Developing System Services in Oracle® Solaris 11.3

Part No: E60814
November 2016

ORACLE®

Developing System Services in Oracle Solaris 11.3

Part No: E60814

Référence: E60814

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

Examples

Using This Documentation

- **Overview** – Describes how to use the Oracle Solaris Service Management Facility (SMF) feature. SMF is one of the components of the wider Oracle Solaris Predictive Self Healing capability.
- **Audience** – System administrators and application developers who create custom services to configure applications
- **Required knowledge** – Experience administering Oracle Solaris systems

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

Introduction to Developing Service Management Facility Services

The Oracle Solaris Service Management Facility (SMF) framework manages system and application services. SMF manages critical system services essential to the working operation of the system and manages application services such as a database or Web server.

SMF replaces the use of configuration files for managing services and is the recommended mechanism to use to start applications. SMF replaces the `init` scripting start-up mechanism, `inetd.conf` configurations, and most `rc?.d` scripts.

This chapter describes:

- Where to get more information about SMF
- New features in this release
- How to gain privileges you need to use some SMF commands

SMF Documentation

This guide describes how to develop an SMF service to provide service support for your application, including the following topics:

- Using the service creation tool.
- Converting `inetd.conf` configurations to SMF services.
- Converting SMF service properties to configuration files. This mechanism provides a bridge for services that are managed by SMF but interact with applications that still require configuration files.
- Creating a service that runs periodically rather than continuously, similar to a `cron` job.

See *Managing System Services in Oracle Solaris 11.3* for the following information:

- Information for system administrators such as inspecting and changing service property values, enabling service instances, troubleshooting installed services.
- Descriptions of concepts and components such as service states, service models, service restarters, service properties, service bundles, and service configuration repository.
- Descriptions of different types of dependencies and their attributes and effects.

- How to create a new instance of an existing service or modify an existing service instance.

The following resources provide additional examples of creating and delivering services to perform tasks such as application configuration:

- Chapter 7, "Automating System Change as Part of Package Installation" in *Packaging and Delivering Software With the Image Packaging System in Oracle Solaris 11.3*
- Chapter 8, "Advanced Topics For Package Updating" in *Packaging and Delivering Software With the Image Packaging System in Oracle Solaris 11.3*.

New Features in This Release

The following SMF features are new in this release:

Periodic services

In contrast to running persistently, a periodic service runs at scheduled intervals.

Method source

Service methods can be written in Python and other languages.

Service Management Privileges

Exporting and developing service manifests and profiles does not require special privilege. Using the `svccfg` and `svcadm` commands to modify service state and configuration requires increased privilege. Use one of the following methods to gain the privilege you need. See *Securing Users and Processes in Oracle Solaris 11.3* for more information about roles and profiles, including how to determine which role or profile you need and how to assign privileges.

Roles

Use the `roles` command to list the roles that are assigned to you. Use the `su` command with the name of the role to assume that role. As this role, you can execute any commands that are permitted by the rights profiles that are assigned to that role. For example, if the role is assigned the Service Configuration rights profile, you can execute the `svccfg` and `svcadm` commands modify service properties and change service state.

Rights profiles

Use the `profiles` command to list the rights profiles that are assigned to you. Use one of the following methods to execute commands that your rights profiles permit you to execute:

- Use a profile shell such as `pfbash` or `pfksh`.
- Use the `pfexec` command in front of the command that you want to execute. In general, you must specify the `pfexec` command with each privileged command that you execute.

sudo command

Depending on the security policy at your site, you might be able to use the `sudo` command with your user password to execute a privileged command.

If you need to require specific privileges of administrators who want to use the service you develop, see "Securing Service Tasks" on page 23.

♦ ♦ ♦ CHAPTER 2

Using SMF to Control Your Application

This chapter presents the following information that applies to all SMF services:

- Naming rules and data type definitions
- Best practices such as storing service files in standard locations
- Service development troubleshooting information

This chapter also describes how to use the `svcbundle` tool to get started creating a new service and how to convert a run control script to an SMF service.

Subsequent chapters discuss creating specific types of services for specific purposes.

Creating an SMF Service

An SMF service consists of one or more service manifests and zero or more profile files. Service instances define methods to perform the work of the instance.

A service manifest contains the complete set of properties associated with a specific service, including instances, dependencies, application configuration properties, and methods to run when the service starts and stops. Manifests also provide template information such as a description of the service.

Profiles can define instances for a service that is already defined in a manifest. Profiles can define new properties for these service instances and new values for properties that are defined in the service manifest. Profiles cannot define template elements.

See the `service_bundle(4)` man page and the `/usr/share/lib/xml/dtd/service_bundle.dtd.1` service bundle DTD for a complete description of the contents and format of SMF manifests and profiles. See also “Naming Services, Instances, Property Groups, and Properties” on page 20 for naming rules and information about assigning property group types, and see “Property Group Types and Property Types” on page 21 for information about the values of different property types.

A method can be a daemon, other binary executable, or an executable script. See “Creating Service Instance Methods” on page 22 for more information.

You can use multiple manifests to describe a single service. This method can be useful, for example, to define a new instance of a service without modifying the existing manifest for the service. See "Creating a Service Using Multiple Manifests" on page 27 for more information.

Use the following best practices when creating a custom service:

- Use the `site` prefix in the service name as described in "Service Names" in *Managing System Services in Oracle Solaris 11.3*. The `site` prefix is reserved for site-specific customizations. A service named `svc:/site/`*service-name* will not conflict with the services delivered in an Oracle Solaris release.
- Add name and description metadata to your manifests so that users can get information about this service from the `svcs` and `svccfg describe` commands. You can also add descriptions of property values. See the `value`, `values`, and `template` elements in the DTD.
- Use the `svccfg validate` command to validate your service manifest file or service instance FMRI.
- Use the `smf_method_exit()` function to document the successful or unsuccessful exit of a method script in the log file of the service instance.
- Store your manifest, profile, and method files in the standard locations shown in the following table.

TABLE 1 Standard Locations of Service Development Files

File	Standard Location
manifest	`/lib/svc/manifest/site`
profile	`/etc/svc/profile/site`
method	`/lib/svc/method`

Manifests and profiles stored in these locations are imported into the service configuration repository by the `svc:/system/early-manifest-import:default` service during the boot process before any services start. Running the import process early ensures that the repository will contain information from the latest manifests before the services are started. Manifests and profiles stored in these standard locations are also imported when the `svc:/system/manifest-import` service is restarted.

With your manifest, profile, and method files in standard locations, restart the `manifest-import` service to install and configure your service instances. Use the `svcs` command to check the status of your service instances.

Creating an SMF Service Using the Service Bundle Generator Tool

You can use the svcbundle service bundle generator tool to create a simple service or to start a more complex service. For more information, see the svcbundle(1M) man page. You can use the service bundle DTD and other service manifests to complete a more complex service.

▼ How to Create an SMF Service Using svcbundle

Note - Do not use this procedure if you are creating a periodic service. See "How to Create a Periodic Service Using svcbundle" on page 37.

1. **Determine the service model.**

   By default, svcbundle creates a transient service. Determine whether the start method script for this service starts any long-running daemon and therefore this service is a contract service. See "Service Models" in *Managing System Services in Oracle Solaris 11.3*, the model property in the svcbundle(1M) man page, and the startd/duration property in the svc.startd(1M) man page for information about service models.

2. **Copy the script to the standard location.**

   The service in this example uses a custom script named ex_svc as the start method. Copy this script to /lib/svc/method/ex_svc.

3. **Create an initial manifest.**

   In this example, the service name is site/ex_svc. This service is a transient service and does not need a stop method.

   ```
   $ svcbundle -o /tmp/ex_svc.xml -s service-name=site/ex_svc \
   -s start-method=/lib/svc/method/ex_svc
   ```

   If this service were a contract service, you would specify contract or daemon as the value of the model or duration property, as in -s model=contract.

4. **Make any necessary changes to the manifest.**

   Verify that the content of the /tmp/ex_svc.xml manifest is what you need. You might need to add a dependency or adjust the method timeout, for example. Add comments to describe what the service does and how the properties of the service are used.

5. **Verify that the manifest is valid.**

   Use the svccfg validate command to ensure the service manifest is valid.

```
$ svccfg validate /tmp/ex_svc.xml
```

6. **Copy the manifest to the standard directory.**

```
$ cp /tmp/ex_svc.xml /lib/svc/manifest/site/ex_svc.xml
```

7. **Import the manifest and start the service.**

```
$ svcadm restart manifest-import
```

8. **List the new service.**

   Verify that the new service exists and is in the expected state.

```
$ svcs ex-svc
```

Example 1 Automatically Installing a Generated Manifest

If you do not need to make any changes to the new service manifest, you can use the `-i` option to install the manifest as soon as it is created. The `svcbundle` command will write the manifest to `/lib/svc/manifest/site` and restart the `manifest-import` service. Any existing file with the same name in the `/lib/svc/manifest/site` directory will be overwritten.

```
$ svcbundle -i -s service-name=site/ex_svc \
-s start-method=/lib/svc/method/ex_svc
```

Naming Services, Instances, Property Groups, and Properties

Service names, instance names, property group names, and property names must fit the following expression:

```
([A-Za-z][_A-Za-z0-9.-]*,)?[A-Za-z][_A-Za-z0-9-]*
```

A service name, instance name, property group name, or property name is case sensitive, must begin with an alphabetic character, and can contain alphanumeric characters, the underscore (_), and the hyphen (-). Optionally, a provider name can be included at the beginning of the service, instance, property group, or property name. The provider name is separated by a comma (,), must begin with an alphabetic character, and can contain one or more periods (.).

In an FMRI, property group and property names are encoded according to the Uniform Resource Identifier (URI) Generic Syntax RFC 3986 Internet standard except that the comma character is not encoded.

The following example shows a full FMRI for a property: the FMRI of the service instance, followed by /:properties/, followed by the name of the property. You can use the -f option of the svcprop command to show the full FMRI of a property.

```
svc:/application/pkg/server:default/:properties/pkg/port
```

For information about service FMRIs, see "Service Names" in *Managing System Services in Oracle Solaris 11.3*.

Property Group Types and Property Types

A property group type is a category for the property group. Property group types include the following:

```
application
configfile
dependency
framework
implementation
method
template
```

You can introduce a new property group type. The name of a property group type is a free form string no longer than 140 characters.

When you create a property group, the type of the property group should be either application or a new type that you created. The property group types configfile, dependency, framework, implementation, method, and template have special use in SMF. Property groups of type application are expected to be of interest only to the service to which this property group is attached.

The following table describes the possible values for properties of various types. This information is also available from the scf_value_create(3SCF) man page.

TABLE 2 Service Property Type Value Descriptions

Property Type	Value Description
boolean	Single bit: true or false
count	Unsigned 64-bit quantity
integer	Signed 64-bit quantity
time	Signed 64-bit seconds or signed 32-bit nanoseconds (*ns*) in the following range: 0 <= *ns* < 1,000,000,000
astring	An 8-bit NULL-terminated string
ustring	An 8-bit UTF-8 string
uri	A URI string

Property Type	Value Description
fmri	A Fault Management Resource Identifier
host	A host name, an IPv4 address, or an IPv6 address
hostname	A fully qualified domain name
net_addr	A valid net_addr_v4 or net_addr_v6 address
net_addr_v4	A dotted-quad IPv4 address with optional network portion
net_addr_v6	A legal IPv6 address with optional network portion

Creating Service Instance Methods

A start method performs the work of the service instance. Other methods perform tasks necessary to disable or refresh a service instance, for example.

In a service manifest or profile, a method is defined in an exec_method element that includes name and exec attributes. Possible values for the name attribute are provided in the restarter man page. For example, the master restarter, /lib/svc/bin/svc.startd, supports start, stop, and refresh methods as described in the svc.startd(1M) man page.

There is no restart method. The svcadm restart and svccfg restart commands run the stop method and then the start method.

The exec attribute defines what the method will execute. Possible values for the exec attribute include a custom method script, an existing executable, or a special token defined in SMF, as shown in the following examples:

```
exec='/lib/svc/method/tcsd.sh start'
```

By convention, a custom script as shown in this example takes an argument that specifies the value of the name attribute of the method. In this way, the same script can be used for all methods of that service instance.

```
exec='/usr/lib/zones/zonestatd'
```

This example specifies an existing executable.

```
exec=':true'
exec=':kill'
```

The tokens :kill and :true are explained in the smf_method(5) man page. The :true token should be used for methods that are required by the restarter but that are not necessary for the particular service instance implementation.

The :kill token causes all processes in the primary service contract to be terminated and therefore is most appropriate for a stop method. In general, a refresh method should not be :kill unless the processes in the contract are programmed to handle those signals gracefully.

Service Method Scripts

A method that is a script can be a Bourne shell compatible script or a Python script, for example.

- The file /lib/svc/share/smf_include.sh defines many helper functions for Bourne shell compatible method scripts.
- The file /usr/lib/*python-version*/vendor-packages/smf_include.py defines many helper functions for Python method scripts, including the following functions that are unique for Python:
  - smf_subprocess() – Starts the specified executable in a subprocess. The process can return immediately, enabling the instance to act as a contract service.
  - smf_main() – Calls the appropriate function from the method script using frame inspection. See the comments in the /usr/lib/*python-version*/vendor-packages/smf_include.py file.
- The file /lib/svc/share/smf_exit_codes.sh defines method exit codes.

Use the smf_method_exit() function to document the exit of a method script in the log file of the service instance. The smf_method_exit() function takes an exit code, a token that summarizes the exit reason, and a string that can describe the exit in greater detail. See the smf_method_exit(3SCF) man page for the syntax of the smf_method_exit() function. See /lib/svc/share/smf_exit_codes.sh for the list of exit codes.

Securing Service Tasks

Use any of the following options to restrict which users can run a service or which privileges a user must have to run a service:

- Use *_authorization properties to specify authorizations that are required to read property values, modify service properties and property values, and perform actions on services. See the smf_security(5) man page for more information.
- Use the method_credential element to specify requirements as values of the following properties:
  - user. The user ID in numeric or text form. If absent or :default, uid 0 and default home directory / are used.
  - group. The group ID in numeric or text form. If absent or :default, the group associated with the user in the passwd database is used.
  - supp_groups. Supplementary group IDs to be associated with the method, separated by commas or spaces. If absent or :default, initgroups(3C) is used.
  - privileges. A comma-separated list of privileges. See the privileges(5) man page.

- `limit_privileges`. A comma-separated list of privileges. See the `privileges(5)` man page.

In Chapter 5, "Create Services to Manage Oracle Database Instances", the instance control service and the listener service specify that the service must be run by user `oracle` in group `dba`.

The `network/ntp` service shows a list of required authorizations as the value of the `privileges` property.

The `network/physical` service shows the use of the `supp_groups` property.

- Specify a rights profile for the method to use. See the MySQL and Apache examples in "Locking Down Resources by Using Extended Privileges" in *Securing Users and Processes in Oracle Solaris 11.3*.

Service Development Best Practices

Follow the guidelines described in this section as you develop your service.

Service Method Best Practices

Follow the guidelines described in this section as you develop your service start method or other service methods.

Use SMF Method Exit and Useful Exit Reason

Services are expected to return a successful status when they have completed initialization and are ready to provide the service.

To exit your start method, use `smf_method_exit()`; do not use `exit()`. The `smf_method_exit()` interface requires the following arguments:

- One of the method exit codes defined in `/lib/svc/share/smf_exit_codes.sh` or in the `smf_method(5)` man page
- A short explanation of the reason for exiting
- A longer explanation of the reason for exiting

Make sure that error messages are informative, including guidance for resolving the problem. You might need to capture messages or other information from commands called by your method. If your method is an existing executable, you might want to call that executable inside a method script to improve the exit messaging. The system administrator will see these messages in the service log file.

See the smf_method_exit(3SCF) man page for more information.

Use Dependencies, Avoid Using Timeouts

Do not exit your start method until initialization of the service is complete. If you exit your start method before service initialization is complete, services that depend on this service cannot be started.

Set appropriate values for `timeout_seconds` properties to avoid failing solely because more time is needed to complete the method tasks.

Do not use `timeout_seconds` values or any other kind of timeout or wait to allow enough time for dependencies to reach the `online` state. Instead, declare dependencies appropriately. In addition to allowing enough time for dependencies to start, if a service on which this service has a `require` dependency fails, then this service should fail with appropriate messages and not continue to wait for the failed dependency to start. Again, appropriately declared `dependency` elements are the correct implementation. See "Showing Service Dependencies" in *Managing System Services in Oracle Solaris 11.3* and the "Dependencies" section of the smf(5) man page.

Provide Documentation

Provide appropriate template information as described in the smf_template(5) man page. Administrators can use the `svccfg describe` command to view this information.

- Provide a short common name for the service as described in "Service and Instance Common Names" in the `smf_template`(5) man page.
- Reference appropriate man pages (`manpage` element) or stable URLs (`doc_link` element) for more information.
- Provide names, descriptions, choices, and constraints for property groups and properties that are specific to this service.

Validate the Service Manifest

Use the `svccfg validate` command to validate your service manifest.

If the `svccfg validate` command fails with the error "Required property group missing," you might be attempting to validate a partial manifest. Specification of a single service can be spread across multiple manifests. To avoid this error, make sure all manifests for a multiple-manifest service specify the `include` property in the `service_bundle` element as described in "Creating a Service Using Multiple Manifests" on page 27.

Use Standard Locations

Copy your service manifests and profiles to standard locations with standard ownership and permissions. Do not use non-standard locations for manifest and profile files. See "Service Bundles" in *Managing System Services in Oracle Solaris 11.3* or Table 1, "Standard Locations of Service Development Files," on page 18 for manifest and profile standard locations.

When you create a service for your own use, use `site` at the beginning of the service name (`svc:/site/`*service-name*`:`*instance-name*), and place the manifest in `/lib/svc/manifest/site`.

To test your service, place the manifest and any associated profiles in the correct standard locations and restart the `manifest-import` service. See Chapter 5, "Configuring Multiple Systems" in *Managing System Services in Oracle Solaris 11.3* and "Repairing an Instance That Is Degraded, Offline, or in Maintenance" in *Managing System Services in Oracle Solaris 11.3* for related information.

Converting a Run Control Script to an SMF Service

This section describes how to replace a run control script with an SMF service manifest so that the run control service can be managed by SMF.

▼ How to Convert a Run Control Script to an SMF Service

This procedure describes how to use the `rc-script` property with the `svcbundle` command to convert a run control script to an SMF service.

1. **Determine the service model.**

   By default, `svcbundle` creates a `transient` service. Determine whether this run control script starts any long-running daemon and therefore this service is a `contract` service. See "Service Models" in *Managing System Services in Oracle Solaris 11.3*, the `model` property in the `svcbundle`(1M) man page, and the `startd/duration` property in the `svc.startd`(1M) man page for information about service models.

2. **Create an initial manifest.**

   To convert a run control script, use the `rc-script` property name with the `-s` option of the `svcbundle` command. See the `rc-script` property in the `svcbundle`(1M) man page for more information or enter `svcbundle help rc-script`.

In this example, the service name is `ex_con` and is a contract service that runs at level 2. The run level is specified after a colon after the script name in the `rc-script` property value.

```
$ svcbundle -o /tmp/ex_con.xml -s service-name=ex_con
      -s rc-script=/etc/init.d/ex_con:2 -s model=contract
```

3. **Make any necessary changes to the manifest.**

   Verify that the content of the `/tmp/ex_con.xml` manifest is what you need. You might need to add a dependency or adjust the method timeout, for example. Add comments to describe what the service does and how the properties of the service are used.

4. **Verify that the manifest is valid.**

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

   ```
   $ svccfg validate /tmp/ex_con.xml
   ```

5. **Copy the manifest to the standard directory.**

   ```
   $ cp /tmp/ex_con.xml /lib/svc/manifest/site/ex_con.xml
   ```

6. **Stop the existing service.**

   ```
   $ /etc/init.d/ex_con stop
   ```

7. **Disable the run control script.**

   Remove any links to the run control script from the appropriate `rc`*n*`.d` directories.

8. **Import the manifest and start the service.**

   ```
   $ svcadm restart manifest-import
   ```

9. **List the new service.**

   Verify that the new service exists and is in the expected state.

   ```
   $ svcs ex-con
   ```

Creating a Service Using Multiple Manifests

If a new package requires a custom configured instance of a service, that package can deliver just the custom instance without modifying any other service.

Using multiple manifests to define a service enables you to deliver service instances only as needed and without modifying the parent service. For example, if instance I of service S is only needed by tool T, then tool T can deliver instance I without redelivering service S. Then if tool

T is not installed on the system, instance I also is not installed, even if service S is installed. Similarly, if tool T is uninstalled, instance I is uninstalled, leaving service S still installed and unmodified. Specify service S in one manifest and instance I in a separate manifest. Deliver the service S manifest in one package, and deliver the instance I manifest in the tool T package. The tool T package has a dependency on the package that delivers service S.

If you use multiple manifests to specify a single service, use the following service manifest design:

- In one manifest, include the service definition, template data, and a default instance.
- In each manifest that defines additional instances of the service, include the following:
  - Specify the same `service_bundle` element as in the manifest that contains the service definition, and then add the `include` attribute. The value of the `include` attribute is the full file name of the manifest that contains the service definition.
  - Specify the same `service` element as in the manifest that contains the service definition. The value of the `name` attribute in the `service` element is exactly the same as in the manifest that contains the service definition.
- Do not deliver the same service or instance in multiple manifests, When this type of conflict is detected, SMF cannot determine which definitions to use, and the instance is placed in the maintenance state.

An example of a service that is delivered in multiple manifests is the `svc:/system/console-login` service. The `console-login` service includes the following instances and manifests:

`svc:/system/console-login:default`

The manifest `/lib/svc/manifest/system/console-login.xml` delivers the service definition, the templates, and the `default` instance.

`svc:/system/console-login:terma`

The manifest `/lib/svc/manifest/system/console-login-terma.xml` delivers the `terma` instance of the `console-login` service.

`svc:/system/console-login:termb`

The manifest `/lib/svc/manifest/system/console-login-termb.xml` delivers the `termb` instance of the `console-login` service.

`svc:/system/console-login:vt?`

The manifest `/lib/svc/manifest/system/console-login-vts.xml` delivers the `vts` instances of the `console-login` service.

The manifest that contains the service definition, `/lib/svc/manifest/system/console-login.xml`, contains the following lines:

```
<service_bundle type="manifest" name="SUNWcs:console">
```

```
<service
        name="system/console-login"
        type="service"
        version="1">

<instance name='default' enabled='true'>
</instance>
```

The manifest that defines additional instances `vt2`, `vt3`, `vt4`, `vt5`, and `vt6`, `/lib/svc/manifest/system/console-login-vts.xml`, contains the following lines:

```
<service_bundle type="manifest" name="SUNWcs:console"
    include="/lib/svc/manifest/system/console-login.xml">

<service
        name="system/console-login"
        type="service"
        version="1">

<instance name='vt2' enabled='true'>

        <dependency
                name='system-console'
                grouping='require_all'
                restart_on='none'
                type='service'>
                <service_fmri value='svc:/system/console-login:default' />
        </dependency>

</instance>
```

The definitions of instances `vt3`, `vt4`, `vt5`, and `vt6` contain the same dependency on `console-login:default` as shown for the `vt2` instance.

The `svcs` command output displays the dependency relationships, as shown in the following examples:

```
$ svcs 'svc:/system/console-login:vt*'
STATE          STIME    FMRI
online         Dec_04   svc:/system/console-login:vt3
online         Dec_04   svc:/system/console-login:vt5
online         Dec_04   svc:/system/console-login:vt2
online         Dec_04   svc:/system/console-login:vt4
online         Dec_04   svc:/system/console-login:vt6
$ svcs -D console-login:default
STATE          STIME    FMRI
online         Dec_04   svc:/system/vtdaemon:default
online         Dec_04   svc:/system/console-login:vt3
online         Dec_04   svc:/system/console-login:vt5
online         Dec_04   svc:/system/console-login:vt2
online         Dec_04   svc:/system/console-login:vt4
```

```
online         Dec_04   svc:/system/console-login:vt6
online         Dec_04   svc:/system/console-reset:default
$ svcs -d 'svc:/system/console-login:vt*'
STATE          STIME    FMRI
...
online         Dec_04   svc:/system/console-login:default
online         Dec_04   svc:/system/vtdaemon:default
```

The `default`, `terma`, and `termb` instances are delivered by the `system/core-os` package. The `vts` instances are delivered by the `system/virtual-console` package. The `system/virtual-console` package contains a `require` dependency on the `system/core-os` package to ensure that the `console-login:default` instance exists.

If a service has only one instance, best practice is to use a single manifest to specify the service: Define the one instance in the manifest that contains the service definition.

♦ ♦ ♦ CHAPTER 3

Creating a Service to Run Periodically

This chapter describes how to create a service that runs a relatively short task at regular intervals. This chapter describes:

- Periodic service definition
- How to create a periodic service
- How to specify the execution schedule of a periodic service

Periodic Services

Most system services are implemented as long-running daemons and run until an administrator intervenes. A *periodic* or *scheduled* service runs a relatively short task at regular intervals. A scheduled service is a type of periodic service. See Chapter 4, “Creating a Service to Run on a Specific Schedule” for more information about scheduled services.

You might want to create a periodic service to perform a task that you previously would have configured `cron` to perform. One advantage of using an SMF service and delivering the service in an IPS package is that you can take advantage of SMF and IPS dependency features to ensure the task only runs when other required software is installed and running. Another advantage is that when the user uninstalls the package, the periodic task is removed and does not need to be separately removed from a `crontab` file.

A periodic service instance is managed by the periodic restarter, `svc.periodicd`, which is invoked by the `svc:/system/svc/periodic-restarter` service at system startup. The periodic restarter runs the start method for the instances it manages at scheduled intervals whenever the instance is in the `online` state. A periodic instance that is enabled transitions to the `online` state as soon as all of the dependencies of the instance are met. If no errors or administrative interventions occur, the periodic service remains in the `online` state between runs of the start method, when no processes associated with the method are running. See the `svc.periodicd`(1M) man page for more information.

Creating a Periodic Service

A periodic service manifest is very simple, as shown in the following example. See also “Creating a Periodic Service Using the Service Bundle Generator Tool” on page 36.

EXAMPLE 2 Periodic Service Manifest

The periodic service instance is completely defined in the `periodic_method` element. For this example service, the periodic restarter executes the start method every 30-35 seconds after an initial delay of 15 seconds. The `template` element is recommended to help administrators understand the purpose of this periodic service.

```
<?xml version='1.0'?>
<!DOCTYPE service_bundle
  SYSTEM '/usr/share/lib/xml/dtd/service_bundle.dtd.1'>
<service_bundle type='manifest' name='site/sample-periodic-svc'>
    <service type='service' version='1' name='site/sample-periodic-svc'>

        <instance name='default' enabled='false'>

            <periodic_method
                period='30'
                delay='15'
                jitter='5'
                exec='/usr/bin/periodic_service_method'
                timeout_seconds='0'>
                    <method_context>
                        <method_credential user='root' group='root' />
                    </method_context>
            </periodic_method>

        </instance>

        <template>
            <common_name>
                <loctext xml:lang="C">
                    Sample Periodic Service
                </loctext>
            </common_name>
            <description>
                <loctext xml:lang="C">
                    What this service does periodically.
                </loctext>
            </description>
        </template>
    </service>
</service_bundle>
```

Specifying the `periodic_method` Element

When a `periodic_method` element exists, the instance is automatically delegated to the periodic restarter. The `periodic_method` element can be specified within a `service` element or within an `instance` element. The `periodic_method` element specifies both method and scheduling information for periodic services and can have the attributes and elements described in this section. The `period` and `exec` attributes are required.

Periodic Service Scheduling Constraints Attributes

The units for a `time` value are seconds, as shown in Table 2, “Service Property Type Value Descriptions,” on page 21.

`delay` attribute

Optional. Value type: `time`. Default value: 0. The fixed number of seconds after the service has transitioned to the `online` state before the first invocation of the start method.

`period` attribute

Required. Value type: `time`. The number of seconds between invocations of the start method.

`jitter` attribute

Optional. Value type: `time`. Default value: 0. The maximum of a random number of seconds after `period` before the start method is run. The final number of seconds that is used ranges between 0 and the value of this property.

Other Periodic Service Scheduling Attributes

`persistent` attribute

Optional. Value type: `boolean`. Default value: `false`. Specifies whether scheduling should be maintained across system downtime.

If the value is `false`, scheduling of the start method restarts as if the periodic service instance has just transitioned to the `online` state after being enabled.

If the value of the `persistent` attribute is `true` and the value of the `recover` attribute (below) is `false`, scheduling of the start method for the instance emerging from downtime continues on the same schedule that was defined before the downtime occurred.

If the value of the `persistent` attribute is `true` and the value of the `recover` attribute is `true`, see the description of the `recover` attribute below.

`recover` attribute

Optional. Value type: `boolean`. Default value: `false`. Specifies whether the instance should have a recovery execution if an invocation was lost during system downtime.

This value has effect only if the value of the `persistent` attribute for this instance is `true`.

If the value of the `persistent` attribute is `true` and the value of the `recover` attribute is `true`, the periodic restarter invokes the start method for the instance as soon as possible as the instance emerges from system downtime. Subsequent invocations occur according to the `period` and `jitter` values.

If the value of the `persistent` property is `true` and the value of the `recover` property is `false`, see the description of the `persistent` property above.

Periodic Service Start Method Attributes and Context

`exec` attribute

Required. Value type: `astring`. The action to take, which must be suitable to pass to the `exec` system call. See the `smf_method`(5) man page.

This start method should perform a task and then terminate within the time specified by the `period` attribute.

The `SMF_EXIT_TEMP_TRANSIENT` exit code does not apply to periodic service start methods because the periodic restarter does not implement transient services. When used in a periodic service start method, the `SMF_EXIT_TEMP_TRANSIENT` exit code is treated the same as the `SMF_EXIT_ERR_OTHER` exit code.

Periodic service instances use only a start method. If any refresh or stop method is defined, a warning message is issued at manifest import and the refresh or stop method is ignored. When a periodic instance is refreshed, the periodic restarter rereads the values of the properties in the `periodic` property group described in "Storing Periodic Service Data in the Service Configuration Repository" on page 35. The periodic restarter does not need a stop method because processes contracted by a periodic instance do not run persistently. Periodic instances run short-lived processes and then wait until the next scheduled time to run.

`timeout_seconds` attribute

Optional. Value type: `integer`. The number of seconds to wait for the method action to complete. Use a value of `0` or `-1` to specify an infinite timeout.

The `timeout_seconds` attribute value is required by the `exec_method` element. If you do not specify a value for the `timeout_seconds` attribute in this `periodic_method` element, the value for the `exec_method` element is assumed to be infinite. See "Scheduling After Start Method Problems" on page 40 for a description of start method scheduling if the start method runs longer than the specified `timeout_seconds` value or if a contracted process still exists when the periodic restarter attempts to invoke the start method for the next period.

`method_context` element

Optional. See the Method Context section of the `smf_method(5)` man page.

Storing Periodic Service Data in the Service Configuration Repository

As described in "Specifying the `periodic_method` Element" on page 33, the `periodic_method` element provides both method information (an `exec_method` property group for the start method) and scheduling information (a `periodic` property group) for periodic services. When a manifest with a `periodic_method` element is imported, the data described in this section is stored in the service configuration repository.

Restarter Properties

The restarter for the service is set to `svc:/system/svc/periodic-restarter:default`. Administrators can use the `svcs -l` command to show the restarter or use the `svccfg` or `svcprop` command to view the `general/restarter` property.

Because a periodic service instance remains online between invocations of the start method, the instance can be in the `online` state with no associated contracted processes running on the system. For an online periodic service instance, the `auxiliary_state` property can have one of the following values to distinguish whether the method action is running:

`running` The instance is online and has associated contracted processes.

`scheduled` The instance is online but has no associated contracted processes.

To check this state, administrators can use the `svcs -o astate` command to show the ASTATE column or use the `svccfg` or `svcprop` command to view the `general/auxiliary_state` property.

`periodic` Property Group

The scheduling attributes of the `periodic_method` element (`period`, `delay`, `jitter`, `persistent`, and `recover`) are stored as properties of a property group named `periodic`. See "Specifying the `periodic_method` Element" on page 33 for definitions of these attributes and see "Scheduling Executions of a Periodic Service Start Method" on page 38 for

examples of how they are used. Administrators can use the `svcprop` and `svccfg` commands to show and modify these `periodic` property group properties.

Last and Next Start Method Invocations

The service configuration repository also stores the following two pieces of scheduling information for the instance:

`last_run` — The absolute time of the last attempt to run the start method of this instance. If the start method has never run, this property does not exist. To check this time, administrators can use the `svcs -o lrun` command to show the LRUN column.

`next_run` — The absolute time of the next attempt to run the start method of this instance. If an absolute time does not exist, this property might be absent or have no value. The value of `next_run` is computed at the time `last_run` is set and is the next scheduled start method invocation as described in "Scheduling Executions of a Periodic Service Start Method" on page 38, including the specific `RAND(`*jitter*`)` value for that invocation. To check this time, administrators can use the `svcs -o nrun` command to show the NRUN column. This time is managed by the `periodic-restarter` service for each periodic service instance. Administrators cannot modify this value.

`start` Property Group

The `exec` and `timeout_seconds` values and `method_context` information are stored as properties of a property group named `start`. This `start` property group represents the start method for the periodic service and is defined in the same way as the start method for any other service.

Creating a Periodic Service Using the Service Bundle Generator Tool

When you use the `svcbundle` command to create a periodic service, you must specify the `period` property as well as both the `service-name` and `start-method` properties. By default, `svcbundle` creates a `transient` service. When you specify `-s period`, `svcbundle` creates a periodic service.

▼ How to Create a Periodic Service Using `svcbundle`

1. Copy the start method to the standard location.

In this example, the start method for this service is named `per_ex`. Copy this executable to `/lib/svc/method/per_ex`.

2. Create an initial manifest.

In this example, the service name is `site/per_ex`. Specify a period for the start method scheduling. Do *not* specify any of the following properties: `bundle-type`, `duration`, `model`, `rc-script`, `refresh-method`, or `stop-method`.

```
$ svcbundle -o /tmp/per_ex.xml -s service-name=site/per_ex \
      -s start-method=/lib/svc/method/per_ex -s period=30
```

When you specify the `period` property, `svcbundle` creates a `periodic_method` element, which causes the restarter for the service to be set to the periodic restarter when the manifest is imported. The value of the `period` property becomes the value of the `period` attribute of the `periodic_method` element. The value of the `start-method` property becomes the value of the `exec` attribute of the `periodic_method` element.

3. Make any necessary changes to the manifest.

Verify that the content of the `/tmp/ex_svc.xml` manifest is what you need. You might want to make changes such as the following:

- Add values for `delay`, `jitter`, and `timeout_seconds` in the `periodic_method` element.
- Add a `method_context` element in the `periodic_method` element.
- Change the value of the `enabled` attribute of the default instance from `true` to `false`.

Add comments to describe what the service does and how the properties of the service are used.

4. Verify that the manifest is valid.

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

5. Copy the manifest to the standard directory.

```
$ cp /tmp/per_ex.xml /lib/svc/manifest/site/per_ex.xml
```

6. Import the manifest and start the service.

```
$ svcadm restart manifest-import
```

7. List the new service.

Verify that the new service exists and is in the expected state.

```
$ svcs per_ex
```

Scheduling Executions of a Periodic Service Start Method

Scheduling executions of the start method of a periodic service instance (specified by the `exec` attribute of the `periodic_method` element) always requires the value of the `period` attribute of the `periodic_method` element. Other attributes of the `periodic_method` element might also be used, and the `next_run` value might be used.

The scheduling attributes of the `periodic_method` element (`period`, `delay`, `jitter`, `persistent`, and `recover`) are stored as properties of the `periodic` property group. For example, the `period` attribute in the manifest becomes the `periodic/period` property when the periodic service is imported. Administrators can use the `svcprop`and `svccfg` commands to view property values in the `periodic` property group and can use the `svccfg` command to modify these values.

The `period` property must have a valid value. The `jitter`, `delay`, `persistent`, and `recover` properties have default values and are not required to be explicitly set. The units for a `time` value are seconds, as shown in Table 2, “Service Property Type Value Descriptions,” on page 21.

In the following scheduling descriptions, italic type indicates the value of the property. For example, *jitter* is the value of the `periodic/jitter` property in the service configuration repository, which is the same as the value of the `jitter` attribute of the `periodic_method` element in the service manifest.

Scheduling After Instance is Initially Enabled

When a periodic service instance is initially enabled (for example at system boot), the first execution of the start method of the instance occurs at the following number of seconds after the instance transitions to the `online` state:

delay + `RAND(`*jitter*`)`

Subsequent executions of the start method of the instance occur at the following relative time:

period + `RAND(`*jitter*`)`

To prevent schedule drift, subsequent executions ignore the *jitter* value of previous executions. For example, the second execution of the start method of an instance occurs at the following number of seconds after the instance transitioned to the `online` state:

delay + *period* + `RAND(`*jitter*`)`

The *n*th execution of the start method of an instance occurs at the following number of seconds after the instance transitioned to the `online` state:

delay + `(`*n*`-1)`*period* + `RAND(`*jitter*`)`

Scheduling After System Downtime

The `persistent` and `recover` properties can modify the time the periodic instance method runs when a periodic instance emerges from system downtime. The value of the `recover` property has effect only if the value of the `persistent` property is `true`.

If the value of the `persistent` property is `false`, the next execution of the start method for a periodic instance emerging from system downtime occurs at the following number of seconds after the instance transitions to the `online` state:

delay + `RAND(`*jitter*`)`

If the value of the `persistent` property is `true` and the value of the `recover` property is `false`, the next execution of the start method for the instance emerging from system downtime is scheduled for the following absolute time:

next_run + `(`*n*`)`*period*

If the value of the `next_run` property is in the future, *n* is 0. If the value of the `next_run` property is in the past, the smallest value of *n* is used that results in a future time. Subsequent invocations occur from that time according to the `period` and `jitter` values. The `next_run` property is described in "Last and Next Start Method Invocations" on page 36.

If the value of the `persistent` property is `true` and the value of the `recover` property is `true`, the periodic restarter invokes the start method for the instance as soon as possible as the instance emerges from system downtime. Subsequent invocations occur from that time according to the `period` and `jitter` values.

Scheduling After Service Restart

The periodic restarter service (`svc:/system/svc/periodic-restarter`) automatically attempts to restart if it terminates. When the periodic restarter service restarts after failure, the start method of each periodic instance is scheduled for the following absolute time:

next_run + `(`*n*`)`*period*

If the value of the `next_run` property is in the future, *n* is 0. If the value of the `next_run` property is in the past, the smallest value of *n* is used that results in a future time. Subsequent invocations occur from that time according to the `period` and `jitter` values. The `next_run` property is described in "Last and Next Start Method Invocations" on page 36.

If a periodic service instance is restarted, the start method for the instance is invoked at the following number of seconds after the instance transitions to the `online` state:

delay + `RAND(`*jitter*`)`

Scheduling After Start Method Problems

The start method should perform a task and then terminate within the time specified by the `period` property. If a contracted process still exists when the periodic restarter attempts to invoke the start method for the next period, then the invocation for that period is skipped and the periodic restarter attempts to invoke the start method again at the following period. The periodic restarter invokes the method again at the following number of seconds after the instance transitioned to the `online` state, where *n* is the smallest value that results in a future time:

delay + (*n*)*period* + RAND(*jitter*)

If the start method runs longer than the number of seconds specified by the `timeout_seconds` value, all processes in the contract are terminated and the invocation is a non-fatal fault. The first time the start method terminates in any non-fatal fault, the instance is placed into the `degraded` state and start method invocations continue as scheduled. If one of the following two invocations succeeds, the instance is placed back into the `online` state. After transitioning from the `degraded` state to the `online` state, the periodic restarter invokes the method at the following number of seconds after the instance initially transitioned from the `offline` to the `online` state, where *n* is the smallest value that results in a future time:

delay + (*n*)*period* + RAND(*jitter*)

After three successive non-fatal faults of the start method, the instance is moved from the `degraded` state to the `maintenance` state. On the first fatal fault of the start method, the service is placed into the `maintenance` state. See “Repairing an Instance That Is Degraded, Offline, or in Maintenance” in *Managing System Services in Oracle Solaris 11.3*.

♦ ♦ ♦ CHAPTER 4

Creating a Service to Run on a Specific Schedule

This chapter describes how to create a service that runs a relatively short task at a specified regular time. This chapter describes:

- Scheduled service definition
- How to create a scheduled service
- How to specify the execution schedule of a scheduled service

Scheduled Services

A *scheduled* service is a type of *periodic* service, described in Chapter 3, "Creating a Service to Run Periodically". Each invocation of a periodic service instance start method occurs at a time relative to the last invocation. Each invocation of a scheduled service instance start method occurs at a specific absolute time. Use a scheduled service when the task must run at a certain time, such as during off-peak hours.

A scheduled service instance is managed by the periodic restarter service, `svc:/system/svc/periodic-restarter`. The start method scheduling parameters for a scheduled service instance are defined in a `scheduled_method` element and zero or more `schedule` property groups in the service manifest. The start method scheduling parameters are stored in property groups of type `schedule` in the service configuration repository. The periodic restarter combines the values of each property of each property group of type `schedule` to schedule invocations of the start method of the scheduled service instance.

Creating a Scheduled Service

The following sample manifest is for a very simple scheduled service. See also "Creating a Scheduled Service Using the Service Bundle Generator Tool" on page 46.

EXAMPLE 3 Scheduled Service Manifest

The scheduled service instance in this example is completely defined in the `scheduled_method` element. A scheduled service should use the fewest scheduling constraints required to schedule the task. For example, the periodic restarter will execute the start method for the following example service on the first day of each month, between 02:00 and 03:00. To invoke the method between 02:00 and 02:01, add the constraint `minute='0'`.

```
<?xml version='1.0'?>
<!DOCTYPE service_bundle
  SYSTEM '/usr/share/lib/xml/dtd/service_bundle.dtd.1'>
<service_bundle type='manifest' name='site/sample-periodic-svc'>
    <service type='service' version='1' name='site/sample-periodic-svc'>

        <instance name='default' enabled='false'>

            <scheduled_method
                interval='month'
                day='1'
                hour='2'
                exec='/usr/bin/scheduled_service_method'
                timeout_seconds='0'>
                    <method_context>
                        <method_credential user='root' group='root' />
                    </method_context>
            </scheduled_method>

        </instance>
    </service>
</service_bundle>
```

Specifying the `scheduled_method` Element

When a `scheduled_method` element exists, the service is automatically delegated to the periodic restarter. The `scheduled_method` element can be specified within a `service` element or within an `instance` element. The `scheduled_method` element specifies both method and scheduling information for scheduled services. The method information is the same for a `scheduled_method` element as for a `periodic_method` element. The `scheduled_method` element can have the attributes that are listed in this section. The `interval` and `exec` attributes are required.

When specifying values of scheduling constraints for a scheduled service, note the following:

- Specify the fewest scheduling constraints that are required to schedule the task. Constraints that represent shorter time periods than the `interval` value specify more precisely when in the interval to invoke the start method for the instance.

- Specify a continuous set of scheduling constraints from the constraint that represents the longest time to the constraint that represents the shortest time. For example, you can specify just a `week_of_year` value, but you cannot specify a `week_of_year` and an `hour` unless you also specify a `day_of_month` or `day`.
- You can specify constraints that represent time periods that are equal to or longer than the `interval` value if the value of `frequency` is greater than `1`. See "Scheduling Executions of a Scheduled Service Start Method" on page 47 for more information about how to use these constraints.

Scheduled Service Scheduling Constraints Attributes

`interval`

Required. Value type: `astring`. The base length of time between invocations of the service start method, depending on the value of `frequency`. The value of `interval` is one of the following: `year`, `month`, `week`, `day`, `day_of_month`, `hour`, `minute`.

`frequency`

Optional. Value type: `count`. Default value: 1. The number of intervals that must occur before the start method is executed by the periodic restarter. For example, if `interval` is `week` and `frequency` is `4`, the start method will be invoked every fourth week. Note that every fourth week is different from the fourth week of every month because some months have five weeks.

If the value of `frequency` is greater than `1`, all constraints from `year` down to the same length of time as the `interval` value must be specified. For example, if `interval` is `week`, values must be specified for `year` and `week_of_year`. See "Scheduling One Invocation Per Multiple Intervals" on page 48 for examples.

`timezone`

Optional. Value type: `astring`. Default value: the system time zone. The time zone to use to create and interpret schedules. Use this attribute to define a schedule relative to a time zone other than the time zone configured for the system.

`year`

Value type: `count`. A Gregorian calendar year. The year in which to invoke the start method.

`week_of_year`

Value type: `integer`. The ordinal number of the week in an ISO 8601 week date year. Valid values are `1` through `53` and `-1` through `-53`. A negative number specifies weeks backward from the last week of the year. For a year that has 52 weeks, `-1` is the same as `52`, and for a year that has 53 weeks, `-1` is the same as `53`. If you specify `53` or `-53`, the start method will not run in years that have only 52 weeks.

The `week_of_year` and `month` attributes are mutually exclusive.

`month`

Value type: `astring`. A month of a Gregorian year. Valid values are `1` through `12`, `-1` through `-12`, the C Locale full name of the month, or the C Locale three-character abbreviation for the name of the month. A negative number specifies months backward from the last month of the year. C Locale names are not case sensitive.

The `week_of_year` and `month` attributes are mutually exclusive.

`day_of_month`

Value type: `integer`. A day in a Gregorian calendar month. Valid values are `1` through `31` and `-1` through `-31`. A negative number specifies days backward from the last day of the month. If you specify a day that does not exist in a particular month, for example `31` or `-1` for April, the start method will run on the last day of that month.

The `day_of_month` and `day` attributes are mutually exclusive.

`weekday_of_month`

Value type: `integer`. The ordinal number of the week in a month in which the specified day (see `day`) occurs. Valid values are `1` through `5` and `-1` through `-5`. A negative number specifies weeks backward from the last week of the month. For example, the third Thursday in the month (`3`) is different from the next-to-last Thursday in the month (`-2`) if the month has five Thursdays. If you specify `5` or `-1`, the start method will not run in months that have only four of the specified `day`.

If `weekday_of_month` is specified, `day` must also be specified.

`day`

Value type: `astring`. A day in an ISO 8601 standard week. Valid values are `1` through `7` and `-1` through `-7`, the C Locale full name of the day, or the C Locale three-character abbreviation of the name of the day. A negative number specifies days backward from the end of the week. C Locale names are not case sensitive.

The `day_of_month` and `day` attributes are mutually exclusive.

`hour`

Value type: `integer`. An hour of an ISO 8601 standard day. Valid values are `0` through `23` and `-1` through `-24`. A negative number specifies hours backward from the end of the day. If the scheduled day includes a transition between standard time and daylight saving time, the specified hour might occur two times that day or might not occur at all that day. If the specified hour occurs more than one time on the scheduled day, the start method will run only on the first occurrence of the hour. If the specified hour does not occur on the scheduled day, the start method will run in the following hour.

`minute`

Value type: `integer`. A minute of an hour. Valid values are `0` through `59` and `-1` through `-60`. A negative number specifies minutes backward from the end of the hour.

Other Scheduled Service Scheduling Attributes

`recover`

Optional. Value type: `boolean`. Default value: `false`. Specifies whether the instance should have a recovery execution if an invocation was lost during system downtime.

If the value of the `recover` attribute is `true`, the periodic restarter invokes the start method for the instance as soon as possible as the instance emerges from system downtime. Subsequent invocations occur according to the specified scheduling constraints.

If the value of the `recover` property is `false`, no recovery invocation is executed. Invocations continue according to the specified scheduling constraints.

Scheduled Service Start Method Attributes and Context

`exec`

Required. Value type: `astring`. The action to take, which must be suitable to pass to the `exec` system call. See the `smf_method(5)` man page. See additional description in "Specifying the `periodic_method` Element" on page 33.

`timeout_seconds`

Optional. Value type: `integer`. The number of seconds to wait for the start method action to complete. Use a value of `0` or `-1` to specify an infinite timeout.

The `timeout_seconds` attribute value is required by the `exec_method` element. If you do not specify a value for the `timeout_seconds` attribute in this `scheduled_method` element, the value for the `exec_method` element is assumed to be infinite. See "Scheduling After Start Method Problems" on page 52 for a description of start method scheduling if the start method runs longer than the specified `timeout_seconds` value or if a contracted process still exists when the periodic restarter attempts to invoke the start method in the next scheduled interval.

`method_context` element

Optional. See the Method Context section of the `smf_method(5)` man page.

Storing Scheduled Service Data in the Service Configuration Repository

The `scheduled_method` element provides both method information (an `exec_method` property group for the start method) and scheduling information (property groups of type `schedule`) for scheduled services. When a manifest with a `scheduled_method` element is imported, the data described in this section is stored in the service configuration repository.

- **Restarter properties.** See "Restarter Properties" on page 35 for information about the `restarter` and `auxiliary_state` properties. Recall that a scheduled service is a type of periodic service.
- `schedule` **property groups.** The scheduling attributes of the `scheduled_method` element are stored as properties of a property group of type `schedule`. Property groups of type `schedule` can have the properties described in "Scheduled Service Scheduling Constraints Attributes" on page 43 and "Other Scheduled Service Scheduling Attributes" on page 45. See "Scheduling Invocations at Irregular Intervals" on page 49 and "Scheduling Executions of a Scheduled Service Start Method" on page 47 for examples of how these properties are used. Administrators can use the `svcprop` and `svccfg` commands to show and modify these properties. Use the `svcprop -g schedule` command to list all the properties of property groups of type `schedule`, as described in "Showing Properties in a Property Group Type" in *Managing System Services in Oracle Solaris 11.3*.
- **Last and next start method invocations.** See "Last and Next Start Method Invocations" on page 36 for descriptions of the `last_run` and `next_run` properties. The value of `next_run` is the next scheduled start method invocation as described in "Scheduling Executions of a Scheduled Service Start Method" on page 47.
- `start` **property group.** The `exec` and `timeout_seconds` values and `method_context` information are stored as properties of a property group named `start`. This `start` property group represents the start method for the scheduled service and is defined in the same way as the start method for any other service.

Creating a Scheduled Service Using the Service Bundle Generator Tool

When you use the `svcbundle` command to create a scheduled service, you must specify the `interval` property as well as both the `service-name` and `start-method` properties. By default, `svcbundle` creates a `transient` service. When you specify `-s period`, `svcbundle` creates a periodic service.

▼ How to Create a Scheduled Service Using `svcbundle`

1. **Copy the start method to the standard location.**

   In this example, the start method for this service is named `sched_ex`. Copy this executable to `/lib/svc/method/sched_ex`.

2. **Create an initial manifest.**

   In this example, the service name is `site/sched_ex`. Specify an interval for the start method scheduling. Do *not* specify any of the following properties: `bundle-type`, `duration`, `model`, `rc-script`, `refresh-method`, or `stop-method`.

```
$ svcbundle -o /tmp/sched_ex.xml -s service-name=site/sched_ex \
      -s start-method=/lib/svc/method/sched_ex -s interval=week
```

When you specify the `interval` property, `svcbundle` creates a `scheduled_method` element, which causes the restarter for the service to be set to the periodic restarter when the manifest is imported. The value of the `interval` property becomes the value of the `interval` attribute of the `scheduled_method` element. The value of the `start-method` property becomes the value of the `exec` attribute of the `scheduled_method` element.

3. **Make any necessary changes to the manifest.**

   Verify that the content of the `/tmp/ex_svc.xml` manifest is what you need. You might want to make changes such as the following:

   - Specify additional constraints in the `scheduled_method` element.
   - Specify additional property groups of type `schedule`, as described in "Scheduling Invocations at Irregular Intervals" on page 49.
   - Add a `method_context` element in the `scheduled_method` element.
   - Change the value of the `enabled` attribute of the default instance from `true` to `false`.

   Add comments to describe what the service does and how the properties of the service are used.

4. **Verify that the manifest is valid.**

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

5. **Copy the manifest to the standard directory.**

   ```
   $ cp /tmp/sched_ex.xml /lib/svc/manifest/site/sched_ex.xml
   ```

6. **Import the manifest and start the service.**

   ```
   $ svcadm restart manifest-import
   ```

7. **List the new service.**

   Verify that the new service exists and is in the expected state.

   ```
   $ svcs sched_ex
   ```

Scheduling Executions of a Scheduled Service Start Method

Scheduling executions of the start method of a scheduled service instance (specified by the `exec` attribute of the `scheduled_method` element) always requires the values of the `interval` and

frequency attributes of the scheduled_method element. The frequency attribute has a default value of 1. The value of the interval attribute must be explicitly set. Other specified attributes of the scheduled_method element or properties of property groups of type schedule are also used, and the next_run value might be used.

Scheduling One Invocation Per Interval

If the value of frequency is 1, the start method of a scheduled service instance is initially invoked at a random time within the smallest specified constraint. Subsequent invocations occur within the same unit of the next shorter constraint in the next execution interval. For example, if the interval is week and no other scheduling constraints are specified, the start method will initially be invoked at a random time during the week. Subsequent invocations will occur within the same day in future weeks so that the start method will not be invoked on two consecutive days, for example. Similarly, if the interval is week, and day and hour are also specified, the start method will initially be invoked at a random time during the specified hour on the specified day; subsequent invocations will occur within the same minute during the specified hour and day in future weeks.

Scheduling One Invocation Per Multiple Intervals

If the value of frequency is greater than 1, scheduling constraints that are the same length of time or longer than the interval value are used to set the time of the initial invocation. All constraints from year down to the same length of time as the interval value must be specified. For example, if interval is week, values must be specified for year and week_of_year.

If interval is week, frequency is 2, year is 2014, and week_of_year is any even number from 2 through 52, then the start method will be invoked every even-numbered week. If week_of_year is any odd number from 1 through 53, then the start method will be invoked every odd-numbered week. Because the year 2014 has already passed, the value of the week_of_year attribute in this example does not indicate the particular week when the start method will initially be invoked, but only whether it will be invoked in the next even week or the next odd week. Subsequent invocations will occur every two weeks, so that after a year that has 53 weeks, such as 2015, the start method that was initially invoked in an even-numbered week will be invoked in odd-numbered weeks.

If interval is week, frequency is 4, year is 2014, and week_of_year is 1, the start method will be invoked in weeks 1, 5, 9, and so forth in 2015. Because 2015 has 53 weeks, the start method will be invoked in week 53, and in 2016 the start method will be invoked in weeks 4, 8, 12, and so forth.

Scheduling Invocations at Irregular Intervals

Sometimes one set of scheduling constraints is not enough to define the schedule. A scheduled service can have multiple property groups of type `schedule`. The periodic restarter combines the values of each property of each property group of type `schedule` to schedule invocations of the start method of the scheduled service instance.

Just as a scheduled service should have only as many scheduling constraints as necessary to define the schedule, a scheduled service should have only as many `schedule` property groups as necessary to define the required schedule for the task.

EXAMPLE 4 Invoking Every Twelve Hours

In this example, the start method is invoked at 06:00 and 18:00 every day. The value of the `hour` attribute could be either `6` or `18`.

```
<?xml version='1.0'?>
<!DOCTYPE service_bundle
  SYSTEM '/usr/share/lib/xml/dtd/service_bundle.dtd.1'>
<service_bundle type='manifest' name='site/sample-periodic-svc'>
    <service type='service' version='1' name='site/sample-periodic-svc'>

        <instance name='default' enabled='false'>

            <scheduled_method
                interval='hour'
                frequency='12'
                hour='6'
                minute='0'
                exec='/usr/bin/scheduled_service_method'
                timeout_seconds='0'>
                    <method_context>
                        <method_credential user='root' group='root' />
                    </method_context>
            </scheduled_method>

        </instance>
    </service>
</service_bundle>
```

EXAMPLE 5 Invoking Daily at 03:00 and 23:00

In this example, an additional `schedule` property group is specified to invoke the start method at 03:00 and 23:00 every day.

```
<?xml version='1.0'?>
<!DOCTYPE service_bundle
```

```
  SYSTEM '/usr/share/lib/xml/dtd/service_bundle.dtd.1'>
<service_bundle type='manifest' name='site/sample-periodic-svc'>
    <service type='service' version='1' name='site/sample-periodic-svc'>

        <instance name='default' enabled='false'>

            <scheduled_method
                interval='day'
                hour='3'
                minute='0'
                exec='/usr/bin/scheduled_service_method'
                timeout_seconds='0'>
                    <method_context>
                        <method_credential user='root' group='root' />
                    </method_context>
            </scheduled_method>

            <property_group name='run2' type='schedule'>
                <propval name='interval' type='astring' value='day' />
                <propval name='hour' type='integer' value='23' />
                <propval name='minute' type='integer' value='0' />
            </property_group>

        </instance>
    </service>
</service_bundle>
```

EXAMPLE 6 Invoking at 03:00 and 23:00 on Tuesday and Thursday

To invoke the start method at 03:00 and 23:00 every Tuesday and Thursday requires three additional `schedule` property groups, as shown in this example.

```
<?xml version='1.0'?>
<!DOCTYPE service_bundle
  SYSTEM '/usr/share/lib/xml/dtd/service_bundle.dtd.1'>
<service_bundle type='manifest' name='site/sample-periodic-svc'>
    <service type='service' version='1' name='site/sample-periodic-svc'>

        <instance name='default' enabled='false'>

            <scheduled_method
                interval='week'
                day='3'
                hour='3'
                minute='0'
                exec='/usr/bin/scheduled_service_method'
                timeout_seconds='0'>
                    <method_context>
                        <method_credential user='root' group='root' />
                    </method_context>
```

```
            </scheduled_method>

            <property_group name='run2' type='schedule'>
                <propval name='interval' type='astring' value='week' />
                <propval name='day' type='astring' value='3' />
                <propval name='hour' type='integer' value='23' />
                <propval name='minute' type='integer' value='0' />
            </property_group>

            <property_group name='run3' type='schedule'>
                <propval name='interval' type='astring' value='week' />
                <propval name='day' type='astring' value='5' />
                <propval name='hour' type='integer' value='3' />
                <propval name='minute' type='integer' value='0' />
            </property_group>

            <property_group name='run4' type='schedule'>
                <propval name='interval' type=astring'' value='week' />
                <propval name='day' type='astring' value='5' />
                <propval name='hour' type='integer' value='23' />
                <propval name='minute' type='integer' value='0' />
            </property_group>

        </instance>
    </service>
</service_bundle>
```

If the service is already imported, you can specify these additional property groups by using a site profile or by using the `svccfg` subcommands `addpg` and `setprop`.

Resolving Multiple Possible Invocations in One Interval

If scheduling constraints are specified such that multiple start method invocation times are possible in a single interval, the start method will be invoked at the earliest time that matches all constraints. An example of multiple possible invocation times in a single interval is when switching from daylight saving time to standard time, when times between 01:00 and 02:00 occur twice.

Scheduling After System Downtime

The `recover` property can modify the time the scheduled instance method runs when a scheduled instance emerges from system downtime.

If the value of the `recover` property is `true`, the periodic restarter invokes the start method for the scheduled service instance as soon as possible as the instance emerges from system

downtime. Subsequent invocations occur according to the `interval` and `frequency` values and any other specified constraints.

If the value of the `recover` property is `false`, the periodic restarter invokes the start method for the scheduled service instance at the time specified by the `next_run` property. If the value of the `next_run` property is in the past, future start method invocations occur according to the `interval` and `frequency` values and any other specified constraints. The `next_run` property is described in "Last and Next Start Method Invocations" on page 36.

Scheduling After Service Restart

The periodic restarter service (`svc:/system/svc/periodic-restarter`) automatically attempts to restart if it terminates. When the periodic restarter service restarts after failure, the start method of each scheduled instance is invoked at the time specified by the `next_run` property. If the value of the `next_run` property is in the past, future start method invocations occur according to the `interval` and `frequency` values and any other specified constraints. The `next_run` property is described in "Last and Next Start Method Invocations" on page 36.

If a scheduled service instance is restarted, the start method for the instance is invoked according to the `interval` and `frequency` values and any other specified constraints.

Scheduling After Start Method Problems

The start method should perform a task and then terminate within the period specified by the `interval` and `frequency` properties. If a contracted process still exists when the periodic restarter attempts to invoke the start method in the next scheduled interval, then the invocation for that period is skipped and the periodic restarter attempts to invoke the start method again at the following period.

If the start method runs longer than the number of seconds specified by the `timeout_seconds` value, all processes in the contract are terminated and the invocation is a non-fatal fault. The first time the start method terminates in any non-fatal fault, the instance is placed into the `degraded` state and start method invocations continue as scheduled according to the `interval` and `frequency` values and any other specified constraints. If one of the following two invocations succeeds, the instance is placed back into the `online` state.

After three successive non-fatal faults of the start method, the instance is moved from the `degraded` state to the `maintenance` state. On the first fatal fault of the start method, the service is placed into the `maintenance` state. See "Repairing an Instance That Is Degraded, Offline, or in Maintenance" in *Managing System Services in Oracle Solaris 11.3*.

♦ ♦ ♦ CHAPTER 5

Create Services to Manage Oracle Database Instances

This chapter presents the following services that help manage the Oracle Database:

- A database service that starts or stops an Oracle Database instance
- A listener service that starts the listener, which is a process that manages the incoming traffic of client connection requests to the database instance

Configuring the Environment

The examples in this chapter use file-backed storage. An alternative to using file-backed storage is to use the Automatic Storage Management (ASM) feature. ASM is a volume manager and a file system for Oracle Database files.

The following environment variables must be set for each installation of the Oracle Database:

`ORACLE_HOME`	The location where the database is installed. In the example in this section, the location of the database installation is `/opt/oracle/product/home`.
`ORACLE_SID`	The systems ID to uniquely identify a particular database on a system.

In the examples in this chapter, these environment variables are set in the service manifests and then used in the method scripts.

Creating a Service to Start or Stop an Oracle Database Instance

This section describes the Oracle Database instance control service manifest and the start/stop method script that is used in that manifest.

Instance Control Service Manifest

The following are some features to note about the Oracle Database instance control service manifest, `/lib/svc/manifest/site/oracle.xml`:

- One service instance is defined, named `svc:/site/application/database/oracle:default`. This instance is enabled by default. See the `create_default_instance` element at the top of the manifest.
- This service requires all local file systems to be mounted. If you are using a file-backed database, the database service should depend on the local filesystem. If you are using ASM, the database service should depend on the service that manages ASM.
- This service depends on the listener service to allow it to establish remote client connections. See the `require_all` dependency on the `network` milestone in "Listener Service Manifest" on page 57.
- The `method_environment` element in the `method_context` element defines the `ORACLE_HOME` and `ORACLE_SID` environment variables, which identify the database instance to start or stop. These values are then available for the method script to use.

  If you create multiple instances of this service, then each instance might need its own `method_context` element to define the unique `ORACLE_HOME` and `ORACLE_SID` values for that particular database.
- Attributes of the `method_context` element can define a resource pool in addition to the project and working directory shown in this example. You can also define either a `method_profile` or `method_credential` element in the `method_context` element. The `method_credential` element can specify `supp_groups` and `limit_privileges` values in addition to the `user`, `group`, and `privileges` values shown in this example. See the DTD for more information.
- The start/stop method script is `/lib/svc/method/oracle`. The number of seconds before the method times out is increased from the default.
- A user must be assigned the `solaris.smf.manage.oracle` authorization to enable or disable this service instance. In this example, the user `oracle` is assigned the `solaris.smf.manage.oracle` authorization.

```
<?xml version="1.0"?>

<!--
 Define a service to control the startup and shutdown of a database instance.
-->

<!DOCTYPE service_bundle SYSTEM "/usr/share/lib/xml/dtd/service_bundle.dtd.1">

<service_bundle type="manifest" name="oracle">
<service name="site/application/database/oracle" type="service" version="1">
    <create_default_instance enabled="true" />

    <!-- Using a file-backed database, so depend on the filesystem. -->
```

```
    <dependency type="service"
        name="fs-local"
        grouping="require_all"
        restart_on="none">
        <service_fmri value="svc:/system/filesystem/local" />
    </dependency>

    <!-- Wait for the listener to be started. -->

    <dependency type="service"
        name="oracle"
        grouping="optional_all"
        restart_on="none">
        <service_fmri value="svc:/site/application/database/listener" />
    </dependency>

    <!-- Define the methods. -->

    <method_context project=":default" working_directory=":default">
        <method_credential user="oracle" group="dba" privileges=":default" />
        <method_environment>
            <envvar name="ORACLE_HOME" value="/opt/oracle/product/home" />
            <envvar name="ORACLE_SID" value="oracle" />
        </method_environment>
    </method_context>

    <exec_method type="method"
        name="start"
        exec="/lib/svc/method/oracle start"
        timeout_seconds="120"/>

    <exec_method type="method"
        name="stop"
        exec="/lib/svc/method/oracle stop"
        timeout_seconds="120" />

    <!--
         What authorization is needed to allow the framework
         general/enabled property to be changed when performing the
         action (enable, disable, etc) on the service.
    -->

    <property_group name="general" type="framework">
        <propval type="astring"
            name="action_authorization"
            value="solaris.smf.manage.oracle" />
    </property_group>

    <stability value="Evolving" />
</service>
```

```
</service_bundle>
```

Add name and description metadata to the manifest so that users can get information about this service from the `svcs` and `svccfg describe` commands. See the `template` element in the DTD.

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

Start/Stop Method Script for the Instance Control Service

The following is the start/stop method script, `/lib/svc/method/oracle`, for the Oracle Database instance control service. This method calls the database `dbstart` and `dbshut` commands.

```
#!/bin/ksh -p

. /lib/svc/share/smf_include.sh

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ORACLE_HOME/lib
export PATH=$PATH:$ORACLE_HOME/bin

function startup
{
        dbstart $ORACLE_HOME
}

function shutdown
{
        dbshut $ORACLE_HOME
}

case $1 in
    start) startup ;;
    stop)  shutdown ;;

    *) echo "Usage: $0 { start | stop }" >&2
       smf_method_exit $SMF_EXIT_ERR_FATAL
       ;;
esac

smf_method_exit $SMF_EXIT_OK
```

Creating an Oracle Database Listener Service

This section describes the Oracle Database listener service manifest and the start/stop method script that is used in that manifest.

The listener is a process that manages the incoming traffic of client connection requests to the database instance. The database service depends on the listener service instance, and the dependencies of the database service should be online before the database service starts.

Listener Service Manifest

The following are some features to note about the Oracle Database listener service manifest, `/lib/svc/manifest/site/listener.xml`:

- One service instance is defined, named `svc:/site/application/database/listener:default`. This instance is enabled by default. See the `create_default_instance` element at the top of the manifest.
- The Oracle Database instance control service, `svc:/site/application/database/oracle`, depends on this listener service. However, the database service is still started if the listener service startup fails. See the `optional_all` dependency in "Instance Control Service Manifest" on page 54.
- The `method_environment` element in the `method_context` element defines the `ORACLE_HOME` and `ORACLE_SID` environment variables, which identify the database instance to start or stop. These values are then available for the method script to use.

  If you create multiple instances of this service, then each instance might need its own `method_context` element to define the unique `ORACLE_HOME` and `ORACLE_SID` values for that particular database.
- Attributes of the `method_context` element can define a resource pool in addition to the project and working directory shown in this example. You can also define either a `method_profile` or `method_credential` element in the `method_context` element. The `method_credential` element can specify `supp_groups` and `limit_privileges` values in addition to the `user`, `group`, and `privileges` values shown in this example. See the DTD for more information.
- The start/stop method script is `/lib/svc/method/listener`. The number of seconds before the method times out is increased from the default.
- A user must be assigned the `solaris.smf.manage.oracle` authorization to enable or disable this service instance.
- The service is `transient`. See "Service Models" in *Managing System Services in Oracle Solaris 11.3*.

```
<?xml version="1.0"?>

<!--
```

```
 Define a service to control the startup and shutdown of a database listener.
-->

<!DOCTYPE service_bundle SYSTEM "/usr/share/lib/xml/dtd/service_bundle.dtd.1">

<service_bundle type="manifest" name="listener">
<service name="site/application/database/listener" type="service" version="1">
    <create_default_instance enabled="true" />

    <!--
         Wait for all local file systems to be mounted.
         Wait for all network interfaces to be initialized.
    -->

    <dependency type="service"
        name="fs-local"
        grouping="require_all"
        restart_on="none">
        <service_fmri value="svc:/system/filesystem/local" />
    </dependency>

    <dependency type="service"
        name="network"
        grouping="require_all"
        restart_on="none">
        <service_fmri value="svc:/milestone/network:default" />
    </dependency>

    <!-- Define the methods. -->

    <method_context project=":default" working_directory=":default">
        <method_credential user="oracle" group="dba" privileges=":default" />
        <method_environment>
            <envvar name="ORACLE_HOME" value="/opt/oracle/product/home" />
            <envvar name="ORACLE_SID" value="oracle" />
        </method_environment>
    </method_context>

    <exec_method type="method"
        name="start"
        exec="/lib/svc/method/listener start"
        timeout_seconds="150"/>

    <exec_method type="method"
        name="stop"
        exec="/lib/svc/method/listener stop"
        timeout_seconds="30" />

    <!--
         What authorization is needed to allow the framework
         general/enabled property to be changed when performing the
```

```
         action (enable, disable, etc) on the service.
    -->

    <property_group name="general" type="framework">
        <propval type="astring"
            name="action_authorization"
            value="solaris.smf.manage.oracle" />
    </property_group>

    <!-- Make the instance transient (since it backgrounds itself). -->

    <property_group name="startd" type="framework">
        <propval name="duration" type="astring" value="transient" />
    </property_group>

    <stability value="Evolving" />
</service>
</service_bundle>
```

Add name and description metadata to the manifest so that users can get information about this service from the `svcs` and `svccfg describe` commands. See the `template` element in the DTD.

Start/Stop Method Script for the Listener Service

The following is the start/stop method script, `/lib/svc/method/listener`, for the Oracle Database instance listener service. This method starts or stop a listener process, `lsnrctl`. When `lsnrctl` starts, it tests the status of the database service.

```
#!/bin/ksh -p

. /lib/svc/share/smf_include.sh

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ORACLE_HOME/lib
export PATH=$PATH:$ORACLE_HOME/bin

function startup
{
        lsnrctl start

        # Wait for the listener to report ready.

        i=0
        while ! lsnrctl status | grep -i ready ; do
                ((i = i+1))
                if (( $i == 120 )) ; then
                        # It's been *at least* 2 minutes, time to give up.
                        echo "The listener failed to report ready." >&2
                        smf_method_exit $SMF_EXIT_ERR_FATAL
                fi
```

```
                sleep 1
        done

        # Ping the database once to prove it is now available.

        if ! tnsping $ORACLE_SID ; then
                smf_method_exit $SMF_EXIT_ERR_FATAL
        fi
}

function shutdown
{
        lsnrctl stop
}

case $1 in
    start) startup ;;
    stop)  shutdown ;;

    *) echo "Usage: $0 { start | stop }" >&2
       smf_method_exit $SMF_EXIT_ERR_FATAL
       ;;
esac

smf_method_exit $SMF_EXIT_OK
```

♦ ♦ ♦ CHAPTER 6

Using a Stencil to Create a Configuration File

If your application cannot use `libscf` library interfaces to read properties, you can use a stencil to create a configuration file. A *stencil service* creates configuration files by using a stencil file and property values defined in the stencil service. A *stencil file* contains a structural definition of a configuration file that is required by a service even though that service is now managed by SMF. Stencil services enable you to take advantage of SMF configuration management with no change to the existing application.

The stencil is used to generate a configuration file immediately before the service instance `start` method is executed. If you update the stenciled property values, restart the service to incorporate the changes into the configuration file before the application starts and reads the configuration file.

In Oracle Solaris, services for Puppet and Kerberos use stencils to provide configuration files.

This chapter describes:

- How to create a stencil service
- The Puppet stencil service in Oracle Solaris
- The Kerberos stencil service in Oracle Solaris

Creating a Stencil Service

A stencil file contains a structural definition of a configuration file that continues to be required by a service even though that service is now managed by SMF. The `svcio` utility generates the configuration file from the definitions in the stencil file and properties in the SMF service. See the `svcio(1)` man page for more information about the `svcio` utility and the `smf_stencil(4)` man page for information about stencil file format.

▼ How to Create a Stencil Service

1. **Create a stencil file.**

The stencil file tells the `svcio` utility the format to use to create the configuration file. The `svcio` utility converts SMF properties into application-specific configuration files based on a template called a stencil.

2. Add a `configfile` property group to the service.

The stencil service property group tells the `svcio` utility the path and ownership to use to create the configuration file. SMF regenerates configuration for all stencil-aware services before running the start or refresh methods. Property groups of type `configfile` tell SMF how to generate configuration files. Each `configfile` property group describes a single configuration file for the service and tells `svcio` how to generate these files from other properties stored in the SMF repository.

To configure a service to be stencil-aware, add a property group for each managed configuration file that contains the paths of both the stencil file to use as a template and the resulting configuration file. The property group has the following properties:

`path`	The path to which to write the configuration file, for example `/etc/svc.conf`.
`stencil`	The path of the stencil file to use, relative to `/lib/svc/stencils`. For example, if the value of the `stencil` property is `svc.stencil`, the `/lib/svc/stencils/svc.stencil` file will be used.
`mode`	The mode to use for the configuration file (`path`), for example 644.
`owner`	The owner to set for the configuration file (`path`). If this property is not set, the owner of the file is the user who invokes `svcio`.
`group`	The group to set for the configuration file (`path`). If this property is not set, the group will be the default group for `path`.

Puppet Stencil Service

Puppet is a toolkit for managing the configuration of many systems. On Oracle Solaris, the Puppet application is managed by SMF.

High Level View of Puppet Services

When you install the `system/management/puppet` package, you get two SMF service instances: `puppet:master` and `puppet:agent`. These instances are disabled by default.

After you enable these instances, the following command shows that both `puppet:master` and `puppet:agent` are contract services:

```
$ svcs -p puppet
STATE          STIME    FMRI
online         17:19:32 svc:/application/puppet:agent
               17:19:32     2565 puppet
online         17:19:32 svc:/application/puppet:master
               17:19:32     2567 puppet
```

The following command shows a little more information about the processes started by the contract services:

```
$ ps -o pid,args -p 2565,2567
  PID COMMAND
 2565 /usr/ruby/1.9/bin/ruby /usr/sbin/puppet agent --logdest /var/log/puppet/puppet-
 2567 /usr/ruby/1.9/bin/ruby /usr/sbin/puppet master --logdest /var/log/puppet/puppet
```

As suggested by the `ps` output, `puppet` is writing to log files in `/var/log/puppet`:

```
$ ls /var/log/puppet
puppet-agent.log  puppet-master.log
```

Initial Puppet Configuration File

Puppet expects to use a configuration file named `/etc/puppet/puppet.conf`. The `/usr/sbin/puppet` application reads configuration information from `/etc/puppet/puppet.conf` and not from properties set in the `application/puppet` service instances. To provide the required configuration file, each `puppet` instance provides a stencil file and `configfile` property group. The `configfile` property group tells the `svcio` utility to run and create the specified configuration file. The stencil file is used to write data from service property values to the configuration file in the correct format.

The following command shows all `puppet` service properties that are in a property group of type `configfile`. This output shows that both instances of the `puppet` service have the same `configfile` properties with the same values. Each `puppet` service instance provides the path to the configuration file, the mode of the configuration file, and the path to the stencil file.

```
$ svcprop -g configfile puppet
svc:/application/puppet:master/:properties/puppet_stencil/mode astring 0444
svc:/application/puppet:master/:properties/puppet_stencil/path astring /etc/puppet/
puppet.conf
svc:/application/puppet:master/:properties/puppet_stencil/stencil astring puppet.stencil
svc:/application/puppet:agent/:properties/puppet_stencil/mode astring 0444
svc:/application/puppet:agent/:properties/puppet_stencil/path astring /etc/puppet/
puppet.conf
svc:/application/puppet:agent/:properties/puppet_stencil/stencil astring puppet.stencil
```

The following commands confirm that these instance properties are inherited from the parent service.

```
$ svccfg -s puppet listprop -l all puppet_stencil
puppet_stencil          configfile  manifest
puppet_stencil/mode    astring     manifest              0444
puppet_stencil/path    astring     manifest              /etc/puppet/puppet.conf
puppet_stencil/stencil astring     manifest              puppet.stencil
$ svccfg -s puppet:agent listprop -l all puppet_stencil
$ svccfg -s puppet:master listprop -l all puppet_stencil
```

For your infrastructure, you might need `puppet:agent1` and `puppet:agent2` instances, for example. In that case, you would customize property values and add properties for each instance as shown in "Modifying the Puppet Configuration File" on page 65.

The following is the initial content of the configuration file, `/etc/puppet/puppet.conf`:

```
# WARNING: THIS FILE GENERATED FROM SMF DATA.
#     DO NOT EDIT THIS FILE.  EDITS WILL BE LOST.
#
# See puppet.conf(5) and http://docs.puppetlabs.com/guides/configuring.html
# for details.
```

Puppet Stencil File

The content of the stencil file tells you what properties and other information are written to the configuration file. The `puppet.stencil` path that is the value of the `puppet_stencil/stencil` property is relative to `/lib/svc/stencils`. The following is the content of the stencil file, `/lib/svc/stencils/puppet.stencil`:

```
# WARNING: THIS FILE GENERATED FROM SMF DATA.
#     DO NOT EDIT THIS FILE.  EDITS WILL BE LOST.
#
# See puppet.conf(5) and http://docs.puppetlabs.com/guides/configuring.html
# for details.
; walk each instance and extract all properties from the config PG
$%/(svc:/$%s:(.*)/:properties)/ {
$%{$%1/general/enabled:?
[$%2]
$%/$%1/config/(.*)/ {
$%3 = $%{$%1/config/$%3} }
}
}
```

In the stencil file, `svc:/$%s:(.*)/:properties` (or `%1`) expands to `svc:/application/puppet:agent/:properties` and `svc:/application/puppet:master/:properties`, where `.*` (or `%2`) matches every instance. The instance name is then used to label the block in the configuration file. The next occurrence of `.*` (or `%3`) matches every property in the `config`

property group for the `%1` service instance. The stencil tells `svcio` to write the property name and the value of that property from the service instance to the configuration file.

Modifying the Puppet Configuration File

As you can see in "Initial Puppet Configuration File" on page 63, initially only the literal comment lines are written to the configuration file. Writing property values to the configuration file is prevented by the test of the value of the `general/enabled` property in the stencil file. The following command shows that by default, the value of the `general/enabled` property is false:

```
$ svcprop -p general/enabled puppet
svc:/application/puppet:master/:properties/general/enabled boolean false
svc:/application/puppet:agent/:properties/general/enabled boolean false
```

Using the `svcadm enable` command to enable an instance does not change the value of the `general/enabled` property. When you change the value of the `general/enabled` property to `true` and restart the instance, all the properties in the `config` property group for that instance are written to the configuration file.

```
$ svccfg -s puppet:agent setprop general/enabled=true
$ svcprop -p general/enabled puppet:agent
false
$ svcadm refresh puppet:agent
$ svcprop -p general/enabled puppet:agent
true
$ svcadm restart puppet:agent
```

The following command shows that initially the only property in the `config` property group is the path to the log file for each instance:

```
$ svcprop -p config puppet
svc:/application/puppet:master/:properties/config/logdest astring /var/log/puppet/
puppet-master.log
svc:/application/puppet:agent/:properties/config/logdest astring /var/log/puppet/puppet-
agent.log
```

The `config` property for the enabled instance has been added to the configuration file in a block labeled with the instance name:

```
# WARNING: THIS FILE GENERATED FROM SMF DATA.
#     DO NOT EDIT THIS FILE.  EDITS WILL BE LOST.
#
# See puppet.conf(5) and http://docs.puppetlabs.com/guides/configuring.html
# for details.

[agent]

logdest = /var/log/puppet/puppet-agent.log
```

The Puppet configuration documentation says that the configuration file can have `[main]`, `[agent]`, and `[master]` blocks. Configuration in the `[main]` block applies to both the agent and the master. For the Puppet agent, configuration in the `[agent]` block overrides the same configuration in the `[main]` block. For the Puppet master, configuration in the `[master]` block overrides the same configuration in the `[main]` block. If you want to provide a `[main]` block for configuration that is common to both the agent and master, create a `puppet:main` instance and appropriate `config` properties for that instance.

The following commands show how to add configuration to your Puppet configuration file.

```
$ svccfg -s puppet:agent
svc:/application/puppet:agent> setprop config/report=true
svc:/application/puppet:agent> setprop config/pluginsync=true
svc:/application/puppet:agent> refresh
svc:/application/puppet:agent> exit
$ svcadm restart puppet:agent
$ cat /etc/puppet/puppet.conf
# WARNING: THIS FILE GENERATED FROM SMF DATA.
#     DO NOT EDIT THIS FILE.  EDITS WILL BE LOST.
#
# See puppet.conf(5) and http://docs.puppetlabs.com/guides/configuring.html
# for details.

[agent]

logdest = /var/log/puppet/puppet-agent.log
pluginsync = true
report = true
```

Similar commands can be used to remove properties and change property values. See Chapter 4, “Configuring Services” in *Managing System Services in Oracle Solaris 11.3*. To add a `main` instance, use the `svccfg add` command as shown in “Adding Service Instances” in *Managing System Services in Oracle Solaris 11.3*.

Kerberos Stencil Service

Another example of an Oracle Solaris service that uses a stencil is Kerberos. The following command shows that the `configfile` property group is `krb5_conf`, the stencil file is `/lib/svc/stencils/krb5.conf.stencil`, and the configuration file is `/etc/krb5/krb5.conf`.

```
$ svcprop -g configfile svc:/system/kerberos/install:default
krb5_conf/disabled boolean true
krb5_conf/group astring sys
krb5_conf/mode integer 644
krb5_conf/owner astring root
krb5_conf/path astring /etc/krb5/krb5.conf
```

```
krb5_conf/stencil astring krb5.conf.stencil
```

These values are set in the service manifest as shown below. They could be changed by using `svccfg setprop`.

```
<property_group type="configfile" name="krb5_conf">
        <propval name="disabled" type="boolean" value="true" />
        <propval name="path" type="astring"
                value="/etc/krb5/krb5.conf" />
        <propval name="stencil" type="astring"
                value="krb5.conf.stencil" />
        <propval name="mode" type="integer" value="0644" />
        <propval name="owner" type="astring" value="root" />
        <propval name="group" type="astring" value="sys" />
</property_group>
```

You can view the stencil file in `/lib/svc/stencils/krb5.conf.stencil` and the resultant configuration file in `/etc/krb5/krb5.conf`.

Index

R

S