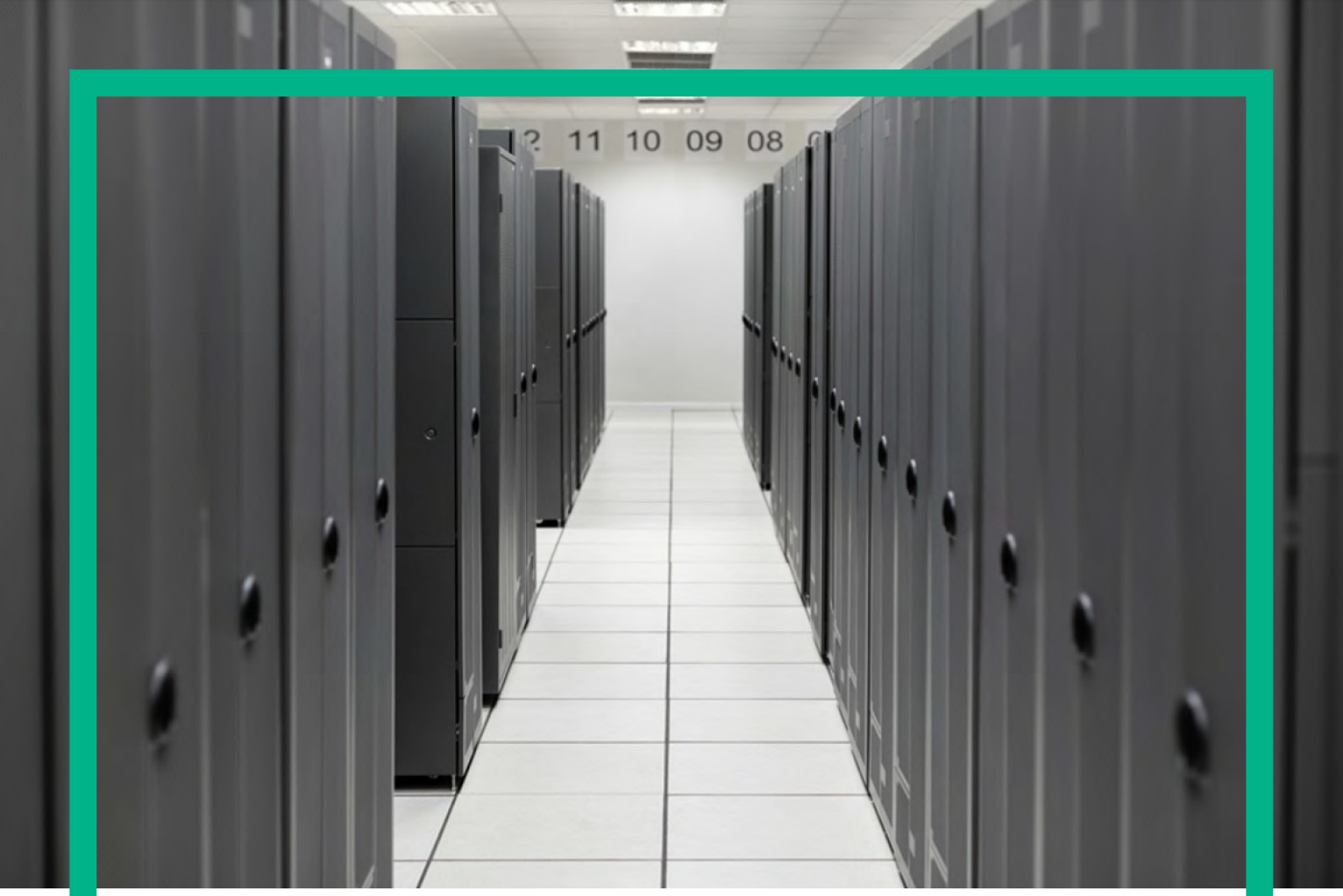

HPE Service Provisioner

for Red Hat Enterprise Linux 6.7 operating systems

Release: 8.0

March 15, 2016

Hewlett Packard Enterprise

Notices

Legal notice

Printed in the US

Trademarks

Document id: p158-pd027602

Contents

List of tables

No table of figures entries found.

List of figures

Conventions

The following typographical conventions are used in this guide.

Font	What the Font Represents	Example
Italic	Book or manual titles, and manpage names	Refer to the *HPE Service Activator—Workflows and the Workflow Manager* and the *Javadocs* for more information
	Provides emphasis	You *must* follow these steps.
`Computer`	Text and items on the computer screen	The system replies: `Press Enter`
	Command names	Use the `InventoryBuilder` command
	Method names	The `get_all_replies()` method does the following…
	File and directory names	Edit the file `$ACTIVATOR_ETC/config/mwfm.xml`
`Computer Bold`	Text that you must type	At the prompt, type: **`ls –l`**

In This Guide

This is the user manual of HPE Service Provisioner (HPE SP) documenting its architecture, customization, and integration.

Audience

This document is intended for project members who are responsible for the customization and integration of HPE Service Provisioner.

Chapter 1
Introduction

1.1 Overview

HPE Service Provisioner (HPE SP) provides an automated solution to provision the most complex service order requests and manage service life-cycle. It is integrated with the rest of the HPE OSS Fulfillment portfolio, that is,HPE Service Activator and HPE Trueview, but can also interoperate with external activation and resource inventory systems.

The key innovation brought by HPE SP 8.0 is the introduction of a new declarative language called Dynamic Service Descriptors (DSD). It allows complex provisioning and activation processes to be described declaratively. This means that HPE SP eliminates the complexity of the traditional template-based approach for provisioning.

Dynamic Service Descriptors allow:

a. To define services in a catalog as a combination of containment (parent-child), links (relationship), and inheritance (multiple inheritance)
b. To orchestrate service instantiation at run time based on policies
c. To automatically synchronize and populate the service inventory of HPE SP based on the result of service instantiation
d. To provide all necessary information to control the visualization of the catalog information in the HPE SP UI or an external UI

In addition, HPE SP 8.0 offers the following features:

- Moving from SOAP API to REST API, which is the most common language for APIs
- Providing API access to the complete lifecycle of catalog objects, service inventory items, and order requests
- Executing order requests in synchronous or asynchronous mode

1.2 HPE SP architecture and user roles

The following figure shows a high-level view of the HPE SP architecture.

Figure 1–1 HPE SP architecture and user roles

The different roles shown in figure HPE SP architecture and user roles are important to understand the functionality of HPE SP.

- Service User: Directly or indirectly invokes and uses HPE SP to create, modify, or delete service instances. One or more persons or systems can act as the Service User.
- Service Designer: Authors the Service Descriptor Specifications that drive the service decomposition and parameter evaluation in HPE SP. Service Designers define the abstract models that represent meaningful system configurations and topologies, independently of concrete infrastructure and infrastructure designs. One or more persons can act as the Service Designer.
- Activation Designer: Designs (and often also creates) the activation interfaces towards the virtual and physical infrastructure where the active services are configured. Activation Designers work with subject-matter experts and physical-logical infrastructure designers to identify and implement working infrastructure configuration actions that make the infrastructure deliver the desired service functionality. One or more persons can act as the Activation Designer.

In other words, Activation Designers create working entity-action building blocks. Service Designers specify meaningful combinations of the building blocks.

1.3 NCC components used in HPE SP

HPE SP uses HPE Service Activator solution components that are tried and tested in live Customer deployments. These components are managed by HPE as a framework called “Network Cloud Controller” (NCC). HPE SP uses the following components from NCC:

- Advanced Template Manager: The module handling abstract interactions with infrastructure shown in figureHPE SP architecture and user roles.
- Common Internationalized Error Handler: The module handing errors and internationalization in HPE SP.
- Extended Common Network Resource Model: The module handling systems and resources in alignment with the HPE SP Common Network Resource Model, as defined and supported by the HPE SP product.
- IPAM: An IP (v4 and v6) address and VLAN resource planning and allocation sub-system that can be used in HPE SP models for automating assignment of IP addresses and VLAN IDs according to pre-defined plans.

Chapter 2 Quick Installation Guide

Note

This is a quick installation guide for single node development environments. Please refer to the HPE Service Activator installation guide for detailed installation instructions for production environments and clusters. AllHPE Service Activator documentation is on the distribution `iso` file in the `Documentation` folder

2.1 Prerequisites

2.1.1.1 Operating System

HPE Service Provisioner is supported on the Red Hat Enterprise Linux 6 operating system. The installation of Red Hat Enterprise Linux 6 is beyond the scope of this quick installation guide.

Kornshell is required. To install:

```
yum install ksh
```

2.1.1.2 Java

HPE Service Provisioner requires Java SE Development Kit 8u73 or later, 64-bit. Installation steps:

- Download Java 8 SE (the RPM package) from Oracle Java
- Install the rpm: `rpm -ihv jdk-8u73-linux-x64.rpm`
- Set the `JAVA_`HOME environment variable so that it points to Java's location. This is required by the HPE Service Provisioner installation. Example (make sure this fits the installed Java version):

```
export JAVA_HOME=/usr/java/jdk1.8.0_73
```

2.1.1.3 Database

The following databases are supported:

- Oracle 12c
- Postgres Plus Advanced Server 9.4 (PPAS)

The installation and configuration of these databases is beyond the scope of this quick installation guide. This guide explains how to install Oracle XE on Red Hat Enterprise Linux 6 development environments:

1. Download Oracle XE from Oracle Database
2. Unzip and install the rpm:

```
unzip oracle-xe-11.2.0-1.0.x86_64.rpm.zip
rpm -ihv Disk1/oracle-xe-11.2.0-1.0.x86_64.rpm
```

Run the Oracle XE configuration tool interactively with /etc/init.d/oracle-xe configure or using a response file for silent installations as follows:

Create the response file called e.g. oracle_xe.rsp:

```
ORACLE_HTTP_PORT=7070
ORACLE_LISTENER_PORT=1521
ORACLE_PASSWORD=<SYS_PASSWORD>
ORACLE_CONFIRM_PASSWORD=<SYS_PASSWORD>
ORACLE_DBENABLE=y
```

Important

You need to replace <SYS_PASSWORD> with the actual password

- Run the configuration tool providing the response file:

```
/etc/init.d/oracle-xe configure responseFile=oracle_xe.rsp
```

3. Load the Oracle environment to to be able to run `sqlplus` later:

```
. /u01/app/oracle/product/11.2.0/xe/bin/oracle_env.sh
```

4. Create the application user for HPE Service Provisioner
   - Create the SQL script e.g. called `create_user.sql`:

```
-- Create hpsa_role
create role hpsa_role;
grant create session to hpsa_role;
grant create table to hpsa_role;
grant create sequence to hpsa_role;
grant create trigger to hpsa_role;
grant create view to hpsa_role;

-- Create user and assign hpsa_role
create user <DB_SA_USERNAME> identified by <DB_SA_PASSWORD>
default tablespace USERS quota unlimited on USERS;
grant hpsa_role to <DB_SA_USERNAME>;
```

Important

You need to replace <DB_SA_USERNAME> and <DB_SA_PASSWORD> with the actual user name and password, respectively

   - Run the script:

```
sqlplus system/<SYS_PASSWORD>@create_user.sql
```

2.2 Installation of HPE Service Provisioner

Note

This is a quick installation guide for single node development environments. Please refer to the HPE Service Activator installation guide for detailed installation instructions for production environments and clusters. AllHPE Service Activator documentation is on the distribution `iso` file in the `Documentation` folder

HPE Service Provisioner is distributed as an `iso` file. Installation steps:

1. Mount the `iso` file, replace `<ServiceProvisioner.iso>` with the actual filename:

```
mkdir /mnt/iso
mount -t iso9660 -o loop <ServiceProvisioner.iso> /mnt/iso/
```

2. Run the installer. `install` supports a `-silent` argument for automated installations. This also installs temporary licenses. Please refer to the HPE Service Activator installation guide for the installation of permanent licenses:

```
/mnt/iso/Binaries/Unix/install
```

3. Run the configuration tool interactively with `/opt/OV/ServiceActivator/bin/ActivatorConfig` or using a response file for silent installations as follows:
    - Create the response file `activatorConfig.xml` in folder `/etc/opt/OV/ServiceActivator/config/`, this is based on the template `activatorConfig_template.xml` in the same folder which contains the documentation of each parameter:

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE ActivatorConfig SYSTEM "activatorConfig.dtd">
<ActivatorConfig>
   <Mode>
      <Param name="PARTIAL_CONFIGURATION" value="false" />
      <Param name="BACKUP_AND_REPLACE" value="true" />
      <Param name="PORT_CONFIGURATION" value="true" />
      <Param name="SSO_CONFIGURATION" value="false" />
      <Param name="DB_CONFIGURATION" value="true" />
      <Param name="JBOSS_MANAGEMENT_CONFIGURATION" value="true" />
      <Param name="VIRTUAL_IP_CONFIGURATION" value="false" />
      <Param name="SCRIPTS_CONFIGURATION" value="false" />
      <Param name="SSH_CONFIGURATION" value="false" />
   </Mode>
   <Port-Mapping>
      <Param name="MWFM_PORT" value="2000"/>
      <Param name="RM_PORT" value="9223"/>
      <Param name="DB_PORT" value="1521"/>
      <Param name="WEBSERVER_PORT" value="8080"/>
   </Port-Mapping>
   <JBossManagement-Mapping>
      <Param name="MANAGEMENT_REALM_USERNAME" value="sys"/>
      <Param name="MANAGEMENT_REALM_PASSWORD" value="123"/>
   </JBossManagement-Mapping>
   <DisasterRecovery-Mapping>
      <Param name="DISASTER_SITE" value="Primary"/>
      <Param name="DISASTER_SITE_NAME" value="PRIMARY"/>
   </DisasterRecovery-Mapping>
   <Db-Mapping>
      <Param name="DB_HOST" value="127.0.0.1"/>
      <Param name="DB_INSTANCE" value="XE"/>
      <Param name="DB_USER" value="<DB_SA_USERNAME>"/>
      <Param name="DB_PASSWORD" value="<DB_SA_PASSWORD>"/>
      <Param name="DB_CREATE" value="true"/>
      <Param name="DB_VENDOR" value="Oracle"/>
   </Db-Mapping>
   <SysUser-Mapping>
      <Param name="SYS_USER" value="sys"/>
      <Param name="SYS_PASSWORD" value="123"/>
   </SysUser-Mapping>
   <SecureShell-Mapping>
      <Param name="SSH_USERNAME" value="sa_adm"/>
      <Param name="SSH_IDENTITY" value=""/>
      <Param name="SSH_BIN_DIR" value=""/>
   </SecureShell-Mapping>
```

```
</ActivatorConfig>
```

Important

You need to replace the following parameters with their real values:

- DB_HOST
- DB_INSTANCE
- DB_USER
- DB_CREATE
- DB_VENDOR
- SYS_USER
- SYS_PASSWORD

- Run the configuration tool providing the response file:

```
/opt/OV/ServiceActivator/bin/ActivatorConfig \
-f /etc/opt/OV/ServiceActivator/config/activatorConfig.xml
```

2. Increase the `JVM_MAX_MEMORY` parameter to 2048M in `/opt/HP/jboss/bin/standalone.conf`
3. Enable modules in `/etc/opt/OV/ServiceActivator/config/mwfm.xml`

- Uncomment the existing `http_sender_module` be removing the `<!--` before and the `-->` after it
- Add a new module right below the `http_sender_module`:

```
<Module>
  <Name>DDE_config_module</Name>
  <Class-Name>com.hp.ov.activator.ci.dde.module.DDEConfigModule</Class-Name>
  <Param name="solutions" value="DDE"/>
</Module>
```

4. Create a dbAccess.cfg file:

```
username=<DB_SA_USERNAME>
password=<DB_SA_PASSWORD>
```

Caution

- This file should never be created in production environments
- You need to replace <DB_SA_USERNAME> and <DB_SA_PASSWORD> with the actual user name and password, respectively

5. Install the DDE component:

```
cd /opt/OV/ServiceActivator/SolutionPacks
./Setup_DDE_<DDE_VERSION>.sh
```

6. Umount the iso file:

```
umount /mnt/iso
```

Note

- Replace <DDE_VERSION> with the actual version in the filename

- This will deploy the `DDE` and `CRModel` components and update the database. For installations on subsequent nodes in a cluster call this with the `-nosql` parameter:

```
./Setup_DDE_<DDE_VERSION>.sh -nosql
```

2.3 Uninstallation of HPE Service Provisioner

Run the following commands as `root` user. *remove.serviceactivator* supports a *-silent* argument for automated uninstallations:

Caution

Be very careful when typing `rm -rf` commands

```
/opt/OV/ServiceActivator/bin/remove.serviceactivator
rm -rf /etc/opt/OV/ServiceActivator
rm -rf /var/opt/OV/ServiceActivator
rm -rf /opt/OV/
rm -rf /opt/HP/jboss
```

Chapter 3 Life-cycle model

3.1 Overview of the service life-cycle states

HPE Service Provisioner (HPE SP) is based on a modified MTOSI state-transition model.

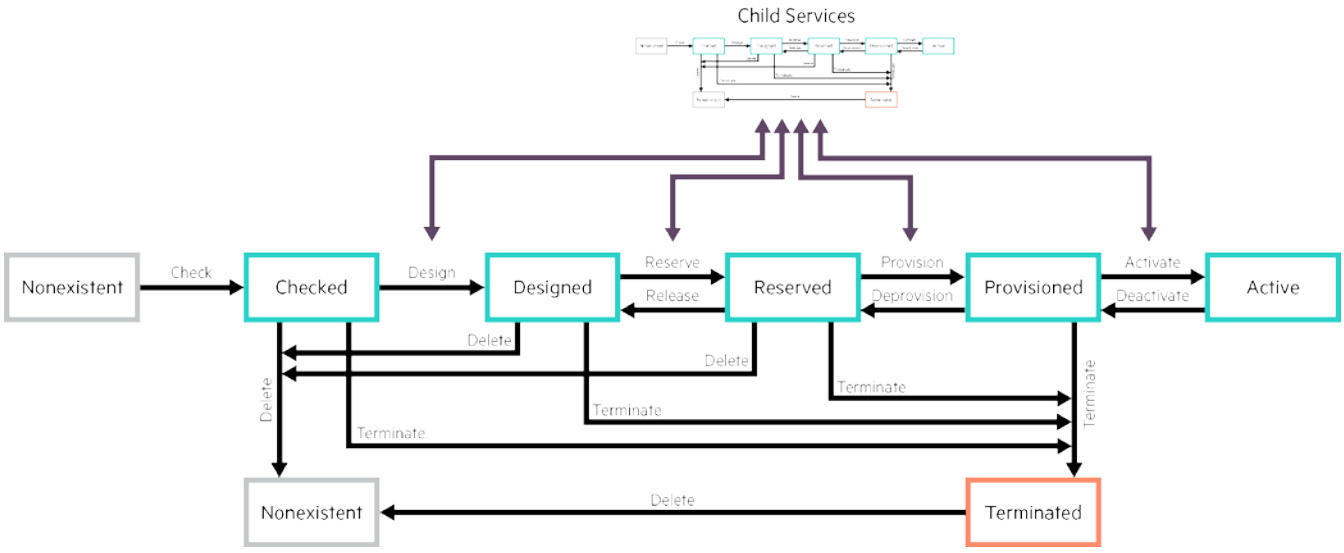

Figure 3–1 Service life-cycle state transition model

With HPE SP, there is full support for service modifications. When a request to modify a service is received, it is compared with the state of any current version of the service. If the service is already Provisioned or Active, then the modified instance is created as an uncommitted "Shadow" instance co-existing in parallel with theProvisioned or Active version. The modification is carried through the life-cycle steps until the Reserved state is reached, and the parameters of the service and any child-services are compared with the existing version.

For services or child-services that are found to be unaltered with respect to the parameters defined in the service descriptors, no changes are executed, and the shadow service is taken through Terminated to Nonexistent state, to release any resources.

3.2 Identification of service instances

There are multiple ways of handling service instance identification. The following fundamental identifiers are used:

- `tenantid`
- `serviceid`
- `servicename`
- `uuid`

3.2.1.1 The `tenantid` identifier

Tenants can be managed through the HPE SP GUI or through the REST API, which is aligned with the OpenStackKeystone API.

Every service instance belongs to one and only one Tenant, so the tenants form the roots of the service hierarchy.

A child service always belongs to the same Tenant as its parent, so a service-tree always belongs to a singleTenant.

A service may, however, reference services owned by other tenants to form a cross-Tenant service graph.

3.2.1.2 The `serviceid` identifier

The `serviceid` is the fundamental internal identifer that the system uses for every service instance. The service identifier is formed as a path from the ID of the Tenant and through a number of services. The service identifier has the following format:

```
<tenantid>/(/<servicename>(.<servicename>)*)+[#<commitstate>]
```

Example:

```
acmecorp//gotham.box[0].pNIC[0]/policy1
```

Every service instance belongs to one and only one Tenant, and is formed by separating the `servicename` identifiers of each parent service by a slash or a dot.

The <commitstate> can be either `shadow` or `committed`. While a service is being modified, an uncommitted version may co-exist with an already committed version of the same service instance. Such uncommitted services have the text `#shadow` appended to the `serviceid` in the HPE SP inventory in order to distinguish them from their committed versions.

The `#committed` form of the <commitstate> is never used internally or in the GUI. It is only supported on the NBI(the REST API) to allow explicitly requesting information about committed and uncommitted service instances.

The `serviceid` identifier can be used as a built-in parameter in service descriptors.

The `serviceid` identifier is not mutable.

3.2.1.3 The `servicename` identifier

The `servicename` is the unique identifier defined by the entity that requests a service through the NBI of HPE SP.

When creating a new service, the requesting system or Tenant usually has its own way of identifying the service. This external identifier is passed as the `servicename`. In case the request fails without a response, the requestor must be able to query HPE SP to verify if the service exists or not.

As each Tenant may have their own scheme for service identification, the `servicename` must be unique perTenant, but two different tenants are allowed to use the same `servicename` to identify different services.

So any requests that identify a service instance using the `servicename`, must also provide a `tenantid`. Unless a `#committed` suffix is appended to the `servicename`, such NBI requests return the latest version of the service. This means that if a shadow version exists, the shadow version is returned.

As explained in section The serviceid identifier above, the `serviceid` is composed of a number of slash-separated `servicename` instances. To ensure that every `serviceid` is unambiguous, a `servicename` must not contain the following characters:

- Whitespace

- `#`
- `/`

The `servicename` identifier is not mutable.

It is mandatory to provide a `servicename` in requests and on the HPE SP GUI, `unless` the Service Specificationexplicitly defines a policy for auto-creating the `servicename`. See section The uuid identifier for more details.

3.2.1.4 The `uuid` identifier

The `uuid` is the unique identifier defined by the entity that requests a service through the NBI of HPE SP.

NBI requests that identify a service using the `uuid`, return the latest version of the service instance unless a `#committed` or `#shadow` suffix is explicitly provided.

The `uuid` is `mutable`. The `uuid` can be provided as follows:

- It can be explicitly set through the NBI.
- It can be defined in service specifications.
- It can be uploaded to reflect a UUID of the service as represented in south-bound systems.

Note

In the overall system architecture, the effect of muting the `uuid` must be carefully considered to avoid problems if the `uuid` of a service does not remain constant.

The `uuid` is expected to follow the standard syntax of universally unique identifiers.

If no `uuid` is explicitly provided, one is automatically generated based on the `serviceid`. So if a service is created, deleted, and then re-created with the same `servicename`, then the `serviceid` and the `uuid` of the new service are identical to the `serviceid` and the `uuid` of the original service.

Example: `8f334328-e11d-3c3e-8e34-2ecb94472f0d`

Chapter 4 Service Descriptors

4.1 ETSI descriptor types

The ETSI NFV MANO specification ETSI GS NFV-MAN 001 V1.1.1 (2014-12) describes relevant information elements for the on-boarding and lifecycle management of virtualized network functions (VNFs) and Network Services.

For each top-level information element, the specification identifies a descriptor type:

- Network Service Descriptor
- VNF Forwarding Graph Descriptor
- VNF Descriptor
- Virtual Link (VL) Descriptor
- Physical Network Function (PNF) Descriptor

The Network Service Descriptor is used by the NFV Orchestrator to instantiate a Network Service, which can be formed by one or more VNF Forwarding Graphs, VNFs, PNFs, and VLs.

The Network Service describes the relationship between VNFs and possibly PNFs that it contains and the links needed to connect VNFs that are implemented in the NFV Infrastructure (NFVI) network. Links are also used to interconnect the VNFs to PNFs and endpoints.

4.2 Specification formats

Two candidate specification formats are presently suggested by ETSI: TOSCA and OpenStack HOT. Neither the ETSI specifications, nor TOSCA or HOT capture the complexities required for dynamic, policy-controlled network service topologies such as those required for virtual CPE services.

The strategic direction for HPE Service Provisioner (HPE SP) is to align to OpenStack. The HOT format is significantly simpler and easier to extend than TOSCA, so for HPE SP, the HOT format is extended.

4.3 Role of Service Descriptors

Service Descriptors can perform the following roles in HPE SP:

- Static validation of consistency at service on-boarding time
- Exposing the accepted request structure and parameters on the NBI of HPE SP
- Validating requests received on the NBI of HPE SP and returning relevant error messages
- Decomposing and mapping request parameters, resource pools, and locations in the infrastructure. With the decomposition, the set of relevant parameters is associated with each component
- Driving resource activation, which is the run-time use of the descriptor

Resource activation is performed as follows:

1. Reserving or creating resources in the selected resource pools using the HPE NFV Director Artifact-Relationship model (not described in this document)

Note

Reservation is not fully implemented yet (but see Function reserve).

2. Provisioning or configuring the reserved resources
3. Activating the resources

4.4 Service Specifications

In this document, the “Descriptor” and “Specification” terms are used as follows:

- Service Descriptor denotes the language-independent internal model that drives the decomposition of service requests.
- Service Specification denotes a textual representation in some language, from which Service Descriptors can be constructed by parsing.

HPE SP supports different Service Specification languages. The parsers create a common semantic representation of “Service Descriptors”.

The default language for the full expressive power of HPE SP is named `DSD`. The legacy names `NDDEDesc` and `NFVDesc` are also supported. Parsers for other languages can be written in Java and plugged into HPE SP.

The Service Specification texts can be loaded from files into the HPE SP inventory, where they are stored as `ServiceSpecification` objects, and parsed into the native Service Descriptor objects.

A `ServiceSpecification` object consists of the following fields:

Name:	The name of the service described.
Version:	The version of the service with this name. Instances must refer to the version of the specification that was used when the service was created.
CurrentVersion:	A Boolean value. It is `true` if this version is the one being instantiated by default. Only one version can be current for each given name.
Contents:	A textual representation of the descriptor authored through the HPE SP GUI or by loading the service specifications from files.
Language:	The specification language used for the contents (`DSD`, HOT, `TOSCA`, and so on). This field is used for selecting the appropriate parser for the contents.
SpecType:	Indicates the type of specification within the selected language, as several types can be associated with the same language (similar to `.c` and `.h` files for the C-language, or `class` and `interface` files for Java). The `DSD` language supports two SpecTypes: `Service` and `Kind`.

Chapter 5 DSD Specifications

5.1 Services and Service Kinds

A fundamental principle of DSD Specifications is the ability to achieve loose coupling between compound services and the service components they use. This is to achieve flexibility of versioning and ability to add multiple implementations of the same kind of service over time.

In order to achieve this, HPE Service Provisioner (HPE SP) introduces the concept of an interface Kind (somewhat corresponding to a TOSCA NodeType or a Java interface). A Service Descriptor can implement several such interface kinds, leaving it up to the instantiation to select an implementation of that kind of functionality.

Hence, two formats are defined with corresponding internal descriptors:

- Kind Specification - identifies a semantic capability and the parameters required to configure it.
- Service Specification - identifies a functional component - simple or composite - that can be instantiated by a service request.

Kinds usually cannot be instantiated, but a Service Specification can indicate that a Service with the same definition as the Kind must be automatically created.

Note

Use `Implements` to implement a Kind. There are two variants: `implements:` and `traits:`. They support exactly the same syntax and behave exactly the same way except for the following difference:

- `implements:` : Referenced Kinds must exist in the system. If one of them is missing, it is an error.
- `traits:` : Referenced Kinds may exist in the system. If they exist, they are used, otherwise, they are ignored.

The initial Service Specification language is derived from the OpenStack Heat Orchestration Template (HOT) format, using YAML syntax.

DDE uses the SnakeYAML parser, which is provided with the Jboss platform.

The syntax descriptions in the following sections must be understood in the context of the YAML definition. Particularly, the `->` arrow means a name-value map in YAML, and `*` means a YAML list. These constructs can be expressed in different ways in YAML syntax, hence the special notation. The order of appearance of the syntactic elements is not significant wherever the element is parsed as a YAML map.

Follow the golden rules of YAML:

- Indentation is significant.

- Always indent using spaces; never use tab characters.

Note

Make sure your editor is not configured to automatically indent using tab characters.

- In maps, never put spaces before the colon and always put at least one space after the colon (": ").

Also note, that parameter expressions that start with `$` or `@` must be quoted.

5.2 Specification Syntax

A Kind Specification describes a number of parameters that exist on all services implementing that Kind. A Kind may inherit from other kinds. The inheritance hierarchy is not allowed to be recursive.

```
Spec            ::=  DSD_(kind | service)_specification: LangVersion
                     SpecHeader Implements  Parameters ParameterGroups
                     Instantiation Children  Graphs  Outputs
LangVersion     ::=  V001
SpecHeader      ::=  SpecPackageSpecNameSpecVersion [ SpecDesc ]
                     [ SpecImpl ] [ SpecRoot ]  [ SpecAuto ]
                     [ NameAuto ] [ UpdateSpec ] [ SpecDfltSource ]
SpecPackage     ::=  package: PackageName
SpecName        ::=  name: BaseSpecName
SpecVersion     ::=  version: Version
SpecDesc        ::=  description: Text
SpecImpl        ::=  makeservice: ( true | false )
SpecRoot        ::=  root: ( true | false )
SpecAuto        ::=  autocreate: ( AutoName | AutoText )
NameAuto        ::=  autoname: ( AutoName | AutoText )
UpdateSpec      ::=  update: ( recreate | set | overlap )
SpecDfltSource  ::=  default_source: ( parent | request )
AutoName        ::=  name: AutoText
AutoText        ::=  ( Text | $AutoParam )*
AutoParam       ::=  ( zystem | linktype | referrer | parent | tenantid )
                     [ digit ]
Implements      ::=  (implements: | traits:) KindName -> version: Version
Parameters      ::=  parameters: ParamName -> ( Type | ParamSpec )
Type            ::=  KindName | string | integer | boolean | service |
object
ParamSpec       ::=  ParamAttr -> ParamValue
ParamExpr       ::=  Constant | ComplexExprSelectors
ComplexExpr     ::=  $ParamName | @FunctionName( [ Args ] ) | ListExpr
Args            ::=  ParamExpr ( , ParamExpr )*
ListExpr        ::=  "[" [ Args ] "]"
Selectors       ::=  ( FieldSelector | IndexSelector )*
FieldSelector   ::=  . FieldName
IndexSelector   ::=  "[" ParamExpr "]"
ParameterGroups ::=  parameter_groups: ParamGrp* ChildGrp
ParamGrp        ::=  [ label: Text ][ description: Text ]
                     [ parameters: ParamName* ]
```

```
ChildGrp        ::=  children: ChildName*
Instantiation   ::=  [ Instance ][ Entity ][ Actions ][ Workflows ]
Instance        ::=  instance: InstanceName
Entity          ::=  entity: EntityName
Actions         ::=  actions: ActionName -> ActionName
Workflows       ::=  workflows: ActionName [ -> WorkflowName | true | false
|
                     external | manual | WFDesc ]
WFDesc          ::=  [ WFDescName ] [ WFDescZystem ] [ WFDescPre ]
[ WFDescPost ] [ Verification ]
WFDescName      ::=  workflow: [ WorkflowName | true | false | external |
manual ]
WFDescZystem    ::=  zystem_param: ParamName
WFDescPre       ::=  pre: ( true | WFDescPrePost )
WFDescPost      ::=  post: ( true | WFDescPrePost )
WFDescPrePost   ::=  [ WFDescAction ] [ Entity ] [ WFDescName2 ]
[ WFDescZystem ]
WFDescName2     ::=  workflow: [ WorkflowName | true | false ]
WFDescAction    ::=  action: ActionName
Verification    ::=  [ Incompleteness ]
Incompleteness  ::=  incomplete: IncompleteSpec*
IncompleteSpec  ::=  [ IncompleteName ] IncompleteCond [ IncompleteMsg ]
IncompleteName  ::=  name: ConditionName
IncompleteCond  ::=  when: ParamExpr
IncompleteMsg   ::=  message: ParamExpr
Children        ::=  [ Resources ] [ Components ] [ Services ]
Resources       ::=  resources: ChildName -> ChildSpec
Components      ::=  components: ChildName -> ChildSpec
Services        ::=  services: ServiceName -> ChildSpec
ChildSpec       ::=  ChildControl [ parameters: Bindings ]
                     [ defaults: Bindings ]
ChildControl    ::=  ( ControlExpr | ControlAttr )*
ControlExpr     ::=  ControlName -> ParamExpr
ControlAttr     ::=  ControlAttr -> literal
Bindings        ::=  BindLVal -> ParamExpr
BindLVal        ::=  ( [ ChildName [ LValIndex ] . ] ParamName ) |
                     ( ParamName [ LValIndex ] )
LValIndex       ::=  "[" Number "]"
Graphs          ::=  graphs: GraphName -> GraphPart
GraphPart       ::=  PartName -> GraphPartSpec
GraphPartSpec   ::=  GraphCondition Bindings
GraphCondition  ::=  when: ParamExpr
Outputs         ::=  outputs: OutputDesc
OutputDesc      ::=  OutputIncludes OutputParams OutputTemplates
OutputIncludes  ::=  includes: [ parameters: true | false ]
                     [ affected: true | false ]
OutputParams    ::=  OutputName -> [ ParamExpr | OutputParam ]
```

```
OutputParam     ::=  value: ParamExpr
OutputTemplates ::=  templates: TemplateName -> OutputTemplate
OutputTemplate  ::=  OutputCondition  OutputDesc
OutputCondition ::=  when:  ParamExpr
```

Note

See section Parameter specifications for details about `ParamSpec : ParamAttr -> ParamValue`

5.2.1.1 Specification naming

A specification is identified uniquely by three name parts:

- PackageName: A name of a group of specifications
- BaseSpecName: The simple name of the specification within the package
- Version: The version of the specification of the form ###.##

5.2.1.2 Qualified names

When a specification refers to a KindName in the same package, the BaseSpecName can be used. If a specification needs to refer to a KindName in a different package, the name must be qualified with the PackageName. Qualified names follow the OpenStack standard of using "::" as package separator:

```
PackageName::BaseSpecName
```

Note

A PackageName is allowed to contain "::"separators as well.

Example:

A Kind with the `Box` BaseSpecName in a package named `vCPE` can be referenced as follows:

```
vCPE::Box
```

If referenced from within a specification that is also in the `vCPE` package, it is sufficient just to write:

```
Box
```

5.2.1.3 Versioning

When there are multiple versions of a specification, one of them is designated as current in the HPE SP inventory. By default, the highest number version is considered as "current", but this can be manually overridden using the inventory GUI.

If a specification references a Kind, it may indicate a specific version. If not, the "current" version is assumed.

A version number consist of:

- A three-digit major number
- A two-digit patch-number

For example, `002.04` is a valid version number.

DDE assumes that a change only in the patch-number indicates that any existing instances of the Service Specification does not need to be migrated (for example, if the change is "backwards compatible").

Always update the major number part of the version if instances of the specification are intended to be migrated at the next opportunity (that is, if a service change or an explicit migration process happens).

5.2.1.4 Kinds as services

The `makeservice` **`SpecImpl`** can only be used on Kind Specifications. If the `makeservice` **`SpecImpl`** is `true`, it means that the Kind auto-creates a service of the same name. For simple services, this saves the need to hand-craft a Service Specification whose contents are identical to those of the corresponding Kind Specification. The default value is `false`.

Presently, Graphs can only be specified in Services, not in Kinds.

5.2.1.5 Root services and child services

Some services are intended as child services and so require a parent service in order to function correctly. Other services are intended to be at the root of the service containment hierarchy. Some services may work equally well as root and as child services.

A service is a root service if it does not require a parent service to be defined.

The **`SpecRoot`** `root: true` specification indicates to the system that the service described is intended to work as a root service. Specifying `root: false` indicates that the service is not intended as a root service.

Leaving `root:` unspecified means that the system will automatically infer if the service can be a root service based on the parameter sources. If any parameters have `source: parent` (see section Auto-creation of service instances), then the service is assumed not to be a root service.

If a Kind Specification indicates `makeservice: true`, then it applies to any Services that implement that Kind. If `makeservice: true` was specified in the Kind, the implicitly generated Service can also be root.

In the HPE SP inventory GUI, there are actions to create services either as root or as a child. If services are created without a parent, the user can select the service-type in a drop-down menu listing only the root Service Specifications.

5.2.1.6 Auto-creation of service names

As described in section The servicename identifier, the `servicename` is one of the fundamental identifiers for all HPE SP services, components, and resources. When creating a new service instance, providing a `servicename` in the request (NBI or GUI) is mandatory unless the Service Specification has an auto-naming policy defined.

If both a `servicename` is provided in the request (NBI or GUI) and an auto-naming policy is defined, the `servicename` from the request is used and the auto-naming policy is not applied.

The **`NameAuto`** syntax is used for defining an auto-naming policy. The syntax is the same as for the `autocreate` feature (see section Auto-creation of service instances), but the set of supported auto-parameters is different.

Examples:

```
autoname: $parent.on_$zystem
autoname: $parent.for_$referrer2
```

The following special parameters are supported for auto-naming:

Note

Do not confuse these with ordinary descriptor parameters.

$parent:	The service name of the parent of the new service.
$parent2:	The service name of the grand-parent of the new created service.
$tenantid:	The tenant ID of the new service.
$transaction:	The transaction ID of the transaction creating the new service.
$time:	A timestamp number (13 digits within the foreseeable future).
$random:	A random number with up to 10 digits.
$randomN:	A random number with up to N digits, where N can be between 2 and 9.
$hash:	A 9 digit number generated from the transaction ID of the transaction creating the new service
$hashN:	As above, but with N digits. The number, N, can be between 2 and 9.
$uuid:	A UUID generated from the transaction ID of the transaction creating the new service.
$nbr_peer:	A sequence-number for all services ever created for the same parent. If there is no parent service, it works like `$nbr_tenant`. NOT YET SUPPORTED
$nbr_tenant:	A sequence-number for all services ever created for the tenant. NOT YET SUPPORTED

In case of failures, a northbound system may re-try the original request to HPE SP with the same `transactionid` parameter. To avoid this causing the creation of duplicate service instances, it is recommended to use either `$transaction` or `$uuid`, as this will produce the same `servicename` as in the original request, and so HPE SPwill use the existing instance instead of creating a new one.

Note

Do not confuse auto-naming with auto-creation. Auto-naming defines service names for externally requested services, whereas auto-creation defines policies for fully automating the internal process of creating and use-counting services from other services as defined in sections Auto-creation of service instances and Naming of auto-created service instances (auto-names).

5.2.1.7 Auto-creation of service instances

The HPE SP system supports auto-creation of service instances being referenced from other services. The mechanism and options for auto-creation are described in detail in section References between services.

The `SpecAuto` syntax is used for defining an auto-creation policy.

Details on how to use the `autocreate:` specification is covered in section Naming of auto-created service instances (auto-names).

Note

Do not confuse auto-creation with auto-naming. Auto-creation defines policies for fully automating theinternal process of creating and use-counting services from other services. The auto-naming feature (sectionAuto-creation of service names) uses the same syntax for defining service-names of externally requested services, and the set of supported auto-parameters is different.

5.2.1.8 Update policy

When a configured service needs to be modified, there are three possible policies. The appropriate one can be defined using the `UpdateSpec` syntax:

update: recreate:
: Tear down the existing version of the service, then create the updated version.

update: set:
: Re-execute the existing actions with the updated parameter values. The underlying systems must be able to set the altered parameter values to reflect the change, simply by overwriting the previous configuration.

update: overlap:
: Create the new version of the service before tearing down the old one. This implies that the old and the new version will co-exist for a period of time, and the systems must be of a nature that allows this. NOT YET SUPPORTED

The default policy is: `set`.

In two cases, `set` cannot work, and so a `recreate` will be executed even if `set` was specified (or if no update policy was specified). That happens if the following parameters are modified:

- zystem_name parameter: If this parameter (see section Special parameters) is changed, or if the indicated Zystem is re-created, then any services provisioned on the old Zystem, will be invalid, and the service must be re-created on the new Zystem.
- ServiceType: If the name or major-version of the ServiceType is modified, then the old service must be destroyed and the new one must be created. If only a patch-version is modified, then the service descriptor is assumed to be backwards compatible, so a patch alone will not trigger a re-creation (see also sectionControlling service comparison).

The update policy is not modifiable or parameterized.

The policy applies to all parameters in the current descriptor that are relevant for activation (see the `activate:` attribute on parameters in section Parameter specifications).

The update policy may be inherited from implemented Kind Specifications. In case there is no specific policy in the service and the implemented Kinds have different explicitly defined update policies, the resulting policy will be `recreate`.

5.2.1.9 Parameter specifications

Parameter values are stored on service objects in the inventory as “metadata” following the XML serialized syntax defined by OpenStack. A number of parameters are specifically defined as database fields.

“Metadata” is the data driving the way parameters are listed in the inventory GUI and in descriptors generated for the request-based APIs supported by HPE SP.

Supported `ParamAttr` parameter attributes and their corresponding `ParamValue` parameter values are defined as follows:

- `type:` `Type`: The type of value to be held by the parameter.
- `label:` Short text: Label for request metadata, for example, for the generated GUI and request API.
- `description:` Text: Description for request metadata, for example, for the generated GUI and request API.
- `optional: true | false`: If false, a null resulting value raises a run-time error.
- `default:` `ParamExpr`: Value may be overridden by the request or by parent bindings.
- `prefer:` `ParamExpr`: Use the value of the expression if not null - if null, use the value from the request.
- `value:` `ParamExpr`: Value is not allowed to be defined from request or parent bindings. For backwards compatibility, `constant:` is also allowed.
- `range:` `ParamExpr`: A list of allowed values for the parameter, see section Declaring a parameter range.
- `validate:` `ParamExpr`: If false, causes a run-time error message for the parameter.
- `hidden: true | false`: Used for passwords.
- `delta: true | false`: Indicates if this is a delta-parameter - see section Using delta parameters.
- `activate: true | false`: Setting `false` indicates that this parameter is not used in any of the activation actions as defined in section Actions and State-transition workflows. Default is `true`.
- `source: request | parent`: Tentatively indicates if the intended source of the parameter is the service-request or the parent service. Used for generating request metadata and presentation.
- `unwrap: true | false`: If `true`, and the value is a one-element list, remove the wrapping list and assign the element as the value of the parameter. The default is `true`.
- `display:` `ParamExpr`: Condition for including/excluding the parameter in display during create or update. Supported values: `true`, `false`, `create`, `update`.
- `displayclasses:` Class*: A list of names that can be used for determining one or more display classes for this parameter. Examples: `main`, `admin`, `auxiliary`. The `main` displayclass is automatically used for empty mandatory parameters.
- `group:` Text: A label of a parameter-group that this parameter must belong to if it is not explicitly listed as part of that parameter group. The parameters defining a group label are appended to the group in no specific order. The group must be previously defined in a `parameter_groups` specification.
- `readonly:` `ParamExpr`: NOT YET IMPLEMENTED Condition under which the parameter is listed as read-only in the metadata.

It is never allowed to use more than one of the following specifications:

- `default:`
- `prefer:`
- `value:`

A parameter which has neither `default:` nor `value:` attribute is automatically assumed to be mandatory (that is, not `optional:`). To make such a parameter optional anyway, specify `optional: true`.

See section Choosing between default, prefer and value parameter expressions for further guidelines on choosing between default, prefer and value.

A specification may re-define the default value of a parameter value inherited from one of the Kinds it implements, as long as the original parameter value is defined as a default. A specification may also redefine the following attributes:

- `label:`
- `description:`
- `range:`
- `validate:`
- `hidden:`
- `activate:`
- `source:`
- `display:`
- `readonly:`

Note that the `display:` attribute takes precedence over `readonly:`. A parameter where the `display:` attribute is `false`, will not be displayed even if the `readonly:` attribute is `true`.

Note

Some parameter names have a special meaning in the system. See section Special parameters for details.

The `location` and `locationname` parameters are propagated automatically from parent to child services. See section Child parameter bindings for details.

5.2.1.10 Declaring a parameter range

The range specification defines a list of allowed values for a parameter. HPE SP validates that any given parameter value satisfies the range, and the process fails in case of a violation.

The range can be given as a simple list `[0, 1, 2]` or as an expression involving other parameters or inventory lookups.

In the HPE SP GUI, the range is shown as a drop-down list for selecting the appropriate value.

If the parameter is declared as optional, then the null value is automatically added as the first allowed value for the range.

If no default value is explicitly declared, the first value of the range automatically becomes the default (null if the parameter is optional).

5.2.1.11 Using delta parameters

Where ordinary parameters always contain absolute values, “delta parameters” handle requests to modify a parameter with a given increment or decrement relative to its current value.

Example:

Consider the following example: A VM service has an integer parameter called `cores` that defines the number of cores of that VM. The number of cores can be set from the northbound interface. In case of proper implementation of actions, calling with `cores: 4` makes DDE ensure four cores on the VM as follows:

- If the current number is 1, DDE adds three cores.
- If the current number is already 4, DDE does nothing.

Some northbound systems are designed to send a "delta value" instead of an absolute value. So instead of sending `cores: 4`, such systems send: `inc_cores: +3`.

The delta parameters of DDE are designed to support these cases.

Parameters declared as `delta: true` are supported in the REST API but are never visible in the HPE SP GUI. A human user can always look at the current value of the parameter in the GUI and decide a new absolute value.

A number of special rules concerning delta parameters must be observed:

- Delta parameters can be provided from the northbound interface or be computed based on other delta parameters:
  - A delta-parameter to be supplied from the northbound interface must define a default value either by an explicit `default:` specification or by defining a range. Using `value:` is not allowed for such parameters as the expression would then overwrite the supplied value.
  - Conversely: A delta-parameter that depends on other delta-parameters must be defined using `value:`.
- The `default:` or `delta:` expression of a delta parameter is allowed to refer to other parameters.
- Non-delta parameters (and other policies such as `count`, `min`, `max`) are ordinarily not allowed to refer to delta-parameters in their expressions.
- The exception to the above is the definition that applies a delta to a non-delta parameter. A delta application is defined as follows:
  - Define a `value:` expression.
  - The expression must contain a reference to the parameter itself (normally this would cause an error).
  - The expression must refer to one or more delta parameters.
  - The default values of all deltas must be chosen carefully to be "idempotent" in the expression. It iscritically important that when the expression is evaluated with the default delta(s), then the value of the non-delta parameter does not change.

The delta value is applied to the absolute parameter value during the design phase of the DDE life-cycle model (see section Life-cycle model). Once the delta is applied, DDE resets the delta to its default value.

The number of times the expression that applies the deltas to a non-delta is evaluated cannot be predicted. If the expression changes value after the deltas are defaulted, the absolute value keeps increasing (or decreasing) in an unpredictable way. It is the responsibility of the Service Designer to ensure that this does not happen.

Example:

Consider the VM service from the previous example. The `cores` and `inc_cores` parameters can be defined as follows:

```
inc_cores:
  type: integer
  default: 0
```

```
  range: [-1,0,1]
  delta: true
  source: request
  description: Delta increment parameter for the number of cores

scale_cores:
  type: integer
  value: "@sum($scale_cores, $inc_cores)"
  source: request
  range: "@numbers(0,7)"
  description: Absolute number of cores to scale up from the default 1 core

cores:
  type: integer
  value: "@sum(1, $scale_cores)"
  description: Absolute number of cores, defaulting to 1
```

Important notes on the example:

- Applying the default value `0` of the `inc_cores` parameter to the `sum` -expression of `scale_cores` will leave the number of `scale_cores` unchanged.
- Both the `inc_cores` and the `scale_cores` parameters may be set through the northbound interface of HPE SP, so it is always possible to set the absolute scale-up of the VM as well as the delta scale.
- It is not possible to collapse `scale_cores` and `cores` into a single parameter by writing something like:

```
cores:
  type: integer
  value: "@sum($scale_cores, $inc_cores, 1)" # ERROR - NOT IDEMPOTENT
  source: request
  description: WILL NOT WORK - the number would keep incrementing
```

In the future, the syntax of HPE SP may be enhanced to avoid the need to introduce intermediate parameters like `scale_cores`.

5.2.1.12 Parameter Groups

The `parameter_groups` section defines a YAML list of parameter groups that control the order and grouping of parameters in the metadata for the northbound interface of HPE SP. Particularly, if the inventory GUI is used for creating or maintaining services, the `parameter_groups` can be used for formatting and grouping of the input and display parameters in the GUI forms.

Each group can define a label and description that characterize a list of parameters. Each of the listed parameter names must refer to a parameter, either defined in the Service Specification itself or in an included service Kind Specification.

Kind Specifications may also define parameter groups, which are inherited by any including Service or Kind. In case of conflicts, the groups in the including service take precedence. If multiple Kinds that are included have mutually conflicting parameter groups, the result is unpredictable. In this case, it is highly recommended to resolve the conflicts using an explicit list of parameter groups in the including Service Specification.

The first parameter group in the list is allowed to have no (or empty) label and description. All subsequent groups must define a non-empty label. In the GUI, labels appear as parameter separators, and the label-less group defines an initial list of parameters that are not preceded by any such separator. Parameters that are not mentioned in any parameter group are automatically added to this initial label-less group, even if that group is not explicitly defined in the specification.

A special parameter group is used for defining the order of appearance of input parameters for child components and resources. This should be the last in the list of parameter-groups (under the `parameter_groups` keyword) and is defined using the keyword `children:` and is followed by a YAML list of child-names. Child components that are not listed are added in random order after the listed ones.

If an included Kind Specification contains a child parameter group for its children, then those are added in order after the ones listed explicitly in the including specification. So the including specification can always override the child order defined in an included specification.

5.2.1.13 Special inventory beans

The InstanceName is the name of the specialized service class to create. Use it if the service instances must be represented in the HPE SP inventory with a native bean class that extends the generic `Service` bean. See HPE Service Activator product documentation on how to create such bean-class extensions.

5.2.1.14 Actions and State-transition workflows

This section of the Service Specification defines the interface to the HPE SP activation system and the Advanced Template Manager as shown in figure HPE SP architecture and user roles. The workflow and template side of the interface is described in detail in section Integration with workflows and templates.

5.2.1.15 Specifying the activation interface

The following parts of the interface can be specified:

workflow: The name of the HPE SP workflow to invoke. Defaults to `DDE_Activate_Service`.

entity: The name of the entity-type to operate on.

action: The action to carry out on the given entity type.

zystem_name: The management system that controls the entity.

In addition to the above:

Parameters: All instance-values for all parameters defined in the Service Specification are passed to the workflow as a meta-data map.

Services: The service instance object itself is passed to the Service Specification, making, for example, the parent service instance and referenced services accessible.

The `Entity` specification defines the EntityName that is passed to activation workflows. If not specified, theEntityName defaults to the BaseSpecName of the Service Specification.

The actions default to the possible STATE_transition pairs defined by figure Service life-cycle state transition model:

- `CHECKED_design`
- `CHECKED_delete`
- `DESIGNED_reserve`
- `DESIGNED_delete`

- `RESERVED_provision` (Activation action)
- `RESERVED_release`
- `RESERVED_delete`
- `PROVISIONED_activate` (Activation action)
- `PROVISIONED_deprovision`
- `PROVISIONED_terminate`
- `ACTIVE_deactivate`
- `TERMINATED_delete`

Additional actions can be defined, but are currently not supported in the northbound interface of HPE SP.

The actual action-names supported by the Advanced Template Manager interface will usually be more descriptive than the plain state-transition names. Therefore, the `Actions` section of the Service Specification allows mapping of the default names to the actually supported action names. For example, `PROVISIONED_activate` on a VM is typically mapped to a `powerup` action.

The Zystem parameter passed to the workflow will default to the value of the parameter named `zystem_name` (see the complete list of parameters with special meaning in section Special parameters). In special cases, different actions may want to operate on different zystems. For that purpose it is possible to specify the name of a different parameter by specifying the `zystem_param`.

Note that the Advanced Template Manager takes the given Zystem as the starting point for locating a suitable management system that supports the specified action on the specified entity. This is based on Zystem-containment and on searching the location hierarchy. It is the task of the Activation Designer to model the correct actions, and to ensure that the HPE SP inventory is populated with adequate systems and locations for this process to work as intended.

Specifying `false` as the workflow name can be used to explicitly indicate that there should be no workflow call. This is particularly useful for overriding the use of an inherited workflow and for indicating that no main workflow exists.

5.2.1.16 Controlling service comparison

Before executing the `RESERVED_provision` or `PROVISIONED_activate` actions (marked as “Activation action” in section Specifying the activation interface), DDE compares the previously activated version of the service (if present) with the new one.

If the two service instances are identical, it means that the active service is already correctly activated, and DDE does not need to tear terminate and re-create it. Instead, DDE leaves the infrastructure unchanged, and just records the new version of the service instance in the service inventory.

Service instances are compared by comparing:

- Service Specification name and version: This allows the system to auto-migrate a service to a new major version of the Service Specification.

> **Note**
>
> Changes in the two minor digits of the version (the patch-number as defined in section Versioning) indicate a backwards compatible change, and hence do not cause existing service instances to be migrated.

- Parameters: Parameter values are compared to identify any changes.

When comparing parameters, it is important to note that in a Service Specification some parameters are merely intermediate values that do not actually impact the Activation actions. The Service Designer may indicate to the DDE comparison engine that a parameter does not influence Activation actions by adding `activate: false` to the `ParamAttr` parameter specification (see section Auto-creation of service instances).

Note, that if the Activation Designer erroneously lets the activation workflows and templates refer to a parameter that was marked by the designer as `activate: false`, then the resulting system does not function correctly. Also note, that all parameters are considered intermediate (administrative) if the service has defined no provisioning or activation actions at all.

So it is important that the Activation Designer understands the meaning of `activate: false` as part of the interface contract with the Service Designer.

5.2.1.17 Pre-processing actions

Two special actions are defined:

- `PreprocessPROVISIONED` (Activation action)
- `PreprocessACTIVE` (Activation action)

If any of these are enabled in the specification, the associated action and workflow are executed once, when the indicated state is reached.

For example, if `PreprocessACTIVE` is mapped to action `configureLicence`, then the first time the service reaches the `ACTIVE` state, the action workflow is executed, but any subsequent transitions between `ACTIVE`, `PROVISIONED`, and back to `ACTIVE` will not execute the workflow again.

If you want to write a custom `RESERVED_provision` or `PROVISIONED_activate` workflow that tests for the same condition, you can use the `service.provisionedcount` or `service.activationcount` attributes, and check if they are equal to 0.

5.2.1.18 Pre-workflow and Post-workflow specifications

The Activation Designer can introduce special actions to verify or ensure that the infrastructure is in a correct state before the actual configuration actions are executed. Such actions can be specified as pre-workflow actions.

Similarly, the Activation Designer can introduce special actions to verify or ensure that, for example, asynchronous actions are completed before the state-transition is complete. Such actions can be specified as post-workflow actions.

Furthermore:

- The pre-action may decide that the service is already configured, so that the activation and/or post-workflow may not need to be executed at all.
- Whether a post-workflow is needed at all may depend on the templates/workflows selected for the Zystem at hand. So the activation workflow may need to specify if the post-workflow action is actually required or not.

For details on how to define the pre-workflows and post-workflows themselves, see section Integration with workflows and templates.

The Service Designer can specify that a pre-workflow or post-workflow is needed by simply adding a `WFDescPre` or `WFDescPost` section saying either:

```
pre: true
or:
```

```
post: true
```

respectively. This will result in the default activation workflow `DDE_Activate_Service` being called with the same entity and Zystem as the activation workflow, and with `pre_` or `post_` prepended to the action. So if the activation action is `powerup`, then the default post-action will be `post_powerup`.

Specifying `false` can be used to explicitly indicate that there should be no workflow call. This is particularly useful for overriding the use of an inherited workflow.

If these defaults are not sufficient for the Activation Designer, then alternate values can be specified for the pre and post workflows, for example:

```
pre:
  workflow: WF_Sleep
  entity: event
  action: wait
  zystem_param: my_watch
```

All of the above settings are optional and will default to the same value as described for `post: true`.

Notice that after `zystem_param:` you must write a literal parameter name (syntactically a ParamName) and not an expression (`ParamExpr`). So in the example above, the specification must explicitly define a parameter with the name `my_watch`.

5.2.1.19 Long-running external and manual processes

Sometimes a state transition needs to be executed by an external system or it may require manual user input, design, or approval. For these use-cases, the main workflow can be specified as respectively `external` or `manual` using the `WFDescName` syntax.

For both types of interaction, the service stops with its desired state set to the indicated state ID as defined in figure Service life-cycle state transition model. The state of the service is progressed only when the desired state is explicitly modified through the northbound interface (that is, through the REST API or the GUI). Parent workflows and sibling workflows that depend on the service also wait in their corresponding states until the external progress request is received.

The `external` and `manual` specifications replace the main action of the given state-transition. If a pre-workflow and/or post-workflow are specified (see section Pre-workflow and Post-workflow specifications), the pre-workflow is executed before waiting for the external response, and the post-workflow is executed once an external action sets the `desiredstate` of the service to a state different from the current `state`.

A common usage pattern for an `external` asynchronous interaction can be summarized as follows:

1. Define a pre-workflow action to send the external request to the remote system.
2. Use the `external` specification to wait for the response.
3. Use a post-workflow action to validate the response.

If the interaction is `manual`, then you can also use a post-workflow to validate the input.

Completeness of the long-running task is decided using the `Incompleteness` specification. See sectionCompleteness of long-running processes.

If an `external` interaction is somehow delayed or failed, a user can still perform the required work as if it were manual.

Example use of `external`:

```
pre:
  entity: order
  action: elaborate
  zystem_param: some_external_system
workflow: external
post:
  workflow: WF_ValidateOrder
  entity: order
  action: validate
```

5.2.1.20 Completeness of long-running processes

By default, a long transaction is considered complete when all mandatory parameters are defined.

The **`Incompleteness`** specification allows you to define additional criteria for when a service should be considered incomplete for a given transaction. A number of named incompleteness conditions can be defined. Naming is optional but recommended to guide the merging of incompleteness criteria when descriptor inheritance is used.

Example use of `incomplete`:

```
workflows:
  CHECKED_design:
    workflow: manual
    incomplete:
      - name: approval
        when: "@equals($Approval, Pending)"
        message: "@cat(Please approve or reject , $serviceid)"
```

If the post-workflow decides that the response was not complete it can indicate that DDE must keep waiting for an additional external request. To do this, the workflow must use the **`uploaded_data`** map (see section Workflow parameters). The validation workflow must set the key named **`desiredstate`** to the name of the current state using

the `uploaded_data` map (see section Workflow parameters). This indicates to DDE that the received data was incomplete and that it must keep waiting for another request.

5.2.1.21 Child specifications

A Service Specification can define a number of named children.

5.2.1.22 Child types

There are three kinds of children:

- `components`: These children are internal "resource-facing" sub-services that are managed exclusively by the service logic. The Service Kind and the number of instances of each component to be created is controlled by the logic defined in the **`ChildControl`** section of the specification. The state of a component is advanced through the life-cycle states in step with the parent service.
- `services`: These children are "customer-facing" sub-services that must be explicitly requested. Child services are not managed by their parent, but are explicitly managed by other mechanisms, for example, from the northbound API of HPE SP. Service children may also be auto-created as a result of the references mechanism described in section References between services.
- `resources`: These children are brought to the `ACTIVE` state immediately after being designed (see the state diagram in Service life-cycle state transition model), that is, before any other child-components are designed. This means that the active resources (in the infrastructure) are accessible during the design of the other child components. So if a resource is a DHCP server, then it may be used to create IP addresses for passing as parameters to subsequently created `components` and `services`. The Service Kind and the number of instances of each resource to be created is controlled by the logic defined in the **`ChildControl`** section of the specification.

The number of instances of each resource and component is controlled by the `count` control-name in the **`ChildControl`** section. The value of the count can be a parameter expression, meaning that any function can be called and other parameters can be used in the computation of the scale-out of the service.

Note that descriptor functions can be used for defining policies for the count, and hence for the scale-out decisions, or the count can be provided in modification requests from external systems, allowing external triggers of scaling.

Consequently, resources and components are identified by an index value, starting from `0`, for example, `nic[0]`. Services, on the other hand, are given a unique name (per Tenant) in the explicit service-request, for example, `router[vsr-site-0ae3]`.

5.2.1.23 Child control attributes

The instantiation of a child is controlled by the following set of special **`ControlAttr`** attributes:

- `kind:` KindName: Name of the specification kind of this child. Use a qualified name to refer to a Kind from a different package.
- `version:` Version: Version of the specification kind - latest version if omitted.
- `type:` **`ParamExpr`**: Only allowed for `resources` and `components`. See explanation below.
- `min:` **`ParamExpr`**: Minimum number of `component` or `resource` instances. The minimum number is validated and is used as the default for the count attribute if the count is not specified. Defining `min` is not allowed for services, because they are explicitly requested, and so their minimum number must be 0.
- `max:` **`ParamExpr`**: Maximum number of instances allowed. The system validates that the `max` number cannot be exceeded.

- `count:` **`ParamExpr`**: Actual number of instances to auto-create of a `resource` or `component`.
- `when:` **`ParamExpr`**: Shorthand for writing: `count: @if(` **`ParamExpr>`** `,1,0)`, cannot be combined with the `count`, `min`, or `max` attributes.
- `preprovisioned:` **`ParamExpr`**: If the expression evaluates to `true`, all activations are called in simulation mode, but the service is not marked as `WasSimulated`, so child services are activated as normal.
- `optional: true | false`: A value of `true` is the same as specifying `min: 0`; `false` means `min: 1`.
- `external: true | false`: Relevant for `services` only. If `true`, the service can be requested on the NBI. If `false`, the service can only be auto-created internally, as described in section References between services. The default value is `true`.
- `autoscale: true | false`: NOT YET SUPPORTED

For `resources` and `components`, the `type` expression can be used to control which actual Service Descriptorto instantiate for the given KindName. However, the `type` expression may not depend on $index, because it is only evaluated once during service decomposition in the `CHECKED_design` phase.

This means that all children of the same `component` or `resource` will be instantiated from the same Service Descriptor. This may change in future versions of HPE SP.

The `type` may be passed as a meta-data value in a service request.

For `services`, the type is always passed explicitly in the service request (or during auto-creation), so in this case, the different child services may have different types.

If the service inherits a component from an implemented Kind descriptor, the system merges any missing control attributes from that Kind. The `kind:` KindName attribute is allowed to specialize an inherited `kind`, but it may not arbitrarily alter the Kind of the child.

For example, if `NetCfg` defines a child, `port`, of `NetPort` Kind, then an implementation of `NetCfg`, such as `NeutronNet`, is allowed to define the `port` to be of `NeutronPort` Kind, provided that `NeutronPort` implements `NetPort`.

5.2.1.24 Child parameter bindings

For each child, the **`Parameters`** and **`Defaults`** sections allow the parent service to compute and pass a number of parameter **`Bindings`** and default **`Bindings`** to the child service.

Each binding consists of a **`BindLVal`** “left-value”, specifying the parameter to bind, and a parameter expression defining the value to pass to the child service.

The bindings in the **`Parameters`** section unconditionally set the child parameter value.

The bindings in the **`Defaults`** section only define the child parameter value if no other non-null value is passed from the parent, for example, using a setting in a northbound request.

The **`BindLVal`** must identify a defined parameter in the Kind Specification, identified in the `kind` and `version` attributes of the **`ChildControl`** control section of the child specification.

If the `BindLVal` contains an index, it binds only the parameter in that number of the child. If there is no index, it binds all instances of the child, except those that are bound by a specific index.

Specifying the ChildName only makes sense for bindings in Graphs (see section Inherited children).

Note that bound parameters must be defined in the child Kind Specification. It is not possible to bind parameters in the service specifications.

The bound parameters cannot be defined as `value` in the Kind Specification.

Note

The `location` and `locationname` special parameters are automatically propagated from parent to child services without specifying an explicit binding in the parent descriptor.

To disable this automatic propagation, `source: request` can be specified in the `location` and `locationname` parameter specifications of the child service. Note that a descriptor can specify a default `source` for all its parameters, so changing or setting the default `source` might unexpectedly change the behaviour of the automatic `location` and `locationname` propagation.

5.2.1.25 Referencing the child being bound

The `$param` parameter references used in child parameter bindings usually refer to parameters in the parent descriptor. A few special parameters, however, refer to parameters in the context of the child service being bound:

- `$this`
- `$parent`
- `$index`
- `$childtype`
- `$kind`
- `$type`
- `$min`
- `$max`
- `$count`
- `$childname`

So in a parameter expression in a child binding, `$this` refers to the child service object being bound; not to the surrounding service object. Similarly, `$index` refers to the index of the child object within the list of children of the same child component.

If there are multiple instances of the same child component, `$this` and `$index` can be used for making the child binding depend on each child instance.

5.2.1.26 Inherited children

A service may inherit children from its implemented Kinds. The child specification in the definition of the current service always supersedes inherited definitions and bindings.

It is an error if the child type does not agree between the inherited and the current child specification.

5.2.1.27 Graphs policies

Graph policies can be understood as conditional parameter bindings for components. This may be used for defining conditions on passing networks to VNFs (or vice versa), hence controlling the service graph created by the descriptor.

Note that the `graph` construct provides a way of specifying policies for complex, dynamic service-graphs. The construct should not be confused with any "intuitive" notion of a graph (so this has nothing to do with for example, any graphical rendition of the service).

Among other things, graph policies can be used to pass different networks to the components depending on any conditions that can be expressed with parameter expressions. Typical functions to use are `defined`, `undefined`, and `size` that test for the presence, absence, and size of child services and components.

Example::

```
graphs:
  access:
    connected:
      when: "@gt(@size($chain), 0)"
      box.wan_tunnel: $wan
      box.customer_subnet: $lan
```

The previous example specification reads as follows:

When the size of the `$chain` service-chain is larger than `0`, then access should be configured by connecting the `wan_tunnel` parameter of the `box` service to the `$wan` network service, and connecting the `customer_subnet` parameter of the `box` to the `$lan` network service.

As the example shows, graphs can be used for deciding which combinations of networks to pass to child VNFs. Each VNF, such as `box`, can define which networks it takes as parameters, and can have its own logic for connecting NICs to the networks. So the parent service can use the graph policy to define the network graph to set up depending on which VNFs are present and how they are scaled out. This means that the ETSI NFVconcepts of "Network Service" and "Forwarding Graphs" are just special cases of DDE Services and Graphs.

The graph conditions are evaluated during the Design phase (that is, the `CHECKED_design` transition), after all other parameters are evaluated (see section Parameter expressions), and after other parameter bindings for child resources, components, and services are computed. As other parameter values cannot refer to the graph policies, circular dependencies are avoided.

Graph policies are grouped into sections. Each section, for example, `access`, deals with a different set of parameter bindings.

It is the responsibility of the specification author to ensure that graph conditions are mutually exclusive when they pertain to the same child-parameter binding. If two conditions are true, and they both bind the same child parameter to different values, then the outcome is unpredictable, as the order of evaluation of graph conditions is not deterministic.

Important

Graphs can only be defined in Service Descriptors. Graphs in Kind Descriptors are ignored.

Important

Currently, it is not allowed to specify indexed bindings in graphs. So do not write a binding such as the following:

```
box[0].wan_tunnel: $wan
```

DDE presently ignores the [0] index, so indexed bindings do not work as expected.

Instead of indexed bindings, you can use the `$index` or `$this` special parameters for this purpose. For example, if a service `$chain` object is defined as a sorted list of child components, then writing:

```
myindex: "@myindex($this, $chain)"
```

will pass the index of each service in the service-chain as a parameter to the service. For more details, see section Referencing the child being bound.

5.2.1.28 Outputs

After the run-time engine executed actions on a service instance, it returns one or more result descriptors. The format of the results is documented for the respective supported protocols - see, for example, the REST API documentation.

Actions on one service may impact many other service instances. By default, the HPE SP run-time engine returns a response-descriptor for each affected instance, and each one contains all metadata (that is, all parameters) of each instance.

In some cases, it is useful to specify a different output-behavior of DDE. Using the **`Outputs`** section of the descriptors, you can specify that:

- Additional name-value pairs should be added to the response metadata (**`OutputParams`**).
- Only the one response-descriptor for the requested service should be returned (`include: affected: false`).
- Only the name-value pairs specified in the **`Outputs`** section should be returned, whereas other service parameters should be removed `(include: parameters: false)`.

Furthermore, it is possible to define a number of output templates that only take effect if a when-condition **`ParamExpr`** is satisfied. Each template can recursively contain **`OutputParams`** and **`OutputIncludes`**. The definitions of the templates take precedence over the previous output definitions.

The output parameters may refer to other parameters and to each other.

The OutputName is reflected in the metadata keys. Where an ordinary ParameterName must be a simple name consisting only of alphanumeric characters and underscores; an OutputName is allowed to contain any characters, such as dot-characters (“.”).

Note, that if an OutputName does not conform to the restrictions for ParameterName, the following limitations apply:

- Only OutputNames that conform to the standards for parameter names can be referenced in expressions defining other output parameters.
- Due to potential problems in marshalling, using exotic characters, whitespaces, or quotes in an OutputName is not recommended.

If an output parameter has no defined expression, it is assumed that it refers to the ordinary descriptor of the same name in the descriptor.

The expression of an output parameter may refer to parameters on child-objects of the service. As the `Outputs` section is evaluated after the DDE has stopped processing the service instances, all parameter values are stable at this point.

Note that if the final MTOSI state of the service is not `ACTIVE`, then some metadata on the service instances may not be defined or accessible.

If multiple responses are returned, then the response for the service identified in the request is always the first one in the list.

The `Outputs` sections for each of the affected responses are also evaluated. If one of these has `affected: false`, then any response for children of that service are removed from the list of response objects.

Support for declaring further attributes on output parameters may be added in future versions of HPE SP.

Example::

```
outputs:
  include:
    parameters: false             # Only include the parameters defined below
    affected: false               # Do not return a response for each affected
service
  cpus:                           # Include the cpus parameter of this service
  flavor: $vm.flavor              # Output flavor found on vm component of this
service
  flavorref: "@cat($flavor,-ref)" # Compute flavorref based on another output
parameter
  vm.name: $vm.servicename        # The output name may contain e.g.,"."-characters
  templates:
    special:                      # Define a conditional template named "special"
      when: $vm.isSpecial         # Use the below definitions when condition is
true
      include:
        parameters: true          # For special VM include all meta-data of service
      vm.users: "@sum($vm.user)"  # Add "vm.users" to the response meta-data
```

For the descriptor shown in the previous example, DDE returns precisely one response. If the VM is not special, the metadata has four entries: `cpus`, `flavor`, `flavorref`, and `vm.name`. If the VM is special, however, then all parameters of the service instance are included, and the `vm.users` is added as well.

5.3 Parameter handling

5.3.1.1 Parameter expressions

Expressions can be used in several places in the syntax. Expressions can be literals (YAML texts) or they can be parameter values (starting with `$`) or function calls (starting with `@`).

List expressions [1, 2, 3] are also supported inside parameter expressions as a short-hand for calling the `@list` function (see section Function list).

Parameter values can come from several sources:

- The service request coming through REST or from the inventory GUI
- A parameter binding from a parent service
- A referrer service (see section References between services)
- A previous value of the parameter on the same service

- Evaluation during the `CHECKED_design` state transition
- Response variables from a workflow (allowing collection of parameter values from the live systems)

As long as a service is in the `CHECKED` state, the evaluation has not been executed, so only the first two sources apply. This means that when a service is in the `CHECKED` state, it may have undefined or out-of-date values for some of its parameters.

Note that `@` cannot start a YAML text, so parameter expressions starting with a function call must be surrounded by double-quotes.

A parameter value or function call may be followed by any number of additional selectors:

- Field selectors - preceded by a dot, for example: `$system.password`
- Index selectors - square brackets, for example: `$nics[2]`

For documentation of the built-in functions, which can be called by `@` FunctionName (**`Args`**), see section Built-in functions.

In order to create a literal value starting with `@` or `$`, use two backslashes, for example, `\\$`. YAML interprets a single backslash as an escaped character, so `\\` is given to the specification parser as a single `\`.

Whitespace before each function argument is ignored. If you want an argument to be a string-literal starting with whitespace, escape it with `\\`.

Note that the system automatically converts between lists (arrays, collections) and scalars. Using a field-selector on a list returns the list of values that are obtained by applying the field-selector to each of the list elements. Applying a field-selector to an empty list returns an empty list.

Index values and function arguments can be entire parameter expressions themselves.

Index values do not have to be integers - if the value is inherently a map, then the index value is used as a key for looking up the corresponding value.

Note that the DDE expression syntax constitutes a major extension of the very limited set of functions available in Heat Orchestration Templates.

5.3.1.2 Types of values

The following types of values are supported:

- Simple values (strings, integers, and booleans)
- Objects (Services and physical or virtual Zystems)
- Lists and arrays
- Maps (any Java object supporting a method with the `String get(String key)` prototype)

Note

The system also interprets a list of pairs or a list of two-element lists as a map.

The system always treats a list containing exactly one element the same as the element it contains. So list-operators can be applied to simple values - treating them as one-element lists; and one-element lists can be used in any context expecting a simple value.

The system interprets service identifiers (strings) and the denoted service objects interchangeably. This means that a parameter that needs to contain a reference to another service may be supplied from the northbound interface simply as the corresponding `serviceid` (see section The serviceid identifier).

One important case, where lists are used is when working with service children (resources, components, or services). The child-name can be referenced as a variable in the service context, and will contain the list of actual child instances.

5.3.1.3 Special parameters

Some parameter names have a special meaning in the system, when declared (except for a few parameters mentioned below):

name: This is displayed in inventory view (unless overruled by other implementation specifics).

index: The index (starting from 0) of the child-service instance within a parent service. This parameter can be referenced within the bindings of a resource or component child specification.

> **Note**
>
> Do not declare or bind this parameter explicitly. This parameter does not need to be declared in the descriptor to be used.

count: The total number of child instances of the child specification in the parent.

zystem_name: ID passed to the Advanced Template Manager (see section Using Advanced Template Manager) as the inventory system where the service is created.

vnfrole: If defined, the service auto-creates and manages a VNF with the specified role in the equipment part of the inventory. The VNF ID can be used as `zystem_name` for other services.

this: Refers to the current object as an object. This is only needed if the object must be passed as a parameter, or if the intention is to access (using a field selector) a specific method on the object, which does not correspond to any of the parameters. This parameter does not need to be declared in the descriptor to be used.

> **Important**
>
> Do not address any ordinary parameters using `$this`, because it can cause problems with the evaluation order of parameters. See section Using field selectors on objects - $this.x.

parent: The parent of the current service. Returns null if the current service does not have a parent service. Can be used interchangeably with the `parentserviceid` parameter. This parameter does not need to be declared in the descriptor to be used.

> **Important**

Do not address any ordinary parameters using `$parent`, because it can cause problems with re-evaluation of the child if the parameters change. See section Using field selectors on objects - $this.x.

parentserviceid:

The identifier of the parent service or null if there is no parent. Can be used interchangeably with the `parent` parameter. This parameter does not need to be declared in the descriptor to be used.

Used in determining the management system to use for auto-creation of VNFs.

location:

Important

This parameter is propagated automatically from parent to child services. See section Child parameter bindings for details.

locationname: Synonym for `location`.

tenant_id: The ID of the Tenant owning the service.

tenantid: Synonym for `tenant_id`.

componentname: Name of the child (component, resource, or service) in the parent service that created the child service.

state: The current state of the service. When declared in a descriptor, any parameter value expressions that use `$state` are re-evaluated for every state-transition made by the engine. So to avoid unnecessary overhead, only introduce this parameter in the descriptor when needed.

children: List of all children of the service. This parameter does not need to be declared in the descriptor to be used.

auth_tenant_id: The ID of the Tenant last authorized to create or manage the service.

auth_username: The ID of the user who last created or managed the service.

transactionid: The ID of the last transaction that created or managed the service.

orderid: The ID of the last order that created or managed the service.

simulate: If true, then activations are simulated. Child-services on simulated services are also simulated (compare this to `preprovisioned` below). When showing or getting a service instance through the northbound interface, the value is shown as a field named `WasSimulated`. Also note that if `simulate` is true, the activation workflows for the service are executed in the DDE `skip_activation` mode so that all southbound actions (see section Pre-processing actions) are simulated.

preprovisioned: If true, then provisioning and activations (and the reverse) are simulated, but child services are not simulated.

autodelete:	If true, then the service is automatically deleted when the last child and last referrer are deleted. If false, then even if the service is auto-created, it remains after the last child is deleted.
softreferences:	Tab-separated list of identifiers for the services that the service depends on, other than the parent and the services in the `reference` and `auxreference` parameters.
customtree:	If false, the generic tree-view is used in the inventory tree; if true, then there must be a handcrafted inventory tree for this service. The parameter is only meaningful for root services.
reference:	See section References between services.
referrer:	See section References between services.
auxreference:	See section References between services.
vendor:	See section References between services.
osversion:	See section References between services.
elementtypev:	See section References between services.
linktype:	See section References between services.
ref_linktype:	See section References between services.

For VNFs (services having the `vnfrole` parameter) the following additional parameters are used in the creation of the VNF as virtual equipment:

ip:	Hostname or IP for connecting to the VNF.
username:	Username for connecting to the VNF
password:	Password for connecting to the VNF
name:	Name of the VNF
ems:	Specifically referenced management system controlling this VNF
role:	Role of the VNF (for example, `CoreRouter`, `TORSwitch`, `EORSwitch`, `AccessSwitch`, `BladeChassis`, `BladeNetwork`, `CPE`, `ComputeProvider`, `NetworkProvider`, `Hypervisor`, `VirtualSwitch`, `EventMonitor`, `EMS`, `NMS`)
vendor:	Company providing the VNF - used by the Advanced Template Manager (see section Using Advanced Template Manager)
osversion:	Version of the operating system - used by the Advanced Template Manager

element type: Type of virtual device - used by the Advanced Template Manager

Device serial number

serialnumber:

Note

This is an optional parameter. Many other optional parameters are available (for details, see the documentation of CNRM).

Note that “virtual” in VNF should be taken in a very broad sense: “virtual equipment” refers to any equipment that is delivered as a service. For example, a physical CPE box may also be modeled as a VNF using a `vnfrole` parameter.

5.4 Built-in functions

Service Descriptors support a library of built-in functions.

Note

More functions are planned, including functions that call custom Java methods, and functions that start HPE SP workflows and use the result.

The following functions are presently defined:

5.4.1.1 Function alldefined

`@alldefined(...)`

The function takes a number of lists as argument and returns a single list containing only the list-elements that were defined (that is, not null). The order of the defined elements is preserved.

5.4.1.2 Function and

`@and(...)`

The function returns `true` if all the given arguments evaluate as “true”, where truth is interpreted as follows:

- Boolean true is true.
- Integer values equaling 0 are true, all other numbers are false.
- The null value is false.
- Lists are true if all elements in the list are true. The empty list is true.
- Any other objects are converted to strings, and are true for all values except for the following:

1. `false`
2. `""`
3. `no`

4. `n`
5. `disabled`
6. `off`
7. `null`
8. `-`

5.4.1.3 Function bean

```
@bean(beanclass, [ find, [ key1, [ key2 ] ] ])
```

The function applies the given key or keys to the given `findBy` method on the given HPE SP inventory bean class. Returns the found inventory bean object or list of objects - or null if none are found.

Example: `@bean(ApplTemplate, findByName, $imageid)`

If only the beanclass is given, then the function returns the result of the `findAll` method.

The method is assumed to take the database connection object (jsql), and the given number of keys as arguments (0, 1, or 2 keys).

5.4.1.4 Function cat

```
@cat(...)
```

The function returns a string concatenated from the list of given arguments converted to strings. Sub-list elements also take part in the result. Null values are skipped.

5.4.1.5 Function children

```
@children(service [, component])
```

The function takes a service (as defined for the `@service` function in section Function service) and returns a list of children of that service. If the component parameter is specified, only children of that component definition are returned.

Important

The child service objects will be the non-shadow versions as currently found in the inventory. The result may be misleading if the services are not designed or provisioned yet.

5.4.1.6 Function copies

```
@copies(n, v)
```

The function creates a list of length `n` where each value is a copy of `v`.

5.4.1.7 Function decrypt

```
@decrypt(arg)
```

The function decrypts a value that was previously encrypted as a HPE SP password. If applied to a list of encrypted values, it returns a list of decrypted values.

5.4.1.8 Function defined

```
@defined(...)
```

The function returns `true` if all the given arguments are defined, that is, not null. Note that the `""` and `null` text-strings are considered as null values.

5.4.1.9 Function divide

```
@divide(n, m)
```

The function works for integers, and returns the result after dividing `n` by `m`.

- If `n` is a list, it uses the size of the list as `n`.
- Division by zero causes an exception.
- Division by `null` is `null`.

Note

Use the Function modulo to get the remainder.

5.4.1.10 Functions eq and equals

```
@eq(arg1, arg2)

@equals(arg1, arg2)
```

The two forms are synonymous.

The functions return true if the two arguments are equal, interpreted as follows:

- The `null` value, the `""` and `"null"` stings, and the following values representing boolean `false` are all treated as equal: `false`, `no`, `n`, `disabled`, `off`, `-`.
- Service objects and Zystem objects are compared by their `serviceid` identifiers.
- If both arguments represent valid life-cycle states (see figure Service life-cycle state transition model), the case is ignored in the comparison.
- All other arguments are compared as strings. The comparison is case sensitive.

5.4.1.11 Function fieldlist

```
@fieldlist(field, object*)
```

The function takes the name of a field and one or more objects, and returns the list of values obtained by selecting the given field from each of the objects.

- If the field-value is `null`, the list contains `null`.
- If the first argument is a list of field-names, the function returns a list of the field values for each of the fields. Each of these list-results will have the same length.

Note

Using the `$objects.field` syntax is usually preferable, as this also distributes over lists of objects.

5.4.1.12 Function firstarg

```
@firstarg (...)
```

The function always returns the first argument, leaving any subsequent arguments unused.

This is useful for enforcing an otherwise undiscovered dependency between a parameter or binding expression and a number of additional parameters. All arguments are evaluated before the return is produced.

5.4.1.13 Function firstdefined

```
@firstdefined(...)
```

The function returns the first defined element found in the list(s) of elements (see also Function defined).

5.4.1.14 Function flatten

```
@flatten(...)
```

The function returns the given arguments as a list where all elements are scalars. So if any of the arguments contains lists or lists-within lists, then the result value has “flattened” the structure.

Null elements are preserved. To produce a list where null-value elements are removed, use the Function alldefined.

5.4.1.15 Function flow

```
@flow(hops, list)
```

This function can compute a linear flow graph between a number of VNFs, network-elements, and so on.

It takes a list of connector objects (for example, network services) and a number of hops (integer), and returns a list of pairs of connecting objects corresponding to the given hops.

The two sides of a pair object can be found as `pair.left` and `pair.right`.

Example:

Assume the list is:

```
list = [netA, netB, netC, netD]
```

Then:

```
@flow(1, $list) = [(netA, netD)]
@flow(2, $list) = [(netA, netB), (netB, netD)]
@flow(3, $list) = [(netA, netB), (netB, netC), (netC, netD)]
```

The idea is, that the flow always starts on `netA`, and always ends on `netD`, and the `hops` parameter decides how many of the intermediate nets are required.

So if you have two Services (VNFs), X and Y, that need to be connected by a linear flow-graph, then the first element of `@flow(2, $list)` identifies `(netA, netB)` as the two networks for Service X, and `(netB, netD)` as the two networks for Service Y.

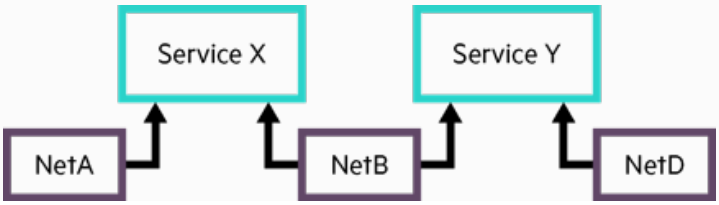

Figure 5–1 Linear flow graph computed by the @flow function

5.4.1.16 Functions ge, gt, le, lt

```
@le(arg1, arg2)

@lt(arg1, arg2)

@ge(arg1, arg2)

@gt(arg1, arg2)
```

The functions compare the values of the first and second arguments by their natural ordering and return a Boolean result.

The two forms ending in “e” return true if the arguments are equal, as defined for the `@eq` function in section Functions eq and equals.

If not equal, then `@ge` returns the same result as `@gt`; and `@le` returns the same result as `@lt`. Furthermore, `@ge` always returns the same result as `@not(@lt)`; and `@le` always returns the same result as `@not(@gt)`.

The natural comparison is defined as follows:

- False (and its equivalent forms `no`, `n`, `disabled`, `off`, ``` `-` ```, `null`, `""`, `"null"`) are always less than any other value of any type.
- True (and its equivalent forms `true`, `yes`, `y`, `enabled`, `on`, `+`) are always greater than any other value of any type.
- Service objects and Zystem objects are compared by their `serviceid` identifiers. A child service is regarded as less than any of the services in its parent hierarchy. Consequently, the root service is greater than any of its children.
- If both objects represent numbers, they are compared by their numeric value.
- If both arguments represent valid life-cycle states (see figure Service life-cycle state transition model), they are compared according to their “definednes” in the state-transitions. The result can be found by comparing the associated numbers below:
  1. NOTTHERE
  2. TERMINATED
  3. CHECKED
  4. DESIGNED
  5. RESERVED
  6. PROVISIONED
  7. ACTIVE
- If none of the above applies, the two arguments are regarded as strings and are compared by lexicographical order. Sorting of characters is numerical by the native Java language Unicode representation, and does not take locale into consideration.

5.4.1.17 Function if

```
@if(cond, trueArg[, falseArg])
```

- If the condition is `true`, then the value of the `trueArg` parameter is returned.
- If the condition is `false`, then the value of the `falseArg` parameter is returned.
- If the condition is `false` and the `falseArg` parameter is not defined, the function returns `null`.

5.4.1.18 Function ipinsubnet

```
@ipinsubnet(subnetip, netsize, ipnumberfromstart)
```

The function returns an IP Address from the specified subnet.

Parameters:

- subnetip – Any IP Address in the intended IPv4 subnet of the given `netsize` (for example, the start address or broadcast address).
- netsize – Size of the subnetwork.
- ipnumberfromstart – Number (between 0 and the size of the net) of the IP Address to return. For example, passing 0 as the `ipnumberfromstart` returns the start-address of the subnet.

5.4.1.19 Function lastdefined

```
@lastdefined(...)
```

Returns the last defined element found in the list(s) of elements (see also Function defined).

5.4.1.20 Function lastname

```
@lastname(arg)
```

If the argument is a string, the function returns the last part of the string not containing any dot "." or slash "/" characters. If the argument is a service object, the function is applied to its `servicename`.

Note

The function is useful for finding the last part of a generated `servicename`, for example, for presentation purposes.

5.4.1.21 Functions le, lt

```
@le(arg1, arg2)

@lt(arg1, arg2)
```

See the common description of `ge`, `gt`, `le`, `lt` in section Functions ge, gt, le, lt.

5.4.1.22 Function list

```
@list(...)
```

The function returns the given arguments as a list. If any of the arguments contain lists or lists-within lists, the result contains these lists-within-list.

If you need the list elements to be all scalars, use the function `@flatten` instead (see section Function flatten).

Null elements are preserved. To produce a list where null-value elements are removed, use the `@alldefined` function (see section Function alldefined).

At any level of expressions, the YAML list syntax may be used instead of `@list`.

For example:

```
range: [a, b, c]
```

has the same meaning as:

```
range: "@list(a, b, c)"
```

5.4.1.23 Function lookup

`@lookup(val, mapping)`

`@lookup(val, fromField, toField, mapping)`

The function looks for a mapping for a given `val` parameter, and if it finds a mapping, it returns the mapped value.

If there is no mapping for `val`, the function returns `null`.

See also the function `translate` in section Function translate. Instead of null, `translate` returns the original `val` value if no mapping is found.

If the `val` argument is a list, then the `lookup` function distributes over the list, that is, it returns the list of mapped values for each of the elements in the `val` list.

The `lookup` function supports two forms of `mapping`: Direct and using Fields.

5.4.1.24 Direct mapping form of @lookup

For the direct mapping form of the function, only the mapping parameter is provided. The value can be one of the following types:

- A Java object implementing the Java Map interface
- A pair object (see section Function flow), defining a mapping from the left value to the right value
- A `list` object with precisely 2 elements, defining a mapping from the first element to the second
- A Java object with a method called `get` that can take the given `val`, and return the resulting value
- A Java object with a getter method corresponding to `val`. So if `val` is `Abc`, then the result of calling getAbc() is returned
- A list of any of the supported mapping types (including this one), where `lookup` finds the first element that defines a mapping for `val`

The following expression uses direct mapping:

```
@lookup($p, [[1, low], [2, medium], [3, high])
```

If `$p` has the `1` value, the result is `low`, if the value is `2`, the result is `medium`, and so on.

5.4.1.25 Fields mapping form of @lookup

For a fields mapping form of the function, two fields must be provided in addition to `val` and `mapping`:

- `fromField`
- `toField`

The `lookup` function searches for a mapping for the `fromField`, according to the rules of the direct form, and if it finds one where the mapped value equals the given `val`, then the `toField` is evaluated and the result is returned.

Example:

```
@lookup(438f3038-e711-3631-9043-97ad27982b55, Uuid, Serviceid,
$services)
```

- If the `$services` is a list of service objects, and `$uuid` contains a UUID , then this function call returns the `serviceid` identifier of the service in the `$services` list that has uuid `438f3038-e711-3631-9043-97ad27982b55`.
- If none of the `$services` objects have the given UUID, then the result is `null`.

5.4.1.26 Function max

`@max(...)`

The function returns the maximum of a number of values.

- If all values are numbers, then it returns the largest one.
- If some of the values cannot be interpreted as numbers, then it returns the lexicographically largest one.
- If some, but not all values can be interpreted as numbers, then it pads the values that are not numbers with 10 zeros to achieve pseudo-numerical comparison.
- Empty and `null` values are skipped.
- If no numbers were compared, it returns `null`.
- If any of the arguments are lists, the function is applied to the list-elements (the arguments are flattened).

5.4.1.27 Function min

`@min(...)`

The function returns the minimum of a number of values.

- If all values are numbers, then it returns the smallest one.
- If some of the values cannot be interpreted as numbers, then it returns the lexicographically smallest one.
- If some, but not all values can be interpreted as numbers, then it pads the values that are not numbers with 10 zeros to achieve pseudo-numerical comparison.
- Empty and `null` values are skipped.
- If no numbers were compared, it returns `null`.
- If any of the arguments are lists, the function is applied to the list-elements (the arguments are flattened).

5.4.1.28 Function minus

```
@minus(arg, list (, list)*)
```

The function subtracts the list of numbers from the first argument.

The parameters are interpreted as numbers as follows:

- Numbers are themselves.
- Boolean true is `1`, Boolean false is `0`.
- Strings are converted as decimal numbers if possible. If not possible, the value is `0`.

5.4.1.29 Function modulo

```
@modulo(n, m [, offset])
```

This function works for integers, and returns the remainder after dividing `n` by `m` and adding the optional `offset`.

- If `n` is a list, it uses the size of the list as `n`.
- An exception is raised if `m` is zero.
- If `m` is `null`, the result is `null`.

5.4.1.30 Function multiply

```
@multiply(...)
```

The function returns the product of the given integer arguments. The elements are interpreted as numbers as follows:

- Numbers are themselves.
- Boolean true is `1`, Boolean false is `0`.
- Strings are converted as decimal numbers if possible. If not possible, the value is `0`.

5.4.1.31 Function myindex

```
@myindex(arg, list (, list)*)
```

If the first argument object is found in the list(s) given, the function returns the index of the object, after concatenating all the lists into one. Null elements in the lists are counted.

If the first argument is not found in any of the lists, the function returns `null`.

5.4.1.32 Function netmask

`@netmask(arg)`

If the argument is a valid IPv4 subnet size (numbers 0-32), then it is converted to the corresponding net-mask. Any other arguments return `null`.

Example:

Calling `@netmask(24)` returns the net-mask `255.255.255.0`.

Note

More functions for manipulating IPv4 and IPv6 addresses are planned.

5.4.1.33 Function nopackage

`@nopackage(arg)`

The argument is interpreted as a string, and the part after the last occurrence of the package-separator `::` is returned. If the argument is `null`, the function returns `null`.

5.4.1.34 Function not

`@not(arg)`

Returns `false` if the given argument is true, and returns `true` if the argument is false.

If `@not` is applied to a list, it returns a list of `true` and `false` values; one for each element in the original list.

Truth is interpreted as follows:

- Boolean `true` is true.
- Integer values equaling `0` are true, all other numbers are false.
- The `null` value is false.
- Lists are true if any one of the elements in the list is true. Empty lists are false.
- Any other objects are converted to strings, and are true for all values except for the following:

1. `"false"`
2. `""`
3. `"no"`
4. `"n"`
5. `"disabled"`
6. `"off"`
7. `"null"`

8. `"-"`

5.4.1.35 Function numbers

```
@numbers(first, last)
```

The function returns a list of numbers starting with the first argument and ending with the last argument.

So calling `@numbers(2, 6)` returns the following list of values: `2, 3, 4, 5, 6`.

5.4.1.36 Function or

```
@or(...)
```

Returns `true` if any of the given arguments evaluates as "true", where truth is interpreted as follows:

- Boolean `true` is true.
- Integer values equaling 0 are true, all other numbers are false.
- The `null` value is false.
- Lists are true if any one of the elements in the list is true. Empty lists are false.
- Any other objects are converted to strings, and are true for all values except for the following:

1. `"false"`
2. `""`
3. `"no"`
4. `"n"`
5. `"disabled"`
6. `"off"`
7. `"null"`
8. `"-"`

5.4.1.37 Function pad

```
@pad(arg)
```

This function pads a string (`param 1`) left or right (`param 4`) to a given number of characters (`param 2`) with a specific character (`param 3`).

- `param 1` and `param 2` are mandatory.
- `param 3`: default is "0"
- `param 4`: default is `left`

5.4.1.38 Function remove

```
@remove(lst, ...)
```

The function removes the elements of all the subsequent arguments from the list given as the first element.

5.4.1.39 Function replace

```
@replace(string, regexp, replacement)
```

The function replaces all matches of regexp inside the first string argument

with the `replacement` string.

Returns `null` if the first argument is `null`. Returns the `string` unchanged if the `regexp` is null. If the `replacement` string is `null`, it is interpreted as an empty string.

If the first argument is a Service object, the `string` is its `serviceid` identifier.

If the first argument is a list, the function applies the replacement to each element individually, and returns the list of results.

5.4.1.40 Function reserve

```
@reserve(services, ...)
```

The function takes one or more services (each interpreted as the `@service` function in section Function service), and returns the first one that is not the target of any `reference` or `auxreference` parameter (see section References between services) of another service.

The function selects the oldest candidate service that is not referenced, ensuring that components are reserved from the lowest number up.

If the service already references one of the services in the list, then that service is returned. This ensures stability of the system.

5.4.1.41 Function select

```
@select(index, arg*)
```

Takes an index and a list of arguments. Returns the index-numbered argument passed to the function, or returns `null` if the index is larger than the number of arguments.

If the first argument (the index) does not evaluate to an integer between 0 and the number of subsequent arguments, then `null` is returned.

Indexing starts from 0, so:

- `@select(0, a, b)` returns `a`.
- `@select(1, a, b)` returns `b`.
- `@select(2, a, b)` returns `null`.

5.4.1.42 Function service

```
@service(arg)
```

Interprets the given argument as a service object, which is returned.

The argument can be:

- A service object
- A service identifier
- A UUID for a service
- A servicename, provided the service being sought belongs to the same Tenant

If the argument cannot be interpreted as a service, the function returns null.

5.4.1.43 Function siblings

```
@siblings([service [, criterion]])
```

The function interprets the given argument as a service object, whose sibling services are returned.

The given service itself is never one of the services in the returned list.

If the given service is a root service, the function returns root objects for the same Tenant instead.

If no arguments are given, the siblings of `$this` are returned.

The supported criteria are:

- `ALL`: No criterion is applied.
- `KIND`: Only services implementing the same Kind as the one given are returned. If the service has a parent, the Kind is that of the corresponding component for this service in the parent. If the service does not have a parent, then only services of the exact same service definition are returned.
- `CHILDTYPE`: Only services of the same type as the given service are returned: SERVICE, COMPONENT or PARENT.
- `CHILDNAME`: Only services with the same child name as that of the given service are returned. This is more restrictive than `KIND`, because there may be several children of the same Kind but with different child names.

If no criterion is given as argument, it defaults to `CHILDNAME` if a parent exists and it defaults to `KIND` if there is no parent.

5.4.1.44 Function size

```
@size(...)
```

The function returns the number of elements in the lists contained in the given argument(s). Null values in the lists are counted.

5.4.1.45 Function sort

```
@sort([ sortspec , ] listargs ...)
```

Sorts elements in the list arguments.

If only one argument is provided, the sorting specification is regarded as omitted, and the list is sorted according to the default comparison for the list element types:

- Numbers are compared numerically.
- Objects that implement the Java `Comparable` interface are compared using the implemented `compare` method. If the two objects are not the same class, the `compare` method is used only if both objects have a common super-class that implements the `Comparable` interface.
- All other values are converted to strings and compared lexicographically.

The sorting specification can consist of field/parameter names and/or ascending/descending sorting directions. The default direction is ascending.

If the same object appears multiple times in a row (after sorting), only one of the objects is returned.

The supported sorting specification syntax is:

SortSpec ::= `SortDir` | `SpecPair` | `Spec`
SortDir ::= asc | desc
Field ::= FieldName
Spec ::= `Field` | `SpecPair`
SpecPair ::= `Field` `SortDir` #Expressed as a YAML list with two elements
A FieldName can select a value related to an object in the list:

- If the list-element is a service, then the field name may refer to a service parameter.
- If the list-element is an object, then the `abc` field name refers to a method called `getAbc()` or a method called `abc()`.

Examples of calls of the `@sort` function:

- `@sort($lst)` Default sort order ascending

- `@sort(asc, $lst)` Default sort order ascending
- `@sort(desc, $lst)` Default sort order descending
- `@sort([nbr], $lst)` Sort ascending according to the value of the field named `nbr`
- `@sort([nbr, desc], $lst)` Sort descending according to the value of the field named `nbr`
- `@sort([[nbr, desc], loc], $lst)` Sort descending according to the value of the field named `nbr` , then ascending according to the field named `loc`
- `@sort([[nbr, desc], [loc, desc]], $lst)` Sort descending according to the value of the field named `nbr` , then descending according to the field named `loc`

5.4.1.46 Function sum

```
@sum(...)
```

Returns the sum of the given integer arguments. The elements are interpreted as numbers as follows:

- Numbers are themselves.
- Boolean true is `1`, Boolean false is `0`.
- Strings are converted as decimal numbers if possible. If not possible, the value is `0`.

5.4.1.47 Function tolower

```
@tolower(arg)
```

The argument is interpreted as a string, and the function returns the value converted to lower-case. If the argument is `null`, the function returns `null`.

5.4.1.48 Function toupper

```
@toupper(arg)
```

The argument is interpreted as a string, and the function returns the value converted to upper-case. If the argument is `null`, the function returns `null`.

5.4.1.49 Function translate

```
@translate(val, mapping)

@translate(val, fromField, toField, mapping)
```

The function looks for a mapping for given a `val` parameter and if it finds a mapping, it returns the mapped value.

If there is no mapping for `val`, the function returns the given `val` value.

In all other respects than the result, if there is no mapping, the `translate` function behaves exactly like the `lookup` function. Instead of `val`, `lookup` also returns `null` if no mapping is found.

For documentation of the mapping functionality, see the description of `lookup` in section Function lookup.

5.4.1.50 Function true

```
@true(arg)
```

The function tests if the argument is explicitly true - as opposed to `@not`, `@and`, and `@or`, which treat strings that are not explicitly false as being true.

If applied to a list, the function returns a list of the same length, where each element is true if and only if the function applied to that element is true.

The following values are true:

- Boolean true is true.
- Integer values equaling 0 are true, all other numbers are false.
- Any other objects are converted to strings, and are true for the following values:

1. `"true"`
2. `"yes"`
3. `"y"`
4. `"enabled"`
5. `"on"`
6. `"+"`

All other values are false.

5.4.1.51 Function undefined

```
@undefined(arg)
```

The function returns `true` if the argument is null.

If applied to a list of values, it returns a list of the same length, with a `true` value for each undefined element, and a `false` value for each defined element.

5.4.1.52 Function unreserved

```
@unreserved(services, ...)
```

The function takes one or more services (each interpreted as the `@service` function in section Function service), and returns the list of those that are not the target of any `reference` or `auxreference` parameter (see section References between services) of another service.

The function sorts by the oldest candidate service that is not referenced, ensuring that components are listed from the lowest number up.

If this service already references one of the services in the list, then that service is returned as the initial value, becoming the default. This ensures stability of the system.

5.4.1.53 Function vnf

`@vnf(arg)`

The function interprets the given argument as an HPE SP inventory object of type Zystem, which is returned.

The argument can be:

- A Zystem object or zystem id (known as `NetworkElementId` in the HPE SP inventory)
- A service object or service id, which defines a Virtual Network Function. That VNF is then returned.

If the argument cannot be interpreted as a system or service, or if the service is not a VNF (that is, the `vnfrole` parameter is not defined) the function returns `null`.

It is important to understand that there can be two or more zystems associated with a service:

- The Zystem where the service is configured. This, by default, is identified by the parameter called `zystem_name` on the service.
- The Zystem that the service creates when the service defines a Virtual Network Function (VNF). Such services are identified by having a parameter named `vnfrole`, whose value is not null.

The `@vnf` function returns the VNF interpretation of the service, whereas the `@zystem` function, described in section Function Zystem, returns the Zystem platform, where the service is created.

Example:

A vRouter service may be implemented as a VM running in a Hypervisor environment. Then:

- `@zystem` returns a Zystem object representing the Hypervisor where the VM is managed. Use this for actions operating on the VM in the Hypervisor environment, for example, start and stop.
- `@vnf` returns a Zystem object representing the VM itself. Use this for configuring services by connecting to the running VM itself.

5.4.1.54 Function Zystem

```
@zystem(arg)
```

Finds the Zystem object where a service is configured.

The argument can be:

- A Zystem object or zystem id (known as `NetworkElementId` in the HPE SP inventory)
- A Zystem object where the given service was configured.

For more details, see the description of the function `@vnf` in section Function vnf.

5.4.1.55 Custom functions

It is possible to add custom functions written in Java. Use the following simple interface for such functions:

```
com.hp.ov.activator.ci.dde.descriptor.service
```

5.5 References between services

There are two dedicated parameter names that indicate a cross-reference between two service hierarchies:

- `reference`
- `auxreference`

Whenever one or both of these parameters are defined in a service descriptor, they will receive special treatment.

The type defined for these parameters must be the name of a defined service or Kind. The value must be defined using the value attribute; it is not allowed to pass the value as a binding.

Figure A reference from the VM hierarchy to the network hierarchy shows a relationship between virtual machines and networks.

Figure 5–2 A reference from the VM hierarchy to the network hierarchy

The descriptor for the Link-service, has a `value` -expression where the result is allowed to be a parent service of the specified type. In the example, the type is specified to be `NetPort`, but the `value` expression may refer to a parameter `$network`, where the value can be of kind `Network` or `Subnet`.

In this case, the system will auto-compute the set of possible instantiations of services in order to auto-create a descendant service of the specified Kind, and set the actual reference to that service.

In the example, the system will auto-create a `NetCfg` subservice of the network, and then a `NetPort` service as a child of the `NetCfg` to bridge the gap from Network to Port.

Both the auto-created service and the referring service are first-class services, with their own service descriptors, parameters, actions and possibly child-services (resources, components, services).

Note that the auto-creation mechanism is prepared to be extended to be able to handle the full complexity of stitching network connectivity between several network management systems, routers, MPLS, etc. However, that functionality is not yet complete.

5.5.1.1 Dependencies between Services

When the engine evaluates services, it uses the dependencies between services to compute the correct order of actions. It always creates a service after any services it depends on, and it always deletes a service before deleting any services it depends on.

A service depends on:

- Its parent service (unless this is a root service)
- Any service referenced by the `reference` parameter
- Any service referenced by the `auxreference` parameter
- Any service or list of services passed from the parent as a parameter binding
- Any service that any of the above depend on (the depends-on relationship is transitive)

The dependencies define the processing sequence for requests to create, modify and delete any service. Two services that do not depend on each other in any of the listed ways may be processed in an unpredictable order or concurrently at the discretion of the engine.

Note

Passing entire objects to child objects should often be preferred over passing parameters of those services individually. There are two reasons for this:

- This will create a dependency relation from the service receiving the parameter and the service being passed to it. This is particularly important for ensuring the correct order of service tear-down, since the service receiving the parameter will always be torn down prior to the one being passed as a parameter.
- If a service receives one or more of its sibling components as a parameter, then those siblings will not have been designed yet at the time of binding the parameter in the parent service. If parameters that have not yet been designed are being passed individually, they may not have the intended values. If the entire object is passed and then its parameters are dereferenced in the receiver, then DDE guarantees that the service will have been designed.

This is particularly the case when a component/resource needs to access parameters of one of its sibling services

5.5.1.2 Values of the referred service available in the referring service **descriptor**

For `$reference` and `$auxreference`, the following fields are populated immutably during auto-creation and may be used in the referencing service:

- `$reference.serviceid`
- `$reference.tenantid`
- `$reference.servicename`
- `$reference.servicetype`
- `$reference.serviceversion`
- `$reference.servicecomponent`
- `$reference.childtype`
- `$reference.childname`
- `$reference.entity`
- `$reference.location`
- `$reference.zystem_id`
- `$reference.zystem_name`
- `$reference.parent`
- `$reference.type`
- `$reference.linktype`
- `$reference.linktype2`
- `$reference.linktype3`
- `$reference.linktype4`
- `$reference.linktype5`
- `$reference.preprovisioned`

If any other parameters on a referenced service is used, DDE will automatically propagate the change to all referring services. If this propagation alters the value of an activation-affecting parameter (see section Controlling service comparison) of a service that is already in `ACTIVE` state, DDE will automatically re-activate the service. So while this will work, it may cause an extra activation call to first set up the service, and then apply the change.

Re-activation means that the service is brought from `ACTIVE` to `PROVISIONED` and back to `ACTIVE`, executing any necessary `ACTIVE_deactivate` and `PROVISIONED_activate` workflows.

Note that this type of propagation will not cause a complete re-design of the service. All parents, references and children will remain completely unaffected.

Also note that list of immutable fields can also be safely used on `$reference.parent`, e.g. `$reference.parent.servicename`, but change propagation will only be executed from the directly referenced service. So when propagation is desired, do make sure the relevant parameters can be accessed directly on `$reference`, and not on `$reference.parent`.

5.5.1.3 Parameter-bindings for auto-created service instances

If the auto-created service instances define a number of designated parameters, these parameters can be populated from the referrer service instance. This applies both to the immediately referenced service, and any auto-created parents. The special parameter names are:

- `zystem_name`

- `zystem_id`
- `location` (NB! This will be populated with the location of the :term:`Zystem` object; not with the location of the referrer service itself)
- `preprovisioned`
- `referrer`
- `ref_linktype`
- `ref_linktype2`
- `ref_linktype3`
- `ref_linktype4`
- `ref_linktype5`

For example, if the auto-created service has an unbound parameter named `"zystem_name"`, then its value will be derived from the referring service, not from the parent service. This applies even if the parent service explicitly attempts to pass a value for the `zystem_name` parameter. The system will search the location-hierarchy in the inventory for a management system that manages the relevant resource type for the referring service.

In order to override the referrer-binding behavior, you should pass the value as a different parameter (e.g. `parent_zystem_name`) and then make an explicit binding policy within the descriptor of the auto-created service.

If the parameter `$preprovisioned` is true on the referrer, then it means that the auto-created service must also have been pre-provisioned. So in this case, the auto-created services will be recorded as pre-provisioned, and will never be actively removed on the supporting systems. Note that the pre-provisioned property will be “sticky” once a service has been auto-created.

So if a NIC is attached to a network, the created `NetCfg` and `NetPort` services will be created on the network management system associated with the NIC location. For a vNIC service on a VM, this means that the `NetCfg` will be created as a `vSwitch` service on the same hypervisor as manages the VM.

This functionality is currently partially hard-coded and may not yet work for other service kinds than `NIC` and `Network`.

If the referrer service needs to transfer additional parameters to the auto-created services, it can do so by defining parameters named `linktype2 ... linktype5`. Their values will be automatically transferred to any respective parameters named `ref_linktype2 ... ref_linktype5`, in any of the auto-created service instance.

Note, that if used, these values should also be used in the auto-name of the service (see section Naming of auto-created service instances (auto-names)), as they will only be populated upon creation - so any subsequent references to the auto-created service will not set these values.

As mentioned in section Deciding the Service Type for auto-created services,
the `linktype2 ... linktype5` parameters do not participate in the process that decides the referenced service types.

In many cases it will be simpler to dereference parameters on the referrer parameter directly.
Using `$referrer.param` will work well in most cases.

5.5.1.4 Naming of auto-created service instances (auto-names)

When the DDE system auto-creates a service instance, being referenced from another service, the system needs to automatically generate the service-name and unique identifier of the new service instance.

Note that this refers specifically to the `"ServiceName"` field in the HPE SP service inventory. It should not be confused with the $name parameter, which is used for presentation in the inventory GUI. So the chosen name should be defined for its functional effect, and it does not need to have any graceful presentation qualities.

The **`SpecAuto`** section of the specification defines the auto-naming strategy. The syntax for this section is similar to that of the **`NameAuto`** (see section Auto-creation of service names), but the set of supported auto-parameters is different.

Examples:

```
autocreate: $parent.on_$zystem
autocreate: $parent.for_$referrer2
```

Since the generated name must be unique (for each Tenant), deciding which parameters are part of the name also implicitly defines when a service should be created, and when an existing one can be re-used.

The name consists of free text marked up with a limited set of special parameters (not to be confused with the ordinary specification parameters defined in section Auto-creation of service instances).

The supported special parameters are (do not confuse these with ordinary descriptor parameters):

$zystem:	The management Zystem defined on the referrer object, if any. DDE ensures that if a `zystem_name` parameter exists, then its value will have been computed at the time of auto-creating the referenced service.
$linktype:	The `linktype` parameter if defined on the referrer object. DDE ensures a value for this parameter will have been computed at the time of auto-creating the referenced service. Note that the linktype also participates in the selection of the service type as explained in section Deciding the Service Type for auto-created services.
$linktypeN:	The `linktype1` … `linktype5` parameters can be used for transferring additional name-components from the referrer. The corresponding `ref_linktype` parameters can also be used as parameters in the referenced service (see section Deciding the Service Type for auto-created services).
$referrer:	The service-name of the referrer service.
$referrer2:	The service-name of the parent (if any) of the referrer service.
$tenantid:	The tenant-id of the referrer service.
$parent:	The service-name of the parent of the auto-created service.
$parent2:	The service-name of the grand-parent of the auto-created service.

Note that it is best practice and highly recommended to start the auto-name with `$parent`. This is not required, but will minimize the size of the generated service-id.

If `$parent` does not occur in the auto-name, then the uniqueness constraint on service-names will be violated if the same name is used for auto-created services having two different parent services.

If `$referrer` occurs in the name, then there will be an auto-created service per referrer.

If `$tenantid` occurs in the name, then there will be an auto-created service per Tenant referring to the service.

If `$referrer2` occurs in the name, then there will be an auto-created service per referrer-parent. Two referrer children sharing the same parent service (`$referrer2`) will reference the same service, because the auto-created services will have identical service-names for both children.

5.5.1.5 Deciding the Service Type for auto-created services

There may be multiple ways to achieve the auto-creation of child services for a reference. For example, there may be a generic `NetCfg` service, and a specialized `VswitchCfg` Service Specification, that also implements the `NetCfg` Kind. So the system must be instructed whether to instantiate the `NetCfg` or `VswitchCfg` descriptors.

As mentioned above, the system first determines the management system (physical or VNF) as recorded in the inventory. It then needs to select the most appropriate Service Descriptor to instantiate. More advanced methods may be introduced later, but the current implementation simply looks for constantly defined values (only literal, no functions or parameters) on the candidate descriptors:

- `vendor`
- `osversion`
- `elementtype`

If any of these parameters are defined, their constant, literal values **(functions and parameter references are not allowed)** will be compared to the names of the corresponding attributes on the management system of the referenced system and the associated names of groups of osversions and elementtypes. See the HPE Service Activator documentation of the "Common Network Resource Model" for more information on this model.

Note that this logic is aligned to that used by the Advanced Template Manager (see section Introduction) for locating activation templates and workflows.

In some cases, the selection of the service type to instantiate will also depend on the referrer. This is particularly relevant if the referrer does not define a management system at all. If these cases, the referrer may specify a parameter called

- `linktype`

The referenced specifications may use `$linktype` in the auto-name, and may specify a corresponding constant-valued parameter:

- `ref_linktype`

This value will participate in the decision of the service type.

Note that the value of the `linktype` parameter may be defined using any parameter-expression - only the `ref_linktype` must have a literal value if specified.

Note that the additional `linktype2 ... linktype5` parameters can appear in the auto-name, but they do not(presently) participate in the decision of the service-type for auto-created services.

There is no special support for a `ref_tenantid` parameter. Simply use `$referrer.tenantid` to access theTenant id of the referrer.

5.5.1.6 Auto-creating a root service

In the use-cases above, the starting point for finding/creating a referenced entity is a `value:` that identifies an existing service, which will become the parent of the created service.

In some cases, it may be desirable to auto-create a root service without any parent. To do this, the `value:` must evaluate to a string that identifies the name of a Service Descriptor. So if there is a service `Pkg::Wrapper-001.03`, and if his Service Specification defines an `autocreate` name, then having the value `"Pkg::Wrapper-001.03"` as the value: for the reference will create a service of that type directly under the Tenant (without a parent service) in the inventory.

To make this mechanism work, some rules should be known:

- The autocreate name in the referenced service type cannot contain $parent or $parent2, because there will be no parent service.
- In the string, identifying the service type, if an explicit version is specified, then the package must also be explicitly specified.
- If no version is specified, the current version will be used
- If no package is specified, then
  - The same package as the referrer is preferred
  - Otherwise, a package referenced by the referrer is preferred
  - Otherwise, a Service Specification of the given name in any package is used
- By default, the new service will be created for the same tenant as the referring service.
  - To auto-create a root service for a different tenant, in the value separate tenantid and root service type by an ampersand: <tenantid> `&` <roottype>

In all other ways, auto-created root services behave like other auto-created services.

5.5.1.7 Auto-deletion of auto-created services

When an auto-created service is no longer "needed" - i.e. the reference that created it no longer exists, then the default behavior is that the now redundant service is automatically deleted. Any tear-down actions on the service will be executed (unless `$preprovisioned` is true).

The auto-deletion behavior is controlled by a field on the service objects in the service inventory named `"AutoDelete"`. This value will usually be set to true for auto-created services. However, it may be overridden by introducing a parameter named `autodelete` in the service-descriptor for the auto-created service.

An example use-case of explicitly setting `autodelete` to `false` is when a VM service creates a reference to a VM-image service. By setting `autodelete` to `false`, the image will be loaded upon first use, but will not be auto-deleted, and so will be staying active, even if the VM service is subsequently deleted and re-created.

Chapter 6 Integration with workflows and templates

The DDE Control Workflow evaluates the service descriptors, and then iterates over a collection of state-transition actions to process.

For each iteration, the DDE Control Workflow checks if the service descriptor specifies that HPE Service Provisioner (HPE SP) workflows should be invoked. The descriptors can define three workflows:

- Pre-workflow
- Workflow
- Post-workflow

Workflows invoked by the DDE Control Workflow must support certain input and output parameters as described in section Workflow parameters.

If no workflow name is explicitly stated in the descriptors (see section Actions and State-transition workflows), the DDE Control Workflow calls a standard HPE SP workflow named `DDE_Activate_Service`. See section Using Advanced Template Manager for details.

Note

For a high-level view of how DDE works within the HPE SP architecture, see section Introduction.

6.1 Workflow parameters

6.1.1.1 Input parameters

Any workflow called from the DDE Control Workflow must support the following input parameters:

entity: The name of the entity-type to operate on, for example, vm, monitor, vswitch, or license.

action: The action to carry out on the given entity type, for example, create, delete, powerup, activate, connect, or disconnect.

zystem_name: The name of the Zystem (as recorded in the HPE SP inventory) on which to carry out the action on the entity.

service: The new version of the service instance that the action is affecting. Can be accessed in templates and workflows.

current_service:

The version of the above service instance as currently stored in the inventory, that is, before the state-transition at hand. This allows the templates to refer to the old values of attributes.

tenant_id: The ID of the Tenant owning the given service instance.

meta_data: A map containing name-value pairs defined on the service instance. The values can be accessed in templates.

actions: A collection of actions used to control the work of the DDE Control Workflow.

6.1.1.2 Output parameters

Any workflow called from the DDE Control Workflow must support the following output parameters:

Result parameters:

The set of five parameters describing success or failure and possible error messages as the result of the workflow. See section Error handling for details.

uploaded_data:

A map containing data returned as output from HPE SP templates and workflows. All the name-value pairs returned in the `uploaded_data` parameter are automatically inserted into the meta-data of the service; overwriting any previous settings for the same names.

Note

This feature must be used with care, as uploaded values may overwrite parameters defined by the service descriptors. The system automatically propagates the effect of a changed parameter value to any services that depend on the value, but the change cannot cause a re-design of the service structure (so decomposition and references do not change).

service:

The service object that was originally passed as input to the workflow. If the workflow modified the service object, the modifications are used by the system.

Note

This feature must be used with care, as changes may break assumptions coming from service descriptors.

actions:

The collection of actions received as input. The workflow may manipulate the actions, for example, `add` actions that were not derived from the service descriptors.

Note

This feature must be used with care, as it directly alters the behavior of the DDE Control Workflow.

preactions:

The collection of preactions received as input. The workflow may manipulate the actions, for example, `add` actions that were not derived from the service descriptors.

Note

This feature must be used with care, as it directly alters the behavior of the DDE Control Workflow.

skipwf:

A string that controls the calls of the action workflow and post-workflow.

The string is interpreted differently when returned from a pre-workflow, an action workflow, or a post-workflow.

1. Pre-workflow: If the value is `all` or `true`, neither action workflow nor post-workflow is invoked. If the value is `main`, no action workflow is executed and only the post-workflow is executed (if defined).
2. Action workflow: If the value is `all`, `true`, `post`, or `postworkflow`, no post-workflow is executed.
3. Post workflow: Ignored, as there are no workflows to skip after the post-workflow.

6.2 Using Advanced Template Manager

Advanced Template Manager is a HPE SP component (see section Introduction) that manages an abstraction of actions on various equipment types.

Advanced Template Manager is accessed by invoking the `DDE_Activate_Service` workflow coming with HPE SP. This is the default workflow that the DDE Control Workflow uses.

Advanced Template Manager selects the template or workflow to be invoked for each combination of the following three input parameters:

entity:	The name of the entity-type to operate on, for example, vm, monitor, vswitch, or license.
action:	The action to carry out on the given entity type, for example, create, delete, powerup, activate, connect, or disconnect.
zystem_name:	This is looked up in the HPE SP resource repository to retrieve an associated **zystem** object.

Advanced Template Manager checks if the stated combination of entity and action is supported by the Zystem object. If not, it seeks a management Zystem that supports the operations, starting from the given Zystem and its location.

For each Zystem, the inventory records the following information:

- Vendor
- OSVersion
  - OperatingSystemGroup for the OSVersion
- ElementType
  - ElementTypeGroup of the ElementType

Figure Element types and OS types represented in the inventory shows the information as defined in the HPE SP inventory.

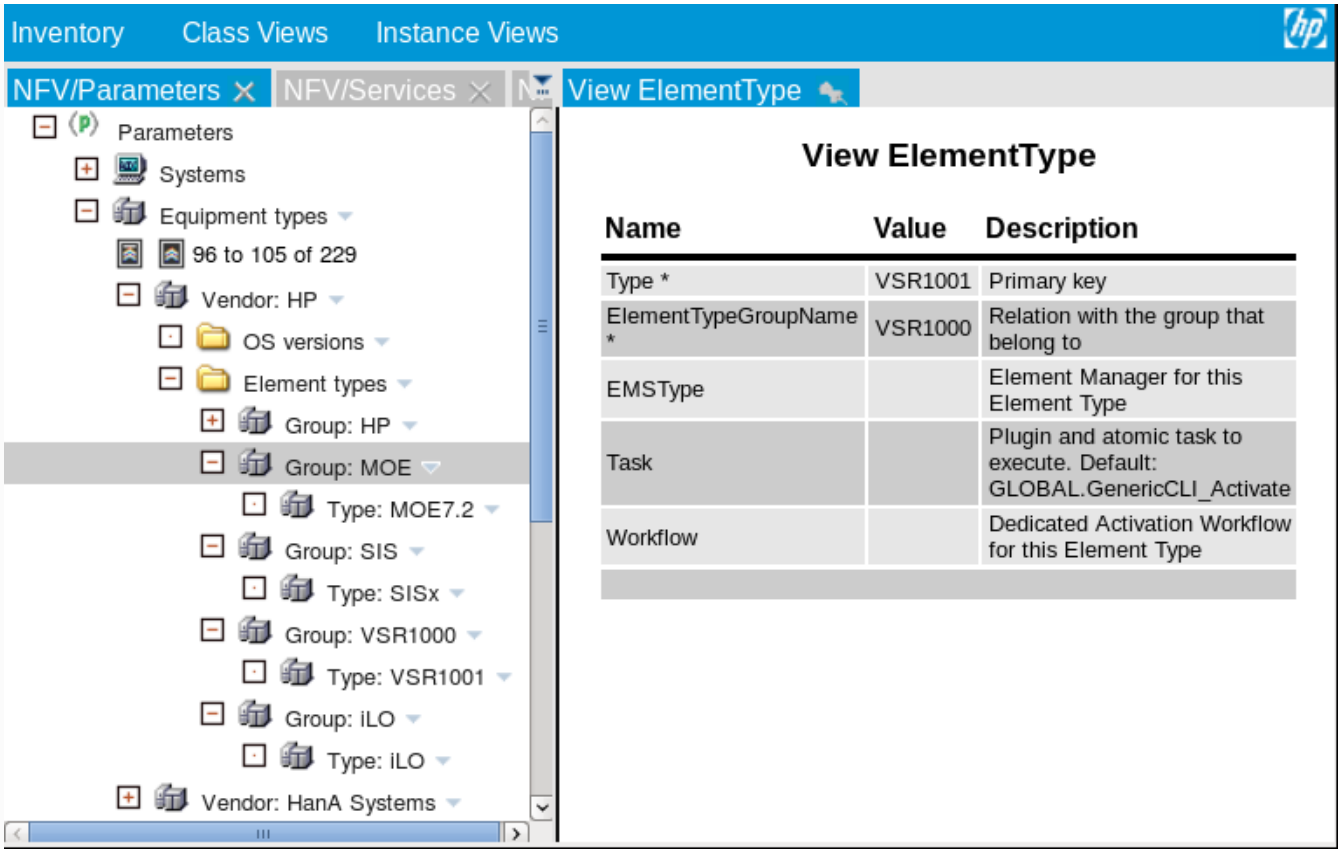

Figure 6–1 Element types and OS types represented in the inventory

Advanced Template Manager uses this information to locate a template or workflow that matches the right element type or OS version and supports the requested action on the indicated entity.

Note

Advanced Template Manager only finds templates and workflows of HPE Service Activator solutions that are registered in the DM_VERSION database table. The database table XP_VERSION is supported for backwards compatibility. The HPE Service Activator deployment manager populates the DM_VERSION table when a HPE Service Activator solution is deployed.

Please verify your solution is registered in the DM_VERSION table when your templates and/or workflows are not found by the Advanced Template Manager

6.2.1.1 Lookup of templates

The CLI template files are deployed under the following directory:

```
$ACTIVATOR_OPT/solutions/<SolutionName>/etc/template_files
```

Advanced Template Manager searches for HPE SP CLI template files in the following order:

```
.../<vendor>/<osgroup>/<osversion>-<element-type>-<entity>-<action>.xsl
.../<vendor>/<osgroup>/ANY-<element-type>-<entity>-<action>.xsl
.../<vendor>/<osgroup>/<osversion>-<element-type-group>-<entity>-<action>.xsl
.../<vendor>/<osgroup>/ANY-<element-type-group>-<entity>-<action>.xsl
.../<vendor>/<osgroup>/<osversion>-ANY-<entity>-<action>.xsl
.../<vendor>/<osgroup>/ANY-ANY-<entity>-<action>.xsl
```

In these names, ANY matches any actual value for this part.

6.2.1.2 Lookup of workflows

The workflow files are deployed under the following directory:

```
$ACTIVATOR_OPT/solutions/<SolutionName>/etc/workflows
```

Advanced Template Manager searches in the order described in section Lookup of templates for HPE SPworkflows whose names match the following naming standard:

```
<SolutionName>-<vendor>-<osgroup>-<osversion>-<element-type[-group]>-<entity>-
<action>
```

where ANY can occur in the same places as in XSL templates.

6.2.1.3 Lookup of protocol and credentials

Advanced Template Manager also identifies the connect-protocol and credentials for logging in to the Zystem.

- If only a CLI template is found, then the HPE SP CLI plugin (`GLOBAL.GenericCLI_Activate`) is invoked with that template.
- If only a workflow is found, then the workflow is started with the same parameters as were passed to the `DDE_Activate_Service` workflow, plus additional connect and credential parameters.
- If both a CLI template and a workflow are found, then the workflow is invoked as above, plus an additional "template" parameter with the contents of the CLI template.

The naming convention is designed to ensure that templates can be shared by elements in a specific OS group or of a specific type that supports the same command-set, but if some commands are only supported by specific OS versions or element types, then those templates will take precedence, if they exist.

6.2.1.4 Examples

CLI Templates:

```
.../HP/RHEL/ANY-VSR1000-vm-setupNAT.xsl
.../ANY-VSR1000-vm-removeNAT.xsl
```

Workflows:

```
OSPLUGIN-HP-HELION-ANY-ANY-server-create.xml
OSPLUGIN-HP-HELION-ANY-ANY-subnet-delete.xml
```

6.2.1.5 On-boarding of new element types or versions

On-boarding a new element type or version can be supported as follows:

1. Create templates for the new element type or version following the naming standard.
   - There must be templates corresponding to each entity-action supported by the solution.
2. Create the `ElementType` and `OSVersion` in the inventory.
   - Also add new vendor or groups as needed.
   - Note that HPE SP will auto-create these if they were not data-loaded up-front.

Chapter 7 Run-time aspects

7.1 Service requests

See separate API documentation.

7.2 Error handling

HPE Service Provisioner (HPE SP) uses the Common Internationalized Error Handler from NCC (see sectionIntroduction). The result of workflows and requests are returned as the following group of values:

Result:

One of the following codes:

- `SUCCESS`
- `TRANSIENT_ERROR`
- `RESOURCE_ERROR`
- `REQUEST_ERROR`
- `CONFIGURATION_ERROR`
- `INTERNAL_ERROR`

ErrorUID:

A unique identifier for each message. All `ErrorUIDs` will have the form:

- `HPSA.` <module>.<id>

The module is the name of a Java class file, a HPE SP JTP file or a HPE SP workflow. The id is chosen by the implementer. The NCC project environment generates standard Java property files for all errors. Integrators may create nationalized versions of these property files, to achieve nationalized error messages, following Java conventions.

OperatorMessage:

A message that is intended for presentation to the HPE SP system operator. This message is not nationalized, and so can be used in contexts were it is impractical to use property files.

CustomerMessage:

A message that is intended for presentation to the end-user (service requestor). This message is not nationalized, and so can be used in contexts were it is impractical to use property files.

ErrorParameters:

A comma-separated list of single-quoted parameter values to be used for expansion into localized strings from a .properties file. Single-quotes in parameter values are represented as: `&#39;`. Ampersand characters are represented as `&amp;`

All HPE SP Workflows should define 5 case-packet variables with the above names, and the REST interface automatically maps these to the corresponding return values for the caller.

Typically, the customer message will be meaningful when an error-condition is due to faulty customer request input or other causes that is under control of the customer. Since the operator will normally not need to take any actions on end-user input errors, the operator message for such conditions will be an unspecific standard message.

Conversely, the operator message will be meaningful for conditions that require operator intervention, whereas the corresponding customer message will be a standard message indicating that there was a problem and that the service provider is working on a resolution.

7.3 Inventory view of services and VNFs

Name

component, componentname, servicecomponent

Index

MetaData keys, and addressing child parameters from the request.

MetaData marshaller:

Chapter 8 Configuration Parameters

HPE Service Provisioner has support for configuration parameters that allow solutions to define string values that can be accessed from service descriptors as well as from action workflows. Configuration parameters are defined in files using YAML syntax and loaded into memory at start-up.

Note

Keep in mind that all configuration parameters are kept in memory; hence, an excessive number of configuration parameters will have an impact on memory consumption. In most practical use cases, however, the memory consumption will be minimal.

8.1 Configuration File Syntax

```
Spec      ::=  SA_config: LangVersion
                    [ CfgVersion ] [ CfgDescription ] CfgPackage
                    CfgSolution  CfgParameters
LangVersion   ::=  V001
CfgVersion    ::=  version: Version
CfgDescription ::=  description: Text
CfgPackage    ::=  package: PackageName
CfgSolution   ::=  solution: SolutionName
CfgParameters  ::=  parameters: ParamName -> ( Constant | ParamSpec )
ParamSpec     ::=  ParamValue [ ParamEncrypted ] [ ParamDesc ]
ParamValue    ::=  value: Constant
ParamEncrypted ::=  encrypted: true | false
ParamDesc     ::=  description: Text
```

8.1.1.1 Grouping of Configuration Parameters

Configuration parameters must always belong to a solution and to a configuration package. In this way it is possible for two different solutions to use the same configuration package name (as well as the same parameter names) without conflicting with each other.

8.1.1.2 Identification of Configuration Parameters

A configuration parameter is uniquely identified by four parts:

SolutionName: The name of the solution that the configuration package is part of

PackageName: The name of the package (within a solution) that the configuration parameter belongs to

Version: The version of the configuration package of the form ###.##

ParamName: The name of the parameter

8.1.1.3 Versioning

If the version of the configuration package is not specified, the version will be automatically set to 000.00. If there are multiple versions of the same configuration package (for the same solution) the one with the highest version number will be marked as “current”; only configuration parameters belonging to “current” configuration packages can be accessed from action workflows and descriptors.

8.1.1.4 Solution Name

The SolutionName is used to identify the name of the solution that the configuration package is part of. TheSolutionName must consist of only alpha-numeric characters and underscores, the first character must be a letter. The length of SolutionName must not exceed 8 characters.

Note

The solution name `GLOBAL` is reserved for and meant for globally accessible configuration parameters. No solution should specify configuration parameter packages belonging to the solution name `GLOBAL`.

8.1.1.5 Configuration Package Name

The PackageName is used to identify the group configuration parameters, typically belonging to the same functional entity. A solution can contain multiple configuration packages. The PackageName must consist of only alpha-numeric characters and underscores, the first character must be a letter. The length of PackageName must not exceed 64 characters.

8.1.1.6 Configuration Package Description

A configuration package may come with an optional human-readable description explaning the intended use of the configuration parameters belonging to the package.

8.1.1.7 Configuration Parameters

Each configuration package may contain any number of parameters. The name of configuration parameters must consist of only alpha-numeric characters and underscores, the first character must be a letter; the length must not exceed 64 characters.

For each configuration parameter a value must be specified. Parameter values can be defined in two ways: Either directly as string value or using the `value:` attribute. If using the latter format it is also possible to specify a parameter description as well as a boolean value to indicate whether or not the parameter value is encrypted.

If a configuration parameter contains a password it is highly recommended to encrypt it to reduce the risk of accidentally revealing password. Encrypted parameter values must be generated using HPE Service Activator’s `crypt` utility.

8.1.1.8 Configuration File Example

```
SA_config: V001
description: Configuration parameters to access database
package: db
solution: AC
parameters:
  host: serengeti.acmecorp.cc
  port: 1521
  name: XE
```

```
user: hal6000
passwd:
  value: t4q0rlkf294JsRdTXn7SJA==
  encrypted: true
  description: This password must be changed every three months
```

8.2 Loading Configuration Parameter Files

In order to find and load configuration parameter files they must have the `.yaml` extension and must be located in the following directory:

- `$SOLUTION_HOME/etc/config/parameters`

The directory may contain any number of configuration files.

In addition, the solution name in which the configuration parameters are stored must be added to the `solutions` parameter in the `DDE_config_module` that is configured in HPE Service Activator's `mwfm.xml` configuration file. The `solutions` parameter must be a list of solution names using commas as delimiters.

Note

The `solutions` parameter in the `DDE_config_module` must include the solution name `DDE`.

8.3 Accessing Configuration Parameters from Workflows

It is possible to access configuration parameter values from action workflows. A workflow node named `GetConfiguration` can be used to access a configuration parameter package and return all configuration parameters in the package to a case-packet variable as key-value pairs (a `HashMap`).

The workflow node `GetConfiguration` supports the following parameters:

`solution`: The name of the solution from which to read configuration parameters; if the `solution` parameter is not specified, it will automatically use the workflow's `SOLUTION_NAME`.

`package`: The name of the configuration package from which to read configuration parameters. Mandatory

`variable`: The name of the case-packet variable in which the `HashMap` is returned. Mandatory

Once the workflow node has been executed, all subsequent workflow nodes can access configuration parameters using standard HPE Service Activator syntax for getting values in a `Map`; for instance, `CFG.abc` or `CFG{"abc"}` looks up the key `abc` in the map `CFG` and returns the associated value. If the configuration parameter value is encrypted, the map will contain the descrypted values (i.e. the clear text values).

8.4 Accessing Configuration Parameters from Descriptors

Configuration parameters can be accessed from descriptors using the `@parameter` function.
The `@parameter` function accepts two or three arguments and has the following syntax:

```
@parameter([SolutionName,] PackageName, ParamName)
```

The function returns the value of the configuration parameter that is identified through the `SolutionName`, `PackageName`, and `ParamName`. If there are multiple versions, the latest version (the "current" version) is returned. If the configuration parameter value is encrypted, the function will return the descrypted value (i.e. the clear text).

8.5 Viewing Configuration Parameters

If is possible to view the configuration parameters from the Inventory UI. Open the inventory tree `SP/Parameters` and then expand the branch `Solution Configuration` to browse the configuration parameters by solution and configuration package name.

Chapter 9 Examples

9.1 Example .knd file content

```
DSD_kind_specification: V001
```

9.2 Example .svc file content

```
DSD_service_specification: V001
package: Tst
name: testService
version: 000.01
description: This describes an NFV Service
```

Chapter 10 Tips, Troubleshooting, and “Gotchas”

10.1 Using field selectors on objects - $this.x

If your Service Descriptor defines a parameter named x, then writing `$this.x` does not have the same meaning as just writing `$x`.

Similarly, using `$parent.x` should not be used in lieu of passing x explicitly as a binding from the parent to the child.

At run-time …

- … the expression `$this.x` applies the field-selector x to the current service as an object.
- … the expression `$x` specifically refers to the value of parameter x.

There are some significant differences:

- The expression `$this.x` will seek to invoke a java method named getX() on the service-object. Only if such a method does not exist, will the value of parameter x be returned instead.
- DDE synchronizes parameter and object-field values, but that only happens after the entire Design phase is complete. So `$this.x` may appear to return an outdated value for x even if the evaluation order is correct.
- Specifically for ServiceIds, DDE needs to maintain “shadow” ids for services during modification. The parameter evaluation system protects you from accidentally using such shadow-ids, but if you refer to `$this.serviceid` you invoke the java method returning the actual id, which may be a transient shadow-id that is unusable outside the internal mechanisms of DDE.
- The service object is really a HPE Service Activator java Inventory Bean object (see HPE Service Activatordocumentation). Such objects may have internal or hand-crafted getter-methods that are not necessarily documented. If your parameter name accidentally coincides with such a name, then `$this.x` will invoke that method and hence may give unexpected results.

Note

Using `$this.x` requires a detailed understanding of DDE and HPE Service Activator internals, and should be avoided if you are not absolutely sure, you understand the implications.

Note that any variable containing a service object or a `serviceid` will be interpreted this way if a field-selector is applied to it. That specifically pertains to `$parent`, `$reference` and `$referrer`. At the time of evaluating `parent` and `referrer`, the referenced objects are already designed, and so there should be no concerns about the evaluation order. For `reference`, the referenced object may not have been designed at all - it is likely to exist only in the CHECKED state. See section Values of the referred service available in the referring service descriptorfor a complete list of field-selectors that may be used on `$reference`.

10.2 Using field selectors on children - $child.x

If your Service Descriptor defines a child (service, component or resource), then writing `$child.x` will cause DDE to re-design the parent after the child is designed. This can cause a noticeable overhead in the fulfillment process, so only use this feature when absolutely necessary.

If the child parameter does not actually influence any workflow calls from the descriptors, consider using the `Outputs` section of the descriptor instead.

To ensure correct inheritance and compositionality of descriptors, it is strongly recommended that the parent service only refers to parameters that are defined in the Kind-descriptor for the child service. Although this is not presently enforced by DDE, such a check is planned to be introduced in the future.

10.3 Graphs

Note

- Graphs only work in Services. They do not work in kinds.
- Indexed bindings are currently not supported in graphs.

In order to avoid having to mention all the "negative" cases in the `Graphs` section of a specification, a binding that is mentioned in any of the graph parts will be set to null, if none of the matching parts set that binding. This can be counter-intuitive in some cases.

Defining graphs requires very careful design work. If the task seems to become too complex, consider defining helper parameters that isolate various parts of the conditions you need when defining the various sections of the graph.

10.4 Overly zealous re-creation of services

When evaluating a service modification, DDE compares all the parameters of the new service instance with the one that is already provisioned or active. If it finds any parameters have changed, it will deactivate the existing service and re-create the new one.

There are two cases to consider: Intended service modification and unintended modification due to changes in "administrative" parameters.

10.4.1.1 Service modification

Future versions of the descriptors may support direct actions to modify sets of parameters on the live service.

In the meantime, the way to avoid the teardown-recreate behavior is to isolate the modifiable set of attributes in a child component. The specification of the child component can then define separate workflow actions that handle the intended modification without affecting the live status of the parent service.

10.4.1.2 Debugging administrative parameters

Sometimes a Service Descriptor will introduce parameters that are for "administrative" use, but are known not to participate in the provisioning or activation of the service. If these parameters change, the service will be unnecessarily re-created, unless you specify `activate: false` on the parameter description.

It can, however, be hard to determine exactly which parameters actually cause the unnecessary change. To assist, there are debug-messages output to the Jboss `server.log` file indicating the exact set of parameters that were modified.

In a standard Linux installation of HPE Service Provisioner (HPE SP), the full path to the log-file is:

```
/opt/HP/jboss/standalone/log/server.log
```

Each log-line has an informative prefix. After the prefixes, the debug message will look like this:

```
COMPARED: Acme//ACMECPE/vSRsvc.nic[2]#shadow
Changed activation parameters: [network]
```

The change types that can be listed are:

```
Changed activation parameters
Changed administrative parameters
Changed children
```

Only a change of `activation parameters` will cause the service to be re-created. So if, in the example above, you are certain that the network parameter is not used by the provisioning or activation workflows, then you can write `activate: false` on the parameter description. That will explicitly tell DDE that the parameter should not cause service recreation.

Note that putting `activate: false` does not influence newly created services or explicit service tear-down.

Also note, that all parameters are considered administrative if the service has defined no provisioning/activation actions at all. If a Service Descriptor is later modified by introducing an activation action, this may cause significantly different behavior of the solution.

10.5 Choosing between default, prefer and value parameter expressions

When defining a parameter value expression, there are three choices: `default` or `prefer` instead of `value`. This choice can be confusing, and the wrong choice can lead to errors. Here is a brief guideline:

default: If a value is provided from the parent service or in the service-request, then that value will be used. Also, if the value was was already computed (and was not null) in a previous design-phase, then that previous value will prevail. In other words: Once evaluated, the value will never change again, unless set from "the outside".

prefer: The preferred expression will be evaluated during every (re-)design of the service. Only if the expression result becomes null, will the previous value, or a value from a parent binding or a service request be used.

value: The parameter will be unconditionally assigned the value of the expression, even if that value is null. If the parameter is not marked as `optional`, then this will cause an error.

Also note that `value` parameters defined in a Specification Kind cannot be overridden by an including Service or Kind.

Expressions defined as `default` or `prefer` can be overridden by an including Service or Kind.

10.6 Mandatory parameters become null during teardown

This happens when parameters in other services are dependent on the service being deleted. The typical case is when parameters depend attributes of `$referrer`. If the referrer service (see section References between services) is being deleted or modified, then these parameters may be re-evaluated, and since the referrer service is no longer there, the values become null.

To avoid this problem, define the value expression of such parameters using `default` or `prefer` instead of `value`. As explained in section Choosing between default, prefer and value parameter expressions, using `default` the expression will only be evaluated once during the initial design of the referenced service, and so the value will not change when the referrer is no longer present. If `prefer` is used, the system will re-evaluate the referenced service whenever the referrer is modified, but if the value becomes null, the last non-null value will be preserved.

10.7 Conversion between integer and boolean values

In contexts where a boolean is required, but an integer is provided (`@and`, ...), the system interprets `0` as true, and any other value as false.

In contexts where an integer is required, but a boolean is provided (`@sum`, ...), the system interprets `false` as 0, and true as 1.

In rare cases, this can lead to problems, because conversion from integer to boolean and back will not yield the original value.

The reason for interpreting `0` as true is that this is the convention in many shells and systems where a non-zero number represents an error code, and `0` represents a successful result. This convention is used in the Advanced Template Manager component of HPE SP. The primary use-case for policies applied to numbers is to test if a response from external systems was successful, and so the most useful interpretation is that `0` is true and other values as false.

A use-case of booleans interpreted as integers would be to sum the number of true values. So in this context, it is most useful if policies interpret `true` as `1` and `false`, `null`, etc. as `0`.

10.8 Why is the value of a parameter value not changing

When a parameter, `p`, defines its value by means of an expression like `@f($a)`, and the value of the parameter, `$a`, used in the expression changes, you expect the value of `$p` to change accordingly.

But sometimes it may look like there is a bug in the value propagation, because once set, the value of `$p` will no longer change, even if `$a` changes.

Often, the cause of this may be that for other reasons, you declared the expression as `default` instead of declaring it as a `value`. The meaning of `default` is that if the parameter value would otherwise be `null`, then use the given expression to provide a value.

But once `$p` was set, its value is no longer null, and so the default expression is no longer evaluated. So henceforth the parameter will ignore any changes in the values that provided the default value.

Now in some cases, you actually want `$p` to be settable from a request, and that may be the reason for writing `default` instead of `value`. To achieve this effect you need to use an auxiliary parameter, `$p1`, to handle the distinction between an externally defined parameter, `$p`, and a computed one. What you would want to write would be something like this:

```
parameters:
  p:
    type: string
    optional: true
  p1:
    type: string
```

```
    value: "@if(@defined($p), $p, @f($a))"
```

With this declaration, you can set the parameter, `p`, externally, and if you do, then that value will define the value of `p1`. But if `p` was not provided, then the expression, `@f($a)`, will unconditionally define the value of `p1`.

Declaring `p1` in this way will also make it quite clear to the reader (as well as to the system), that the value of `p` overrides the value of `@f($a)`.

10.9 Why are service not deleted in the reverse order of their creation

Usually, services will in fact be deleted in the reverse order, but this is not always the case. There can be several reasons for this. The order of deletion is determined by the service dependencies as defined in sectionDependencies between Services, and for any independent services, the order is not predictable. Also, note that `reference` and `auxreference` values may change over time, and so dependencies may not be the same at deletion-time as they were at creation time.

One thing that can be confusing is the direction of the "depends on" relationship when a reference has auto-created a service (as described in section References between services). The auto-created service is "created" by its referrer and is "deleted" when no longer referenced, so it may appear that the life-cycle of the referenced service "depends on" its referrer. This interpretation is not correct. The referenced service may already exist and may also continue to exist after the referrer is deleted. So the dependency is from the referrer to the referenced service, not vice versa.

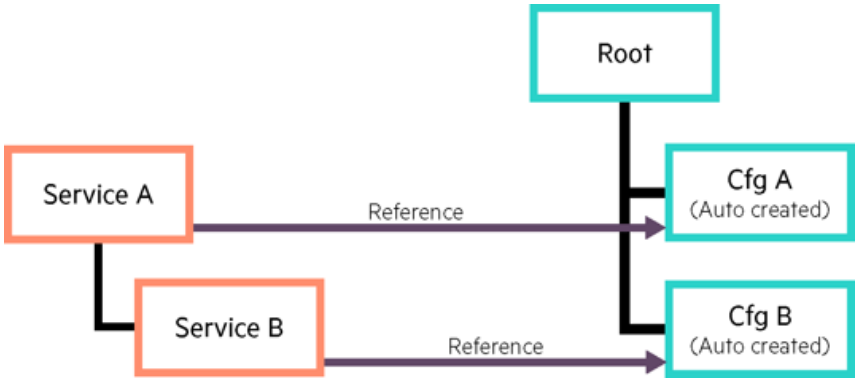

Figure 10–1 Auto-created configurations, unrelated

Figure Auto-created configurations, unrelated shows a Service A, and child-service B, each causing the auto-creation of some configuration. The services will be initially created in the following order: `A, CfgA, B, CfgB`.

The engine will, however, ensure that `CfgA` is activated before `A`, because `A` depends on `CfgA`. So the activation sequence will actually be: `CfgA, A, CfgB, B`.

Now, looking at the deletion sequence, you might expect the opposite order: `B, CfgB, A, CfgA`. This may turn out to be the order, but since there are no dependencies between `CfgA` and `CfgB`, a number other sequences are possible and correct. For example, the deletion sequence may well become: `B, A, CfgA, CfgB`.

If the two configurations are really quite independent of each other, then this may be acceptable, but if you really need `CfgB` to be deleted before `CfgA` because an error would otherwise ensue, then your model must reflect the dependency.

There are several ways that you could modify the model to reflect a dependency of `CfgB` on `CfgA`.

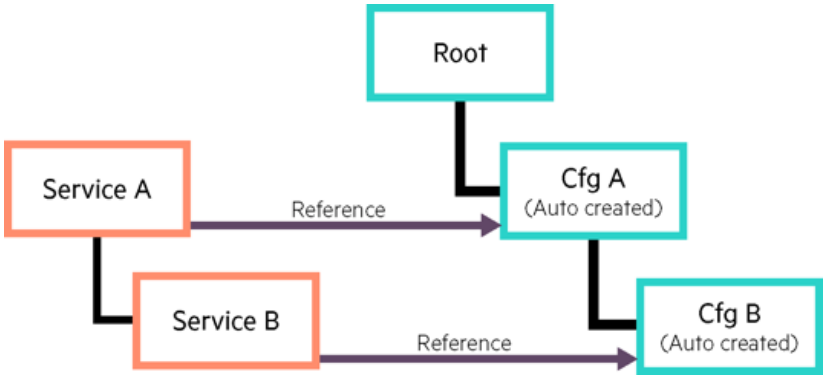

Figure 10–2 Auto-created configurations, dependent

Figure Auto-created configurations, dependent shows a solution that is likely to be useful in most cases. With this model, the engine will ensure that `CfgB` is always deleted before `CfgA`. With this model, the only two possible deletion sequences would become:

```
B, CfgB, A, CfgA
```

or

```
B, A, CfgB, CfgA
```

10.10 Automatic propagation of location and locationname

The `location` and `locationname` parameters are propagated automatically from parent to child services. See section Child parameter bindings for details.

Chapter 11 Deviations from HOT

The DSD Service Descriptor format deviates from OpenStack Heat Orchestration Templates (HOT) in a number of ways:

- A resource can have nested sub-resources. The notable case for this is the NICs on a server resource.
- The sub-components of a HOT templates are called resources. We extend this as follows:

  1. Keyword `Services`: This section lists "customer-facing" sub-services that must be individually managed from the northbound service API of HPE Service Provisioner (HPE SP). Setting a minimum number other than zero is forbidden, as services cannot be auto-created by automatic decomposition.
  2. Keyword `Components`: This section lists internal "resource-facing" sub-services that are managed exclusively by the service logic.
  3. Keyword `Resources`: This section lists sub-components that are bound to atomic resources in the Virtual Infrastructure Management systems - i.e. Networks and VMs. Note that once created, resource attributes can be accessed at run-time to get parameters for other components. So if a resource is a DHCP server, then it may be used to create IP addresses for passing as parameters to services and components.
- Parameter constraints are currently not supported, except for validation and range policies
- An `implements` section is added for describing a list of kinds of functionality that the service component may provide. The services can be instantiated by all kinds that provide all non-optional parameters of the service.
- Innate HOT Functions, `get_param`, `get_attr`, and `get_resource` are supported, but it is recommended to use the following alternative notations:

  1. Instead of `get_param p`, use `$p`.
  2. Instead of `get_resource id`, use `^id`.
  3. Instead of `get\_attr [id1, id2, a]`, use `$id1.id2.a`.
- For optional and scale-out resources/sub-services, the individual instances can be referred to by the component name:

- name[0]
- name[1]
- ...
- name[*n*]

  Indexing is in the order each of the resources/subservices were created. The list of all the currently scaled out resources/sub-services is referred to as:

  id1

  The current number of resources/subservices can be referenced as:

  @size(id1)
- The current service, as an object can be referred with the parameter `$this`.
- If a parent service exists (optional), it can always be referenced as if it were a parameter `$parent`.
- DSD Service Descriptors should not be strongly tied to OpenStack, so node types like `OS::Nova::Server` should not be used.
- The HOT parameter types `number`, `json` and `comma_delimited_list` are not supported.

- A number of special types are supported by DSD Service Descriptors to facilitate mapping of the same objects from service instance requests
- DSD Service Descriptors support “kind” instead of “type” for resources
- HOT today only supports scale up/down (i.e add/remove vCPU), but not scale out/in. There exists a proposed HOT extension for “scaling groups” , but it is work in progress. The proposal does not extend the normal HOTformat well. Instead we assume that all resources are potentially grouped, but the default is no scaling, amounting to creation of a single instance.
- HOT does not have anything comparable to links and network graphs. These are added in DSD Service Descriptors.

Chapter 12 Using the Asynchronous REST NBI

This document describes how to use the asynchronous REST API of HPE Service Provisioner (HPE SP) through examples. The reader of this document is expected to have a deep knowledge of the synchronous REST API.

12.1 Important

1. Transaction Identifiers
   - The REST API now stores incoming requests in the so-called “Service Order Registry” using the transaction id as a key. This is used to ensure that if a request is received with a transaction that matches a previously received transaction id, then the old result will be sent back to the caller (i.e. the new request will *not* trigger an operation in HPE SP).
2. `http_sender_module` configuration
   - The `http_sender_module` that needs to be enabled in the HPE SP Workflow Manager’s configuration file `/etc/opt/OV/ServiceActivator/config/mwfm.xml` should set the `fault_tolerant` parameter to `false` to prevent a potential “head-of-line” blocking.

Example of `http_sender_module` configuration:

```
<Module>
  <Name>http_sender_module</Name>
  <Class-Name>com.hp.ov.activator.mwfm.engine.module.HTTPSenderModule</Class-Name>
  <Param name="url" value="http://127.0.0.1:9080/helloworld/HelloWorld"/>
  <Param name="connect_timeout" value="10000"/>
  <Param name="read_timeout" value="10000"/>
  <Param name="fault_tolerant" value="false"/>
  <Param name="read_message_from_db" value="true"/>
</Module>
```

12.2 Terminology

Term	Meaning
Request	This is the message sent from an external system to HPE SP
Acknowledge	This is the “immediate” message sent back from HPE SP to the external system to indicate that the request was received
Response	This is the “final” message sent from HPE SP to the external system containing the full result

12.3 Callback REST API (only for testing purposes)

For all HPE SP asynchronous REST API call, there is a corresponding “callback” REST API call that may be used to receive asynchronous responses. These “callback” REST API calls can be used for testing purposes; however,when used in production the “callback” REST API calls will be disabled.

The following callback REST API calls exist:

Operation	URL	Method
Create Service Callback	/callback/{tenant_id}/services	POST
Delete Service Callback	/callback/{tenant_id}/services/{serviceid}	DELETE
Modify Service Callback	/callback/{tenant_id}/services/{serviceid}	PUT
Recreate Service Callback	/callback/{tenant_id}/services/{serviceid}/recreate	PUT

The callback REST API calls do not perform any actions in HPE SP. Their only function is to receive the response and save it in a file subject to manual or automatic inspection. Such files will be stored in the following directory:
* `/var/opt/OV/ServiceActivator/tmp/async_test`

Note

Cleaning up files in this directory is a manual task

12.4 Example: Create Service

The following is an example of a “Create Service” request. Pay special attention to the HTTP header named `Callback`. This header tells HPE SP how the asynchronous response shall be delivered. In this example, the asynchronous response shall be delivered to the URL `http://localhost:8081/v2/callback/api_tenant/services` using the HTTP method `POST`; the secret token is `verySecret`.

12.4.1.1 Request

HTTP method and URL:

```
POST http://127.0.0.1:8081/v2/async/api_tenant/services?simulate=true
```

HTTP headers:

```
Callback: <http://localhost:8081/v2/callback/api_tenant/services>;
          method="POST"; secret="verySecret"
Accept: application/json
X-Auth-Token: 5f4db455-698f-4471-8ad1-d19149db4792
Content-Type: application/json
```

HTTP body:

```
{
  "service": {
    "metadata": {
      "flavor": "small",
      "OCMP.username": "hello",
      "OCMP.password": "world",
      "val2": "VAL@",
      "val1": "VAL!",
      "OCMP.ip": "192.168.0.7",
      "OCMP.name": "Fjodor",
      "OCMP.nics": "1"
    },
    "service": {
      "action": "create",
      "servicetype": "Tst::testService",
      "version": "000.01",
```

```
      "state": "Ready",
      "serviceid": "1352"
    },
    "transactionid": "trId3_create_1441053203804",
    "orderid": "333111",
  }
}
```

12.4.1.2 Acknowledge

HTTP response code: `200 OK`

HTTP headers:

```
Content-Type: application/json
```

HTTP body:

```
{
  "service": {
    "transactionid": "trId3_create_1441053203804",
  }
}
```

12.4.1.3 Response

Notice that the `Secret` HTTP header contains the secret that was delivered through the `Callback` header in the original request.

HTTP method and URL:

```
POST http://127.0.0.1:8081/v2/callback/api_tenant/services
```

HTTP headers:

```
Content-Type:application/json
Accept:application/json
Secret:verySecret
X-Timestamp:1441053208683
```

HTTP body:

```
{
  "service": {
    "serviceresponse": [
      {
        "transactionid": "trId3_create_1441053203804",
        "orderid": "333111",
        "serviceid": "api_tenant//1352",
        "tenantid": "api_tenant",
        "servicename": "1352",
        "uuid": "2bf10f8e-d97a-32b2-9f4c-8245e3b4020e",
        "uri": "http://localhost:8081/v2/api_tenant/services/
               2bf10f8e-d97a-32b2-9f4c-8245e3b4020e",
        "committeduri": "http://localhost:8081/v2/api_tenant/services/committed/
                        2bf10f8e-d97a-32b2-9f4c-8245e3b4020e",
        "shadowuri": "http://localhost:8081/v2/api_tenant/services/shadow/
                     2bf10f8e-d97a-32b2-9f4c-8245e3b4020e",
        "result": "SUCCESS",
        "state": "ACTIVE",
        "childtype": "SERVICE",
        "servicetype": "Tst::testService",
        "version": "000.01",
        "metadata": {
```

```
            "flavor": "small"
          },
          "catalog": null,
          "parentsvc": null,
          "referencesvc": null,
          "auxreferencesvc": null,
          "servicetypeversion": null,
          "parentserviceid": null,
          "reference": null,
          "auxreference": null
        }
      ],
      "transactionid": "trId3_create_1441053203804",
      "orderid": "333111",
      "result": "SUCCESS"
    }
}
```