# Packaging and Delivering Software With the Image Packaging System in Oracle® Solaris 11.3

**Part No: E54820**
November 2016

Packaging and Delivering Software With the Image Packaging System in Oracle Solaris 11.3

**Part No: E54820**

**Documentation Accessibility**

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

**Access to Oracle Support**

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

**Référence: E54820**

**Accessibilité de la documentation**

Pour plus d’informations sur l’engagement d’Oracle pour l’accessibilité à la documentation, visitez le site Web Oracle Accessibility Program, à l’adresse `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

**Accès aux services de support Oracle**

Les clients Oracle qui ont souscrit un contrat de support ont accès au support électronique via My Oracle Support. Pour plus d’informations, visitez le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` ou le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` si vous êtes malentendant.

# Contents

# Using This Documentation

- **Overview** – Describes how to use the Oracle Solaris Image Packaging System (IPS) feature to create software packages for the Oracle Solaris 11 operating system (OS)
- **Audience** – Software developers who want to create packages that can be installed on the Oracle Solaris 11 OS and maintained using IPS, and developers and system administrators who want to better understand IPS and how the Oracle Solaris OS is packaged using IPS
- **Required knowledge** – Experience with IPS and with the Oracle Solaris Service Management Facility (SMF) feature

## Product Documentation Library

Documentation and resources for this product and related products are available at `http://www.oracle.com/pls/topic/lookup?ctx=E53394`.

## Feedback

Provide feedback about this documentation at `http://www.oracle.com/goto/docfeedback`.

♦ ♦ ♦ CHAPTER 1

# IPS Design Goals, Concepts, and Terminology

This guide explains how to create packages that can be installed on the Oracle Solaris 11 OS and maintained using IPS, and how the Oracle Solaris OS is packaged using IPS.

This chapter discusses the following topics:

- IPS design concepts, to help you understand and use the more advanced features of IPS
- Software self-assembly: the ability of installed software to build itself into a working configuration
- Phases of the IPS package lifecycle
- Concepts such as installable image, package publisher, and package actions

## IPS Design Goals

IPS is designed to eliminate some long-standing issues with previous software distribution, installation, and maintenance mechanisms that have caused significant problems for Oracle Solaris customers, developers, maintainers, and ISVs.

Principle IPS design goals include:

**Minimize downtime.**

Minimize planned downtime by making software update possible while systems are in production.

Minimize unplanned downtime by supporting quick reboot to known working software configurations.

**Automate installation and update.**

Automate, as much as possible, the installation of new software and updates to existing software.

**Reduce media requirement.**

Resolve the difficulties with ever-increasing software size and limited distribution media space.

**Verify correct software installation.**

Ensure that it is possible to determine whether a package is correctly installed as defined by the author (publisher) of the package. Such a check should not be spoofable.

**Enable easy virtualization.**

Incorporate mechanisms to allow for the easy virtualization of Oracle Solaris at a variety of levels, in particular using zones.

**Simplify upgrade.**

Reduce the effort required to generate patches or upgrades for existing systems.

**Enable easy package creation.**

Enable other software publishers (ISVs and end-users themselves) to easily create and publish packages for Oracle Solaris.

These goals led to the following ideas:

**Create boot environments as needed.**

Leverage ZFS snapshot and clone facilities to dynamically create boot environments on an as-needed basis.

- Since Oracle Solaris 11 requires ZFS as the root file system, zone file systems need to be on ZFS as well.
- Users can create as many boot environments as desired.
- IPS can automatically create boot environments on an as-needed basis, either for backup purposes prior to modifying the running system, or for installation of a new version of the OS.

**Unify installation, patch, and update.**

Eliminate duplicated mechanisms and code used to install, patch, and update.

This idea results in several significant changes to the way Oracle Solaris is maintained, including the following important examples:

- All OS software updates and patching are done directly with IPS.
- Any time a new package is installed, it is already exactly at the correct version.

**Minimize opportunities to install incorrectly.**

The requirement for unspoofable verification of package installation has the following consequences:

- If a package needs to support installation in multiple ways, those ways must be specified by the developer so that the verification process can take this into account.
- Scripting is inherently unverifiable since the packaging system cannot determine the intent of the script writer. This, along with other issues discussed later, led to the elimination of scripting during packaging operations.
- A package cannot have any mechanism to edit its own manifest, since verification is then impossible.

- If the administrator wants to install a package in a manner incompatible with the original publisher's definition, the packaging system should enable the administrator to easily republish the package he wants to alter so that the scope of his changes is clear, not lost across upgrades, and can be verified in the same manner as the original package.

**Provide software repositories.**

The need to avoid size restrictions led to a software repository model, accessed using several different methods. Different repository sources can be composited to provide a complete set of packages, and repositories can be distributed as a single file. In this manner, no single media is ever required to contain all the available software. To support disconnected or firewalled operations, tools are provided to copy and merge repositories.

**Include metadata as part of the software package.**

The desire to enable multiple (possibly competing) software publishers led to the decision to store all the packaging metadata in the packages themselves: No master database exists for information such as all packages and dependencies. A catalog of available packages from a software publisher is part of the repository for performance reasons, but the catalog can also be regenerated from the data contained in the packages.

# Software Self-Assembly

Given the goals and ideas described above, IPS introduces the general concept of software self-assembly: Any collection of installed software on a system should be able to build itself into a working configuration when that system is booted, by the time the packaging operation completes, or at software runtime.

Software self-assembly eliminates the need for install-time scripting in IPS. The software is responsible for its own configuration, rather than relying on the packaging system to perform that configuration on behalf of the software. Software self-assembly also enables the packaging system to safely operate on alternate images, such as boot environments that are not currently booted, or offline zone roots. In addition, since the self-assembly is performed only on the running image, the package developer does not need to cope with cross-version or cross-architecture runtime contexts.

Some operating system image preparation must be done before boot, and IPS manages this transparently. Image preparation includes updating boot blocks, preparing a boot archive (`ramdisk`), and, on some architectures, managing the menu of boot choices.

## Tools for Software Self-Assembly

The following IPS features and characteristics facilitate software self-assembly.

## Atomic Software Objects

An *action* is the atomic unit of software delivery in IPS. Each action delivers a single software object. That software object can be a file system object such as a file, directory, or link, or a more complex software construct such as a user, group, or device driver. In the SVR4 packaging system, these more complex action types are handled by using class action scripts. In IPS, no scripting is required.

Actions, grouped together into packages, can be installed, updated, and removed from both live images and offline images. A live image is the image mounted at `/` of the active, running boot environment of the current zone.

Actions are discussed in more detail in "Package Content: Actions" on page 21.

## Configuration Composition

Rather than using scripting to update configuration files during packaging operations, IPS encourages delivering fragments of configuration files. The fragments can be used in the following ways:

- The packaged application can be written to be aware of the file fragments. The application can access the configuration file fragments directly when reading its configuration, or the application can assemble the fragments into the complete configuration file before reading the file.
- An SMF service can reassemble the configuration file whenever fragments of the configuration are installed, removed, or updated.

See "Assembling a Custom File from Fragment Files" on page 91 for an example of creating a package that delivers a service that assembles configuration files.

## Actuators and SMF Services

An *actuator* is a tag applied to any action delivered by the packaging system that causes a system change when that action is installed, removed, or updated. These changes are typically implemented as SMF services. See Chapter 7, "Automating System Change as Part of Package Installation" for more information about actuators.

SMF services can configure software directly, or SMF services can construct configuration files using data delivered in the SMF manifest or in files installed on the system.

SMF services have a rich syntax to express dependencies. Each service runs only when all of its required dependencies have been satisfied. See *Managing System Services in Oracle Solaris 11.3* for more information about SMF services.

Any service can add itself as a dependency on the svc:/milestone/self-assembly-complete:default SMF milestone. Once the booting operating system has reached this milestone, all self-assembly operations should be completed.

A special type of zone called an *Immutable Zone* is a zone that can be configured to have restricted write access to portions of its file system. See the discussion of file-mac-profile in the zonecfg(1M) man page. To complete self-assembly in this type of zone, boot the zone read/write, as described in "Options for Booting a Read-Only Zone With a Writable Root File System" in *Creating and Using Oracle Solaris Zones*. After the self-assembly-complete SMF milestone is online, the zone is automatically booted to the required file-mac-profile setting.

# Examples of Software Self-Assembly in Oracle Solaris

The following examples describe packages that are delivered as part of Oracle Solaris.

## Apache Web Server Configuration

The Oracle Solaris package for Apache Web Server, pkg:/web/server/apache-22, delivers an httpd.conf file that contains the following Include directive referencing configuration files in the /etc/apache2/2.2/conf.d directory:

```
Include /etc/apache2/2.2/conf.d/*.conf
```

To apply custom configuration, you can create one or more packages that deliver custom .conf files to that conf.d directory and use a refresh_fmri actuator to automatically refresh the Apache instance whenever a package that delivers a new .conf file is installed, updated, or removed.

```
file etc/apache2/2.2/conf.d/custom.conf path=etc/apache2/2.2/conf.d/custom.conf \
    owner=root group=bin mode=0644 refresh_fmri=svc:/network/http:apache22
```

See "Add Any Facets or Actuators That Are Needed" on page 47 and Chapter 7, "Automating System Change as Part of Package Installation" for information about how to use the refresh_fmri actuator.

Refreshing the Apache service instance causes the web server to rebuild its configuration. To verify this, use the following command to show the name of the method that runs when the Apache service instance is refreshed:

```
$ svcprop -p refresh/exec http:apache22
/lib/svc/method/http-apache22\ refresh
```

A look at the method shows that refreshing the http:apache22 instance restarts the Apache httpd daemon by invoking apachectl with the graceful command.

### User Attributes Configuration

User attributes are configured in `/etc/user_attr` and in additional configuration files in `/etc/user_attr.d`.

The `/etc/user_attr` configuration file is used to configure extended attributes for roles and users on the system. In Oracle Solaris 11, the `/etc/user_attr` file is used for local changes only. Complete configuration is read from the separate files delivered into the `/etc/user_attr.d` directory. Multiple packages deliver fragments of the complete configuration. For example, `/etc/user_attr.d/core-os` is delivered by the `system/core-os` package, and `/etc/user_attr.d/ikev2-daemon` is delivered by the `system/network/ike` package.

No services are restarted or refreshed as a result of installing these configuration files. No scripting is needed when these files are installed, uninstalled, or updated. The files in `/etc/user_attr.d` are composed by the name service cache daemon, `nscd`. The behavior of the `nscd` daemon is managed by the `svc:/system/name-service/cache` service.

```
$ svcs -p cache
STATE          STIME    FMRI
online         15:54:24 svc:/system/name-service/cache:default
               15:54:24    100698 nscd
```

The name service cache daemon provides configuration composition for most name service requests in the same way as described for `user_attr`. See the `nscd`(1M) man page.

### Security Configuration

The `/etc/security/exec_attr.d/` directory stores security configuration files.

In earlier Oracle Solaris releases, an SMF service merged the files delivered in `exec_attr.d` into a single database, `/etc/security/exec_attr`. In Oracle Solaris 11, functions in the security attributes database library, `libsecdb`, read the fragments in `exec_attr.d` directly, eliminating the need for a service to perform the merge.

Other directories in `/etc/security` that contain fragments of configuration, such as `auth_attr.d` and `prof_attr.d`, are handled in a similar way.

## IPS Package Lifecycle

This section provides high-level descriptions of each state in the IPS package lifecycle. For best results, both package developers and system administrators should understand the various phases of the package lifecycle.

**Create**

Packages can be created by anyone. IPS does not impose any particular software build system or directory hierarchy on package authors. For details about package creation, see Chapter 2, “Packaging Software With IPS”. Aspects of package creation are discussed throughout the remaining chapters of this guide.

**Publish**

Packages are published to an IPS repository, either to an HTTP location or to the file system. A published package can also be converted to a `.p5p` package archive file. To access software from an IPS repository, the repository can be added to the system using the `pkg set-publisher` command, or the repository can be accessed as a temporary source by using the `-g` option with `pkg` commands. Examples of package publication are shown in Chapter 2, “Packaging Software With IPS”.

**Install**

Packages can be installed on a system, either from an IPS repository accessed over `http://`, `https://`, or `file://` URLs, or from a `.p5p` package archive. Package installation is described in more detail in Chapter 3, “Installing, Removing, and Updating Software Packages”.

**Update**

Updated versions of packages might become available, either published to an IPS repository, or delivered as a new `.p5p` package archive. Installed packages can then be brought up to date, either individually, or as part of an entire system update.

Note that IPS does not use the same concept of “patching” that the SVR4 packaging system did. All changes to IPS packaged software are delivered by updated packages.

Package updates are performed in much the same way as package installations, but the packaging system is optimized to install only the changed portions delivered by an updated package. Package updating is described in more detail in Chapter 3, “Installing, Removing, and Updating Software Packages”.

**Rename**

During the life of a package, you might want to rename the package. A package might be renamed for organizational reasons or to refactor packages. Examples of package refactoring include combining several packages into a single package or breaking a single package into multiple smaller packages.

IPS gracefully handles content that moves between packages. IPS also allows old package names to persist on the system, automatically installing the new packages when a user asks to install a renamed package. Package renaming is described in more detail in “Renaming, Merging, and Splitting Packages” on page 96.

**Obsolete**

Eventually a package might reach the end of its life. A package publisher might decide that a package will no longer be supported, and that it will not have any more updates made available. IPS allows publishers to mark such packages as obsolete.

Obsolete packages can no longer be used as a target for most dependencies from other packages, and any packages upgraded to an obsolete version are automatically removed from the system. Package obsoletion is described in more detail in "Renaming, Merging, and Splitting Packages" on page 96.

**Remove** Finally, a package can be removed from the system if no other packages have dependencies on it. Package removal is described in more detail in Chapter 3, "Installing, Removing, and Updating Software Packages".

# IPS Terminology and Components

This section defines IPS terms and describes IPS components.

## Installable Image

IPS is designed to install packages in an image. An image is a directory tree, and can be mounted in a variety of locations as needed. An image is one of the following types:

Full In a full image, all dependencies are resolved within the image itself, and IPS maintains the dependencies in a consistent manner.

Zone Non-global zone images are linked to a full image (the parent global zone image), but do not provide a complete system on their own. In a zone image, IPS maintains the non-global zone consistent with its global zone as defined by dependencies in the packages.

User User images contain only relocatable packages.

Images are created or cloned by installers, by the `beadm` and `zonecfg` commands, and by the `pkg` command with options such as `--be-name`.

## Package Identifier: FMRI

Each package is represented by a Fault Management Resource Identifier (FMRI). The full FMRI for a package consists of the scheme, a publisher, the package name, and a version string in the following format:

*scheme*`://`*publisher*`/`*name*`@`*version*

The scheme for every IPS package FMRI is `pkg`. In the following example package FMRI for the `suri` storage library, `solaris` is the publisher, `system/library/storage/suri` is the package name, and `0.5.11,5.11-0.175.3.0.0.19.0:20150329T164922Z` is the version:

```
pkg://solaris/system/library/storage/suri@0.5.11,5.11-0.175.3.0.0.19.0:20150329T164922Z
```

FMRIs can be specified in abbreviated form if the resulting FMRI is still unique. The scheme, publisher, and version can be omitted. Leading components can be omitted from the package name.

- When the FMRI starts with `pkg://` or `//`, the first word following `//` must be the publisher name, and no components can be omitted from the package name. When no components are omitted from the package name, the package name is considered complete, or *rooted*.
- When the FMRI starts with `pkg:/` or `/`, the first word following the slash is the package name, and no components can be omitted from the package name. No publisher name can be present.
- When the version is omitted, the package generally resolves to the latest version of the package that can be installed.

## Package Publisher

A publisher is an entity that develops and constructs packages. A publisher name, or prefix, identifies this source in a unique manner. Publisher names can include upper and lower case letters, numbers, hyphens, and periods: the same characters as a valid host name. Internet domain names or registered trademarks are good choices for publisher names, since these provide natural namespace partitioning.

`pkg` clients combine all specified sources of packages for a given publisher when computing packaging solutions.

## Package Name

Package names are hierarchical with an arbitrary number of components separated by forward slash (`/`) characters. Package name components must start with a letter or number, and can include underscores (`_`), hyphens (`-`), periods (`.`), and plus signs (`+`). Package name components are case sensitive.

Leading components of package names can be omitted if the package name that is used is unique. For instance, `/driver/network/ethernet/e1000g` can be reduced to `network/ethernet/e1000g`, `ethernet/e1000g`, or even simply `e1000g`. FMRIs can also be specified using an asterisk (`*`) to match any portion of a package name. Thus `/driver/*/e1000g` and `/dri*00g` both expand to `/driver/network/ethernet/e1000g`.

Package names should be chosen to reduce ambiguities as much as possible. Package names form a single namespace across publishers. Packages with the same name and version but different publishers are assumed to be interchangeable in terms of external dependencies and interfaces. See "Avoiding Conflicting Package Content" on page 95 for a discussion of package names and dependencies.

When a package name starts with `pkg:/`, `/`, `pkg://`*publisher*`/`, or `//`*publisher*`/`, the package name is considered to be complete, or *rooted*. If the `pkg` client complains about ambiguous package names, specify more components of the package name or specify the full, rooted name.

If an FMRI contains a publisher name, then the full, rooted package name must be specified.

Scripts should refer to packages by their full, rooted names, although the publisher can be omitted.

## Package Version

The package version has the following four parts:

*component_version*`,`*release*`-`*branch_version*`:`*time_stamp*

The component version, release, and branch version can be arbitrarily long and must consist of only integers and period characters (.). A sequence of more than one integer cannot begin with a zero (0). See "Construct an Appropriate Package Version String" on page 44 for help converting your product version to an IPS package version.

The time stamp has the following parts. The date and time must consist of only integers.

*date*`T`*time*`Z`

The component version and release are separated by a comma (,). The release and branch version are separated by a hyphen (-). The branch version and time stamp are separated by a colon (:).

The following example package version is described below:

```
0.5.11,5.11-0.175.3.0.0.19.0:20150329T164922Z
```

Component version: `0.5.11`

For components tightly bound to the operating system, the component version usually includes the value of `uname -r` for that version of the operating system. For a component with its own development lifecycle, the component version is a dotted release number, such as 2.4.10.

Release: `5.11`

The release, if present, must follow a comma (,). Oracle Solaris uses this sequence to specify the release of the OS for which the package was compiled.

Branch version: `0.175.3.0.0.19.0`

The branch version, if present, must follow a hyphen (-). The branch version provides vendor-specific information. This sequence can contain a build number or provide some other information. This value can be incremented when the packaging metadata is changed, independently of the component. See "Oracle Solaris Package Versioning" on page 141 for a description of how the branch version fields are used in Oracle Solaris.

Time stamp: `20150329T164922Z`

The time stamp, if present, must follow a colon (:). The time stamp is the time the package was published in ISO-8601 basic format: *YYYYMMDD*`T`*HHMMSS*`Z`. The time stamp is automatically updated when the package is published.

The package versions are ordered using left-to-right precedence: The number immediately after the `@` is the most significant part of the version space. The time stamp is the least significant part of the version space.

The `pkg.human-version` attribute can be used to provide a human-readable version string. The value of the `pkg.human-version` attribute can be provided in addition to the package version described above for the package FMRI but cannot replace the package FMRI version. The human-readable version string is used only for display purposes. See "Set Actions" on page 29 for more information.

By allowing arbitrary version lengths, IPS can accommodate a variety of different models for supporting software. For example, a package author can use the build or branch versions and assign one portion of the versioning scheme to security updates, another for paid versus unpaid support updates, another for minor bug fixes, or whatever information is needed.

A version can also be the token `latest`, which specifies the latest version known.

Appendix B, "How IPS Is Used To Package the Oracle Solaris OS" describes how Oracle Solaris implements versioning.

## Package Content: Actions

Actions define the software that comprises a package; they define the data needed to create this software component. Package contents are expressed in a package manifest file as a set of actions.

Package manifests are largely created using programs. Package developers provide necessary information about the object to be installed, and the manifest is completed using package development tools as described in Chapter 2, "Packaging Software With IPS".

Actions are expressed in the following form in package manifest files:

*action_name attribute1=value1 attribute2=value2 . . .*

In the following example action, `dir` indicates this action specifies a directory. Attributes in the form *name*=*value* describe properties of that directory:

```
dir path=a/b/c group=sys mode=0755 owner=root
```

Action metadata is freely extensible. Additional attributes can be added to actions as needed. Attribute names cannot include spaces, quotation marks, or equals signs (=). Attribute values can have all of those, although values with spaces must be enclosed in single or double quotation marks. Single quotation marks need not be escaped inside a string enclosed in double quotation marks, and double quotation marks need not be escaped inside a string enclosed in single quotation marks. A quotation mark can be prefixed with a backslash character (\) to prevent terminating the quoted string. Backslashes can be escaped with backslashes. Custom attribute names should use a unique prefix to prevent accidental namespace overlap. See the discussion of publisher names in "Package Publisher" on page 19.

Actions can have multiple attributes. Some attributes can be named multiple times with different values for a single action. Multiple attributes with the same name are treated as unordered lists.

Actions with many attributes can create long lines in a manifest file. Such lines can be wrapped by terminating each incomplete line with a backslash character. Note that this continuation character must occur between attribute/value pairs. Neither attributes nor their values nor the combination can be split.

Some attributes cause additional operations to be executed outside of the packaging context. See Chapter 7, "Automating System Change as Part of Package Installation" for more information.

Most actions have a key attribute. The *key attribute* is the attribute that makes this action unique from all other actions in the image. For file system objects, the key attribute is the path for that object.

Actions that are installed to a path must not deliver content to any of the following paths:

- `/system/volatile`
- `/tmp`
- `/var/pkg`
- `/var/share`
- `/var/tmp`

The following sections describe each IPS action type and the attributes that define these actions. The action types are detailed in the `pkg(5)` man page, and are repeated here for reference. Each section contains an example action as it would appear in a package manifest during package creation. Other attributes might be automatically added to the action during publication.

## File Actions

The file action is by far the most common action. A file action represents an ordinary file. The file action references a payload, and has the following four standard attributes:

path
: The file system path where the file is installed. This is the key attribute of a file action. The value of the path attribute is relative to the root of the image. Do not include the leading /.

mode
: The access permissions of the file. The value of the mode attribute is simple permissions in numeric form, not ACLs.

owner
: The name of the user that owns the file.

group
: The name of the group that owns the file.

The payload attribute is positional: The payload attribute is the first word after the action name and usually has no attribute name. If in a manifest that has not yet been published, the value of the payload attribute is the full path to the payload file, less the leading slash character (/). If the payload value includes an equal symbol (=), use hash= in front of the payload attribute value. If both positional and hash payload attributes are used in the same action, the values must be identical.

The following example is a file action that you might see output by the pkgsend generate command as shown in "Generate a Package Manifest" on page 41:

```
file opt/mysoftware path=opt/mysoftware group=bin mode=0644 owner=root
```

The following example shows how to specify a path that includes an equal symbol:

```
file hash=opt/my=software path=opt/my=software group=root mode=0644 owner=root
```

In a published manifest, the value of the payload attribute is a hash of the file contents, which is used by the package system. See the pkgsend(1) man page for more information.

The preserve and overlay attributes affect whether and how a file action is installed.

preserve

Specifies when and how files are preserved during package operations.

When a package is initially installed, if a file delivered by the package has a preserve attribute defined with any value except abandon or install-only and file already exists in the image, the existing file is stored in /var/pkg/lost+found and the packaged file is installed.

When a package is initially installed, if a file delivered by the package has a preserve attribute defined and the file does not already exist in the image, whether that file is installed depends on the value of the preserve attribute:

- If the value of preserve is abandon or legacy, the packaged file is not installed.

- If the value of preserve is not abandon or legacy, the packaged file is installed.

When a package is downgraded, if a file delivered by the downgraded version of the package has a preserve attribute defined with any value except abandon or install-only and all of the following conditions are true, the file that currently exists in the image is renamed with the extension .update, and the file from the downgraded package is installed.

- The file exists in the image.
- The content of the file delivered by the downgraded version of the package is different from the content of the file delivered by the currently installed version of the package.
- The content of the file delivered by the downgraded version of the package is different from the content of the file that exists in the image.

If any of the above conditions is not true, the file is treated the same as if the package is being upgraded, rather than downgraded.

When a package is upgraded, if a file action delivered by the upgraded version of the package has a preserve attribute defined with any value and the file action is the same as the file action delivered by the currently installed version of the package, the file is not installed, and the file that exists in the image is not modified. Any modifications made since the previous version was installed are preserved.

When a package is upgraded, if a file action delivered by the upgraded version of the package has a preserve attribute defined and the file action is new or is different from the file action delivered by the currently installed version of the package, the upgrade is done in the following way:

- If the file delivered by the upgraded version of the package has a preserve value of abandon or install-only in the upgraded package, the new file is not installed and the existing file is not modified.
- If the file does not exist in the image, the new file is installed.
- If the file delivered by the upgraded version of the package exists in the image, did not exist in the currently installed version of the package, and was not renamed or moved by using the original_name attribute, then the existing file is stored in /var/pkg/lost+found and the file delivered by the upgraded version of the package is installed. See the original_name attribute description below.
- If the file delivered by the upgraded version of the package exists in the image and has different content from the file delivered by the currently installed version of the package, the upgrade is done according to the value of the preserve attribute:
  - If the file delivered by the upgraded version of the package has a preserve value of renameold, the existing file is renamed with the extension .old, and the new file is installed with updated permissions and timestamp (if present). See the timestamp attribute description below.
  - If the file delivered by the upgraded version of the package has a preserve value of renamenew, the new file is installed with the extension .new and the existing file is not modified.

  - If the file delivered by the upgraded version of the package has a `preserve` value of `true`, the new file is not installed, but the permissions and timestamp (if present) are reset on the existing file.
- If the file delivered by the upgraded version of the package exists in the image, has the same content as the file delivered by the currently installed version of the package, and has a `preserve` value of either `renameold` or `renamenew`, the existing file is replaced by the file delivered by the upgraded version of the package, including replacing permissions and timestamp (if present).
- If the file delivered by the upgraded version of the package exists in the image, has a `preserve` value of `legacy` in the upgraded package, and has a different `preserve` value in the currently installed version of the package, the existing file is renamed with the extension `.legacy`, and the new file is installed with updated permissions and timestamp (if present).
- If the file delivered by the upgraded version of the package exists in the image and has a `preserve` value of `legacy` in both the upgraded package and the currently installed version of the package, the permissions and timestamp (if present) are reset on the existing file.

When a package is uninstalled, if a `file` action delivered by the currently installed version of the package has a `preserve` value of `abandon` or `install-only` and the file exists in the image, the file is not removed.

`overlay`

Specifies whether the action allows other packages to deliver a file at the same location or whether it delivers a file intended to overlay another file. This functionality is intended for use with configuration files that do not participate in any self-assembly and that can be safely overwritten.

If `overlay` is not specified, multiple packages cannot deliver files to the same location.

The `overlay` attribute can have one of the following values:

`allow` One other package is allowed to deliver a file to the same location. This value has no effect unless the `preserve` attribute is also set.

`true` The file delivered by the action overwrites any other action that has specified `allow`.

Changes to the installed file are preserved based on the value of the `preserve` attribute of the overlaying file. On removal, the contents of the file are preserved if the action being overlaid is still installed, regardless of whether the `preserve` attribute was specified. Only one action can overlay another action, and the `mode`, `owner`, and `group` attributes must match.

The following attributes are recognized for ELF files:

`elfarch` The architecture of the ELF file. This value is the output of `uname -p` on the architecture for which the file is built.

`elfbits` This value is 32 or 64.

`elfhash` This value is the hash of the ELF sections in the file that are mapped into memory when the binary is loaded. These are the only sections necessary to consider when determining whether the executable behavior of two binaries will differ.

The following additional attributes are recognized for `file` actions:

`original_name`

This attribute is used to handle editable files moving from package to package, from place to place, or both. The value of this attribute is the name of the originating package, followed by a colon, followed by the original path to the file. Any file being deleted is recorded either with its package and path, or with the value of the `original_name` attribute if specified. Any editable file being installed that has the `original_name` attribute set uses the file of that name if it is deleted as part of the same packaging operation.

Once this attribute is set, do not change its value, even if the package or file are repeatedly renamed. Keeping the same value permits upgrade to occur from all previous versions.

`release-note`

This attribute is used to indicate that this file contains release note text. The value of this attribute is a package FMRI. If the FMRI specifies a package name that is present in the original image and a version that is newer than the version of the package in the original image, this file will be part of the release notes. A special FMRI of `feature/pkg/self` refers to the containing package. If the version of `feature/pkg/self` is 0, this file will only be part of the release notes on initial installation.

`revert-tag`

This attribute is used to tag editable files that should be reverted as a set. The value of the `revert-tag` attribute is a *tagname*. Multiple `revert-tag` attributes can be specified for a single `file` action. The file reverts to its manifest-defined state when `pkg revert` is invoked with any of those tags specified. See "Reverting Tagged Files and Directories" in *Adding and Updating Software in Oracle Solaris 11.3* and the `pkg`(1) man page for information about the `pkg revert` command.

The `revert-tag` attribute can also be specified at the directory level. See "Directory Actions" below.

`sysattr`

This attribute is used to specify any system attributes that should be set for this file. The value of the `sysattr` attribute can be a comma-separated list of verbose system attributes or a string sequence of compact system attribute options, as shown in the following examples. Supported system attributes are explained in the `chmod`(1) man page. System attributes specified in the manifest are set additionally to system attributes that might have been set by other subsystems of the operating system.

```
file path=opt/secret_file sysattr=hidden,sensitive
file path=opt/secret_file sysattr=HT
```

`timestamp`

This attribute is used to set the access and modification time on the file. The `timestamp` attribute value must be expressed in UTC in ISO-8601 format, omitting the colons and hyphens.

The `timestamp` attribute is essential when packaging `.pyc` or `.pyo` files for Python. The related `.py` file for the `.pyc` or `.pyo` files must be marked with the timestamp embedded within those files, as shown in the following example:

```
file path=usr/lib/python2.6/vendor-packages/pkg/__init__.pyc ...
file path=usr/lib/python2.6/vendor-packages/pkg/__init__.py \
     timestamp=20150331T111615Z ...
```

The following attributes for `file` actions are automatically generated by the system and should not be specified by package developers: `hash`, `chash`, `pkg.size`, `pkg.csize`, and `pkg.content-hash`.

An example `file` action is:

```
file path=usr/bin/pkg owner=root group=bin mode=0755
```

## Directory Actions

The `dir` action is like the `file` action in that it represents a file system object. The `dir` action represents a directory instead of an ordinary file. The `dir` action has the same `path`, `mode`, `owner`, and `group` attributes that the `file` action has, and `path` is the key attribute. The `dir` action also accepts the `revert-tag` attribute, but the value of the attribute is different for `file` and `dir` actions.

Directories are reference counted in IPS. When the last package that either explicitly or implicitly references a directory no longer does so, that directory is removed. If that directory contains unpackaged file system objects, those items are moved into `$IMAGE_META/lost+found`. The value of `$IMAGE_META` is typically `/var/pkg`.

`revert-tag`

This attribute is used to identify unpackaged files that should be removed as a set. See "File Actions" above for a description of how to specify this attribute for `file` actions. For directories, the value of the `revert-tag` attribute is *tagname*=*pattern*. Multiple `revert-tag` attributes can be specified for a single `dir` action. When `pkg revert` is invoked with a matching *tagname*, any unpackaged files or directories under this `dir` directory that match *pattern* (using shell globbing characters) are removed. See "Reverting Tagged Files and Directories" in *Adding and Updating Software in Oracle Solaris 11.3* and the `pkg(1)` man page for information about the `pkg revert` command.

`salvage-from`

This attribute can be used to move unpackaged contents into a new directory. The value of this attribute is the name of a directory of salvaged items. A directory with a `salvage-from` attribute inherits on creation any contents of the directory named in the value of the `salvage-from` attribute.

During installation, `pkg` checks that all instances of a given directory action on the system have the same `owner`, `group`, and `mode` attribute values. The `dir` action is not installed if conflicting values are found on the system or in other packages to be installed in the same operation.

An example of a `dir` action is:

```
dir path=usr/share/lib owner=root group=sys mode=0755
```

## Link Actions

The `link` action represents a symbolic link. The `link` action has the following standard attributes:

`path` The file system path where the symbolic link is installed. This is the key attribute for a `link` action.

`target` The target of the symbolic link. The file system object to which the link resolves.

The `link` action also takes the following attributes that allow for multiple versions or implementations of a given piece of software to be installed on the system at the same time: `mediator`, `mediator-version`, `mediator-implementation`, and `mediator-priority`. Such links are mediated, and allow administrators to easily toggle which links point to which version or implementation as desired. These mediated link attributes are discussed in detail in “Delivering Multiple Implementations of an Application” on page 112. Mediations are also discussed in “Specifying a Default Application Implementation” in *Adding and Updating Software in Oracle Solaris 11.3*.

An example of a `link` action is:

```
link path=usr/lib/libpython2.6.so target=libpython2.6.so.1.0
```

## Hardlink Actions

The `hardlink` action represents a hard link. It has the same `path` and `target` attributes as the `link` action, and `path` is also its key attribute.

An example of a `hardlink` action is:

```
hardlink path=opt/myapplication/hardlink target=foo
```

## Set Actions

The set action represents a package-level attribute, or metadata, such as the package description.

The following attributes are recognized:

name — The name of the attribute.

value — The value given to the attribute.

The set action can deliver any metadata the package author chooses. The following attribute names have specific meaning to the packaging system:

info.classification

One or more tokens that a pkg client can use to classify the package. The value should have a scheme (such as org.opensolaris.category.2008 or org.acm.class.1998) and the actual classification (such as Applications/Games), separated by a colon (:). A set of info.classification values is provided in Appendix A, “Classifying Packages”.

pkg.description

A detailed description of the contents and functionality of the package, typically a paragraph or so in length. This value should describe why someone might want to install this package.

pkg.fmri

The name and version of the containing package. See “Package Version” on page 20.

pkg.human-version

The version scheme used by IPS is strict. See “Package Version” on page 20. A more flexible version can be provided as the value of the pkg.human-version attribute. The value is displayed by IPS tools such as pkg info, pkg contents, and pkg search. The pkg.human-version value is not used as a basis for version comparison and cannot be used in place of the pkg.fmri version.

pkg.obsolete

When true, the package is marked obsolete. An obsolete package can have no actions other than more set actions, and must not be marked renamed. Package obsoletion is covered in “Obsoleting Packages” on page 98.

pkg.renamed

When true, the package has been renamed. The package must include one or more depend actions as well, which point to the package versions to which this package has been renamed. A package cannot be marked both renamed and obsolete, but otherwise can have any number of set actions. Package renaming is covered in "Renaming, Merging, and Splitting Packages" on page 96.

pkg.summary

A brief synopsis of the description. This value is shown at the end of each line of pkg list -s output, as well as in one line of the output of pkg info. This value should be no longer than 60 characters. This value should describe what the package is, and should not repeat the name or version of the package.

Some additional informational attributes, as well as some used by Oracle Solaris are described in Appendix B, "How IPS Is Used To Package the Oracle Solaris OS".

An example of a set action is:

```
set name=pkg.summary value="Image Packaging System"
```

## Driver Actions

The driver action represents a device driver. The driver action does not reference a payload. The driver files themselves must be installed as file actions. The following attributes are recognized. See the add_drv(1M) man page for more information about these attribute values.

name

The name of the driver. This is usually, but not always, the file name of the driver binary. This is the key attribute of the driver action.

alias

An alias for the driver. A given driver can have more than one alias attribute. No special quoting rules are necessary.

class

A driver class. A given driver can have more than one class attribute.

perms

The file system permissions for the device nodes of the driver.

clone_perms

The file system permissions for the minor nodes of the clone driver for this driver.

policy

Additional security policy for the device. A given driver can have more than one policy attribute, but no minor device specification can be present in more than one attribute.

privs

Privileges used by the driver. A given driver can have more than one privs attribute.

devlink

An entry in /etc/devlink.tab. The value is the exact line to go into the file, with tabs denoted by \t. See the devlinks(1M) man page for more information. A given driver can have more than one devlink attribute.

An example of a driver action is:

```
driver name=vgatext \
    alias=pciclass,000100 \
    alias=pciclass,030000 \
    alias=pciclass,030001 \
    alias=pnpPNP,900 variant.arch=i386 variant.opensolaris.zone=global
```

## Depend Actions

The `depend` action represents an inter-package dependency. A package can depend on another package because the first package requires functionality in the second package for the functionality in the first package to work or to install. Dependencies can be optional. If a dependency is not satisfied at the time of installation, the packaging system attempts to install or update the dependent package to a sufficiently new version, subject to other constraints. Dependencies are covered in more detail in Chapter 4, "Specifying Package Dependencies".

The following attributes are recognized:

`fmri`

The FMRI representing the target of the dependency. This is the key attribute of the `depend` action. The FMRI value must not include the publisher. The package name is assumed to be rooted, even if it does not begin with a forward slash (`/`). Dependencies of type `require-any` can have multiple `fmri` attributes. A version is optional on the `fmri` value, though for some types of dependencies, an FMRI with no version has no meaning.

The FMRI value cannot use asterisks (`*`), and cannot use the `latest` token for a version.

`type`

The type of the dependency.

`require`

The target package is required and must have a version equal to or greater than the version specified in the `fmri` attribute. If the version is not specified, any version satisfies the dependency. A package cannot be installed if any of its `require` dependencies cannot be satisfied.

`optional`

The dependency target, if present, must be at the specified version level or greater.

`exclude`

The containing package cannot be installed if the dependency target is present at the specified version level or greater. If no version is specified, the target package cannot be installed concurrently with the package specifying the dependency.

`incorporate`

The dependency is optional, but the version of the target package is constrained. See Chapter 4, "Specifying Package Dependencies" for a discussion of constraints and freezing.

`require-any`

Any one of multiple target packages as specified by multiple `fmri` attributes can satisfy the dependency, following the same rules as the `require` dependency type.

`conditional`

The dependency target is required only if the package defined by the `predicate` attribute is present on the system.

`origin`

Prior to installation of this package, the dependency target must, if present, be at the specified value or greater on the image to be modified. If the value of the `root-image` attribute is `true`, the target must be present on the image rooted at `/` in order to install this package. If the value of the `root-image` attribute is `true` and the value of the `fmri` attribute starts with `pkg:/feature/firmware/`, the remainder of the `fmri` value is treated as a command in `/usr/lib/fwenum` that evaluates the firmware dependency.

`group`

The dependency target is required unless the package is on the image avoid list. Note that obsolete packages silently satisfy the `group` dependency. See the `avoid` subcommand in the `pkg(1)` man page for information about the image avoid list.

`group-any`

Any one of multiple target packages as specified by multiple `fmri` attributes can satisfy the dependency. The same rules apply to a `group-any` dependency that apply to a `group` dependency with the exception that non-obsolete package stems are preferred over obsolete package stems.

`parent`

The dependency is ignored if the image is not a child image, such as a zone. If the image is a child image, then the dependency target must be present in the parent image. The version matching for a `parent` dependency is the same as that used for `incorporate` dependencies.

`predicate`

The FMRI that represents the predicate for conditional dependencies.

`root-image`

Has an effect only for `origin` dependencies as mentioned above.

An example of a `depend` action is:

```
depend fmri=crypto/ca-certificates type=require
```

## License Actions

The `license` action represents a license or other informational file associated with the package contents. A package can deliver licenses, disclaimers, or other guidance to the package installer through the `license` action.

The payload of the `license` action is delivered into the image metadata directory related to the package, and should only contain human-readable text data. The `license` action payload should not contain HTML or any other form of markup. Through attributes, `license` actions can indicate to `pkg` clients that the related payload must be displayed or accepted. The method of display or acceptance is at the discretion of `pkg` clients.

The following attributes are recognized:

`license`

Provides a meaningful description for the license to assist users in determining the contents without reading the license text itself. This is the key attribute of the `license` action.

Some example values include:

- ABC Co. Copyright Notice
- ABC Co. Custom License
- Common Development and Distribution License 1.0 (CDDL)
- GNU General Public License 2.0 (GPL)
- GNU General Public License 2.0 (GPL) Only
- MIT License
- Mozilla Public License 1.1 (MPL)
- Simplified BSD License

Wherever possible, including the version of the license in the description is recommended as shown above. The `license` value must be unique within a package.

`must-accept`

When `true`, this license must be accepted by a user before the related package can be installed or updated. Omission of this attribute is equivalent to `false`. The method of acceptance (interactive or configuration-based, for example) is at the discretion of `pkg` clients. For package updates, this attribute is ignored if the license action or payload has not changed.

`must-display`

When `true`, the payload of the `license` action must be displayed by `pkg` clients during packaging operations. Omission of this attribute is equivalent to `false`. This attribute should not be used for copyright notices. This attribute should only be used for licenses or other material that must be displayed during operations. The method of display is at the discretion of `pkg` clients. For package updates, this attribute is ignored if the license action or payload has not changed.

An example of a `license` action is:

```
license license="Apache v2.0"
```

## Legacy Actions

The `legacy` action represents package data used by the legacy SVR4 packaging system. The attributes associated with the `legacy` action are added into the databases of the legacy SVR4 packaging system so that the tools querying those databases can operate as if the legacy package were actually installed. In particular, specifying the `legacy` action should cause the package named by the `pkg` attribute to satisfy SVR4 dependencies.

The following attributes are recognized. See the `pkginfo`(4) man page for description of the associated parameters.

`category` The value for the CATEGORY parameter. The default value is `system`.

`desc` The value for the DESC parameter.

`hotline` The value for the HOTLINE parameter.

`name` The value for the NAME parameter. The default value is `none provided`.

`pkg` The abbreviation for the package being installed. The default value is the name from the FMRI of the package. This is the key attribute of the `legacy` action.

`vendor` The value for the VENDOR parameter.

`version` The value for the VERSION parameter. The default value is the version from the FMRI of the package.

An example of a `legacy` action is:

```
legacy pkg=SUNWcsu arch=i386 category=system \
    desc="core software for a specific instruction-set architecture" \
    hotline="Please contact your local service provider" \
    name="Core Solaris, (Usr)" vendor="Oracle Corporation" \
    version=11.11,REV=2009.11.11 variant.arch=i386
```

## Signature Actions

Signature actions are used as part of the support for package signing in IPS. Signature actions are covered in detail in Chapter 9, “Signing IPS Packages”.

## User Actions

The user action defines a UNIX user as specified in the /etc/passwd, /etc/shadow, /etc/group, and /etc/ftpd/ftpusers files. Information from user actions is added to the appropriate files.

The user action is intended to define a user for a daemon or other software to use. Do not use the user action to define administrative or interactive accounts.

The following attributes are recognized:

username The unique name of the user.

password The encrypted password of the user. The default value is *LK*.

uid The unique numeric ID of the user. The default value is the first free value under 100.

group The name of the user's primary group. This name must be found in /etc/group.

gcos-field The real name of the user, as represented in the GECOS field in /etc/passwd. The default value is the value of the username attribute.

home-dir The user's home directory. This directory must be in the system image directories and not under another mount point such as /home. The default value is /.

login-shell The user's default shell. The default value is empty.

group-list Secondary groups to which the user belongs. See the group(4) man page.

ftpuser Can be set to true or false. The default value of true indicates that the user is permitted to login via FTP. See the ftpusers(4) man page.

lastchg The number of days between January 1, 1970, and the date that the password was last modified. The default value is empty.

min The minimum number of days required between password changes. This field must be set to 0 or above to enable password aging. The default value is empty.

`max` The maximum number of days the password is valid. The default value is empty. See the `shadow(4)` man page.

`warn` The number of days before `password` expires that the user is warned.

`inactive` The number of days of inactivity allowed for the user. This is counted on a per-system basis. The information about the last login is taken from the `lastlog` file of the system.

`expire` An absolute date expressed as the number of days since the UNIX Epoch (January 1, 1970). When this number is reached, the login can no longer be used. For example, an expire value of 13514 specifies a login expiration of January 1, 2007.

`flag` Set to empty.

A example of a `user` action is:

```
user ftpuser=false gcos-field="AI User" group=aiuser uid=61 username=aiuser
```

### Group Actions

The `group` action defines a UNIX group as specified in the `group(4)` file. No support is provided for group passwords. Groups defined with the `group` action initially have no user list. Users can be added with the `user` action.

The following attributes are recognized:

`groupname` The value for the name of the group.

`gid` The unique numeric ID of the group. The default value is the first free group under 100.

An example of a `group` action is:

```
group gid=61 groupname=aiuser
```

## Package Repository

A software repository contains packages for one or more publishers. Repositories can be configured for access in a variety of different ways: HTTP, HTTPS, file (on local storage or via NFS or SMB), and as a self-contained package archive file, usually with the `.p5p` extension.

Package archives allow for convenient distribution of IPS packages. See “Deliver as a Package Archive File” on page 54 for more information.

A repository accessed via HTTP or HTTPS is managed by a `pkg/server` SMF service and `pkg.depotd` process or possibly by a `pkg/depot` SMF service. The `pkg/depot` service is delivered by the `package/pkg/depot` package. For file repositories, the repository software runs as part of the accessing `pkg` client.

See "Publish the Package" on page 49 and "Deliver the Package" on page 53 for examples. See *Copying and Creating Package Repositories in Oracle Solaris 11.3* for more information about creating, accessing, updating, and configuring IPS package repositories.

♦ ♦ ♦ CHAPTER 2

# Packaging Software With IPS

This chapter gets you started constructing your own packages, including:

- Designing, creating, and publishing a new package
- Converting a SVR4 package to an IPS package

## Designing a Package

Many of the criteria for good package development described in this section require you to make trade-offs. Satisfying all requirements equally is often difficult. The following criteria are presented in order of importance. However, this sequence is meant to serve as a flexible guide depending on your circumstances. Although each of these criteria is important, it is up to you to optimize these requirements to produce a good set of packages.

**Select a package name.**

Oracle Solaris uses a hierarchical naming strategy for IPS packages. Wherever possible, design your package names to fit into the same scheme. Try to keep the last part of your package name unique so that users can specify a short package name to commands such as `pkg install`.

**Optimize for client-server configurations.**

Consider the various patterns of software use (client and server) when laying out packages. Good package design divides the affected files to optimize installation of each configuration type. For example, for a network protocol implementation, the package user should be able to install the client without necessarily installing the server. If client and server share implementation components, create a base package that contains the shared bits.

**Package by functional boundaries.**

Packages should be self-contained and distinctly identified with a set of functionality. For example, a package that contains ZFS should contain all ZFS utilities and be limited to only ZFS binaries.

Packages should be organized from a user's point of view into functional units.

**Package along license or royalty boundaries.**

Put code that requires royalty payments due to contractual agreements or that has distinct software license terms in a dedicated package or group of packages. Do not disperse the code into more packages than necessary.

**Avoid or manage overlap between packages.**

Packages that overlap cannot be installed at the same time. An example of packages that overlap are packages that deliver different content to the same file system location. Since this error might not be caught until the user attempts to install the package, overlapping packages can provide a poor user experience. The `pkglint(1)` tool can help to detect this error during the package authoring process.

If the package content must differ, declare an `exclude` dependency so that IPS does not allow these packages to be installed together.

**Correctly size packages.**

A package represents a single unit of software, and is either installed or not installed. (See the discussion of facets in "Optional Software Components" on page 78 to understand how a package can deliver optional software components.) Packages that are always installed together should be combined. Since IPS downloads only changed files on update, even large packages update quickly if change is limited.

Do not deliver content to any of the following paths:

- `/system/volatile`
- `/tmp`
- `/var/pkg`
- `/var/share`
- `/var/tmp`

## Creating and Publishing a Package

Packaging software with IPS is usually straightforward due to the amount of automation that is provided. Automation avoids repetitive tedium, which seems to be the principal cause of most packaging bugs.

Publication in IPS consists of the following steps:

1. Generate a package manifest.
2. Add necessary metadata to the generated manifest.
3. Evaluate dependencies.
4. Add any facets or actuators that are needed.
5. Verify the package.

6. Publish the package.
7. Test the package.

## Generate a Package Manifest

The easiest way to get started is to organize the component files into the same directory structure that you want on the installed system.

Two ways to do this are:

- If the software you want to package is already in a tarball, unpack the tarball into a subdirectory. For many open source software packages that use the autoconf utility, setting the DESTDIR environment variable to point to the desired prototype area accomplishes this. The autoconf utility is available in the pkg:/developer/build/autoconf package.
- Use the install target in a Makefile.

Suppose your software consists of a binary, a library, and a man page, and you want to install this software in a directory under /opt named mysoftware. Create a directory in your build area under which your software appears in this layout. In the following example, this directory is named proto:

```
proto/opt/mysoftware/lib/mylib.so.1
proto/opt/mysoftware/bin/mycmd
proto/opt/mysoftware/man/man1/mycmd.1
```

Use the pkgsend generate command to generate a manifest for this proto area. Pipe the output package manifest through pkgfmt to make the manifest more readable. See the pkgsend(1) and pkgfmt(1) man pages for more information.

In the following example, the proto directory is in the current working directory:

```
$ pkgsend generate proto | pkgfmt > mypkg.p5m.1
```

The output mypkg.p5m.1 file contains the following lines:

```
dir  path=opt owner=root group=bin mode=0755
dir  path=opt/mysoftware owner=root group=bin mode=0755
dir  path=opt/mysoftware/bin owner=root group=bin mode=0755
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0644
dir  path=opt/mysoftware/lib owner=root group=bin mode=0755
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644
```

The path of the files to be packaged appears twice in the `file` action:

- The first word after the word `file` describes the location of the file in the `proto` area.
- The path in the `path=` attribute specifies the location where the file is to be installed.

This double entry enables you to modify the installation location without modifying the `proto` area. This capability can save significant time, for example if you repackage software that was designed for installation on a different operating system.

Notice that `pkgsend generate` has applied default values for directory owners and groups. In the case of `/opt`, the defaults are not correct. Delete that directory because it is delivered by other packages already on the system, and `pkg`(1) will not install the package if the attributes of `/opt` conflict with those already on the system. "Add Necessary Metadata to the Generated Manifest" on page 42 below shows a programmatic way to delete the unwanted directory.

If a file name contains an equal symbol (`=`), double quotation mark (`"`), or space character, `pkgsend` generates a `hash` attribute in the manifest, as shown in the following example:

```
$ mkdir -p proto/opt
$ touch proto/opt/my\ file1
$ touch proto/opt/"my file2"
$ touch proto/opt/my=file3
$ touch proto/opt/'my"file4'
$ pkgsend generate proto
dir group=bin mode=0755 owner=root path=opt
file group=bin hash=opt/my=file3 mode=0644 owner=root path=opt/my=file3
file group=bin hash="opt/my file2" mode=0644 owner=root path="opt/my file2"
file group=bin hash='opt/my"file4' mode=0644 owner=root path='opt/my"file4'
file group=bin hash="opt/my file1" mode=0644 owner=root path="opt/my file1"
```

When the package is published (see "Publish the Package" on page 49), the value of the `hash` attribute becomes the SHA-1 hash of the file contents, as noted in "File Actions" on page 23.

## Add Necessary Metadata to the Generated Manifest

A package should define the following metadata. See "Set Actions" on page 29 for more information about these values and how to set these values.

`pkg.fmri`

The name and version of the package as described in "Package Name" on page 19 and "Package Version" on page 20. See "Avoiding Conflicting Package Content" on page 95 for a discussion of package names and dependencies. The publisher name is added automatically when the package is published, as shown in "Publish the Package" on page 49. See "Construct an Appropriate Package Version String" on page 44 for additional help setting your package version. See "Oracle Solaris Package Versioning" on page 141 for a description of versioning in Oracle Solaris.

pkg.description

A description of the contents of the package.

pkg.summary

A one-line synopsis of the description.

variant.arch

Each architecture for which this package is suitable. If the entire package can be installed on any architecture, variant.arch can be omitted. Producing packages that have different components for different architectures is discussed in Chapter 5, "Allowing Variations".

info.classification

A grouping scheme used by the packagemanager(1) GUI. The supported values are shown in Appendix A, "Classifying Packages". The example in this section specifies an arbitrary classification.

This example also adds a link action to /usr/share/man/index.d that points to the man directory under mysoftware. This link is discussed further in "Add Any Facets or Actuators That Are Needed" on page 47.

Rather than modifying the generated manifest directly, use pkgmogrify(1) to edit the generated manifest. See Chapter 6, "Modifying Package Manifests Programmatically" for a full description of using pkgmogrify to modify package manifests.

Create the following pkgmogrify input file to specify the changes to be made to the manifest. Name this file mypkg.mog. In this example, a macro is used to define the architecture, and regular expression matching is used to delete the /opt directory from the manifest.

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.summary value="This is an example package"
set name=pkg.description value="This is a full description of \
all the interesting attributes of this example package."
set name=variant.arch value=$(ARCH)
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
link path=usr/share/man/index.d/mysoftware target=/opt/mysoftware/man
<transform dir path=opt$->drop>
```

Run pkgmogrify on the mypkg.p5m.1 manifest with the mypkg.mog changes:

```
$ pkgmogrify -DARCH=`uname -p` mypkg.p5m.1 mypkg.mog | pkgfmt > mypkg.p5m.2
```

The output `mypkg.p5m.2` file has the following content. The `dir` action for `path=opt` has been removed, and the metadata and link contents from `mypkg.mog` have been added to the original `mypkg.p5m.1` contents.

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.summary value="This is an example package"
set name=pkg.description \
    value="This is a full description of all the interesting attributes of this
example package."
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
set name=variant.arch value=i386
dir  path=opt/mysoftware owner=root group=bin mode=0755
dir  path=opt/mysoftware/bin owner=root group=bin mode=0755
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0644
dir  path=opt/mysoftware/lib owner=root group=bin mode=0755
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644
link path=usr/share/man/index.d/mysoftware target=/opt/mysoftware/man
```

## Construct an Appropriate Package Version String

The component version, release, and branch version (see "Package Version" on page 20) of an IPS package version string have the following constraints:

- All content must be only periods or integers. If your product version contains other characters, such as alphabetic characters, select a version for the IPS package that conveys the same meaning using only integers and periods. For example, if your product version is P17-u4-r3, then 17.4.3 might work for the package version.
- A sequence of more than one integer cannot begin with a zero. This format enables sorting by package version. For example, version 1.20 sorts newer than version 1.0.2, but 1.02 is invalid. If your product version 17.03 indicates the third test release of version 17 of the product, and the final released product will be version 17.0, then you could use package versions such as 16.99.3, 16.99.4, and so on for the test releases. Version 17.0 will be seen as newer than 16.99 versions, whereas 17.0 would not be newer than 17.0.3.

You can use the `pkg.human-version` attribute to provide your actual product version string as shown in the following example:

```
set name=pkg.human-version value="P17-u4-r3"
```

The value of the `pkg.human-version` attribute can be provided in addition to the package version in the package FMRI but cannot replace the package FMRI version. The `pkg.human-`

version version string is used only for display purposes. See "Set Actions" on page 29 for more information.

# Evaluate Dependencies

Use the pkgdepend(1) command to automatically generate dependencies for the package. The generated depend actions are defined in "Depend Actions" on page 31 and discussed further in Chapter 4, "Specifying Package Dependencies".

Dependency generation is composed of two separate steps:

1. Dependency generation. Determine the files on which the software depends. Use the pkgdepend generate command.
2. Dependency resolution. Determine the packages that contain those files on which the software depends. Use the pkgdepend resolve command.

## Generate Package Dependencies

**Tip -** Use pkgdepend to generate dependencies, rather than declaring depend actions manually. Manual dependencies can become incorrect or unnecessary as the package contents change over time. For example, when a file that an application depends on gets moved to a different package, any manually declared dependencies on the previous package would then be incorrect for that dependency.

Some manually declared dependencies might be necessary if pkgdepend is unable to determine dependencies completely. In such a case, you should add explanatory comments to the manifest.

In the following command, the -m option causes pkgdepend to include the entire manifest in its output. The -d option passes the proto directory to the command.

```
$ pkgdepend generate -md proto mypkg.p5m.2 | pkgfmt > mypkg.p5m.3
```

The output mypkg.p5m.3 file has the following content. The pkgdepend utility added notations about a dependency on libc.so.1 by both mylib.so.1 and mycmd. The internal dependency between mycmd and mylib.so.1 is silently omitted.

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.summary value="This is an example package"
set name=pkg.description \
    value="This is a full description of all the interesting attributes of this
example package."
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
set name=variant.arch value=i386
```

```
dir  path=opt/mysoftware owner=root group=bin mode=0755
dir  path=opt/mysoftware/bin owner=root group=bin mode=0755
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0644
dir  path=opt/mysoftware/lib owner=root group=bin mode=0755
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644
link path=usr/share/man/index.d/mysoftware target=/opt/mysoftware/man
depend fmri=__TBD pkg.debug.depend.file=libc.so.1 \
    pkg.debug.depend.reason=opt/mysoftware/bin/mycmd \
    pkg.debug.depend.type=elf type=require pkg.debug.depend.path=lib \
    pkg.debug.depend.path=opt/mysoftware/lib pkg.debug.depend.path=usr/lib
depend fmri=__TBD pkg.debug.depend.file=libc.so.1 \
    pkg.debug.depend.reason=opt/mysoftware/lib/mylib.so.1 \
    pkg.debug.depend.type=elf type=require pkg.debug.depend.path=lib \
    pkg.debug.depend.path=usr/lib
```

## Resolve Package Dependencies

To resolve dependencies, pkgdepend examines the packages currently installed in the image used for building the software. By default, pkgdepend puts its output in mypkg.p5m.3.res. This step takes a while to run since it loads a large amount of information about the system on which it is running. The pkgdepend utility can resolve many packages at once if you want to amortize this time over all packages. Running pkgdepend on one package at a time is not time efficient.

```
$ pkgdepend resolve -m mypkg.p5m.3
```

When this completes, the output mypkg.p5m.3.res file contains the following content. The pkgdepend utility has converted the notation about the file dependency on libc.so.1 to a package dependency on pkg:/system/library, which delivers that file.

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.summary value="This is an example package"
set name=pkg.description \
    value="This is a full description of all the interesting attributes of this
example package."
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
set name=variant.arch value=i386
dir  path=opt/mysoftware owner=root group=bin mode=0755
dir  path=opt/mysoftware/bin owner=root group=bin mode=0755
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0644
dir  path=opt/mysoftware/lib owner=root group=bin mode=0755
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644
link path=usr/share/man/index.d/mysoftware target=/opt/mysoftware/man
depend fmri=pkg:/system/library@0.5.11-0.175.2.0.0.18.0 type=require
```

## Add Any Facets or Actuators That Are Needed

A *facet* denotes an action that is not required but can be optionally installed. An *actuator* specifies system changes that must occur when the associated action is installed, updated, or removed. Facets are discussed in more detail in Chapter 5, “Allowing Variations”, and actuators are discussed in more detail in Chapter 7, “Automating System Change as Part of Package Installation”.

This example package delivers a man page in `/opt/mysoftware/man/man1`. This section shows how to add a facet tag to indicate that man pages are optional. The user could choose to install all of the package except the man page. (If the user sets the facet property `doc.man=false` as described in “Controlling Installation of Optional Components” in *Adding and Updating Software in Oracle Solaris 11.3*, no actions tagged with `facet.doc.man=true` are installed from any package.)

To include the man page in the index, the `svc:/application/man-index:default` SMF service must be restarted when the package is installed. This section shows how to add the `restart_fmri` actuator to perform that task. The `man-index` service looks in `/usr/share/man/index.d` for symbolic links to directories that contain man pages, adding the target of each link to the list of directories it scans. To include the man page in the index, this example package includes a link from `/usr/share/man/index.d/mysoftware` to `/opt/mysoftware/man`. Including this link and this actuator is a good example of the self-assembly discussed in “Software Self-Assembly” on page 13 and used throughout the packaging of the Oracle Solaris OS.

A set of `pkgmogrify` transforms that you can use are available in `/usr/share/pkg/transforms`. These transforms are used to package the Oracle Solaris OS, and are discussed in more detail in Chapter 6, “Modifying Package Manifests Programmatically”.

The file `/usr/share/pkg/transforms/documentation` contains transforms similar to the transforms needed in this example to set the man page facet and restart the `man-index` service. Since this example delivers the man page to `/opt`, the `documentation` transforms must be modified as shown below. These modified transforms include the regular expression `opt/.+/man(/.+)?` which matches all paths beneath `opt` that contain a `man` subdirectory. Save the following modified transforms to `/tmp/doc-transform`:

```
<transform dir file link hardlink path=opt/.+/man(/.+)? -> \
    default facet.doc.man true>
<transform file path=opt/.+/man(/.+)? -> \
    add restart_fmri svc:/application/man-index:default>
```

Use the following command to apply these transforms to the manifest:

```
$ pkgmogrify mypkg.p5m.3.res /tmp/doc-transform | pkgfmt > mypkg.p5m.4.res
```

The input `mypkg.p5m.3.res` manifest contains the following three man-page-related actions:

```
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644
```

After the transforms are applied, the output `mypkg.p5m.4.res` manifest contains the following modified actions:

```
dir  path=opt/mysoftware/man owner=root group=bin mode=0755 facet.doc.man=true
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755 \
    facet.doc.man=true
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644 \
    restart_fmri=svc:/application/man-index:default facet.doc.man=true
```

**Tip -** For efficiency, these transforms could have been added when metadata was originally added, before evaluating dependencies.

## Verify the Package

The last step before publication is to run `pkglint(1)` on the manifest to find errors that can be identified before publication and testing. Some of the errors that `pkglint` can find would also be found either at publication time or when a user attempts to install the package, but of course you want to identify errors as early as possible in the package authoring process.

Examples of errors that `pkglint` reports include:

- Delivering files already owned by another package.
- Difference in metadata for shared, reference-counted actions such as directories.
  An example of this error is discussed at the end of "Generate a Package Manifest" on page 41.

Use the `pkglint -L` command to show the full list of checks that `pkglint` performs. Detailed information about how to enable, disable, and bypass particular checks is given in the `pkglint(1)` man page. The man page also details how to extend `pkglint` to run additional checks.

You can run pkglint in one of the following modes:

- Directly on the package manifest. This mode is usually sufficient to quickly check the validity of your manifests.
- On the package manifest, also referencing a package repository. Use this mode at least once before publication to a repository.

  By referencing a repository, pkglint can perform additional checks to ensure that the package interacts well with other packages in that repository.

The following output shows problems with the example manifest:

```
$ pkglint mypkg.p5m.4.res
Lint engine setup...
Starting lint run...
WARNING pkglint.action005.1       obsolete dependency check skipped: unable
to find dependency pkg:/system/library@0.5.11-0.175.2.0.0.18.0 for
pkg:/mypkg@1.0,5.11-0
```

This warning is acceptable for this example. The pkglint.action005.1 warning says that pkglint could not find a package called pkg:/system/library@0.5.11-0.175.2.0.0.18.0, on which this example package depends. The dependency package is in a package repository and could not be found since pkglint was called with only the manifest file as an argument.

In the following command, the -r option references a repository that contains the dependency package. The -c option specifies a local directory used for caching package metadata from the lint and reference repositories:

```
$ pkglint -c ./solaris-reference -r http://pkg.oracle.com/solaris/release mypkg.p5m.4.
res
```

## Publish the Package

Publish your package to a local file-based repository. This repository is for developing and testing this new package. If you create a repository for general use, you should include additional steps such as creating a separate file system for the repository. For information about creating package repositories for general use, see *Copying and Creating Package Repositories in Oracle Solaris 11.3*.

To test the package with non-global zones, the repository location must be accessible through the system repository. Use the pkg publisher or pkg list command inside a non-global zone to confirm that the package is accessible.

Use the pkgrepo(1) command to create a repository on your system:

```
$ pkgrepo create my-repository
$ ls my-repository
```

```
pkg5.repository
```

Set the default publisher for this repository. The default publisher is the value of the `publisher/prefix` property of the repository.

```
$ pkgrepo -s my-repository set publisher/prefix=mypublisher
```

Use the `pkgsend publish` command to publish the new package. If multiple `pkgsend publish` processes might be publishing to the same `-s` repository simultaneously, specifying the `--no-catalog` option is recommended because updates to publisher catalogs must be performed serially. Publication performance might be significantly reduced if the `--no-catalog` option is not used when multiple processes are simultaneously publishing packages. After publication is complete, the new packages can be added to the respective publisher catalogs by using the `pkgrepo refresh` command.

```
$ pkgsend -s my-repository publish -d proto mypkg.p5m.4.res
pkg://mypublisher/mypkg@1.0,5.11-0:20130720T005452Z
PUBLISHED
```

Notice that the repository default publisher has been applied to the package FMRI.

Verify that the new repository permissions, content, and signatures are correct:

```
$ pkgrepo verify -s my-repository
```

You can use the `pkgrepo` and `pkg list` commands to examine the repository:

```
$ pkgrepo info -s my-repository
PUBLISHER   PACKAGES STATUS           UPDATED
mypublisher 1        online           2013-07-20T00:54:52.758591Z
$ pkgrepo list -s my-repository
PUBLISHER   NAME                    O VERSION
mypublisher mypkg                     1.0,5.11-0:20130720T005452Z
$ pkg list -afv -g my-repository
FMRI                                                     IFO
pkg://mypublisher/mypkg@1.0,5.11-0:20130720T005452Z      ---
```

A value in the `O` column of `pkgrepo list` indicates whether the package is obsolete (`o`) or renamed (`r`).

Publishing the new package directly to an HTTP repository is not recommended since no authorization or authentication checks are performed on the incoming package when publishing over HTTP. Instead of publishing the package to an HTTP repository, deliver the already-published package to an HTTP repository as described in “Deliver to a Package Repository” on page 53. Publishing to HTTP repositories can be convenient on secure networks or when testing the same package across several systems when NFS or SMB access to the file repository is not possible. If you publish directly to an HTTP repository, that repository must be hosted on a system with a read/write instance of the `svc:/application/pkg/server` service (the value of the `pkg/readonly` property is `false`).

## Sign the Package

If you want to sign your package, do that now, and then test the package and deliver the package to the general use repository or package archive.

The following command signs the package using the hash value of the package manifest. You can also specify your own signature key and certificate. See Chapter 9, “Signing IPS Packages” for more information. Notice that the time stamp of the package is not changed.

```
$ pkgsign -s my-repository -a sha256 '*'
Signed pkg://mypublisher/mypkg@1.0,5.11-0:20130720T005452Z
```

## Test the Package

The final step in package development is to install the package to test whether the published package has been packaged properly.

To test installation without requiring root privilege, assign the test user the Software Installation profile. Use the `-P` option of the `usermod` command to assign the test user the Software Installation profile.

**Note -** If this image has child images (non-global zones) installed, you cannot use the `-g` option with the `pkg install` command to test installation of this package. You must configure the `mypublisher` publisher in the image.

The following `pkg set-publisher` command adds all publishers in the `my-repository` repository to the list of publishers configured in this image:

```
$ pkg publisher
PUBLISHER    TYPE    STATUS P LOCATION
solaris      origin  online F http://pkg.oracle.com/solaris/release/
$ pkg set-publisher -p my-repository
pkg set-publisher:
  Added publisher(s): mypublisher
$ pkg publisher
PUBLISHER    TYPE    STATUS P LOCATION
solaris      origin  online F http://pkg.oracle.com/solaris/release/
mypublisher  origin  online F file:///home/username/my-repository/
```

Use the `-nv` options with the `pkg install` command to see what the install command will do without making any changes to the image. The following command actually installs the package:

```
$ pkg install mypkg
           Packages to install:  1
       Create boot environment: No
Create backup boot environment: No
```

```
            Services to change:  1

DOWNLOAD                                  PKGS       FILES   XFER (MB)   SPEED
Completed                                  1/1         3/3     0.0/0.0  787k/s

PHASE                                           ITEMS
Installing new actions                          16/16
Updating package state database                  Done
Updating image state                             Done
Creating fast lookup database                    Done
Reading search index                             Done
Updating search index                             1/1
```

Examine the software that was delivered on the system:

```
$ find /opt/mysoftware
/opt/mysoftware
/opt/mysoftware/bin
/opt/mysoftware/bin/mycmd
/opt/mysoftware/lib
/opt/mysoftware/lib/mylib.so.1
/opt/mysoftware/man
/opt/mysoftware/man/man1
/opt/mysoftware/man/man1/mycmd.1
/opt/mysoftware/man/man-index
/opt/mysoftware/man/man-index/term.dic
/opt/mysoftware/man/man-index/term.req
/opt/mysoftware/man/man-index/term.pos
/opt/mysoftware/man/man-index/term.exp
/opt/mysoftware/man/man-index/term.doc
/opt/mysoftware/man/man-index/.index-cache
/opt/mysoftware/man/man-index/term.idx
```

In addition to the binaries and man page, the system has also generated the man page indexes as a result of the actuator restarting the `man-index` service.

The `pkg info` command shows the metadata that was added to the package:

```
$ pkg info mypkg
          Name: mypkg
       Summary: This is an example package
   Description: This is a full description of all the interesting attributes of
                this example package.
      Category: Applications/Accessories
         State: Installed
     Publisher: mypublisher
       Version: 1.0
 Build Release: 5.11
        Branch: 0
Packaging Date: July 20, 2013 00:54:52 AM
          Size: 12.95 kB
          FMRI: pkg://mypublisher/mypkg@1.0,5.11-0:20130720T005452Z
```

The `pkg search` command returns hits when querying for files that are delivered by `mypkg`:

```
$ pkg search -l mycmd
INDEX      ACTION VALUE                    PACKAGE
basename   file   opt/mysoftware/bin/mycmd pkg:/mypkg@1.0-0
```

# Deliver the Package

IPS provides three different ways to deliver a package so that users can install the package:

**Local file-based repository**

Users access this repository over the local network. The publisher origin is the path to the repository, such as `/net/host1/export/ipsrepo`.

**Remote HTTP-based repository**

Users access this repository over HTTP or HTTPS. The publisher origin is an address such as `http://pkg.example.com/`.

**Package archive**

A package archive is a standalone file. The publisher origin is the path to the archive file, such as `/net/host1/export/ipsarchive.p5p`.

In each of these cases, the package was already published using the `pkgsend publish` command as described in "Publish the Package" on page 49. Use the `pkgrecv` command to retrieve the package to an existing repository or package archive for general use. See the `pkgrecv`(1) man page for more information. See *Copying and Creating Package Repositories in Oracle Solaris 11.3* for information about how to create and maintain a repository for general use.

## Deliver to a Package Repository

The following example shows how to deliver the new package from the test repository to a local file repository that has been set up for general use. The get and send sizes are zero because the package in this example is small.

```
$ pkgrecv -s my-repository -d /net/host1/export/ipsrepo mypkg
Processing packages for publisher mypublisher ...
Retrieving and evaluating 1 package(s)...
PROCESS                                         ITEMS   GET (MB)   SEND (MB)
Completed                                         1/1    0.0/0.0     0.0/0.0
```

Verify the presence of the package in the new repository:

```
$ pkgrepo info -s /net/host1/export/ipsrepo
PUBLISHER   PACKAGES STATUS           UPDATED
```

```
solaris     4455     online          2013-07-09T23:41:24.312974Z
mypublisher 1        online          2013-07-22T20:57:36.951042Z
$ pkgrepo list -p mypublisher -s /net/host1/export/ipsrepo
PUBLISHER   NAME                    O VERSION
mypublisher mypkg                     1.0,5.11-0:20130720T005452Z
```

Use the same `pkgrecv` command to deliver the package to an HTTP or HTTPS repository. In this case, specify the value of the `pkg/inst_root` property of the appropriate `pkg/server` service instance as the `-d` argument. This repository is served to users by the `svc:/application/pkg/server` service, which runs `pkg.depotd`. See the `pkg.depotd`(1M) man page for more information.

If this image has no child images (non-global zones), users can use the `-g` option to install the new package, as shown in the following command. The `-g` option adds the `mypublisher` publisher to the list of publishers configured in this image.

```
$ pkg install -g /net/host1/export/ipsrepo mypkg
```

If this image does have child images, users must configure the `mypublisher` publisher in the image, as shown in the following command.

```
$ pkg set-publisher -p /net/host1/export/ipsrepo
```

## Deliver as a Package Archive File

A package archive is a standalone file that contains publisher information and one or more packages provided by that publisher. Delivering packages as a package archive is convenient for users who cannot access your package repositories. Package archives can be easily downloaded from a web site, copied to a USB key, or burned to a DVD.

The `pkgrecv` command can add packages to package archives from package repositories or add packages to package repositories from package archives. When adding packages to a package repository from a package archive, note that a package archive does not contain repository configuration such as a default publisher prefix. Most `pkgrepo` subcommands do not work with package archives. The `pkgrepo list` command works with package archives.

The following command creates a package archive of the `mypkg` package. Because this archive does not yet exist, you must specify the `-a` option. By convention, package archives have the file extension `.p5p`.

```
$ pkgrecv -s my-repository -a -d myarchive.p5p mypkg
Retrieving packages for publisher mypublisher ...
Retrieving and evaluating 1 package(s)...
DOWNLOAD                                  PKGS       FILES   XFER (MB)   SPEED
Completed                                  1/1         3/3     0.0/0.0  782k/s

ARCHIVE                                              FILES   STORE (MB)
myarchive.p5p                                        14/14      0.0/0.0
```

If this image has no child images (non-global zones), users can use the `-g` option to install the new package, as shown in the following command. The `-g` option adds the `mypublisher` publisher to the list of publishers configured in this image.

```
$ pkg install -g myarchive.p5p mypkg
```

If this image does have child images, users must configure the `mypublisher` publisher in the image, as shown in the following command.

```
$ pkg set-publisher -p myarchive.p5p
```

Package archives can be set as sources of local publishers in non-global zones.

### Using Package Repositories and Archives

Use the `pkgrepo` command to list the newest available packages from a repository or archive:

```
$ pkgrepo list -s my-repository '*@latest'
PUBLISHER   NAME                                        O VERSION
mypublisher mypkg                                         1.0,5.11-0:20130720T005452Z
$ pkgrepo list -s myarchive.p5p '*@latest'
PUBLISHER   NAME                                        O VERSION
mypublisher mypkg                                         1.0,5.11-0:20130720T005452Z
```

This output can be useful for constructing scripts to create archives with the latest versions of all packages from a given repository.

# Converting SVR4 Packages To IPS Packages

This section shows an example of converting a SVR4 package to an IPS package and highlights areas that might need special attention.

To convert a SVR4 package to an IPS package, follow the same steps described in above in this chapter for packaging any software in IPS. Most of these steps are the same for conversion from SVR4 to IPS packages and are not explained again in this section. This section describes the steps that are different when converting a package rather than creating a new package.

## Generate an IPS Package Manifest from a SVR4 Package

The *source* argument of the `pkgsend generate` command can be a SVR4 package. See the `pkgsend(1)` man page for a complete list of supported sources. When *source* is a SVR4 package, pkgsend generate uses the pkgmap(4) file in that SVR4 package, rather than the directory inside the package that contains the files delivered.

While scanning the prototype file, the pkgsend utility also looks for entries that could cause problems when converting the package to IPS. The pkgsend utility reports those problems and prints the generated manifest.

The example SVR4 package used in this section has the following pkginfo(4) file:

```
VENDOR=My Software Inc.
HOTLINE=Please contact your local service provider
PKG=MSFTmypkg
ARCH=i386
DESC=A sample SVR4 package of My Sample Package
CATEGORY=system
NAME=My Sample Package
BASEDIR=/
VERSION=11.11,REV=2011.10.17.14.08
CLASSES=none manpage
PSTAMP=linn20111017132525
MSFT_DATA=Some extra package metadata
```

The example SVR4 package used in this section has the following corresponding prototype(4) file:

```
i pkginfo
i copyright
i postinstall
d none opt 0755 root bin
d none opt/mysoftware 0755 root bin
d none opt/mysoftware/lib 0755 root bin
f none opt/mysoftware/lib/mylib.so.1 0644 root bin
d none opt/mysoftware/bin 0755 root bin
f none opt/mysoftware/bin/mycmd 0755 root bin
d none opt/mysoftware/man 0755 root bin
d none opt/mysoftware/man/man1 0755 root bin
f none opt/mysoftware/man/man1/mycmd.1 0644 root bin
```

Running the pkgsend generate command on the SVR4 package built using these files generates the following IPS manifest:

```
$ pkgsend generate ./MSFTmypkg | pkgfmt
pkgsend generate: ERROR: script present in MSFTmypkg: postinstall

set name=pkg.summary value="My Sample Package"
set name=pkg.description value="A sample SVR4 package of My Sample Package"
set name=pkg.send.convert.msft-data value="Some extra package metadata"
dir path=opt owner=root group=bin mode=0755
dir path=opt/mysoftware owner=root group=bin mode=0755
dir path=opt/mysoftware/bin owner=root group=bin mode=0755
file reloc/opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0755
dir path=opt/mysoftware/lib owner=root group=bin mode=0755
file reloc/opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir path=opt/mysoftware/man owner=root group=bin mode=0755
dir path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file reloc/opt/mysoftware/man/man1/mycmd.1 \
    path=opt/mysoftware/man/man1/mycmd.1 owner=root group=bin mode=0644
legacy pkg=MSFTmypkg arch=i386 category=system \
    desc="A sample SVR4 package of My Sample Package" \
    hotline="Please contact your local service provider" \
    name="My Sample Package" vendor="My Software Inc." \
    version=11.11,REV=2011.10.17.14.08
license install/copyright license=MSFTmypkg.copyright
```

Note the following points regarding the pkgsend generate output:

- The pkg.summary and pkg.description attributes were automatically created from data in the pkginfo file.
- A set action was generated from the extra parameter in the pkginfo file. This set action is set beneath the pkg.send.convert.* namespace. Use pkgmogrify(1) transforms to convert such attributes to more appropriate attribute names.
- A legacy action was generated from data in the pkginfo file.
- A license action was generated that points to the copyright file used in the SVR4 package.
- An error message was emitted regarding a scripting operation that cannot be converted.

The following check shows the error message and the non-zero return code from pkgsend generate:

```
$ pkgsend generate MSFTmypkg > /dev/null
pkgsend generate: ERROR: script present in MSFTmypkg: postinstall
$ echo $?
1
```

The SVR4 package is using a postinstall script that cannot be converted directly to an IPS equivalent. The script must be manually inspected.

The postinstall script in the package has the following content:

```
#!/usr/bin/sh
catman -M /opt/mysoftware/man
```

You can achieve the same results as this script by using a restart_fmri actuator that points to an existing SMF service, svc:/application/man-index:default, as described in "Add Any Facets or Actuators That Are Needed" on page 47. See Chapter 7, "Automating System Change as Part of Package Installation" for a thorough discussion of actuators.

The pkgsend generate command also checks for the presence of class-action scripts and produces error messages that indicate which scripts should be examined.

In any conversion of a SVR4 package to an IPS package, the needed functionality probably can be implemented by using an existing action type or SMF service. See "Package Content: Actions" on page 21 for details about available action types. See Chapter 7, "Automating System Change as Part of Package Installation" for information about SMF and package actions.

Adding package metadata and resolving dependencies are done in the same way as described in "Creating and Publishing a Package" on page 40 and therefore are not discussed in this section. The next package creation step that might present unique issues for converted packages is the verification step.

## Verify the Converted Package

A common source of errors when converting SVR4 packages is mismatched attributes between directories delivered in the SVR4 package and the same directories delivered by IPS packages.

In the SVR4 package in this example, the directory action for `/opt` in the sample manifest has different attributes than the attributes defined for this directory by the system packages.

The "Directory Actions" on page 27 section stated that all reference-counted actions must have the same attributes. When trying to install the version of `mypkg` that has been generated so far, the following error occurs:

```
$ pkg install mypkg
Creating Plan /
pkg install: The requested change to the system attempts to install multiple actions
for dir 'opt' with conflicting attributes:

    1 package delivers 'dir group=bin mode=0755 owner=root path=opt':
        pkg://mypublisher/mypkg@1.0,5.11-0:20111017T020042Z
    1 package delivers 'dir group=sys mode=0755 owner=root path=opt':
        pkg://solaris/system/core-os@0.5.11,5.11-0.175.3.13.0.3.0:20160926T221733Z

These packages may not be installed together. Any non-conflicting set may
be, or the packages must be corrected before they can be installed.
```

To catch the error before publishing the package, rather than at install time, use the `pkglint(1)` command with a reference repository, as shown in the following example:

```
$ pkglint -c ./cache -r file:///scratch/solaris-repo ./mypkg.mf.res
Lint engine setup...

PHASE                                          ITEMS
4                                          4292/4292
Starting lint run...

ERROR pkglint.dupaction007        path opt is reference-counted but has different
 attributes across 5
```

```
duplicates: group: bin -> mypkg group: sys -> developer/build/onbld system/core-os
 system/ldoms/ldomsmanager
```

Notice the error message about `path opt` having different attributes in different packages.

The extra `ldomsmanager` package that `pkglint` reports is in the reference package repository, but is not installed on the test system. The `ldomsmanager` package is not listed in the error reported previously by `pkg install` because that package is not installed.

## Other Package Conversion Considerations

While it is possible to install SVR4 packages directly on an Oracle Solaris 11 system, you should create IPS packages instead. Installing SVR4 packages is an interim solution.

Apart from the `legacy` action described in "Legacy Actions" on page 34, no links exist between the two packaging systems, and SVR4 and IPS packages do not reference package metadata from each other.

IPS has commands such as `pkg verify` that can determine whether packaged content has been installed correctly. However, errors can result if another packaging system legitimately installs packages or runs install scripts that modify directories or files installed by IPS packages.

The IPS `pkg fix` and `pkg revert` commands can overwrite files delivered by SVR4 packages as well as by IPS packages, potentially causing the packaged applications to malfunction.

Commands such as `pkg install`, which normally check for duplicate actions and common attributes on reference-counted actions, might fail to detect potential errors when files from a different packaging system conflict.

With these potential errors in mind, and given the comprehensive package development tool chain in IPS, developing IPS packages instead of SVR4 packages is recommended for Oracle Solaris 11.

♦ ♦ ♦ CHAPTER 3

# Installing, Removing, and Updating Software Packages

This chapter describes how the IPS pkg client works internally when installing, updating, and removing the software installed in an image.

Understanding how pkg performs these operations is important for understanding the various errors that can occur and for more quickly resolving package dependency problems.

## How Package Changes Are Performed

The following steps are executed when pkg is invoked to modify the software installed in the image:

- Check the input for errors
- Determine the system end-state
- Run basic checks
- Run the solver
- Optimize the solver results
- Evaluate actions
- Download content
- Execute actions
- Process actuators

When executing these steps in the global zone, pkg can also operate on any non-global zones on the system. For example, pkg ensures that dependencies are correct between the global zone and non-global zones, and downloads content and executes actions as needed for non-global zones. Chapter 10, “Handling Non-Global Zones” discusses zones in detail.

## Check Input for Errors

Basic error checking is performed on the options presented on the command line.

## Determine the System End State

A description of the desired end state of the system is constructed. In the case of updating all packages in the image, the desired end state might be something like “all the packages currently installed, or newer versions of them.” In the case of package removal, the desired end state is “all the packages currently installed without this one.”

IPS attempts to determine what the user intends this end state to look like. In some cases, IPS might determine an end state that is not what the user intended, even though that end state does match what the user requested.

When troubleshooting, it is best to be as specific as possible. The following command is not specific:

```
$ pkg update
```

If this command fails with a message such as “No updates available for this image,” then you might want to try a more specific command such as the following command:

```
$ pkg update "*@latest"
```

This command defines the end state more precisely, and can produce more directed error messages.

## Run Basic Checks

The desired end state of the system is reviewed to make sure that a solution is possible. During this basic review, `pkg` checks that a plausible version exists of all dependencies, and that desired packages do not exclude each other.

If an obvious error exists, then `pkg` prints an appropriate error message and exits.

## Run the Solver

The solver forms the core of the computation engine used by `pkg(5)` to determine the packages that can be installed, updated, or removed, given the constraints in the image and constraints introduced by any new packages for installation.

This problem is an example of a *Boolean satisfiability problem*, and can be solved by a SAT solver.

The various possible choices for all the packages are assigned Boolean variables, and all the dependencies between those packages, any required packages, and so on, are cast as Boolean expressions in conjunctive normal form.

The set of expressions generated is passed to MiniSAT. If MiniSAT cannot find any solution, the error handling code attempts to walk the set of installed packages and the attempted operation and print the reasons that each possible choice was eliminated.

If the currently installed set of packages meets the requirements but no other set does, `pkg` reports that there is nothing to do.

As mentioned previously, the error message generation and specificity is determined by the inputs to `pkg`. Being as specific as possible in commands issued to `pkg` produces the most useful error messages.

If MiniSAT finds a possible solution, the optimization phase begins.

## Optimize the Solver Results

The optimization phase is necessary because there is no way to describe some solutions as more desirable than others to a SAT solver. Instead, once a solution is found, IPS adds constraints to the problem to separate less desirable choices, and to separate the current solution as well. IPS then repeatedly invokes MiniSAT and repeats the above operation until no more solutions are found. The last successful solution is taken as the best one.

The difficulty of finding a solution is proportional to the number of possible solutions. Being more specific about the desired result produces solutions more quickly.

Once the set of package FMRIs that best satisfy the posed problem is found, the evaluation phase begins.

## Evaluate Actions

In the evaluation phase, IPS compares the packages currently installed on the system with the end state, and compares package manifests of old and new packages to determine three lists:

- Actions that are being removed.
- Actions that are being added.
- Actions that are being updated.

The action lists are then updated in the following ways:

- Directory and link actions are reference counted, and mediated link processing is done.
- Hard links are marked for repair if their target file is updated. This is done because updating a target of a hard link in a manner that is safe for currently executing processes breaks the hard links.
- Editable files moving between packages are correctly handled so that any user edits are not lost.

- Action lists are sorted so that removals, additions, and updates occur in the correct order.

All currently installed packages are then cross-checked to make sure that no packages conflict. Example conflicts include two packages that deliver a file to the same location, or two packages that deliver the same directory with different directory attributes.

If conflicts exist, the conflicts are reported and `pkg` exits with an error message.

Finally, the action lists are scanned to determine whether any SMF services need to be restarted if this operation is performed, whether this change can be applied to a running system, whether the boot archive needs to be rebuilt, and whether the amount of space required is available.

## Download Content

If `pkg` is running without the `-n` flag, processing continues to the download phase.

For each action that requires content, IPS downloads any required files by hash and caches them. This step can take some time if the amount of content to be retrieved is large.

Once downloading is complete, if the change is to be applied to a live system (the image is rooted at `/`), and a reboot is required, the running system is cloned and the target image is switched to the clone.

## Execute Actions

Executing actions involves actually performing the install or remove methods specific to each action type on the image.

Execution begins with all the removal actions being executed. If any unexpected content is found in directories being removed from the system, that content is placed in `/var/pkg/lost+found`.

Execution then proceeds to install and update actions. Note that all the actions have been blended across all packages. Thus all the changes in a single package operation are applied to the system at once rather than package by package. This permits packages to depend on each other and exchange content safely. For details on how files are updated, see “File Actions” on page 23.

## Process Actuators

If the changes are being applied to a live system, any pending actuators are executed at this point. These are typically SMF service restarts and refreshes. Once these are launched, IPS updates the local search indexes. Actuators are discussed in detail in Chapter 7, “Automating System Change as Part of Package Installation”.

## Update Boot Archive

If necessary, the boot archive is updated.

♦ ♦ ♦ CHAPTER 4

# Specifying Package Dependencies

Dependencies define how packages are related. This chapter describes:

- Types of package dependencies. Dependency types were introduced in "Depend Actions" on page 31. This chapter provides more detail.
- How each dependency type can be used to control software installation. How to use dependencies and freezing to construct working software systems.

## Dependency Types

In IPS, a package cannot be installed unless all package dependencies are satisfied. IPS allows packages to be mutually dependent (to have circular dependencies). IPS also allows packages to have different kinds of dependencies on the same package at the same time.

Each section in this chapter contains an example depend action as it would appear in a manifest during package creation. Note that dependency specifications do not include the publisher name. See "Avoiding Conflicting Package Content" on page 95 for a discussion of possible conflicts.

### require Dependency

The most basic type of dependency is the require dependency. These dependencies are typically used to express functional dependencies such as libraries, or interpreters such as Python or Perl.

If a package pkg-a@1.0 contains a require dependency on package pkg-b@2, then if pkg-a@1.0 is installed, the pkg-b package at version 2 or higher must also be installed. This acceptance of higher versioned packages reflects the implicit expectation of binary compatibility in newer versions of existing packages.

If any version of the package named in the depend action is acceptable, you can omit the version portion of the specified FMRI.

An example require dependency is:

```
depend fmri=pkg:/system/library type=require
```

## require-any Dependency

The require-any dependency is used if any one of multiple target packages as specified by multiple fmri attributes can satisfy the dependency. IPS chooses one of the packages to install if the dependency is not already satisfied.

For example, you could use a require-any dependency to ensure that at least one version of Perl is installed on the system. The versioning is handled in the same manner as for the require dependency.

The order in which packages are listed in a require-any dependency has no effect on which package will be selected to satisfy the dependency. For example, the first package listed is not preferred over any other. The exception is that a package that is already installed and that will not be removed by some other part of the pkg operation is preferred over any package that is not already installed.

An example require-any dependency is:

```
depend type=require-any fmri=pkg:/editor/gnu-emacs/gnu-emacs-gtk \
    fmri=pkg:/editor/gnu-emacs/gnu-emacs-no-x11 \
    fmri=pkg:/editor/gnu-emacs/gnu-emacs-x11
```

## optional Dependency

The optional dependency specifies that if the given package is installed, it must be at the given version or greater.

This type of dependency is typically used to handle cases where packages transfer content. In this case, each version of the package post-transfer would contain an optional dependency on the post-transfer version of the other package, so that it would be impossible to install incompatible versions of the two packages. Omitting the version on an optional dependency makes the dependency meaningless, but is permitted.

An example optional dependency is:

```
depend fmri=pkg:/x11/server/xorg@1.9.99 type=optional
```

## conditional Dependency

The conditional dependency has a predicate attribute and an fmri attribute. If the package specified in the value of the predicate attribute is present on the system at the specified or greater version, the conditional dependency is treated as a require dependency on the package in the fmri attribute. If the package specified in the predicate attribute is not present on the system or is present at a lower version, the conditional dependency is ignored.

The conditional dependency is most often used to install optional extensions to a package if the requisite base packages are present on the system.

For example, an editor package that has both X11 and terminal versions might place the X11 version in a separate package, and include a conditional dependency on the X11 version from the text version with the existence of the requisite X client library package as the predicate.

In the following example conditional dependency, package version numbers are not needed because the named packages are already sufficiently version constrained:

```
depend fmri=library/python/pycurl-27 predicate=runtime/python-27 type=conditional
```

## group Dependency

The group dependency is used to construct groups of packages.

The group dependency ignores the version specified. Any version of the named package satisfies this dependency.

The named package is required unless the package has been the object of one of the following operations:

- The package has been placed on the avoid list. See the pkg(1) man page for information about the avoid list.
- No packages that match the fmri value are known.
- The package has been rejected with pkg install --reject.
- The package has been uninstalled with pkg uninstall.

These three options enable administrators to deselect packages that are the subject of a group dependency. If any of these three options has been used, IPS will not reinstall the package during an update unless the package was subsequently required by another dependency. If the new dependency is removed by another subsequent operation, then the package is uninstalled again.

Obsolete packages silently satisfy the group dependency, effectively ignoring the dependency.

A good example of how to use these dependencies is to construct packages containing group dependencies on packages that are needed for typical uses of a system. Some examples might be solaris-large-server, solaris-desktop, or developer-gnu. "Oracle Solaris Group Packages" on page 144 shows a set of Oracle Solaris packages that deliver group dependencies.

Installing group packages provides confidence that over subsequent updates to newer versions of the OS, the appropriate packages will be added to the system.

An example `group` dependency is:

```
depend fmri=package/pkg type=group
```

## `group-any` Dependency

The `group-any` dependency is used if any one of multiple target packages as specified by multiple `fmri` attributes can satisfy the dependency. IPS chooses one of the packages to install if the dependency is not already satisfied. The same rules apply to a `group-any` dependency that apply to a `group` dependency with the exception that non-obsolete package stems are preferred over obsolete package stems.

The order in which packages are listed in a `group-any` dependency has no effect on which package will be selected to satisfy the dependency. For example, the first package listed is not preferred over any other. The following selection preferences exist:

- A package that is already installed and that will not be removed by some other part of the `pkg` operation is preferred over any package that is not already installed.
- If some of the target packages are avoided, another target will be used to satisfy the dependency. If all target packages are avoided, the dependency is ignored.
- If some of the target packages are obsolete, another target will be used to satisfy the dependency. If all target packages are obsolete, the dependency is ignored.

An example `group` dependency is:

```
depend type=group-any \
    fmri=runtime/python-26 \
    fmri=runtime/python-27
```

## `origin` Dependency

The `origin` dependency exists to resolve upgrade issues that require intermediate transitions. The default behavior is to specify the minimum version of a package (if installed) that must be present on the image being updated. If the value of the `root-image` attribute is `true`, the package must be present on the image rooted at / in order to install this package.

For example, a typical use might be a database package version 5 that supports upgrade from version 3 or greater, but not earlier versions. In this case, version 5 would have an `origin` dependency on itself at version 3. Thus, if version 5 was being freshly installed, installation would proceed. However, if version 1 of the package was installed, the package could not be upgraded directly to version 5. In this case, `pkg update database-package` would not select version 5 but instead would select version 3 as the latest possible version to which to upgrade.

If the value of the `root-image` attribute is `true`, the dependency target must be at the specified version or greater if it is present in the running system, rather than on the image being updated. This form of the `origin` dependency is generally used for operating system issues such as dependencies on boot block installers.

An example `origin` dependency is:

```
depend fmri=pkg:/database/mydb@3.0 type=origin
```

## Device Driver with Manually Maintained Firmware

Device drivers should manage their own firmware: Firmware should be delivered in the driver package and should be updated when the administrator uses the `pkg update` command to update the driver. See "Firmware Compatibility" in *Writing Device Drivers for Oracle Solaris 11.2* for driver design information. Drivers also should continue to function with downrev firmware, even if some new features might not be supported.

A few drivers require manual intervention to update the device firmware, separate from running `pkg update` to update the driver. A few of these drivers with manually maintained firmware are not compatible with all older versions of the firmware and have a minimum version requirement for the firmware. The `origin` dependency can be used to prevent installation of a driver that is not compatible with the currently installed firmware, which can prevent a system upgrade that results in a system that is not fully functioning.

The `origin` dependency can be used to specify the minimum version of the device firmware that is compatible with the version of the driver that is being delivered. If the value of the `root-image` attribute is `true` and the value of the `fmri` attribute starts with `pkg:/feature/firmware/`, the remainder of the `fmri` value is treated as a command in `/usr/lib/fwenum` that evaluates the firmware dependency. When an administrator attempts to update a package that specifies this type of dependency and the firmware enumerator determines that the firmware dependency is not satisfied, an error message is displayed and the update is not performed: the system is not changed. The error message shows the firmware version that is required for devices managed by this driver. Once the firmware has been updated, the administrator can attempt the `pkg update` again.

The following is an example of an `origin` dependency with a minimum firmware version requirement:

```
depend fmri=pkg:/feature/firmware/mpt_sas minimum-version=1.0.0.0 \
root-image=true type=origin variant.opensolaris.zone=global
```

The `pkg` client invokes the firmware enumerator as shown in the following example:

```
/usr/lib/fwenum/mpt_sas minimum-version=1.0.0.0
```

The following sample message from the pkg client tells the administrator that two devices that are managed by the mpt_sas driver have firmware whose version does not satisfy the minimum requirement. The message also states that minimum required firmware version.

```
There are 2 instances of downrev firmware for the mpt_sas devices present on this
 system;
upgrade each to version 1.0.0.0 or greater to permit installation of this version of
 Solaris.
```

If a driver supports the same device from multiple vendors, the dependency can specify a vendor attribute in addition to the minimum-version attribute.

## incorporate Dependency

The incorporate dependency specifies that if the given package is installed, it must be at the given version, to the given version accuracy. For example, if the dependent FMRI has a version of 1.4.3, then no version less than 1.4.3 or greater than or equal to 1.4.4 satisfies the dependency. Version 1.4.3.7 does satisfy this example dependency.

The common way to use incorporate dependencies is to put many of them in the same package to define a surface in the package version space that is compatible. Packages that contain such sets of incorporate dependencies are often called incorporations. Incorporations are typically used to define sets of software packages that are built together and are not separately versioned. The incorporate dependency is heavily used in Oracle Solaris to ensure that compatible versions of software are installed together.

An example incorporate dependency is:

```
depend type=incorporate \
    fmri=pkg:/driver/network/ethernet/e1000g@0.5.11,5.11-0.175.0.0.0.2.1
```

## parent Dependency

The parent dependency is used for zones or other child images. In this case, the dependency is only checked in the child image, and specifies a package and version that must be present in the parent image or global zone. The version specified must match to the level of precision specified.

For example, if the parent dependency is on A@2.1, then any version of A beginning with 2.1 matches. This dependency is often used to require that packages are kept in sync between non-global zones and the global zone. As a shortcut, the special package name feature/package/dependency/self is used as a synonym for the exact version of the package that contains this dependency.

The parent dependency is used to keep key operating system components, such as libc.so.1, installed in the non-global zone synchronized with the kernel installed in the global zone. The parent dependency is also discussed in Chapter 10, “Handling Non-Global Zones”.

An example parent dependency is:

```
depend type=parent fmri=feature/package/dependency/self \
    variant.opensolaris.zone=nonglobal
```

### exclude Dependency

The package that contains the exclude dependency cannot be installed if the dependent package is installed in the image at the specified version level or greater.

If the version is omitted from the FMRI of an exclude dependency, then no version of the excluded package can be installed concurrently with the package specifying the dependency.

The exclude dependency is seldom used. These constraints can be frustrating to administrators, and should be avoided where possible.

An example exclude dependency is:

```
depend fmri=pkg:/x11/server/xorg@1.10.99 type=exclude
```

# Constraints and Freezing

By carefully using the dependency types described above, you can constrain how your packages are allowed to be upgraded.

- The incorporate dependency enables you to define a supported software surface that updates together.
- Freezing enables an administrator to keep the surface or other software at a particular version.
- Version lock facets enable an administrator to disable version constraints on some components of a surface.

## Constraining Installable Package Versions

Typically, you want a set of packages installed on a system to be supported and upgraded together: Either all packages in the set are updated, or none of the packages in the set is updated. To treat packages as a set in this way, use the incorporate dependency.

"Installing a Custom Incorporation" in *Adding and Updating Software in Oracle Solaris 11.3* shows an example of creating a custom incorporation to constrain the version of the pkg:/entire incorporation that can be installed. The remainder of this section is a more general discussion of incorporations.

The following three partial package manifests show the relationship between the pkg-a and pkg-b packages and the myincorp incorporation package.

The following excerpt is from the pkg-a package manifest:

```
set name=pkg.fmri value=pkg-a@1.0
dir path=opt/tool-a owner=root group=bin mode=0755
depend fmri=myincorp type=require
```

The following excerpt is from the pkg-b package manifest:

```
set name=pkg.fmri value=pkg-b@1.0
dir path=opt/tool-b owner=root group=bin mode=0755
depend fmri=myincorp type=require
```

The following excerpt is from the myincorp package manifest:

```
set name=pkg.fmri value=myincorp@1.0
depend fmri=pkg-a@1.0 type=incorporate
depend fmri=pkg-b@1.0 type=incorporate
```

The pkg-a and pkg-b packages both have a require dependency on the myincorp incorporation. The myincorp package has incorporate dependencies that constrain the pkg-a and pkg-b packages in the following ways:

- The pkg-a and pkg-b packages can be upgraded to at most version 1.0: to the level of granularity defined by the version number specified in the dependency.
- If the pkg-a and pkg-b packages are installed, they must be at least at version 1.0 or greater.

The incorporate dependency on version 1.0 allows version 1.0.1 or 1.0.2.1, for example, but does not allow version 1.1, 2.0, or 0.9, for example. When an updated incorporation package is installed that specifies incorporate dependencies at a higher version, the pkg-a and pkg-b packages are allowed to update to those higher versions.

Because pkg-a and pkg-b both have require dependencies on the myincorp package, the incorporation package is installed if either pkg-a or pkg-b is installed.

## Freezing Installable Package Versions

The previous section discussed constraints applied during the package authoring process by modifying the package manifests. The administrator can also apply constraints to the system at runtime.

Using the pkg freeze command, the administrator can prevent a given package from being changed from either its current installed version, including time stamp, or a version specified on the command line. This capability is effectively the same as an incorporate dependency.

See *Adding and Updating Software in Oracle Solaris 11.3* and the pkg(1) man page for more information about the freeze command.

To apply more complex dependencies to an image, create and install a package that includes those dependencies.

## Enabling Administrators to Relax Constraints on Installable Package Versions

An administrator might want to disable a dependency version constraint. You can provide version-lock facet tags to enable administrators to disable those tagged incorporate dependencies. The administrator can use the pkg change-facet command to set the value of the corresponding facet image property to false. For general information about facet tags, see Chapter 5, “Allowing Variations”.

Continuing the previous example, perhaps pkg-b can function independently of pkg-a, but you want pkg-a to remain within the series of versions defined by the incorporate dependency in the incorporation. The myincorp package manifest could contain the following lines, including a version-lock facet tag on the pkg-b dependency. By convention, version-lock facet tags are named facet.version-lock.*package-name*, where *package-name* is the name specified in the fmri of that depend action, without the version.

```
set name=pkg.fmri value=myincorp@1.0
depend fmri=pkg-a@1.0 type=incorporate
depend fmri=pkg-b@1.0 type=incorporate facet.version-lock.pkg-b=true
```

By default, this incorporation includes the depend action on the pkg-b package, constraining pkg-b to version 1.0. The following command relaxes this constraint:

```
$ pkg change-facet version-lock.pkg-b=false
```

After successful execution of this command, the pkg-b package is free from the incorporation constraints and can be upgraded to a higher version if necessary.

The following example specifies that this incorporation requires the java-8-incorporation package to be installed, and requires it at version 1.8.0.92.14-0. However, the specified facet.version-lock facet enables an administrator to attempt to install a different version.

```
depend fmri=consolidation/java-8/java-8-incorporation type=require
depend facet.version-lock.consolidation/java-8/java-8-incorporation=true \
    fmri=consolidation/java-8/java-8-incorporation@1.8.0.92.14-0 type=incorporate
```

Perhaps a higher version of the `java-8-incorporation` package would also work with this incorporation. The administrator can use the `pkg change-facet` command to set the `version-lock.consolidation/java-8/java-8-incorporation` facet property to `false` and then try to update the `java-8-incorporation` package separately from updating the incorporation.

♦ ♦ ♦ CHAPTER 5

# Allowing Variations

This chapter explains how to provide different installation options to the end user. Installation of mutually exclusive components is controlled by *variants*, and installation of optional components is controlled by *facets*.

Both variants and facets are comprised of the following two components:

- Tags set on actions in a package manifest
- A property set on the image

"Controlling Installation of Optional Components" in *Adding and Updating Software in Oracle Solaris 11.3* explains how variants and facets affect installation and how administrators can change the values of variant and facet properties in the image.

## Mutually Exclusive Software Components

Variants appear in the following two places in a package:

- A `set` action names the variant and defines the values that apply to this package.
- Any action that can only be installed for a subset of the variant values named in the `set` action has a tag that specifies the name of the variant and the value on which this action is installed.

One example of a mutually exclusive component is system architecture. Actions can contain multiple tags for different variant names. For example, a package might include both debug and nondebug binaries for both SPARC and x86.

A variant has two parts: its name, and the list of possible values. The `pkg variant -v` command displays all possible variant values that can be set for installed packages:

```
$ pkg variant -v
VARIANT                   VALUE
arch                      i386
arch                      sparc
debug.osnet               false
debug.osnet               true
opensolaris.zone          global
opensolaris.zone          nonglobal
```

IPS supports multiple architectures in a single package by specifying different variant tag values on actions for different architectures. Variant tags are applied to any actions that differ between architectures. Components that are delivered on both SPARC and x86 receive no variant tag. For example, a package that delivers the symbolic link `/var/ld/64` might include the following definitions:

```
set name=variant.arch value=sparc value=i386
dir group=bin mode=0755 owner=root path=var/ld
dir group=bin mode=0755 owner=root path=var/ld/amd64 variant.arch=i386
dir group=bin mode=0755 owner=root path=var/ld/sparcv9 variant.arch=sparc
link path=var/ld/32 target=.
link path=var/ld/64 target=sparcv9 variant.arch=sparc
link path=var/ld/64 target=amd64 variant.arch=i386
```

Another mutually exclusive component is global or non-global zones. Kernel components usually are not included in non-global zones. To prevent kernel components from being installed in a non-global zone, apply the `opensolaris.zone` variant tag to each of those actions with the value set to `global`.

Use `pkgmogrify` rules to apply variant tags in the manifest during publication. Using the `pkgmogrify` command is described in detail in Chapter 6, "Modifying Package Manifests Programmatically". Then use `pkgmerge` to merge packages from SPARC and x86 builds. The `pkgmerge` command merges across multiple different variants at the same time if needed. See the `pkgmogrify(1)` and `pkgmerge(1)` man pages for more information.

Unknown variant property values are `false` in the image by default. Therefore, if you want to introduce a new variant tag name, the values of that variant can only be `true` or `false`. Set the variant tag on the action to `true`, and inform administrators to use the `pkg change-variant` command to change the value of the variant property in the image to `true` to install that action.

Variants whose names start with `variant.debug.` are `false` in the image by default. You can provide debug versions of components and tag those components with custom `variant.debug.` variant tags.

**Note -** Variants are set per image. Select a variant name that is unique at the appropriate resolution for that piece of software.

## Optional Software Components

Some portions of your software that belong with the main body might be optional, and some users might not want to install them. Examples include localization files for different locales, man pages and other documentation, and header files needed only by developers or DTrace users.

Traditionally, optional content has been delivered in separate packages. Administrators installed optional content by installing these optional packages. One problem with this solution is that the administrator must discover optional packages to install by examining lists of available packages.

IPS uses facets to deliver optional package content. Facets are similar to variants: Each facet has a name and a value, and actions can contain multiple tags for different facet names.

- In the image, the default value for all facet properties that start with `facet.debug.` or `facet.optional.` is `false`. The default value for all other facet properties is `true`.
- On a packaged action, the value of a facet tag can be specified as either `true` or `all`.

  Actions that have a facet tag with a value of `all` are installed only if every facet on that action that has a value of `all` has a value of `true` in the image.

  Actions that have a facet tag with a value of `true` are installed if any facet on that action that has a value of `true` has a value of `true` in the image.

Use the `pkgmogrify` command to add facet tags to your package manifests, using regular expressions to match different types of files. Using the `pkgmogrify` command is described in detail in Chapter 6, "Modifying Package Manifests Programmatically".

To produce a list of facets that are available for you to set on your packaged actions, use the `pkg facet -a` command to display the values of all facets that are explicitly set in the image and all facets that are set in installed packages.

If you introduce a new facet in your software, whether that action will be installed depends on what name you choose for the facet as well as what other facets are set on that action. If your new facet is the only facet set on the action, the action will be installed as follows:

- If you name the facet `facet.debug.`*mysoftware* or `facet.optional.`*mycomponent*, the action will be installed only if the user sets the value of the facet image property to `true`.
- If you choose any other name for your new facet, the action will be installed unless the user sets the value of the facet image property to `false`.

In addition to specifying optional components, you can use `facet.version-lock.` facet tags to specify dependency version restrictions as described in "Constraints and Freezing" on page 73.

♦ ♦ ♦ CHAPTER 6

# Modifying Package Manifests Programmatically

This chapter explains how package manifests can be programmatically edited to automatically annotate and check the manifests.

Chapter 2, "Packaging Software With IPS" covers all the techniques that are necessary to publish a package. This chapter provides additional information that can help you perform the following tasks:

- Publish a large package
- Publish a large number of packages
- Republish packages over a period of time

Your package might contain many actions that need to be tagged with variants or facets as discussed in Chapter 5, "Allowing Variations", or that need to be tagged with service restarts as discussed in Chapter 7, "Automating System Change as Part of Package Installation". Rather than edit package manifests manually or write a script or program to do this work, use the IPS `pkgmogrify` utility to transform the package manifests quickly, accurately, and repeatably.

The `pkgmogrify` utility applies two types of rules:

- Transform rules modify actions.
- Include rules cause other files to be processed.

The `pkgmogrify` utility reads these rules from a file and applies them to the specified package manifest.

## Transform Rules

This section shows an example transform rule and describes the parts of all transform rules.

In Oracle Solaris, files delivering in a subdirectory named `kernel` are treated as kernel modules and are tagged as requiring a reboot. The following tag is applied to actions whose `path` attribute value includes `kernel`:

```
reboot-needed=true
```

To apply this tag, the following rule is specified in the `pkgmogrify` rule file:

```
<transform file path=.*kernel/.+ -> default reboot-needed true>
```

delimiters
: The rule is enclosed with `<` and `>`. The portion of the rule to the left of the `->` is the selection section or matching section. The portion to the right of the `->` is the execution section of the operation.

`transform`
: The type of the rule.

`file`
: Apply this rule only to `file` actions. This is called the selection section of the rule.

`path=.*kernel/.+`
: Transform only `file` actions with a `path` attribute that matches the regular expression `path=.*kernel/.+`. This is called the matching section of the rule.

`default`
: Add the attribute and value that follow `default` to any matching action that does not already have a value set for that attribute.

`reboot-needed`
: The attribute being set.

`true`
: The value of the attribute being set.

The selection or matching section of a transform rule can restrict by action type and by action attribute value. See the `pkgmogrify` man page for detail about how these matching rules work. Typical uses are for selecting actions that deliver to specified areas of the file system. For example, in the following rule, *operation* could be used to ensure that `usr/bin` and everything delivered inside `usr/bin` defaults to the correct user or group.

```
<transform file dir link hardlink path=usr/bin.* -> operation>
```

The `pkgmogrify(1)` man page describes the many operations that `pkgmogrify` can perform to add, remove, set, and edit action attributes as well as add and remove entire actions.

## Include Rules

Include rules enable transforms to be spread across multiple files and subsets reused by different manifests. Suppose you need to deliver two packages: `A` and `B`. Both packages should have their `source-url` set to the same URL, but only package `B` should have its files in `/etc` set to be `group=sys`.

The manifest for package `A` should specify an include rule that pulls in the file with the `source-url` transform. The manifest for package `B` should specify an include rule that pulls in the file containing the file group setting transform. Finally, an include rule that pulls in the file with the source-url transform should be added either to either package B or to the file with the transform that sets the group.

## Transform Order

Transforms are applied in the order in which they are encountered in a file. The ordering can be used to simplify the matching portions of transforms.

Suppose all files delivered in /foo should have a default group of sys, except those files delivered in /foo/bar, which should have a default group of bin.

You could write a complex regular expression that matches all paths that begin with /foo except for paths that begin with /foo/bar. Using the ordering of transforms makes this matching much simpler.

When ordering default transforms, always go from most specific to most general. Otherwise the latter rules will never be used.

For this example, use the following two rules:

```
<transform file path=foo/bar/.* -> default group bin>
<transform file path=foo/.* -> default group sys>
```

Using transforms to add an action using the matching described above would be difficult since you would need to find a pattern that matched each package delivered once and only once. The pkgmogrify tool creates synthetic actions to help with this issue. As pkgmogrify processes manifests, for each manifest that sets the pkg.fmri attribute, a synthetic pkg action is created by pkgmogrify. You can match against the pkg action as if it were actually in the manifest.

For example, suppose you wanted to add to every package an action containing the web site example.com, where the source code for the delivered software can be found. The following transform accomplishes that:

```
<transform pkg -> emit set info.source-url=http://example.com>
```

## Packaged Transforms

As a convenience to developers, a set of the transforms that were used when packaging the Oracle Solaris OS are available in the following files in /usr/share/pkg/transforms:

developer — Sets facet.devel on *.h header files delivered to /usr/.*/include, archive and lint libraries, pkg-config(1) data files, and autoconf(1) macros.

`documentation` Sets a variety of `facet.doc.*` facets on documentation files.

`locale` Sets a variety of `facet.locale.*` facets on files that are locale-specific.

`smf-manifests` Adds a `restart_fmri` actuator that points to the `svc:/system/manifest-import:default` on any packaged SMF manifests so that the system will import that manifest after the package is installed.

♦ ♦ ♦ CHAPTER 7

# Automating System Change as Part of Package Installation

This chapter explains how to use the Service Management Facility (SMF) to automatically handle any necessary system changes that should occur as a result of package installation. See *Managing System Services in Oracle Solaris 11.3* for more information about SMF services.

This chapter describes:

- How to use service actuators on a package action
- How to deliver an SMF service in an IPS package
- How to deliver a first boot service in an IPS package
- How to deliver an SMF service that assembles multiple files into one file, such as a configuration file, on package installation

## Specifying System Changes on Package Actions

First determine which actions should cause a change to the system when they are installed, updated, or removed. For example, some system changes are needed to implement the software self-assembly concept described in "Software Self-Assembly" on page 13.

For each of those package actions, determine which existing SMF service provides the necessary system change. Alternatively, write a new service that provides the needed functionality and ensure that service is delivered to the system as described in "Delivering an SMF Service" on page 86.

When you have determined the set of actions that should cause a change to the system when they are installed, tag those actions in the package manifest to cause that system change to occur. The value of a tag that causes system change to occur is called an *actuator*.

The following actuator tags can be added to any action in a manifest:

`reboot-needed`

This actuator takes the value `true` or `false`. This actuator declares that update or removal of the tagged action must be performed in a new boot environment if the package system is operating on a live image. Creation of a new boot environment is controlled by the `be-`

`policy` image property. See the “Image Properties” section in the `pkg`(1) man page for more information about the `be-policy` property.

SMF Actuators

These actuators are related to SMF services.

SMF actuators take a single service FMRI as a value, possibly including globbing characters to match multiple FMRIs. If the same service FMRI is tagged by multiple actions, possibly across multiple packages being operated on, IPS only triggers that actuator once.

The following list of SMF actuators describes the effect on the service FMRI that is the value of each named actuator. In these descriptions, “uninstalling the package” also includes moving the `file` action that delivers the service to a different package.

`disable_fmri`

Disable (`svcadm disable`) the specified service prior to uninstalling the package.

`refresh_fmri`

Refresh (`svcadm refresh`) the specified service after installing, updating, or uninstalling the package.

`restart_fmri`

Restart (`svcadm restart`) the specified service after installing, updating, or uninstalling the package.

`suspend_fmri`

Temporarily disable (`svcadm disable -t`) the specified service prior to installing the package, and then enable (`svcadm enable`) the service after installing the package.

These SMF actuators are not executed in the following cases:

- When operating on an alternate root (`pkg -R` */path/to/BE*).
- When recursing from the global zone (`pkg` *subcommand* `-r`).

## Delivering an SMF Service

To deliver a new SMF service, create a package that delivers the SMF manifest file and method script. On the manifest file action, include the following actuator to restart the manifest import service to re-read all service manifests on the system.

```
restart_fmri=svc:/system/manifest-import:default
```

This actuator ensures that when the manifest is added, updated, or removed, the `manifest-import` service is restarted, causing the service delivered by that SMF manifest to be added, updated, or removed.

The following example shows a complete file action with a restart_fmri attribute:

```
file lib/svc/manifest/network/network-ipmgmt.xml \
    path=lib/svc/manifest/network/network-ipmgmt.xml \
    group=sys mode=0444 owner=root \
    restart_fmri=svc:/system/manifest-import:default
```

If the package is added to a live system, this action is performed once all packages have been added to the system during that packaging operation. If the package is added to an alternate boot environment, this action is performed during the first boot of that boot environment.

If the environment where the package is installed has immutable non-global zones, a reboot is required to install new directories in the immutable zone. Immutable zones boot as far as the svc:/milestone/self-assembly-complete:default milestone in read/write mode, before rebooting read-only. "Delivering a Service that Runs Once" on page 87 shows how to make your service a dependency of the self-assembly-complete milestone service.

## Delivering a Service that Runs Once

This section shows an example of a package that delivers an SMF service that performs a one-time configuration.

The following package manifest delivers the run-once service:

```
set name=pkg.fmri value=myapp-run-once@1.0
set name=pkg.summary value="Deliver a service that runs once"
set name=pkg.description \
    value="This example package delivers a service that runs once. The service
is marked with a flag so that it will not run again."
set name=org.opensolaris.smf.fmri value=svc:/site/myapplication-run-once \
    value=svc:/site/myapplication-run-once:default
set name=variant.arch value=i386
file lib/svc/manifest/site/myapplication-run-once.xml \
    path=lib/svc/manifest/site/myapplication-run-once.xml owner=root group=sys \
mode=0444 restart_fmri=svc:/system/manifest-import:default
file lib/svc/method/myapplication-run-once.sh \
    path=lib/svc/method/myapplication-run-once.sh owner=root group=bin \
    mode=0755
depend fmri=pkg:/shell/ksh93@93.21.0.20110208,5.11-0.175.3.0.0.19.0 type=require
depend fmri=pkg:/system/core-os@0.5.11,5.11-0.175.3.0.0.19.0 type=require
```

The following script does the configuration work of the service. This method script uses a property, config/ran, that has been set in the service to ensure that the script runs only once. The property is set to one value in the service manifest and to another value in the method script. The comment in the exit call will be displayed by the svcs command.

```
#!/bin/sh
```

```
# Load SMF shell support definitions
. /lib/svc/share/smf_include.sh

# If nothing to do, exit with temporary disable.
ran=$(/usr/bin/svcprop -p config/ran $SMF_FMRI)
if [ "$ran" == "true" ] ; then
    smf_method_exit $SMF_EXIT_TEMP_DISABLE done "service ran"
fi

# Do the configuration work.

# Record that this run-once service has done its work.
svccfg -s $SMF_FMRI setprop config/ran = true
svccfg -s $SMF_FMRI refresh

smf_method_exit $SMF_EXIT_TEMP_DISABLE done "service ran"
```

The following listing shows the SMF service manifest for this example. Some features of this manifest are described following the listing.

```
<?xml version="1.0" ?>
<!DOCTYPE service_bundle SYSTEM '/usr/share/lib/xml/dtd/service_bundle.dtd.1'>
<service_bundle type="manifest" name="myapplication-run-once">
<service
    name='site/myapplication-run-once'
    type='service'
    version='1'>
    <dependency
        name='fs-local'
        grouping='require_all'
        restart_on='none'
        type='service'>
            <service_fmri value='svc:/system/filesystem/local:default' />
    </dependency>
    <dependent
        name='myapplication-run-once-complete'
        grouping='optional_all'
        restart_on='none'>
        <service_fmri value='svc:/milestone/self-assembly-complete' />
    </dependent>
    <instance enabled='true' name='default'>
        <exec_method
            type='method'
            name='start'
            exec='/lib/svc/method/myapplication-run-once.sh'
            timeout_seconds='60'/>
        <exec_method
            type='method'
            name='stop'
            exec=':true'
            timeout_seconds='0'/>
        <property_group name='startd' type='framework'>
```

```
            <propval name='duration' type='astring' value='transient' />
        </property_group>
        <property_group name='config' type='application'>
            <propval name='ran' type='boolean' value='false' />
        </property_group>
    </instance>
    <template>
         <common_name>
             <loctext xml:lang='C'>
                 Run-once service
             </loctext>
         </common_name>
         <description>
             <loctext xml:lang='C'>
                 This service checks and sets a property so that it runs
                 only once. This service is a dependency of the
                 self-assembly-complete milestone.
             </loctext>
         </description>
    </template>
</service>
</service_bundle>
```

- In the dependent element, this service adds itself as a dependency to the self-assembly-complete system milestone.
- This service has a startd/duration property set to transient so that svc.startd(1M) does not track processes for this service.
- This service has a config/ran property set to false. The service method sets this property to true so that the service will run only one time.
- This service has timeout_seconds set to 60 for the start method. If timeout_seconds is set to 0, SMF will wait indefinitely for the method script to exit.

Be sure to include a comment in the method script exit and a service name and description in the service template data to help users understand why this service runs only one time.

Make sure the service manifest is valid:

```
$ svccfg validate proto/lib/svc/manifest/site/myapplication-run-once.xml
```

Publish your package as described in "Publish the Package" on page 49.

Run pkg verify before and after installing the package. Compare the output of each run to ensure that the script does not attempt to modify any files that are not marked as editable.

After you install the package, check the following output:

- Use the svcs command to show the state of the service. Different options of the svcs command show additional information. The log file (-L) shows that the service method ran. Comments and service description explain why the service is disabled.

```
$ svcs myapplication-run-once
STATE          STIME    FMRI
disabled       16:10:26 svc:/site/myapplication-run-once:default
$ svcs -x myapplication-run-once
svc:/site/myapplication-run-once:default (Run-once service)
 State: disabled since March 30, 2015 04:10:26 PM PDT
Reason: Temporarily disabled by an administrator.
   See: http://support.oracle.com/msg/SMF-8000-1S
   See: /var/svc/log/site-myapplication-run-once:default.log
Impact: This service is not running.
$ svcs -l myapplication-run-once
fmri         svc:/site/myapplication-run-once:default
name         Run-once service
enabled      false (temporary)
state        disabled
next_state   none
state_time   March 30, 2015 04:10:26 PM PDT
logfile      /var/svc/log/site-myapplication-run-once:default.log
restarter    svc:/system/svc/restarter:default
manifest     /lib/svc/manifest/site/myapplication-run-once.xml
dependency   require_all/none svc:/system/filesystem/local:default (online)
$ svcs -xL myapplication-run-once
svc:/site/myapplication-run-once:default (Run-once service)
 State: disabled since March 30, 2015 04:10:26 PM PDT
Reason: Temporarily disabled by an administrator.
   See: http://support.oracle.com/msg/SMF-8000-1S
   See: /var/svc/log/site-myapplication-run-once:default.log
Impact: This service is not running.
   Log:
[ 2015 Mar 30 16:10:22 Enabled. ]
[ 2015 Mar 30 16:10:22 Rereading configuration. ]
[ 2015 Mar 30 16:10:25 Executing start method ("/lib/svc/method/myapplication-run-
once.sh"). ]
[ 2015 Mar 30 16:10:26 Method "start" exited with status 101. ]
[ 2015 Mar 30 16:10:26 "start" method requested temporary disable: "service ran" ]
   Use: 'svcs -Lv svc:/site/myapplication-run-once:default' to view the complete log.
```

- Use the `-d` option of the `svcs` command to show that the `myapplication-run-once` service is a dependency of the `self-assembly-complete` service.

```
$ svcs -d svc:/milestone/self-assembly-complete:default | grep once
disabled       16:37:20 svc:/site/myapplication-run-once:default
```

- Check the value of the property that is being used as a flag to prevent the service from running again.

```
$ svcprop -p config/ran myapplication-run-once
```

```
true
```

The following svccfg command shows that the value of the property was set to false in the service manifest and then later was reset to true.

```
$ svccfg -s myapplication-run-once:default listprop -l all config/ran
config/ran boolean     admin                 true
config/ran boolean     manifest              false
```

If you enable the service, you see that the “Rereading configuration” line is absent from the log file, and the service exited without re-doing the configuration work.

## Assembling a Custom File from Fragment Files

This section shows how to use an IPS package to deliver multiple files and deliver an SMF service that assembles these multiple files into one file.

The following package manifest delivers the self-assembly service. The isvapp-self-assembly service assembles the files inc1, inc2, and inc3 in the /opt/isvapp/config.d directory into the single /opt/isvapp/isvconf file.

```
set name=pkg.fmri value=isvappcfg@1.0
set name=pkg.summary value="Deliver isvapp config files and assembly service"
set name=pkg.description \
    value="This example package delivers a directory with fragment configuration
 files and a service to assemble them."
set name=org.opensolaris.smf.fmri value=svc:/site/isvapp-self-assembly \
    value=svc:/site/isvapp-self-assembly:default
set name=variant.arch value=i386
file lib/svc/manifest/site/isvapp-self-assembly.xml \
    path=lib/svc/manifest/site/isvapp-self-assembly.xml owner=root group=sys \
    mode=0444 restart_fmri=svc:/system/manifest-import:default
file lib/svc/method/isvapp-self-assembly.sh \
    path=lib/svc/method/isvapp-self-assembly.sh owner=root group=bin \
    mode=0755
dir  path=opt/isvapp owner=root group=bin mode=0755
dir  path=opt/isvapp/config.d owner=root group=bin mode=0755
file opt/isvapp/config.d/inc1 path=opt/isvapp/config.d/inc1 owner=root \
    group=bin mode=0644
file opt/isvapp/config.d/inc2 path=opt/isvapp/config.d/inc2 owner=root \
    group=bin mode=0644
file opt/isvapp/config.d/inc3 path=opt/isvapp/config.d/inc3 owner=root \
    group=bin mode=0644
file opt/isvapp/isvconf path=opt/isvapp/isvconf owner=root group=bin mode=0644
depend fmri=pkg:/shell/ksh93@93.21.0.20110208,5.11-0.175.3.0.0.19.0 type=require
depend fmri=pkg:/system/core-os@0.5.11,5.11-0.175.3.0.0.19.0 type=require
```

If you want to allow other packages to deliver configuration files with these same names, add overlay and preserve attributes to the files. See "Delivering a File That Is Also Delivered by Another Package" on page 110 for an example.

To reassemble the configuration file when new fragments of the configuration are installed, removed, or updated, add restart_fmri or refresh_fmri actuators to the configuration files. See "Apache Web Server Configuration" on page 15 for an example.

The following script does the configuration assembly for the service. The comment in the exit call will be displayed by the svcs command.

```
#!/bin/sh

# Load SMF shell support definitions
. /lib/svc/share/smf_include.sh

# If files exist in /opt/isvapp/config.d,
# and if /opt/isvapp/isvconf exists,
# merge all into /opt/isvapp/isvconf

# After this script runs, the service does not need to remain online.
smf_method_exit $SMF_EXIT_TEMP_DISABLE done "/opt/isvapp/isvconf assembled"
```

The following listing shows the SMF service manifest for this example. This manifest was created by using the svcbundle command and specifies the default dependency on the multi-user milestone service. You might want to change this dependency section as shown in "Delivering a Service that Runs Once" on page 87.

```
<?xml version="1.0" ?>
<!DOCTYPE service_bundle
  SYSTEM '/usr/share/lib/xml/dtd/service_bundle.dtd.1'>
<!--
    Manifest created by svcbundle (2015-Mar-30 13:22:37-0700)
-->
<service_bundle type="manifest" name="site/isvapp-self-assembly">
    <service version="1" type="service" name="site/isvapp-self-assembly">
        <!--
            The following dependency keeps us from starting until the
            multi-user milestone is reached.
        -->
        <dependency restart_on="none" type="service"
            name="multi_user_dependency" grouping="require_all">
            <service_fmri value="svc:/milestone/multi-user"/>
        </dependency>
        <exec_method timeout_seconds="60" type="method" name="start"
            exec="/lib/svc/method/isvapp-self-assembly.sh"/>
        <!--
            The exec attribute below can be changed to a command that SMF
            should execute to stop the service.  See smf_method(5) for more
            details.
        -->
        <exec_method timeout_seconds="60" type="method" name="stop"
            exec=":true"/>
        <!--
            The exec attribute below can be changed to a command that SMF
            should execute when the service is refreshed.  Services are
            typically refreshed when their properties are changed in the
            SMF repository.  See smf_method(5) for more details.  It is
            common to retain the value of :true which means that SMF will
            take no action when the service is refreshed.  Alternatively,
            you may wish to provide a method to reread the SMF repository
            and act on any configuration changes.
        -->
        <exec_method timeout_seconds="60" type="method" name="refresh"
            exec=":true"/>
        <property_group type="framework" name="startd">
            <propval type="astring" name="duration" value="transient"/>
        </property_group>
        <instance enabled="true" name="default"/>
        <template>
            <common_name>
                <loctext xml:lang="C">
                    ISV app self-assembly
                </loctext>
            </common_name>
            <description>
                <loctext xml:lang="C">
                    Assembly of configuration fragment files for ISV app.
                </loctext>
            </description>
        </template>
    </service>
</service_bundle>
```

Use the `svccfg validate` command to make sure the service manifest is valid.

After you publish and install the package, the `svcs` command shows that the `isvapp-self-assembly` service is temporarily disabled, and the log file contains the method exit comment that the file assembly is complete.

In contrast to the example shown in "Delivering a Service that Runs Once" on page 87, when you enable the `isvapp-self-assembly` service, the service start script runs again before the service is again disabled.

♦ ♦ ♦ CHAPTER 8

# Advanced Topics For Package Updating

This chapter discusses the following topics:

- Avoiding conflicting package content
- Modifying package content
- Renaming, merging, and splitting packages
- Obsoleting packages
- Preserving files that move or that are not packaged
- Sharing information across boot environments
- Delivering multiple implementations of an application

## Avoiding Conflicting Package Content

In general, packages should not deliver multiple actions with the same path. The following actions are the only exceptions:

- Directories, links, and hardlinks that have identical attribute values
- Mediated links and hardlinks

For the best user experience, avoid installation and update errors by using the `pkglint` utility to check for conflicts. See "Verify the Package" on page 48 and the `pkglint`(1) man page for more information about `pkglint`.

When a single package must deliver different versions of the same content to the same path, use a variant to allow the administrator to choose between the different versions of content to install. See "Mutually Exclusive Software Components" on page 77 and the `pkg`(7) man page for more information.

When multiple packages (not two versions of the same package) must deliver different versions of the same content to the same path, use a mediated link to enable the administrator to choose between different versions of that content. Conflicting content must be installed in different parent directories, and a link created in the target location that points to each version of the content. The administrator can switch between different content by using the `pkg set-mediator` command to change the target of the link. See "Delivering Multiple Implementations of an Application" on page 112 for a description of how to mediate conflicting package content.

**Note -** Mediation is only allowed when both packages deliver an action of the same type, and only for `link` and `hardlink` actions. In addition, both packages must implement the content mediation. If one package delivers `/opt/tool` as a link and another package delivers `/opt/tool` as a link but without the mediation, a conflict will still be present and users will receive error messages and may be unable to install both packages.

If two different packages deliver some of the same content but only one of the packages should be installed, make sure the packages have different names. If publisher `example.com` delivers some of the same content as publisher `solaris`, the end user might be able to avoid conflicts by specifying the full package name, including publisher, in the installation command. However, problems could still arise with dependencies. The `fmri` attribute of a `depend` action specifies the full package name except for the publisher. The following dependency matches both `pkg://solaris/cat/subcat/tool` and `pkg://example.com/cat/subcat/tool`:

```
depend fmri=pkg:/cat/subcat/tool type=require
```

To differentiate these dependencies to install the correct package, change the package name. The following suggestions change the package name but still keep the name `tool`:

```
pkg://example.com/cat/subcat/example.com/tool
pkg://example.com/cat/subcat/example.com,tool
pkg://example.com/cat/subcat/vendor/example.com/tool
```

## Modifying Package Content

Package manifests represent the complete content of the package.

- If you remove a file or other action from a package manifest, that action is removed from the image when the package is updated by using the `pkg` command.
- If you rename a file or other action in a package manifest, the new action is installed in the image and the old action is removed from the image when the package is updated by using the `pkg` command. Review "Avoiding Conflicting Package Content" on page 95 before you rename actions.

## Renaming, Merging, and Splitting Packages

The desired organization of a software component can change because of mistakes in the original packages, changes in the product or its usage over time, or changes in the surrounding software environment. Sometimes just the name of a package needs to change. When planning such changes, consider the user who is performing an upgrade, to ensure that unintended side effects do not occur.

Three types of package reorganization are discussed in this section, in order of increasingly complex considerations for `pkg update`:

1. Renaming single packages
2. Merging two packages
3. Splitting a package

## Renaming a Single Package

Renaming a single package is straightforward. IPS provides a mechanism to indicate that a package has been renamed.

To rename a package, publish a new version of the existing package with no content and with the following two actions:

- A `set` action in the following form:

  ```
  set name=pkg.renamed value=true
  ```
- A `require` dependency on the new package.

  ```
  depend fmri=pkg:/newpkgname@version type=require
  ```

A renamed package cannot deliver content other than `depend` or `set` actions.

The new package must ensure that it cannot be installed at the same time as the original package before the rename. If both packages are covered by the same incorporation dependency, this restriction is automatic. If not, the new package must contain an `optional` dependency on the old package at the renamed version. This ensures that the solver will not select both packages, which would fail conflict checking.

A user who installs this renamed package automatically receives the new named package, since it is a dependency of the old version. If a renamed package is not depended upon by any other packages, it is automatically removed from the system. The presence of older software can cause a number of renamed packages to be shown as installed. When that older software is removed, the renamed packages are automatically removed as well.

Packages can be renamed multiple times without issue, though this is not recommended since it can be confusing to users.

## Merging Two Packages

Merging packages is straightforward as well. The following two cases are examples of merging packages:

- One package absorbs another package at the renamed version.

  Suppose package A@2 must absorb package B@3. To do this, rename package B to package A@2. Remember to include an optional dependency in A@2 on B@3, unless both packages are incorporated so that they update together as described above. A user upgrading B to B@3 now gets A installed since A has absorbed B.
- Two packages are renamed to the same new package name.

  In this case, rename both packages to the name of the new merged package, including two optional dependencies on the old packages in the new one if they are not otherwise constrained.

## Splitting a Package

When you split a package, rename each resulting new package as described in "Renaming a Single Package" on page 97. If one of the resulting new packages is not renamed, the pre-split and post-split versions of that package are not compatible and might violate dependency logic when the end user tries to update the package.

Rename the original package, and include require dependencies on all new packages that resulted from the split. This ensures that any package that had a dependency on the original package will get all the new pieces.

Some components of the split package can be absorbed into existing packages as a merge. See "How to Enable Your Application to Use a Shared Area" on page 104.

# Obsoleting Packages

Package obsoletion is the mechanism by which packages are emptied of contents and removed from the system.

A package is made obsolete by publishing a new version with no content and with the following set action. An obsoleted package cannot deliver content other than set actions.

```
set name=pkg.obsolete value=true
```

If the package being obsoleted was previously renamed, you must also obsolete those renamed packages and remove their rename dependencies. A package cannot be marked both renamed and obsolete. In the renamed package, change pkg.renamed to pkg.obsolete and remove the depend action that specifies the package to which this package was renamed. See "Renaming a Single Package" on page 97 for a reminder of what was done to rename the package.

An obsoleted package does not satisfy require dependencies. Update fails if an installed package has a require dependency on a package that is obsoleted in the update, unless the update also provides a newer version of the dependent package that no longer contains the `require` dependency on the obsolete package.

An obsolete package can be made non-obsolete by publishing a newer version that is not marked obsolete. If a user performs an update when an obsolete package is installed, the obsolete package is removed from the system. If a user performs an update before the package was obsolete and does not update again until after a newer, non-obsolete version of the package is published, the update installs that newer version.

## Preserving Packaged Editable Files that Migrate

Packaged editable files might need to move between packages or change location in the installed file system.

- Migrating editable files between packages.

  IPS attempts to migrate editable files that move between packages if the file name and file path have not changed. Renaming a package is an example of moving files between packages.
- Migrating editable files in the file system.

  If the file path changes, ensure the `original_name` attribute is assigned to preserve the user's customizations of the file.

  If the `file` action in the package that originally delivered this file does not contain the `original_name` attribute, add that attribute in the updated package. Set the value of the attribute to the name of the originating package, followed by a colon and the original path to the file without a leading `/`.

  Once the `original_name` attribute is present on an editable file, do not change the attribute value. This value acts as a unique identifier for all moves going forward so that the user's content is properly preserved regardless of the number of versions skipped on an update.

## Preserving Unpackaged Files

By default, unpackaged content is automatically salvaged to `/var/pkg/lost+found` when the containing directory is no longer referenced by any installed package. This section shows how to implement the following alternatives:

- Move the unpackaged content to a new packaged location
- Keep the unpackaged content where it is, even though all packaged content is uninstalled from that area

## Moving Unpackaged Files on Directory Removal

This example shows how to use IPS to salvage unpackaged content to another packaged directory.

In this example, the package myapp@1.0 installs the directory /opt/myapp/logfiles. The myapp application writes log files to that directory.

The myapp@2.0 package delivers the /opt/myapp/history directory and does not deliver the /opt/myapp/logfiles directory. Users who update their installed myapp@1.0 package to myapp@2.0 will no longer have an /opt/myapp/logfiles directory. These users will see a message at the end of their pkg update output telling them that content from /opt/myapp/logfiles has been saved in /var/pkg/lost+found/opt/myapp/logfiles.

To use IPS to move the file content from /opt/myapp/logfiles to /opt/myapp/history at the time the myapp package is updated, use the salvage-from attribute on the /opt/myapp/history directory. Your pkgmogrify input file needs the following entry:

```
<transform dir path=opt/myapp/history -> \
    add salvage-from /opt/myapp/logfiles>
```

After you run pkgmogrify, your package manifest action for this directory will look like the following:

```
dir  path=opt/myapp/history owner=root group=bin mode=0755 \
    salvage-from=/opt/myapp/logfiles
```

After a user runs pkg update myapp, the /opt/myapp/logfiles directory is gone, the new /opt/myapp/history directory is installed, and the file content from /opt/myapp/logfiles is in /opt/myapp/history.

See "How to Migrate Unshared Content to a Shared Area" on page 108 shows another example that uses the salvage-from attribute.

## Packaging the Directory Separately

To keep the unpackaged content where it is, even though all packaged content is uninstalled from that directory, package and install the directory separately. The directory remains installed as long as the package that installed the directory remains installed, even if all other packaged content is uninstalled from that directory.

For example, if you perform the following steps, and if no other installed IPS package delivers content to *dir*, the content of the *dir* directory is salvaged to /var/pkg/lost+found, including the application that was not delivered by IPS:

1. Install an application that is not delivered as an IPS package into *dir*.
2. Install an IPS package that installs content into *dir*.

3. Uninstall the IPS package that installs content into *dir*.

To maintain the unpackaged software you installed in *dir*, package the *dir* directory in its own IPS package.

Create an IPS package that delivers the directory or directory structure that you want. Install that package. That directory structure remains in place until you uninstall that package. Uninstalling a different package that delivers content to that directory will not remove the directory.

You should not create a package that delivers a directory that IPS already delivers. If an update would install the directory with different ownership, permissions, or other attributes, the update might not succeed. See the `pkgmogrify` step in the following procedure.

## ▼ How to Preserve a Directory After Content Uninstall

1. **Create the directory structure you want to deliver.**

   This example shows `/usr/local`. You can easily expand this to include `/usr/local/bin` or different directory structures that are not delivered by IPS packages.

   ```
   $ mkdir -p usrlocal/usr/local
   ```

2. **Create the initial package manifest.**

   ```
   $ pkgsend generate usrlocal | pkgfmt > usrlocal.p5m.1
   $ cat usrlocal.p5m.1
   dir path=usr owner=root group=bin mode=0755
   dir path=usr/local owner=root group=bin mode=0755
   ```

3. **Exclude directories already delivered by IPS.**

   Create a `pkgmogrify` input file to add metadata and to exclude delivering `/usr` since that directory is already delivered by Oracle Solaris. You might also want to add transforms to change directory ownership or permissions from the default.

   ```
   $ cat usrlocal.mog
   set name=pkg.fmri value=pkg://site/usrlocal@1.0
   set name=pkg.summary value="Create the /usr/local directory."
   set name=pkg.description value="This package installs the /usr/local \
   directory so that /usr/local remains available for unpackaged files."
   set name=variant.arch value=$(ARCH)
   <transform dir path=usr$->drop>
   ```

4. **Apply the changes to the initial manifest.**

   ```
   $ pkgmogrify -DARCH=`uname -p` usrlocal.p5m.1 usrlocal.mog | \
   pkgfmt > usrlocal.p5m.2
   $ cat usrlocal.p5m.2
   set name=pkg.fmri value=pkg://site/usrlocal@1.0
   ```

```
set name=pkg.summary value="Create the /usr/local directory."
set name=pkg.description \
    value="This package installs the /usr/local directory so that /usr/local
remains available for unpackaged files."
set name=variant.arch value=i386
dir path=usr/local owner=root group=bin mode=0755
```

5. **Check your work.**

```
$ pkglint usrlocal.p5m.2
Lint engine setup...
Starting lint run...
$
```

6. **Publish the package to your repository.**

   In this example, the default publisher for the `yourlocalrepo` repository has already been set to `site`.

```
$ pkgsend -s yourlocalrepo publish -d usrlocal usrlocal.p5m.2
pkg://site/usrlocal@1.0,5.11:20140303T180555Z
PUBLISHED
```

7. **Make sure you can see the new package that you want to install.**

```
$ pkg refresh site
$ pkg list -a usrlocal
NAME (PUBLISHER)      VERSION      IFO
usrlocal (site)       1.0          ---
```

8. **Install the package.**

```
$ pkg install -v usrlocal
           Packages to install:         1
     Estimated space available:  20.66 GB
Estimated space to be consumed: 454.42 MB
       Create boot environment:        No
Create backup boot environment:        No
          Rebuild boot archive:        No

Changed packages:
site
  usrlocal
    None -> 1.0,5.11:20140303T180555Z
PHASE                                          ITEMS
Installing new actions                           5/5
Updating package state database                 Done
Updating package cache                           0/0
Updating image state                            Done
Creating fast lookup database                   Done
Reading search index                            Done
Updating search index                            1/1
```

9. **Make sure the package is installed.**

```
$ pkg list usrlocal
NAME (PUBLISHER)      VERSION      IFO
usrlocal (site)       1.0          i--
$ pkg info usrlocal
          Name: usrlocal
       Summary: Create the /usr/local directory.
   Description: This package installs the /usr/local directory so that
                /usr/local remains available for unpackaged files.
         State: Installed
     Publisher: site
       Version: 1.0
 Build Release: 5.11
        Branch: None
Packaging Date: March  3, 2014 06:05:55 PM
          Size: 0.00 B
          FMRI: pkg://site/usrlocal@1.0,5.11:20140303T180555Z
$ ls -ld /usr/local
drwxr-xr-x   2 root     bin            2 Mar  3 10:17 /usr/local/
```

# Sharing Content Across Boot Environments

IPS package content can only be installed into file systems that are part of a BE. For example, on a default Oracle Solaris 11 installation, only datasets under rpool/ROOT/*BEname*/ are supported for package operations. Using IPS to directly deliver content that is outside any BE can result in a system that is no longer able to boot or clone older BEs.

Some content is shared across BEs, as described in "Existing Shared Content in Oracle Solaris" on page 103.

To deliver packaged content to a shared area, use a link as described in "Delivering Content to a Shared Area" on page 104.

## Existing Shared Content in Oracle Solaris

Some files must be shared across BEs to preserve normal system operation in an environment with multiple BEs. The following directories are already shared across BEs by IPS:

```
/var/audit
/var/cores
/var/crash
/var/mail
```

Within each BE, these directories are symbolic links to the following shared directories:

```
/var/share/audit
```

```
/var/share/cores
/var/share/crash
/var/share/mail
```

These shared directories are in the `VARSHARE` dataset, which is a shared dataset mounted at `/var/share`.

Other data that needs to be shared across BEs can be handled similarly.

# Delivering Content to a Shared Area

To share data across BEs, use a shared dataset and a symbolic link from a directory structure inside the BE pointing into that shared dataset. An IPS package delivers a symbolic link inside the BE. The same package or another package delivers an SMF service that creates and mounts the shared dataset. The link from the BE where the package is installed into the new shared dataset is similar to the links to `/var/share` shown in "Existing Shared Content in Oracle Solaris" on page 103. Applications running in the BE write to and read from the shared area via the link.

Best practice is to create one dataset where many applications running in different BEs can share content. Creating a separate dataset for each directory of data that you want to share results in creating many datasets in each non-global zone, and creating many datasets per zone is not desirable. For example, you could create an `OPTSHARE` dataset mounted at `/opt/share`. Different applications could share data in different directories under `/opt/share`.

While you could use a separate package and service to create the shared dataset, note that such a package could be installed from a different BE. The shared dataset might be available even though the package and service that created the dataset are not installed in the current BE. The service delivered by the application package shown in these examples checks whether the dataset already exists and creates the dataset if it does not already exist.

## How to Enable Your Application to Use a Shared Area

This procedure shows how to provide a shared dataset and a link from the current BE into the shared dataset to enable this application to share data with multiple applications in multiple BEs. The application package delivers the following:

- An SMF service that creates the shared dataset
- A link in the current BE whose target is in the shared dataset

The package in this example only shows the actions needed to create and link to the shared dataset. Other actions for the application, such as executables and configuration files, are omitted for this example.

1. **Create a package development area.**

Create an area for your package development that contains the directories that you need in the BE and a link to the shared area outside the BE.

a. **Create the structure needed to deliver the service manifest and start method that will create the shared dataset.**

```
$ mkdir -p proto/lib/svc/manifest/site
$ mkdir -p proto/lib/svc/method
```

b. **Deliver a link that applications can use to access the shared dataset.**

```
$ mkdir -p proto/opt/myapp
$ ln -s ../../opt/share/myapp/logfiles proto/opt/myapp/logfiles
```

2. **Create the service start method.**

In `proto/lib/svc/method`, create a script that performs the following tasks:

- Create the dataset, `rpool/OPTSHARE`, that is shared across BEs. Creating the shared dataset needs to be done only one time for each pool. All current and future BEs in the pool can access that dataset. Check whether the dataset already exists before you create it.
- Create the directory structure you want in the shared dataset, including the target of the link in this example: `/opt/share/myapp/logfiles`.

This script is the start method of the service. In this example, the script is named `myapp-share-files.sh`. You will need this file name when you create the service manifest in the next step.

The script needs the elements shown in the following prototype. Recall that by default, `sh` is `ksh93`. The `smf_include.sh` file is needed for `smf_method_exit`. The third argument to `smf_method_exit` will appear in the service log file and in output from the `svcs` command.

```
#!/bin/sh

# Load SMF shell support definitions
. /lib/svc/share/smf_include.sh

# Create rpool/OPTSHARE with mount point /opt/share if it does not already exist
# Create /opt/share/myapp/logfiles if it does not already exist

# After this script runs, the service does not need to remain online.
smf_method_exit $SMF_EXIT_TEMP_DISABLE done "shared area created"
```

3. **Create the service manifest.**

In `proto/lib/svc/manifest/site`, use the `svcbundle` command to create the service. In the `service-name`, include the category `site` because this is an internal service. The value of `start-method` is the name of the script that was created in the previous step.

```
$ svcbundle -s service-name=site/myapp-share-files \
-s start-method=/lib/svc/method/myapp-share-files.sh -o myapp-share-files.xml
```

Edit the resulting service manifest to add `common_name` and `description` information in the `template` data area. You can also add `documentation` and other template data. You might also want to change some of the default settings, such as the milestone dependency or the timeout for the start method. By default, the instance that is created is named `default` and is enabled.

Make sure the service manifest is valid:

```
$ svccfg validate myapp-share-files.xml
```

4. **Generate the initial package manifest.**

   Use the `pkgsend generate` command to create the initial package manifest from your package development area.

```
$ pkgsend generate proto | pkgfmt > share.p5m.1
$ cat share.p5m.1
dir  path=lib owner=root group=bin mode=0755
dir  path=lib/svc owner=root group=bin mode=0755
dir  path=lib/svc/manifest owner=root group=bin mode=0755
dir  path=lib/svc/manifest/site owner=root group=bin mode=0755
file lib/svc/manifest/site/myapp-share-files.xml \
    path=lib/svc/manifest/site/myapp-share-files.xml owner=root group=bin \
    mode=0644
dir  path=lib/svc/method owner=root group=bin mode=0755
file lib/svc/method/myapp-share-files.sh \
    path=lib/svc/method/myapp-share-files.sh owner=root group=bin mode=0755
dir  path=opt owner=root group=bin mode=0755
dir  path=opt/myapp owner=root group=bin mode=0755
link path=opt/myapp/logfiles target=../../opt/share/myapp/logfiles
```

5. **Add metadata and actuators.**

   a. **Create the following `pkgmogrify` input file named `share.mog`.**

   - Give the package a name, version, summary, and description.
   - Delete the `opt`, `lib/svc/manifest/site`, and `lib/svc/method` directory actions because these directories are already delivered by other packages.
   - Change the group for the service manifest to `sys` to match other manifests in `/lib/svc/manifest`.
   - Change the mode of the service manifest to `0444` and the mode of the service method to `0555` to match other manifests and methods on the system.
   - Add actuators for the service manifest and method files to restart the `manifest-import` service whenever those files are installed or updated.

```
set name=pkg.fmri value=myapp@2.0
set name=pkg.summary value="Deliver shared directory"
set name=pkg.description value="This example package delivers a directory \
and link that allows myapp content to be shared across BEs."
```

```
set name=variant.arch value=$(ARCH)
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
<transform dir path=opt$->drop>
<transform dir path=lib$->drop>
<transform dir path=lib/svc$->drop>
<transform dir path=lib/svc/manifest$->drop>
<transform dir path=lib/svc/manifest/site$->drop>
<transform dir path=lib/svc/method$->drop>
<transform file path=lib/svc/manifest/site/myapp-share-files.xml -> \
    edit group bin sys>
<transform file path=lib/svc/manifest/site/myapp-share-files.xml -> \
    edit mode 0644 0444>
<transform file path=lib/svc/manifest/site/myapp-share-dir.xml -> \
    add restart_fmri svc:/system/manifest-import:default>
<transform file path=lib/svc/method/myapp-share-files.sh -> \
    edit mode 0755 0555>
<transform file path=lib/svc/method/myapp-share-dir.sh -> \
    add restart_fmri svc:/system/manifest-import:default>
```

b. **Run `pkgmogrify` on the `share.p5m.1` manifest with the `share.mog` changes.**

```
$ pkgmogrify -DARCH=`uname -p` share.p5m.1 share.mog | pkgfmt > share.p5m.2
$ cat share.p5m.2
set name=pkg.fmri value=myapp@2.0
set name=pkg.summary value="Deliver shared directory"
set name=pkg.description \
    value="This example package delivers a directory and link that allows myapp
content to be shared across BEs."
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
set name=variant.arch value=i386
file lib/svc/manifest/site/myapp-share-files.xml \
    path=lib/svc/manifest/site/myapp-share-files.xml owner=root group=sys \
    mode=0444
file lib/svc/method/myapp-share-files.sh \
    path=lib/svc/method/myapp-share-files.sh owner=root group=bin mode=0755
dir  path=opt/myapp owner=root group=bin mode=0555
link path=opt/myapp/logfiles target=../../opt/share/myapp/logfiles
```

6. **Evaluate and resolve package dependencies.**

Use the `pkgdepend` command to automatically generate and resolve dependencies for the package. The output from resolving dependencies is automatically stored in a file with suffix `.res`.

In this example, dependencies are generated for `ksh93`. Additional dependencies might be generated, depending on what your start method does. Also, the service and service instance are declared in `org.opensolaris.smf.fmri`.

```
$ pkgdepend generate -md proto share.p5m.2 | pkgfmt > share.p5m.3
$ pkgdepend resolve -m share.p5m.3
```

The following lines are added in the output `share.p5m.3.res` file:

```
set name=org.opensolaris.smf.fmri value=svc:/site/myapp-share-files \
    value=svc:/site/myapp-share-files:default

depend fmri=pkg:/shell/ksh93@93.21.0.20110208-0.175.2.0.0.37.1 type=require
depend fmri=pkg:/system/core-os@0.5.11-0.175.2.0.0.37.0 type=require
```

7. **Verify the package.**

```
$ pkglint share.p5m.3.res
```

8. **Publish the package.**

```
$ pkgsend -s site publish -d protosl share.p5m.3.res
pkg://site/myapp@2.0,5.11:20140417T000014Z
PUBLISHED
```

9. **Test the package.**

Install the package.

```
$ pkg install -g ./site myapp
```

Verify that the dataset and link exist.

```
$ zfs list rpool/OPTSHARE
NAME            USED  AVAIL  REFER  MOUNTPOINT
rpool/OPTSHARE  38.5K  24.8G  38.5K  /opt/share
$ ls -l /opt/myapp
lrwxrwxrwx   1 root   root   21 Apr 16 17:32 logfiles -> /opt/share/myapp/logfiles
```

Uninstall the package. The `/opt/myapp/logfiles` link should be gone, and the service manifest and method script should be gone. The `rpool/OPTSHARE` dataset should still exist because that is not packaged content: It was created by the service.

## How to Migrate Unshared Content to a Shared Area

This procedure extends the previous procedure. In this example, some data that needs to be shared already exists. The application package delivers a staging area and copies the data to be shared to the staging area. The SMF service moves the data from the staging area to the shared area.

- In addition to the link, the package delivers a staging area in the BE to save any unpackaged content that already exists in the directory that will be redefined to be a link.
- In addition to creating the shared dataset, the SMF service moves any content that exists in the staging area to the shared area.

1. **Create a package development area.**

Change all occurrences of `logfiles` to `logs`.

```
$ mkdir -p proto/lib/svc/manifest/site
$ mkdir -p proto/lib/svc/method
$ mkdir -p proto/opt/myapp
$ ln -s ../../opt/share/myapp/logs proto/opt/myapp/logs
```

In this example, the `myapp@1.0` package installed `/opt/myapp/logs` as a directory and the `myapp` application wrote content to this directory. When this new `myapp@3.0` package installs `/opt/myapp/logs` as a link, any content in the `/opt/myapp/logs` directory will be saved in `/var/pkg/lost+found`. To instead save that content to the new shared area, deliver an area to hold a copy of that content.

```
$ mkdir -p proto/opt/myapp/.migrate/logs
```

The service will move the content from this staging area in the BE to the shared area.

Content that the `myapp@3.0` package writes to `/opt/myapp/logs` will go directly to the shared area through the link, as in the previous example.

2. **Create the service start method.**

Add the following task to the `proto/lib/svc/method/myapp-share-files.sh` script that you created in the previous procedure: Move content from the staging area to the shared area.

Do not use the service to remove the empty staging area. The staging area is packaged content and should only be removed by uninstalling the package.

```
#!/bin/sh

# Load SMF shell support definitions
. /lib/svc/share/smf_include.sh

# Create rpool/OPTSHARE with mount point /opt/share if it does not already exist
# Create /opt/share/myapp/logfiles if it does not already exist
# Move any content from /opt/myapp/.migrate/logs to /opt/share/myapp/logs

# After this script runs, the service does not need to remain online.
smf_method_exit $SMF_EXIT_TEMP_DISABLE done "myapp shared files moved"
```

3. **Create the service manifest.**

```
$ svcbundle -s service-name=site/myapp-share-files \
-s start-method=/lib/svc/method/myapp-share-files.sh -o myapp-share-files.xml
```

Use the `svccfg validate` command to make sure the service manifest is valid.

4. **Generate the initial package manifest.**

The manifest is the same as the package manifest in the previous example with the following modifications:

- All occurrences of `logfiles` are changed to `logs`.
- The following two actions are added:

```
dir  path=opt/myapp/.migrate owner=root group=bin mode=0755
dir  path=opt/myapp/.migrate/logs owner=root group=bin mode=0755
```

5. **Add metadata and actuators.**

   Add the following lines to your `share.mog` input file for `pkgmogrify` from the previous example. The `salvage-from` attribute moves any unpackaged content in the `/opt/myapp/logs` directory to the `/opt/myapp/.migrate/logs` directory. The service then moves the content from `/opt/myapp/.migrate/logs` to `/opt/share/myapp/logs`.

```
<transform dir path=opt/myapp/.migrate/logfiles -> \
    add salvage-from /opt/myapp/logfiles>
```

   Name this package `myapp@3.0`.

   Run `pkgmogrify` as in the previous example.

6. **Evaluate and resolve package dependencies.**

7. **Verify the package.**

8. **Publish the package.**

9. **Test the package.**

   Create `/opt/myapp/logs` as a regular directory and put some files in it.

   Install the `myapp@3.0` package. Verify that the existence of the dataset was correctly detected and handled, the new link exists, the `/opt/myapp/logs` directory is empty, the `/opt/myapp/.migrate/logs` directory exists and is empty, and the `/opt/share/myapp/logs` directory exists and contains the content that was initially in the `/opt/myapp/logs` directory.

## Delivering a File That Is Also Delivered by Another Package

You might want to use an IPS package to provide a customized version of a file that is already delivered by another package. By default, only one IPS package can deliver a file to any particular location. To use an IPS package to deliver a custom version of a file that is delivered by another IPS package, make sure the following attributes are set on the `file` action:

- The `overlay=allow` and `preserve=true` attributes are set on the file you want to replace.
- The `overlay=true` attribute and the `preserve` attribute with any value are set on the replacement file.

See the descriptions of the overlay and preserve attributes in "File Actions" on page 23.

The version of the file with the overlay=true attribute replaces the version with the overlay=allow attribute, and the version of the file with the overlay=allow attribute is saved in /var/pkg/lost+found/.

For example, suppose you install a package named isvapp that has the following file action:

```
file opt/isvapp/isvconf path=opt/isvapp/isvconf owner=root group=bin mode=0644 \
    overlay=allow preserve=true
```

The package installs the following file:

```
-rw-r--r--   1 root     bin        11358 Apr 17 18:44 /opt/isvapp/isvconf
```

You want a site-specific version of this file on all of your systems. You create a package named isvconf with the following file action to deliver the new version of the file:

```
file opt/isvapp/isvconf path=opt/isvapp/isvconf owner=root group=bin mode=0644 \
    overlay=true preserve=renameold
```

After isvconf is installed, the following files are on the system:

```
$ ls -l /opt/isvapp/isvconf
-rw-r--r--   1 root     bin        72157 Apr 17 18:47 /opt/isvapp/isvconf
$ ls -l /var/pkg/lost+found/opt/isvapp
total 24
-rw-r--r--   1 root     bin        11358 Apr 17 18:44 isvconf-20140417T184756Z
```

If you attempt to install another package, isvconf2 in this example, that would deliver a file with the same path, the installation fails with the following explanation:

```
Creating Plan (Checking for conflicting actions): -
pkg install: The following packages all deliver file actions to opt/isvapp/isvconf:

  pkg://site/isvconf2@1.0,5.11:20140417T190405Z
  pkg://site/isvapp@1.0,5.11:20140417T182316Z
  pkg://site/isvconf@1.0,5.11:20140417T185420Z

These packages may not be installed together. Any non-conflicting set may
be, or the packages must be corrected before they can be installed.
```

You can deliver a new version of the file in an update of the package that delivered the first replacement file. After isvconf@2.0 is installed, the following files are on the system:

```
$ ls -l /opt/isvapp/isvconf*
-rw-r--r--   1 root     bin        64064 Apr 17 18:52 /opt/isvapp/isvconf
-rw-r--r--   1 root     bin        54365 Apr 17 18:47 /opt/isvapp/isvconf.old
$ ls -l /var/pkg/lost+found/opt/isvapp
total 24
-rw-r--r--   1 root     bin        11358 Apr 17 18:44 isvconf-20140417T184756Z
```

The existing file was saved in isvconf.old because both of the following two conditions exist:

- The isvconf package specifies preserve=renameold.
- The file was edited after isvconf@1.0 was installed and before isvconf@2.0 was installed.

The lost+found area has not changed and still contains the original file delivered by isvapp.

## Delivering Multiple Implementations of an Application

You might want to deliver multiple implementations of a given application with the following characteristics:

- All implementations are available in the image.
- One of the implementations is available from a common directory such as /usr/bin for ease of discovery.
- An administrator can easily change which implementation is available from the common directory, without adding or removing any packages.

Oracle Solaris 11 delivers multiple implementations of several different applications, such as Java and Python. To specify which implementation is available from a common directory such as /usr/bin, and to enable an administrator to easily change that selection, use a mediated link.

A *mediated link* manages multiple implementations of an application in a single image. A mediated link is a symbolic link with mediator attributes set (see "Attributes of Mediated Links" on page 113). Software that is packaged with a link action that has mediator attributes is a participant in a *mediation*. The mediation participant that is available from a common directory such as /usr/bin is called the preferred version. The preferred version in a mediation is determined in one of the following ways:

Specified in the package manifest

You can specify a version (mediator-version) or a versioned implementation (mediator-implementation) for each participant in the mediation. You can specify an overriding priority in case of conflicts (mediator-priority).

Selected by the system

If a participant in the mediation has a priority specified, the participant with the highest value priority is selected as the preferred implementation.

If no participant in the mediation has a priority specified, and a participant has a version specified, the participant with the highest value version is selected as the preferred implementation.

If no participant in the mediation has a priority or version specified, an arbitrary participant is selected as the preferred implementation. If the mediator-implementation of the selected participant includes a version string, the participant with the highest value version string for that `mediator-implementation` is selected as the preferred implementation.

Specified by an administrator

An administrator can set the preferred implementation by using the `pkg set-mediator` command. See "Specifying a Default Application Implementation" in *Adding and Updating Software in Oracle Solaris 11.3*.

If only one instance of a particular mediation is installed in an image, then that instance is automatically selected as the preferred implementation of that mediation. If the preferred implementation is set by a system administrator after package installation, installing additional participants in this same mediation does not change the preferred implementation set by the administrator.

## Attributes of Mediated Links

The following attributes can be set on `link` actions to control how mediated links are delivered:

`mediator`

Specifies the entry in the mediation namespace shared by all path names that participate in a given mediation group. Examples include `java`, `python`, and `ruby`.

Every link that has a `mediator` attribute must also have either a `mediator-version` attribute or a `mediator-implementation` attribute. All mediated links for a given path name must specify the same `mediator`. However, not all mediator versions and implementations need to provide a link at a given path. If a mediation participant does not provide a link, then the link is removed when that participant is selected as the preferred implementation.

`mediator-version`

Specifies the version of the interface described by the `mediator` attribute. This attribute is required if `mediator` is specified and `mediator-implementation` is not specified. The value of `mediator-version` is a dot-separated sequence of integers. For ease of use, the value specified should match the version of the package that is delivering the link. For example, the `runtime/ruby-19` package should specify `mediator-version=1.9`. Setting the version value appropriately helps administrators determine what software is participating in the mediation, which packages delivered that software, and which version of the software is currently set as the preferred version. If no participant in the mediation has a `mediator-priority` set, IPS selects the mediation participant with the highest value `mediator-version` as the preferred implementation.

`mediator-implementation`

Specifies the implementation of the interface described by the `mediator` attribute. This attribute is required if `mediator` is specified and `mediator-version` is not specified.

Implementation strings are not considered to be ordered. An implementation is arbitrarily selected by IPS as the preferred implementation if no participant in the mediation has a `mediator-version` or `mediator-priority` set.

The value of `mediator-implementation` can be a string of arbitrary length composed of alphanumeric characters and spaces. If the implementation itself can be versioned, then the version should be specified at the end of the string, after an at sign (`@`). The version is a dot-separated sequence of integers. If multiple versions of an implementation exist, the implementation with the highest version is selected. For example, a `mediator-implementation` value of `4DB@12` would be selected over a `mediator-implementation` value of `4DB@11`.

`mediator-priority`

Specifies the priority of the interface described by the `mediator` attribute. Either a `mediator-version` or a `mediator-implementation` must also be specified. For example, if one participant in the mediation has a `mediator-version` value of `1.6` and another participant has a `mediator-version` value of `1.7`, the participant with the `mediator-version` value of `1.6` can be specified as the preferred implementation by assigning a `mediator-priority` attribute.

The `mediator-priority` attribute can have one of the following values:

`vendor` The link is preferred over those that do not have a `mediator-priority` specified.

`site` The link is preferred over those that do not have a `mediator-priority` specified and over those that have a `mediator-priority` value of `vendor`.

## Specifying Mediated Links

The following command shows the currently selected preferred implementations of Python, Ruby, and Secure Shell:

```
$ pkg mediator python ruby ssh
MEDIATOR    VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
python      vendor    2.6     vendor
ruby        system    1.9     system
ssh         vendor            vendor     sunssh
```

The following command shows all participants in each of these mediations:

```
$ pkg mediator -a python ruby ssh
MEDIATOR    VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
python      vendor    2.6     vendor
python      system    2.7     system
ruby        system    1.9     system
ruby        system    1.8     system
ssh         vendor            vendor     sunssh
```

The lower version was selected by the system as the preferred Python implementation because it has a `mediator-priority` specified, as shown by the `VER. SRC.` and `IMPL. SRC.` and by the following command:

```
$ pkg contents -Ho action.raw -t link -a path=usr/bin/python 'runtime/python*'
link mediator=python mediator-version=2.7 path=usr/bin/python pkg.linted.pkglint
.dupaction010.2=true target=python2.7
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
python target=python2.6
```

If you specify `mediator=python` as the argument to the `-a` option, the output shows many more links in the `python` mediation. Remember to include all necessary paths in the mediation.

```
$ pkg contents -Ho action.raw -t link -a mediator=python runtime/python-26
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
2to3 target=2to3-2.6
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
python target=python2.6
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
pydoc target=pydoc-2.6
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
idle target=idle-2.6
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
python-config target=python2.6-config
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
amd64/python target=python2.6 variant.arch=i386
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
amd64/python-config target=python2.6-config variant.arch=i386
link facet.doc.man=all mediator=python mediator-priority=vendor mediator-version
=2.6 path=usr/share/man/man1/python.1 target=python2.6.1
```

The `pkg.linted.pkglint.dupaction010.2=true` attribute in the `usr/bin/python` mediated link in the `runtime/python-27` package indicates that the `/usr/bin/python` link is delivered by more than one package and is a valid mediated link. Mediated links are an exception to the rule that an action can be delivered by only one package. The `pkglint` utility checks for duplicate actions. Setting `pkg.linted.`*check.id* to `true` bypasses checks for *check.id* for that action. See “Verify the Package” on page 48 and the `pkglint(1)` man page. Use the `pkglint -L` command to show the full list of checks that `pkglint` performs. The description of the `pkglint.dupaction010` check is “Mediated links should be valid.”

The higher version was selected by the system as the preferred Ruby implementation.

```
$ pkg contents -Ho action.raw -t link -a path=usr/bin/ruby runtime/ruby-19
link mediator=ruby mediator-version=1.9 path=usr/bin/ruby pkg.linted.pkglint.dup
action010.2=true target=./ruby19
```

The `ssh` mediation has only one participant. If you anticipate delivering additional implementations of an application, define the mediation in the original package so that the original package will be a participant in the mediation when other implementations are delivered. Otherwise, you will need to deliver an update to the original package, or users will not be able to select the original implementation as the preferred implementation.

In addition to showing the link action, the following command shows the name of the package where this action is defined.

```
$ pkg contents -o pkg.name,action.raw -t link -a path=usr/bin/ssh '*'
PKG.NAME    ACTION.RAW
network/ssh link mediator=ssh mediator-implementation=sunssh mediator-priority=
vendor path=usr/bin/ssh target=../lib/sunssh/bin/ssh
```

A mediated link that specifies a `mediator-implementation` can also specify a `mediator-version`, a `mediator-priority`, or both. If all participants in the mediation specify only a `mediator-implementation`, the system selects the preferred implementation arbitrarily. If the selected `mediator-implementation` is versioned, the highest version is selected, as shown by the following commands:

```
$ pkg mediator -a myapp
MEDIATOR    VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
myapp       system            system     db@12
myapp       system            system     db@11
myapp       system            system     db
$ pkg mediator myapp
MEDIATOR    VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
myapp       system            system     db@12
```

If another implementation is added to the mediation, that implementation might be selected by the system, as shown by the following commands:

```
$ pkg mediator -a myapp
MEDIATOR    VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
myapp       system            system     aa
myapp       system            system     db@12
myapp       system            system     db@11
myapp       system            system     db
$ pkg mediator myapp
MEDIATOR    VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
myapp       system            system     aa
```

## Best Practices for Mediated Links

Do not deliver the same path as a link in one package and a directory or file in another package. In general, do not deliver the same path more than once. If you deliver the same link path more than once, make sure each instance has a different target, and make sure each instance participates in the same mediation.

Remember to include all necessary paths in the mediation. Libraries, configuration files, man pages, and other file system objects might be different for each implementation.

If you anticipate delivering additional implementations of an application, define the mediation in the original package so that the original package will be a participant in the mediation when other implementations are delivered. Otherwise, you will need to deliver an update to the original package, or users will not be able to select the original implementation as the preferred implementation.

If other software has a dependency on software that participates in a mediation, and if any version of that software satisfies the dependency, use a `require-any` dependency. See "Depend Actions" on page 31 for information about `require-any` dependencies.

For ease of use, the value specified for `mediator-version` should match the version of the package that is delivering the link. Setting the version value appropriately helps administrators determine what software is participating in the mediation, which packages delivered that software, and which version of the software is currently set as the preferred version.

♦ ♦ ♦ CHAPTER 9

# Signing IPS Packages

This chapter describes IPS package signing and how developers and quality assurance organizations can sign either new packages or existing, already signed packages.

- The ability to validate that the software installed on the system is actually as originally specified by the publisher is an important feature of IPS. This ability to validate the installed system is key for both the user and the support engineering staff.
- In addition to validation, signatures can also be used to indicate approval by other organizations or parties. For example, the internal QA organization could sign manifests of packages once the packages are qualified for production use. Such approvals could be required for installation.
- Packages can be signed multiple times, to indicate approval at multiple levels. Signing a package adds a `signature` action to the manifest but does not alter the package in any other way. Signing a package does not remove or invalidate previous signatures.
- Signature policies can be set for the image or for specific publishers. Policies include ignoring signatures, verifying existing signatures, requiring signatures, and requiring specific common names in the chain of trust.

## Signature Actions

Signatures are represented as actions just as all other manifest content is represented as actions. Since manifests contain all the package metadata (such as file permissions, ownership, and content hashes), a `signature` action that validates that the manifest has not be altered since it was published is an important part of system validation.

The `signature` actions form a tree that includes the delivered binaries such that complete verification of the installed software is possible.

A `signature` action has the following form:

```
signature hash_of_certificate algorithm=signature_algorithm \
    value=signature_value \
    chain="hashes_of_certificates_needed_to_validate_primary_certificate" \
    version=pkg_version_of_signature
```

The payload and `chain` attributes represent the packaging hash of the PEM (Privacy Enhanced Mail) files, containing the x.509 certificates which can be retrieved from the originating repository. The payload certificate is the certificate that verifies the value in `value`. The `value` is the signed hash of the message text of the manifest, prepared as discussed below.

The other certificates presented need to form a certificate path that leads from the payload certificate to the trust anchors.

Two types of signature algorithms are supported:

RSA

The first type of signature algorithm is the RSA group of algorithms. An example of an RSA signature algorithm is `rsa-sha256`. The string after the hyphen (`sha256` in this example) specifies the hash algorithm to use to change the message text into a single value that the RSA algorithm can use.

Hash only

The second type of signature algorithm is compute the hash only. This type of algorithm exists primarily for testing and process verification purposes and presents the hash as the signature value. A signature action of this type is indicated by the lack of a payload certificate hash. This type of signature action is verified if the image is configured to check signatures. However, its presence does not count as a signature if signatures are required. The following example shows a hash-only `signature` action:

```
signature algorithm=hash_algorithm value=hash \
    version=pkg_version_of_signature
```

## Signing Packages

A manifest can have multiple independent signatures. Signatures can be added or removed without invalidating other signatures that are present. This feature facilitates production handoffs, with signatures used along the path to indicate completion along the way. Subsequent steps can optionally remove previous signatures at any time. See the `pkgsign`(1) man page for descriptions of options of the `pkgsign` command and examples of use.

Take the following two steps to sign a package. The second step can be performed as many times as needed, adding multiple signatures.

1. Publish the package unsigned to a repository as shown in “Publish the Package” on page 49.
2. Use the `pkgsign` command to append a signature action to the manifest in the repository, as shown in “Sign the Package” on page 51. Except for adding a `signature` action, the package is unaltered, including its time stamp. Signing the package should be the last step of the package development before the package is tested.

The pkgsign command enables someone other than the package publisher to add a signature action to the package without invalidating the original publisher's signature. Republishing a package creates a new time stamp and invalidates the original signature. With the pkgsign command, the QA department, for example, could sign all packages that are installed internally to indicate that they have been approved for use without republishing the packages.

---

**Note -** Using the pkgsign command is the only way to add a signature to a signed package. If you publish a package that already contains a signature, that signature is removed and a warning is emitted.

---

Signature actions with variants are ignored. Therefore, performing a pkgmerge on a pair of manifests invalidates any signatures that were previously applied.

# Using a Custom Certificate Authority Certificate

A custom Certificate Authority (CA) certificate is used to sign other certificates. The system determines whether a key and certificate are valid by verifying that the CA referenced on a certificate has a corresponding CA certificate in /etc/certs/CA.

## ▼ How to Use a Custom Certificate Authority Certificate

1. **Create your custom CA certificate.**

   See "Creating a Self-Signed Server Certificate Authority" in *Copying and Creating Package Repositories in Oracle Solaris 11.3* for a description of creating and testing your own CA certificate.

2. **Put the CA certificate in the directory specified by the trust-anchor-directory property.**

   See "Configure Image and Publisher Properties" on page 123 for a description of the trust-anchor-directory image property.

   - Put the CA certificates directly in the directory named by trust-anchor-directory. Do not put the certificates in another subdirectory.
   - Do not put a CA certificate in the directory that is a duplicate of a certificate that is already in the directory.
   - Do not put a file in the directory that is not a valid certificate file.

3. **Refresh the ca-certificates service.**

```
$ svcadm refresh svc:/system/ca-certificates:default
```

Verify that the service is online:

```
$ svcs ca-certificates
```

If the service is not in the `online` state, or if the CA does not appear in `/etc/certs/ca-certificates.crt`, check the service log file:

```
$ svcs -xL ca-certificates
```

4. **(Optional) Package the custom certificate and key.**

   Updating the certificate and key for multiple systems is easier if you package the certificate and key files. If the certificates need to change, update the package, and then `pkg update` the package on each system.

   a. **Add a refresh actuator to each certificate and key file that you deliver.**

   ```
   file group=sys mode=0644 owner=root path=etc/certs/CA/mycert.pem \
   refresh_fmri=svc:/system/ca-certificates:default
   ```

   The following `pkgmogrify` rule automates adding this refresh actuator:

   ```
   <transform file path=etc/certs/CA/.*\.pem ->
     add refresh_fmri svc:/system/ca-certificates:default>
   ```

   b. **Do not deliver the `/etc/certs/CA` directory in your package.**

   The `/etc`, `/etc/certs`, and `/etc/certs/CA` directories are already delivered by the system. See "Add Necessary Metadata to the Generated Manifest" on page 42 and "Verify the Package" on page 48.

# Troubleshooting Signed Packages

The `pkgsign` tool does not perform all possible checks for its inputs when signing packages. Therefore, it is important to check signed packages to ensure that they can be properly installed after being signed.

This section shows errors that can occur when attempting to install or update a signed package and provides explanations of the errors and solutions to the problems.

A signed package can fail to install or update for reasons that are unique to signed packages. For example, if the signature of a package fails to verify, or if the chain of trust cannot be verified or anchored to a trusted certificate, the package fails to install.

## Configure Image and Publisher Properties

The image and publisher properties described in this section influence the checks that are performed on signed packages.

To configure image properties, use the `set-property`, `add-property-value`, `remove-property-value`, and `unset-property` subcommands of the `pkg` command.

To specify signature policy and required names for a particular publisher, use the `--set-property`, `--add-property-value`, `--remove-property-value`, and `--unset-property` options of the `set-publisher` subcommand.

See "Configuring Package Signature Properties" in *Adding and Updating Software in Oracle Solaris 11.3* for examples.

Configure the following image properties to use signed packages:

`signature-policy`

The value of this property determines the checks that will be performed on manifests when installing, updating, modifying, or verifying packages in the image. The final policy applied to a package depends on the combination of image policy and publisher policy. The combination will be at least as strict as the stricter of the two policies taken individually. By default, the package client does not check whether certificates have been revoked. To enable those checks, which might require the client to contact external web sites, set the `check-certificate-revocation` image property to `true`. The following values are allowed:

`ignore`

Ignore signatures for all manifests.

`verify`

Verify that all manifests with signatures are validly signed but do not require all installed packages to be signed.

This is the default value.

`require-signatures`

Require that all newly installed packages have at least one valid signature. The `pkg fix` and `pkg verify` commands also warn if an installed package does not have a valid signature.

`require-names`

Follow the same requirements as `require-signatures` but also require that the strings listed in the `signature-required-names` image property appear as a common name of the certificates used to verify the chains of trust of the signatures.

signature-required-names

The value of this property is a list of names that must be seen as common names of certificates while validating the signatures of a package.

trust-anchor-directory

The relative path name (relative to the root of the image) of the directory that contains the trust anchors for the image. The default is etc/certs/CA.

If you create your own SSL Certificate Authority certificates, put those certificates in the directory named by trust-anchor-directory and refresh the ca-certificates service as described in "How to Use a Custom Certificate Authority Certificate" on page 121. Put the CA certificates directly in the directory named by trust-anchor-directory; do not put the certificates in another subdirectory.

Configure the following publisher properties to use signed packages from a particular publisher:

signature-policy

The function of this property is identical to the function of the signature-policy image property except that this property applies only to packages from the specified publisher.

signature-required-names

The function of this property is identical to the function of the signature-required-names image property except that this property applies only to packages from the specified publisher.

## Chain Certificate Not Found

The following error occurs when a certificate in the chain of trust is missing or otherwise erroneous.

```
pkg install: The certificate which issued this certificate:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs1_ch1_ta3/emailAddress=cs1_ch1_ta3
could not be found. The issuer is:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=ch1_ta3/emailAddress=ch1_ta3
The package involved is: pkg://test/example_pkg@1.0,5.11-0:20110919T184152Z
```

In this example, there were three certificates in the chain of trust when the package was signed. The chain of trust was rooted in the trust anchor, a certificate named ta3. The ta3 certificate signed a chain certificate named ch1_ta3, and ch1_ta3 signed a code signing certificate named cs1_ch1_ta3.

When the pkg command tried to install the package, it was able to locate the code signing certificate, cs1_ch1_ta3, but could not locate the chain certificate, ch1_ta3, so the chain of trust could not be established.

The most common cause of this problem is failing to provide the correct certificates to the `-i` option of `pkgsign`.

## Authorized Certificate Not Found

The following error is similar to the error shown in the previous example but the cause is different.

```
pkg install: The certificate which issued this certificate:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs1_cs8_ch1_ta3/emailAddress=cs1_cs8_ch1_ta3
could not be found. The issuer is:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs8_ch1_ta3/emailAddress=cs8_ch1_ta3
The package involved is: pkg://test/example_pkg@1.0,5.11-0:20110919T201101Z
```

In this case, the package was signed using the `cs1_cs8_ch1_ta3` certificate, which was signed by the `cs8_ch1_ta3` certificate.

The problem is that the `cs8_ch1_ta3` certificate was not authorized to sign other certificates. Specifically, the `cs8_ch1_ta3` certificate had the `basicConstraints` extension set to `CA:false` and marked critical.

When the `pkg` command verifies the chain of trust, it does not find a certificate that is allowed to sign the `cs1_cs8_ch1_ta3` certificate. Since the chain of trust cannot be verified from the leaf to the root, the `pkg` command prevents the package from being installed.

## Untrusted Self-Signed Certificate

The following error occurs when a chain of trust ends in a self-signed certificate that is not trusted by the system.

```
pkg install: Chain was rooted in an untrusted self-signed certificate.
The package involved is:pkg://test/example_pkg@1.0,5.11-0:20110919T185335Z
```

When you create a chain of certificates using OpenSSL for testing, the root certificate is usually self-signed, since there is little reason to have an outside company verify a certificate that is only used for testing.

In a test situation, there are two solutions:

- The first solution is to add the self-signed certificate that is the root of the chain of trust into `/etc/certs/CA` and refresh the `system/ca-certificates` service. This mirrors the likely situation customers will encounter where a production package is signed with a certificate that is ultimately rooted in a certificate that is delivered with the operating system as a trust anchor.

- The second solution is to approve the self-signed certificate for the publisher that offers the package for testing by using the `--approve-ca-cert` option with the `pkg set-publisher` command.

## Signature Value Does Not Match Expected Value

The following error occurs when the value on the `signature` action could not be verified using the certificate that the action claims was paired with the key used to sign the package.

```
pkg install: A signature in pkg://test/example_pkg@1.0,5.11-0:20110919T195801Z
could not be verified for this reason:
The signature value did not match the expected value. Res: 0
The signature's hash is 0ce15c572961b7a0413b8390c90b7cac18ee9010
```

There are two possible causes for an error like this:

- The first possible cause is that the package has been changed since it was signed. This is unlikely but is possible if the package manifest has been hand edited since signing. Without manual intervention, the package should not have changed since it was signed because `pkgsend` strips existing `signature` actions during publication because the old signature is invalid when the package gets a new time stamp.
- The second, more likely cause is that the key and certificate used to the sign the package were not a matched pair. If the certificate given to the `-c` option of `pkgsign` was not created with the key given to the `-k` option of `pkgsign`, the package is signed, but its signature will not be verified.

## Unknown Critical Extension

The following error occurs when a certificate in the chain of trust uses a critical extension that `pkg` does not understand.

```
pkg install: The certificate whose subject is
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs2_ch1_ta3/emailAddress=cs2_ch1_ta3
could not be verified because it uses a critical extension that pkg5 cannot
handle yet. Extension name:issuerAltName
Extension value:<EMPTY>
```

Until `pkg` learns how to process that critical extension, the only solution is to regenerate the certificate without the problematic critical extension.

## Unknown Extension Value

The following error is similar to the previous error except that the problem is not with an unfamiliar critical extension but with a value that `pkg` does not understand for an extension that `pkg` does understand.

```
pkg install: The certificate whose subject is
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs5_ch1_ta3/emailAddress=cs5_ch1_ta3
could not be verified because it has an extension with a value that pkg(7)
does not understand.
Extension name:keyUsage
Extension value:Encipher Only
```

In this case, `pkg` understands the `keyUsage` extension, but does not understand the value `Encipher Only`. The error looks the same whether the extension in question is critical or not.

The solution, until `pkg` learns about the value in question, is to remove the value from the extension, or remove the extension entirely.

## Unauthorized Use of Certificate

The following error occurs when a certificate has been used for a purpose for which it was not authorized.

```
pkg install: The certificate whose subject is
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=ch1_ta3/emailAddress=ch1_ta3
could not be verified because it has been used inappropriately.
The way it is used means that the value for extension keyUsage must include
'DIGITAL SIGNATURE' but the value was 'Certificate Sign, CRL Sign'.
```

In this case, the certificate `ch1_ta3` has been used to sign the package. The `keyUsage` extension of the certificate means that the certificate is only valid to sign other certificates and CRLs (Certificate Revocation Lists).

## Unexpected Hash Value

The following error indicates that the certificate has been changed since it was last retrieved from the publisher.

```
pkg install: Certificate
/tmp/ips.test.7149/0/image0/var/pkg/publisher/test/
certs/0ce15c572961b7a0413b8390c90b7cac18ee9010
has been modified on disk. Its hash value is not what was expected.
```

The certificate at the provided path is used to verify the package being installed, but the hash of the contents on disk do not match what the signature action thought they should be.

The simple solution is to remove the certificate and allow `pkg` to download the certificate again.

## Revoked Certificate

The following error indicates the certificate in question, which was in the chain of trust for the package to be installed, was revoked by the issuer of the certificate.

```
pkg install: This certificate was revoked:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs1_ch1_ta4/emailAddress=cs1_ch1_ta4
for this reason: None
The package involved is: pkg://test/example_pkg@1.0,5.11-0:20110919T205539Z
```

♦ ♦ ♦ CHAPTER 10

# Handling Non-Global Zones

Developing packages that work consistently with zones usually involves little or no additional work. This chapter describes:

- How IPS handles zones
- How to package software that delivers both global zone and non-global zone components

## Packaging Considerations for Non-Global Zones

When considering zones and packaging, two questions need to be answered:

- Does anything in my package have an interface that crosses the boundary between the global zone and non-global zones?
- How much of the package should be installed in the non-global zone?

### Does the Package Cross the Global, Non-Global Zone Boundary?

If `pkgA` delivers both kernel and userland functionality, and both sides of that interface must be updated accordingly, then whenever `pkgA` is updated in a non-global zone, `pkgA` must also be updated in any other zones where `pkgA` is installed.

To ensure this update is done correctly, use a `parent` dependency in `pkgA`. If a single package delivers both sides of the interface, then a `parent` dependency on `feature/package/dependency/self` ensures that the global zone and the non-global zones contain the same version of the package, preventing version skew across the interface.

The `parent` dependency also ensures that if the package is in a non-global zone, then it is also present in the global zone.

If the interface spans multiple packages, then the package that contains the non-global zone side of the interface must contain a `parent` dependency on the package that delivers the global zone side of the interface. The `parent` dependency is also discussed in "Dependency Types" on page 67.

## How Much of a Package Should Be Installed in a Non-Global Zone?

If all of a package should be installed when the package is being installed in a non-global zone, then nothing needs to be done to the package to enable it to function properly. For consumers of the package, though, it can be reassuring to see that the package author properly considered zone installation and decided that this package can function in a zone. For that reason, you should explicitly state that the package functions in both global and non-global zones. To do this, add the following action to the manifest:

```
set name=variant.opensolaris.zone value=global value=nonglobal
```

If no content in the package can be installed in a non-global zone (for example a package that only delivers kernel modules or drivers), then the package should specify that it cannot be installed in a non-global zone. To do this, add the following action to the manifest:

```
set name=variant.opensolaris.zone value=global
```

If some but not all of the content in the package can be installed in a non-global zone, then take the following steps:

1. Use the following `set` action to state that the package can be installed in both global and non-global zones:

   ```
   set name=variant.opensolaris.zone value=global value=nonglobal
   ```
2. Identify the actions that are relevant only in the global zone or only in a non-global zone. Assign the following attribute to actions that are relevant only in the global zone:

   ```
   variant.opensolaris.zone=global
   ```

   Assign the following attribute to actions that are relevant only in a non-global zone:

   ```
   zone:variant.opensolaris.zone=nonglobal
   ```

If a package has a `parent` dependency or has pieces that are different in global and non-global zones, test to ensure that the package works as expected in a non-global zone as well as in the global zone.

If the package has a `parent` dependency on itself, then the global zone must configure the repository that delivers the package as one of its origins. Install the package in the global zone first, and then in the non-global zone for testing.

## Troubleshooting Package Installations in Non-Global Zones

This section discusses problems that users might encounter when attempting to install a package in a non-global zone.

## Packages that Have Parent Dependencies on Themselves

If you encounter a problem installing a package in a non-global zone, ensure that the following services are online in the global zone:

```
svc:/application/pkg/zones-proxyd:default
svc:/application/pkg/system-repository:default
```

Ensure that the following service is online in the non-global zone:

```
svc:/application/pkg/zones-proxy-client:default
```

These three services provide publisher configuration to the non-global zone and a communication channel that the non-global zone can use to make requests to the repositories assigned to the system publishers served from the global zone.

You cannot update the package in the non-global zone, since it has a parent dependency on itself. Initiate the update from the global zone; `pkg` updates the non-global zone along with the global zone.

Once the package is installed in the non-global zone, test the functionality of the package.

## Packages that Do Not Have Parent Dependencies on Themselves

If the package does not have a `parent` dependency on itself, then you do not need to configure the publisher in the global zone, and you should not install the package in the global zone. Updating the package in the global zone will not update the package in the non-global zone. In this case, updating the package in the global zone can cause unexpected results when testing the older non-global zone package.

The simplest solution in this situation is to make the publisher available to the non-global zone, and install and update the package from within the non-global zone.

If the zone cannot access the publisher's repositories, configuring the publisher in the global zone enables the `zones-proxy-client` and `system-repository` services to proxy access to the publisher for the non-global zone. Then install and update the package in the non-global zone.

♦ ♦ ♦ C H A P T E R 1 1

# Modifying Published Packages

Occasionally, you might need to modify packages that you did not produce. For example, you might need to override attributes, replace a portion of the package with an internal implementation, or remove binaries that are not permitted on your systems.

This chapter describes how you can modify existing packages for local conditions.

This chapter discusses the following topics:

- Republishing packages
- Changing package metadata
- Changing the package publisher

## Republishing Packages

IPS enables you to easily republish an existing package with your modifications, even if you did not originally publish the package. You can also republish new versions of the modified package so that `pkg update` continues to work as users expect.

Use the following steps to modify and republish a package:

1. Use `pkgrecv(1)` to download the package to be republished in a raw format to a specified directory. All of the files are named by their hash value, and the manifest is named `manifest`. Remember to set any required proxy configuration in the `http_proxy` environment variable.
2. Use `pkgmogrify(1)` to make the necessary modifications to the manifest. See "Add Necessary Metadata to the Generated Manifest" on page 42 and Chapter 6, "Modifying Package Manifests Programmatically".
   - Remove any time stamp from the internal package FMRI to prevent confusion during publication.
   - Remove any signature actions.
3. Use `pkglint(1)` to verify the resulting package.
4. Use `pkgsend(1)` to republish the package. Republication strips the package of any signatures that are present and ignores any time stamp specified by `pkg.fmri`. To prevent a warning message, remove signature actions in the `pkgmogrify` step.

If you do not have permission to publish to the original source of the package, use pkgrepo(1) to create a repository, and then use the following command to set the new publisher ahead of the original publisher in the publisher search order:

```
$ pkg set-publisher --search-before=original_publisher  new_publisher
```

5. If necessary, use pkgsign(1) to sign the package. To prevent client caching issues, sign the package before you install the package, even for testing. See Chapter 9, "Signing IPS Packages".

## Changing Package Metadata

In the following example, the original pkg.summary value is changed to be "IPS has lots of features." The package is downloaded using the --raw option of pkgrecv. By default, only the newest version of the package is downloaded. The package is then republished to a new repository.

```
$ mkdir republish; cd republish
$ pkgrecv -d . --raw -s http://pkg.oracle.com/solaris/release package/pkg
$ cd package* # The package name contains a '/' and is url-encoded.
$ cd *
$ cat > fix-pkg
# Change the value of pkg.summary
<transform set name=pkg.summary -> edit value '.*' "IPS has lots of features">
# Delete any signature actions
<transform signature -> drop>
# Remove the time stamp from the fmri so that the new package gets a new time stamp
<transform set name=pkg.fmri -> edit value ":20.+" "">
^D
$ pkgmogrify manifest fix-pkg > new-manifest
$ pkgrepo create ./mypkg
$ pkgsend -s ./mypkg publish -d . new-manifest
```

## Changing Package Publisher

Another common use case is to republish packages under a new publisher name. One case when this is useful is to consolidate packages from multiple repositories into a single repository. For example, you might want to consolidate packages from repositories of several different development teams into a single repository for integration testing.

To republish under a new publisher name, use the pkgrecv, pkgmogrify, pkgrepo, and pkgsend steps shown in the previous example.

The following sample transform changes the publisher to mypublisher:

```
<transform set name=pkg.fmri -> edit value pkg://[^/]+/ pkg://mypublisher/>
```

You can use a simple shell script to iterate over all packages in the repository. Use the output from `pkgrecv --newest` to process only the newest packages from the repository.

The following script saves the above transform in a file named `change-pub.mog`, and then republishes from `development-repo` to a new repository `new-repo`, changing the package publisher along the way:

```
#!/usr/bin/ksh93
pkgrepo create new-repo
pkgrepo -s new-repo set publisher/prefix=mypublisher
mkdir incoming
for package in $(pkgrecv -s ./development-repo --newest); do
    pkgrecv -s development-repo -d incoming --raw $package
done
for pdir in incoming/*/* ; do
    pkgmogrify $pdir/manifest change-pub.mog > $pdir/manifest.newpub
    pkgsend -s new-repo publish -d $pdir $pdir/manifest.newpub
done
```

This script could be modified to do tasks such as select only certain packages, make additional changes to the versioning scheme of the packages, and show progress as it republishes each package.

♦ ♦ ♦ APPENDIX A

# Classifying Packages

This appendix describes:

- How to specify a classification for your package
- Classification scheme definitions

## Assigning Classifications

The Package Manager GUI uses the `info.classification` package attribute, with scheme `org.opensolaris.category.2008`, to display packages by category. Users can also use the `pkg search` command to display packages that have a given classification.

Use a `set` action to assign a classification to a package, as shown in the following example:

```
set name=info.classification \
    value="org.opensolaris.category.2008:System/Administration and Configuration"
```

The category and subcategory are separated by a forward slash character. Spaces in the attribute value require quoting.

A package can have more than one classification, as shown in the following example:

```
set name=info.classification \
    value="org.opensolaris.category.2008:Meta Packages/Group Packages" \
    value="org.opensolaris.category.2008:Web Services/Application and Web Servers"
```

## Classification Values

The following category and subcategory values are defined:

Meta Packages

Group Packages

Incorporations

Applications

Accessories
Configuration and Preferences
Games
Graphics and Imaging
Internet
Office
Panels and Applets
Plug-ins and Run-times
Sound and Video
System Utilities
Universal Access

Desktop (GNOME)

Documentation
File Managers
Libraries
Localizations
Scripts
Sessions
Theming
Trusted Extensions
Window Managers

Development

C
C++
Databases
Distribution Tools
Editors
Fortran
GNOME and GTK+
GNU
High Performance Computing
Java
Objective C
Other Languages
PHP
Perl
Python
Ruby
Source Code Management
Suites

- System
- X11

Drivers

- Display
- Media
- Networking
- Other Peripherals
- Ports
- Storage

System

- Administration and Configuration
- Core
- Databases
- Enterprise Management
- File System
- Fonts
- Hardware
- Internationalization
- Libraries
- Localizations
- Media
- Multimedia Libraries
- Packaging
- Printing
- Security
- Services
- Shells
- Software Management
- Text Tools
- Trusted
- Virtualization
- X11

Web Services

- Application and Web Servers
- Communications

♦ ♦ ♦ APPENDIX B

# How IPS Is Used To Package the Oracle Solaris OS

This appendix describes:

- Details of package FMRI version strings
- How `incorporate` dependencies, `facet.version-lock.*` facets, `group` dependencies, and action tags are used to define working package sets for the Oracle Solaris OS

## Oracle Solaris Package Versioning

"Package Identifier: FMRI" on page 18 described the `pkg.fmri` attribute and the different parts of the version field, including how the version field can be used to support different models of software development. This section explains how the Oracle Solaris OS uses the version field, and provides insight into the reasons that a fine-grained versioning scheme can be useful. In your packages, you do not need to follow the same versioning scheme that the Oracle Solaris OS uses.

The meaning of each part of the version string in the following sample package FMRI is given below:

```
pkg://solaris/system/library/storage/suri@0.5.11,5.11-0.175.3.0.0.19.0:20150329T164922Z
```

`0.5.11`

Component version. For packages that are part of the Oracle Solaris OS, this is the OS `major.minor` version. For other packages, this is the upstream version. For example, the component version of the following Apache Web Server package is 2.2.29:

```
pkg:/web/server/apache-22@2.2.29,5.11-0.175.3.0.0.19.0:20150329T181125Z
```

`5.11`

Release. This is used to define the OS release that this package was built for. The release should always be 5.11 for packages created for Oracle Solaris 11.

`0.175.3.0.0.19.0`

Branch version. Oracle Solaris packages show the following information in the branch version portion of the version string of a package FMRI:

`0.175` Major release number. The major or marketing development release build number. In this example, `0.175` indicates Oracle Solaris 11.

`3` Update release number. The update release number for this Oracle Solaris release. The update value is 0 for the first customer shipment of an Oracle Solaris release, 1 for the first update of that release, 2 for the second update of that release, and so forth. In this example, `3` indicates Oracle Solaris 11.3.

`0` SRU number. The Support Repository Update (SRU) number for this update release. SRUs are approximately monthly updates that fix bugs, fix security issues, or provide support for new hardware. SRUs do not include new features. The Oracle Support Repository is available only to systems under a support contract.

`0` Reserved. This field is not currently used for Oracle Solaris packages.

`19` Release or SRU build number. The build number of the SRU, or the respin number for the major release.

`0` Nightly build number. The build number for the individual nightly builds.

If the package is an Interim Diagnostic Relief (IDR), then the branch version of the package FMRI contains the following two additional fields. IDRs are package updates that help diagnose customer issues or provide temporary relief for a problem until a formal package update is issued. The following examples are for `idr824`, which has FMRI `pkg://solaris/idr824@4,5.11:20131114T034951Z` and contains packages such as `pkg:/system/library@0.5.11-0.175.1.6.0.4.2.824.4`:

`824`

Name of the IDR.

`4`

Version of the IDR.

`20150329T164922Z`

Time stamp. The time stamp is defined when the package is published.

# Oracle Solaris Incorporation Packages

Oracle Solaris is delivered as a set of packages, with subsets of packages constrained by incorporations. See "Incorporation Packages" in *Adding and Updating Software in Oracle Solaris 11.3* for more information about incorporation packages.

Use the following command to list available incorporations:

```
$ pkg list -as entire \*incorporation
```

Incorporate packages contain `incorporate` dependencies. See "`incorporate` Dependency" on page 72 for more information.

The `pkg:/entire` package is a special incorporation that constrains other incorporations to the same build by including both `require` and `incorporate` dependencies on each incorporation package. In this way, the `pkg:/entire` incorporation defines a software surface such that core Oracle Solaris system packages are upgraded as a single group.

# Relaxing Dependency Constraints

Some incorporations use `facet.version-lock.*` facets to enable the administrator to use the `pkg change-facet` command to relax the constraint to the incorporation for the specified packages. See "Enabling Administrators to Relax Constraints on Installable Package Versions" on page 75 for more information.

The following list shows examples of `facet.version-lock.*` definitions in the `pkg:/entire` incorporation.

**Caution -** Unlocking facet versions in `pkg:/entire` can result in an unsupported system. Those packages should only be unlocked on advice from Oracle support.

```
depend fmri=consolidation/desktop/gnome-incorporation type=require
depend facet.version-lock.consolidation/desktop/gnome-incorporation=true \
       fmri=consolidation/desktop/gnome-incorporation@0.5.11-0.175.3.10.0.4.0
 type=incorporate
depend fmri=consolidation/desktop/gnome-incorporation@0.5.11-0.175.3 type=incorporate
depend fmri=consolidation/ips/ips-incorporation type=require
depend facet.version-lock.consolidation/ips/ips-incorporation=true \
       fmri=consolidation/ips/ips-incorporation@0.5.11-0.175.3.13.0.3.0 type=incorporate
depend fmri=consolidation/ips/ips-incorporation@0.5.11-0.175.3 type=incorporate
depend fmri=consolidation/java-8/java-8-incorporation type=require
depend facet.version-lock.consolidation/java-8/java-8-incorporation=true \
       fmri=consolidation/java-8/java-8-incorporation@1.8.0.102.14-0 type=incorporate
depend fmri=consolidation/java-8/java-8-incorporation@1.8.0 type=incorporate
```

# Oracle Solaris Group Packages

Oracle Solaris defines several group packages, which contain `group` dependencies. These group packages enable convenient installation of common sets of packages. See “`group` Dependency” on page 69 and “Group Packages” in *Adding and Updating Software in Oracle Solaris 11.3* for more information.

Use the following command to list available group packages:

```
$ pkg list -as entire group\*
```

# Attributes and Tags

This section describes general attributes, Oracle Solaris action attributes, and Oracle Solaris attribute tags.

## Informational Attributes

The following attributes are not necessary for correct package installation, but having a shared convention reduces confusion between publishers and users.

`info.classification`

See “Set Actions” on page 29 for information about the `info.classification` attribute. See a list of classifications in Appendix A, “Classifying Packages”.

`info.keyword`

A list of additional terms that should cause this package to be returned by a search.

`info.maintainer`

A human readable string that describes the entity that provides the package. This string should be the name, or name and email of an individual, or the name of an organization.

`info.maintainer-url`

A URL associated with the entity that provides the package.

`info.upstream`

A human readable string that describes the entity that creates the software. This string should be the name, or name and email of an individual, or the name of an organization.

`info.upstream-url`

A URL associated with the entity that creates the software delivered in the package.

`info.source-url`

A URL to the source code bundle for the package, if appropriate.

`info.repository-url`

A URL to the source code repository for the package, if appropriate.

`info.repository-changeset`

A changeset ID for the version of the source code contained in `info.repository-url`.

## Oracle Solaris Attributes

`org.opensolaris.arc-caseid`

One or more case identifiers (for example, PSARC/2008/190) associated with the ARC case (Architecture Review Committee) or cases associated with the component delivered by the package.

`org.opensolaris.smf.fmri`

One or more FMRIs that represent SMF services delivered by this package. These attributes are automatically generated by `pkgdepend` for packages that contain SMF service manifests. See the `pkgdepend(1)` man page.

## Organization-Specific Attributes

To provide additional metadata for a package, use an organization-specific prefix on the attribute name. Organizations can use this method to provide additional metadata for packages developed in that organization or to amend the metadata of an existing package. To amend the metadata of an existing package, you must have control over the repository where the package is published. For example, a service organization might introduce an attribute named `service.example.com,support-level` or `com.example.service,support-level` to describe a level of support for a package and its contents.

## Oracle Solaris Tags

`variant.opensolaris.zone`

Specifies which actions in a package can be installed in a non-global zone, in the global zone, or in either a non-global or the global zone. See Chapter 10, “Handling Non-Global Zones” for more information.