column)
	advancedSearch	false	requests generation of a special bean method needed for advanced search
	makeIndex	true	indicates whether a database index shall be created for the column holding the field
	restrict	false	for foreign key: indicates whether to restrict (prevent) deletion of referenced resource instance
	nullOnDelete	false	indicates whether to foreign key to null when referenced resource instance is deleted
	uniqueResults	false	true: findBy method must return unique results
	distinct	false	true: findBy method must return distinct results

Parent tag	Attribute name/ child element tag	Mandatory/ default value	Description
	joinedBy	AND	specifies how <KeyField> values are combined in findBy method, values: AND, OR, MAP
	<Name>	composed from names of key fields	defines a name for the key
	<KeyField>		associates a field with the key, multiple fields can be combined to form a key
	<JoinBridge>		join bridge used in <Key> may reference global definition or may be defined where it is used
	<WhereMap>		when joinedBy=”MAP”: explicit where clause map
<KeyField>	foreignField		for foreign key only: identifies the foreign field, i.e. the field on the foreign bean that this field matches
	externalBean		name of external resource which holds key for findBy method that will join the tables
	externalSolution	*Solution*	solution to which the external bean belongs
	alias		alternative name of externalBean table to be used in generated SQL
	joinField		for use when target resource has multiple foreign keys to external resource to select the one to use for join
	externalJoinField		for use when external resource has multiple foreign keys to target resource to select the one to use for join
	ignoreCase	false	ignore upper/lower case of letters in findBy parameter generated from the key field
	comparator	=	SQL operator used to compare the value of the key field
	compareTo	findBy argument	operand (in SQL) compared to key field
<Operations>	<Store>		empty element indicating Create operation shall be possible
	<Update>		empty element indicating Edit operation shall be possible
	<Remove>		empty element indicating Delete operation shall be possible

Parent tag	Attribute name/ child element tag	Mandatory/ default value	Description
	<CreatePartial>		empty element indicating CreatePartial operation shall be possible
	<DeletePartial>		empty element indicating DeletePartial operation shall be possible
	<FindBy>		repeatable element specifying how results from a findBy method shall be ordered
<FindBy>	key		*Keyname* identifying a findBy*Keyname* method; omitted for findAll
	<OrderField>		name of field by whose value results of the method will be ordered
<OrderField>	desc	false	specifies sorting by descending value of the field
	convertTo		convert order field to number, string or date
	format		used with date format
<FormValidations>	<FormValidation>		repeatable, contains Java code for one validation action that will be appended to the validate method of the generated form bean
<FormValidation>	entry		defines the name of a message property to use in the Java code
	message		must be present together with entry: specifies the actual message string referenced by the property

Resource Definition Elements

This section contains the detailed explanations, some of them with examples, of the elements of the resource definition and the features of the inventory subsystem which they are used to control. There are four subsections, one for each of the main parts of the resource definition:

- the outermost element <Bean>, its attributes and immediate child elements, except <Fields>, <Keys> and <Operations>, which are described in the other subsections;
- the <Field> element, its attributes and child elements, which is used to define a field of the resource; in object-oriented terms the field equates to an object attribute, and in database terms it equates to a column of the table that stores the resource;
- the <Key> element, its attributes and child elements, which is used to indicate which fields of a resource are searchable with database indexes and to establish foreign key relationships to other resources, modelling entity relationships;
- the <Operations> element and its child elements, which are used to define methods of the generated Java bean and operations that can be used in the presentation tree.

Each of these four sections have a subsection for each attribute or child element that is described. The headings of the subsections comprise the tag of the parent element combined with the name of the attribute or the tag of the child element.

The <Bean> Element

The attributes and child elements of the <Bean> element define properties of the resource as a whole.

<Bean> audit

Service Activator can be configured to build an audit trail in its audit repository with records of worfklow actions. The audit trail can be inspected in the Work Area/Audit Messages frame in the main Service Activator UI window. Records of inventory operations can also be included in the audit trail. Recorded information includes the time and date that the action took place, the user who performed it, the operation, the bean and the primary key. Recorded operations include create, update and delete.

Writing of audit records must be enabled per resource bean. Use the “audit” attribute of the <Bean> element to enable auditing.

```
<Bean audit="true">
```

<Bean> maxCount - Reservable Resource, <Bean> hideCount

Service Activator inventory provides functions that support the management of reservable resources. There are two ways to declare a resource to be reservable. One way is to set the maxCount attribute on the bean field, as described here. The other way is to set the maxCount attribute on a field of the resource, as described in the subsection “<Field> maxCount”. In both cases there will be a field to hold a value indicating - at any time - how many units are available for reservation on the resource. When its value reaches 0, the resource is fully occupied.

With the maxCount attribute on a Bean element you must provide a value specifying the maximum number of simultaneous reservations that shall be possible; a field named count__ will automatically be generated as the “availability counter” for the resource and initialized to the specified value whenever a new instance of the resource is created. The count__ field will only be settable by the methods to reserve and release resources.

For example, to specify that a resource you are defining can be reserved for up to 5 simultaneous users, declare your bean as follows:

```
<Bean maxCount="5">
```

The count__ field is hidden from the View and Edit forms when the hideCount attribute is “true”.

For more information on reservable resources, refer to the section “Reservable Resources” at the end of this chapter.

<Bean> inheritsFrom, <Bean> inheritsSolution, <Bean> <ParentFields>

A resource may inherit the properties of another resource. Inheritance is a concept from object orientation. In Java a subclass can inherit the properties of a superclass. With respect to the generated Java bean, inheritance in Service Activator inventory is similar to Java inheritance. Service Activator only supports single inheritance.

In the database separate tables are created for the super- and subclass resources. Generated Java code and SQL statements will join the two tables to access subclass resources. The primary key field(s) of the superclass resource are duplicated in the table of the inheriting resource to enable the joining. It is not possible to redefine or extend the primary key in the definition of the inheriting resource.

Use the inheritsFrom attribute of the <Bean> tag to specify the superclass bean when you are defining a resource that will use inheritance. For example, to define a resource that must inherit from the WebServer bean, declare the bean as follows:

```
<Bean inheritsFrom="WebServer">
```

NOTE

No field added to an inheriting bean can have the same name as any field of the superclass bean. Setting the extAttributes attribute on the superclass will not enable the user to add fields to the inheriting resource. It is recommended not to set extAttributes on a superclass resource.

If the superclass belongs to a different solution, specify that solution with the inheritsSolution attribute.

In the definition of an inheriting (subclass) resource it is possible to modify a number of the properties of fields of the supercless. You can do this with the <ParentFields> element, where for each field you can include a <ParentField> element to override a number of the attributes and elements specified with the corresponding <Field> elements in the resource definition of the superclass. The attributes and child elements of <ParentField> have the same names as those they override.

<Bean> extAttributes

The feature controlled by this attribute allows the user to extend a resource with additional fields. Additional fields can be added while the solution is in use from the inventory UI (see “Operations in Class View” in chapter 4). The primary table that is created based on the resource definition, with a column per field, is not modified by this process. Additional fields are placed in a secondary extension table with the same name as the primary table followed by the suffix ‘Ext’. The extension table has the same primary key as the primary table. For each field that is added, a column named like the field is added to the extension table to hold the field value for each resource instance. A record of the relationship between the field and the resource is written as a row in the table EXTENDED_ATTRIBUTES_CATALOG, which controls the processing of added fields.

NOTE

If your inventory uses a database different from the one specified for Service Activator’s repositories at installation time, make sure the EXTENDED_ATTRIBUTES_CATALOG table is created in the database that stores the resource you are extending. You can use the SQL script file named createInvExtAttrDB.sql found in *$ACTIVATOR_ETC*\sql.

From the description above you will appreciate that the processing of user added fields adds some overhead compared to fields specified in the resource definition.

To allow user defined fields to be added to a resource bean, declare the bean as follows:

```
<Bean extAttributes="class">
```

<Bean> history

Setting the history attribute ”true” causes SQL statements to be generated that will create a table intended to hold historical records moved from the main table that stores the resource, i.e. with identical columns. By default the history table is named by prefixing the name of the main table with ‘Hist’; it can be overriden by a name specified with the <HistDBTable> element. Methods will be included in the Java bean that can move records between the two tables (moveToHistory and recoverFromHistory).

NOTE If you use inheritance, superclass and subclass resources must be defined with the same value of the history attribute.

<Bean> <Name>

The value of the <Name> element specifies the name of the resource. It is used to name the generated Java bean and, unless you override it using a <DBTable> element, also the database table that will hold the resource instances.

The bean name must only contain alphanumeric characters and underscores, and it must begin with a letter.

NOTE The bean name cannot be a Java keyword or an HPSA reserved keyword. Unless overridden with <DBTable> it also cannot be an SQL reserved word. See “Resource Definition Validation” in “Using Inventory Builder” for more detailed information about the validation.

<Bean> <Solution>

The value of the <Solution> element optionally specifies the name of the solution that the resource belongs to. It must be the same in all resource definitions that belong together and are processed together by the Inventory Builder.

NOTE The solution name becomes part of the complete class name for generated Java classes for Struts, therefore should consist of alphanumeric and underscore characters only.

<Bean> <Package>

You must place the resource Java bean in a Java package specified as the value of the <Package> element.

It is recommend to begin the package path with “com.hp.activator” and then add your own hierarchy, starting with the solution name.

If you specify:

```
<Bean>
   <Name>mybean</Name>
   <Solution>MySolution</Solution>
   <Package>com.hp.activator.mysolution</Package>
</Bean>
```

the fully qualified Java class name of the bean will be “com.hp.activator.mysolution.mybean”.

<Bean> <DBTable>

By default the bean name will also be used for the database table which will hold the resource data. You may specify a different table name by using the <DBTable> tag.

In order to ensure that database tables for different solutions have unique names (and that all tables belonging to the same solution appear contiguously in alphabetically sorted lists), it is recommended to use the solution name (possibly abbreviated) as a prefix for the database table name.

NOTE	The table name must adhere to SQL naming restrictions. It can be neither a SQL keyword nor an HPSA reserved keyword. Table names are not case sensitive, although Java classes are.

<Bean> <Fields>

The <Fields> child element of the <Bean> element encloses the definitions of the fields of the resource, each one given as a <Field> element, as described in the section "The <Field> Element" below.

<Bean> <DisplaySequence>

By default the fields of a resource are displayed in generated forms in the order of definition, with fields of a subclass appearing after fields of its superclass. To change the order, typically if you want to move some subclass fields up, you can specify the complete sequence of fields for display purposes with the <DisplaySequence> element, which will contain a sequence of <FieldName> child elements, each one with the name of a field as its value.

<Bean> <Keys>

The <Keys> child element of the <Bean> element encloses the definitions of the keys of the resource, each one given as a <Key> element, as described in the section "The <Key> Element and findBy Methods" below.

<Bean> <Operations>

See the section "The <Operations> Element" below.

<Bean> <Construction>

As all Java classes, the bean for the resource has a constructor which is invoked when the bean is instantiated at run-time, for example as a case packet variable in a workflow job in response to a query. With the <Construction> element you can specify code to be added to the constructor. To use this feature, it is recommended that you first prepare the resource definition without the <Construction> element and process it with the Inventory Builder. Then study the constructor in the generated java source file and write your additional code so it will work in the context, edit it into the <Construction> element, and rerun Inventory Builder.

<Bean> <DBInitCustomCode>

With this element you can include some Java code to be executed in methods which retrieve data from the database to instantiate the bean (i.e. findBy methods, see "The <Key> Element and findBy Methods"). Your code will be included after the constructor and all other actions to initialize fields of the bean, including code specified with the <Field> <Loader> element.

<Bean> <GuiStorage>, <Bean> <GuiUpdate>, <Bean> <GuiDeletion>

Use these elements if you want to add some code to the Struts action beans that are generated for the Create, Update and Delete forms for the bean without having to find, edit and maintain the action bean code. Your code is inserted just before the call of the main method (store, update or delete). You will have available three parameters `con`, `bean` and `formBean`. The first one is a database connection which you must use if you need to make database access. The other two are the resource bean and the Struts form bean.

<Bean> <Validation>

The Inventory Builder automatically generates code that validates whether all of the mandatory fields contain a value. This validation code is run within the JavaBean, prior to a completing a store or update method. It is applied regardless of whether the method is called from the UI or from a workflow.

You can write Java code that will be added to the validation method to perform some specific validation of the properties of your resource instances, for example reject numbers outside of a meaningful range.

Use the <Validation> element to provide your validation code. It is recommended to follow a process similar to the one described above for <Construction> when applying the <Validation> element.

Do not use the less-than or greater-than symbols (<, >) in your validation code because the XML parser will confuse them with the beginning and end of a tag. Use < and > as shown in the following example:

```
<Validation>
   if (WebPort &lt;= 0)
     throw new RuntimeException( "Incorrect value for WebPort" );
</Validation>
```

You can also find an example in the definition of the Switch resource in the DocExample.

<Bean> <FormValidations>

Form validation is done in the form validation Java classes which are deployed with Struts for the inventory UI. These Java classes are generated and deployed by the Inventory Builder. Validations done on forms are based directly on the data that is entered into the fields of the forms. With the <FormValidations> element you can append code to the validate method that is generated for the resource in the form validation class. After running Inventory Builder you can find form validation classes as well as other generated Struts (action) classes in a directory hierarchy under struts_classes. The form validation Java file for the *Bean* resource will be named *Bean*Form.java. Along with each validation you can define an error message as a property that will appear in the file *Bean*ApplicationResources.properties file which is colocated with the generated Java files. The property must consist of an entry name that you reference in the Java code and a displayable string. You can then localize the displayable message by editing the properties file.

The <FormValidations> element encloses definitions of individual form validations for the resource, each one given as a <FormValidation> element, which is repeatable.

The value of the <FormValidation> element is the Java code to be added to the validate method in the (Struts) form validation class for the resource. If you choose to define an error message as a property, use the attributes entry and message to define the name of the property and the message, respectively.

You can find an example form validation in the definition of the Switch resource in the DocExample..

The <Field> Element

This section details how to define fields for your resources. Each field in your resource definition translates into a property in the JavaBean and a column in the database table.

Each field has a type. Service Activator supports the field types shown in Table 2-2.

Table 2-2 Supported Field Types

Type	Description	Default value	Database type
String	Text string with a maximum length of 200 characters	null	VARCHAR2(200)
String(n)	0<n<=4000, defines max length of string	null	VARCHAR2(n)
int	32-bit integer number	0	NUMERIC(10)

Type	Description	Default value	Database type
long	64-bit integer number	0	NUMERIC(20)
float	Floating point number	0.0	REAL
double	Floating point number (double precision)	0.0	FLOAT
boolean	Boolean value: “true” or “false”	“false”	CHAR(1)
Date	Displayed as a formatted string; stored as seconds since January 1, 1970	null	TIMESTAMP
Blob	Binary large object	null	BLOB
Clob	Character large object	null	CLOB

Values for fields of Blob type can be assigned by workflows, but they cannot be entered from or shown on the inventory UI. These fields cannot have the mandatory attribute set to “true”, and they will be hidden in Create and Edit operation forms, regardless of the hiddenCreate and hiddenUpdate attributes. In the View operation form, the value is shown as ‘Binary data (XX bytes)’.

Fields of Clob type will be truncated to 200 characters when shown in View operation forms.

The notation for field values in a resource definition follows Java. Surrounding quotes are omitted when an element value is interpreted as a field value (such as in the <Default> and <Value> elements).

All field declarations are enclosed within the <Fields>...</Fields> tags. A field declaration has the following general form:

```
<Field>
   <Name>fieldname</Name>
   <Column>columnname</Column>
   <Label>showname</Label>
   <Type>String</Type>
   <Description>A meaningful description of the field</Description>
   <ListOfValues>
      <Value>valA</Value>
      <Value>valB</Value>
      <Value>valC</Value>
   </ListOfValues>
</Field>
```

The only mandatory tags are <Name> and <Type>.

<Field> mandatory

By default each field must have a value (mandatory=“true”, except for fields of type Blob). This is enforced when an instance of the resource is created or updated by an operation from the inventory UI, and also when it is written to the database as a table row (columns are declared as NOT NULL in SQL statements to create tables). The latter is significant when resource instances are written to the database from workflows or by SQL scripts to populate the database.

You can override both behaviours by setting the mandatory attribute “false”.

NOTE

Fields of certain types, including String, do not have default values unless explicitly defined (see Table 2-2). For these fields, if an initial value is not defined by the <Default> element or as a parameter of the Create action specified in the tree definition (see “Inventory Actions” in chapter 4), you must set the mandatory attribute “false”. For type Blob it is not possible in any way to define a default or initial value; hence for a field of this type the default is mandatory=“false”.

The syntax for allowing the value of a field to be omitted is:

```
<Field mandatory="false">
```

<Field> hiddenView

By default each field of a resource will be shown with its value in the View operation form on the inventory UI. You can force a field to be hidden in the View form by setting its hiddenView attribute “true”.

The syntax for declaring a field to be hidden in the View operation form is:

```
<Field hiddenView="true">
```

<Field> hiddenUpdate

By default each field of a resource will be shown with an editable value in the Update operation form on the inventory UI. You can force a field to be hidden in the Update form by setting its hiddenUpdate attribute “true”.

The syntax for declaring a field to be hidden in the Update operation form is:

```
<Field hiddenUpdate="true">
```

<Field> update

By default each field of a resource that is shown will be editable in the Update operation form on the inventory UI. You can prevent editing of the value by setting the update attribute “false”.

The syntax for preventing editing of a field in the Update operation form is:

```
<Field update="false">
```

<Field> hiddenCreate

By default each field of a resource will be shown and allow a value to be entered in the Create operation form on the inventory UI. You can force a field to be hidden in the Create form by setting its hiddenCreate attribute “true”.

When data entry is prevented in this way the value for the field may be a default value for the field or the type, or it may originate from a parameter for the Create action specified in the tree definition.

The syntax for declaring a field to be hidden in the Create operation form is:

```
<Field hiddenCreate="true">
```

<Field> create

By default each field of a resource that is shown will allow a value to be entered or edited in the Create operation form on the inventory UI. You can prevent editing of the value by setting its create attribute “false”. The prepopulated field value that is shown is then greyed out. It may be a default value for the field or the type, or it may originate from a parameter for the Create action specified in the tree definition.

The syntax for preventing editing of a field in the Create operation form is:

```
<Field create="false">
```

<Field> sequence, <Field> sequenceStart, <Bean> <SequenceName>

You can specify by setting the sequence attribute ”true”, that the value for a field must come from a database sequence. This feature is suited to generate unique primary keys for resources.

The type of the field must be int, long or String.

By default the sequence attribute is ”false”, and no sequence is created.

By default, values will start from 1, incrementing by 1 for each resource instance. The starting value can be controlled by the sequenceStart attribute.

The syntax for generating sequence numbers, starting from 100, for a field is:

```
<Field sequence="true" sequenceStart="100">
```

NOTE A field whose value comes from a database sequence is hidden on the Create form, as the value has not been drawn yet, and there is nothing to enter.

By default, the name of the database sequence is generated from the name of the bean and the field. You can specify a value to use with the <SequenceName> child element of the <Bean> element.

NOTE Only one field in a bean can be associated with a database sequence.

<Field> searchable

By default all fields can be used to build search conditions on the inventory UI, and all field values are shown in search result lists. You can override this by setting the searchable attribute “false”, the field will then be hidden in search operation forms.

<Field> maxCount

Setting the maxCount attribute “true” on a field declares the resource to which the field belongs to be reservable, as if the maxCount attribute had been set on the <Bean> element (see the section “<Bean> maxCount - Reservable Resource”, above). The value assigned to the field, when an instance of the resource is created, will serve as the initial value of the “availability counter” implemented with the automatically generated field named count__. In this way the number of reservable units does not have to be specified statically in the resource definition, and it can vary among instances of the resources.

The field must be of type int or long. The maxCount attribute cannot be set on more than one field, and if it is set on the bean, it cannot be set on a field. The value must be specified as a boolean, the default value is “false”.

<Field> dateFormat

By default fields of type Date are formatted according to the locale and must also be entered in this format. You can override the format defined by the locale by setting the dateFormat attribute. The legal values for this attribute are those of the Java type SimpleDateFormat, for example:

```
<Field dateFormat="yyyy-MM-dd">
```

<Field> integerFormat

By default, when values of type int or long are shown on the UI, they will be formatted according to the locale setting. Typically, punctuation is used as a thousands separator. If you want to show the raw integer format, you can disable this behaviour for a field:

```
<Field integerFormat="false">
```

<Field> password

By setting the password attribute “true” you declare that the value of the field shall be treated as a password, i.e. it is shown on the UI as a sequence of asterisks. By default the attribute is “false”.

The password attribute should not be used for primary key fields or foreign key fields.

<Field> encrypt

By setting the encrypt attribute “true” on a field which also has password = “true”, you specify that the field value shall be stored in the database in encrypted form. By default the attribute is “false”.

NOTE There are several ways to decrypt the password when it has been retrieved from the database. It can be done in a workflow or in a plug-in before the password is sent to a target.

<Field> store, <Field> <Loader>

By default each field of a resource will be stored in the database and its value retrieved when it needs to be shown. You can define a field with a value that is calculated when it needs to be shown in the UI, but without any corresponding column in the database table. This behaviour is accomplished by setting the store attribute ”false” and providing Java code to calculate the value using the <Loader> element.

If the field is a primary key, the store attribute will be ignored.

The Java code you provide in the <Loader> element will be inserted as the body for a method in the bean, which will be used to define the value for the field. The method has a parameter `con`, a database connection which makes it possible for you to perform database queries in the body code. Disregarding the database connection, your code should look like this (assuming the type of your field is int):

```
int val;
// your calculations here, setting a value for val
this.fieldname = val;
```

For further study: The class WebServer in the DocExample uses this feature. Look at the definition of the <Loader> element used for the field AverageDelay and explore further by finding the getValueAverageDelay method in the generated Java code.

<Field> <Name>

The field name must be unique within each resource. It must contain only alphanumeric characters or underscores, and must start with a letter. It is used to name the corresponding property (with getter and setter) in the Java bean as well as the database column, unless overwritten with the <Column> element. If you do not override it with the <Label> element, it will also appear on the UI.

NOTE The field name must not equal the name of the bean to which the field belongs, or of any bean in the inheritance chain.

NOTE The field name cannot be a Java keyword or an HPSA reserved keyword. Unless overridden with <Column>, it also cannot be an SQL reserved word. See “Resource Definition Validation” in “Using Inventory Builder” for more detailed information about the validation.

<Field> <Column>

Each field in your resource has a corresponding column in a database table. By default the name of the column is the same as the name of the field. You can override this default with the <Column> element. Do so if the name you want for the field is an SQL reserved word. Column names must contain only alphanumeric characters or underscores, and must start with a letter.

As an example, in the definition file for WebServer, there is a field with name “user”. As this is an SQL reserved word, we define the column name “User_Name” as follows:

```
<Field>
   <Name>user</Name>
   <Column>User_Name</Column>
</Field>
```

Note : The column name must adhere to SQL naming restrictions.

<Field> <Label>

Each field in your resource has a corresponding label in inventory UI. By default the value of the label is the same as the name of the field. You can override this default by using the <Label> element.

```
<Field>
   <Name>ipaddress</Name>
   <Column>ip_addr</Column>
   <Label>IP</Label>
</Field>
```

<Field> <Type>

Every field must have its type specified. Use the <Type> element to define the type of field you are creating. See Table 2-2 for a list of supported types.

<Field> <Default>

Use the <Default> element to specify the default value for the field.

NOTE It is not possible to specify a value of type Blob. Clob values are like String values, unquoted strings. When you specify the default value for a field of date type, use the dateFormat value that applies, for example:

```
<Default>2008-01-30</Default>
```

<Field> <Description>

Use the <Description> element to specify a short description that is presented on the UI forms to describe the field. The description should add information to the name. If the name says it all, omit the <Description>.

<Field> <ListOfValues>

The <ListOfValues> element controls the behaviour of the resource in operation forms on the inventory UI. It has no impact on the generated Java bean code. With this element you specify the valid values for the field, so that on the Create and Edit operation forms the user is presented with a drop-down list to select from, and will not need to type a value for the field.

There are two ways to specify the valid values. The first way is to enumerate them, using a <Value> element for each one. The second way uses a findBy method to retrieve candidate instances of a specified bean.

The <Value> child element of <ListOfValues> specifies one valid value for the field. The value displayed in the drop-down list is also the value assigned to the field if picked by the user, unless the display value is overriden with a string specified using the show attribute on the <Value> element. The selected attribute can be set “true” on at most one <Value>, causing that value to be preselected in the drop-down list. Here is a simple example showing the form of the <ListOfValues> element:

```
<Field>
   <Name>example</Name>
   <Type>String</Type>
   <ListOfValues>
      <Value>A</Value>
      <Value selected=”true”>B</Value>
      <Value>C</Value>
   </ListOfValues>
</Field>
```

Alternatively, you can use the <BeanName>, <Label> (optional), <Method> (optional) and <Param> (optional) child elements of <ListOfValues>, in that order, to specify a query that will retrieve the values to be shown in the drop-down list, as follows:

<BeanName>	the complete Java class name of a bean (not necessarily a resource bean)
<Label>	names a field on the foreign bean whose value will be shown in the drop-down list
<Method>	the method on the bean which is used to retrieve candidate instances; default is findAll; it must be a method which returns an array of instances of the specified bean
<Param>	parameter for the method (findAll takes no parameters); here you use constants and names of non-editable fields of the target bean (the one in whose definition the <ListOfValues> occurs)

By default, the specification with <BeanName>, etc., applies to a foreign key field (see "The <Key> Element and findBy Methods" below). In this case the specified bean must be the foreign bean or a specialized (child) bean that inherits from it.

The default can be overridden by setting the attribute withoutForeignKey "true" on the <ListOfValues> element. Then any bean can be used.

The value selected for the target field will be the primary key of the selected bean instance, regardless of which field is shown as determined by <Label>. If, for example, the primary key is a sequence number, then the <Label> allows you to present a more user-friendly name. The primary key must consist of single field.

NOTE The value of the <Label> field of the the foreign bean will also be shown in the View form.

<Field> <ShowConditions>

With the <ShowConditions> element you can make the appearance of the field in UI forms dependent on conditions involving other fields of the resource. There can be several conditions, each one represented by one <ShowCondition> element; they will be combined using an operator specified by the operator attribute, which can be 'and' (the default) - all conditions must be true - or 'or' - at least one condition must be true.

Each condition refers to one field specified by <FieldName> and is either the "empty" condition (if the empty attribute is "true") or a pattern match condition (the default). The empty condition is true if the field has no value (is null). The pattern match condition is true if the value of the field matches the regular pattern specified by <Pattern>.

Show conditions are evaluated dynamically when field values change. For example, if field A is shown on condition that field B is empty, then field A will appear when the contents of field B is deleted.

The <Key> Element and findBy Methods

By defining keys for a resource you control important aspects of the Java bean and the database table for the resource:

- the methods that can be used to find unique resource instances or lists of resource instances matching supplied field values
- the primary key of the database table
- indexes on the database table to facilitate efficient searches
- uniqueness constraints on the database table
- modelling of references between resource instances enforcing referential integrity

A key on a resource is one or more fields used in combination. You must define exactly one primary key for each resource. It may consist of multiple fields, but in most real cases it is a single field. The primary key must be unique: it is not possible to create a new resource instance with a primary key value equal to that of an existing instance. Hence, when you search for instances with a given primary key value, you will get zero or one match. The same is true for any other unique key.

For every key you define on a resource, the generated Java bean will contain a findBy*Keyname* method to find all instances which match a value specified by a parameter - or several parameters in the case of a multi-field key.

NOTE

The *Keyname* used to name the findBy method is derived from the name of the key (see "<Key> <Name>" below) as follows: if the first character is in lower case, it is replaced with a capital letter; capital letters in any other position are changed to lower case.

If the key is unique, the findBy*Keyname* method will either fail or return a bean holding the single matching instance. If the key is not unique, the method may return multiple matching instances and will therefore always return an array of beans (0, 1 or more). Every resource Java bean will also contain a findAll method with no parameters, which will return all instances of the resource, one for each row in the database table. The findAll method can be used whereever you can specify a findBy method.

If you want a findBy method to join two (or more) tables, you can specify a key field which belongs to a different resource than the key. For details, see "<Key> <KeyField>" below.

The QueryInventory and ReserveResource nodes which you can use in workflows to retrieve or reserve resources make use of the findBy methods. The findBy methods are also used in the inventory UI to retrieve the lists of branches which appear when an existing branch is expanded.

NOTE

You can set up the QueryInventory node to return an array of beans, even if the key used by the findBy method you specify is unique, so that the method returns a single bean or fails.

All key definitions in a bean are enclosed between the <Keys>...</Keys> tags. Each one is given as a <Key> element. Each field of the key is defined by a <KeyField> child element.

Descriptions of attributes and elements of the <Key> element follow.

<Key> pk

Exactly one key in each bean must be declared as the primary key by setting its pk attribute "true" (default is "false"). The primary key will be enforced as unique. The types of each field of a primary key must be one of: String, Date, boolean, int, long.

NOTE

If the resource you are defining inherits from another type, then you should not repeat the primary key specification. A derived type automatically inherits the same primary key as the base type.

<Key> unique

If you set the unique attribute "true" (default is "false"), the key will be enforced as unique, and an index will be generated for the key on the database table, regardless of the setting of the makeIndex attribute.

<Key> makeIndex

By default, each key translates into an index on your database table. Indexes are automatically used by the database when they apply to queries that are made, for example by findBy methods. You can overrule this and prevent generation of the index by setting makeIndex "false". Do this if you want the resulting findBy method, but you know that an index on the key would be inefficient.

<Key> foreignBean, <Key> foreignSolution

A foreign key is a key which can reference an instance of a specified resource (normally different from the resource it is part of). The referenced resource is called the foreign bean. When a foreign key has a value (if not mandatory, it could also be null), it must equal the value of the primary key of an existing instance of the foreign bean (referential integrity).

The case where the primary key of the foreign bean comprises multiple fields is not ruled out by the syntax, but it is not generally supported. Some simple cases may work. If you want to make foreign bean references to a resource it is recommended to define a single-field primary key for it, for example a sequence number, even when some combination of other fields is known to be unique and might serve as the primary key.

You define the foreign key relationship by setting the foreignBean attribute equal to the name of the foreign bean. For each field of the foreign bean's primary key you must define a <KeyField> child element to define the matching field within the key. In the <KeyField> element the corresponding field in the foreign bean is specified by the foreignField arribute.

For example, assume that you want to set up a foreign key relationship between a new bean and the WebServer bean from the DocExample as foreign bean. The primary key of the WebServer has only one field, name. It appears as the foreignField of the foreign key specification. You have already defined a field of your new bean named webserver to be used for the foreign key and then declare the foreign key as follows:

```
<Key foreignBean="WebServer">
   <KeyField foreignField="name">webserver</KeyField>
</Key>
```

If the foreign bean is in a different solution then that solution must be specified with the foreignSolution attribute.

<Key> restrict

By default, entity relationships may cause cascade deletion (restrict="false"). If bean A has a foreign key which references bean B, then when an instance of B is deleted, those instances of A which are related to that instance (have a primary key matching its foreign key) will also be deleted, and this behavior may cascade to other beans which have foreign key relationship to bean A, etc.

If cascade deletion is unwanted, you can set restrict "true". Then the attempt to delete an instance which would cause cascade deletion, such as the B instance described above, will be prevented. In other words, the system checks that any references to an instance have been removed before that instance can be deleted.

<Key> nullOnDelete

The nullOnDelete attribute provides an alternative to the restrict attribute. When restrict="false", but nullOnDelete="true", a bean instance A which references B that is deleted will not be (cascade) deleted, but its foreign key field(s) will be set to null. Default is "false".

<Key> advancedSearch

This construct is related to advanced searches in the inventory UI, described in "*HP Service Activator, User's and Administrator's Guide*". Advanced searches are based on nested child branches in the tree structure. For each child branch the child instances are determined by a findBy method. Typically this method involves a field on the child which is a foreign key identifying the parent. It does not have to be the only key field of the findBy method; there can be additional fields to narrow the results. Advanced search relies on a special method beased on the <Key> behind the findBy method used in tree, in addition to the findBy method itself. Whenever a <Key> includes a key field which on its own is a foreign key, such an additional method is automatically included in the generated bean code. In particular this is true for the <Key> with foreignBean.

Now, if you want to use a <Key> to select the instances of a child branch, but it does not include a foreign key field, and you want to be able to use the child branch in advanced searches, then you must set advancedSearch “true” for that <Key>, and the necessary method will then be included in the bean.

<Key> joinedBy, <Key> <WhereMap>

With this attribute and element you can control how multiple fields of a key are combined in the generated findBy method. There are three possible values for the joinedBy attribute: “AND”, “OR” and “MAP”; default is “AND”, meaning the findBy method will retrieve resource instances which match all key field arguments. “OR” means at least one key field must match.

The value “MAP” allows you to control in more detail with the value of the <WhereMap> element how to construct the SQL select statement to be used; see the section “Generated SQL”.

<Key> <Name>

The important use of the name of a key is in the naming of the generated findBy method. By default the key inherits its name from the field(s) of the key. If there is more than one key field, the names are concatenated. You can override the default naming by explicitly specifying a name with the <Name> element. Example:

```
<Key>
   <Name>addrAndPort</Name>
   <KeyField>addr</KeyField>
   <KeyField>port</KeyField>
</Key>
```

The findBy method will then be findByAddrandport. Without the <Name> element, the name of this key would have been addrport (the findBy method would have been findByAddrport).

<Key> <KeyField>

The <KeyField> element can be repeated; there must be at least one (in each key). Each occurrence of the <KeyField> element specifies a key field by stating the name of the field, which must be defined by a <Field> element. The order of the key fields determines how the (default) name of the key is constructed.

NOTE

In the definition of an inheriting (subclass) resource it is possible to use a field of the superclass as key field, if such a key was not already defined on the superclass. The subclass will then have a findBy method which the superclass (and sibling classes) does not have. You can find an example of this in the UnixWebServer resource definition.

When resources are related by foreign keys, you may want to be able to find instances of one resource, the *target* resource, based on the value of a field of another related resource (in database terms you want to join the tables). You can then use the externalBean attribute to specify that a key field does not belong to the same resource (the target) as the key, but to a different, *external* resource, identified by the value of the attribute. The foreign key, which makes the joining possible, is not mentioned in the definition of the key field. The relationship can go either way: the target resource can have a foreign key that references the external resource or vice versa. It cannot go both ways; that is an impossible construction. But it is possible that there can be more than one foreign key in the same direction, you must then resolve the ambiguity by means of the joinField or externalJoinField attribute.

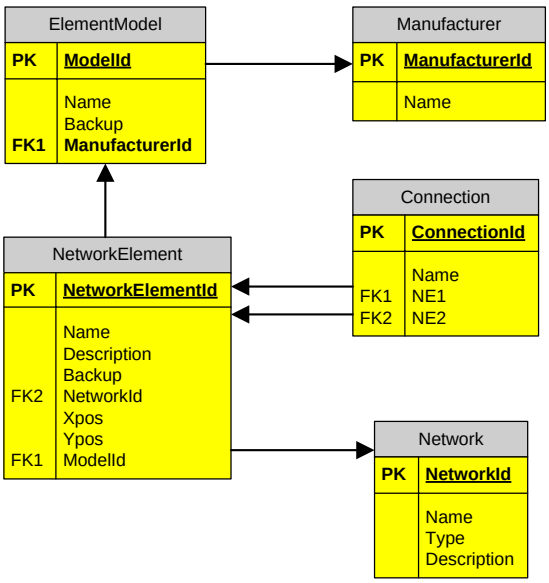
The <KeyField> element has the following attributes:

foreignField	for a foreign key, defines the foreign field (see “<Key> foreignBean” above)
externalBean	optional; specifies a resource different from the target resource and requests that the findBy method shall join the two resources
alias	optional; if a key contains two or more external fields belonging to (different instances of) the same external bean, the alias attribute must define different

	names for the occurrences of the external resource to be used in the generated SQL statement
joinField	optional; if the target resource has multiple foreign keys referencing the external resource, the value of joinField must identify the one to use
externalJoinField	optional; if the external resource has multiple foreign keys referencing the target resource, the value of externalJoinField must identify the one to use
ignoreCase	optional; when “true”, generated findBy method will not distinguish letters in upper and lower case when parameters are matched to values of key fields
comparator, compareTo	for fine control (along with <WhereMap>) of the where clause of the SQL select statement that will be generated by the findBy method. See the section “Generated SQL” for details.

As an example of the use of external keys, consider the resources depicted in Figure 2-1.

Figure 2-1 Find by External Key



If you want a findBy method (in the NetworkElement bean) to find those NetworkElements which are produced by a specific manufacturer - with known ManufacturerId, you will need to use ElementModel as an externalBean, because the ManufacturerId is not found as a field on the NetworkElement itself. You can define the <Key> to cause this method to be generated as part of the definition of NetworkElement as follows:

```
<Key>
   <KeyField externalBean=”ElementModel”>
      ManufacturerId
   </KeyField>
<Key>
```

Remember this external key was enabled by the existence of the foreign key from NetworkElement to ElementModel:

```
<Key foreignBean=”ElementModel”>
   <KeyField foreignField=”ModelId”>
      ModelId
   </KeyField>
<Key>
```

The Inventory Builder will combine the information defined by these <Key> elements in the findBy method generated for the key ManufacturerId.

By using a key with two externalBean fields you could request a method to find all NetworkElements of a named model in a named network:

```
<Key>
   <Name>ModelAndNetwork</Name>
   <KeyField externalBean=”ElementModel”>
      Name
   </KeyField>
   <KeyField externalBean=”Network”>
      Name
   </KeyField>
<Key>
```

A more complex example is to find all Connections between two named NetworkElements. Here you need the alias and joinField attributes to distinguish between the two occurrences. This <Key> definition will belong to the Connection resource:

```
<Key>
   <Name>FirstToSecond</Name>
   <KeyField externalBean=”NetworkElement alias=”first” joinField=”NE1”>
      Name
   </KeyField>
   <KeyField externalBean=”NetworkElement alias=”second” joinField=”NE2”>
      Name
   </KeyField>
<Key>
```

<Key> <JoinBridge>, <Bean> <JoinBridges>

The <JoinBridge> element is used with external key fields which are described in the preceding section “*<Key> <KeyField>*”.

Join bridges are used in cases where you need additional resources to bridge a gap between the target resource, the one you want to create a findBy method for, and the external resource, the one you can provide a parameter value for. In the first example above, to find the NetworkElements produced by a certain manufacturer, suppose you want to provide the name of the manufacturer as parameter instead of the Id, then your external resource will need to be the Manufacturer, not the ElementModel, because the ElementModel does not hold the manufacturer’s name. However, there is no direct relationship between the NetworkElement and Manufacturer resources. You can solve that with a <JoinBridge> element to specify the intermediate resource that you need to jump from external to target. To cover also the case where your bridge will require more than one jump, the jump is specified with a repeatable child element of <JoinBridge>, called <Jump>. In the example, the jump resource needed is ElementModel, because it has the necessary relationships to both the external and the target resource. The <Key> element (belonging to the target, i.e. NetworkElement) will look like this:

```
<Key>
   <KeyField externalBean=”Manufacturer”>
      Name
   </KeyField>
   <JoinBridge origin=”NetworkElement” destination=”Manufacturer”>
      <Jump>ElementModel</Jump>
   </JoinBridge>
<Key>
```

Even with a bridge, the relationships between neighboring resources in the chain can go both ways. The findBy method will be generated so it takes the direction of the relationship into account.

The <JoinBridge> element has two mandatory attributes origin and destination. They identify (by name) the target and external beans that are being joined and establish forward (origin to destination) and reverse direction. Identification of the external bean is important if there is more than one external key field in a <Key> element.

The findBy methods generated by keys with <JoinBridge> elements may produce duplicate results. To avoid that, set uniqueResults=“true” on the <Key> element.

The value of the <Jump> element identifies the jump resource by its name. To resolve the ambiguity if there is more than one foreign key from a jump resource to a neighbour (other jump, external or target), the <Jump> element has two optional attributes to point out which one to use: jumpField for the forward direction, inverseJumpField for the reverse direction. Similarly jumpField and inverseJumpField can be used on the <JoinBridge> element to resolve ambiguity on the origin and destination beams, respectively.

NOTE

Wherever there can be a reference to a bean outside of the one being defined, that bean may belong to a different solution, in which case the solution must also be identified. For the externalBean attribute in the KeyField element this is done with the externalSolution attribute, likewise for origin and destination in JoinBridge with originSolution and destinationSolution, and for the Jump element with the solution attribute.

Join bridges can defined - with origin, destination and jumps - globally within the <JoinBridges> element, or locally within the <Key> element. A global definition introduces also a name for the join bridge. Global join bridges can be applied in several <Key> elements; the <JoinBridge> element which appears at the point of application (in the <Key>) then contains only the name of the join bridge.

The <Operations> Element

The inventory UI uses Java Server Pages and other files that are deployed in the Struts framework to show resource-specific operations form to create, view, edit, delete and search for resources. Some of these files (for Create, Edit and Delete) along with methods they call in the Java bean (store, update and remove) are only generated when requested in the resource definition. Normally you will want to generate them, but if your inventory is populated from an external source, you may not want to enable these operations; then you can save generation of the files.

NOTE

To enable operations on the UI you must also add them to branches of the presentation tree, as described in chapter 4.

You use the child elements of the <Operations> element to specify which UI operations and associated bean methods you want to be able to use.

<Operations> <Store>

This element has no value. Its presence specifies that the UI files needed for the Create operation and the associated store() method in the Java bean shall be generated.

<Operations> <Update>

This element has no value. Its presence specifies that the UI files needed for the Edit operation and the associated update() method in the Java bean shall be generated.

<Operations> <Remove>

This element has no value. Its presence specifies that the UI files needed for the Delete operation and the associated delete() method in the Java bean shall be generated.

<Operations> <CreatePartial>

This element has no value. It can only be used in a subclass bean (inheriting resource definitions). Its presence specifies that the UI files needed for the operation to extend an instance of the superclass (parent bean) to become an instance of the subclass (child bean) and the associated createPartial() method in the Java bean shall be generated.

<Operations> <DeletePartial>

This element has no value. It can only be used in a subclass bean (inheriting resource definitions). Its presence specifies that the UI files needed for the operation to reduce an instance of the

subclass (child bean) to an instance of the superclass (parent bean) and the associated deletePartial() method in the Java bean shall be generated.

<Operations> <FindBy> and <FindBy> <OrderField> Element and Attributes

The optional <FindBy> element belongs functionally to the <Key> element, see “The <Key> Element and findBy Methods” above. It is used to specify how the results returned by the findBy method that is generated for a key shall be ordered and is only relevant for non-unique keys. The result order controls how instances of a child branch are ordered when their parent branch is expanded in the instance view of the inventory UI. It is also significant when workflows use the methods to make queries.

The <FindBy> element must be repeated for each findBy method for which you want to control the order of results returned. You specify the method by the key attribute:

key	specifies the findBy method to control by the name of the key from which it was generated; omit the key attribute to specify the findAll method

The ordering of results is specified by the <OrderField> child element:

<OrderField>	The value of the element is the name of a field whose value is used to sort the results; details of the sorting process can be controlled by attributes of this element:
desc	if “true”, sorting will be done in descending order, default value is “false”

For example, to specify that the findByPort method (for the Web server resource) shall return results ordered by their IP addresses, use:

```
<FindBy key="port">
   <OrderField>ipaddr</OrderField>
</FindBy>
```

Reservable Resources

Often in activation solutions it will be necessary to allocate and reserve specific resources. In some integrated solutions, this function may be performed by a different subsystem. In TMF eTOM (Release 8, see *GB921, eTOM Business Process Framework, Addendum D*) allocation of resources is identified as part of the level 3 process 1.1.3.2.1, Allocate & Install Resource, whereas activation takes place in the level 3 process 1.1.3.2.2, Configure & Activate Resource, both of these belonging to the level 2 process 1.1.3.2, Resource Provisioning.

Typically a resource is reserved for a unique end customer (service subscriber) of the service provider, for example a port on an access device where the subscriber’s access connection is attached, or an immaterial resource such as an IP address. Other types of resources may offer a capacity that can be shared among multiple users, up to a maximum number. The selection and identification of resources is a significant step in the overall provisioning process, as discussed in the eTOM standard reference above.

To facilitate the reservation of resources in Service Activator inventory, a reservable resource is modelled with an availability counter in an autogenerated int field named count__with label ‘Unused’ on the resource bean, where a value of 0 indicates that the resource is fully booked, and a value greater than 0 is the number of units that are available for reservation. A new booking is made by executing a reserveResource method, which will decrement the availability counter. Conversely a resource is released from a reservation by a call on a releaseResource method which will increment the availability counter.

The initial (and maximum) value of the count__ attribute can be defined in two ways: as a static value defined on the bean, using the maxCount attribute (see “<Bean> maxCount - Reservable Resource”), or as a value supplied when an instance of the bean is created. In the latter case you must define a field to hold the value and set the maxCount attribute on this field (to “true”, not the maximum number, see “<Field> maxCount”).

Normally reservation and releasing of resources take place in workflows. Workflow nodes dedicated to these functions - ReserveResource and ReleaseResource - are available in the workflow node library (see *HP Service Activator, Workflows and the Workflow Manager*). These workflow nodes use methods that will automatically be generated as part of the Java bean for a reservable resource (see below).

It is also possible to reserve and release resources by performing operations on selected resources from the inventory UI. These operations will also use the methods in the Java bean. See “Inventory Actions” in chapter 4.

Methods to Reserve Resources

The Java bean for a reservable resource will include, for each findBy*Keyname* method, a matching method reserveResourceBy*Keyname*. For example, a method findByPrimaryKey() has a matching method reserveResourceByPrimaryKey(). The reserveResource() method matches the findAll() method, i.e. the candidates are all instances of the resource class.

If the key is unique, the method will attempt to reserve the unique resource identified by the parameter(s). If the key is not unique, then the method will attempt to select a resource instance that matches the key parameter(s) and is not already reserved.

In the implementation, the reserveResource*Keyname*() method performs the equivalent findBy*Keyname*() query with the additional qualification that the availability counter must be greater than 0. It then chooses the first resource returned from the query, reserves that resource, and returns it to the caller.

Inheriting Reservability

The methods to reserve resources can also be called on beans of an inheriting subclass, in other words the property of being reservable is inherited.

When defining inheriting resources that must be reservable, set the maxCount attribute on either the superclass or the subclass, but not on both. Where to set it should be decided based on what is being modelled. If all instances of a superclass will be reservable for a common reason, set it on the superclass. If reservability is thought of as a property of the subclass(es), and it might not apply to all possible subclasses of a superclass, set it on the subclass(es).

Generated SQL

All the generated SQL statements that are used by generated Java beans are separated out from the Java code and placed in subdirectories of the jsql directory under the inventory directory.

findBy Where Clauses

For each <Key> element in a resource definition a findBy method is generated. This method will use an SQL select statement which will include a where clause to express the condition determined from the <KeyField> elements. There will be an expression for each key field, combined by operators. The expression will compare values in a given column of the database table holding the candidate beans instances to an argument (or several) of the findBy method. The first operand in each expression is the name of the column, the second is a string including ‘?’ (question marks) to represent the arguments. By default the operator is ‘=’ (equal sign) and the second operator a single ‘?’. For example, the name key in bean Rack in in the Doc_Example:

```
<Key pk="true">
   <KeyField>name</KeyField>
</Key>
```

results in the where clause:

```
name = ?
```

With the attributes comparator and compareTo you can specify the operator and the second operand to overrule the defaults. You will rarely need to define compareTo, as there is a suitable default for each possible value of comparator, as shown in the following list:

Comparator	default value of CompareTo
=	?
LIKE	'%'||?||'%'
BETWEEN	? AND ?
IN	(?, ?, ?, ... , ?)

When the key has multiple key fields the where clause will contain an expression (as described above) for each one, combined with AND, OR and possibly parentheses. By default, all the expressions will be combined with AND. You can use the joinedBy attribute on the <Key> to overrule this default. You can set it to OR, meaning all expressions will be combined with OR, or you can the value MAP, which lets you define a map for the where clause with the <WhereMap> element. The map combines references to key fields with AND, OR and parentheses, for example:

```
(${kf1} AND ${kf2}) OR ${kf3}
```

Here `${kf}` represents the expression for the key field whose name is `kf`.

The references to key fields take different forms depending on how the key field is defined.

Key field named `kf`, defined as:	Key field reference takes this form:
simple local key field	`${kf}`
external key field, external bean named `eb`	`${eb.kf}`
external key field, external bean named `eb`, with joinField named `jf`	`${eb.jf.kf}`

When an external bean is used, the code to join the tables by the foregn key (or join bridge) that ties them together is automatically inserted in the expression generated for each key field reference.

The map may also contain parts of the where clause which do not refer to key fields, written in literal form; only the recognized key field references will be expanded to expressions.

It is even possible for a <KeyField> with externalBean to leave empty the value of the field, in the terms from above `kf` is the empty string. Then you cannot use the first form, but the second and third forms are still possible, now they will be only `${eb}` and `${eb.jf}`(no dot before the empty `kf`) The code generated for these forms of the key field reference will only be the join code, any comparison expression must be written explicitly.

Table Name Aliases

All generated sql statements use aliases for the table names. For example for resource definition Rack in the Doc_Example the alias will be DocEx#Rack. The alias is introduced in the SQL statement from clause. When the alias is taken into account, the where clause shown above for the name of Rack will be:

```
DocEx#Rack.name = ?
```

You can overrule the generated alias by including <DBAlias>. This is mandatory if the bean name is longer than 18 characters. The maximum allowed length of the alias is 27 characters.

Unique and distinct findBy results

Two attributes are available to avoid duplicates in the results from findBy methods: uniqueResults and distinct. The difference is in the SQL keyword that is inserted to achieve the effect, the first one uses UNIQUE, the seconds uses DISTINCT.

Generated Java Bean Classes

For every resource definition (we name it `bean`), two Java class source files will be generated:

`bean_.java` this file contains the bulk of the generated code in a superclass named `bean_`

`bean.java` this a short file defining the class `bean` which inherits from class `bean_`; there is only a small amount of code in this file that depends on the contents of the resource definition

The two file approach is used to allow you to extend the generated code with explicitly Java code, typically findBy methods in addition to those generated from <Key> elements of the resource definition. If you do that, place your own code in the `bean.java` file. Then, if you happen to modify the resource definition, the task of merging your hand written code with the generated code will be quite simple. All the complex generated code goes in file `bean_.java`, which you should not need to modify.

3 Inventory Builder

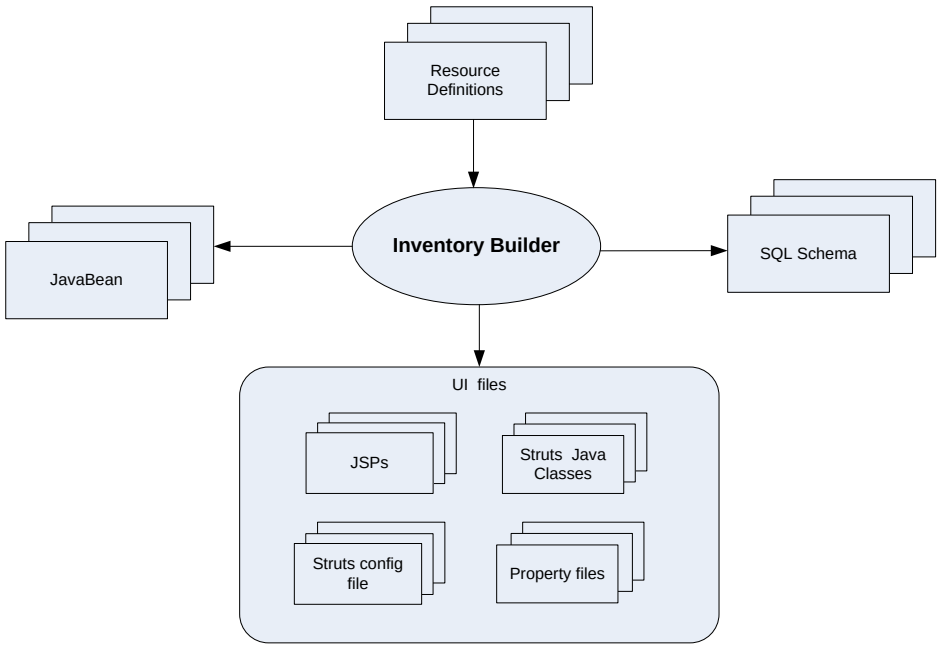
This chapter describes how to use the Inventory Builder tool to process resource definition files to generate the Java files, SQL files and UI files and to deploy them.

Inventory Builder is invoked from a command line. It has no UI for control.

For each resource definition, as shown in Figure 3-1, Inventory Builder typically generates:

- One Java bean code file.
- SQL data definition statements to create and drop the table, sequences, and indexes needed to implement the resource in the database.
- UI files: Structs form action and validation classes with associated property files, JSPs for presenting forms in a web browser, and Struts configuration file to tie it all together.

Figure 3-1 Inventory Builder



You use Inventory Builder in two steps:

- In step 1 you generate the artifacts (files) described above; this typically takes place in directories outside of the hierarchy used by a running system.
- In step 2 you compile the Java code and deploy the final artifacts as parts of an operational Service Activator system. The Java classes are packed in a Java archive file (.jar file) and copied along with UI files to the proper locations for use by the JEE engine (including workflow manager). The SQL data definition scripts are executed to create the database tables to store resources.

You must run the Inventory Builder in each step with different arguments.

For step 1 the command line syntax is:

```
InventoryBuilder <generation option>* <sql option>* <other option>*
<source file>+
```

Typically, without any options, to process all resource definition files in the inventory directory:

```
C:\....\inventory> InventoryBuilder *.xml
```

For step 2 the command line syntax is:

```
InventoryBuilder <deployment option>+ <db option>* <other option>*
```

Typically, with deployment options to compile all Java files and to deploy all compiled beans and JSPs:

```
C:\....\inventory> InventoryBuilder -compile -deployBEAN -deployJSP
```

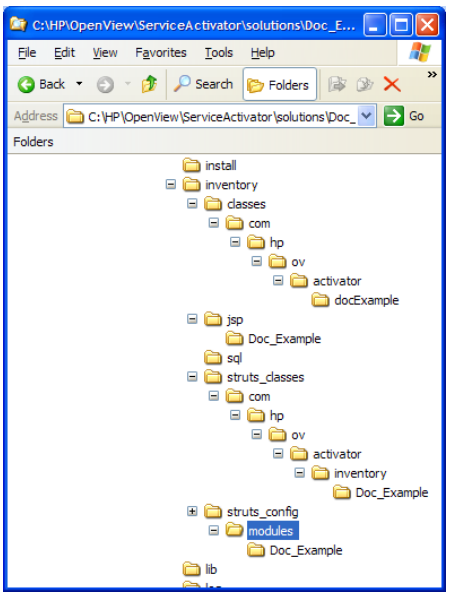
This is typical, but not complete; the example is missing the `–deploySQL` option to create the database tables, but it is often convenient to do that separately.

The presence of at least one <deployment option> determines that the Inventory Builder is running in deployment mode (step 2).

NOTE Options are not case-sensitive. Values containing space(s) must be surrounded by double-quotes (").

For both steps, the current directory of the command line processor, shown as 'inventory' in the examples above, plays an important roles. By default, all the resulting files from step 1 processing are written to subdirectories of the current directory, and in step 2 these files are read from the same directories. In Figure 3-2 you can see the hierarchy of directories created in step 1 under the inventory directory.

Figure 3-2 Inventory Builder Directory Structure



Files generated by the Inventory Builder are placed as follows:

classes	resource bean Java files, one for each resource, in the subdirectory determined by the complete Java class name including package
jsp	JSPs for operation forms; if a solution is specified in the resource definitions, the JSPs are placed in a sub-directory named like the solution; there are several JSPs for each resource, for different operations

sql	SQL scripts with data definitions to create and delete tables, etc; statements for all resource definitions processed together are grouped in two files: create.sql<Solution> and delete<Solution>.sql, to create and delete, respectively, the tables, indexes and sequences for the resources
struts_classes	Java classes for operation forms and form validation, several for each resource, in subdirectories determined by the complete Java class name including package
struts_config	Struts configuration file. If a solution is specified, a Struts module is generated and its configuration file is placed in a subdirectory named like the solution under the modules directory. If not, the configuration file is placed directly in the struts_config directory.

In step 2 all the Java class files will be packed in a Java archive (.jar) file and placed in the lib directory (not shown in Figure 3-2). It is the .jar file which is actually deployed to the JEE engine.

Details for Step 1

Command line syntax for step 1:

```
InventoryBuilder <generation option>* <sql option>* <other option>*
<source file>+
```

Resulting artifacts (files) are generated for resource definition files which are specified as <source file> on the command line. You can process a single file or many files together. However, all artifacts that have been generated in the results directories are deleted for each new run.

Resource definition files can be located anywhere, referenced by path names, possibly in multiple directories. Wildcarding can be used in the file name, as shown in the (typical) example above. If resource definition files to be processed contain references to other resources (transitively), which are not listed as files to be processed, then the definitions for those resources must be located in one or more directories declared with the -xmlpath option.

Table 3-1 Generation Options

Option	Description
-noJSP	omit generation of JSPs (and other files to be deployed in Struts)
-noSQL	omit generation of SQL data definitions scripts
-noBEAN	omit generation of resource bean Java source code files
-user_classes_path	absolute path of file folder containing user-written Java bean source code for the resources in question (see the section “Generated Java Bean Classes” in chapter 2), i.e. the parent folder of the `com` root folder. These Java files will be copied instead of generating from the resource definitions.

Table 3-2 SQL Options

Option	Description
-hpsa_data_tablespace <tablespace>	generated data definitions will place data tables (except history data) in the specified tablespace
-hpsa_index_tablespace <tablespace>	generated data definitions will place indexes for data tables in the specified tablespace
-hist_data_tablespace <tablespace>	generated data definitions will place history data tables in the specified tablespace
-hist_index_tablespace <tablespace>	generated data definitions will place indexes for history data tables in the specified tablespace

Table 3-3 Other Options (for step 1)

Option	Description
-app <directory>	overrides the current directory of the command processor as the master directory for generated files
-xmlpath	specifies directory(ies) other than current directory where Inventory Builder will look for referenced resource definition files. Multiple directories can be separated with ":" or ";"
-i <imports>	specifies name of java class(es) for which import statement(s) will be included in each generated Java Bean class. If there is more than one class, they can be separated with ":" or ";"
-scf <directory>	specifies struts config file in the specified directory will be merged with the generated struts config file. Note: The names of the files to be merged must be identical; the file name may include the solution name.
-debug	adds debug information to generated Java bean
-verbose	causes Inventory Builder to output verbose progress information including stack trace if an errors occurs
-help	outputs list of valid options
-version	outputs the version of the product

Details for Step 2

Command line syntax for step 2:

```
InventoryBuilder <deployment option>+ <db option>* <other option>*
```

There are six different deployment actions, each one needs to be specified with one of the options listed in Table 3-4 Several or all actions can be specified with a single command; compilation will be done before classes are deployed, database drops will be done before creates.

Java classes and UI files (forms implemented with Struts) are deployed by copying them to the JBoss JEE application server runtime locations under *$JBOSS_ACTIVATOR*.

As an alternative to the SQL options you can extract the data definition SQL commands from the generated files and execute them with a different tool such as sqlplus.

Table 3-4 Deployment Options

Option	Description
-compile	compile all generated Java classes under classes and struts-classes
-beanjar	name the .jar file containing compiled Java classes; the recommended name is inventory_*solution*.jar, where *solution* is the name of the solution
-deployJSP	copy UI files: JSPs, Struts classes and configuration files, to run-time locations
-deployBEAN	copy .jar archive with compiled beans to run-time location
-deploySQL	execute the generated file create.sql to create tables, indexes and sequences in database; requires DB options to connect to database
-undeploySQL	execute the generated file delete.sql to drop tables, indexes and sequences from database; requires DB options to connect to database

Table 3-5 DB Options

Option	Description
-dbHost <DBHOST>	name of the database host
-dbName <DBINSTANCE>	name of database instance
-dbPort <DBPORT>	port where database is accessed
-dbUser <DBUSER>	name of database user
-dbPassword <DBPASSWORD>	password for database user

DB options are used only with SQL deployment options to specify and provide credentials to access the database instance for the inventory data.

The first three db options, if omitted, get default values from data source for the workflow manager (mwfmDB) in the JBoss configuration file *$JBOSS_ACTIVATOR*/`standalone.xml`. This file is created when Service Activator is installed with the values specified at that time.

NOTE Regardless of the underlying database product, Oracle or Postgres Plus Advanced Server (PPAS) from EnterpriseDB, the database instance and the database user in combination specify the database schema that is used to store the inventory data.

The username and password, if omitted, can get values from the file *$ACTIVATOR_ETC*`/config/dbAccess.cfg`. The syntax is:

```
username=username
password=password
```

NOTE The dbAccess.cfg file is intended as a convenience for a development environment; it should not be present on a production system.

Table 3-6 Other Options (for Step 2)

Option	Description
-filebyfile	specifies Java files are compiled one by one, not grouped by directory; slow, but easier to track
-app <directory>	specifies a directory to override current directory as master directory for generated artifacts
-i <imports>	specifies Java class/jar files required at compile time; only needed if you have added import statements to generated code (by editing, or with features in resource definition)
-verbose	causes Inventory Builder to output verbose progress information including stack trace if an error occurs
-help	outputs list of valid options
-version	outputs the version of the product

4 Inventory Tree Definitions

Presentation of inventory data and associated operations is organized by means of explorer-style trees. You can and you must customize one or more trees; there is no default inventory data model, and there is no auto-generated presentation. You must think about and design how to organize the resources you have defined based on their entity relationships. This chapter is about the definition of an inventory presentation tree. Before you read it, you must have an idea about the actual user interface, how it works when a tree definition has been prepared and deployed. That topic is covered, based on the same DocExample that is used for this manual, in the chapter "Inventory User Interface" in *HP Service Activator, User's and Administrator's Guide*. It is assumed here that you have studied it.

A tree definition first defines some general aspects of the tree: its name, the solution it belongs to, declarations of operator privileges associated with branches and operations, and then the bulk of it will be branch definitions. For each branch there are a number of items to define: the associated resource, icon and label to display on the branch line, privilege needed to see the branch, method to determine the instances of the branch, given its specific ancestors in the instance view (for example, show the switches which belong to a particular rack), operations associated with the branch. Operations typically make up the bulky part of branch definitions.

You can define several inventory trees, each one in a separate file.

The XML syntax for a resource definition is specified in file inventoryTree.dtd that is found in *$ACTIVATOR_ETC*/config. As you can see in the DocExample files, resource definition files must begin with a header including a <!DOCTYPE> element to reference inventoryTree.dtd. The syntax is also easily understood from Table 4-1 below.

Inventory Tree Designer

The Inventory Tree Designer is a tool with a graphical user interface provided to help you create and edit tree definition files. Like other HP Service Activator tools it can be launched (on Windows) from the `start -> All Programs -> HP Service Activator` menu or from a desktop icon.

As you can see in the screenshot in Figure 4-1, the Inventory Tree Designer window has a Beans frame to the left and a Tree frame to the right. The beans frame provides reference (view and copy) to resource definitions that are relevant for the tree being defined, typically those defined for the solution the tree belongs to. The tree frame is the working area where a tree definition is created and modified. It has two subframes, the upper frame showing the tree structure, the lower frame showing details of the root or branch selected in the top frame.

The directory (multiple directories are also possible) from which resource definitions are retrieved can be configured persistently with the `Set beans directory...` command in the Settings menu, or a directory can be opened for a single session with the `Import beans...` command in the File menu.

The File menu has the usual commands for opening tree definitions from source files or as new, closing them and saving to files.

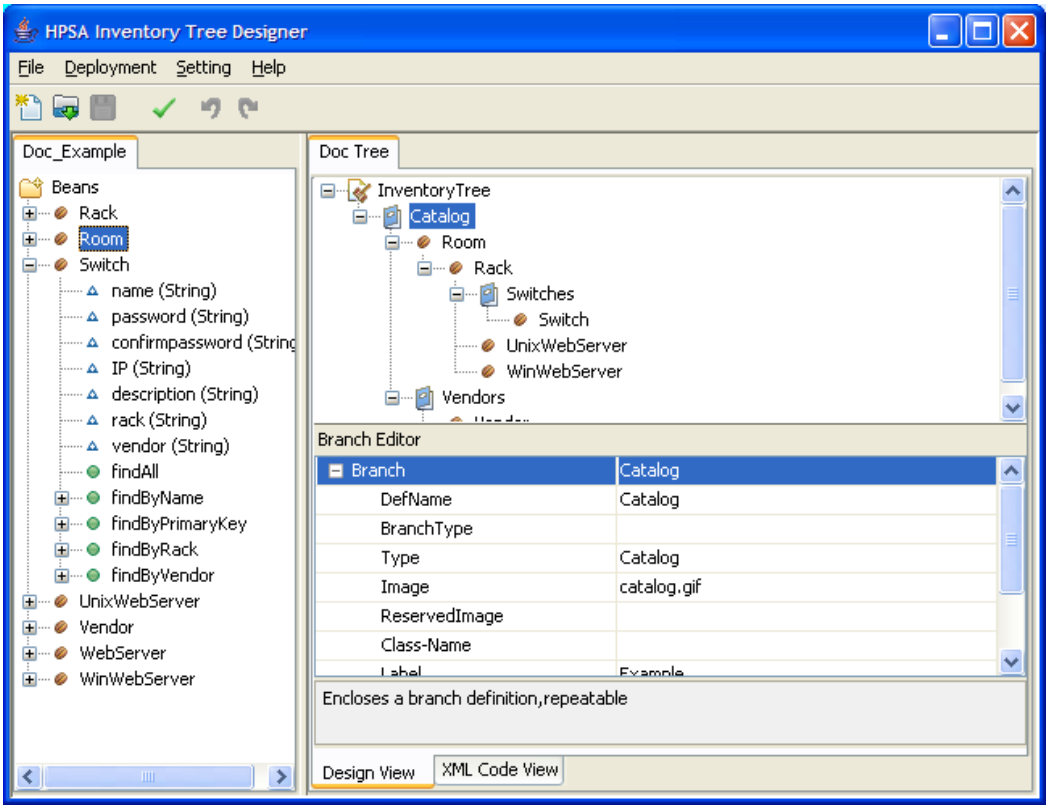
The commands in the Deployment menu are used to manage deployment of inventory tree definitions. You can deploy the currently open tree, or manage deployed trees. The latter function

allows you to list all deployed tree definitions and to select one of them to extract (download) to a file or to delete (undeploy).

The Deployment menu also has a function to assign a sequence number to a solution. This assignment which is also stored in the static repository defines the ordering of the tree tabs in the inventory user interface window when trees for multiple solutions are open.

The inventory tree deployment functions are described in more detail in chapter 5.

Figure 4-1 Inventory Tree Designer



The root of the tree shown in the upper part of the tree definition frame is always InventoryTree, i.e. the outermost element of the XML document. When it is selected, the lower part will show the non-branch child elements of the <InventoryTree> element; these elements define general properties of the target tree. The first branch (like Catalog in Figure 4-1), specifies the root branch of the target tree.

You can edit the branch structure of the target tree in the upper frame and details of each branch in the lower frame, when the branch is selected in the upper frame. Likewise general properties of the tree can also be edited in the lower frame, when InventoryTree is selected in the upper frame.

Menus that allow you to add new empty branches, and to copy or delete existing branches appear when you right click on a branch in the upper frame. You can also move a branch up or down among its siblings.

In the lower frame each tagged element or attribute of an element appears on a separate line, with element encloser and element-attribute relationships shown as indentation. Within each element is shown first its attributes, whose names begin with lower case letters, then its child elements, whose names begin with capital letters. Elements can be expanded or closed by clicking the +/- icon. The values of attributes and elements that appear in the right hand column are editable.

The tree designer is aware of the resource definitions in the beans frame. This is helpful when you add a new branch to a tree: the designer will let you choose the bean class to be associated with the branch from a drop-down list. Similarly when you edit a <Find-By> element in the context of a

resource, you can choose from the methods defined for that resource, and the <Key> child elements will be prepopulated.

NOTE For a non-leaf element (which does not itself have a value) the value that appears and can be edited to the right of the element name is actually the value of its first child leaf element. Most often, but not always, this is a <Name> element. The tag of the leaf element is not shown.

In many cases a menu will appear when you click on an element. If the element is optional, you can delete the occurrence. If it has optional child elements for which additional occurrences can be added, you can do that. You can also move an element up or down among its similar siblings.

Localization

A tree definition includes several strings which will be displayed to the user. In general these strings can be localized through resource bundle files. By means of the attribute bundle which applies to many elements you can specify the name of a file where the localized value of such a string element will be defined. The file name is interpreted relative to the classpath. It is recommeded to put your resource bundle file(s) under *$JBOSS_ACTIVATOR*/WEB-INF/classes/resources/<Solution>. If you do this, the specified file name must start with resources.

The value of the string element is then not taken as the final value to be displayed, but only as the key to access the actual value within the resource bundle.

The bundle attribute can be used on several elements. It can be specified at an outer level of the tree, e.g. on the <InventoryTree> element, which means the whole tree definition, and will then apply to all inner elements where it is relevant.

XML Vocabulary Quick Reference

Table 4-1 describes the XML vocabulary for presentation tree definitions. For each element that can occur in a resource definition, starting with the root tag <InventoryTree>, all its attributes and tagged child elements are listed and briefly explained in one row each, attributes before tagged child elements, tags enclosed in angle brackets <..>. You can print Table 4-1 as a handy reference when you work with resource definitions. For more thorough descriptions, refer to the following sections.

In a resource definition you will define several names and strings that will be shown on the UI. All of these names are candidates for localization. There are just a couple of names, of the inventory tree and the solution, for which you cannot use special characters and non-ASCII characters.

Table 4-1 Presentation Tree Definition Quick Reference

Parent tag	Attribute name/ child element tag	Mandatory/ default value	Description
<InventoryTree>	default	false	a tree with default=”true” is opened automatically when the inventory UI is launched
	allowClassView	true	determines if class view shall be possible for the tree
	bundle		name of localization resource bundle (applies to the whole tree)
	<Name>	M	name of the tree, shown in tree menu
	<SequenceNumber>	0	determines order in which trees are shown in menu on inventory UI

Parent tag	Attribute name/ child element tag	Mandatory/ default value	Description
	<Solution>		encloses solution definition
	<Description>		description of the tree, displayed by tools
	<DataSource>		names the database module to access database tables
	<OperationTypeName>		encloses an operation privilege definition, repeatable
	<BranchTypeName>		encloses a branch privilege definition, repeatable
	<Branch>		encloses a branch definition, repeatable
	<Filter>		encloses filter definition
	<InitialCasePacket>		encloses definitions of case packet “variables”
<Name>	bundle		name of localization resource bundle
<Solution>	<Name>	M	name of the solution
	<Label>		overrides solution name for display on inventory UI
<OperationTypeName>	<Name>	M	name of the operation privilege
	<Description>		description of the operation privilege
<BranchTypeName>	<Name>	M	name of the branch privilege
	<Description>		description of the branch privilege
<Branch>	bundle		name of localization resource bundle
	<Name>	M	name of the branch
	<DefName>		overrides branch name for display in class view
	<Condition>		boolean expression, repeatable; when present, at least one expression must evaluate to “true”, otherwise the branch is not shown in instance view
<DefName>	bundle		name of localization resource bundle
<Condition>	role		condition is true if user has the role given as attribute value
<Branch> (cont’d)	<BranchType>		privilege required for the branch, must be defined by <BranchTypeName>
	<Type>		name for reference to one of several branches used in similar roles in the tree
	<Image>		file name of the branch icon, when unconditional; repeatable with conditions
<Image>	<Condition>		boolean expression; when a condition evaluates to “true”, icon name is taken from <Value>
	<Value>		file name of branch icon
<Branch> (cont’d)	<ReservedImage>		file name of icon shown instead of one defined by <Image>, when the branch instance is reserved

Parent tag	Attribute name/ child element tag	Mandatory/ default value	Description
	<Class-Name>		complete name of Java bean class (including package) for branch resource
	<Find-By>		encloses specification of how to find instances of branch upon expansion of parent branch, repeatable with conditions
<Find-By>	<Condition>		boolean expression, repeatable; when one condition evaluates to “true”, the method (in the same <Find-By>) is selected for execution
	<Method>	M	name of method to call on the branch class to find branch instances, normally a findBy method
	<Key>		repeatable element; each occurrence specifies one parameter for the findBy method, order of <Key> elements must match order of parameters
<Key>	parentKey	false	“true” to indicate that the <Key> (on a method with multiple parameters) identifies the parent instance (if such methods are used in the tree, this attribute must be set in order for advanced search to work correctly)
	type	String	specifies the type of the parameter
<Branch> (cont’d)	<Font>		encloses specifications for the font to display the branch, repeatable with conditions
<Font>	bold	false	when “true”, branch is displayed in bold
	italic	false	when “true”, branch is displayed in italic
	<Condition>		boolean expression, when a condition evaluates to “true”, the <Font> element it belongs to is selected
	<TextColor>		color of text, name or hexadecimal RGB color value
<Branch> (cont’d)	<Label>		label for the branch icon, when unconditional; repeatable with conditions
<Label>	bundle		name of localization resource bundle
	<Condition>		boolean expression; when a condition evaluates to “true”, label is taken from <Value>
	<Value>		actual label value
<Value>	bold	false	when “true”, label is displayed in bold
	italic	false	when “true”, label is displayed in italic
	textColor		color of text, name or hexadecimal RGB color value
	bundle		name of localization resource bundle
<Branch> (cont’d)	<Scroll>		declares children of the branch shall be shown as scrolling list, value is number of children in the list

Parent tag	Attribute name/ child element tag	Mandatory/ default value	Description
	<Parent-Find-By>		specifies a method on parent class and parameters for it to determine parent of orphan
	<Children>		encloses child specifications
<Children>	<Child>		repeatable, name of child branch
<Branch> (cont'd)	<Operation>		repeatable, specifies an operation that can be launched from the branch in instance view
<Operation>	default	"false"	if "true", operation can be launched by clicking the branch
	warning	"false"	if "true", confirmation will be required before operation is launched
	flag		specifies generic operation: "reserve" or "release"
	bundle		name of localization resource bundle
	<Name>	M	name of the operation, displayed in menu and on tab
	<Condition>		boolean expression, repeatable; when present, at least one expression must evaluate to "true", otherwise the operation is disabled (not shown in menu)
	<Image>	M	specifies operation icon, similar to <Image> in <Branch> element
	<Object>		can show branch label as part of operation name
	<OperationType>		privilege required for the operation, must be defined by <OperationTypeName>
	<Action>	M	specification of action to execute operation
<Action>	<Page>	M	Struts action name or JSP file name to be invoked
	<Param>		repeatable, specifies one parameter for Struts action or JSP; not used together with <Workflow>
	<Workflow>		specification of workflow to run, including input parameters and result
<Param>	<Name>		parameter name
	<Value>	M	parameter value
<Workflow>	method	startJob	specifies whether or not to wait for workflow job completion: "startJob" or "startAndWaitForJob"
	<Name>	M	name of workflow to start
	<Label>		name to display in UI
	<Bean>		repeatable, encloses specification of a bean object to retrieve as source for workflow input parameters

Parent tag	Attribute name/ child element tag	Mandatory/ default value	Description
	<WFParam>		repeatable, encloses specification of an input parameter for the workflow, i.e. initialization of a case-packet variable
	<Result>		repeatable, encloses specification of a case-packet variable whose final value is displayed as a result of running the workflow
<Bean>	<Name>	M	name of bean object, used for reference from <WFParam>
	<Class-Name>	M	complete name of Java bean class (including package) for the bean object
	<Find-By>	M	encloses specification of how to find the bean object (possibly multiple instances); not repeatable, no conditions
<WFParam>	source	branch	“branch”: <PValue> interpreted relative to the tree; “bean”: <PValue> interpreted as field of bean object
	dataType	String	type of the parameter, one of: String, int, float, long, double, boolean, Date
	dateFormat		specifies format of field of type Date
	editable	true	false: value not editable when presented to the user before workflow start
	dropdown	false	true: value selectable from dropdown list
	<Name>	M	name of the parameter, i.e. the workflow case-packet variable
	<DisplayName>		displayed name of parameter, if omitted defaults to <Name>
	<Description>		description of parameter, displayed together with name and value
	<BeanName>		when source=”bean”, specifies name of the bean object
	<PValue>		repeatable, specifies value of the parameter according to source
<Result>	<Name>	M	name of workflow case-packet variable to be displayed as result
	<DisplayName>		displayed name of result
<Branch> (cont’d)	<DefOperation>		repeatable, specifies an operation that can be launched from the branch in class view, similar to <Operation>, but cannot be gated by <Conditions>
<DefOperation>	defFlag	extend	specifies generic operation to add user-defined fields
	default	false	true: operation can be launched by clicking the branch

Parent tag	Attribute name/ child element tag	Mandatory/ default value	Description
	warning	false	true: confirmation will be required before operation is launched
	bundle		name of localization resource bundle
	<Name>	M	name of the operation
	<Image>	M	specifies operation icon, similar to <Image> on <Branch> and <Operation> elements, but not repeatable
	<Object>		not used
	<OperationType>		privilege required for the operation, must be defined by <OperationTypeName>
	<Action>	M	specification of action to execute operation
<Filter>	<Field>		defines one variable of the filter
<Field>	variable	M	name of the variable
	label		label to show for the variable
	bundle		name of localization resource bundle, to localize the label
	type	M	type of the variable, can be text, checkbox, select or date
	group	default	name of group the variable belongs to
	<ListOfValues>		defines for a variable of type select
<ListOfValues>	<Value>		repeatable, defines one value
	<BeanName>		alternative to <Value>: full name of a (bean) class to use
	<Attribute>		name of field on the bean class
	<Method>		name of (findBy) method on the class which must return an array of beans
	<ParameterValue>		repeatable, specifies a parameter for the method
<InitialCasePacket>	<VariableValue>		defines one case packet “variable”
<VariableValue>	name		name of the variable
	value		value of the variable, constant:*value* or session:*attribute*

General Properties of InventoryTree

A tree is opened and shown automatically when the Inventory UI is launched, if its default attribute is “true”.

The first child elements of <InventoryTree>: <Name>, <SequenceNumber>, <Solution> and <DataSource> specify some general properties of the tree. Other general properties are privileges, case packet, and filter.

Every tree has a name, defined as the value of the <Name> element. When multiple trees are deployed on the same system, the tree the user wants to view can be selected by name in the tree menus of the inventory UI.

Available trees are listed in the menus ordered first by sequence number of the solution, second by the sequence number of the tree, and finally by the name of the tree. Trees which do not belong to a solution are shown first. The tree sequence number is given as the value of the <SequenceNumber> element, which must be a positive number; default is 0. The solution sequence number is settable by means of a special command of the Inventory Tree Deployer; see chapter 5.

When a tree belongs to a solution, the solution must be identified by means of the <Solution> element. The <Solution> element includes both <Name> and <Label>, the latter being optional. The name of the solution is also shown in the tree menus of the inventory UI as part of the identification of the tree. Here the <Label>, if present, will override the <Name>. To localize the displayed name, use the <Label>.

The inventory UI web server component will need to access the database through a so-called data source, which is actually a pool of database connections to a specific database schema (combination of database instance and database user). Each tree uses a unique data source which can be named by the <DataSource> element. This means you cannot combine data from different databases in a single tree. Normally there is only a single database, and this database is accessed by default if you the <DataSource> element is omitted from the tree definition. See the section “Adding a Data Source for Inventory UI” below for the case where you need a add non-default data source.

Privileges for Trees, Branches and Operations

Privileges that you can use to control which users can view individual trees, view specific branches in the tree and perform specific operations, can be declared, named and associated with individual branches and operations.

For each tree you define a privilege is implicitly defined, which makes the views of the tree selectable in the main menu of the inventory UI. The privilege has the same name as the tree.

The privilege to view and access a certain collection of branches is declared by the <BranchTypeName> element. In this element you provide a name and a description for the privilege. Membership of the collection is declared by the <BranchType> child element of <Branch>.

NOTE — Do not confuse the branch privilege with branch type, defined by the <Type> child element of <Branch>

The privilege to perform a certain set of operations is declared by the <OperationTypeName> element which is similar to a <BranchTypeName> element. Membership of the set is declared by the <OperationType> child element of <Operation>.

Assignment of privileges to user roles can be done by a system administrator from functions in the User Management UI. The descriptions of the privileges are displayed in the forms that are displayed to the system administrator.

For general information about user roles and privileges and a description of the User Management UI refer to chapter 6 of *HP Service Activator, Introduction and Overview*.

Case Packet

With the <InitialCasePacket> element you can declare and assign values to so-called case packet variables, which you can reference as parameter values or in expressions throughout the tree definition. There is no way of setting values other than the initial one to these “variables”, which are actually more like centralled defined constants.

You can define values for variables either as explicit constant values, using the notation constant:*value*, or as user session attributes, using the notation session:*attribute*. In this example, the session attribute refers to the user name that was authenticated for the session:

```
<InitialCasePacket>
   <VariableValue>name=”debuglevel” value=”constant:3”</VariableValue>
   <VariableValue>name=”user” value=”session:user”</VariableValue>
</InitialCasePacket>
```

These variables can then be referenced as `variable:debuglevel` and `variable:user`, respectively.

Filter

The filter, which optionally is part of a tree definition, is another collection of variables in addition to the case-packet. Values for filter variable can be set in a pop-up window from the user interface. The variables can be organized in groups in the window.

The values of the variables may be used in condition expressions and as arguments for <Find-By> methods on branches in order to filter the data shown in the instance tree and leave only the branches which are of interest to the user, hence the name filter (you can do the same with case-packet variables). See the next section ,”Branches”, to understand how to select which branches will appear in the instance tree. You must gate your references to the variables by use of the boolean operand `isFiltered` in a condition. This operand will be true when the filter has been applied to the active tree, otherwise false. Here is a simple example of a boolean (checkbox) filter variable applied in a branch condition:

```
<Condition>!isFiltered || (variable.filter:showWWS == constant:true)</Condition>
```

An instance of the filter, i.e. a set of values for the variables, can be named and saved as a *stored filter*, to be recalled and applied at a later time, by the same user or even a different user. Stored filters can be made available to users by the system administrator; how it is done is described in *HP Service Activator, User’s and Administrator’s Guide*. See that manual also for a description of how to set filter variable values and apply stored filters from the user interface.

NOTE

It is not mandatory that all variables have values when a filter is applied. When you use a filter variable in a branch definition, typically as an argument for a <Find-By> method you may need to gate the usage by a condition to ensure that the variable has a value (!= null) to avoid passing a null value to the method.

With the <Filter> element you can define the set of filter variables for a tree. The following example shows the definition of a filter with one group (g1) and two variables (velocity and color).

```
<Filter>
   <Field variable=”velocity” label=”speed” type=”text” group=”g1”/>
   <Field variable=”color” label=”colour” type=”select” group=”g1”/>
      <ListOfValues>
         <Value>constant:red</Value>
         <Value>constant:green</Value>
      </ListOfValues>
   </Field>
</Filter>
```

Each <Field> element defines a variable in the filter. The variable attribute names the variable. Groups are defined when they are mentioned. The label, if present, is shown instead of the variable name in the pop-up window where values are set for the filter variables. The type can be text, in which case variable values are simply typed in an open field, or select, in which case possible values are defined in a <ListOfValues>, and the actual value is selected from a drop-down list. Values are given with the `constant` prefix as for case packet variables. Alternatively you can

refer to a value already defined for a case-packet variable by using the `prefix` variable and the name of that variable.

You can also specify with an second variant of the <ListOfValues> element that the value must be selected from a list returned from a method of a bean class, using <BeanName> to specify the complete name of the bean class, <Attribute> to specify a field of the bean, <Method> to specify a method of the bean with argument values specified with <Parameter> elements. The method must return an array of occurrences of the bean. From each element of the array is picked the value of the specified field (<Attribute>), the rest is discarded. These values are shown in a drop-down list and one of them must be selected as the value for the filter variable.

Two more types allowed for filter variables are checkbox (boolean with values true and false, but shown to the user as Yes or No in a dropdown list) and date.

For <Parameter> you can use the same forms as for <Value> in the first variant of <ListOfValues>, as described above.

Branches

The bulk of an inventory tree definition file will consist of branch specifications. There will generally be a branch specification for each resource in your inventory, though you may choose not to display every resource. It is also possible to define more than one presentation for each resource, for use in different contexts. You can also define title branches which are not associated with resources, but only serve to make the tree more readable. When a branch is expanded in instance view, each child branch which is a title branch will be shown just once, whereas for a child branch associated with a resource a findBy method will be executed to determine the list of instances of the resource to show.

An important part of a branch definition specifies the operations that can be launched from an occurrence of the branch in instance view. Operations are executed as JSPs or Struts action forms. Definitions of operations are discussed in the section “Operations”.

Parameter Values

Within a branch definition you can specify parameters for findBy methods and for operations (actions, see “Actions”). In a parameter specification you can use the name of a branch to refer to an instance of a resource associated with an occurrence of the named branch in the instance view tree; it is always evaluated for the actual instances that are represented in the tree as it is shown. You can use the name of the branch in which the specification occurs or any of its ancestors in the tree except title branches.

A parameter value can take four forms:

branch	the name of a branch, refers to the value of the primary key of the associated resource instance; commonly used to identify the immediate parent as parameter of a findBy*ParentKey* method and the branch itself for operations such as View, Edit, Delete
branch.field	the name of a branch and the name of one of its field, refers to the value of that field on the assocated resource instance
constant:*value*	a value stated right there
variable:*name*	the value of the case packet variable named *name*, which must be declared and defined with the <InitialCasePacket> element at the outher level of the tree definition
variable.filter:*name*	the value of the filter variable named *name*, which must be declared and defined with the <Filter> element at the outer level of the tree definition

Condition Expressions

Conditions are used in several elements within a <Branch> definition. A condition is a boolean expression that can be evaluated at run-time. The elements that can have conditions are <Branch> (i.e. the condition is directly enclosed by the <Branch> tag), <Image>, <Font>, <Label> and <Find-By>.

Boolean expressions can be composed of operands and operators. Operands can be specified in the following ways:

- a parameter value in one of the forms described above
- *branch*.BeanName representing the resource bean
- a field name
- an SQL expression enclosed within a pair of '@' characters; expressions may include parameters in the forms described above, quoted within pairs of '?' characters
- `isFiltered`, true when a filter has been applied to the active tree, otherwise false

Instead of the branch name, *branch* (in any operand of a boolean expression) can also be a branch type. This will refer to the nearest branch in the lineage (ancestor path) which has a matching <Type> value.

The following operators are allowed:

!	boolean not
||	boolean or
&&	boolean and
==	equals
!=	not equals
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Expressions with more than one operator must include parentheses to determine the order the operands are applied. The types of the operands must match the operators that are applied.

When your expression includes characters, such as '<' or '>' which could be interpreted as XML-delimeters, you must use CDATA sections to escape these characters. A CDATA section is like this : `<![CDATA[` *`your escaped data including <>/ characters`* `]]>`.

Here is a simple example <Condition> (which may not make much sense):

```
<Condition>
   variable:user == Switch.name
</Condition>
```

And here is a condition using an SQL statement (think about the effect of adding this condition to the Switch branch in the tree definition of the DocExample):

```
<Condition>
   @select count(*) from Doc_Switch where rack = '?Rack.name?'@ != constant:0
</Condition>
```

As an alternative to using an expression a condition can be defined by the role attribute of the <Condition> element. The value of the role attribute can be given as a parameter value in one of the forms described above. The value of the attribute is interpreted as the name of a role, and the value of the condition will be true if the user has the role in question. For example:

```
<Condition role="Region.ManagerRole"/>
```

Assuming Region is the current branch or one of its ancestors and associated with a resource instance representing a region of a network, the ManagerRole field could hold the value of the operator role responsible for managing that region.

General Properties of Branch

The following child elements of <Branch> define overall properties of the branch: <Name>, <DefName>, <BranchType>, <Type>, <Class-Name>, <Font>,<Label>, <Scroll>, <Image> and <ReservedImage>.

<Name> - Every branch has a name, defined as the value of the <Name> element. The name is mainly used to refer to the branch, for example as described for parameter values above. In the class view the name is displayed to represent the branch unless it is overridden with the optional <DefName> element.

<BranchType> - Membership of the collection of branches for which a given privilege is required before occurrences can appear when parent branches are expanded is declared by <BranchType> element, whose value must equal the name of the privilege, as defined with <BranchTypeName>. See “Privileges for Trees, Branches and Operations” above.

NOTE <BranchType> refers to a privilege, not to the <Type> of the branch.

<Condition> - Appearance of a branch can be gated by one or more conditions, specified with the <Condition> element. If one condition evaluates to “true”, the branch is shown.

<Type> - Suppose you define a branch which can appear as a child of more than one parent branch in the tree, and you need to make reference from the child branch to a field of the parent resource. To make it possible to name the parent without having to use its branch name, a name that can be common to multiple branches is introduced with the <Type> element. The common name does not need to be declared outside of the branches which it can name.

NOTE The branches that belong to the same type must be similar enough to have the same names for fields that you want to reference using the type name.

<Class-Name> - For a branch to be associated with a resource, the complete Java class name of the resource Java bean must be specified. Omit the <Class-Name> element for a title branch.

<Font> - For each branch you can specify, some attributes of the font with which the branch is displayed are controllable with the <Font> element. Bold and italic font can be specified by setting the Bold and Italic attribute, respectively, “true”. The color of the font can be specified by the value of <TextColor> child element. The value of <TextColor> can be a color name supported the browser or a hexadecimal RGB color value.

The <Font> element can be repeated, with a <Condition> element in each occurrence except the last one. The first element for which the condition evaluates to “true” at the time the branch is displayed, or the last unconditional one, will then be applied.

<Label> - Each branch in the instance view has a label which is displayed to identify the branch and/or its associated instance to the user. The default label for a branch that is associated with a resource instance is the primary key of the instance. The default label for a title branch is the branch name.

You can specify a label different from the default one as the value of the <Label> element. You can use any string, and in the string you can quote a branch instance field value by using the same notation as in parameter values (see “Parameter Values” above), enclosed within $-characters. For example, you can combine the name of the resource, which you give literally, with the quoted name of the instance, like this:

```
<Label>Switch $Switch$</Label>
```

The <Label> element can be repeated, with a <Condition> element in each occurrence except the last one. The first element for which the condition evaluates to “true” at the time the branch is

displayed, or the last unconditional one, will then be applied. If there is no unconditional <Label>, the default value will be the fall-back.

In a conditional <Label> element, you specify the label with an inner <Value> element, like:

```
<Label>
   <Condition>your condition here</Condition>
   <Value>label shown when condition is true</Value>
</Label>
```

<Scroll> - When a branch is expanded in instance view, a huge number of child branches can potentially appear. When a <Scroll> is present (in the parent, the branch which is expanded) only a limited number of child branches will be shown in a window that can be scrolled. The value of the <Scroll> element specifies the size of the scroll window. When a branch has a smaller number of child branches than the size of the scroll window, they are shown with no scrolling adornments.

NOTE If the branch has more than one child branch, there will be a scroll window for each one.

<Image> - The icon that will graphically represent the branch in the presentation tree is specified with the <Image> element. The value must be the name of a file that contains the image. You can use all of the formats supported by the web browser (JPEG, GIF, etc.). File names are relative to *$JBOSS_ACTIVATOR*/images/inventory-gui/tree.

The <Image> element can be repeated, with a <Condition> element in each occurrence except the last one. The first element for which the condition evaluates to "true" at the time the branch is displayed, or the last unconditional one, will then be applied. When a condition is present, the file name must be given in a <Value> element.

Determining Instances of a Branch

When a branch is expanded in the instance view, then all appropriate instances of its child branches must be found and shown. How to find the instances is specified in the defintion of the (child) branch by <Find-By> elements. There can be multiple <Find-By> elements, and each one can be gated by conditions. The method defined in the first <Find-By> element that passes its gate, i.e. if there are conditions then at least one of them evaluates to "true", is executed and returns a list of instances of the child class. Parameters for findBy methods are specified by <Key> elements; here you can use the forms described in the section "Parameter Values" above. If the type of the parameter is not string, it must be specified by the type attribute of the <Key> element.

The most common cases are findAll (no parameters), at the root of a tree or when lineage does not provide information to filter, and findby*ParentKey* (one parameter), when a (child) branch resource has a foreign key that identifies its parent, to retrieve all children of a parent branch.

NOTE The advanced search function works top down in the class tree. For each child branch it assumes the <Find-By> methods relates the instances to the immediate parent in the instance tree. This is normally the case (findby*ParentKey* is used). If it is not the case, advanced search will not work.

NOTE findBy methods are not used for title branches.

If the <Branch> element is also gated by one or more <Condition> elements, each instance (or title branch) must also pass at least one of these conditions before it is added to the instance view tree.

Orphan's Parent

If the definition of a branch (or one of its children) contains a reference to the parent branch, then - when the branch is to be presented (expanded when the occurrence is in a child branch) - it is necessary to know the parent instance. Normally, this is straightforward, because the parent as well as other ancestors are part of the same tree structure. But, if the branch was selected from a list of search results to become the root of an instance tree, then it is an orphan, and there is no immediate way to know the parent.

The resource class of the parent will be known from the parent branch definition in the tree. The <Parent-Find-By> element allows you to specify within the definition of the child branch a method that will be evaluated as if it occurred on the parent branch and provide parameters for it that will retrieve the desired parent instance. Typically the child resource bean will have a foreign key that points to the parent, but a method evaluated in the context of the parent branch cannot have a child field value as a normal parameter, so you will need to resort to a where clause, which can be passed as a parameter to the findAll method. You can find an example of this on the Switch branch in the DocExample.

Child Branches

The branching structure of the tree is defined by declaring child branches for each branch, using the <Children> and <Child> elements. A branch without children is a leaf. Branch definitions may occur in any order, parent can come before or after child.

Operations in Instance View

Most operations are launched from branches in instance view. Each of these operations is defined by an occurrence of the <Operation> element inside the <Branch> element. One operation on a branch can be designated as the default operation, by setting its default attribute “true”. The default operation can be launched by clicking on the branch, other operations must be selected from the right-click menu of the branch. The operation is launched in the context of the selected branch, so field values of the instances associated with the branch and its lineage can be used as parameters.

The standard operations are Create, View, Edit and Delete. These operations are implemented by Struts actions and form beans generated by the Inventory builder. Generic actions and forms can be used to perform Reserve and Release operations on reservable resources and to launch workflow jobs.

General Properties of Operation

<Operation> elements have three attributes: default (see above), warning and flag. If you set “warning” true, a confirmation window pops up before the operation form is shown. The flag must be set for generic operations (reserve, release) to indicate that the action class is generic, not specific to the solution. The operation which has flag=“reserve” is hidden when all units of the resource instance are already reserved, the operation with flag=“release” is hidden when no units are reserved.

The following child elements of <Operation> define overall properties of the operation: <Name>, <Condition>, <Image>, <Object> and <OperationType>.

<Name> and <Object> - The name of an operation is shown in the operations menu, and once the operation has been launched, also in tab for the operation form. It is concatenated from two parts, defined with the <Name> and <Object> elements, respectively. Typically one of the names ‘Create’, ‘View’, ‘Edit’ and ‘Delete’ (as well as ‘Reserve’ and ‘Release’) followed by the name of the resource (bean) are used for the standard operations, but you are free to use different names. <Object> can be used if you want to include the label of the branch in the operation name; to achieve this effect the value specified with <Object> must equal the name of the branch. Any other value specified with <Object> will be taken literally.

<Condition> - The appearance of an operation in the menu can be gated by one or more conditions, specified with the <Condition> element. If one condition evaluates to “true”, the branch is shown, otherwise it is not.

<Image> - The icon that will graphically represent the operation in the menu is specified with the <Image> element. The value must be the name of a file that contains the image. You can use all of the formats supported by the web browser (JPEG, GIF, etc.). File names are relative to `$JBOSS_ACTIVATOR/images/inventory-gui/tree`.

The <Image> element can be repeated, with a <Condition> element in each occurrence except the last one. The first element for which the condition evaluates to “true” at the time the branch is

displayed, or the last unconditional one, will then be applied. When a condition is present, the file name must be given in a <Value> element.

<OperationType> - An operation may require a privilege, specified with the <OperationType> element, and defined with an <OperationTypeName> element of the tree definition. If the user does not have a role to which the privilege has been assigned, the operation will not appear in the menu. See “Privileges for Trees, Branches and Operations” above.

Inventory Actions

The most important part of each operation is the action it performs. The standard inventory operations - Create, View, Edit, Delete - use actions which are executed in the Struts framework as action classes and JSPs that are generated by the Inventory Builder based on resource definitions.

In addition to the standard operations which only affect inventory it is possible to launch a workflow job. This is specified using a different form of the <Action> element, as described in the next section.

Parameters can be specified for actions as (name, value) pairs. Of the actions for the standard operations, Create needs no parameters (everything must be specified by the user in the form), for the other operations you need to specify (field name, field value) for each field which belongs to the primary key, normally only one.

For the standard operations, you need to provide the name of the form action name, including the name of the resource, which enables Struts to run the action, as follows:

Create	CreationForm*Bean*Action.do
View	ViewForm*Bean*Action.do
Update	UpdateForm*Bean*Action.do
Delete	DeleteForm*Bean*Action.do

Reserve and release actions are generic, they don’t need the name of the resource:

Reserve	InstanceReserveAction.do
Release	InstanceReleaseAction.do

You use the <Action> element with its enclosed elements <Page>, <Param>, <Name> and <Value> to provide the form action name (with <Page>) and parameter names and values. An example from the DocExample is shown here, note that ‘Switch’ in <Value> is the name of the branch, which refers to the value of the primary key of the associated instance:

```
<Action>
   <Page>UpdateFormSwitchAction.do</Page>
   <Param>
      <Name>name</Name>
      <Value>Switch</Value>
   </Param>
</Action>
```

Workflow Actions

In addition to the inventory actions which work directly on the inventory it is also possible as an operation associated with a branch in the inventory tree to launch a workflow job. This feature comes in two flavors, selectable by the method attribute of the <Workflow> element:

- method=”startJob”: start the workflow job without waiting for completion, display the job id
- method=”startAndWaitForJob”: wait for the job to finish and display selected case packet variables as results

In an <Operation> element used to invoke a workflow the value of the <Page> element must be StartWorkflow.do; the <Param> element is not used; and the details of the workflow are all specified by the <Workflow> element: its name and case-packet variables to be initialized as input parameters and retrieved for display as output parameters.

Input parameter case-packet variables and values to initialize their values are specified using <WFParam> child elements of the <Workflow> element. The data type of each value can be specified using the dataType attribute, which can take the values String (default), int, float, long, double, boolean and Data. It should match the data type declared for the case-packet variable in the workflow. The name of the case-packet variable is specified with the <Name> child element.

To allow flexibility in the specification of input parameter values several different sources can be used:

- <WFParam> element with source="branch" (default): field values from resource instances associated with branches in the inventory tree, either the branch that the workflow operation belongs to or a branch in its lineage, using one of the forms *branch* or *branch.field*, as described under "Parameter Values" in the "Branches" section. The other forms specified under "Parameter Values", i.e. constants and case-packet variables of the tree, can also be used.
- <WFParam> element with source="bean": field values from resource bean objects that are retrieved specifically for this purpose, independently of the inventory tree, as specified by <Bean> child elements of the <Workflow> element. The reference from the <WFParam> element to the bean object is made by the value of the <BeanName> child element which must equal the name of the bean object. The name of the field to extract is specified with the <PValue> child element.
- User input. Before the values from <WFParam> elements are passed to the workflow to be started, they are shown to the user and can be edited. Parameters for which no source value has been specified (no <PValue> child element in the <WFParam> element) will appear as empty input fields. A <WFParam> element may specify more than one possible value, and the user can then select one of the options from a dropdown list, provided the dropdown attribute on the <WFParam> element is "true". When source="branch", multiple values can be specified with multiple occurrences of the <PValue> child element. When source="bean", multiple values may occur when a non-unique findBy method is used.

Before a bean object can be used in a <WFParam> element, it must be defined with a <Bean> element. The bean is named and can be referenced in <WFParam> elements with the value of the <Name> child element. The class of the bean is specified as the value of the <Class> child element, and the findBy method with its key field parameters is specified by a <Find-By> child element, which has the same form as when it is used to determine occurrences of a branch, except that <Condition> cannot be used (see "Determing Instances of a Branch" under "Branches").

Both <Workflow> and <WFParam> have a child element <Name>, defining the name of the workflow and case-packet variable, respectively. These names will also be used on the UI, unless overridden with values specified by a <DisplayName> element.

The <WFParam> element has attributes dateFormat and editable, to control how a value of type Date is shown and whether the value can be edited by the user (editable="true" by default). The format of values of type Data behaves in the same way as for resource fields, see the section "<Field> dateFormat", under "The <Field> Element" in chapter 2.

When a workflow is run with method="startAndWaitForJob", final values of case-packet variables can be shown as results. The case-packet variables to be shown are specified with <Result> child elements of <Workflow>. The name (possibly overridden with <DisplayName>) of each result case-packet variables is specified with the <Name> child element of <Result>.

A complete example of an operation that can be included in the inventory definition for the DocExample is shown below. A workflow named DocExampleWF will be executed. Three case packet-variables of the workflow: a, b and c, will be initialized as input parameter. Two other case-packet varaiables: d and e, will be retrieved when the workflow has finished and shown as results.

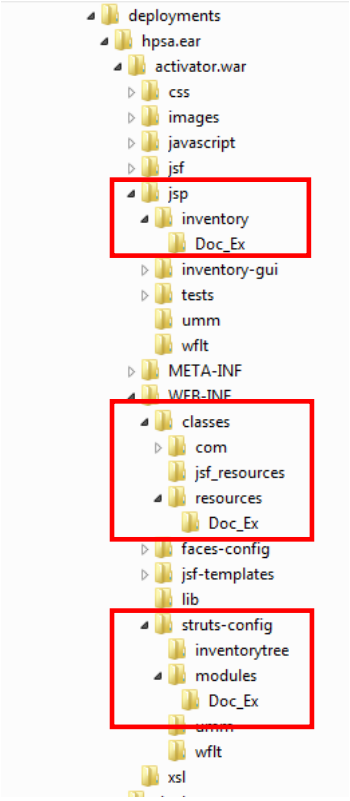
The value for a will come from the Switch branch, the value for b will be selected from a dropdown list among the names of all switch resource instances, and the value for c will come from user input.

```
<Operation>
  <Name>Run Workflow</Name>
  <Image>conmutar.gif</Image>
  <Action>
    <Page>/activator/inventory/startWorkflow.do</Page>
    <Workflow method="startAndWaitForJob">
      <Name>DocExampleWF</Name>

      <Bean>
        <Name>Switch</Name>
        <Class>com.hp.ov.activator.docExample.Switch</Class>
        <Find-By>
          <Method>findAll</Method>
        </Find-By>
      </Bean>

      <WFParam>
        <Name>a</Name>
        <PValue>Switch.name</PValue>
      </WFParam>

      <WFParam source="bean" dropdown="true">
        <Name>b</Name>
        <BeanName>Switch</BeanName>
        <PValue>name</PValue>
      </WFParam>

      <WFParam>
        <Name>c</Name>
      </WFParam>

      <Result>
        <Name>d</Name>
      </Result>

      <Result>
        <Name>e</Name>
      </Result>
    </Workflow>
  </Action>
</Operation>
```

Customizing and Adding Own Operations

If you want to change the action forms and JSPs generated by the Inventory Builder, you will first need to locate the generated files. See chapter 3 for this information. All the Java code needed for the actions of the different standard operations is in action classes. The Java code for validation of data entered in forms is in form validation classes. The screen forms are in JSPs. In the running system, after deployment, all these files are located under *`$JBOSS_ACTIVATOR`*, as shown in Figure 4-2 where the Doc_Example is deployed as a solution. JSPs are under `jsp/inventory`, action classes under `WEB-INF/classes/com`, localizable resources under `WEB-INF/classes/resources`, and the Struts configuration which ties everything together is under `struts-config/modules`.

Figure 4-2 Structure of Deployed Struts Action Classes and JSPs



If you add your own non-standard operations, you can choose to develop them as classic JSPs with embedded Java code, or use a Struts approach.

If you write classic JSPs (not integrated into the Struts structure) and place them in the `jsp/inventory/<your solution>` directory, you can refer to them from <Page> elements within <Operation> <Action> simply by stating the file names.

NOTE If you need to make cross-references between JSPs (for example, to execute another JSP after a Submit button is pressed), you need to state a full path name like `/activator/jsp/inventory/<your solution>/<your jsp>`

If you use the Struts approach you will need to provide one or more Struts configuration files with Struts action specifications and/or modify the generated one for the Struts module that is generated for the solution. If you create additional files, you must deploy them in the `WEB-INF/struts-config/modules/<your solution>` directory, alongside the file containing specifications generated by the Inventory Builder.

You can use the Inventory Builder in combination with the Deployment Manager to manage the files you change or add in a solution hierarchy outside of the running system, to compile any Java files and to deploy the resulting files into the runtime structure. For pure additions, you can create a structure similar to the necessary part of the hierarchy under activator.war and place it under `$ACTIVATOR_OPT/solutions/<your solution>/UI`. For example, if you write classic JSPs, it will just be your JSPs in `jsp/inventory/<your solution>` (in Figure 4-2 the solution is Doc_Example).

Operations in Class View

The main operations that can be performed in the class view are the search operations, simple and advanced search. They are available for all classes, cannot be enabled, disabled or customized.

In addition to searches, the operation to add user defined fields to a resource is also available in the class view, but only as specified by a <DefOperation> element, just like the instance operations need to be specified with <Operation>.

The way to specify general properties of operations in class view is the same as for operations in instance view, except you use child elements of <DefOperation> instead of <Operation>. There is one exception: these operations cannot be gated by conditions.

Actions for operations in class view are also specified with <Action> elements in exactly the same way as for operations in instance view.

The <Page> value to specify for the operation to add user fields is ext_attribute_class.do?beanClassName=*complete name of bean Java class*. This action takes no parameters. The defFlag attribute on the <DefOperation> element must have the value “extend”, like this example:

```
<DefOperation defFlag="extend"'>
    <Name>Add fields</Name>
    <Image>edit.gif</Image>
    <Action>
        <Page>ext_attribute_class.do?beanClassName=com.hp.sa.inventory.AccessDe
vice</Page>
    </Action>
</DefOperation>
```

You can also implement your own actions, as JSPs with or without Struts classes, and make them available as operations, just as for the instance view.

Adding a Data Source for Inventory UI

To enable the inventory UI web server function to access database tables which do not belong to the database user and the database instance that was specified at Service Activator installation time, an additional data source will be needed.

See *HP Service Activator, User’s and Administrator’s Guide* for information about how to configure the data source itself. We assume here it is named *DSNAME*. In the tree definition for the inventory UI, you must specify the name as the value of the <DataSource> element. Additionally, to make the data source available for the inventory UI, you must add corresponding elements that refer to it in two files in the directory *$JBOSS_ACTIVATOR*\WEB-INF.

In “jboss-web.xml” add:

```
<resource-ref>
    <res-ref-name>jdbc/DSNAME</res-ref-name>
    <jndi-name>java:/hpsa/jdbc/DSNAME</jndi-name>
</resource-ref>
```

In “web.xml” add (to the other similar data sources):

```
<resource-ref>
    <res-ref-name>jdbc/DSNAME</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

To make the changes take effect, restart Service Activator.

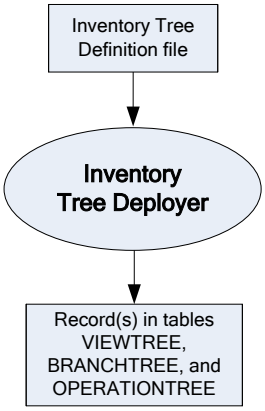
5 Inventory Tree Deployer

This chapter describes how to use the Inventory Tree Deployer to process tree definition files and deploy them to a running system. The Inventory Tree Deployer is actually the same program as the Inventory Tree Designer described in chapter 4, just called from a command line.

The servlet supporting inventory UI builds its run-time representation of the trees that are displayed in client browsers from a database table representation of the tree definitions in Service Activator's static repository. It regularly updates the run-time representation from the repository, so it is not necessary to restart Service Activator after deploying one or more tree definitions. The update frequency can be configured as parameter reload-time in the UI configuration file *$JBOSS_ACTIVATOR*/WEB-INF/web.xml.

The essential function of the Inventory Tree Deployer, the deploy function, is to analyze tree definitions, translate them to the tabular representation and write it into the repository, as shown in Figure 5-1. Information about privilges for branches and operations is written into tables not shown in this figure. It also provides complementary functions for managing inventory trees: list known trees by tree name and solution name, delete a tree from the repository, extract a tree definition from the repository to XML format.

Figure 5-1 Inventory Tree Deployer



The command line syntax is:

```
InventoryTreeDesigner <function> <DB options> <other options> <tree file>*
```

The <function> part identifies the requested function and function-specific options. The other parts are similar for all functions. The common parts are described first, followed by a small section for each function.

NOTE Options are not case-sensitive. Value containing space(s) must be surrounded by double-quotes (").

Table 5-1 DB Options

Option	Description
-dbHost <DBHOST>	optional, name of the database host
-dbName <DBSID>	optional, name of database instance
-dbPort <DBPORT>	optional, port where database is accessed
-dbUser <DBUSER>	mandatory, name of database user
-dbPassword <DBPASSWORD>	mandatory, password for database user

DB options can get values from configuration files, in the same way as for the Inventory Builder. See chapter 3 for details.

Table 5-2 Other Options

Option	Description
-verbose	causes Inventory Tree Deployer to output verbose progress information including stack trace if an error occurs
-help	outputs list of valid options and syntax information
-version	outputs the version of the product

Deploy Trees

In the command to deploy one or more trees the <function> part has this form:

```
-deployTrees
```

It is possible to specify the name of a single tree definition file or the name of a directory; in the latter case all files with .xml suffix in the directory are processed as tree definitions. The file names are not stored in the repository. Tree definitions are identified by the tree name and, optionally, solution name that are defined by the respective elements of the tree definition.

List Trees

In the command to list trees found in the repository the <function> part has this form:

```
-listTrees
```

No names of tree definition files can be entered for this function.

Delete Tree

In the command to delete a tree from the repository the <function> part has this form:

```
-deleteTree [-force] [-solution <SOLUTION>] -treeName <TREENAME>
```

No names of tree definition files can be entered for this function.

The -force option must be present if the specified tree contains definitions of privileges for branches or operations.

The <SOLUTION> and <TREENAME> values are the names whereby the tree is identified in the repository. They originate from the respective elements of the tree definition.

Extract Tree

In the command to extract a tree from the repository to XML format the <function> part has this form:

```
-downloadTree [-solution <SOLUTION>] -treeName <TREENAME>
```

One file name must be given to name the target file. The options identify the tree definition in the repository in the same way as for -deleteTree.

Set Solution Sequence Number

The sequence number of a solution, which determines the order in which trees are shown in the tree menus of the inventory UI, is controlled by using a <function> part with the keyword -setSolutionSequence:

```
-setSolutionSequence <number> -solution <SOLUTION>
```

The number must be greater than 0. The command will apply to all trees which belong to the specified solution.

6 Localizing Inventory

You may want to customize the labels, strings, error messages, etc., which appear on the inventory UI, in a language different from English.

For the tree definitions which control a major part of the UI, all of these strings can be localized using resource bundles as described in the section “Localization” in the beginning of chapter 4. In resource definition files they are defined as values of XML elements. The elements (tags) that are suitable for localization are listed in the beginning of chapter 2 (for resource definitions) .

Some additional items can be localized in the properties files that are generated by the Inventory Builder in step 1 for use by the inventory UI servlet (Struts). When deployed, the properties files for the resources belonging to your solution will be placed in the directory *$JBOSS_ACTIVATOR*/WEB-INF/classes/resources/<*solution-name*> (see Figure 4-2).

You can prepare a localized version of each properties file with an editor. Then, using the Java utility native2ascii, you should convert the file to an ASCII encoding and add the identification of the local language to the file name, like this:

```
>native2ascii example.properties example_zh_CN.properties
```

When you have localized all desired property files, you can copy them to deployment directory. See “Customizing and Adding Own Operations” in chapter 4 for a discussion of how to arrange files for a customized inventory UI for use with the DeploymentManager.

You may also want to localize the built-in properties of the inventory UI. They are found (already deployed) in *$JBOSS_ACTIVATOR*/WEB-INF/classes/InventoryResources.properties.