***A Service Model***

***for Cloud***

***Environments***

**Version 0.3 (dev)**

**Uwe Krüger**

**SAP SE**

History

* 1. first version
  2. description examples
  3. fix structure specification

Abstract

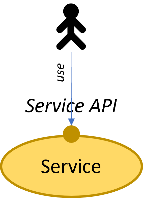
*Simple service dependency modellings applicable for instance-based on-premise installations are not sufficient for cloud-based environments. An appropriate model must combine the requirements for installing and updating meshes of interconnected services (for example for sovereign clouds) and the new kind of services able to provide service instances on demand in a cloud landscape. This paper describes the minimal requirements and layout for such a description model, which also builds the connection to an installation framework to provide the bases for a completely automated lifecycle management for cloud landscapes. By re-using the identity scheme and artifact description model provided by the Open Component Model, it also provides the bridge to all the artifacts required to deliver and install software in fenced environments.*

# Terminology and Required Elements

This section works out the elements for such a service description model and defines their names and meaning.

## Services

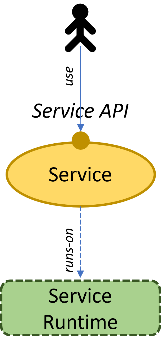
Before we can consider a service description model, it must be clarified and defined what exactly is meant by a service, and what this means in a cloud-based environment.

*A* ***service*** *in its most general form is just an entity, which offers some API to its consumers.*

Example: This spans a wide range of implementations. It might be a software module like a Go package offering some API used as part of an application program, it can be some services offering a REST API to be used over a public or private network like an OCI registry, or it is a runtime environment like Kubernetes or Cloud Foundry used to run other services.

A service typically lives on a service runtime.

*A* ***service runtime*** *is a service used to embed other services and their API.*



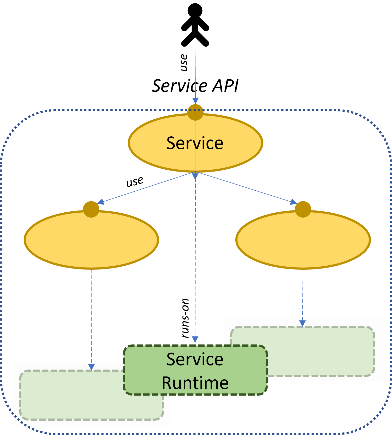
Examples are shown in the following table:

|  |  |
| --- | --- |
| Service | Runtime Service |
| VM | Hypervisor |
| VM | VPC |
| Process | VM or Machine |
| K8S Cluster | VPC |
| ABAP Application | ABAP-System |
| DB Schema | Database |
| Module/Package | Executable |

A database scheme is a service in the sense of the definition of a service used here, because it is an entity with dedicated users, permissions and maintained elements (like tables) separated from elements in other schemes. It has an own API and API endpoint. A scheme has formally an own endpoint although it is typically reachable over a common API of the database. This is comparable to objects and classes in an object-oriented environment. Technically the methods and their implementations are bound to the classes. But formally they belong to the objects, the access point is not the class (like for abstract data types), but the object identity.

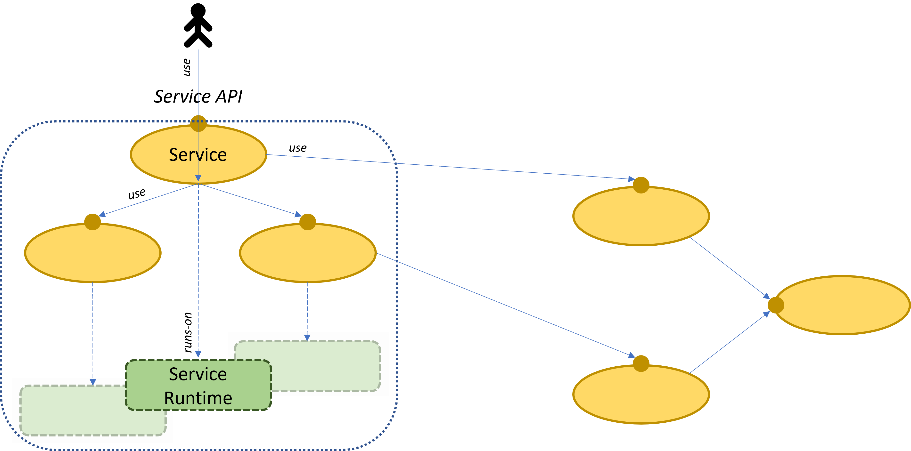
## Service Dependencies

If we take a closer look at such a service, it can be decomposed into multiple other (micro) services. In general, it may consist of a set of other services, which are integral part of it. Those services are typically part of its installation and are required to run the service. These dependencies are called implementation dependencies.

**Implementation dependencies** are dependencies to other services which describe an internal decomposition of the service into smaller services.

Besides those implementation dependencies a service may be orchestrated in an external set of interconnected services to finally provide its functionality towards its users. Those eservices are required to fulfill API requests but are not part of its decomposition. Typically, this set of services is building some kind of business context. The business context binds together a set of services, which share a common interpretation context for elements like business partners, customers, or cost centers.

The particular dependencies are called orchestration dependencies.

*****Orchestration dependencies*** *are dependencies of a service to separately maintained services used to embed the service into some kind of business context.*

Example: A financial accounting service may internally use a database to store its data. This is an implementation dependency. But additionally, it requires a master data management service to get information about customers, products and orders. This kind of dependency is an orchestration dependency. It does not belong to the implementation and specific functionality; it just provides information about the context the service is used.

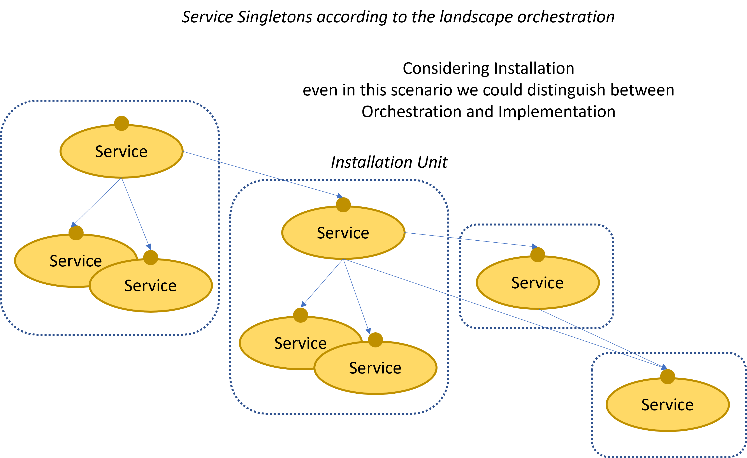
Summarizing this, a service may have two kinds of dependencies:

* Implementation Dependencies
* Orchestration Dependencies

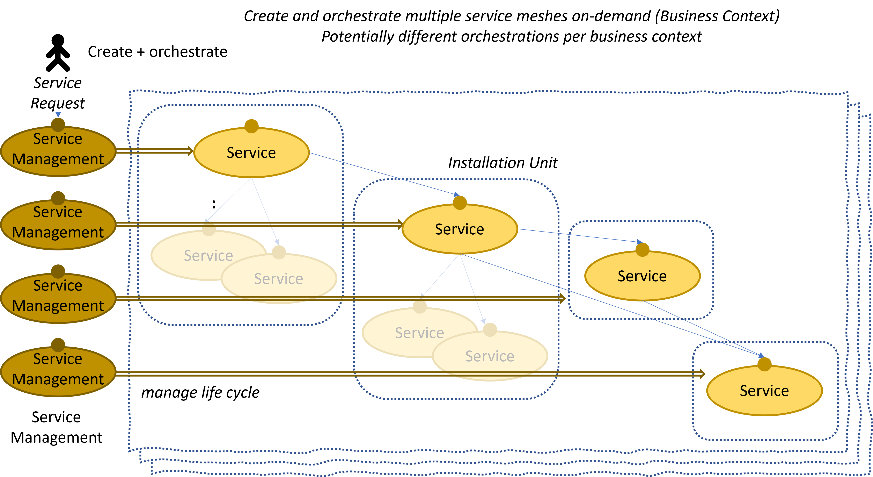
A formal dependency must be resolved to a service endpoint of a dedicated service instance of the described service kind.

## On-Premise vs Cloud-based Environments

A typical standard on-premise landscape is just a net of interconnected top-level services building the business or landscape context according to the orchestration dependencies of the involved services.

Every service may be decomposed into a set of subsequent other services according to the implementation dependencies of these services.

A typical service description model for such a realm just describes the orchestration dependencies of all involved services. Additionally, implementation dependencies are used for all those services as long as their lifecycles are not covered by the installation procedure of the particular service.



For a cloud-based environment this is not sufficient, because we must handle a completely new kind of service. While in on-premise scenarios dedicated service instances are explicitly installed per landscape context, cloud means an arbitrary number of business contexts are created on-demand by customers of the cloud landscape. This requires a completely new kind of service: Services managing the lifecycle of other (managed) services capable to create and orchestrate such service instances on demand.

*A* ***service provider service*** *(or short just* ***service provider****) is a service offering a lifecycle API for service instances of one or more kinds*.

*A* ***managed service instance*** *(or short* ***managed service****) is a service (instance) whose lifecycle is managed by a service provider.*

Compared to a typical on-premise landscape the service providers are able to create the service instances finally used to orchestrate a dedicated business context.

This requires three different features:

* A service provider is used to manage service instances for a dedicated consumer. Therefore, it requires consumer specific tenants or accounts.
* The service provider API can again be decomposed into two different APIs:
  + Tenant/account creation
  + Lifecycle API as part of a particular tenant

If sub accounts are supported (preferred), the account is a further resource manageable by resources in an account.

* To support the orchestration of managed services in a business context the configuration of service requests must include the dependencies used to embed a service instance into its interconnected business context. The consumer placing an order for an instance must include the intended orchestration dependencies (used to connect the new instance with other services in the business context) in the specification of the order.

Dependencies to a service provider are used to maintain digital twins for managed service instances. Therefore, such a dependency must always be resolved to an account specific formal endpoint, when dependencies of a service are resolved. In terms of such a service model, a service management therefore provides at least two kinds of service instances: a service instance representing an account in the server provider and the payload services instances. This behavior is used as recursion base for working with a service provider.

The cloud environment itself is therefore a landscape (context) of service providers with an explicit account management for its customers. This account management should be reflected in the various participating service providers. To achieve a good encapsulation of the providers, the providers should use their own account/tenant management for their internal implementation, which is then mapped to the central account management provided by the cloud landscape. The customer facing account is then a set of related internal provider specific accounts. This assures that providers are able to provide their functionality without a tight coupling with the central cloud infrastructure.

## Installation

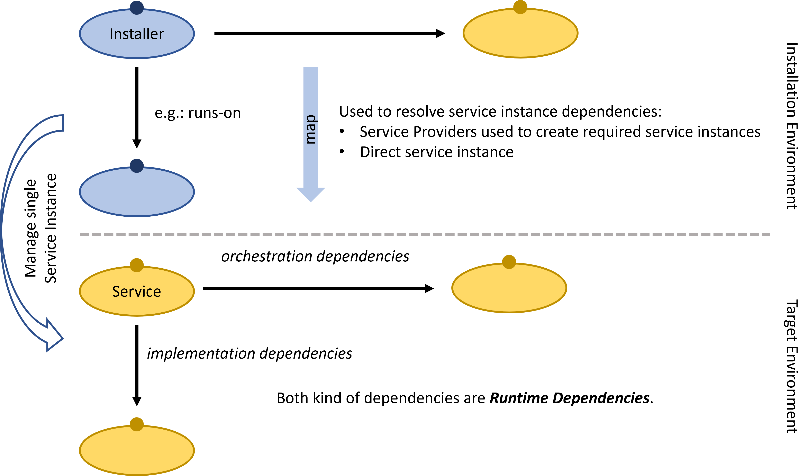
Every service must come into life before it can be orchestrated and used by its consumer. In on-premise scenarios this is typically handed by an installer, which is executed by the landscape administrator.

It is configured with the orchestration dependencies required by the installed service instance. Additionally, some implementation dependencies of the service instance to be installed must be given, as long as they are not yet covered by the installer itself (for example the chosen runtime service instance). They are covered by the installer, if it maintains those instances as part of the installation process or if they are implicitly given by the runtime of the installer.

In a cloud-based scenario, this must be completely automated. The task is handled by the service provider behind its lifecycle management API. This way any number and kind of customer business landscape can be created on-demand.

Nevertheless, instead of installing the service instances participating in an on-premise business context now the service providers used to manage service instances for an arbitrary number of business contexts must be installed. The services building the cloud landscape (the orchestration of service providers) must be installed to create a new cloud instance. There will typically never be a single instance of some kind of public cloud. Instead, there will be several sovereign cloud instances to fulfill legal and other requirements.

So, we still need explicit installers and an automation to setup and maintain such landscapes. All this must be describable by the intended service description model.

As we have seen, even an installer might have service dependencies. In on-premise scenarios they are used to determine the service runtime and to orchestrate the intended business context, in cloud scenarios they are used to connect the various service managements.

For example, the Steampunk Service Provider requires the Hana Service Provider to provision Hana Databases required by the managed ABAP service instances. And both use the Gardener Service provider to provision Kubernetes runtimes used to install their service instances. Even more, the Gardener service Provider is also used by the installation of both services to provision the runtimes required to run the service managements.

According to the service definition given, the installer is again some kind of special service. It has an API used by a landscape management tool to issue lifecycle operations for a dedicated service provider (instance) and has dependencies to other services (here in most of the cases dependencies to service providers).

For managed services the installation part is hidden by the service provider, but nevertheless the installation process features the same kinds of dependencies. For services (mainly service providers) there must be an explicit installer which can be executed by a landscape setup tool. It should act like a reconciler (K8S) to be able to handle drift and execute installation as well and updates.

The landscape setup tool is basically a managed service provider for arbitrary service providers, which will be possible by using standardized installation/lifecycle APIs for the installers.

# Service Description Model

An appropriate service description model must be based on the definitions, findings and different kinds of services identified in the first section.

This leads to the following requirements such a model must meet:

* The service model should describe services of different types together with their features. There are specific features for the particular types, but common features, also. One such common feature are the dependencies to other services.
* Service descriptions should be development objects, because they should be used as a technical base to control the setup/lifecycle of a complete landscape consisting of dozens or even hundreds of service providers.
* As development objects, descriptions might evolve over time, for example the dependencies to other services might change. This means, that the descriptions must be versioned.
* A service will be described in a declarative manner in form of a document stored as resource in an OCM component version to be deliverable as artifact together with the software artifacts required to install and run the service providers. The structure of this document is defined as part of the definition of the *service description model*.
* Such a model must be capable to describe dependencies among services. Therefore, every service must be uniquely identifiable.
* The descriptions must relate to appropriate installers or installation artifacts delivered together with the other software artifacts required to run the software.

The service description model still just describes service kinds, which can be instantiated in a concrete service landscape. It describes the dependencies, which must be resolved in a concrete landscape. The service model never describes dedicated service instances in a service landscape and their meshing. It just describes all the information required to instantiate a mesh of services in a dedicated landscape and does not make any assumption whether at all or how many instances of a service are available in such a landscape.

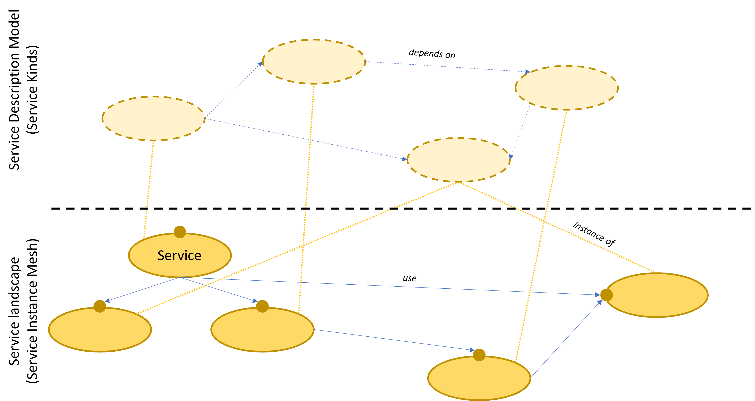
This model description just contains fields required by an automation to manage service landscapes. It does not contain informational fields which have no technical relevance, like Organization or Responsible persons. Those fields can just be added in a final specification.

## Service, Kinds and Instances

When talking about services this typically means to talk about instances of a dedicated service kind, but not always. If required to understand the actual meaning in a particular context we have to explicitly distinguish between kind and instance (class and object in OO environments).

*A* ***service kind*** *describes a dedicated implementation by all its deployment artifacts and the API specification.*

*A* ***service instance*** *is a particular installation of a service kind.*



When describing a service in a service description model, always a service kind is meant, the model never describes dedicated instances. This has also a consequence for the meaning of service dependencies. A dependency on the level of service kinds means, that an instance of the dependent service kind is required for a service instance of the kind featuring the dependency.

This is different from services in particular service landscapes. Here we talk about service instances and their wiring. In a service landscape there may be multiple instances of the same service kind. The wiring must meet the rules described for the service kind dependencies but might be resolved to different service instances depending on the context.

In the scope of the service description model by default the term service is used as simplified substitution for service kind.

## Service Descriptions and the Open Component Model

One important requirement for a service description model is the link between the service descriptions and the resources finally required to install a service. This means a relation to the software description model. The Open Component Model is such a model, which will be used to describe and deliver all kinds of software artifacts. Therefore, the service description model should be bound to this model, but without a tight dependency.

There are only two contact points between both models:

* The service description is a resource like any other artifact required to deploy a software product. Therefore, the service description should be handled as artifact like any other software artifact, but with a dedicated type describing the purpose of this kind of artifact.
* It must be possible to find the description for a service based on its identity. This should be possible even if there is no kind of service registry where all the service descriptions are gathered. Because the service description is stored as resource in an OCM component version it is close at hand to somehow use the identity of the component used to carry the service description to identify the service.

### Delivery Artifacts

Service descriptions should be delivered as regular artifact together with all the other software artifact required to run or install a service. Therefore, the service description is described as regular artifact with the resource type ora.org/SoftwareDescriptionModel in a component version.

A component version may contain any number of such artifacts. When analyzing a component version to extract its service description, all such artifacts will be combined to a single component version specific description.

The description should provide the link to artifacts required to install a service. OCM already supports such links by *relative resource references*. This is a reference, which can be evaluated relative to the containing component version to describe any resource in the component version graph spanned by this component version. This link can be replaced by using an installer to manage the lifecycle of a service. Then a concrete installer has those links to concrete artifacts. Different environments might require different installers and different software artifacts.

### Service Identities

The Open Component Model provides a globally unique identity scheme, consisting of a hierarchical component name, a version of a component and local identities of resources described by such a component version. Service descriptions are stored as resources in an OCM component version. Therefore, it is obvious to bind the service identity scheme to the open component model identity scheme. This allows finding the description of a service directly from an OCM repository just by knowing its identity.

To be handy a service description format should be able to describe multiple services in one document. This means that a part of the final service identity should be a name stored in the service description artifact.

Putting all this together leads to the following service identity structure:

* Component name (with leading realm identity consisting of at least a DNS name prefix.
* A service name as used in the service descriptions.
* The version of the service description is described by the version of the component containing the description.

An example for a service identity will be

acme.org/sampleservice/servicemanagement

It is exposed in the same component as

acme.org/sampleservice/instance

The component containing the description resources is then

acme.org/sampleservice

When accessing the description for a component version, all descriptions are merged and the content if transformed into an absolute identity scheme. The stored resource contains partial names local to the component version: service names and relative resource references. Those names are transformed into an absolute representation incorporating the component’s name and version and the described component version references.

The result is a machine-readable document, which describes a set of services, and which can be interpreted with the knowledge of the (component version) origin of the contained sub-descriptions. This way documents from any number of component versions can be merged (or stored in a service registry).

### Versions

Version names must conform to the semantic versioning model.

When defining a service implementation kind, the version described the complete implementation version, including pre version and build extensions. But versions are also used to describe dependencies.

Here, the usage of complete implementation versions would require providing new description versions up the complete usage hierarchy. But this provides no benefits, because the usage is typically based on the API version of the used service. This should always be expressed by the basic semantic version consisting of major, minor and patch version. Therefore, usage constraints should only refer this part of a version name.

## Service Description Document

Services are described as part of a service description document. The standard format for such a document is YAML.

Such a document has a format version. The actual version is v1. It has the following fields:

* type (*string*): The format version and document type
* services (*[]ServiceDescription*) A list of service descriptions.

There are two flavors for such a document:

* serviceModelDescription/v1: The absolute form of service descriptions. Here absolute service and OCM resource identities are used.
* relativeServiceModelDescription/v1: The relative form of service descriptions used to describe services as resource document in an OCM component version. Described services are denoted just by their relative names in the context of the containing component version, and referenced resources (used to refer to delivery artifacts) are described by relative resource references, which have to be interpreted in the context of the actual component version (see 2.2).

Besides the representation of identities, both flavors are identical for the same format version (here v1).

## Service Description

A service description describes the features and attributes of a dedicated kind of service. It does not describe a concrete service instance in a particular service landscape. The service is identified by its service identity (2.2.2).

It has at least the following fields:

* service (*service identity/name*): The identity of the service which the actual one depends on. In the persistence format this is always the local service name.
* version (*string*): The version of the service (for the absolute form, only).
* type (*string*): the type of the service (see 2.5)
* variant (*map[sting]string*): a specification of the variant
* inheritFrom (*map[sting]string*): an optional base variant.
* abstract (*boolean*): usable as base variant, only
* shortName (*sting*): a short expressive name for the service
* description (*string*): a description of the service
* labels ([]*label*): a list of arbitrary labels

A service may exist in multiple variants, for example deployment environments. Every variant may require different dependencies. This could be described by a complex formalism to describe conditional dependencies and other conditional information, wherever it is required (for example the managed services). Here, we propose to use completely different specifications and just an additional field to describe the variant together with the option to inherit from a base variant.

Depending on the type of the service there may be additional fields.

### Labels

A label has a type, version and a formally arbitrary value. The format of the value should be defined by the label type and version. The default version is v1.

Here are the formal fields of a label:

* type (*string*): The type of the label. The type represents the meaning of the label value.
* version (*string*): The format version of the label value.
* value (*any*): The value of the label according to its format version.

### Service Dependencies

As described in the first section, a general service has two kinds of dependencies:

* Implementation dependencies and
* Orchestration Dependencies.

Dependencies are described by the identity of the service (according to 2.2.2) and semver-based version constraints. The version constraints can be used by an installation framework to decide on versions for installations and possible upgrade-paths for version updates.

A dependency has the following formal fields:

* name (*string*): The name of the dependency.
* service (*service identity/name*): The identity of the service which the actual one depends on.
* variant (*map[sting]string*): optionally a dedicated required variant specification.
* kind (string): Dependency type, one of`implementation` or `orchestration`.
* versionConstraints (*[]Semver Version Constraint*): Version constraints of the form like 1.2.x or >=1.2.1.
* accounts (*boolean*): Used for service providers to indicate that the account management API is required. (this is still quite fluffy and has still to be concretized)
* serviceInstances (*[]instance*): for service providers the list of used instance types
  + service (*service identity/name*): service identity/name
  + versions *([]string*): version list
  + dynamic (*boolean*): arbitrary number of instances
  + static: (*[]static*): list of logical static instances
    - name (*string*): logical name of static instance
* optional (*boolean*): Indicates an option dependency.
* description (*string*): a description what this dependency is used for.
* labels ([]*label*): a list of arbitrary labels

Labels can be used to assign arbitrary but formal information to a dependency.

## Service Types

When describing a service, the different types identified in the first section have to be distinguished. This also means, that *something*, which is called as service in some human context, has to be split into its formal parts for a description of the technical reality. The most common misunderstandings arise when talking about a service mixing service provider, service instance and may be even its installation.

To be able to describe technical relevant information usable to automatically setup service landscapes for cloud scenarios, we have to distinguish four different types of services, which are described in detail in the next sub sections:

* *Ordinary Services*
* *Service Providers*: services providing a request/lifecycle API to manage other services (managed service instances) and an account API
* *Installers*: services usable by an installation framework to install a dedicated instance of a service
* *Contracts*: abstract services describing just the API of a service, but not a dedicated installation.

### Ordinary Services

First of all, there are ordinary services. It is a simple service without any specific meaning in the realm of the service description model. Their formal type is `OrdinaryService`. Those services just feature the typical dependencies. An ordinary service always represents a dedicated service implementation featuring some defined API.

To be usable in a service landscape an ordinary service must be installable. This can either be described by assigning an installation service or there is a service provider for this service.

It has the following additional fields in the service description:

* external (*boolean*): (optional) indicate an external service without further description in terms of this model.
* dependencies ([]*service dependency*): the dependencies of the service.
* installers (*[]installer description*): A list of potential installer descriptions.
* contracts (*[]contract implementation description*): A list of potential contracts implemented by this kind of service.

#### Installer Descriptions

An installer description has the following formal fields:

* service (*service identity/name*): The identity of the installer service.
* version (*string*): The version of the installer service (required only for the absolute form)
* variant (*map[sting]string*): optionally a dedicated required variant specification.
* description (*string*): a description what this dependency is used for.
* labels ([]*label*): a list of arbitrary labels

#### Contract Implementation Descriptions

A contract implementation description lists the contracts implemented by the service. It has the following formal fields:

* service (*service identity/name*): The identity of the service contract.
* version (*string*): The version of the service contract (required only for the absolute form)
* description (*string*): A description to explain remarks on the implementation.
* labels ([]*label*): a list of arbitrary labels

#### Example

An example for an ordinary service can be an ABAP system as provided by the Steampunk service provider. Its description could look like this:

type: relativeServiceModelDescription/v1

services:

- service: abap

type: OrdinaryService

shortName: ABAP System

description: An ABAP system provided by the Steampunk service provider

dependencies:

- name: runtime

kind: implementation

service: kubernetes.io/apis/cluster

versionConstraints:

- 1.22.x

- name: database

kind: implementation

service: acme.org/hana/apis/database

versionConstraints:

- 1.5.x

It does not provide an installer, because it will be managed by a service provider. To run the ABAP instance a vanilla Kubernetes cluster (here in version 1.22.x) and a HANA database is required.

If this document is stored in an OCM component version acme.org/steampunk/apis:v8.0.0 the described service has the identity acme.org/steampunk/apis/abap and the version v8.0.0.

### Service Providers

A service provider is a service with an API usable to manage the lifecycle of other kinds of services. (For example, Gardener is a service usable to create and manage Kubernetes Clusters. The type name is `ServiceProvider`.

A service provider description has the following additional fields:

* dependencies ([]*service dependency*): the dependencies of the service
* external (*boolean*): (optional) indicate an external service without further description in terms of this model.
* installers (*[]installer description*): A list of potential installer descriptions.
* contracts (*[]contract implementation description*): A list of potential contracts implemented by this kind of service.
* managedServices ([]*managed service description*): a list service kinds manageable by this service provider

#### Managed Service Descriptions

A description of a managed service describes a service instance manageable by a service provider. It has the following fields:

* service (*service identity/name*): The identity of the managed service.
* versions ([]*string*): The version of the service (required only for the absolute form)
* variant (*map[sting]string*): optionally a dedicated required variant specification.
* labels ([]*label*): a list of arbitrary labels
* dependencyResolutions: (*[]resolution*): classification of the dependencies of the described service.

A resolution entry describes how the dependency of a managed service is resolved. There are two possibilities:

* managed: The service dependency is managed by the service provider.
* configured : The service dependency is exposed as configuration option to the service request.

A service provider can freely decide how to resolve dependencies of a managed service. For example, the dependency to the service runtime: It is typically an implementation dependency. But the provider can decide to support the deployment into a consumer-provided runtime (vertical service deployment). In such a case it maps the implementation dependency of the managed service to a configured dependency in the context of its service provisioning. For horizontal service deployments the provider manages the runtime on its own, either by providing an own runtime per instance or by sharing managed runtimes for multiple service instances.

A resolution entry has the following fields:

* name (*string*): The dependency name as used in the managed service.
* resolution (string): the dependency resolution mode.
* usage (string): (for managed dependencies) it describes whether the dependency
  + is resolved exclusively by a dedicated instance (exclusive) or whether
  + it is shared (shared) or
  + configurable (configured).
* labels ([]*label*): a list of arbitrary labels.

This specification generally describes the way how dependencies of the managed service are handled by the service provider.

#### Example

An example for a service provider can be the Steampunk service provider able to manage ABAP systems as described in section 2.5.1.3. Its description could look like this:

type: relativeServiceModelDescription/v1

services:

- service: provider

type: ServiceProvider

shortName: ABAP as a Service

description: The Steampunk service provider managing ABAP systems.

dependencies:

- name: haas

service: acme.org/hana/service/provider

versionConstraints:

- v1.x.x

serviceInstances:

- service: acme.org/hana/apis/database

dynamic: true

- name: kaas

service: acme.org/gardener/service/provider

versionConstraints:

- v1.x.x

serviceInstances:

- service: acme.org/gardener/apis/cluster

dynamic: true

static:

- name: runtime

managedServices:

- service: acme.org/steampunk/apis/abap

versions:

- v8.0.0

dependencyResolutions:

- name: runtime

resolution: managed

usage: shared

- name: database

resolution: managed

usage: exclusive

installers:

- service: installer

The service provider offers service instances as described in section 2.5.1.3 with the version v8.0.0. Hereby, the dependencies of such instances (runtime and database) are managed by the service provider and are not exposed to the requester. This means, the service provider takes care about the lifecycle of the required services. Therefore, it uses service providers declared as dependencies. Multiple ABAP instances may share the same runtime (Kubernetes) but use exclusive database instances.

The HaaS service provider is used to request databases. The number of requested databases is non-static, because for every ABAP instance an own database is used. For the runtime reference the Gardener service provider is used to provide Gardener Kubernetes clusters, which (must) meet the vanilla Kubernetes API described by the ABAP service definition. It uses a static instance (runtime) for the service management and an arbitrary number of clusters to run the ABAP instances.

Additionally, it can be installed by an installer with the local name installer, described in the same component version (see below).

Under the assumption, that this document is stored in the OCM component version acme.org/steampunk/service:v1.0.0 the described service has the identity acme.org/steampunk/service/provider and the version v1.0.0.

### Installation Service

An installation service is a special type of service usable by a landscape management environment to manage the lifecycle of a dedicated service instance in a service landscape. There is always one instance of an installation service managing an instance of the installed service kind. The type name of an installation service is `InstallationService`.

Such a landscape management environment is the recursion base in the automatic deployment hierarchy of services and service providers. Basically, it is some kind of generic service provider, which can handle installations as managed service based on known installation services. Therefore, the installation service must relate to a technical installer type described by a component version of the open component model. This is typically expressed by the resource type of the referenced resource artifact and must not necessarily be expressed by the service description.

An installation service has the following additional fields:

* dependencies ([]*service dependency*): the dependencies of the installation service.
* contracts ([]contract description): A list of potential contracts implemented by this kind of service.
* targetEnvironment (*map{string]string*): a property set describing the target environment for the installation.
* installedService (*service id/name*): the service installable by the installer.
* versions (*[]string*): the versions installable by this installer.
* installerResource (*(relative) resource reference*): The artifact in a component version describing the installer. The artifact type technically describes the kind of installer and how it is executed.
* installerType (*string*): optionally the resource type of the installer artifact describing the technical kind of installer.We have to decide whether we want to express this kind of contract as dedicated service contract.

The service dependencies describe the services required by the installer to install an instance. This might be plain service instances or service providers used by the installer to create service instance of service kinds required for the installed service instance. With its dependencies the installer must be able to resolve the dependencies of the installed service instance.

#### Example

An example for an installation service can be the installer for the Steampunk service provider. The installer will be used by a landscape installation system, to handle the lifecycle in a service landscape. The description could look like this:

…

- service: installer

type: InstallationService

shortName: Installer for Steampunk

installedService: provider

targetEnvironment:

type: KubernetesCluster

dependencies:

- name: haas

kind: orchestration

service: acme.org/hana/service/provider

versionContraints:

- v1.x.x

- name: kaas

kind: orchestration

service: acme.org/gardener/service/provider

versionContraints:

- v1.x.x

installerType: Deplomat

installerResource: ... ref to installer resource in OCM

The installer supports the installation contract to run in the Deplomat environment (a landscape installation system). It can be used install the Steampunk service provider. To do so, it requires an instance of

The HaaS service provider and the gardener Service provider to resolve the dependencies of the Steampunk service provider. Additionally, it describes the technical artifact, which contains the Deplomat description for an installer in form of a relative resource reference to a resource stored along with the service description artifact.

### Service Contracts

A service contract can be used to describe a dependency instead of a regular service implementation kind. It does not describe a dedicated implementation together with its API, but an API, only. It just describes how any service implementation conforming to this contract can be used. The type name is `ServiceContract`.

The description may contain fields specifying the exact behavior in any formalism. But this is optional. Formally the behavior is bound to the identity of the service. Selection based on the service description model just match the service identities, the concrete behavior is of no interest on this level.

As API representation it does not describe service dependencies as it is required for service implementation kinds.

The only additional formal fields relate to an optional description of the formal API specification in a dedicated format:

* apiSpecificationType (*string*): The type of the API specification.
* apiSpecificationVersion (*string*): the format version.
* specification (*any*): specification according to specified format.
* artifact (*(relative) resource reference*): alternatively, it can refer to an OCM artifact containing the specification.

#### Example

An example for a contract can be the description of a vanilla Kubernetes cluster as runtime service. It is not bound to a dedicated implementation, but it just describes the behavior (or the service API). A description stored in the OCM component version kubernetes.io/apis:v1.22.0 could look as follows:

type: relativeServiceModelDescription/v1

services:

- service: cluster

type: contract

shortName: vanilla kubernetes cluster implementation version 1.22.x

It is requested as dependency for the ABAP system described as service in 2.5.1.3. To fulfill this dependency in the context of the Steampunk service provider described in 2.5.2.2, the Kubernetes service managed by Gardener must implement this contract, as described in the service instance description for Gardener-provided Kubernetes clusters stored in an OCM component version like acme.org/gardener/apis:v1.x.x.

type: relativeServiceModelDescription/v1

services:

- service: cluster

type: OrdinaryService

shortName: Kubernets Cluster managed by Gardener with version 1.22.x

contracts:

- service: kubernetes.io/apis/cluster

version: v1.22.0

# Specialized Landscape Services

To support a complete automation for a landscape of service providers some specialized services, or better service providers, could be used.

As described in the previous section, the model uses installation services as recursion base for a landscape setup. This kind of service is handled as managed service by a *landscape installation system* (LISY*)*.

The second area, where a standardized service provider could support a landscape lifecycle is the management of technical accounts used by the service providers for their technical interaction: a *service account management* (SAM).

## Landscape Installation Service

The task of the landscape installation service is to manage the lifecycle of top-level services used to compose the landscape. In most of the cases these are service providers, but not necessarily. In terms of the described service model, this a service provider. The managed service instances are the top-level service installations in the landscape.

As described by the description model, there is a special kind of service, the installation service, which can be used to install a dedicated instance of a particular service kind. The managed instance of such a landscape installation service just specifies such an installer together with its configuration. If the technical execution API of this service (formalized either by a separate contract, or by the OCM resource type of the bound delivery artifact) the service lifecycle of the service can be managed by this meta service provider by executing the lifecycle operations of the specified installation service.

Based on dependencies among those digital twins the service provider can detect installation orders, when executing updates.

These kinds of service instance requests require to resolve dependencies of the installers as input for the installers. For service providers this must be account endpoints. This could be done via configuration objects maintained in the landscape installation service. Or another kind of service provider could be established, a service account management.

## Service Account Management

Every service provider has a management API used to manage the lifecycle of managed service instances. To support this, it has to offer accounts used by its consumers to manage their instances. This requires using account endpoints for service wiring. Every consuming service must work under the control of some account with a service provider.

Therefore, a service provider requires some API to manage accounts in the context of this service provider. There are basically two different ways this could be done:

* The service provider uses a special management API to manage accounts.
* Accounts are regular instances managed as part of accounts by the service provider. Then there is one root account, which can be used to manage the final top-level consumer accounts.

Regardless of the chosen variant, the landscape installation service must have the possibility to create access points to accounts in service providers, which are required to resolve installation dependencies of managed installations. Those accounts could be configuration objects, but this would require the LISY to get access to all account management APIs.

Another possibility could be to use another kind of service provider, the *service account management*. This is a service used to manage service accounts of service providers connected to the account management service. As regular service provider it also provides accounts for its users. As managed service it offers account requests for any connected service provider (which requires an identity of the provider in the landscape). The result of such a request is the access point and the required credentials useable to access the requested service provider.

Once the LISY has one account in such a service, it can:

* create further accounts in the SAM to be used by particular managed service providers.
* use those accounts to requests accounts in other service providers required to resolve installation dependencies.

Both informations are then passed to the installation requests and finally reach the installed service provider, which uses this information to issue further requests to fulfill its tasks.

The SAM account exclusively created for the installed service provider can then be used by an adapter as part of the service provider to look for and handle account requests for this service provider, without the need to explicitly expose an own consumer facing account management API. SAM acts as recursion base for the technical account management. (Please be aware, that the primary use case for this is the inter-service-provider interaction)

For example, let’s refer to our Steampunk example:

* The Steampunk service requires access to the Gardener and Hana service provider to request Kubernetes runtimes and databases for the managed ABAP systems.
* The Hana service requires access to the Gardener to request Kubernetes runtimes for Hana Databases.

Every such service provides an adapter for the SAM to enable LISY to provide appropriate accounts.

Gardener as base-runtime service provider must be installed before LISY, but then always the same happens:

* The service provider to be installed has dependencies to a service provider (which has been installed earlier by LISY)
* LIS uses its SAM account to create a SAM account for the service provider to install.
* It either
  + passes this account to the installer, which uses this account to create the required accounts in the dependent service providers or
  + uses this account to request accounts in the dependent services providers and uses those endpoints to resolve the installer dependencies.
* The installed service provider can use the given account(s) to access the used foreign service providers (in our example, the Gardener to request Kubernetes clusters in required regions, or the Hana service to request Hana databases)

If the technical accounts are also used to implement provider local customer accounts, it is even possible to create accounts in used service providers on-behalf of the initially requesting account. This might be important, if account information is gathered or grouped by accounts. In our example, this would mean, that the Steampunk service creates accounts in the Hana Service for its own accounts to be able to place required Hana databases requested for an ABAP-System in one if its own accounts in an associated account of the Hana service. Therefore, Steampunk would use its own SAM account.

## Landscape Modeler

Based on the service description documents a database can be filled, which can be used to follow or resolve dependencies. Starting with some desired root services, the dependencies can be used to lookup required definitions in an OCM repository used to host all the component versions to automatically complete the content of the database.

This information can then be used to provide an UI usable to create or enrich service landscape templates

by using the service description model to propose already configured instances or even new service implementations to create service provider instances and resolve service dependencies in a landscape. The result is a complete set of service installations with the intended versions.

The same way version constraints can be examined to find intermediate upgrade paths represented as new versions of a landscape template.

Those landscape templates can then be passed to the LISY to control the service set and versions in a landscape, which then executes the upgrade in the correct order.