SOASIS TOSCA Simple Profile in YAML Version 1.2

OASIS Standard

17 January 2019

Specification URIs

This version:

https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/os/TOSCA-Simple-Profile-YAML-v1.2-os.pdf (Authoritative)

https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/os/TOSCA-Simple-Profile-YAML-v1.2-os.html

https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/os/TOSCA-Simple-Profile-YAML-v1.2-os.docx

Previous version:

http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/csd01/TOSCA-Simple-Profile-YAML-v1.2-csd01.pdf (Authoritative)

http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/csd01/TOSCA-Simple-Profile-YAML-v1.2-csd01.html

http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/csd01/TOSCA-Simple-Profile-YAML-v1.2-csd01.docx

Latest version:

https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/TOSCA-Simple-Profile-YAML-v1.2.pdf (Authoritative)

https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/TOSCA-Simple-Profile-YAML-v1.2.html

https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/TOSCA-Simple-Profile-YAML-v1.2.docx

Technical Committee:

OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC

Chairs:

Paul Lipton (paul.lipton@ca.com), CA Technologies Simon Moser (smoser@de.ibm.com), IBM

Editors:

Matt Rutkowski (mrutkows@us.ibm.com), IBM Luc Boutier (luc.boutier@fastconnect.fr), FastConnect Chris Lauwers (lauwers@ubicity.com), Individual Member

Related work:

This specification replaces or supersedes:

- TOSCA Simple Profile in YAML Version 1.1. Edited by Matt Rutkowski and Luc Boutier. Latest version: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.1/TOSCA-Simple-Profile-YAML-v1.1.html.
- TOSCA Simple Profile in YAML Version 1.0. Edited by Derek Palma, Matt Rutkowski, and Thomas Spatzier. Latest version: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.html.

This specification is related to:

 Topology and Orchestration Specification for Cloud Applications Version 1.0. Edited by Derek Palma and Thomas Spatzier. 25 November 2013. OASIS Standard. http://docs.oasisopen.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html.

Declared XML namespace:

http://docs.oasis-open.org/tosca/ns/simple/yaml/1.2

Abstract:

This document defines a simplified profile of the TOSCA version 1.0 specification in a YAML rendering which is intended to simplify the authoring of TOSCA service templates. This profile defines a less verbose and more human-readable YAML rendering, reduced level of indirection between different modeling artifacts as well as the assumption of a base type system.

Status:

This document was last revised or approved by the membership of OASIS on the above date. The level of approval is also listed above. Check the "Latest version" location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Technical Committee (TC) are listed at https://www.oasisopen.org/committees/tc_home.php?wg_abbrev=tosca#technical.

TC members should send comments on this specification to the TC's email list. Others should send comments to the TC's public comment list, after subscribing to it by following the instructions at the "Send A Comment" button on the TC's web page at https://www.oasis-open.org/committees/tosca/.

This specification is provided under the RF on Limited Terms Mode of the OASIS IPR Policy, the mode chosen when the Technical Committee was established. For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC's web page (https://www.oasis-open.org/committees/tosca/ipr.php).

Note that any machine-readable content (Computer Language Definitions) declared Normative for this Work Product is provided in separate plain text files. In the event of a discrepancy between any such plain text file and display content in the Work Product's prose narrative document(s), the content in the separate plain text file prevails.

Citation format:

When referencing this specification the following citation format should be used:

[TOSCA-Simple-Profile-YAML-v1.2]

TOSCA Simple Profile in YAML Version 1.2. Edited by Matt Rutkowski, Luc Boutier, and Chris Lauwers. 17 January 2019. OASIS Standard. https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/os/TOSCA-Simple-Profile-YAML-v1.2-os.html. Latest version: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/TOSCA-Simple-Profile-YAML/v1.2/TOSCA-Simple-Profile-YAML-v1.2.html.

Notices

Copyright © OASIS Open 2019. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name "OASIS" is a trademark of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see https://www.oasis-open.org/policies-guidelines/trademark for above guidance.

Table of Contents

Та	able of Examples	7
Та	able of Figures	7
1	Introduction	8
	1.0 IPR Policy	8
	1.1 Objective	8
	1.2 Summary of key TOSCA concepts	8
	1.3 Implementations	9
	1.4 Terminology	9
	1.5 Notational Conventions	9
	1.6 Normative References	9
	1.7 Non-Normative References	10
	1.8 Glossary	10
2	TOSCA by example	12
	2.1 A "hello world" template for TOSCA Simple Profile in YAML	
	2.2 TOSCA template for a simple software installation	14
	2.3 Overriding behavior of predefined node types	
	2.4 TOSCA template for database content deployment	
	2.5 TOSCA template for a two-tier application	19
	2.6 Using a custom script to establish a relationship in a template	
	2.7 Using custom relationship types in a TOSCA template	
	2.8 Defining generic dependencies between nodes in a template	
	2.9 Describing abstract requirements for nodes and capabilities in a TOSCA template	
	2.10 Using node template substitution for model composition	
	2.11 Using node template substitution for chaining subsystems	
	2.12 Grouping node templates	40
	2.13 Using YAML Macros to simplify templates	
	2.14 Passing information as inputs to Nodes and Relationships	44
	2.15 Topology Template Model versus Instance Model	
	2.16 Using attributes implicitly reflected from properties	
3	· • • • · · · · · · · · · · · · · · · ·	
	3.1 TOSCA Namespace URI and alias	
	3.2 Using Namespaces	
	3.3 Parameter and property types	
	3.4 Normative values	
	3.5 TOSCA Metamodel	
	3.6 Reusable modeling definitions	
	3.7 Type-specific definitions	
	3.8 Template-specific definitions	
	3.9 Topology Template definition	
	3.10 Service Template definition	
4		
	4.1 Reserved Function Keywords	
	4.2 Environment Variable Conventions	153

	4.3 Intrinsic functions	. 156
	4.4 Property functions	. 158
	4.5 Attribute functions	. 160
	4.6 Operation functions	. 162
	4.7 Navigation functions	. 162
	4.8 Artifact functions	. 163
	4.9 Context-based Entity names (global)	. 165
5	TOSCA normative type definitions	166
	5.1 Assumptions	166
	5.2 TOSCA normative type names	166
	5.3 Data Types	166
	5.4 Artifact Types	. 178
	5.5 Capabilities Types	. 181
	5.6 Requirement Types	192
	5.7 Relationship Types	192
	5.8 Interface Types	195
	5.9 Node Types	201
	5.10 Group Types	213
	5.11 Policy Types	214
6	TOSCA Cloud Service Archive (CSAR) format	216
	6.1 Overall Structure of a CSAR	216
	6.2 TOSCA Meta File	216
	6.3 Archive without TOSCA-Metadata	217
7	TOSCA workflows	218
	7.1 Normative workflows	218
	7.2 Declarative workflows	218
	7.3 Imperative workflows	222
	7.4 Making declarative more flexible and imperative more generic	. 237
8	TOSCA networking	240
	8.1 Networking and Service Template Portability	. 240
	8.2 Connectivity semantics	. 240
	8.3 Expressing connectivity semantics	241
	8.4 Network provisioning	243
	8.5 Network Types	247
	8.6 Network modeling approaches	. 253
9	Non-normative type definitions	. 258
	9.1 Artifact Types	. 258
	9.2 Capability Types	. 258
	9.3 Node Types	. 260
1(Component Modeling Use Cases	. 264
1	1 Application Modeling Use Cases	271
	11.1 Use cases	271
12	2 TOSCA Policies	319
	12.1 A declarative approach	319
	12.2 Consideration of Event, Condition and Action	319

12.3 Types of policies	319
12.4 Policy relationship considerations	320
12.5 Use Cases	321
13 Artifact Processing and creating portable Service Templates	324
13.1 Artifacts Processing	324
13.2 Dynamic Artifacts	328
13.3 Discussion of Examples	328
13.4 Artifact Types and Metadata	335
14 Abstract nodes and target node filters matching	336
14.1 Reminder on types	336
14.2 Orchestrator catalogs	336
14.3 Abstract node template matching	337
14.4 Target node filter matching	342
14.5 Post matching properties	347
15 Conformance	348
15.1 Conformance Targets	348
15.2 Conformance Clause 1: TOSCA YAML service template	348
15.3 Conformance Clause 2: TOSCA processor	348
15.4 Conformance Clause 3: TOSCA orchestrator	348
15.5 Conformance Clause 4: TOSCA generator	349
15.6 Conformance Clause 5: TOSCA archive	349
Appendix A. Known Extensions to TOSCA v1.0	350
A.1 Model Changes	350
A.2 Normative Types	350
Appendix B. Acknowledgments	352
Appendix C. Revision History	354

Table of Examples

2
3
5
6
7
0
2
4
5
5
7
8
9
0
2
3
5
7
8
0
1
3
6

Table of Figures

Figure 1: Using template substitution to implement a database tier	31
Figure 2: Substitution mappings	33
Figure 3: Chaining of subsystems in a service template	35
Figure 4: Defining subsystem details in a service template	
Figure-5: Typical 3-Tier Network	244
Figure-6: Generic Service Template	
Figure-7: Service template with network template A	
Figure-8: Service template with network template B	254

1 1 Introduction

2 1.0 IPR Policy

3 This specification is provided under the RF on Limited Terms Mode of the OASIS IPR Policy, the mode

4 chosen when the Technical Committee was established. For information on whether any patents have

5 been disclosed that may be essential to implementing this specification, and any offers of patent licensing

6 terms, please refer to the Intellectual Property Rights section of the TC's web page (https://www.oasis-

7 open.org/committees/tosca/ipr.php).

8 1.1 Objective

9 The TOSCA Simple Profile in YAML specifies a rendering of TOSCA which aims to provide a more

10 accessible syntax as well as a more concise and incremental expressiveness of the TOSCA DSL in order

to minimize the learning curve and speed the adoption of the use of TOSCA to portably describe cloudapplications.

13 This proposal describes a YAML rendering for TOSCA. YAML is a human friendly data serialization

14 standard (http://yaml.org/) with a syntax much easier to read and edit than XML. As there are a number of

- 15 DSLs encoded in YAML, a YAML encoding of the TOSCA DSL makes TOSCA more accessible by these
- 16 communities.

17 This proposal prescribes an isomorphic rendering in YAML of a subset of the TOSCA v1.0 XML

18 specification ensuring that TOSCA semantics are preserved and can be transformed from XML to YAML

19 or from YAML to XML. Additionally, in order to streamline the expression of TOSCA semantics, the YAML

20 rendering is sought to be more concise and compact through the use of the YAML syntax.

1.2 Summary of key TOSCA concepts

22 The TOSCA metamodel uses the concept of service templates to describe cloud workloads as a topology 23 template, which is a graph of node templates modeling the components a workload is made up of and as relationship templates modeling the relations between those components. TOSCA further provides a type 24 25 system of node types to describe the possible building blocks for constructing a service template, as well 26 as relationship type to describe possible kinds of relations. Both node and relationship types may define 27 lifecycle operations to implement the behavior an orchestration engine can invoke when instantiating a 28 service template. For example, a node type for some software product might provide a 'create' operation 29 to handle the creation of an instance of a component at runtime, or a 'start' or 'stop' operation to handle a 30 start or stop event triggered by an orchestration engine. Those lifecycle operations are backed by

31 implementation artifacts such as scripts or Chef recipes that implement the actual behavior.

An orchestration engine processing a TOSCA service template uses the mentioned lifecycle operations to instantiate single components at runtime, and it uses the relationship between components to derive the order of component instantiation. For example, during the instantiation of a two-tier application that includes a web application that depends on a database, an orchestration engine would first invoke the

36 'create' operation on the database component to install and configure the database, and it would then

invoke the 'create' operation of the web application to install and configure the application (which includes

38 configuration of the database connection).

The TOSCA simple profile assumes a number of base types (node types and relationship types) to be supported by each compliant environment such as a 'Compute' node type, a 'Network' node type or a generic 'Database' node type. Furthermore, it is envisioned that a large number of additional types for use in service templates will be defined by a community over time. Therefore, template authors in many cases will not have to define types themselves but can simply start writing service templates that use existing types. In addition, the simple profile will provide means for easily customizing and extending existing types, for example by providing a customized 'create' script for some software.

46 **1.3 Implementations**

Different kinds of processors and artifacts qualify as implementations of the TOSCA simple profile. Those
 that this specification is explicitly mentioning or referring to fall into the following categories:

- TOSCA YAML service template (or "service template"): A YAML document artifact containing a
 (TOSCA) service template (see sections 3.9 "Service template definition") that represents a Cloud
 application. (see sections 3.8 "Topology template definition")
- TOSCA processor (or "processor"): An engine or tool that is capable of parsing and interpreting a
 TOSCA service template for a particular purpose. For example, the purpose could be validation,
 translation or visual rendering.
- TOSCA orchestrator (also called orchestration engine): A TOSCA processor that interprets a
 TOSCA service template or a TOSCA CSAR in order to instantiate and deploy the described
 application in a Cloud.
- TOSCA generator: A tool that generates a TOSCA service template. An example of generator is
 a modeling tool capable of generating or editing a TOSCA service template (often such a tool
 would also be a TOSCA processor).
- TOSCA archive (or TOSCA Cloud Service Archive, or "CSAR"): a package artifact that contains a
 TOSCA service template and other artifacts usable by a TOSCA orchestrator to deploy an
 application.

64 The above list is not exclusive. The above definitions should be understood as referring to and

65 implementing the TOSCA simple profile as described in this document (abbreviated here as "TOSCA" for 66 simplicity).

67 1.4 Terminology

68 The TOSCA language introduces a YAML grammar for describing service templates by means of

69 Topology Templates and towards enablement of interaction with a TOSCA instance model perhaps by

70 external APIs or plans. The primary currently is on design time aspects, i.e. the description of services to

- 71 ensure their exchange between Cloud providers, TOSCA Orchestrators and tooling.
- 72
- The language provides an extension mechanism that can be used to extend the definitions with additional
 vendor-specific or domain-specific information.

75 **1.5 Notational Conventions**

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

79 **1.5.1 Notes**

• Sections that are titled "Example" throughout this document are considered non-normative.

81 **1.6 Normative References**

Reference Tag	Description
[RFC2119]	S. Bradner, <i>Key words for use in RFCs to Indicate Requirement Levels</i> , http://www.ietf.org/rfc/rfc2119.txt, IETF RFC 2119, March 1997.
[TOSCA-1.0]	Topology and Orchestration Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0, an OASIS Standard, 25

	November 2013, http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.pdf
[YAML-1.2]	YAML, Version 1.2, 3rd Edition, Patched at 2009-10-01, Oren Ben-Kiki, Clark Evans, Ingy döt Net http://www.yaml.org/spec/1.2/spec.html
[YAML-TS-1.1]	Timestamp Language-Independent Type for YAML Version 1.1, Working Draft 2005-01-18, http://yaml.org/type/timestamp.html

82 1.7 Non-Normative References

Reference Tag	Description
[Apache]	Apache Server, https://httpd.apache.org/
[Chef]	Chef, https://wiki.opscode.com/display/chef/Home
[NodeJS]	Node.js, https://nodejs.org/
[Puppet]	Puppet, http://puppetlabs.com/
[WordPress]	WordPress, https://wordpress.org/
[Maven- Version]	Apache Maven version policy draft: https://cwiki.apache.org/confluence/display/MAVEN/Version+number+policy
[JSON-Spec]	 The JSON Data Interchange Format (ECMA and IETF versions): http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf https://tools.ietf.org/html/rfc7158
[JSON-Schema]	JSON Schema specification: http://json-schema.org/documentation.html
[XMLSpec]	XML Specification, W3C Recommendation, February 1998, http://www.w3.org/TR/1998/REC-xml-19980210
[XML Schema Part 1]	XML Schema Part 1: Structures, W3C Recommendation, October 2004, 🔛 http://www.w3.org/TR/xmlschema-1/
[XML Schema Part 2]	XML Schema Part 2: Datatypes, W3C Recommendation, October 2004, 📰 http://www.w3.org/TR/xmlschema-2/

83 1.8 Glossary

84 The following terms are used throughout this specification and have the following definitions when used in 85 context of this document.

Term	Definition
Instance Model	A deployed service is a running instance of a Service Template. More precisely, the instance is derived by instantiating the Topology Template of its Service Template, most often by running a special plan defined for the Service Template, often referred to as build plan.
Node Template	A <i>Node Template</i> specifies the occurrence of a software component node as part of a Topology Template. Each Node Template refers to a Node Type that defines the semantics of the node (e.g., properties, attributes, requirements, capabilities, interfaces). Node Types are defined separately for reuse purposes.
Relationship Template	A Relationship Template specifies the occurrence of a relationship between nodes in a Topology Template. Each Relationship Template refers to a Relationship Type that defines the semantics relationship (e.g., properties,

	attributes, interfaces, etc.). Relationship Types are defined separately for reuse purposes.
Service Template	A Service Template is typically used to specify the "topology" (or structure) and "orchestration" (or invocation of management behavior) of IT services so that they can be provisioned and managed in accordance with constraints and policies.
	Specifically, TOSCA Service Templates optionally allow definitions of a TOSCA Topology Template, TOSCA types (e.g., Node, Relationship, Capability, Artifact, etc.), groupings, policies and constraints along with any input or output declarations.
Topology Model	The term Topology Model is often used synonymously with the term Topology Template with the use of "model" being prevalent when considering a Service Template's topology definition as an abstract representation of an application or service to facilitate understanding of its functional components and by eliminating unnecessary details.
Topology Template	A Topology Template defines the structure of a service in the context of a Service Template. A Topology Template consists of a set of Node Template and Relationship Template definitions that together define the topology model of a service as a (not necessarily connected) directed graph.
	The term Topology Template is often used synonymously with the term Topology Model. The distinction is that a topology template can be used to instantiate and orchestrate the model as a <i>reusable pattern</i> and includes all details necessary to accomplish it.
Abstract Node	
Template	An abstract node template is a node that doesn't define an implementation artifact for the create operation of the TOSCA lifecycle.
Template	
Template	artifact for the create operation of the TOSCA lifecycle.

2 TOSCA by example

87 This non-normative section contains several sections that show how to model applications with TOSCA

88 Simple Profile using YAML by example starting with a "Hello World" template up through examples that 89 show complex composition modeling.

2.1 A "hello world" template for TOSCA Simple Profile in YAML

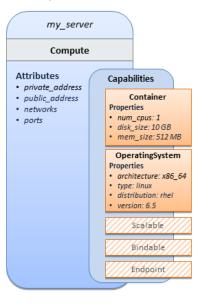
As mentioned before, the TOSCA simple profile assumes the existence of a small set of pre-defined, normative set of node types (e.g., a 'Compute' node) along with other types, which will be introduced through the course of this document, for creating TOSCA Service Templates. It is envisioned that many additional node types for building service templates will be created by communities some may be published as profiles that build upon the TOSCA Simple Profile specification. Using the normative TOSCA Compute node type, a very basic "Hello World" TOSCA template for deploying just a single server would look as follows:

98 Example 1 - TOSCA Simple "Hello World"

```
tosca definitions version: tosca simple yaml 1 0
description: Template for deploying a single server with predefined properties.
topology template:
  node templates:
    my server:
      type: tosca.nodes.Compute
      capabilities:
        # Host container properties
        host:
         properties:
           num_cpus: 1
           disk size: 10 GB
           mem size: 4096 MB
        # Guest Operating System properties
        os:
          properties:
            # host Operating System image properties
            architecture: x86 64
            type: linux
            distribution: rhel
            version: 6.5
```

99 The template above contains a very simple topology template with only a single 'Compute' node template 100 that declares some basic values for properties within two of the several capabilities that are built into the 101 Compute node type definition. All TOSCA Orchestrators are expected to know how to instantiate a 102 Compute node since it is normative and expected to represent a well-known function that is portable 103 across TOSCA implementations. This expectation is true for all normative TOSCA Node and 104 Relationship types that are defined in the Simple Profile specification. This means, with TOSCA's

- approach, that the application developer does not need to provide any deployment or implementation
- 106 artifacts that contain code or logic to orchestrate these common software components. TOSCA
- 107 orchestrators simply select or allocate the correct node (resource) type that fulfills the application
- 108 topologies requirements using the properties declared in the node and its capabilities.
- 109 In the above example, the "host" capability contains properties that allow application developers to
- optionally supply the number of CPUs, memory size and disk size they believe they need when the
- 111 Compute node is instantiated in order to run their applications. Similarly, the "os" capability is used to
- 112 provide values to indicate what host operating system the Compute node should have when it is
- 113 instantiated.
- 114
- 115 The logical diagram of the "hello world" Compute node would look as follows:



116

117

- 118 As you can see, the **Compute** node also has attributes and other built-in capabilities, such as **Bindable**
- and **Endpoint**, each with additional properties that will be discussed in other examples later in this
- 120 document. Although the Compute node has no direct properties apart from those in its capabilities, other
- 121 TOSCA node type definitions may have properties that are part of the node type itself in addition to
- 122 having Capabilities. TOSCA orchestration engines are expected to validate all property values provided
- in a node template against the property definitions in their respective node type definitions referenced in
- the service template. The tosca_definitions_version keyname in the TOSCA service template
- identifies the versioned set of normative TOSCA type definitions to use for validating those types defined
- in the TOSCA Simple Profile including the Compute node type. Specifically, the value
- 127 tosca_simple_yaml_1_0 indicates Simple Profile v1.0.0 definitions would be used for validation. Other 128 type definitions may be imported from other service templates using the import keyword discussed later.

129 **2.1.1 Requesting input parameters and providing output**

- 130 Typically, one would want to allow users to customize deployments by providing input parameters instead
- 131 of using hardcoded values inside a template. In addition, output values are provided to pass information
- that perhaps describes the state of the deployed template to the user who deployed it (such as the private
- 133 IP address of the deployed server). A refined service template with corresponding **inputs** and **outputs**
- 134 sections is shown below.
- 135 *Example 2 Template with input and output parameter sections*

tosca_definitions_version: tosca_simple_yaml_1_0

```
description: Template for deploying a single server with predefined properties.
topology template:
  inputs:
    cpus:
      type: integer
      description: Number of CPUs for the server.
      constraints:
        - valid_values: [ 1, 2, 4, 8 ]
  node_templates:
   my server:
      type: tosca.nodes.Compute
      capabilities:
        # Host container properties
        host:
          properties:
            # Compute properties
            num_cpus: { get_input: cpus }
            mem size: 2048 MB
            disk_size: 10 GB
 outputs:
    server ip:
      description: The private IP address of the provisioned server.
      value: { get_attribute: [ my_server, private_address ] }
```

136 The inputs and outputs sections are contained in the topology_template element of the TOSCA

137 template, meaning that they are scoped to node templates within the topology template. Input parameters

138 defined in the inputs section can be assigned to properties of node template within the containing topology template; output parameters can be obtained from attributes of node templates within the

139

140 containing topology template.

141 Note that the **inputs** section of a TOSCA template allows for defining optional constraints on each input

142 parameter to restrict possible user input. Further note that TOSCA provides for a set of intrinsic functions

143 like get input, get property or get attribute to reference elements within the template or to

144 retrieve runtime values.

2.2 TOSCA template for a simple software installation 145

146 Software installations can be modeled in TOSCA as node templates that get related to the node template

147 for a server on which the software would be installed. With a number of existing software node types (e.g.

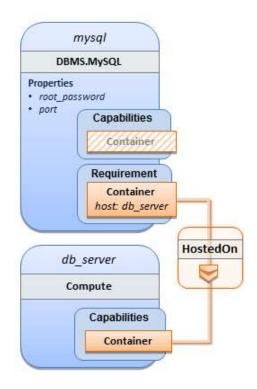
148 either created by the TOSCA work group or a community) template authors can just use those node types

149 for writing service templates as shown below. 150 Example 3 - Simple (MySQL) software installation on a TOSCA Compute node

```
tosca definitions version: tosca simple yaml 1 0
description: Template for deploying a single server with MySQL software on top.
topology_template:
  inputs:
    # omitted here for brevity
  node templates:
    mysql:
      type: tosca.nodes.DBMS.MySQL
      properties:
        root_password: { get_input: my_mysql_rootpw }
        port: { get_input: my_mysql_port }
      requirements:
        - host: db server
   db server:
      type: tosca.nodes.Compute
      capabilities:
        # omitted here for brevity
```

The example above makes use of a node type **tosca.nodes.DBMS.MySQL** for the **mysql** node template to install MySQL on a server. This node type allows for setting a property **root_password** to adapt the password of the MySQL root user at deployment. The set of properties and their schema has been defined in the node type definition. By means of the **get_input** function, a value provided by the user at deployment time is used as value for the **root_password** property. The same is true for the **port** property.

The **mysql** node template is related to the **db_server** node template (of type **tosca.nodes.Compute**) via the **requirements** section to indicate where MySQL is to be installed. In the TOSCA metamodel, nodes get related to each other when one node has a requirement against some feature provided by another node. What kinds of requirements exist is defined by the respective node type. In case of MySQL, which is software that needs to be installed or hosted on a compute resource, the underlying node type named DBMS has a predefined requirement called **host**, which needs to be fulfilled by pointing to a node template of type **tosca.nodes.Compute**. 164 The logical relationship between the **mysql** node and its host **db_server** node would appear as follows:



165

Within the **requirements** section, all entries simple entries are a map which contains the symbolic name of a requirement definition as the *key* and the identifier of the fulfilling node as the *value*. The value is essentially the symbolic name of the other node template; specifically, or the example above, the **host** requirement is fulfilled by referencing the **db_server** node template. The underlying TOSCA **DBMS** node type already defines a complete requirement definition for the **host** requirement of type **Container** and assures that a **HostedOn** TOSCA relationship will automatically be created and will only allow a valid

assures that a HostedOn TOSCA relationship will automatically be created and will only allow a valid
 target host node is of type Compute. This approach allows the template author to simply provide the

172 name of a valid **Compute** node (i.e., **db** server) as the value for the **mysql** node's **host** requirement and

174 not worry about defining anything more complex if they do not want to.

175 **2.3 Overriding behavior of predefined node types**

Node types in TOSCA have associated implementations that provide the automation (e.g. in the form of scripts such as Bash, Chef or Python) for the normative lifecycle operations of a node. For example, the node type implementation for a MySQL database would associate scripts to TOSCA node operations like

179 **configure**, **start**, or **stop** to manage the state of MySQL at runtime.

180 Many node types may already come with a set of operational scripts that contain basic commands that

181 can manage the state of that specific node. If it is desired, template authors can provide a custom script

182 for one or more of the operation defined by a node type in their node template which will override the

default implementation in the type. The following example shows a **mysql** node template where the

- 184 template author provides their own configure script:
- 185 Example 4 Node Template overriding its Node Type's "configure" interface

tosca_definitions_version: tosca_simple_yaml_1_0

description: Template for deploying a single server with MySQL software on top.

```
topology_template:
  inputs:
    # omitted here for brevity
  node_templates:
    mysql:
      type: tosca.nodes.DBMS.MySQL
      properties:
        root_password: { get_input: my_mysql_rootpw }
        port: { get input: my mysql port }
      requirements:
        - host: db server
      interfaces:
        Standard:
          configure: scripts/my_own_configure.sh
    db server:
      type: tosca.nodes.Compute
      capabilities:
```

omitted here for brevity

186 In the example above, the **my_own_configure.sh** script is provided for the **configure** operation of the 187 MySQL node type's **Standard** lifecycle interface. The path given in the example above (i.e., 'scripts/') is 188 interpreted relative to the template file, but it would also be possible to provide an absolute URI to the 189 location of the script.

190 In other words, operations defined by node types can be thought of as "hooks" into which automation can

be injected. Typically, node type implementations provide the automation for those "hooks". However,
within a template, custom automation can be injected to run in a hook in the context of the one, specific
node template (i.e. without changing the node type).

194 2.4 TOSCA template for database content deployment

In the Example 4, shown above, the deployment of the MySQL middleware only, i.e. without actual
 database content was shown. The following example shows how such a template can be extended to
 also contain the definition of custom database content on-top of the MySQL DBMS software.

198 Example 5 - Template for deploying database content on-top of MySQL DBMS middleware

tosca_definitions_version: tosca_simple_yaml_1_0

description: Template for deploying MySQL and database content.

topology_template:

inputs:

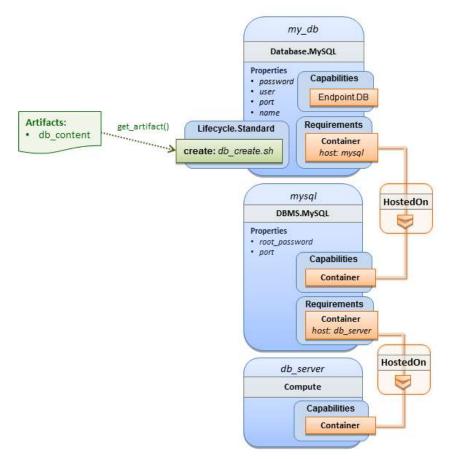
omitted here for brevity

```
node_templates:
 my db:
   type: tosca.nodes.Database.MySQL
    properties:
      name: { get_input: database_name }
      user: { get_input: database_user }
      password: { get_input: database_password }
      port: { get_input: database_port }
    artifacts:
      db content:
        file: files/my db content.txt
        type: tosca.artifacts.File
    requirements:
      - host: mysql
    interfaces:
      Standard:
        create:
          implementation: db create.sh
          inputs:
            # Copy DB file artifact to server's staging area
            db_data: { get_artifact: [ SELF, db_content ] }
 mysql:
   type: tosca.nodes.DBMS.MySQL
    properties:
      root_password: { get_input: mysql_rootpw }
      port: { get_input: mysql_port }
    requirements:
      - host: db_server
 db server:
   type: tosca.nodes.Compute
    capabilities:
      # omitted here for brevity
```

In the example above, the my_db node template or type tosca.nodes.Database.MySQL represents an
 actual MySQL database instance managed by a MySQL DBMS installation. The requirements section of
 the my_db node template expresses that the database it represents is to be hosted on a MySQL DBMS
 node template named mysql which is also declared in this template.

In its **artifacts** section of the **my_db** the node template, there is an artifact definition named **db_content** which represents a text file **my_db_content.txt** which in turn will be used to add content to the SQL

- 205 database as part of the **create** operation. The **requirements** section of the **my_db** node template 206 expresses that the database is hosted on a MySQL DBMS represented by the **mysql** node.
- As you can see above, a script is associated with the create operation with the name db_create.sh.
- The TOSCA Orchestrator sees that this is not a named artifact declared in the node's artifact section, but instead a filename for a normative TOSCA implementation artifact script type (i.e.,
- 210 tosca.artifacts.Implementation.Bash). Since this is an implementation type for TOSCA, the
- 211 orchestrator will execute the script automatically to create the node on db_server, but first it will prepare
- the local environment with the declared inputs for the operation. In this case, the orchestrator would see
- 213 that the **db_data** input is using the **get_artifact** function to retrieve the file (**my_db_content.txt**)
- which is associated with the **db_content** artifact name prior to executing the **db_create.sh** script.
- 215 The logical diagram for this example would appear as follows:



216

- 217 Note that while it would be possible to define one node type and corresponding node templates that
- 218 represent both the DBMS middleware and actual database content as one entity, TOSCA normative node
- types distinguish between middleware (container) and application (containee) node types. This allows on
- 220 one hand to have better re-use of generic middleware node types without binding them to content running
- on top of them, and on the other hand this allows for better substitutability of, for example, middleware
- 222 components like a DBMS during the deployment of TOSCA models.

223 **2.5 TOSCA template for a two-tier application**

The definition of multi-tier applications in TOSCA is quite similar to the example shown in section 2.2, with the only difference that multiple software node stacks (i.e., node templates for middleware and application layer components), typically hosted on different servers, are defined and related to each other. The example below defines a web application stack hosted on the **web server** "compute" resource, and a

database software stack similar to the one shown earlier in section 6 hosted on the **db_server** compute resource. 230 Example 6 - Basic two-tier application (web application and database server tiers)

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: Template for deploying a two-tier application servers on two
topology_template:
  inputs:
    # Admin user name and password to use with the WordPress application
    wp_admin_username:
      type: string
    wp_admin_password:
      type: string
    wp_db_name:
      type: string
    wp db user:
      type: string
    wp db password:
      type: string
    wp_db_port:
      type: integer
    mysql_root_password:
      type: string
    mysql port:
      type: integer
    context_root:
      type: string
  node_templates:
    wordpress:
      type: tosca.nodes.WebApplication.WordPress
      properties:
        context_root: { get_input: context_root }
        admin_user: { get_input: wp_admin_username }
        admin_password: { get_input: wp_admin_password }
        db_host: { get_attribute: [ db_server, private_address ] }
      requirements:
        - host: apache
        - database endpoint: wordpress db
      interfaces:
        Standard:
          inputs:
```

```
db_host: { get_attribute: [ db_server, private_address ] }
        db_port: { get_property: [ wordpress_db, port ] }
        db_name: { get_property: [ wordpress_db, name ] }
        db_user: { get_property: [ wordpress_db, user ] }
        db_password: { get_property: [ wordpress_db, password ] }
apache:
  type: tosca.nodes.WebServer.Apache
  properties:
    # omitted here for brevity
  requirements:
    - host: web server
web_server:
  type: tosca.nodes.Compute
  capabilities:
    # omitted here for brevity
wordpress db:
  type: tosca.nodes.Database.MySQL
  properties:
    name: { get_input: wp_db_name }
    user: { get_input: wp_db_user }
    password: { get_input: wp_db_password }
    port: { get_input: wp_db_port }
  requirements:
    - host: mysql
mysql:
  type: tosca.nodes.DBMS.MySQL
  properties:
    root_password: { get_input: mysql_root_password }
    port: { get_input: mysql_port }
  requirements:
    - host: db_server
db server:
  type: tosca.nodes.Compute
  capabilities:
    # omitted here for brevity
```

231 The web application stack consists of the **wordpress** [WordPress], the **apache** [Apache] and the

web_server node templates. The wordpress node template represents a custom web application of type
 tosca.nodes.WebApplication.WordPress which is hosted on an Apache web server represented by the

apache node template. This hosting relationship is expressed via the host entry in the requirements

section of the **wordpress** node template. The **apache** node template, finally, is hosted on the

236 **web_server** compute node.

The database stack consists of the **wordpress_db**, the **mysql** and the **db_server** node templates. The **wordpress_db** node represents a custom database of type **tosca.nodes.Database.MySQL** which is hosted on a MySQL DBMS represented by the **mysql** node template. This node, in turn, is hosted on the

240 **db_server** compute node.

The **wordpress** node requires a connection to the **wordpress_db** node, since the WordPress application needs a database to store its data in. This relationship is established through the **database_endpoint** entry in the **requirements** section of the **wordpress** node template's declared node type. For configuring the WordPress web application, information about the database to connect to is required as input to the **configure** operation. Therefore, the input parameters are defined and values for them are retrieved from the properties and attributes of the **wordpress_db** node via the **get_property** and **get_attribute**

functions. In the above example, these inputs are defined at the interface-level and would be available to

248 all operations of the **Standard** interface (i.e., the **tosca.interfaces.node.lifecycle.Standard**

249 interface) within the wordpress node template and not just the configure operation.

250 **2.6 Using a custom script to establish a relationship in a template**

In previous examples, the template author did not have to think about explicit relationship types to be used to link a requirement of a node to another node of a model, nor did the template author have to think about special logic to establish those links. For example, the **host** requirement in previous examples just pointed to another node template and based on metadata in the corresponding node type definition the relationship type to be established is implicitly given.

In some cases, it might be necessary to provide special processing logic to be executed when
 establishing relationships between nodes at runtime. For example, when connecting the WordPress

application from previous examples to the MySQL database, it might be desired to apply custom

configuration logic in addition to that already implemented in the application node type. In such a case, it is possible for the template author to provide a custom script as implementation for an operation to be

261 executed at runtime as shown in the following example.

262 Example 7 - Providing a custom relationship script to establish a connection

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: Template for deploying a two-tier application on two servers.
topology_template:
    inputs:
        # omitted here for brevity
node_templates:
        wordpress:
        type: tosca.nodes.WebApplication.WordPress
        properties:
        # omitted here for brevity
        requirements:
```

```
- host: apache
      - database endpoint:
          node: wordpress db
          relationship: my custom database connection
 wordpress_db:
   type: tosca.nodes.Database.MySQL
    properties:
      # omitted here for the brevity
    requirements:
      - host: mysql
relationship_templates:
 my_custom_database_connection:
    type: ConnectsTo
    interfaces:
      Configure:
        pre configure source: scripts/wp db configure.sh
```

other resources not shown for this example ...

The node type definition for the wordpress node template is WordPress which declares the complete database_endpoint requirement definition. This database_endpoint declaration indicates it must be fulfilled by any node template that provides an Endpoint.Database Capability Type using a ConnectsTo relationship. The wordpress_db node template's underlying MySQL type definition indeed provides the Endpoint.Database Capability type. In this example however, no explicit relationship template is declared; therefore, TOSCA orchestrators would automatically create a ConnectsTo relationship to establish the link between the wordpress node and the wordpress_db node at runtime.

The **ConnectsTo** relationship (see 5.7.4) also provides a default **Configure** interface with operations that optionally get executed when the orchestrator establishes the relationship. In the above example, the author has provided the custom script **wp_db_configure.sh** to be executed for the operation called **pre_configure_source**. The script file is assumed to be located relative to the referencing service template such as a relative directory within the TOSCA Cloud Service Archive (CSAR) packaging format. This approach allows for conveniently hooking in custom behavior without having to define a completely new derived relationship type.

277 2.7 Using custom relationship types in a TOSCA template

In the previous section it was shown how custom behavior can be injected by specifying scripts inline in
the requirements section of node templates. When the same custom behavior is required in many
templates, it does make sense to define a new relationship type that encapsulates the custom behavior in
a re-usable way instead of repeating the same reference to a script (or even references to multiple
scripts) in many places.

283 Such a custom relationship type can then be used in templates as shown in the following example.

284 Example 8 - A web application Node Template requiring a custom database connection type

```
tosca definitions version: tosca simple yaml 1 0
description: Template for deploying a two-tier application on two servers.
topology_template:
  inputs:
    # omitted here for brevity
  node templates:
    wordpress:
      type: tosca.nodes.WebApplication.WordPress
      properties:
        # omitted here for brevity
      requirements:
        - host: apache
        - database endpoint:
            node: wordpress db
            relationship: my.types.WordpressDbConnection
    wordpress_db:
      type: tosca.nodes.Database.MySQL
      properties:
        # omitted here for the brevity
      requirements:
        - host: mysql
   # other resources not shown here ...
```

In the example above, a special relationship type my.types.WordpressDbConnection is specified for
 establishing the link between the wordpress node and the wordpress_db node through the use of the
 relationship (keyword) attribute in the database reference. It is assumed, that this special relationship
 type provides some extra behavior (e.g., an operation with a script) in addition to what a generic
 "connects to" relationship would provide. The definition of this custom relationship type is shown in the
 following section.

291 2.7.1 Definition of a custom relationship type

The following YAML snippet shows the definition of the custom relationship type used in the previous section. This type derives from the base "ConnectsTo" and overrides one operation defined by that base relationship type. For the **pre_configure_source** operation defined in the **Configure** interface of the ConnectsTo relationship type, a script implementation is provided. It is again assumed that the custom configure script is located at a location relative to the referencing service template, perhaps provided in some application packaging format (e.g., the TOSCA Cloud Service Archive (CSAR) format). 298 Example 9 - Defining a custom relationship type

```
tosca_definitions_version: tosca_simple_yaml_1_0
```

description: Definition of custom WordpressDbConnection relationship type

relationship_types:

```
my.types.WordpressDbConnection:
    derived_from: tosca.relationships.ConnectsTo
    interfaces:
        Configure:
            pre_configure_source: scripts/wp_db_configure.sh
```

In the above example, the **Configure** interface is the specified alias or shorthand name for the TOSCA interface type with the full name of **tosca.interfaces.relationship.Configure** which is defined in the appendix.

302 2.8 Defining generic dependencies between nodes in a template

In some cases, it can be necessary to define a generic dependency between two nodes in a template to influence orchestration behavior, i.e. to first have one node processed before another dependent node gets processed. This can be done by using the generic **dependency** requirement which is defined by the TOSCA Root Node Type and thus gets inherited by all other node types in TOSCA (see section 5.9.1).

307 Example 10 - Simple dependency relationship between two nodes

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: Template with a generic dependency between two nodes.
topology_template:
  inputs:
    # omitted here for brevity
  node_templates:
    my app:
      type: my.types.MyApplication
      properties:
        # omitted here for brevity
      requirements:
        - dependency: some service
    some service:
      type: some.nodetype.SomeService
      properties:
        # omitted here for brevity
```

As in previous examples, the relation that one node depends on another node is expressed in the requirements section using the built-in requirement named dependency that exists for all node types in TOSCA. Even if the creator of the MyApplication node type did not define a specific requirement for SomeService (similar to the database requirement in the example in section 2.6), the template author who knows that there is a timing dependency and can use the generic dependency requirement to

313 express that constraint using the very same syntax as used for all other references.

2.9 Describing abstract requirements for nodes and capabilities in a TOSCA template

316 In TOSCA templates, nodes are either:

- **Concrete**: meaning that they have a deployment and/or one or more implementation artifacts that are declared on the "create" operation of the node's Standard lifecycle interface, or they are
- Abstract: where the template describes the node type along with its required capabilities and properties that must be satisfied.

321

317

318

TOSCA Orchestrators, by default, when finding an abstract node in TOSCA Service Template during deployment will attempt to "select" a concrete implementation for the abstract node type that best matches and fulfills the requirements and property constraints the template author provided for that abstract node. The concrete implementation of the node could be provided by another TOSCA Service Template (perhaps located in a catalog or repository known to the TOSCA Orchestrator) or by an existing resource or service available within the target Cloud Provider's platform that the TOSCA Orchestrator already has knowledge of.

329

TOSCA supports two methods for template authors to express requirements for an abstract node within a
 TOSCA service template.

332

336

3331. Using a target node_filter: where a node template can describe a requirement (relationship) for334another node without including it in the topology. Instead, the node provides a node_filter to335describe the target node type along with its capabilities and property constrains

- Using an abstract node template: that describes the abstract node's type along with its property constraints and any requirements and capabilities it also exports. This first method you have already seen in examples from previous chapters where the Compute node is abstract and selectable by the TOSCA Orchestrator using the supplied Container and OperatingSystem capabilities property constraints.
- 342

These approaches allow architects and developers to create TOSCA service templates that are
 composable and can be reused by allowing flexible matching of one template's requirements to another's
 capabilities. Examples of both these approaches are shown below.

346

The following section describe how a user can define a requirement for an orchestrator to select an
 implementation and replace a node. For more details on how an orchestrator may perform matching and
 select a node from it's catalog(s) you may look at section 14 of the specification.

2.9.1 Using a node_filter to define hosting infrastructure requirements for a software

Using TOSCA, it is possible to define only the software components of an application in a template and just express constrained requirements against the hosting infrastructure. At deployment time, the provider can then do a late binding and dynamically allocate or assign the required hosting infrastructure andplace software components on top.

This example shows how a single software component (i.e., the mysql node template) can define its **host** requirements that the TOSCA Orchestrator and provider will use to select or allocate an appropriate host **Compute** node by using matching criteria provided on a **node filter**.

359 Example 11 - An abstract "host" requirement using a node filter

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: Template with requirements against hosting infrastructure.
topology_template:
  inputs:
    # omitted here for brevity
  node_templates:
    mysql:
      type: tosca.nodes.DBMS.MySQL
      properties:
        # omitted here for brevity
      requirements:
        - host:
            node filter:
              capabilities:
                # Constraints for selecting "host" (Container Capability)
                - host:
                    properties:
                       - num cpus: { in range: [ 1, 4 ] }
                       - mem_size: { greater_or_equal: 2 GB }
                # Constraints for selecting "os" (OperatingSystem Capability)
                - os:
                    properties:
                       - architecture: { equal: x86 64 }
                       - type: linux
                       - distribution: ubuntu
```

In the example above, the mysql component contains a host requirement for a node of type Compute which it inherits from its parent DBMS node type definition; however, there is no declaration or reference to any node template of type Compute. Instead, the mysql node template augments the abstract "host" requirement with a node_filter which contains additional selection criteria (in the form of property constraints that the provider must use when selecting or allocating a host Compute node.

Some of the constraints shown above narrow down the boundaries of allowed values for certain properties such as mem_size or num_cpus for the "host" capability by means of qualifier functions such as greater_or_equal. Other constraints, express specific values such as for the architecture or distribution properties of the "os" capability which will require the provider to find a precise match. Note that when no qualifier function is provided for a property (filter), such as for the **distribution** property, it is interpreted to mean the **equal** operator as shown on the **architecture** property.

2.9.2 Using an abstract node template to define infrastructure requirements for software

373 This previous approach works well if no other component (i.e., another node template) other than mysql

node template wants to reference the same **Compute** node the orchestrator would instantiate. However,

perhaps another component wants to also be deployed on the same host, yet still allow the flexible
 matching achieved using a node-filter. The alternative to the above approach is to create an abstract

377 node template that represents the **Compute** node in the topology as follows:

378 Example 12 - An abstract Compute node template with a node filter

```
tosca definitions version: tosca simple yaml 1 0
description: Template with requirements against hosting infrastructure.
topology_template:
  inputs:
    # omitted here for brevity
  node templates:
    mysql:
      type: tosca.nodes.DBMS.MySQL
      properties:
        # omitted here for brevity
      requirements:
        - host: mysql compute
    # Abstract node template (placeholder) to be selected by provider
    mysql compute:
      type: Compute
      node filter:
        capabilities:
          - host:
              properties:
                num_cpus: { equal: 2 }
                mem_size: { greater_or_equal: 2 GB }
          - os:
              properties:
                architecture: { equal: x86_64 }
                type: linux
                distribution: ubuntu
```

As you can see the resulting mysql_compute node template looks very much like the "hello world"
 template as shown in Chapter 2.1 (where the Compute node template was abstract), but this one also
 allows the TOSCA orchestrator more flexibility when "selecting" a host Compute node by providing flexible
 constraints for properties like mem_size.

As we proceed, you will see that TOSCA provides many normative node types like **Compute** for commonly found services (e.g., **BlockStorage**, **WebServer**, **Network**, etc.). When these TOSCA normative node types are used in your application's topology they are always assumed to be "selectable" by TOSCA Orchestrators which work with target infrastructure providers to find or allocate the best match for them based upon your application's requirements and constraints.

2.9.3 Using a node_filter to define requirements on a database for an application

In the same way requirements can be defined on the hosting infrastructure (as shown above) for an
application, it is possible to express requirements against application or middleware components such as
a database that is not defined in the same template. The provider may then allocate a database by any
means, (e.g. using a database-as-a-service solution).

394 Example 13 - An abstract database requirement using a node filter

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: Template with a TOSCA Orchestrator selectable database requirement
using a node filter.
topology_template:
  inputs:
    # omitted here for brevity
  node_templates:
    my_app:
      type: my.types.MyApplication
      properties:
        admin user: { get input: admin username }
        admin password: { get input: admin password }
        db endpoint url: { get property: [SELF, database endpoint, url path ] }
      requirements:
        - database endpoint:
            node: my.types.nodes.MyDatabase
            node filter:
              properties:
                - db version: { greater or equal: 5.5 }
```

In the example above, the application my_app requires a database node of type MyDatabase which has a
 db_version property value of greater_or_equal to the value 5.5.

This example also shows how the **get_property** intrinsic function can be used to retrieve the **url_path** property from the database node that will be selected by the provider and connected to **my_app** at runtime due to fulfillment of the **database_endpoint** requirement. To locate the property, the get_property's first argument is set to the keyword **SELF** which indicates the property is being referenced from something in the node itself. The second parameter is the name of the requirement named **database_endpoint** which contains the property we are looking for. The last argument is the name of the property itself (i.e., **url path**) which contains the value we want to retrieve and assign to **db endpoint url**.

The alternative representation, which includes a node template in the topology for database that is still selectable by the TOSCA orchestrator for the above example, is as follows: 406 Example 14 - An abstract database node template

```
tosca definitions version: tosca simple yaml 1 0
description: Template with a TOSCA Orchestrator selectable database using node
template.
topology_template:
  inputs:
    # omitted here for brevity
  node templates:
    my app:
      type: my.types.MyApplication
      properties:
        admin_user: { get_input: admin_username }
        admin_password: { get_input: admin_password }
        db_endpoint_url: { get_property: [SELF, database_endpoint, url_path ] }
      requirements:
        - database endpoint: my abstract database
    my_abstract_database:
      type: my.types.nodes.MyDatabase
      properties:
        - db version: { greater or equal: 5.5 }
```

407 2.10 Using node template substitution for model composition

From an application perspective, it is often not necessary or desired to dive into platform details, but the platform/runtime for an application is abstracted. In such cases, the template for an application can use generic representations of platform components. The details for such platform components, such as the underlying hosting infrastructure at its configuration, can then be defined in separate template files that can be used for substituting the more abstract representations in the application level template file.

413 2.10.1 Understanding node template instantiation through a TOSCA 414 Orchestrator

When a topology template is instantiated by a TOSCA Orchestrator, the orchestrator has to look for realizations of abstract nodes in the topology template according to the node types specified for each abstract node template. Such realizations can either be node types that include the appropriate implementation artifacts and deployment artifacts that can be used by the orchestrator to bring to life the real-world resource modeled by a node template. Alternatively, separate topology templates may be annotated as being suitable for realizing a node template in the top-level topology template.

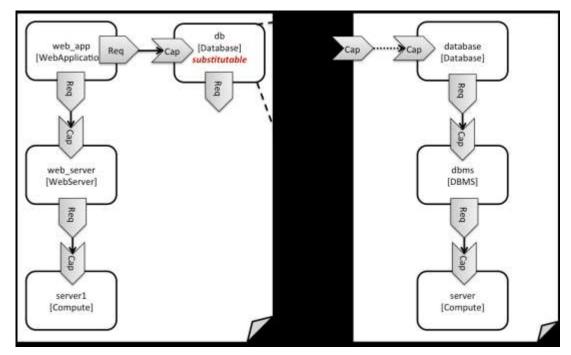
In the latter case, a TOSCA Orchestrator will use additional substitution mapping information provided as
part of the substituting topology templates to derive how the substituted part gets "wired" into the overall
deployment, for example, how capabilities of a node template in the top-level topology template get
bound to capabilities of node templates in the substituting topology template.

- 426
- Thus, in cases where no "normal" node type implementation is available, or the node type corresponds to a whole subsystem that cannot be implemented as a single node, additional topology templates can be used for filling in more abstract placeholders in top level application templates.

430 2.10.2 Definition of the top-level service template

431 The following sample defines a web application **web_app** connected to a database **db**. In this example, the

- 432 complete hosting stack for the application is defined within the same topology template: the web
- 433 application is hosted on a web server **web_server**, which in turn is installed (hosted) on a compute node 434 server.
- The hosting stack for the database **db**, in contrast, is not defined within the same file but only the
- 436 database is represented as a node template of type **tosca.nodes.Database**. The underlying hosting
- 437 stack for the database is defined in a separate template file, which is shown later in this section. Within
- 438 the current template, only a number of properties (**user**, **password**, **name**) are assigned to the database
- 439 using hardcoded values in this simple example.



440



Figure 1: Using template substitution to implement a database tier

When a node template is to be substituted by another service template, this has to be indicated to an orchestrator by means of a special "*substitutable*" directive. This directive causes, for example, special processing behavior when validating the left-hand service template in Figure 1. The hosting requirement of the **db** node template is not bound to any capability defined within the service template, which would normally cause a validation error. When the "*substitutable*" directive is present, the orchestrator will however first try to perform substitution of the respective node template and after that validate if all mandatory requirements of all nodes in the resulting graph are fulfilled.

449

450 Note that in contrast to the use case described in section 2.9.2 (where a database was abstractly referred 451 to in the **requirements** section of a node and the database itself was not represented as a node

- template), the approach shown here allows for some additional modeling capabilities in cases where this
 is required.
- 454
- 455 For example, if multiple components need to use the same database (or any other sub-system of the TOSCA-Simple-Profile-YAML-v1.2-os 17 January

- 456 overall service), this can be expressed by means of normal relations between node templates, whereas
- 457 such modeling would not be possible in **requirements** sections of disjoint node templates.
- 458 Example 15 Referencing an abstract database node template

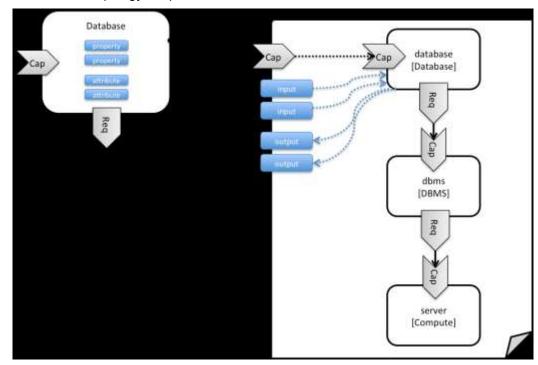
```
tosca_definitions_version: tosca_simple_yaml_1_0
topology template:
  description: Template of an application connecting to a database.
  node_templates:
    web_app:
      type: tosca.nodes.WebApplication.MyWebApp
      requirements:
        - host: web server
        - database endpoint: db
    web_server:
      type: tosca.nodes.WebServer
      requirements:
        - host: server
    server:
      type: tosca.nodes.Compute
      # details omitted for brevity
    db:
      # This node is abstract (no Deployment or Implementation artifacts on
create)
      # and can be substituted with a topology provided by another template
      # that exports a Database type's capabilities.
      type: tosca.nodes.Database
      properties:
        user: my_db_user
        password: secret
        name: my_db_name
```

459 2.10.3 Definition of the database stack in a service template

The following sample defines a template for a database including its complete hosting stack, i.e. the template includes a **database** node template, a template for the database management system (**dbms**) hosting the database, as well as a computer node **server** on which the DBMS is installed.

This service template can be used standalone for deploying just a database and its hosting stack. In the context of the current use case, though, this template can also substitute the database node template in the previous snippet and thus fill in the details of how to deploy the database.

- In order to enable such a substitution, an additional metadata section substitution_mappings is added to the topology template to tell a TOSCA Orchestrator how exactly the topology template will fit into the context where it gets used. For example, requirements or capabilities of the node that gets substituted by the topology template have to be mapped to requirements or capabilities of internal node templates for allow for a proper wiring of the resulting overall graph of node templates.
- 471 In short, the **substitution_mappings** section provides the following information:
- 472 1. It defines what node templates, i.e. node templates of which type, can be substituted by the473 topology template.
- 474
 475
 475
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
 476
- 477 3. It defines how requirements of the substituted node (or the requirements defined by the node type
 478 of the substituted node template, respectively) are bound to requirements of node templates
 479 defined in the topology template.



480 481

Figure 2: Substitution mappings

The **substitution_mappings** section in the sample below denotes that this topology template can be used for substituting node templates of type **tosca.nodes.Database**. It further denotes that the **database endpoint** capability of the substituted node gets fulfilled by the **database endpoint**

- 485 capability of the database node contained in the topology template.
- 486 Example 16 Using substitution mappings to export a database implementation

tosca_definitions_version: tosca_simple_yaml_1_0

topology_template:

description: Template of a database including its hosting stack.

inputs:

```
db user:
    type: string
 db password:
    type: string
 # other inputs omitted for brevity
substitution_mappings:
 node_type: tosca.nodes.Database
 capabilities:
    database endpoint: [ database, database endpoint ]
node templates:
 database:
   type: tosca.nodes.Database
    properties:
      user: { get input: db user }
      # other properties omitted for brevity
    requirements:
      - host: dbms
 dbms:
   type: tosca.nodes.DBMS
   # details omitted for brevity
 server:
   type: tosca.nodes.Compute
   # details omitted for brevity
```

487 Note that the substitution_mappings section does not define any mappings for requirements of the 488 Database node type, since all requirements are fulfilled by other nodes templates in the current topology 489 template. In cases where a requirement of a substituted node is bound in the top-level service template 490 as well as in the substituting topology template, a TOSCA Orchestrator should raise a validation error.

Further note that no mappings for properties or attributes of the substituted node are defined. Instead, the inputs and outputs defined by the topology template are mapped to the appropriate properties and attributes or the substituted node. If there are more inputs than the substituted node has properties, default values must be defined for those inputs, since no values can be assigned through properties in a substitution case.

496 2.11 Using node template substitution for chaining subsystems

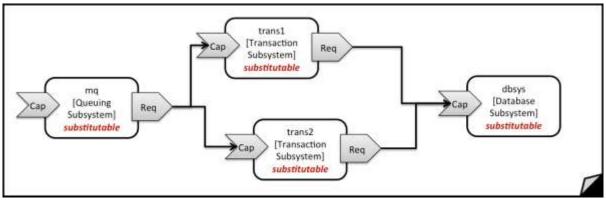
A common use case when providing an end-to-end service is to define a chain of several subsystems that
together implement the overall service. Those subsystems are typically defined as separate service
templates to (1) keep the complexity of the end-to-end service template at a manageable level and to (2)
allow for the re-use of the respective subsystem templates in many different contexts. The type of
subsystems may be specific to the targeted workload, application domain, or custom use case. For
example, a company or a certain industry might define a subsystem type for company- or industry specific

503 data processing and then use that subsystem type for various end-user services. In addition, there might 504 be generic subsystem types like a database subsystem that are applicable to a wide range of use cases.

505 2.11.1 Defining the overall subsystem chain

506 Figure 3 shows the chaining of three subsystem types – a message queuing subsystem, a transaction

- 507 processing subsystem, and a databank subsystem that support, for example, an online booking
- application. On the front end, this chain provides a capability of receiving messages for handling in the
- 509 message queuing subsystem. The message queuing subsystem in turn requires a number of receivers, 510 which in the current example are two transaction processing subsystems. The two instances of the
- 511 transaction processing subsystem might be deployed on two different hosting infrastructures or
- 512 datacenters for high-availability reasons. The transaction processing subsystems finally require a
- 513 database subsystem for accessing and storing application specific data. The database subsystem in the
- 514 backend does not require any further component and is therefore the end of the chain in this example.



515 516

Figure 3: Chaining of subsystems in a service template

All of the node templates in the service template shown above are abstract and considered substitutable where each can be treated as their own subsystem; therefore, when instantiating the overall service, the orchestrator would realize each substitutable node template using other TOSCA service templates. These service templates would include more nodes and relationships that include the details for each subsystem. A simplified version of a TOSCA service template for the overall service is given in the following listing.

523

```
524 Example 17 - Declaring a transaction subsystem as a chain of substitutable node templates
```

```
requirements:
    - receiver: trans1
    - receiver: trans2
trans1:
  type: example.TransactionSubsystem
  properties:
    mq_service_ip: { get_attribute: [ mq, service_ip ] }
    receiver port: 8080
  capabilities:
    message receiver:
      # details omitted for brevity
  requirements:
    - database endpoint: dbsys
trans2:
  type: example.TransactionSubsystem
  properties:
    mq_service_ip: { get_attribute: [ mq, service_ip ] }
    receiver_port: 8080
  capabilities:
    message_receiver:
      # details omitted for brevity
  requirements:
    - database endpoint: dbsys
dbsys:
  type: example.DatabaseSubsystem
  properties:
    # properties omitted for brevity
  capabilities:
    database endpoint:
      # details omitted for brevity
```

```
525
```

- 526 As can be seen in the example above, the subsystems are chained to each other by binding requirements
- 527 of one subsystem node template to other subsystem node templates that provide the respective 528 capabilities. For example, the **receiver** requirement of the message queuing subsystem node template
- 529 mg is bound to transaction processing subsystem node templates trans1 and trans2.

530 Subsystems can be parameterized by providing properties. In the listing above, for example, the IP

- address of the message queuing server is provided as property **mq_service_ip** to the transaction
- processing subsystems and the desired port for receiving messages is specified by means of the
- 533 **receiver_port** property.

534 If attributes of the instantiated subsystems need to be obtained, this would be possible by using the 535 get attribute intrinsic function on the respective subsystem node templates.

536 2.11.2 Defining a subsystem (node) type

537 The types of subsystems that are required for a certain end-to-end service are defined as TOSCA node

types as shown in the following example. Node templates of those node types can then be used in the end-to-end service template to define subsystems to be instantiated and chained for establishing the end-

540 to-end service.

541 The realization of the defined node type will be given in the form of a whole separate service template as 542 outlined in the following section.

- 543
- 544 Example 18 Defining a TransactionSubsystem node type

```
tosca definitions version: tosca simple yaml 1 0
node types:
  example.TransactionSubsystem:
    properties:
      mq_service_ip:
        type: string
      receiver_port:
        type: integer
    attributes:
      receiver ip:
        type: string
      receiver_port:
        type: integer
    capabilities:
      message receiver: tosca.capabilities.Endpoint
    requirements:
      - database endpoint: tosca.capabilities.Endpoint.Database
```

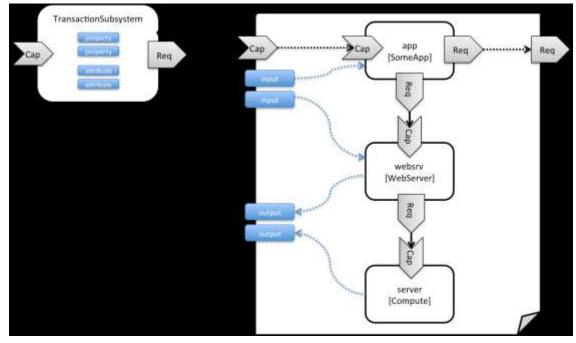
545

546 Configuration parameters that would be allowed for customizing the instantiation of any subsystem are 547 defined as properties of the node type. In the current example, those are the properties **mq_service_ip** 548 and **receiver_port** that had been used in the end-to-end service template in section 2.11.1.

549 Observable attributes of the resulting subsystem instances are defined as attributes of the node type. In 550 the current case, those are the IP address of the message receiver as well as the actually allocated port 551 of the message receiver endpoint.

552 2.11.3 Defining the details of a subsystem

The details of a subsystem, i.e. the software components and their hosting infrastructure, are defined as node templates and relationships in a service template. By means of substitution mappings that have been introduced in section 2.10.2, the service template is annotated to indicate to an orchestrator that it can be used as realization of a node template of certain type, as well as how characteristics of the node type are mapped to internal elements of the service template. 558



559 560

Figure 4: Defining subsystem details in a service template

Figure 1 illustrates how a transaction processing subsystem as outlined in the previous section could be
 defined in a service template. In this example, it simply consists of a custom application app of type
 SomeApp that is hosted on a web server websrv, which in turn is running on a compute node.

SolueApp that is nosted on a web server websity, which in turn is furning on a compute node.

564 The application named **app** provides a capability to receive messages, which is bound to the

565 message_receiver capability of the substitutable node type. It further requires access to a database, so 566 the application's database_endpoint requirement is mapped to the database_endpoint requirement of 567 the TransactionSubsystem node type.

568 Properties of the TransactionSubsystem node type are used to customize the instantiation of a

subsystem. Those properties can be mapped to any node template for which the author of the subsystem
 service template wants to expose configurability. In the current example, the application app and the web
 server middleware websrv get configured through properties of the TransactionSubsystem node type.
 All properties of that node type are defined as inputs of the service template. The input parameters in
 turn get mapped to node templates by means of get_input function calls in the respective sections of

574 the service template.

575 Similarly, attributes of the whole subsystem can be obtained from attributes of particular node templates. 576 In the current example, attributes of the web server and the hosting compute node will be exposed as

577 subsystem attributes. All exposed attributes that are defined as attributes of the substitutable

578 **TransactionSubsystem** node type are defined as outputs of the subsystem service template.

579 An outline of the subsystem service template is shown in the listing below. Note that this service template 580 could be used for stand-alone deployment of a transaction processing system as well, i.e. it is not

- restricted just for use in substitution scenarios. Only the presence of the **substitution_mappings**
- 582 metadata section in the **topology_template** enables the service template for substitution use cases.
- 583
- 584 Example 19 Implementation of a TransactionSubsytem node type using substitution mappings

tosca_definitions_version: tosca_simple_yaml_1_0

```
topology_template:
  description: Template of a database including its hosting stack.
  inputs:
    mq_service_ip:
      type: string
      description: IP address of the message queuing server to receive messages
from
    receiver port:
      type: string
      description: Port to be used for receiving messages
    # other inputs omitted for brevity
  substitution mappings:
    node_type: example.TransactionSubsystem
    capabilities:
      message_receiver: [ app, message_receiver ]
    requirements:
      database_endpoint: [ app, database ]
  node_templates:
    app:
      type: example.SomeApp
      properties:
        # properties omitted for brevity
      capabilities:
        message receiver:
          properties:
            service_ip: { get_input: mq_service_ip }
            # other properties omitted for brevity
      requirements:
        - database:
            # details omitted for brevity
        - host: websrv
    websrv:
      type: tosca.nodes.WebServer
      properties:
        # properties omitted for brevity
      capabilities:
        data_endpoint:
```

```
properties:
    port_name: { get_input: receiver_port }
    # other properties omitted for brevity
    requirements:
        - host: server
    server:
    type: tosca.nodes.Compute
    # details omitted for brevity
outputs:
    receiver_ip:
    description: private IP address of the message receiver application
    value: { get_attribute: [ server, private_address ] }
    receiver_port:
    description: Port of the message receiver endpoint
    value: { get_attribute: [ app, app endpoint, port ] }
```

585 2.12 Grouping node templates

In designing applications composed of several interdependent software components (or nodes) it is often
desirable to manage these components as a named group. This can provide an effective way of
associating policies (e.g., scaling, placement, security or other) that orchestration tools can apply to all
the components of group during deployment or during other lifecycle stages.

590 In many realistic scenarios it is desirable to include scaling capabilities into an application to be able to

react on load variations at runtime. The example below shows the definition of a scaling web server stack,
where a variable number of servers with apache installed on them can exist, depending on the load on
the servers.

594 Example 20 - Grouping Node Templates for possible policy application

```
requirements:
    - host: server
server:
    type: tosca.nodes.Compute
    # details omitted for brevity
groups:
webserver_group:
    type: tosca.groups.Root
```

members: [apache, server]

The example first of all uses the concept of grouping to express which components (node templates) need to be scaled as a unit – i.e. the compute nodes and the software on-top of each compute node. This is done by defining the **webserver_group** in the **groups** section of the template and by adding both the **apache** node template and the **server** node template as a member to the group.

599 Furthermore, a scaling policy is defined for the group to express that the group as a whole (i.e. pairs of 600 **server** node and the **apache** component installed on top) should scale up or down under certain 601 conditions.

In cases where no explicit binding between software components and their hosting compute resources is
 defined in a template, but only requirements are defined as has been shown in section 2.9, a provider
 could decide to place software components on the same host if their hosting requirements match, or to
 place them onto different hosts.

606 It is often desired, though, to influence placement at deployment time to make sure components get
 607 collocation or anti-collocated. This can be expressed via grouping and policies as shown in the example
 608 below.

609 Example 21 - Grouping nodes for anti-colocation policy application

```
tosca_definitions_version: tosca_simple_yaml_1_0
```

description: Template hosting requirements and placement policy.

```
topology_template:
    inputs:
    # omitted here for brevity
```

```
node_templates:
```

```
wordpress_server:
```

```
type: tosca.nodes.WebServer
```

```
properties:
```

```
# omitted here for brevity
```

```
requirements:
```

```
- host:
```

```
# Find a Compute node that fulfills these additional filter reqs.
          node_filter:
            capabilities:
              - host:
                  properties:
                    - mem_size: { greater_or_equal: 512 MB }
                    - disk_size: { greater_or_equal: 2 GB }
              - os:
                  properties:
                    - architecture: x86 64
                    - type: linux
 mysql:
    type: tosca.nodes.DBMS.MySQL
    properties:
      # omitted here for brevity
    requirements:
      - host:
          node: tosca.nodes.Compute
          node_filter:
            capabilities:
              - host:
                  properties:
                    - disk_size: { greater_or_equal: 1 GB }
              - os:
                  properties:
                    - architecture: x86_64
                    - type: linux
groups:
 my_co_location_group:
    type: tosca.groups.Root
    members: [ wordpress_server, mysql ]
policies:
  - my_anti_collocation_policy:
      type: my.policies.anticolocateion
      targets: [ my_co_location_group ]
      # For this example, specific policy definitions are considered
      # domain specific and are not included here
```

- 610 In the example above, both software components **wordpress_server** and **mysq1** have similar hosting
- 611 requirements. Therefore, a provider could decide to put both on the same server as long as both their
- 612 respective requirements can be fulfilled. By defining a group of the two components and attaching an anti-
- 613 collocation policy to the group it can be made sure, though, that both components are put onto different
- 614 hosts at deployment time.

615 2.13 Using YAML Macros to simplify templates

The YAML 1.2 specification allows for defining of aliases, which allow for authoring a block of YAML (or node) once and indicating it is an "anchor" and then referencing it elsewhere in the same document as an "alias". Effectively, YAML parsers treat this as a "macro" and copy the anchor block's code to wherever it is referenced. Use of this feature is especially helpful when authoring TOSCA Service Templates where similar definitions and property settings may be repeated multiple times when describing a multi-tier application.

622

For example, an application that has a web server and database (i.e., a two-tier application) may be described using two **Compute** nodes (one to host the web server and another to host the database). The author may want both Compute nodes to be instantiated with similar properties such as operating system, distribution, version, etc.

To accomplish this, the author would describe the reusable properties using a named anchor in the **"ds1 definitions**" section of the TOSCA Service Template and reference the anchor name as an alias

- 629 in any **Compute** node templates where these properties may need to be reused. For example:
- 630 Example 22 Using YAML anchors in TOSCA templates

tosca_definitions_version: tosca_simple_yaml_1_0

description: >

TOSCA simple profile that just defines a YAML macro for commonly reused Compute properties.

```
dsl_definitions:
```

```
my_compute_node_props: &my_compute_node_props
  disk_size: 10 GB
  num_cpus: 1
  mem_size: 2 GB
```

```
topology_template:
```

node_templates:

```
my_server:
```

type: Compute

capabilities:

- host:
 - properties: *my_compute_node_props

```
my_database:
    type: Compute
```

```
capabilities:
    - host:
    properties: *my_compute_node_props
```

631 2.14 Passing information as inputs to Nodes and Relationships

632 It is possible for type and template authors to declare input variables within an **inputs** block on interfaces 633 to nodes or relationships in order to pass along information needed by their operations (scripts). These 634 declarations can be scoped such as to make these variable values available to all operations on a node 635 or relationships interfaces or to individual operations. TOSCA orchestrators will make these values 636 available as environment variables within the execution environments in which the scripts associated with 637 lifecycle operations are run.

638 **2.14.1 Example: declaring input variables for all operations on a single** 639 **interface**

```
node_templates:
wordpress:
  type: tosca.nodes.WebApplication.WordPress
  requirements:
    ...
    . database_endpoint: mysql_database
  interfaces:
    Standard:
    inputs:
    wp_db_port: { get_property: [ SELF, database_endpoint, port ] }
```

640 2.14.2 Example: declaring input variables for a single operation

```
node_templates:
wordpress:
type: tosca.nodes.WebApplication.WordPress
requirements:
...
- database_endpoint: mysql_database
interfaces:
Standard:
create: wordpress_install.sh
configure:
implementation: wordpress_configure.sh
inputs:
wp_db_port: { get_property: [ SELF, database_endpoint, port ] }
```

- 641 In the case where an input variable name is defined at more than one scope within the same interfaces
- section of a node or template definition, the lowest (or innermost) scoped declaration would override
- those declared at higher (or more outer) levels of the definition.

644 **2.14.3 Example: setting output variables to an attribute**

```
node_templates:
    frontend:
    type: MyTypes.SomeNodeType
    attributes:
        url: { get_operation_output: [ SELF, Standard, create, generated_url ] }
    interfaces:
        Standard:
            create:
            implementation: scripts/frontend/create.sh
```

645

646 In this example, the Standard create operation exposes / exports an environment variable named

647 **"generated_url"** attribute which will be assigned to the WordPress node's **url** attribute.

648 2.14.4 Example: passing output variables between operations

649 In this example, the **Standard** lifecycle's **create** operation exposes / exports an environment variable 650 named "**data_dir**" which will be passed as an input to the **Standard** lifecycle's **configure** operation.

651 2.15 Topology Template Model versus Instance Model

652 A TOSCA service template contains a **topology template**, which models the components of an 653 application, their relationships and dependencies (a.k.a., a topology model) that get interpreted and 654 instantiated by TOSCA Orchestrators. The actual node and relationship instances that are created 655 represent a set of resources distinct from the template itself, called a **topology instance (model)**. The 656 direction of this specification is to provide access to the instances of these resources for management 657 and operational control by external administrators. This model can also be accessed by an orchestration engine during deployment - i.e. during the actual process of instantiating the template in an incremental 658 fashion, That is, the orchestrator can choose the order of resources to instantiate (i.e., establishing a 659 660 partial set of node and relationship instances) and have the ability, as they are being created, to access 661 them in order to facilitate instantiating the remaining resources of the complete topology template.

662 2.16 Using attributes implicitly reflected from properties

Most entity types in TOSCA (e.g., Node, Relationship, Capability Types, etc.) have property definitions,

which allow template authors to set the values for as inputs when these entities are instantiated by an
orchestrator. These property values are considered to reflect the desired state of the entity by the author.
Once instantiated, the actual values for these properties on the realized (instantiated) entity are
obtainable via attributes on the entity with the same name as the corresponding property.

668 In other words, TOSCA orchestrators will automatically reflect (i.e., make available) any property defined 669 on an entity making it available as an attribute of the entity with the same name as the property.

670

Use of this feature is shown in the example below where a source node named my_client, of type ClientNode, requires a connection to another node named my_server of type ServerNode. As you can see, the ServerNode type defines a property named notification_port which defines a dedicated port number which instances of my_client may use to post asynchronous notifications to it during runtime. In this case, the TOSCA Simple Profile assures that the notification_port property is implicitly reflected as an attribute in the my_server node (also with the name notification_port) when its node template is instantiated.

- 678
- 679 Example 23 Properties reflected as attributes

tosca_definitions_version: tosca_simple_yaml_1_0

description: >

TOSCA simple profile that shows how the (notification_port) property is reflected as an attribute and can be referenced elsewhere.

node_types:

ServerNode:

derived_from: SoftwareComponent

properties:

notification_port:

type: integer

capabilities:

omitted here for brevity

```
ClientNode:
```

derived_from: SoftwareComponent

properties:

omitted here for brevity

requirements:

 server: capability: Endpoint

```
node: ServerNode
```

relationship: ConnectsTo

```
topology_template:
  node_templates:
   my server:
      type: ServerNode
      properties:
        notification_port: 8000
   my client:
      type: ClientNode
      requirements:
        - server:
            node: my server
            relationship: my_connection
  relationship_templates:
   my connection:
      type: ConnectsTo
      interfaces:
        Configure:
          inputs:
            targ_notify_port: { get_attribute: [ TARGET, notification_port ] }
            # other operation definitions omitted here for brevity
```

680

581 Specifically, the above example shows that the **ClientNode** type needs the **notification_port** value 582 anytime a node of **ServerType** is connected to it using the **ConnectsTo** relationship in order to make it 583 available to its **Configure** operations (scripts). It does this by using the **get_attribute** function to 584 retrieve the **notification_port** attribute from the **TARGET** node of the **ConnectsTo** relationship (which is 585 a node of type **ServerNode**) and assigning it to an environment variable named **targ_notify_port**.

686

687 It should be noted that the actual port value of the **notification_port** attribute may or may not be the

value **8000** as requested on the property; therefore, any node that is dependent on knowing its actual

689 "runtime" value would use the **get_attribute** function instead of the **get_property** function.

3 TOSCA Simple Profile definitions in YAML

691 Except for the examples, this section is **normative** and describes all of the YAML grammar, definitions 692 and block structure for all keys and mappings that are defined for the TOSCA Version 1.2 Simple Profile 693 specification that are needed to describe a TOSCA Service Template (in YAML).

694 **3.1 TOSCA Namespace URI and alias**

The following TOSCA Namespace URI alias and TOSCA Namespace Alias are reserved values which SHALL be used when identifying the TOSCA Simple Profile version 1.2 specification.

Namespace Alias	Namespace URI	Specification Description
tosca_simple_yaml_1_2	http://docs.oasis- open.org/tosca/ns/simple/yaml/1.2	The TOSCA Simple Profile v1.2 (YAML) target namespace and namespace alias.

697 **3.1.1 TOSCA Namespace prefix**

698 The following TOSCA Namespace prefix is a reserved value and SHALL be used to reference the default 699 TOSCA Namespace URI as declared in TOSCA Service Templates.

Namespace Prefix	Specification Description
tosca	The reserved TOSCA Simple Profile Specification prefix that can be associated with the default TOSCA Namespace URI

700 3.1.2 TOSCA Namespacing in TOSCA Service Templates

- In the TOSCA Simple Profile, TOSCA Service Templates MUST always have, as the first line of YAML,
 the keyword "tosca_definitions_version" with an associated TOSCA Namespace Alias value. This
 single line accomplishes the following:
- Establishes the TOSCA Simple Profile Specification version whose grammar MUST be used to parse and interpret the contents for the remainder of the TOSCA Service Template.
- Establishes the default TOSCA Namespace URI and Namespace Prefix for all types found in the document that are not explicitly namespaced.
- Automatically imports (without the use of an explicit import statement) the normative type
 definitions (e.g., Node, Relationship, Capability, Artifact, etc.) that are associated with the TOSCA
 Simple Profile Specification the TOSCA Namespace Alias value identifies.
- Associates the TOSCA Namespace URI and Namespace Prefix to the automatically imported
 TOSCA type definitions.

713 3.1.3 Rules to avoid namespace collisions

TOSCA Simple Profiles allows template authors to declare their own types and templates and assign
them simple names with no apparent namespaces. Since TOSCA Service Templates can import other
service templates to introduce new types and topologies of templates that can be used to provide
concrete implementations (or substitute) for abstract nodes. Rules are needed so that TOSCA
Orchestrators know how to avoid collisions and apply their own namespaces when import and nesting
occur.

720 3.1.3.1 Additional Requirements

• The URI value "http://docs.oasis-open.org/tosca", as well as all (path) extensions to it, SHALL be reserved for TOSCA approved specifications and work. That means Service Templates that do

723		not originate from a TOSCA approved work product MUST NOT use it, in any form, when
724		declaring a (default) Namespace.
725	•	Since TOSCA Service Templates can import (or substitute in) other Service Templates, TOSCA
726		Orchestrators and tooling will encounter the "tosca_definitions_version" statement for each
727		imported template. In these cases, the following additional requirements apply:
728		 Imported type definitions with the same Namespace URI, local name and version SHALL
729		be equivalent.
730		 If different values of the "tosca_definitions_version" are encountered, their
731		corresponding type definitions MUST be uniquely identifiable using their corresponding
732		Namespace URI using a different Namespace prefix.
733	•	Duplicate local names (i.e., within the same Service Template SHALL be considered an error.
734		These include, but are not limited to duplicate names found for the following definitions:
735		 Repositories (repositories)
736		 Data Types (data_types)
737		 Node Types (node_types)
738		 Relationship Types (relationship_types)
739		 Capability Types (capability_types)
740		 Artifact Types (artifact_types)
741		 Interface Types (interface_types)
742	•	Duplicate Template names within a Service Template's Topology Template SHALL be considered
743		an error. These include, but are not limited to duplicate names found for the following template
744		types:
745		 Node Templates (node_templates)
746		 Relationship Templates (relationship_templates)
747		 Inputs (inputs)
748		 Outputs (outputs)
749	•	Duplicate names for the following keynames within Types or Templates SHALL be considered an
750		error. These include, but are not limited to duplicate names found for the following keynames:
751		 Properties (properties)
752		 Attributes (attributes)
753		 Artifacts (artifacts)
754		 Requirements (requirements)
755		 Capabilities (capabilities)
756		 Interfaces (interfaces)
757		 Policies (policies)
758		 Groups (groups)

759 3.2 Using Namespaces

- As of TOSCA version 1.2, Service template authors may declare a namespace within a Service Template that would be used as the default namespace for any types (e.g., Node Type, Relationship Type, Data
- 762 Type, etc.) defined within the same Service template.
- 763

Specifically, a Service Template's namespace declaration's URI would be used to form a unique, fully
qualified Type name when combined with the locally defined, unqualified name of any Type in the same
Service Template. The resulatant, fully qualified Type name would be used by TOSCA Orchestrators,
Processors and tooling when that Service Template was imported into another Service Template to avoid
Type name collision.

770 3.2.1.1.1.1 Example – Importing a Service Template and Namespaces

- For example, let say we have two Service Templates, A and B, both of which define Types and a 771
- 772 Namespace. Service Template B contains a Node Type definition for "MyNode" and declares its (default) Namespace to be "http://companyB.com/service/namespace/":
- 773

774 Service Template B

775

```
tosca definitions version: tosca simple yaml 1 2
description: Service Template B
namespace: http://companyB.com/service/namespace/
```

```
node_types:
 MyNode:
   derived_from: SoftwareComponent
   properties:
      # omitted here for brevity
    capabilities:
      # omitted here for brevity
```

776

- Service Template A has its own, completely different, Node Type definition also named "MyNode". 777
- 778

```
779
      Service Template A
```

780

```
tosca definitions version: tosca simple yaml 1 2
description: Service Template A
namespace: http://companyA.com/product/ns/
```

imports:

```
- file: csar/templates/ServiceTemplateB.yaml
 namespace_prefix: templateB
```

node_types: MyNode:

```
derived from: Root
properties:
  # omitted here for brevity
capabilities:
  # omitted here for brevity
```

781 As you can see, Service Template A also "imports" Service Template B (i.e., "ServiceTemplateB.yaml") 782 bringing in its Type definitions to the global namespace using the Namespace URI declared in Service 783 Template B to fully qualify all of its imported types.

784

- 785 In addition, the import includes a "namespace_prefix" value (i.e., "templateB"), that can be used to qualify
- 786 and disambiguate any Type reference from from Service Template B within Service Template A. This
- 787 prefix is effectively the local alias for the corresponding Namespace URI declared within Service
- 788 Template B (i.e., "http://companyB.com/service/namespace/").

789

- 790 To illustrate conceptually what a TOSCA Orchestrator, for example, would track for their global
- namespace upon processing Service Template A (and by import Service Template B) would be a list of
- global Namespace URIs and their associated Namespace prefixes, as well as a list of fully qualified Type
- names that comprises the overall global namespace.

794 Conceptual Global Namespace URI and Namespace Prefix tracking

795

Entry #	Fully Qualifed URI	Namespace Prefix	Added by Key (Source file)
1	http://open.org/tosca/ns/simple/yaml/1.2/	tosca	 tosca_definitions_version: from Service Template A
2	http://companyA.com/product/ns/	<none></none>	 namespace: from Service Template A
3	http://companyB.com/service/namespace/	templateB	 namespace: from Service Template B namespace_prefix: from Service Template A, during import

796

797 In the above table,

- Entry 1: is an entry for the default TOSCA namespace, which is required to exist for it to be a valid Service template. It is established by the "tosca_definitions_version" key's value. By default, it also gets assigned the "tosca" Namespace prefix.
- Entry 2: is the entry for the local default namespace for Service Template A as declared by the
 "namespace" key.
 - Note that no Namespace prefix is needed; any locally defined types that are not qualified (i.e., not a full URI or using a Namespace Prefix) will default to this namespace if not found first in the TOSCA namespace.
- Entry 3: is the entry for default Namespace URI for any type imported from Service Template B.
 The author of Service Template A has assigned the local Namespace Prefix "template" that can be used to qualify reference to any Type from Service Template B.

809

803

804

805

As per TOSCA specification, any Type, that is not qualified with the 'tosca' prefix or full URI name, should be first resolved by its unqualified name within the TOSCA namespace. If it not found there, then it may be resolved within the local Service Template's default namespace.

813

814 Conceptual Global Namespace and Type tracking

815

Entry#	Owning Namespace URI	Full Name	Short Name	Type Classification
1	http://open.org/tosca/ns/simple/yaml/1.2/	tosca.nodes.Compute	Compute	node
2		tosca.nodes.SoftwareComponent	SoftwareComponent	

	3		tosca.relationships.ConnectsTo	ConnectsTo	relationship
	 100	http://companyA.com/product/ns/	N/A	MyNode	node
	200	<pre>http://companyB.com/service/namespace/</pre>	N/A	MyNode	node
816	•••				
817	In the	above table,			
818 819 820 821 822 823 824 825 826 827 828 829 830 831 832	•	 Entry 1, is an example of one of the TOSC brought into the global namespace via the oldstain of the space via the formation of the space via the sp	e "tosca_definitions_version" k rt that are unique to TOSCA ty type by either its unqualified s ce to the same fully qualified ⁻ sca.nodes.Compute" or "Comp rvice Template would be treate <i>rg/tosca/ns/simple/yaml/1.2/to</i> CA Relationship Type ntifer for the Node Type "MyNo	ey. ypes for historical short or full names is Type name (i.e., its f oute" (i.e., an unqua ed as its fully qualifie sca.nodes.Compute". ode" from Service	full lified
833 834 835 836 837 838 839 840 841 842 843 844	the (U Th No up Fo	s you can see, although both templates defir e TOSCA Orchestrator, processor or tool tra IRI). ne classification column is included as an ex ode Type and "Compute" capability type if th oon context in a Service Template. or example, if the short name "Compute" we e matching type would not be the Compute used upon the Requirement clause being the	ample on how to logically different able would be used to "sea re used in a template on a Reconstruction of the Construction o	qualified Type Nam erentiate a "Compute rch" for a match bas quirements clause, t	e e" sed then

845 **3.3 Parameter and property types**

This clause describes the primitive types that are used for declaring normative properties, parameters and grammar elements throughout this specification.

848 3.3.1 Referenced YAML Types

849 Many of the types we use in this profile are built-in types from the YAML 1.2 specification (i.e., those 850 identified by the "tag:yaml.org,2002" version tag) [YAML-1.2].

The following table declares the valid YAML type URIs and aliases that SHALL be used when possible when defining parameters or properties within TOSCA Service Templates using this specification:

Valid aliases	Type URI
string	tag:yaml.org,2002:str (default)
integer	tag:yaml.org,2002:int
float	tag:yaml.org,2002:float
boolean	tag:yaml.org,2002:bool (i.e., a value either 'true' or 'false')
timestamp	tag:yaml.org,2002:timestamp [YAML-TS-1.1]
null	tag:yaml.org,2002:null

853 **3.3.1.1 Notes**

• The "string" type is the default type when not specified on a parameter or property declaration.

• While YAML supports further type aliases, such as "str" for "string", the TOSCA Simple Profile specification promotes the fully expressed alias name for clarity.

857 3.3.2 TOSCA version

TOSCA supports the concept of "reuse" of type definitions, as well as template definitions which could be version and change over time. It is important to provide a reliable, normative means to represent a

version string which enables the comparison and management of types and templates over time.

861 Therefore, the TOSCA TC intends to provide a normative version type (string) for this purpose in future

862 Working Drafts of this specification.

Shorthand Name	version
Type Qualified Name	tosca:version

863 3.3.2.1 Grammar

866

864 TOSCA version strings have the following grammar:

<major_version>.<minor_version>[.<fix_version>[.<qualifier>[-<build_version]]]</pre>

- 865 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
 - **major_version**: is a required integer value greater than or equal to 0 (zero)
- **minor_version**: is a required integer value greater than or equal to 0 (zero).
- **fix_version**: is an optional integer value greater than or equal to 0 (zero).
- qualifier: is an optional string that indicates a named, pre-release version of the associated
 code that has been derived from the version of the code identified by the combination
 major_version, minor_version and fix_version numbers.
- build_version: is an optional integer value greater than or equal to 0 (zero) that can be used to
 further qualify different build versions of the code that has the same qualifer_string.

874 3.3.2.2 Version Comparison

- When comparing TOSCA versions, all component versions (i.e., *major*, *minor* and *fix*) are compared in sequence from left to right.
- TOSCA versions that include the optional qualifier are considered older than those without a qualifier.

- TOSCA versions with the same major, minor, and fix versions and have the same qualifier string, but with different build versions can be compared based upon the build version.
- Qualifier strings are considered domain-specific. Therefore, this specification makes no
 recommendation on how to compare TOSCA versions with the same major, minor and fix
 versions, but with different qualifiers strings and simply considers them different named branches
 derived from the same code.

885 **3.3.2.3 Examples**

886 Examples of valid TOSCA version strings:

```
# basic version strings
6.1
2.0.1
# version string with optional qualifier
3.1.0.beta
# version string with optional qualifier and build version
1.0.0.alpha-10
```

887 3.3.2.4 Notes

888 [Maven-Version] The TOSCA version type is compatible with the Apache Maven versioning policy.

890 3.3.2.5 Additional Requirements

- A version value of zero (i.e., '0', '0.0', or '0.0.0') SHALL indicate there no version provided.
- A version value of zero used with any qualifiers SHALL NOT be valid.

3.3.3 TOSCA range type

The range type can be used to define numeric ranges with a lower and upper boundary. For example, this allows for specifying a range of ports to be opened in a firewall.

Shorthand Name	range
Type Qualified Name	tosca:range

896 **3.3.3.1 Grammar**

897 TOSCA range values have the following grammar:

[<lower_bound>, <upper_bound>]

- 898 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- **lower_bound**: is a required integer value that denotes the lower boundary of the range.
- **upper_bound**: is a required integer value that denotes the upper boundary of the range. This
 value MUST be greater than **lower_bound**.

902 3.3.3.2 Keywords

903 The following Keywords may be used in the TOSCA range type:

Keyword	Applicable Types	Description
UNBOUNDED	scalar	Used to represent an unbounded upper bounds (positive) value in a set for a scalar type.

904 **3.3.3.3 Examples**

905 Example of a node template property with a range value:

```
# numeric range between 1 and 100
a_range_property: [ 1, 100 ]
```

```
# a property that has allows any number 0 or greater
```

num_connections: [0, UNBOUNDED]

906

907 3.3.4 TOSCA list type

908 The list type allows for specifying multiple values for a parameter of property. For example, if an

application allows for being configured to listen on multiple ports, a list of ports could be configured usingthe list data type.

911 Note that entries in a list for one property or parameter must be of the same type. The type (for simple 912 entries) or schema (for complex entries) is defined by the **entry schema** attribute of the respective

913 property definition, attribute definitions, or input or output parameter definitions.

Shorthand Name	list
Type Qualified Name	tosca:list

914 3.3.4.1 Grammar

915 TOSCA lists are essentially normal YAML lists with the following grammars:

916 3.3.4.1.1 Square bracket notation

```
[ <list_entry_1>, <list_entry_2>, ... ]
```

917 3.3.4.1.2 Bulleted (sequenced) list notation

- <list_entry_1>
- ...
- <list_entry_n>

- 918 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:
- **<list_entry_*>**: represents one entry of the list.

920 **3.3.4.2 Declaration Examples**

921 3.3.4.2.1 List declaration using a simple type

922 The following example shows a list declaration with an entry schema based upon a simple integer type 923 (which has additional constraints):

```
<some_entity>:
...
properties:
listen_ports:
type: list
entry_schema:
description: listen port entry (simple integer type)
type: integer
constraints:
        - max_length: 128
```

924 3.3.4.2.2 List declaration using a complex type

925 The following example shows a list declaration with an entry schema based upon a complex type:

```
<some_entity>:
...
properties:
products:
   type: list
   entry_schema:
    description: Product information entry (complex type) defined elsewhere
    type: ProductInfo
```

926 3.3.4.3 Definition Examples

- 927 These examples show two notation options for defining lists:
- A single-line option which is useful for only short lists with simple entries.
- A multi-line option where each list entry is on a separate line; this option is typically useful or
 more readable if there is a large number of entries, or if the entries are complex.

931 3.3.4.3.1 Square bracket notation

listen_ports: [80, 8080]

932 3.3.4.3.2 Bulleted list notation

listen_ports:

- 80
- 8080

933 3.3.5 TOSCA map type

The map type allows for specifying multiple values for a parameter of property as a map. In contrast to the list type, where each entry can only be addressed by its index in the list, entries in a map are named elements that can be addressed by their keys.

Note that entries in a map for one property or parameter must be of the same type. The type (for simple entries) or schema (for complex entries) is defined by the entry_schema attribute of the respective
 property definition, attribute definition, or input or output parameter definition.

Shorthand Name	map
Type Qualified Name	tosca:map

940 3.3.5.1 Grammar

941 TOSCA maps are normal YAML dictionaries with following grammar:

942 3.3.5.1.1 Single-line grammar

{ <entry_key_1>: <entry_value_1>, ..., <entry_key_n>: <entry_value_n> }

```
...
<entry_key_n>: <entry_value_n>
```

943 3.3.5.1.2 Multi-line grammar

```
<entry_key_1>: <entry_value_1>
...
<entry_key_n>: <entry_value_n>
```

- 944 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:
- entry_key_*: is the required key for an entry in the map
- entry_value_*: is the value of the respective entry in the map

947 3.3.5.2 Declaration Examples

948 3.3.5.2.1 Map declaration using a simple type

949 The following example shows a map with an entry schema definition based upon an existing string type 950 (which has additional constraints):

```
<some_entity>:
...
properties:
   emails:
    type: map
   entry_schema:
      description: basic email address
      type: string
      constraints:
```

```
- max_length: 128
```

951 3.3.5.2.2 Map declaration using a complex type

952 The following example shows a map with an entry schema definition for contact information:

```
<some_entity>:
...
properties:
contacts:
   type: map
entry_schema:
    description: simple contact information
   type: ContactInfo
```

953 3.3.5.3 Definition Examples

954 These examples show two notation options for defining maps:

- A single-line option which is useful for only short maps with simple entries.
- A multi-line option where each map entry is on a separate line; this option is typically useful or 957 more readable if there is a large number of entries, or if the entries are complex.

958 3.3.5.3.1 Single-line notation

notation option for shorter maps
user_name_to_id_map: { user1: 1001, user2: 1002 }

959 3.3.5.3.2 Multi-line notation

```
# notation for longer maps
user_name_to_id_map:
    user1: 1001
    user2: 1002
```

960 3.3.6 TOSCA scalar-unit type

The scalar-unit type can be used to define scalar values along with a unit from the list of recognized units provided below.

963 **3.3.6.1 Grammar**

964 TOSCA scalar-unit typed values have the following grammar:

<scalar> <unit>

- 965 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- scalar: is a required scalar value.
- 967 **unit**: is a required unit value. The unit value MUST be type-compatible with the scalar.

968 3.3.6.2 Additional requirements

- 969 <u>Whitespace</u>: any number of spaces (including zero or none) SHALL be allowed between the
 970 scalar value and the unit value.
- It **SHALL** be considered an error if either the scalar or unit portion is missing on a property or 972 attribute declaration derived from any scalar-unit type.
- When performing constraint clause evaluation on values of the scalar-unit type, both the scalar value portion and unit value portion SHALL be compared together (i.e., both are treated as a single value). For example, if we have a property called storage_size. which is of type scalar-unit, a valid range constraint would appear as follows:
- 977
- storage_size: in_range [4 GB, 20 GB]
- 978 where **storage_size**'s range would be evaluated using both the numeric and unit values 979 (combined together), in this case '4 GB' and '20 GB'.

980 **3.3.6.3 Concrete Types**

Shorthand Names	scalar-unit.size, scalar-unit.size
Type Qualified Names	tosca:scalar-unit.size, tosca:scalar-unit.time

981

983

984

- 982 The scalar-unit type grammar is abstract and has two recognized concrete types in TOSCA:
 - scalar-unit.size used to define properties that have scalar values measured in size units.
 - **scalar-unit.time** used to define properties that have scalar values measured in size units.
- 985 scalar-unit.frequency used to define properties that have scalar values measured in units per second.
- 987 These types and their allowed unit values are defined below.

988 3.3.6.4 scalar-unit.size

989 3.3.6.4.1 Recognized Units

Unit	Usage	Description	
В	size	byte	
kB	size	kilobyte (1000 bytes)	
KiB	size	kibibytes (1024 bytes)	
MB	size	megabyte (1000000 bytes)	
MiB	size	mebibyte (1048576 bytes)	
GB	size	gigabyte (100000000 bytes)	
GiB	size	gibibytes (1073741824 bytes)	
ТВ	size	terabyte (10000000000 bytes)	
TiB	size	tebibyte (1099511627776 bytes)	

990 3.3.6.4.2 Examples

Storage size in Gigabytes
properties:
 storage_size: 10 GB

991 3.3.6.4.3 Notes

992 The unit values recognized by TOSCA Simple Profile for size-type units are based upon a 993 subset of those defined by GNU at 994 http://www.gnu.org/software/parted/manual/html node/unit.html, which is a non-normative reference to this specification. 995 TOSCA treats these unit values as case-insensitive (e.g., a value of 'kB', 'KB' or 'kb' would be 996 • equivalent), but it is considered best practice to use the case of these units as prescribed by 997 GNU. 998 Some Cloud providers may not support byte-level granularity for storage size allocations. In 999 1000 those cases, these values could be treated as desired sizes and actual allocations would be based upon individual provider capabilities. 1001

1002 3.3.6.5 scalar-unit.time

1003 3.3.6.5.1 Recognized Units

Unit	Usage	Description	
d	time	days	
h	time	hours	
m	time	minutes	
s	time	seconds	
ms	time	milliseconds	
us	time	microseconds	
ns	time	nanoseconds	

1004 3.3.6.5.2 Examples

Response time in milliseconds
properties:
 respone time: 10 ms

1005 3.3.6.5.3 Notes

1009

The unit values recognized by TOSCA Simple Profile for time-type units are based upon a subset of those defined by International System of Units whose recognized abbreviations are defined within the following reference:

http://www.ewh.ieee.org/soc/ias/pub-dept/abbreviation.pdf

1010oThis document is a non-normative reference to this specification and intended for publications1011or grammars enabled for Latin characters which are not accessible in typical programming1012languages

1013 **3.3.6.6 scalar-unit.frequency**

1014 3.3.6.6.1 Recognized Units

Unit	Usage	Description	
Hz	frequency	Hertz, or Hz. equals one cycle per second.	
kHz	frequency	Kilohertz, or kHz, equals to 1,000 Hertz	
MHz	frequency	Megahertz, or MHz, equals to 1,000,000 Hertz or 1,000 kHz	
GHz	frequency	Gigahertz, or GHz, equals to 1,000,000,000 Hertz, or 1,000,000 kHz, or 1,000 MHz.	

1015 **3.3.6.6.2 Examples**

Processor raw clock rate
properties:
 clock_rate: 2.4 GHz

1016 **3.3.6.6.3 Notes**

The value for Hertz (Hz) is the International Standard Unit (ISU) as described by the Bureau
 International des Poids et Mesures (BIPM) in the "SI Brochure: The International System of Units
 (SI) [8th edition, 2006; updated in 2014]", http://www.bipm.org/en/publications/si-brochure/

1020 3.4 Normative values

1021 3.4.1 Node States

As components (i.e., nodes) of TOSCA applications are deployed, instantiated and orchestrated over their lifecycle using normative lifecycle operations (see section 5.8 for normative lifecycle definitions) it is important define normative values for communicating the states of these components normatively between orchestration and workflow engines and any managers of these applications.

1026 The following table provides the list of recognized node states for TOSCA Simple Profile that would be set 1027 by the orchestrator to describe a node instance's state:

Node State	Node State		
Value	Transitional	Description	
initial	no	Node is not yet created. Node only exists as a template definition.	
creating	yes	Node is transitioning from initial state to created state.	
created	no	Node software has been installed.	
configuring	yes	Node is transitioning from created state to configured state.	

Node State	Node State		
Value	Transitional	Description	
configured	no	Node has been configured prior to being started.	
starting	yes	Node is transitioning from configured state to started state.	
started	no	Node is started.	
stopping	yes	Node is transitioning from its current state to a configured state.	
deleting	yes	Node is transitioning from its current state to one where it is deleted and its state is no longer tracked by the instance model.	
error	no	Node is in an error state.	

1028 **3.4.2 Relationship States**

- 1029 Similar to the Node States described in the previous section, Relationships have state relative to their (normative) lifecycle operations.
- 1031 The following table provides the list of recognized relationship states for TOSCA Simple Profile that would 1032 be set by the orchestrator to describe a node instance's state:

Node State				
Value	Transitional	Description		
initial	no	Relationship is not yet created. Relationship only exists as a template definition.		

1033 **3.4.2.1 Notes**

Additional states may be defined in future versions of the TOSCA Simple Profile in YAML
 specification.

1036 **3.4.3 Directives**

1037 There are currently no directive values defined for this version of the TOSCA Simple Profile.

1038 **3.4.4 Network Name aliases**

1039 The following are recognized values that may be used as aliases to reference types of networks within an 1040 application model without knowing their actual name (or identifier) which may be assigned by the 1041 underlying Cloud platform at runtime.

Alias value	Description
PRIVATE	An alias used to reference the first private network within a property or attribute of a Node or Capability which would be assigned to them by the underlying platform at runtime.
	A private network contains IP addresses and ports typically used to listen for incoming traffic to an application or service from the Intranet and not accessible to the public internet.

Alias value	Description
PUBLIC	An alias used to reference the first public network within a property or attribute of a Node or Capability which would be assigned to them by the underlying platform at runtime.
	A public network contains IP addresses and ports typically used to listen for incoming traffic to an application or service from the Internet.

1042 3.4.4.1 Usage

1043 These aliases would be used in the **tosca.capabilities.Endpoint** Capability type (and types derived 1044 from it) within the **network_name** field for template authors to use to indicate the type of network the 1045 Endpoint is supposed to be assigned an IP address from.

1046 **3.5 TOSCA Metamodel**

1047 This section defines all modelable entities that comprise the TOSCA Version 1.0 Simple Profile 1048 specification along with their keynames, grammar and requirements.

1049 3.5.1 Required Keynames

The TOSCA metamodel includes complex types (e.g., Node Types, Relationship Types, Capability Types, Data Types, etc.) each of which include their own list of reserved keynames that are sometimes marked as **required**. These types may be used to derive other types. These derived types (e.g., child types) do not have to provide required keynames as long as they have been specified in the type they have been derived from (i.e., their parent type).

1055 **3.6 Reusable modeling definitions**

1056 **3.6.1 Description definition**

1057 This optional element provides a means include single or multiline descriptions within a TOSCA Simple 1058 Profile template as a scalar string value.

1059 **3.6.1.1 Keyname**

1060 The following keyname is used to provide a description within the TOSCA Simple Profile specification:

description

1061 3.6.1.2 Grammar

1062 Description definitions have the following grammar:

description: <<u>string</u>>

1063 **3.6.1.3 Examples**

1064 Simple descriptions are treated as a single literal that includes the entire contents of the line that 1065 immediately follows the **description** key:

description: This is an example of a single line description (no folding).

1066 The YAML "folded" style may also be used for multi-line descriptions which "folds" line breaks as space 1067 characters. description: >

This is an example of a multi-line description using YAML. It permits for line breaks for easier readability...

if needed. However, (multiple) line breaks are folded into a single space character when processed into a single string value.

1068 **3.6.1.4 Notes**

Use of "folded" style is discouraged for the YAML string type apart from when used with the description keyname.

1071 **3.6.2 Metadata**

1072 This optional element provides a means to include optional metadata as a map of strings.

1073 **3.6.2.1 Keyname**

1074 The following keyname is used to provide metadata within the TOSCA Simple Profile specification:

metadata

1075 3.6.2.2 Grammar

1076 Metadata definitions have the following grammar:

metadata: map of <<u>string</u>>

1077 3.6.2.3 Examples

metadata:
 foo1: bar1
 foo2: bar2
 ...

1078 **3.6.2.4 Notes**

 Data provided within metadata, wherever it appears, MAY be ignored by TOSCA Orchestrators and SHOULD NOT affect runtime behavior.

1081 **3.6.3 Constraint clause**

A constraint clause defines an operation along with one or more compatible values that can be used to
 define a constraint on a property or parameter's allowed values when it is defined in a TOSCA Service
 Template or one of its entities.

1085 3.6.3.1 Operator keynames

1086 The following is the list of recognized operators (keynames) when defining constraint clauses:

Operator	Туре	Value Type	Description
equal	scalar	any	Constrains a property or parameter to a value equal to ('=') the value declared.
greater_than	scalar	comparable	Constrains a property or parameter to a value greater than ('>') the value declared.
greater_or_equal	scalar	comparable	Constrains a property or parameter to a value greater than or equal to ('>=') the value declared.
less_than	scalar	comparable	Constrains a property or parameter to a value less than ('<') the value declared.
less_or_equal	scalar	comparable	Constrains a property or parameter to a value less than or equal to ('<=') the value declared.
in_range	dual scalar	comparable, range	Constrains a property or parameter to a value in range of (inclusive) the two values declared.
			Note: subclasses or templates of types that declare a property with the in_range constraint MAY only further restrict the range specified by the parent type.
valid_values	list	any	Constrains a property or parameter to a value that is in the list of declared values.
length	scalar	string, list, map	Constrains the property or parameter to a value of a given length.
min_length	scalar	string, list, map	Constrains the property or parameter to a value to a minimum length.
max_length	scalar	string, list, map	Constrains the property or parameter to a value to a maximum length.
pattern	regex	string	Constrains the property or parameter to a value that is allowed by the provided regular expression.
			Note : Future drafts of this specification will detail the use of regular expressions and reference an appropriate standardized grammar.
schema	string	string	Constrains the property or parameter to a value that is allowed by the referenced schema.

1087 **3.6.3.1.1 Comparable value types**

In the Value Type column above, an entry of "comparable" includes integer, float, timestamp, string,
 version, and scalar-unit types while an entry of "*any*" refers to any type allowed in the TOSCA simple
 profile in YAML.

1091 **3.6.3.2 Schema Constraint purpose**

1092 TOSCA recognizes that there are external data-interchange formats that are widely used within Cloud 1093 service APIs and messaging (e.g., JSON, XML, etc.).

The 'schema' Constraint was added so that, when TOSCA types utilize types from these externally
defined data (interchange) formats on Properties or Parameters, their corresponding Property definitions'
values can be optionally validated by TOSCA Orchestrators using the schema string provided on this
operator.

1098 **3.6.3.3 Additional Requirements**

- If no operator is present for a simple scalar-value on a constraint clause, it SHALL be interpreted as being equivalent to having the "equal" operator provided; however, the "equal" operator may be used for clarity when expressing a constraint clause.
- The "length" operator SHALL be interpreted mean "size" for set types (i.e., list, map, etc.).
- Values provided by the operands (i.e., values and scalar values) SHALL be type-compatible with their associated operations.
- Future drafts of this specification will detail the use of regular expressions and reference an appropriate standardized grammar.
- The value for the keyname 'schema' SHOULD be a string that contains a valid external schema definition that matches the corresponding Property definitions type.
- 1109 1110

When a valid 'schema' value is provided on a Property definition, a TOSCA Orchestrator MAY choose use the contained schema definition for validation.

1111 3.6.3.4 Grammar

0

1112 Constraint clauses have one of the following grammars:

```
# Scalar grammar
<operator>: <scalar_value>
# Dual scalar grammar
<operator>: [ <scalar_value_1>, <scalar_value_2> ]
# List grammar
<operator> [ <value_1>, <value_2>, ..., <value_n> ]
# Regular expression (regex) grammar
pattern: <regular_expression_value>
# Schema grammar
schema: <schema_definition>
```

- 1113 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- 1114 operator: represents a required operator from the specified list shown above (see section 1115 3.6.3.1 "Operator keynames"). 1116 scalar value, scalar value *: represents a required scalar (or atomic quantity) that can 1117 hold only one value at a time. This will be a value of a primitive type, such as an integer or string 1118 that is allowed by this specification. value *: represents a required value of the operator that is not limited to scalars. 1119 **regular** expression value: represents a regular expression (string) value. 1120 schema definition: represents a schema definition as a string. 1121

1122 **3.6.3.5 Examples**

1123 Constraint clauses used on parameter or property definitions:

equal

```
equal: 2
# greater_than
greater_than: 1
# greater_or_equal
greater_or_equal: 2
# less than
less_than: 5
# less or equal
less_or_equal: 4
# in_range
in_range: [ 1, 4 ]
# valid values
valid values: [ 1, 2, 4 ]
# specific length (in characters)
length: 32
# min_length (in characters)
min_length: 8
# max_length (in characters)
max_length: 64
# schema
schema: <</pre>
  {
     # Some schema syntax that matches corresponding property or parameter.
  }
```

1124 3.6.4 Property Filter definition

A property filter definition defines criteria, using constraint clauses, for selection of a TOSCA entity basedupon it property values.

1127 3.6.4.1 Grammar

1128 Property filter definitions have one of the following grammars:

1129 **3.6.4.1.1 Short notation**:

1130 The following single-line grammar may be used when only a single constraint is needed on a property:

<property_name>: <property_constraint_clause>

1131 **3.6.4.1.2 Extended notation**:

1132 The following multi-line grammar may be used when multiple constraints are needed on a property:

<property_name>:

- <property_constraint_clause_1>
- ...
- property_constraint_clause_n>
- 1133 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:
- property_name: represents the name of property that would be used to select a property definition with the same name (property_name) on a TOSCA entity (e.g., a Node Type, Node Template, Capability Type, etc.).
- property_constraint_clause_*: represents constraint clause(s) that would be used to filter
 entities based upon the named property's value(s).

1139 3.6.4.2 Additional Requirements

• Property constraint clauses must be type compatible with the property definitions (of the same name) as defined on the target TOSCA entity that the clause would be applied against.

1142 **3.6.5 Node Filter definition**

1143 A node filter definition defines criteria for selection of a TOSCA Node Template based upon the 1144 template's property values, capabilities and capability properties.

1145 3.6.5.1 Keynames

1146 The following is the list of recognized keynames for a TOSCA node filter definition:

Keyname	Required	Туре	Description
properties	no	list of property filter definition	An optional sequenced list of property filters that would be used to select (filter) matching TOSCA entities (e.g., Node Template, Node Type, Capability Types, etc.) based upon their property definitions' values.
capabilities	no	list of capability names or capability type names	An optional sequenced list of capability names or types that would be used to select (filter) matching TOSCA entities based upon their existence.

1147 **3.6.5.2 Additional filtering on named Capability properties**

1148 Capabilities used as filters often have their own sets of properties which also can be used to construct a 1149 filter.

Keyname	Required	Туре	Description
<capability name_or_type> name>: properties</capability 	no	list of property filter definitions	An optional sequenced list of property filters that would be used to select (filter) matching TOSCA entities (e.g., Node Template, Node Type, Capability Types, etc.) based upon their capabilities' property definitions' values.

1150 **3.6.5.3 Grammar**

1151 Node filter definitions have following grammar:

```
<filter name>:
  properties:
    - <property filter def 1>
    - ...
    - <property filter def n>
  capabilities:
    - <capability name or type 1>:
        properties:
          - <cap 1 property filter def 1>
          - ...
          - < cap m property filter def n>
      . . .
    - <capability_name_or_type_n>:
        properties:
          - <cap 1 property filter def 1>
          - ...
          - < cap m property filter def n>
```

- 1152 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- property_filter_def_*: represents a property filter definition that would be used to select
 (filter) matching TOSCA entities (e.g., Node Template, Node Type, Capability Types, etc.) based
 upon their property definitions' values.
- capability_name_or_type_*: represents the type or name of a capability that would be used to select (filter) matching TOSCA entities based upon their existence.

```
    cap_*_property_def_*: represents a property filter definition that would be used to select
    (filter) matching TOSCA entities (e.g., Node Template, Node Type, Capability Types, etc.) based
    upon their capabilities' property definitions' values.
```

1161 3.6.5.4 Additional requirements

 TOSCA orchestrators SHALL search for matching capabilities listed on a target filter by assuming the capability name is first a symbolic name and secondly it is a type name (in order to avoid namespace collisions).

1165 **3.6.5.5 Example**

1166 The following example is a filter that would be used to select a TOSCA Compute node based upon the 1167 values of its defined capabilities. Specifically, this filter would select Compute nodes that supported a specific range of CPUs (i.e., **num_cpus** value between 1 and 4) and memory size (i.e., **mem_size** of 2 or greater) from its declared "host" capability.

1170

```
my_node_template:
# other details omitted for brevity
requirements:
        - host:
        node_filter:
        capabilities:
        # My "host" Compute node needs these properties:
        - host:
        properties:
            - num_cpus: { in_range: [ 1, 4 ] }
            - mem_size: { greater_or_equal: 512 MB }
```

1171 **3.6.6 Repository definition**

- 1172 A repository definition defines a named external repository which contains deployment and
- 1173 implementation artifacts that are referenced within the TOSCA Service Template.

1174 **3.6.6.1 Keynames**

1175 The following is the list of recognized keynames for a TOSCA repository definition:

Keyname	Required	Туре	Constraints	Description
description	no	description	None	The optional description for the repository.
url	yes	string	None	The required URL or network address used to access the repository.
credential	no	Credential	None	The optional Credential used to authorize access to the repository.

1176 **3.6.6.2 Grammar**

1177 Repository definitions have one the following grammars:

1178 **3.6.6.2.1 Single-line grammar (no credential):**

<repository_name>: <repository_address>

1179 3.6.6.2.2 Multi-line grammar

<repository name>:
 description: <repository description>
 url: <repository address>
 credential: <authorization credential>

1180 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- 1181 **repository_name**: represents the required symbolic name of the repository as a string.
- **repository_description**: contains an optional description of the repository.
- 1183 repository_address: represents the required URL of the repository as a string.
- authorization_credential: represents the optional credentials (e.g., user ID and password)
 used to authorize access to the repository.

1186 **3.6.6.3 Example**

1187 The following represents a repository definition:

```
repositories:
  my_code_repo:
    description: My project's code repository in GitHub
    url: https://github.com/my-project/
```

1188 **3.6.7 Artifact definition**

1189 An artifact definition defines a named, typed file that can be associated with Node Type or Node

1190 Template and used by orchestration engine to facilitate deployment and implementation of interface 1191 operations.

1192 **3.6.7.1 Keynames**

1193 The following is the list of recognized keynames for a TOSCA artifact definition when using the extended 1194 notation:

Keyname	Required	Туре	Description	
type	yes	string	The required artifact type for the artifact definition.	
file	yes	string	The required URI string (relative or absolute) which can be used to locate the artifact's file.	
repository no string		string	The optional name of the repository definition which contains the location of the external repository that contains the artifact. The artifact is expected to be referenceable by its $file$ URI within the repository.	
description	no	description	The optional description for the artifact definition.	
deploy_path	ploy_path no string		The file path the associated file would be deployed into within the target node's container.	

1195 **3.6.7.2 Grammar**

1196 Artifact definitions have one of the following grammars:

1197 **3.6.7.2.1 Short notation**

1198 The following single-line grammar may be used when the artifact's type and mime type can be inferred 1199 from the file URI:

<artifact_name>: <artifact_file_URI>

1200 **3.6.7.2.2 Extended notation:**

1201 The following multi-line grammar may be used when the artifact's definition's type and mime type need to 1202 be explicitly declared:

< <u>artifa</u>	<u>ct_name</u> >:
descr	iption: < <u>artifact_descrip</u>
type:	< <u>artifact_type_name</u> >

file: <artifact file URI>
repository: <artifact repository name>
deploy_path: <file_deployment_path>

1203 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

tion>

- **artifact_name**: represents the required symbolic name of the artifact as a string.
- **artifact_description**: represents the optional description for the artifact.
- **artifact_type_name**: represents the required **artifact type** the artifact definition is based upon.
- artifact_file_URI: represents the required URI string (relative or absolute) which can be
 used to locate the artifact's file.
- artifact_repository_name: represents the optional name of the repository definition to use to retrieve the associated artifact (file) from.
- file_deployement_path: represents the optional path the artifact_file_URI would be
 copied into within the target node's container.

1213 3.6.7.3 Example

1214 The following represents an artifact definition:

my_file_artifact: ../my_apps_files/operation_artifact.txt

1215 **3.6.8 Import definition**

- 1216 An import definition is used within a TOSCA Service Template to locate and uniquely name another
- 1217 TOSCA Service Template file which has type and template definitions to be imported (included) and 1218 referenced within another Service Template.

1219 3.6.8.1 Keynames

1220 The following is the list of recognized keynames for a TOSCA import definition:

Keyname	Required	Туре	Constraints	Description
file	yes	string	None	The required symbolic name for the imported file.
repository	no	string	None	The optional symbolic name of the repository definition where the imported file can be found as a string.
namespace_prefix	no	string	None	The optional namespace prefix (alias) that will be used to indicate the namespace_uri when forming a qualified name (i.e., qname) when referencing type definitions from the imported file.
namespace_uri	no	string	Deprecated	The optional, deprecated namespace URI to that will be applied to type definitions found within the imported file as a string.

1221 3.6.8.2 Grammar

1222 Import definitions have one the following grammars:

1223 **3.6.8.2.1 Single-line grammar**:

imports:

- <URI_1>
- <URI 2>

1224 **3.6.8.2.2 Multi-line grammar**

imports:

```
- file: <file_URI>
repository: <repository_name>
namespace_uri: <definition_namespace_uri> # deprecated
namespace_prefix: <definition_namespace_prefix>
```

- 1225 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- 1226 file_uri: contains the required name (i.e., URI) of the file to be imported as a string.
- repository_name: represents the optional symbolic name of the repository definition where the imported file can be found as a string.
- namespace_uri: represents the optional namespace URI to that will be applied to type definitions found within the imported file as a string.
- namespace_prefix: represents the optional namespace prefix (alias) that will be used to
 indicate the default namespace as declared in the imported Service Template when forming a
 qualified name (i.e., qname) when referencing type definitions from the imported file as a string.

1234 3.6.8.2.3 Requirements

1249

1250

1251

- The imports key "namespace_uri" is now deprecated. It was intended to be able to define a default namespace for any types that were defined within the Service Template being imported; however, with version 1.2, Service Templates MAY now declare their own default Namespace which SHALL be used in place of this key's value.
- Please note that TOSCA Orchestrators and Processors MAY still use
 the "namespace_uri" value if provided, if the imported Service Template has no declared
 default Namespace value. Regardless it is up to the TOSCA Orchestrator or Processor
 to resolve Namespace collisions caused by imports as they see fit, for example, they may
 treat it as an error or dynamically generate a unique namepspace themselves on import.

1244 3.6.8.2.4 Import URI processing requirements

1245 TOSCA Orchestrators, Processors and tooling SHOULD treat the <file_URI> of an import as follows:

- URI: If the <file_URI> is a known namespace URI (identifier), such as a well-known URI defined
 by a TOSCA specification, then it SHOULD cause the corresponding Type definitions to be
 imported.
 - This implies that there may or may not be an actual Service Template, perhaps it is a known set Types identified by the well-known URI.
 - This also implies that internet access is NOT needed to import.
- Alias If the <file_URI> is a reserved TOSCA Namespace alias, then it SHOULD cause the corresponding Type definitions to be imported, using the associated full, Namespace URI to uniquely identify the imported types.

- URL If the <file_URI> is a valid URL (i.e., network accessible as a remote resource) and the
 location contains a valid TOSCA Service Template, then it SHOULD cause the remote Service
 Template to be imported.
- Relative path If the <file_URI> is a relative path URL, perhaps pointing to a Service Template
 located in the same CSAR file, then it SHOULD cause the locally accessible Service Template to
 be imported.
 - If the "repository" key is supplied, this could also mean relative to the repository's URL in a remote file system;
 - If the importing file located in a CSAR file, it should be treated as relative to the current document's location within a CSAR file's directory structure.
- Otherwise, the import SHOULD be considered a failure.

1266 **3.6.8.3 Example**

1261

1262

1263

1264

1267 The following represents how import definitions would be used for the imports keyname within a TOSCA 1268 Service Template:

imports:

- some_definition_file: path1/path2/some_defs.yam1
- another_definition_file:

file: path1/path2/file2.yam1

repository: my_service_catalog
namespace_uri: http://mycompany.com/tosca/1.0/platform
namespace prefix: mycompany

1269 **3.6.9 Property definition**

A property definition defines a named, typed value and related data that can be associated with an entity defined in this specification (e.g., Node Types, Relationship Types, Capability Types, etc.). Properties are used by template authors to provide input values to TOSCA entities which indicate their "desired state" when they are instantiated. The value of a property can be retrieved using the **get_property**

1274 function within TOSCA Service Templates.

1275 3.6.9.1.1 Attribute and Property reflection

1276 The actual state of the entity, at any point in its lifecycle once instantiated, is reflected by Attribute

definitions. TOSCA orchestrators automatically create an attribute for every declared property (with the
 same symbolic name) to allow introspection of both the desired state (property) and actual state
 (attribute).

1280 3.6.9.2 Keynames

1281 The following is the list of recognized keynames for a TOSCA property definition:

Keyname	Required	Туре	Constraints	Description
type	yes	string	None	The required data type for the property.
description	no	description	None	The optional description for the property.
required	no	boolean	default: true	An optional key that declares a property as required (true) or not (false).

Keyname	Required	Туре	Constraints	Description
default	no	<any></any>	None	An optional key that may provide a value to be used as a default if not provided by another means.
status	no	string	default: supported	The optional status of the property relative to the specification or implementation. See table below for valid values.
constraints	no	list of constraint clauses	None	The optional list of sequenced constraint clauses for the property.
entry_schema	no	string	None	The optional key that is used to declare the name of the Datatype definition for entries of set types such as the TOSCA list or map.
external- schema	no	string	None	The optional key that contains a schema definition that TOSCA Orchestrators MAY use for validation when the "type" key's value indicates an External schema (e.g., "json") See section "External schema" below for further explanation and usage.
metadata	no	map of string	N/A	Defines a section used to declare additional metadata information.

1282 **3.6.9.3 Status values**

1283 The following property status values are supported:

Value	Description		
supported Indicates the property is supported. This is the default value for all property definitions.			
unsupported	ported Indicates the property is not supported.		
experimental	perimental Indicates the property is experimental and has no official standing.		
deprecated Indicates the property has been deprecated by a new specification version.			

1284 3.6.9.4 Grammar

1285 Named property definitions have the following grammar:

```
<property_name>:
type: <property_type>
description: <property_description>
required: <property_required>
default: <default_value>
status: <status_value>
constraints:
    - <property_constraints>
entry_schema:
    description: <<u>entry_description</u>>
```

```
type: <<u>entry type</u>>
```

```
constraints:
    - <<u>entry_constraints</u>>
metadata:
<metadata_map>
```

1286	In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
1287 1288 1290 1291 1292 1293 1294 1295 1296 1297 1298 1299 1300 1301 1302 1303 1304 1305	 property_name: represents the required symbolic name of the property as a string. property_description: represents the optional description of the property. property_type: represents the required data type of the property. property_required: represents an optional boolean value (true or false) indicating whether or not the property is required. If this keyname is not present on a property definition, then the property SHALL be considered required (i.e., true) by default. default_value: contains a type-compatible value that may be used as a default if not provided by another means. status_value: a string that contains a keyword that indicates the status of the property relative to the specification or implementation. property_constraints: represents the optional sequenced list of one or more constraint clauses on the property definition. schema_definition: represents the optional description of the entry schema. entry_description: represents the optional description of the entry schema. entry_type: represents the required type name for entries in a list or map property type. entry_constraints: represents the optional sequenced list of one or more constraint clauses on entries in a list or map property type. entry_type: represents the required type name for entries in a list or map property type. entry_constraints: represents the optional sequenced list of one or more constraint clauses on entries in a list or map property type.
1306	3.6.9.5 Additional Requirements
1307 1308 1309 1310 1311 1312 1313 1314 1315 1316 1317 1318 1319	 Implementations of the TOSCA Simple Profile SHALL automatically reflect (i.e., make available) any property defined on an entity as an attribute of the entity with the same name as the property. A property SHALL be considered required by default (i.e., as if the required keyname on the definition is set to true) unless the definition's required keyname is explicitly set to false. The value provided on a property definition's default keyname SHALL be type compatible with the type declared on the definition's type keyname. Constraints of a property definition SHALL be type-compatible with the type defined for that definition. If a 'schema' keyname is provided, its value (string) MUST represent a valid schema definition that matches the recognized external type provided as the value for the 'type' keyname as described by its correspondig schema specification. TOSCA Orchestrators MAY choose to validate the value of the 'schema' keyname in accordance with the corresponding schema specification for any recognized external types.
1320	3.6.9.6 Notes
1321 1322	• This element directly maps to the PropertiesDefinition element defined as part of the schema for most type and entities defined in the TOSCA v1.0 specification.
1323 1324 1325	 In the TOSCA v1.0 specification constraints are expressed in the XML Schema definitions of Node Type properties referenced in the PropertiesDefinition element of NodeType definitions.

1326 **3.6.9.7 Example**

1327 The following represents an example of a property definition with constraints:

```
properties:
    num_cpus:
    type: integer
    description: Number of CPUs requested for a software node instance.
    default: 1
    required: 1
    required: true
    constraints:
        - valid_values: [ 1, 2, 4, 8 ]
```

1328 **3.6.10 Property assignment**

- 1329 This section defines the grammar for assigning values to named properties within TOSCA Node and 1330 Relationship templates that are defined in their corresponding named types.
- 1331 **3.6.10.1 Keynames**
- 1332 The TOSCA property assignment has no keynames.
- 1333 3.6.10.2 Grammar
- 1334 Property assignments have the following grammar:

1335 **3.6.10.2.1 Short notation:**

1336 The following single-line grammar may be used when a simple value assignment is needed:

<property_name>: <property_value> | { <property_value_expression> }

- 1337 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:
- property_name: represents the name of a property that would be used to select a property definition with the same name within on a TOSCA entity (e.g., Node Template, Relationship Template, etc.,) which is declared in its declared type (e.g., a Node Type, Node Template, Capability Type, etc.).
- property_value, property_value_expression: represent the type-compatible value to assign to the named property. Property values may be provided as the result from the evaluation of an expression or a function.

1345 **3.6.11 Attribute definition**

An attribute definition defines a named, typed value that can be associated with an entity defined in this specification (e.g., a Node, Relationship or Capability Type). Specifically, it is used to expose the "actual state" of some property of a TOSCA entity after it has been deployed and instantiated (as set by the TOSCA orchestrator). Attribute values can be retrieved via the **get_attribute** function from the instance model and used as values to other entities within TOSCA Service Templates.

1351 **3.6.11.1 Attribute and Property reflection**

TOSCA orchestrators automatically create Attribute definitions for any Property definitions declared on
 the same TOSCA entity (e.g., nodes, node capabilities and relationships) in order to make accessible the
 actual (i.e., the current state) value from the running instance of the entity.

1355 **3.6.11.2 Keynames**

Keyname	Required	Туре	Constraints	Description
type	yes	string	None	The required data type for the attribute.
description	no	description	None	The optional description for the attribute.
default	no	<any></any>	None	An optional key that may provide a value to be used as a default if not provided by another means. This value SHALL be type compatible with the type declared by the property definition's type keyname.
status	no	string	default: supported	The optional status of the attribute relative to the specification or implementation. See supported status values defined under the Property definition section.
entry_schema	no	string	None	The optional key that is used to declare the name of the Datatype definition for entries of set types such as the TOSCA list or map.

1356 The following is the list of recognized keynames for a TOSCA attribute definition:

1357 **3.6.11.3 Grammar**

1362

1358 Attribute definitions have the following grammar:

attributes:
< <u>attribute_name</u> >:
type: < <u>attribute_type</u> >
<pre>description: <<u>attribute_description</u>></pre>
<pre>default: <default_value></default_value></pre>
<pre>status: <<u>status_value</u>></pre>

- 1359 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- **attribute_name**: represents the required symbolic name of the attribute as a string.
- **attribute_type**: represents the required data type of the attribute.
 - attribute_description: represents the optional description of the attribute.
- default_value: contains a type-compatible value that may be used as a default if not provided by another means.
- status_value: contains a value indicating the attribute's status relative to the specification
 version (e.g., supported, deprecated, etc.). Supported status values for this keyname are defined
 under Property definition.

1368 3.6.11.4 Additional Requirements

- In addition to any explicitly defined attributes on a TOSCA entity (e.g., Node Type, RelationshipType, etc.), implementations of the TOSCA Simple Profile **MUST** automatically reflect (i.e., make available) any property defined on an entity as an attribute of the entity with the same name as the property.
- Values for the default keyname **MUST** be derived or calculated from other attribute or operation
 output values (that reflect the actual state of the instance of the corresponding resource) and not
 hard-coded or derived from a property settings or inputs (i.e., desired state).

1376 **3.6.11.5 Notes**

1381

1382 1383

- Attribute definitions are very similar to Property definitions; however, properties of entities reflect an input that carries the template author's requested or desired value (i.e., desired state) which the orchestrator (attempts to) use when instantiating the entity whereas attributes reflect the actual value (i.e., actual state) that provides the actual instantiated value.
 - For example, a property can be used to request the IP address of a node using a property (setting); however, the actual IP address after the node is instantiated may by different and made available by an attribute.

1384 **3.6.11.6 Example**

1385 The following represents a required attribute definition:

actual_cpus: type: integer description: Actual number of CPUs allocated to the node instance.

1386 **3.6.12 Attribute assignment**

1387 This section defines the grammar for assigning values to named attributes within TOSCA Node and 1388 Relationship templates which are defined in their corresponding named types.

1389 **3.6.12.1 Keynames**

- 1390 The TOSCA attribute assignment has no keynames.
- 1391 3.6.12.2 Grammar
- 1392 Attribute assignments have the following grammar:

1393 **3.6.12.2.1 Short notation:**

1394 The following single-line grammar may be used when a simple value assignment is needed:

<attribute_name>: <attribute_value> | { <attribute_value_expression> }

1395 **3.6.12.2.2 Extended notation:**

1396 The following multi-line grammar may be used when a value assignment requires keys in addition to a 1397 simple value assignment:

```
<attribute_name>:
    description: <attribute_description>
    value: <attribute_value> | { <attribute_value_expression> }
```

- 1398 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:
- attribute_name: represents the name of an attribute that would be used to select an attribute definition with the same name within on a TOSCA entity (e.g., Node Template, Relationship Template, etc.) which is declared (or reflected from a Property definition) in its declared type (e.g., a Node Type, Node Template, Capability Type, etc.).
- attribute_value, attribute_value_expression: represent the type-compatible value to assign to the named attribute. Attribute values may be provided as the result from the evaluation of an expression or a function.

• **attribute_description**: represents the optional description of the attribute.

1407 3.6.12.3 Additional requirements

 Attribute values MAY be provided by the underlying implementation at runtime when requested by the get_attribute function or it MAY be provided through the evaluation of expressions and/or functions that derive the values from other TOSCA attributes (also at runtime).

1411 3.6.13 Parameter definition

A parameter definition is essentially a TOSCA property definition; however, it also allows a value to be assigned to it (as for a TOSCA property assignment). In addition, in the case of output parameters, it can optionally inherit the data type of the value assigned to it rather than have an explicit data type defined for it.

1416 **3.6.13.1 Keynames**

1417 The TOSCA parameter definition has all the keynames of a TOSCA Property definition, but in addition 1418 includes the following additional or changed keynames:

Keyname	Required	Туре	Constraints	Description
type	no	string	None	The required data type for the parameter.
				Note : This keyname is required for a TOSCA Property definition, but is not for a TOSCA Parameter definition.
value	no	<any></any>	N/A	The type-compatible value to assign to the named parameter. Parameter values may be provided as the result from the evaluation of an expression or a function.

1419 3.6.13.2 Grammar

1420 Named parameter definitions have the following grammar:

```
constraints:
    type: constraints:
    constraints:
    constraints:
    constraints:
    constraints:
    constraints:
    constraints:
    constraints:
    constraints:
    constraints>
```

1421 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

1422	•	parameter_name: represents the required symbolic name of the parameter as a string.
1423	•	parameter_description: represents the optional description of the parameter.
1424	•	parameter_type: represents the optional data type of the parameter. Note, this keyname is
1425		required for a TOSCA Property definition, but is not for a TOSCA Parameter definition.
1426	•	parameter_value, parameter_value_expression: represent the type-compatible value to
1427		assign to the named parameter. Parameter values may be provided as the result from the
1428		evaluation of an expression or a function.
1429	•	parameter_required: represents an optional boolean value (true or false) indicating whether or
1430		not the parameter is required. If this keyname is not present on a parameter definition, then the
1431		property SHALL be considered required (i.e., true) by default.
1432	•	default_value: contains a type-compatible value that may be used as a default if not provided
1433		by another means.
1434	•	status_value: a string that contains a keyword that indicates the status of the parameter
1435		relative to the specification or implementation.
1436	•	parameter_constraints: represents the optional <u>sequenced</u> list of one or more constraint
1437		clauses on the parameter definition.
1438	•	entry_description: represents the optional description of the entry schema.
1439	•	entry_type: represents the required type name for entries in a list or map parameter type.
1440	•	entry_constraints: represents the optional sequenced list of one or more constraint clauses
1441		on entries in a list or map parameter type.
1442	3 6 13	.3 Additional Requirements
	0.0.10	
1443	•	A parameter SHALL be considered <u>required by default</u> (i.e., as if the required keyname on the
1444		definition is set to true) unless the definition's required keyname is explicitly set to false.
1445	•	The value provided on a parameter definition's default keyname SHALL be type compatible
1446		with the type declared on the definition's type keyname.

• Constraints of a parameter definition **SHALL** be type-compatible with the type defined for that definition.

1449 **3.6.13.4 Example**

1450 The following represents an example of an input parameter definition with constraints:

```
inputs:
cpus:
  type: integer
  description: Number of CPUs for the server.
  constraints:
      - valid_values: [ 1, 2, 4, 8 ]
```

1451 The following represents an example of an (untyped) output parameter definition:

```
outputs:
    server_ip:
    description: The private IP address of the provisioned server.
    value: { get_attribute: [ my_server, private_address ] }
```

1452

1453 **3.6.14 Operation implementation definition**

1454 An operation implementation definition specifies one or more artifacts (e.g. scripts) to be used as the 1455 implementation for an operation in an interface.

1456 3.6.14.1 Keynames

1457 The following is the list of recognized keynames for a TOSCA operation implementation definition:

Keyname	Req uire d	Туре	Description
primary	no	Artifact definition	The optional implementation artifact (i.e., the primary script file within a TOSCA CSAR file).
dependencies	no	list of Artifact definition	The optional ordered list of one or more dependent or secondary implementation artifacts which are referenced by the primary implementation artifact (e.g., a library the script installs or a secondary script).
timeout	No	integer	Timeout value in seconds
operation_host	no	string	The node on which operations should be executed (for TOSCA call_operation activities). If the operation is associated with an interface on a node type or a relationship template, valid_values are SELF or HOST – referring to the node itself or to the node that is the target of the HostedOn relationship for that node. If the operation is associated with a relationship type or a relationship template, valid_values are SOURCE or TARGET – referring to the relationship source or target node. In both cases, the value can also be set to ORCHESTRATOR to indicated that the operation must be executed in the orchestrator environment rather than within the context of the service being orchestrated.

1458 3.6.14.2 Grammar

1459 Operation implementation definitions have the following grammars:

1460 **3.6.14.2.1 Short notation for use with single artifact**

1461 The following single-line grammar may be used when only a primary implementation artifact name is 1462 needed:

implementation: <primary_artifact_name>

- 1463This notation can be used when the primary artifact name uniquely identifies the artifact, either because it1464refers to a named artifact specified in the artifacts section of a type or template, or because it represents
- the name of a script in the CSAR file that contains the definition.

1466 **3.6.14.2.2 Short notation for use with multiple artifact**

1467 The following multi-line short-hand grammar may be used when multiple artifacts are needed, but each of 1468 the artifacts can be uniquely identified by name as before:

```
implementation:
    primary: <primary_artifact_name>
    dependencies:
        - <list of dependent artifact names>
    operation_host : SELF
    timeout : 60
```

1469 3.6.14.2.3 Extended notation for use with single artifact

- 1470 The following multi-line grammar may be used in Node or Relationship Type or Template definitions when
- only a single artifact is used but additional information about the primary artifact is needed (e.g. to specify the repository from which to obtain the artifact, or to specify the artifact type when it cannot be derived
- 1473 from the artifact file extension):

implementation:
 primary:
 <primary_artifact_definition>
 operation_host : HOST
 timeout : 100

1474 3.6.14.2.4 Extended notation for use with multiple artifacts

1475 The following multi-line grammar may be used in Node or Relationship Type or Template definitions when 1476 there are multiple artifacts that may be needed for the operation to be implemented and additional 1477 information about each of the artifacts is required:

```
implementation:
    primary:
        <primary_artifact_definition>
    dependencies:
        - <list_of_dependent_artifact_definitions>
    operation_host: HOST
    timeout: 120
```

1478 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- primary_artifact_name: represents the optional name (string) of an implementation artifact
 definition (defined elsewhere), or the direct name of an implementation artifact's relative filename
 (e.g., a service template-relative, path-inclusive filename or absolute file location using a URL).
- **primary_artifact_definition**: represents a full inline definition of an implementation artifact.
- 1ist_of_dependent_artifact_names: represents the optional ordered list of one or more dependent or secondary implementation artifact names (as strings) which are referenced by the primary implementation artifact. TOSCA orchestrators will copy these files to the same location as the primary artifact on the target node so as to make them accessible to the primary implementation artifact when it is executed.
- 1488 Ilist_of_dependent_artifact_definitions: represents the ordered list of one or more inline
 definitions of dependent or secondary implementation artifacts. TOSCA orchestrators will copy

1490these artifacts to the same location as the primary artifact on the target node so as to make them1491accessible to the primary implementation artifact when it is executed.

1492 **3.6.15 Operation definition**

1493 An operation definition defines a named function or procedure that can be bound to an operation 1494 implementation.

1495 3.6.15.1 Keynames

1496 The following is the list of recognized keynames for a TOSCA operation definition:

Keyname	Required	Туре	Description
description	no	description	The optional description string for the associated named operation.
implementation	no	Operation implementation definition	The optional definition of the operation implementation
inputs	no	list of parameter definitions	The optional list of input properties definitions (i.e., parameter definitions) for operation definitions that are within TOSCA Node or Relationship Type definitions. This includes when operation definitions are included as part of a Requirement definition in a Node Type.
	no	list of property assignments	The optional list of input property assignments (i.e., parameters assignments) for operation definitions that are within TOSCA Node or Relationship Template definitions. This includes when operation definitions are included as part of a Requirement assignment in a Node Template.

1497 **3.6.15.2 Grammar**

1498 Operation definitions have the following grammars:

1499 **3.6.15.2.1 Short notation**

- 1500 The following single-line grammar may be used when the operation's implementation definition is the only
- keyname that is needed, and when the operation implementation definition itself can be specified using a
 single line grammar

<operation_name>: <implementation_artifact_name>

1503 Extended notation The following multi-line grammar may be used in Node or Relationship Template or 1504 Type definitions when additional information about the operation is needed:

<<u>operation_name</u>>:

description: <<u>operation_description</u>>

implementation: <Operation implementation definitionOperation implementation definition>
inputs:

<property_definitions>

- 1505 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:
- **operation_name**: represents the required symbolic name of the operation as a string.

1507	•	operation_description: represents the optional description string for the corresponding
1508		operation_name.
1509	•	operation_implementation_definition : represents the optional specification of the
1510		operation's implementation).
1511	٠	property_definitions: represents the optional list of property definitions which the TOSCA
1512		orchestrator would make available (i.e., or pass) to the corresponding implementation artifact
1513		during its execution.
1514	•	property_assignments: represents the optional list of property assignments for passing
1515		parameters to Node or Relationship Template operations providing values for properties defined
1516		in their respective type definitions.
1010		
1517	3.6.15	.3 Additional requirements
1518	•	The default sub-classing behavior for implementations of operations SHALL be override. That is,
1519		implementation artifacts assigned in subclasses override any defined in its parent class.
1520	•	Template authors MAY provide property assignments on operation inputs on templates that do
1520	•	
		not necessarily have a property definition defined in its corresponding type.
1522	٠	Implementation artifact file names (e.g., script filenames) may include file directory path names
1523		that are relative to the TOSCA service template file itself when packaged within a TOSCA Cloud
1524		Service ARchive (CSAR) file.
1525	3.6.15	.4 Examples

1526 3.6.15.4.1 Single-line example

interfaces:

Standard:

start: scripts/start_server.sh

1527 **3.6.15.4.2** Multi-line example with shorthand implementation definitions

interfaces:

Configure:

pre_configure_source:

implementation:

primary: scripts/pre_configure_source.sh

dependencies:

- scripts/setup.sh
- binaries/library.rpm
- scripts/register.py

1528 **3.6.15.4.3** Multi-line example with extended implementation definitions

interfaces:

Configure:

pre_configure_source:

```
implementation:
    primary:
        file: scripts/pre_configure_source.sh
        type: tosca.artifacts.Implementation.Bash
        repository: my_service_catalog
    dependencies: - file : scripts/setup.sh
        type : tosca.artifacts.Implementation.Bash
        Repository : my_service_catalog
```

1529 **3.6.16 Interface definition**

1530 An interface definition defines a named interface that can be associated with a Node or Relationship Type

1531 **3.6.16.1 Keynames**

1532 The following is the list of recognized keynames for a TOSCA interface definition:

Keyname	Required	Туре	Description
inputs	no	list of property definitions	The optional list of input property definitions available to all defined operations for interface definitions that are within TOSCA Node or Relationship Type definitions. This includes when interface definitions are included as part of a Requirement definition in a Node Type.
	no	list of property assignments	The optional list of input property assignments (i.e., parameters assignments) for interface definitions that are within TOSCA Node or Relationship Template definitions. This includes when interface definitions are referenced as part of a Requirement assignment in a Node Template.

1533 **3.6.16.2 Grammar**

1534 Interface definitions have the following grammar:

1535 3.6.16.2.1 Extended notation for use in Type definitions

1536 The following multi-line grammar may be used in Node or Relationship Type definitions:

<interface definition name>:
 type: <interface type name>
 inputs:
 <property_definitions>
 <operation_definitions>

1537 **3.6.16.2.2 Extended notation for use in Template definitions**

1538 The following multi-line grammar may be used in Node or Relationship Template definitions:

<<u>interface_definition_name</u>>: inputs:

operty_assignments> <operation_definitions>

- 1539 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:
- interface_definition_name: represents the required symbolic name of the interface as a string.
- interface_type_name: represents the required name of the Interface Type for the interface
 definition.
- property_definitions: represents the optional list of property definitions (i.e., parameters)
 which the TOSCA orchestrator would make available (i.e., or pass) to all defined operations.
 - This means these properties and their values would be accessible to the implementation artifacts (e.g., scripts) associated to each operation during their execution.
- property_assignments: represents the optional list of property assignments for passing
 parameters to Node or Relationship Template operations providing values for properties defined
 in their respective type definitions.
- **operation_definitions**: represents the required name of one or more operation definitions.

1552 **3.6.17 Event Filter definition**

An event filter definition defines criteria for selection of an attribute, for the purpose of monitoring it, within a TOSCA entity, or one its capabilities.

1555 **3.6.17.1 Keynames**

1546

1547

1556 The following is the list of recognized keynames for a TOSCA event filter definition:

Keyname	Required	Туре	Description
node	yes	string	The required name of the node type or template that contains either the attribute to be monitored or contains the requirement that references the node that contains the attribute to be monitored.
requirement	no	string	The optional name of the requirement within the filter's node that can be used to locate a referenced node that contains an attribute to monitor.
capability	no	string	The optional name of a capability within the filter's node or within the node referenced by its requirement that contains the attribute to monitor.

1557 **3.6.17.2 Grammar**

1558 Event filter definitions have following grammar:

node: <node_type_name> | <node_template_name>
requirement: <requirement_name>
capability: <capability_name>

- 1559 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- node_type_name: represents the required name of the node type that would be used to select
 (filter) the node that contains the attribute to monitor or contains the requirement that references
 another node that contains the attribute to monitor.

- node_template_name: represents the required name of the node template that would be used to
 select (filter) the node that contains the attribute to monitor or contains the requirement that
 references another node that contains the attribute to monitor.
- requirement_name: represents the optional name of the requirement that would be used to select (filter) a referenced node that contains the attribute to monitor.
- capability_name: represents the optional name of a capability that would be used to select
 (filter) the attribute to monitor.

1570 **3.6.18 Trigger definition**

1571 A trigger definition defines the event, condition and action that is used to "trigger" a policy it is associated 1572 with.

1573 3.6.18.1 Keynames

1574 The following is the list of recognized keynames for a TOSCA trigger definition:

Keyname	Required	Туре	Description
description	no	description	The optional description string for the named trigger.
event_type	no	string	The required name of the event type that activates the trigger's action.
schedule	no	TimeInterval	The optional time interval during which the trigger is valid (i.e., during which the declared actions will be processed).
target_filter	no	event filter	The optional filter used to locate the attribute to monitor for the trigger's defined condition. This filter helps locate the TOSCA entity (i.e., node or relationship) or further a specific capability of that entity that contains the attribute to monitor.
condition	no	List of condition clause definition	The optional condition which contains a condition clause definition specifying one or multiple attribute constraint that can be monitored. Note: this is optional since sometimes the event occurrence itself is enough to trigger the action.
action	yes	string or operation	The if of the workflow to be invoked when the event is triggered and the condition is met (i.e, evaluates to true). Or The required operation to invoke when the event is triggered and the condition is met (i.e., evaluates to true).

1575 **3.6.18.2 Additional keynames for the extended condition notation**

Keyname	Required	Туре	Description	
constraint	no	List of condition clause definition	The optional condition which contains a condition clause definition specifying one or multiple attribute constraint that can be monitored. Note: this is optional since sometimes the event occurrence itself is enough to trigger the action.	
period	no	scalar-unit.time	unit.time The optional period to use to evaluate for the condition.	
evaluations	no	integer	The optional number of evaluations that must be performed over the period to assert the condition exists.	
method	no	string	The optional statistical method name to use to perform the evaluation of the condition.	

1576 **3.6.18.3 Grammar**

1577 Trigger definitions have the following grammars:

1578 **3.6.18.3.1 Short notation**

1579

<<u>trigger_name</u>>:

```
description: <<u>trigger_description</u>>
event: <event_type_name>
schedule: <time_interval_for_trigger>
target_filter:
    <event_filter_definition>
condition:
    <condition_clause_definition>
action:
    <operation_definition>
```

1580 **3.6.18.3.2 Extended notation:**

1581

```
<trigger name>:
description: <trigger description>
event:
type: <event_type_name>
schedule: <time_interval_for_trigger>
target_filter:
<event_filter_definition>
condition:
constraint: <condition_clause_definition>
period: <scalar-unit.time> # e.g., 60 sec
evaluations: <integer> # e.g., 1
method: <string> # e.g., average
action:
<operation_definition>
```

1582

1590

for.

1583 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- trigger_name: represents the required symbolic name of the trigger as a string.
 trigger_description: represents the optional description string for the corresponding trigger_name.
- event_type_name: represents the required name of the TOSCA Event Type that would be monitored on the identified resource (node).
- time_interval_for_trigger: represents the optional time interval that the trigger is valid

- event_filter_definition: represents the optional filter to use to locate the resource (node)
 or capability attribute to monitor.
- attribute_constraint_clause: represents the optional attribute constraint that would be used to test for a specific condition on the monitored resource.
- operation_definition: represents the required action to take if the event and (optionally)
 condition are met.

1597 **3.6.19 Workflow activity definition**

1598 A workflow activity defines an operation to be performed in a TOSCA workflow. Activities allows to: 1599

- Delegate the workflow for a node expected to be provided by the orchestrator
- Set the state of a node
 - Call an operation defined on a TOSCA interface of a node, relationship or group
- Inline another workflow defined in the topology (to allow reusability)

1604 **3.6.19.1 Keynames**

1600

1602

1605 The following is the list of recognized keynames for a TOSCA workflow activity definition. Note that while 1606 each of the key is not required, one and only one of them is required (mutualy exclusive).

Keyname	Required	Туре	Description	
delegate	no	string	The name of the delegate workflow.	
			This activity requires the target to be provided by the orchestrator (no-op node or relationship)	
set_state	no	string	Value of the node state.	
call_operation	no	string	A string that defines the name of the interface and operation to be called on the node using the <interface_name>.<operation_name> notation.</operation_name></interface_name>	
inline	no	string	The name of a workflow to be inlined.	

1607 3.6.19.2 Grammar

1608 Workflow activity definitions have one of the following grammars:

1609 3.6.19.2.1 Delegate activity

- delegate: <delegate_workflow_name>

- 1610 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- delegate_workflow_name: represents the name of the workflow of the node
 provided by the TOSCA orchestrator.

1613 **3.6.19.2.2 Set state activity**

- set_state: <new_node_state>

- 1614 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- 1615 new_node_state: represents the state that will be affected to the node once 1616 the activity is performed.

1617 3.6.19.2.3 Call operation activity:

```
- call_operation: <interface_name>.<operation_name>
```

- 1618 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- interface_name: represents the name of the interface in which the operation to
 be called is defined.
- operation_name: represents the name of the operation of the interface that
 will be called during the workflow execution.

1623 3.6.19.2.4 Inline activity

- inline: <workflow_name>
- 1624 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- 1625 workflow_name: represents the name of the workflow to inline.

1626 **3.6.19.3 Additional Requirements**

Keynames are mutually exclusive, i.e. an activity MUST define only one of delegate, set_state, call_operation or inline keyname.

1629 **3.6.19.4 Example**

- 1630 following represents a list of workflow activity definitions:
 - delegate: deploy
 - set_state: started
 - call_operation: tosca.interfaces.node.lifecycle.Standard.start
 - inline: my_workflow

1631

1632 **3.6.20 Assertion definition**

1633 A workflow assertion is used to specify a single condition on a workflow filter definition. The assertion 1634 allows to assert the value of an attribute based on TOSCA constraints.

1635 3.6.20.1 Keynames

1636 The TOSCA workflow assertion definition has no keynames.

1637 3.6.20.2 Grammar

1638 Workflow assertion definitions have the following grammar:

<attribute_name>: <list_of_constraint_clauses>

- 1639 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:
- attribute_name: represents the name of an attribute defined on the assertion context entity
 (node instance, relationship instance, group instance) and from which value will be evaluated
 against the defined constraint clauses.

list_of_constraint_clauses: represents the list of constraint clauses that will be used to validate the attribute assertion.

1645 **3.6.20.3 Example**

1646 Following represents a workflow assertion with a single equals constraint:

my_attribute: [{equal : my_value}]

1647 Following represents a workflow assertion with multiple constraints:

my_attribute:

- min_length: 8
- max_length : 10

1648 **3.6.21 Condition clause definition**

1649 A workflow condition clause definition is used to specify a condition that can be used within a workflow 1650 precondition or workflow filter.

1651 3.6.21.1 Keynames

1652 The following is the list of recognized keynames for a TOSCA workflow condition definition:

Keyname	Required	Туре	Description		
and	no	list of condition clause definition	An and clause allows to define sub-filter clause definitions that must all be evaluated truly so the and clause is considered as true.		
or	no	list of condition clause definition	An or clause allows to define sub-filter clause definitions where one of them must all be evaluated truly so the or clause is considered as true.		
assert	no	list of assertion definition	A list of filter assertions to be evaluated on entity attributes. Assert acts as a and clause, i.e. every defined filter assertion must be true so the assertion is considered as true.		

1653

1654 Note : It is allowed to add assertion definition directly as keynames of the condition clause definition. An 1655 and clause is performed for all direct assertion definition.

1656 3.6.21.2 Grammar

- 1657 Workflow assertion definitions have the following grammars:
- 1658 3.6.21.2.1 And clause

and: <list_of_condition_clause_definition>

- 1659 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:
- 1660 list_of_condition_clause_definition: represents the list of condition clauses. All condition clauses MUST be asserted to true so that the and clause is asserted to true.

1662 **3.6.21.2.2 Or clause**

or: <list_of_condition_clause_definition>

- 1663 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:
- 1664 list_of_condition_clause_definition: represents the list of condition clauses. One of the condition clause have to be asserted to true so that the or clause is asserted to true.

1666 **3.6.21.2.3 Assert clause**

assert: <list_of_assertion_definition>

- 1667 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:
- 1668 Iist_of_assertion_definition: represents the list of assertions. All assertions MUST be asserted to true so that the assert clause is asserted to true.

1670 3.6.21.3 Direct assertion definition

```
<attribute_name>: <list_of_constraint_clauses>
```

- 1671 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:
- attribute_name: represents the name of an attribute defined on the assertion context entity (node instance, relationship instance, group instance) and from which value will be evaluated against the defined constraint clauses.
 list of constraint clauses: represents the list of constraint clauses that will be used to validate
- 1675 Iist_of_constraint_clauses: represents the list of constraint clauses that will be used to validate
 1676 the attribute assertion.

1677 3.6.21.4 Additional Requirement

Keynames are mutually exclusive, i.e. a filter definition can define only one of *and*, *or*, or *assert* keyname.

1680 3.6.21.5 Notes

The TOSCA processor SHOULD perform assertion in the order of the list for every defined
 condition clause or assertion definition.

1683 3.6.21.6 Example

1684 Following represents a workflow condition clause with a single equals constraint:

```
condition:
```

- assert:

- my_attribute: [{equal: my_value}]

1685 Following represents a workflow condition clause with a single equals constraints on two different 1686 attributes:

condition:

- assert:

- my_attribute: [{equal: my_value}]}
- my_other_attribute: [{equal: my_other_value}]}
- 1687 Following represents a workflow condition clause with a or constraint on two different assertions:

```
condition:
    or:
        assert:
        my_attribute: [{equal: my_value}]}
        assert:
        my_other_attribute: [{equal: my_other_value}]}
```

Following represents multiple levels of condition clauses with direct assertion definition usage to build the following logic: one_attribute equal one_value AND (my_attribute equal my_value OR my_other_attribute equal my_other_value):

```
condition:
    one_attribute: [{equal: one_value }]
    or:
        assert:
        my_attribute: [{equal: my_value}]}
        assert:
        my_other_attribute: [{equal: my_other_value}]}
```

1691 3.6.22 Workflow precondition definition

A workflow condition can be used as a filter or precondition to check if a workflow can be processed or not based on the state of the instances of a TOSCA topology deployment. When not met, the workflow will not be triggered.

1695 3.6.22.1 Keynames

Keyname	Required Type		Description	
target yes string		string	The target of the precondition (this can be a node template name, a group name)	
target_relationship	no	string	The optional name of a requirement of the target in case the precondition has to be processed on a relationship rather than a node or group. Note that this is applicable only if the target is a node.	
condition	no	list of condition clause definitions	A list of workflow condition clause definitions. Assertion between elements of the condition are evaluated as an AND condition.	

1696 The following is the list of recognized keynames for a TOSCA workflow condition definition:

1697 3.6.22.2 Grammar

1698 Workflow precondition definitions have the following grammars:

```
- target: <target_name>
```

```
target_relationship: <target_requirement_name>
condition:
    <list_of_condition_clause_definition>
```

1699 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- target_name: represents the name of a node template or group in the topology.
- target_requirement_name: represents the name of a requirement of the node template (in case target name refers to a node template.
- 1703 list_of_condition_clause_definition: represents the list of condition clauses
 1704 to be evaluated. The value of the resulting condition is evaluated as an AND
 1705 clause between the different elements.

1706 **3.6.23 Workflow step definition**

1707 A workflow step allows to define one or multiple sequenced activities in a workflow and how they are 1708 connected to other steps in the workflow. They are the building blocks of a declarative workflow.

1709 **3.6.23.1 Keynames**

1710 The following is the list of recognized keynames for a TOSCA workflow step definition:

Keyname	Required	Туре	Description
target	yes	string	The target of the step (this can be a node template name, a group name)
target_relationship	no	string	The optional name of a requirement of the target in case the step refers to a relationship rather than a node or group. Note that this is applicable only if the target is a node.
operation_host	no	string	The node on which operations should be executed (for TOSCA call_operation activities). This element is required only for relationships and groups target. If target is a relationships operation_host is required and valid_values are SOURCE or TARGET – referring to the relationship source or target node. If target is a group operation_host is optional. If not specified the operation will be triggered on every node of the group. If specified the valid_value is a node_type or the name of a node template.
filter	no	list of constraint clauses	Filter is a map of attribute name, list of constraint clause that allows to provide a filtering logic.
activities	yes	list of activity_definition	The list of sequential activities to be performed in this step.
on_success	no	list of string	The optional list of step names to be performed after this one has been completed with success (all activities has been correctly processed).
on_failure	no	list of string	The optional list of step names to be called after this one in case one of the step activity failed.

1711 3.6.23.2 Grammar

1712 Workflow step definitions have the following grammars:

steps:

<step_name>

- 1713 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- target_name: represents the name of a node template or group in the topology.
- target_requirement_name: represents the name of a requirement of the node template (in case target name refers to a node template.
- **operation_host:** the node on which the operation should be executed
- <list_of_condition_clause_definition>: represents a list of condition clause definition.
- list_of_activity_definition: represents a list of activity definition
- target_step_name: represents the name of another step of the workflow.

1721 **3.7 Type-specific definitions**

1722 **3.7.1 Entity Type Schema**

An Entity Type is the common, base, polymorphic schema type which is extended by TOSCA base entity type schemas (e.g., Node Type, Relationship Type, Artifact Type, etc.) and serves to define once all the commonly shared keynames and their types. This is a "meta" type which is abstract and not directly instantiatable.

1727 **3.7.1.1 Keynames**

1728 The following is the list of recognized keynames for a TOSCA Entity Type definition:

Keyname	Required	Туре	Constraints	Description
derived_from	no	string	'None' is the only allowed value	An optional parent Entity Type name the Entity Type derives from.
version	no	version	N/A	An optional version for the Entity Type definition.
metadata	no	map of string	N/A	Defines a section used to declare additional metadata information.
description	no	description	N/A	An optional description for the Entity Type.

1729 **3.7.1.2 Grammar**

1730 Entity Types have following grammar:

<entity_keyname>:

```
# The only allowed value is 'None'
derived_from: None
version: <version_number>
metadata:
   <metadata_map>
description: <<u>interface_description</u>>
```

- 1731 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- **version_number**: represents the optional TOSCA version number for the entity.
- **entity_description**: represents the optional description string for the entity.
- **metadata_map**: represents the optional map of string.

1735 3.7.1.3 Additional Requirements

- The TOSCA Entity Type SHALL be the common base type used to derive all other top-level base
 TOSCA Types.
- The TOSCA Entity Type SHALL NOT be used to derive or create new base types apart from
 those defined in this specification or a profile of this specification.

1740 3.7.2 Capability definition

1741 A capability definition defines a named, typed set of data that can be associated with Node Type or Node 1742 Template to describe a transparent capability or feature of the software component the node describes.

1743 **3.7.2.1 Keynames**

1744 The following is the list of recognized keynames for a TOSCA capability definition:

Keyname	Required	Туре	Constraints	Description
type	yes	string	N/A	The required name of the Capability Type the capability definition is based upon.
description	no	description	N/A	The optional description of the Capability definition.
properties	no	list of property definitions	N/A	An optional list of property definitions for the Capability definition.
attributes	no	list of attribute definitions	N/A	An optional list of attribute definitions for the Capability definition.
valid_source_types	no	string[]	N/A	An optional list of one or more valid names of Node Types that are supported as valid sources of any relationship established to the declared Capability Type.
occurrences	no	range of integer	implied default of [1,UNBOUNDED]	The optional minimum and maximum occurrences for the capability. By default, an exported Capability should allow at least one relationship to be formed with it with a maximum of UNBOUNDED relationships. Note: the keyword UNBOUNDED is also supported to represent any positive integer.

1745 **3.7.2.2 Grammar**

1746 Capability definitions have one of the following grammars:

1747 3.7.2.2.1 Short notation

1748 The following grammar may be used when only a list of capability definition names needs to be declared:

<<u>capability definition name</u>>: <<u>capability type</u>>

1749 3.7.2.2.2 Extended notation

1750 The following multi-line grammar may be used when additional information on the capability definition is 1751 needed:

```
<<u>capability definition name</u>>:

type: <<u>capability type</u>>

description: <<u>capability description</u>>

properties:

<<u>property definitions</u>>

attributes:

<<u>attribute definitions</u>>

valid_source_types: [ <<u>node type names</u>> ]
```

1752 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

1753 1754 1755	•	capability_definition_name: represents the symbolic name of the capability as a string. capability_type : represents the required name of a capability type the capability definition is based upon.
1756	٠	capability_description: represents the optional description of the capability definition.
1757	٠	property_definitions: represents the optional list of property definitions for the capability
1758		definition.
1759	٠	attribute_definitions: represents the optional list of attribute definitions for the capability
1760		definition.
1761	•	node_type_names : represents the optional list of one or more names of Node Types that the

node_type_names. represents the optional list of one of more names of Node Types that the
 Capability definition supports as valid sources for a successful relationship to be established to
 itself.

1764 **3.7.2.3 Examples**

1765 The following examples show capability definitions in both simple and full forms:

1766 3.7.2.3.1 Simple notation example

Simple notation, no properties defined or augmented some_capability: mytypes.mycapabilities.MyCapabilityTypeName

1767 3.7.2.3.2 Full notation example

Full notation, augmenting properties of the referenced capability type

```
some_capability:
  type: mytypes.mycapabilities.MyCapabilityTypeName
  properties:
    limit:
    type: integer
    default: 100
```

1768 3.7.2.4 Additional requirements

- Any Node Type (names) provides as values for the valid_source_types keyname SHALL be
 type-compatible (i.e., derived from the same parent Node Type) with any Node Types defined
 using the same keyname in the parent Capability Type.
 - Capability symbolic names SHALL be unique; it is an error if a capability name is found to occur more than once.

1774 3.7.2.5 Notes

1772

1773

- The Capability Type, in this example MyCapabilityTypeName, would be defined
 elsewhere and have an integer property named limit.
- This definition directly maps to the CapabilitiesDefinition of the Node Type entity as defined in the TOSCA v1.0 specification.

1779 **3.7.3 Requirement definition**

The Requirement definition describes a named requirement (dependencies) of a TOSCA Node Type or
Node template which needs to be fulfilled by a matching Capability definition declared by another TOSCA
modelable entity. The requirement definition may itself include the specific name of the fulfilling entity
(explicitly) or provide an abstract type, along with additional filtering characteristics, that a TOSCA

1784 orchestrator can use to fulfill the capability at runtime (implicitly).

1785 3.7.3.1 Keynames

1786 The following is the list of recognized keynames for a TOSCA requirement definition:

Keyname	Required	Туре	Constraints	Description
capability	yes	string	N/A	The required reserved keyname used that can be used to provide the name of a valid Capability Type that can fulfill the requirement.
node	no	string	N/A	The optional reserved keyname used to provide the name of a valid Node Type that contains the capability definition that can be used to fulfill the requirement.
relationship	no	string	N/A	The optional reserved keyname used to provide the name of a valid Relationship Type to construct when fulfilling the requirement.
occurrences	no	range of integer	implied default of [1,1]	The optional minimum and maximum occurrences for the requirement. Note: the keyword UNBOUNDED is also supported to represent any positive integer.

1787 **3.7.3.1.1 Additional Keynames for multi-line relationship grammar**

1788 The Requirement definition contains the Relationship Type information needed by TOSCA Orchestrators

to construct relationships to other TOSCA nodes with matching capabilities; however, it is sometimes

1790 recognized that additional properties may need to be passed to the relationship (perhaps for

1791 configuration). In these cases, additional grammar is provided so that the Node Type may declare

additional Property definitions to be used as inputs to the Relationship Type's declared interfaces (or

1793 specific operations of those interfaces).

Keyname	Required	Туре	Constraints	Description
type	yes	string	N/A	The optional reserved keyname used to provide the name of the Relationship Type for the requirement definition's relationship keyname.
interfaces	no	list of interface definitions	N/A	The optional reserved keyname used to reference declared (named) interface definitions of the corresponding Relationship Type in order to declare additional Property definitions for these interfaces or operations of these interfaces.

1794 **3.7.3.2 Grammar**

1795 Requirement definitions have one of the following grammars:

1796 **3.7.3.2.1 Simple grammar (Capability Type only)**

<requirement_definition_name>: <capability_type_name>

1797 **3.7.3.2.2 Extended grammar (with Node and Relationship Types)**

<requirement definition name>:
 capability: <capability type name>
 node: <node type name>
 relationship: <relationship type name>
 occurrences: [<min_occurrences>, <max_occurrences>]

17983.7.3.2.3 Extended grammar for declaring Property Definitions on the1799relationship's Interfaces

1800 The following additional multi-line grammar is provided for the relationship keyname in order to declare 1801 new Property definitions for inputs of known Interface definitions of the declared Relationship Type.

<requirement definition name>:
 # Other keynames omitted for brevity
 relationship:
 type: <<u>relationship type name</u>>
 interfaces:
 <<u>interface_definitions</u>>

1802 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- 1803 **requirement definition name:** represents the required symbolic name of the requirement 1804 definition as a string.
- capability type name: represents the required name of a Capability type that can be used to 1805 • 1806 fulfill the requirement.
- **node type name:** represents the optional name of a TOSCA Node Type that contains the 1807 • 1808 Capability Type definition the requirement can be fulfilled by.
- 1809 relationship type name: represents the optional name of a Relationship Type to be used to • 1810 construct a relationship between this requirement definition (i.e., in the source node) to a 1811 matching capability definition (in a target node).
- 1812 min occurrences, max occurrences: represents the optional minimum and maximum • 1813 occurrences of the requirement (i.e., its cardinality).
- 1814 interface definitions: represents one or more already declared interface definitions in the • 1815 Relationship Type (as declared on the type keyname) allowing for the declaration of new 1816 Property definition for these interfaces or for specific Operation definitions of these interfaces.

3.7.3.3 Additional Requirements 1817

- Requirement symbolic names SHALL be unique; it is an error if a requirement name is found to 1818 • 1819 occur more than once.
- 1820 If the occurrences keyname is not present, then the occurrence of the requirement SHALL be 1821 one and only one; that is a default declaration as follows would be assumed: 1822
 - occurrences: [1,1] 0

3.7.3.4 Notes 1823

- 1824 This element directly maps to the **RequirementsDefinition** of the Node Type entity as defined • 1825 in the TOSCA v1.0 specification.
- 1826 The requirement symbolic name is used for identification of the requirement definition only and • 1827 not relied upon for establishing any relationships in the topology.

3.7.3.5 Requirement Type definition is a tuple 1828

- 1829 A requirement definition allows type designers to govern which types are allowed (valid) for fulfillment 1830 using three levels of specificity with only the Capability Type being required.
- 1831 1. Node Type (optional)
- 2. Relationship Type (optional) 1832
- 1833 3. Capability Type (required)

1834 The first level allows selection, as shown in both the simple or complex grammar, simply providing the node's type using the **node** keyname. The second level allows specification of the relationship type to use 1835 1836 when connecting the requirement to the capability using the **relationship** keyname. Finally, the specific named capability type on the target node is provided using the **capability** keyname. 1837

3.7.3.5.1 Property filter 1838

1839 In addition to the node, relationship and capability types, a filter, with the keyname **node_filter**, may be 1840 provided to constrain the allowed set of potential target nodes based upon their properties and their 1841 capabilities' properties. This allows TOSCA orchestrators to help find the "best fit" when selecting among multiple potential target nodes for the expressed requirements. 1842

1843 **3.7.4 Artifact Type**

1844 An Artifact Type is a reusable entity that defines the type of one or more files that are used to define 1845 implementation or deployment artifacts that are referenced by nodes or relationships on their operations.

1846 3.7.4.1 Keynames

The Artifact Type is a TOSCA Entity and has the common keynames listed in section 3.7.1 TOSCA EntitySchema.

Keyname	Required	Туре	Description
mime_type	no	string	The required mime type property for the Artifact Type.
file_ext	no	string[]	The required file extension property for the Artifact Type.
properties	no	list of property definitions	An optional list of property definitions for the Artifact Type.

1849 In addition, the Artifact Type has the following recognized keynames:

1850 **3.7.4.2 Grammar**

1851 Artifact Types have following grammar:

```
<artifact type name>:
  derived_from: >parent_artifact_type_name>
  version: <version_number>
  metadata:
      <map of string>
  description: <artifact_description>
  mime_type: <mime_type_string>
  file_ext: [ <file_extensions> ]
  properties:
      property_definitions>
```

1852 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

1853	•	artifact_type_name: represents the name of the Artifact Type being declared as a string.
1854	•	parent_artifact_type_name: represents the name of the Artifact Type this Artifact Type
1855		definition derives from (i.e., its "parent" type).
1856	•	version_number: represents the optional TOSCA version number for the Artifact Type.
1857	•	artifact_description: represents the optional description string for the Artifact Type.
1858	•	mime_type_string: represents the optional Multipurpose Internet Mail Extensions (MIME)
1859		standard string value that describes the file contents for this type of Artifact Type as a string.
1860	•	file_extensions: represents the optional list of one or more recognized file extensions for this
1861		type of artifact type as strings.
1862	•	property_definitions : represents the optional list of property definitions for the artifact type.

1863 **3.7.4.3 Examples**

my_artifact_type:

description: Java Archive artifact type derived_from: tosca.artifact.Root mime_type: application/java-archive file_ext: [jar]

1864 3.7.4.4 Notes

The 'mime_type' keyname is meant to have values that are Apache mime types such as those defined here: http://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types

1867 **3.7.5 Interface Type**

1868 An Interface Type is a reusable entity that describes a set of operations that can be used to interact with 1869 or manage a node or relationship in a TOSCA topology.

1870 3.7.5.1 Keynames

1871 The Interface Type is a TOSCA Entity and has the common keynames listed in section 3.7.1 TOSCA1872 Entity Schema.

1873 In addition, the Interface Type has the following recognized keynames:

Keyname	Required	Туре	Description
inputs	no	list of property definitions	The optional list of input parameter definitions.

1874 3.7.5.2 Grammar

1881

1875 Interface Types have following grammar:

```
<interface type name>:
  derived_from: caparent interface type name>
  version: <version_number>
  metadata:
        <map of string>
  description: <interface_description>
  inputs:
        property_definitions>
        <operation_definitions>
```

1876 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- 1877 interface_type_name: represents the required name of the interface as a string.
- parent_interface_type_name: represents the name of the Interface Type this Interface Type definition derives from (i.e., its "parent" type).
- **version_number**: represents the optional TOSCA version number for the Interface Type.
 - **interface_description**: represents the optional description string for the Interface Type.
- property_definitions: represents the optional list of property definitions (i.e., parameters)
 which the TOSCA orchestrator would make available (i.e., or pass) to all implementation artifacts
 for operations declared on the interface during their execution.

• operation_definitions: represents the required list of one or more operation definitions.

1886 **3.7.5.3 Example**

1887 The following example shows a custom interface used to define multiple configure operations.

mycompany.mytypes.myinterfaces.MyConfigure: derived_from: tosca.interfaces.relationship.Root description: My custom configure Interface Type inputs: mode: type: string pre_configure_service: description: pre-configure operation for my service post_configure_service: description: post-configure operation for my service

1888 3.7.5.4 Additional Requirements

- Interface Types **MUST NOT** include any implementations for defined operations; that is, the
 implementation keyname is invalid.
- The **inputs** keyname is reserved and **SHALL NOT** be used for an operation name.

1892 **3.7.6 Data Type**

1893 A Data Type definition defines the schema for new named datatypes in TOSCA.

1894 **3.7.6.1 Keynames**

- The Data Type is a TOSCA Entity and has the common keynames listed in section 3.7.1 TOSCA EntitySchema.
 - Keyname Required Description Туре constraints list of The optional list of sequenced constraint clauses for the Data no constraint clauses Type. properties The optional list property definitions that comprise the schema no list of for a complex Data Type in TOSCA. property definitions
- 1897 In addition, the Data Type has the following recognized keynames:

1898 3.7.6.2 Grammar

1899 Data Types have the following grammar:

<<u>data_type_name</u>>: derived_from: <<u>existing_type_name</u>> version: <<u>version_number</u>> metadata:

```
<<u>map</u> of <u>string</u>>
description: <<u>datatype_description</u>>
constraints:
  - <type_constraints>
properties:
  property_definitions>
```

1900	In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
1901 1902 1903 1904 1905 1906 1907 1908 1909	 data_type_name: represents the required symbolic name of the Data Type as a string. version_number: represents the optional TOSCA version number for the Data Type. datatype_description: represents the optional description for the Data Type. existing_type_name: represents the optional name of a valid TOSCA type this new Data Type would derive from. type_constraints: represents the optional <u>sequenced</u> list of one or more type-compatible constraint clauses that restrict the Data Type. property_definitions: represents the optional list of one or more property definitions that provide the schema for the Data Type.
1910	3.7.6.3 Additional Requirements
1911 1912 1913 1914 1915	 A valid datatype definition MUST have either a valid derived_from declaration or at least one valid property definition. Any constraint clauses SHALL be type-compatible with the type declared by the derived_from keyname. If a properties keyname is provided, it SHALL contain one or more valid property definitions.
1916	3.7.6.4 Examples
1917	The following example represents a Data Type definition based upon an existing string type:
1918	3.7.6.4.1 Defining a complex datatype
	<pre># define a new complex datatype mytypes.phonenumber: description: my phone number datatype</pre>

properties:

countrycode:

type: integer

areacode:

type: integer

number:

type: integer

3.7.6.4.2 Defining a datatype derived from an existing datatype 1919

define a new datatype that derives from existing type and extends it

mytypes.phonenumber.extended:

derived_from: mytypes.phonenumber

```
description: custom phone number type that extends the basic phonenumber type
properties:
   phone_description:
    type: string
    constraints:
        - max_length: 128
```

1920 3.7.7 Capability Type

A Capability Type is a reusable entity that describes a kind of capability that a Node Type can declare to
 expose. Requirements (implicit or explicit) that are declared as part of one node can be matched to (i.e.,
 fulfilled by) the Capabilities declared by another node.

1924 **3.7.7.1 Keynames**

1925 The Capability Type is a TOSCA Entity and has the common keynames listed in section 3.7.1 TOSCA1926 Entity Schema.

Keyname	Required	Туре	Description
properties	no	list of property definitions	An optional list of property definitions for the Capability Type.
attributes	no	list of attribute definitions	An optional list of attribute definitions for the Capability Type.
valid_source_types	no	string[]	An optional list of one or more valid names of Node Types that are supported as valid sources of any relationship established to the declared Capability Type.

1927 In addition, the Capability Type has the following recognized keynames:

1928 **3.7.7.2 Grammar**

1929 Capability Types have following grammar:

```
<<u>capability type name</u>>:
derived_from: <<u>parent capability type name</u>>
version: <<u>version_number</u>>
description: <<u>capability description</u>>
properties:
<<u>property definitions</u>>
attributes:
<<u>attribute definitions</u>>
valid_source_types: [ <<u>node type names</u>> ]
```

- 1930 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- capability_type_name: represents the required name of the Capability Type being declared as a string.
- **parent_capability_type_name**: represents the name of the Capability Type this Capability
 Type definition derives from (i.e., its "parent" type).

1935	•	version_number: represents the optional TOSCA version number for the Capability Type.
1936	•	capability_description: represents the optional description string for the corresponding
1937		capability_type_name.
1938	•	property_definitions: represents an optional list of property definitions that the Capability
1939		type exports.
1940	•	attribute_definitions: represents the optional list of attribute definitions for the Capability
1941		Туре.
1942	•	node_type_names: represents the optional list of one or more names of Node Types that the
1943		Capability Type supports as valid sources for a successful relationship to be established to itself.

1944 **3.7.7.3 Example**

mycompany.mytypes.myapplication.MyFeature: derived_from: tosca.capabilities.Root description: a custom feature of my company's application properties: my_feature_setting: type: string my_feature_value: type: integer

1945 **3.7.8 Requirement Type**

A Requirement Type is a reusable entity that describes a kind of requirement that a Node Type can
declare to expose. The TOSCA Simple Profile seeks to simplify the need for declaring specific
Requirement Types from nodes and instead rely upon nodes declaring their features sets using TOSCA
Capability Types along with a named Feature notation.

1950 Currently, there are no use cases in this TOSCA Simple Profile in YAML specification that utilize an 1951 independently defined Requirement Type. This is a desired effect as part of the simplification of the 1952 TOSCA v1.0 specification.

1953 **3.7.9 Node Type**

A Node Type is a reusable entity that defines the type of one or more Node Templates. As such, a Node
 Type defines the structure of observable properties via a *Properties Definition, the Requirements and Capabilities of the node as well as its supported interfaces.*

1957 3.7.9.1 Keynames

The Node Type is a TOSCA Entity and has the common keynames listed in section 3.7.1 TOSCA EntitySchema.

Keyname	Required	Туре	Description
attributes	no	list of attribute definitions	An optional list of attribute definitions for the Node Type.
properties	no	list of property definitions	An optional list of property definitions for the Node Type.

1960 In addition, the Node Type has the following recognized keynames:

Keyname	Required	Туре	Description
requirements	no	list of requirement definitions	An optional <u>sequenced</u> list of requirement definitions for the Node Type.
capabilities	no	list of capability definitions	An optional list of capability definitions for the Node Type.
interfaces	no	list of interface definitions	An optional list of interface definitions supported by the Node Type.
artifacts	no	list of artifact definitions	An optional list of named artifact definitions for the Node Type.

1961 3.7.9.2 Grammar

1962 Node Types have following grammar:

```
<node_type_name>:
  derived_from: cparent_node_type_name>
  version: <<u>version_number</u>>
 metadata:
    <<u>map</u> of <u>string</u>>
  description: <<u>node_type_description</u>>
  attributes:
    <attribute definitions>
  properties:
    <property_definitions>
  requirements:
    - <requirement_definitions>
  capabilities:
    <capability_definitions>
  interfaces:
    <interface definitions>
  artifacts:
    <artifact definitions>
```

1963	In the	above grammar, the pseudo values that appear in angle brackets have the following meaning:
1964	•	node_type_name : represents the required symbolic name of the Node Type being declared.
1965	•	parent_node_type_name: represents the name (string) of the Node Type this Node Type
1966		definition derives from (i.e., its "parent" type).
1967	•	version_number: represents the optional TOSCA version number for the Node Type.
1968	•	node_type_description: represents the optional description string for the corresponding
1969		node_type_name.
1970	•	property_definitions: represents the optional list of property definitions for the Node Type.
1971	•	attribute_definitions: represents the optional list of attribute definitions for the Node Type.
1972	•	requirement_definitions: represents the optional sequenced list of requirement definitions for
1973		the Node Type.

- 1974 capability_definitions: represents the optional list of capability definitions for the Node
 1975 Type.
- 1976 interface_definitions: represents the optional list of one or more interface definitions
 1977 supported by the Node Type.
- 1978 artifact_definitions: represents the optional list of artifact definitions for the Node Type.

1979 **3.7.9.3 Additional Requirements**

Requirements are intentionally expressed as a sequenced list of TOSCA Requirement definitions
 which **SHOULD** be resolved (processed) in sequence order by TOSCA Orchestrators.

1982 3.7.9.4 Best Practices

- It is recommended that all Node Types SHOULD derive directly (as a parent) or indirectly (as an ancestor) of the TOSCA Root Node Type (i.e., tosca.nodes.Root) to promote compatibility and portability. However, it is permitted to author Node Types that do not do so.
- TOSCA Orchestrators, having a full view of the complete application topology template and its
 resultant dependency graph of nodes and relationships, MAY prioritize how they instantiate the nodes
 and relationships for the application (perhaps in parallel where possible) to achieve the greatest
 efficiency

1990 **3.7.9.5 Example**

```
my_company.my_types.my_app_node_type:
```

derived_from: tosca.nodes.SoftwareComponent

description: My company's custom applicaton

properties:

my_app_password:

type: string

description: application password

constraints:

```
- min_length: 6
```

- max_length: 10

attributes:

```
my_app_port:
```

type: integer

description: application port number

requirements:

some_database:
 capability: EndPoint.Database
 node: Database
 relationship: ConnectsTo

1991 **3.7.10 Relationship Type**

A Relationship Type is a reusable entity that defines the type of one or more relationships between NodeTypes or Node Templates.

1994 **3.7.10.1 Keynames**

1995 The Relationship Type is a TOSCA Entity and has the common keynames listed in section 3.7.1 TOSCA 1996 Entity Schema.

Keyname	Required	Definition/Type	Description
properties	no	list of property definitions	An optional list of property definitions for the Relationship Type.
attributes	no	list of attribute definitions	An optional list of attribute definitions for the Relationship Type.
interfaces	no	list of interface definitions	An optional list of interface definitions interfaces supported by the Relationship Type.
valid_target_types	no	string[]	An optional list of one or more names of Capability Types that are valid targets for this relationship.

1997 In addition, the Relationship Type has the following recognized keynames:

1998 **3.7.10.2 Grammar**

2001

2002

2005

2006

2007

1999 Relationship Types have following grammar:

```
<relationship type_name>:
    derived_from: > version: <version number>
    metadata:
        <map of string>
    description: <relationship description>
    properties:
        property definitions>
    attributes:
        <attribute_definitions>
    interfaces:
        <interface_definitions>
        valid_target_types: [ <capability_type_names> ]
```

2000 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- **relationship_type_name**: represents the required symbolic name of the Relationship Type being declared as a string.
- parent_relationship_type_name: represents the name (string) of the Relationship Type this
 Relationship Type definition derives from (i.e., its "parent" type).
 - relationship_description: represents the optional description string for the corresponding relationship_type_name.
 - version_number: represents the optional TOSCA version number for the Relationship Type.
- property_definitions: represents the optional list of property definitions for the Relationship
 Type.

- attribute_definitions: represents the optional list of attribute definitions for the Relationship
 Type.
- interface_definitions: represents the optional list of one or more names of valid interface
 definitions supported by the Relationship Type.
- capability_type_names: represents one or more names of valid target types for the relationship (i.e., Capability Types).

2016 3.7.10.3 Best Practices

- For TOSCA application portability, it is recommended that designers use the normative
 Relationship types defined in this specification where possible and derive from them for
 customization purposes.
- The TOSCA Root Relationship Type (tosca.relationships.Root) SHOULD be used to derive new types where possible when defining new relationships types. This assures that its normative configuration interface (tosca.interfaces.relationship.Configure) can be used in a deterministic way by TOSCA orchestrators.

2024 3.7.10.4 Examples

mycompanytypes.myrelationships.AppDependency: derived_from: tosca.relationships.DependsOn valid_target_types: [mycompanytypes.mycapabilities.SomeAppCapability]

2025 **3.7.11 Group Type**

A Group Type defines logical grouping types for nodes, typically for different management purposes. Groups can effectively be viewed as logical nodes that are not part of the physical deployment topology of an application, yet can have capabilities and the ability to attach policies and interfaces that can be applied (depending on the group type) to its member nodes.

2030

Conceptually, group definitions allow the creation of logical "membership" relationships to nodes in a service template that are not a part of the application's explicit requirement dependencies in the topology template (i.e. those required to actually get the application deployed and running). Instead, such logical membership allows for the introduction of things such as group management and uniform application of policies (i.e., requirements that are also not bound to the application itself) to the group's members.

2036 3.7.11.1 Keynames

The Group Type is a TOSCA Entity and has the common keynames listed in section 3.7.1 TOSCA EntitySchema.

2039 In addition, the Group Type has the following recognized keynames:	2039	In addition, the Grou	p Type has the foll	lowing recognized keynames:
---	------	-----------------------	---------------------	-----------------------------

Keyname	Required	Туре	Description
attributes	no	list of attribute definitions	An optional list of attribute definitions for the Group Type.
properties	no	list of property definitions	An optional list of property definitions for the Group Type.

Keyname	Required	Туре	Description
members	no	string[]	An optional list of one or more names of Node Types that are valid (allowed) as members of the Group Type.
			Note: This can be viewed by TOSCA Orchestrators as an implied relationship from the listed members nodes to the group, but one that does not have operational lifecycle considerations. For example, if we were to name this as an explicit Relationship Type we might call this "MemberOf" (group).
requirements	no	list of requirement definitions	An optional <u>sequenced</u> list of requirement definitions for the Group Type.
capabilities	no	list of capability definitions	An optional list of capability definitions for the Group Type.
interfaces	no	list of interface definitions	An optional list of interface definitions supported by the Group Type.

2040 3.7.11.2 Grammar

2041 Group Types have one the following grammars:

```
<group type name>:
derived_from: <parent group type name>
version: <version number>
metadata:
    <map of string>
description: <group description>
properties:
    <property definitions>
members: [ <list_of_valid_member_types> ]
requirements:
    - <requirement definitions>
capabilities:
    <capability definitions>
interfaces:
    <interface definitions>
```

```
2042 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
```

```
2043
                 group_type_name: represents the required symbolic name of the Group Type being declared as
2044
                 a string.
2045
                 parent group type name: represents the name (string) of the Group Type this Group Type
                 definition derives from (i.e., its "parent" type).
2046
2047
                 version_number: represents the optional TOSCA version number for the Group Type.
                 group description: represents the optional description string for the corresponding
2048
2049
                 group type name.
                 property definitions: represents the optional list of property definitions for the Group Type.
2050
2051
                 list of valid member types: represents the optional list of TOSCA types (e.g.,, Node,
                 Capability or even other Group Types) that are valid member types for being added to (i.e.,
2052
2053
                 members of) the Group Type.
```

 interface_definitions: represents the optional list of one or more interface definitions supported by the Group Type.

2056 3.7.11.3 Additional Requirements

- Group definitions SHOULD NOT be used to define or redefine relationships (dependencies)
 between nodes that can be expressed using normative TOSCA Relationships (e.g., HostedOn,
 ConnectsTo, etc.) within a TOSCA topology template.
- The list of values associated with the "members" keyname **MUST** only contain types that or 2061 homogenous (i.e., derive from the same type hierarchy).

2062 3.7.11.4 Example

2063 The following represents a Group Type definition:

group_types:

mycompany.mytypes.groups.placement:

```
description: My company's group type for placing nodes of type Compute
members: [ tosca.nodes.Compute ]
```

2064 **3.7.12 Policy Type**

A Policy Type defines a type of requirement that affects or governs an application or service's topology at some stage of its lifecycle, but is not explicitly part of the topology itself (i.e., it does not prevent the application or service from being deployed or run if it did not exist).

2068 3.7.12.1 Keynames

- The Policy Type is a TOSCA Entity and has the common keynames listed in section 3.7.1 TOSCA Entity Schema.
- 2071 In addition, the Policy Type has the following recognized keynames:

Keyname	Required	Туре	Description
properties	no	list of property definitions	An optional list of property definitions for the Policy Type.
targets	no	string[]	An optional list of valid Node Types or Group Types the Policy Type can be applied to.
			Note: This can be viewed by TOSCA Orchestrators as an implied relationship to the target nodes, but one that does not have operational lifecycle considerations. For example, if we were to name this as an explicit Relationship Type we might call this "AppliesTo" (node or group).
triggers	no	list of trigger	An optional list of policy triggers for the Policy Type.

2072 3.7.12.2 Grammar

2073 Policy Types have the following grammar:

<policy_type_name>:

derived_from: parent_policy_type_name>

```
version: <version_number>
metadata:
    <map of string>
description: <policy_description>
properties:
    <property_definitions>
targets: [ <list_of_valid_target_types> ]
triggers:
    <list_of_trigger_definitions>
```

- 2074 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
 - **policy_type_name**: represents the required symbolic name of the Policy Type being declared as a string.
- parent_policy_type_name: represents the name (string) of the Policy Type this Policy Type definition derives from (i.e., its "parent" type).
- version_number: represents the optional TOSCA version number for the Policy Type.
- policy_description: represents the optional description string for the corresponding
 policy_type_name.
- property_definitions: represents the optional list of property definitions for the Policy Type.
- **list_of_valid_target_types**: represents the optional list of TOSCA types (i.e., Group or Node Types) that are valid targets for this Policy Type.
- **list_of_trigger_definitions**: represents the optional list of trigger definitions for the policy.

2086 **3.7.12.3 Example**

2075

2076

2087 The following represents a Policy Type definition:

```
policy_types:
  mycompany.mytypes.policies.placement.Container.Linux:
    description: My company's placement policy for linux
    derived_from: tosca.policies.Root
```

2088 3.8 Template-specific definitions

The definitions in this section provide reusable modeling element grammars that are specific to the Node or Relationship templates.

2091 **3.8.1 Capability assignment**

- A capability assignment allows node template authors to assign values to properties and attributes for a named capability definition that is part of a Node Template's type definition.
- 2094 3.8.1.1 Keynames
- 2095 The following is the list of recognized keynames for a TOSCA capability assignment:

Keyname	Required	Туре	Description
properties	no	list of property assignments	An optional list of property definitions for the Capability definition.

Keyname	Required	Туре	Description
attributes	no	list of attribute assignments	An optional list of attribute definitions for the Capability definition.

2096 3.8.1.2 Grammar

2097 Capability assignments have one of the following grammars:

```
<<u>capability definition name</u>>:
properties:
<<u>property assignments</u>>
attributes:
<<u>attribute assignments</u>>
```

- 2098 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:
- **capability_definition_name:** represents the symbolic name of the capability as a string.
- property_assignments: represents the optional list of property assignments for the capability definition.
- attribute_assignments: represents the optional list of attribute assignments for the capability definition.
- 2104 3.8.1.3 Example
- 2105 The following example shows a capability assignment:

2106 **3.8.1.3.1 Notation example**

```
node_templates:
   some_node_template:
      capabilities:
        some_capability:
        properties:
        limit: 100
```

2107 3.8.2 Requirement assignment

A Requirement assignment allows template authors to provide either concrete names of TOSCA
 templates or provide abstract selection criteria for providers to use to find matching TOSCA templates
 that are used to fulfill a named requirement's declared TOSCA Node Type.

2111 3.8.2.1 Keynames

2112 The following is the list of recognized keynames for a TOSCA requirement assignment:

Keyname	Required	Туре	Description
capability	no	string	 The optional reserved keyname used to provide the name of either a: Capability definition within a <i>target</i> node template that can fulfill the requirement. Capability Type that the provider will use to select a type-compatible <i>target</i> node template to fulfill the requirement at runtime.
node	no	string	 The optional reserved keyname used to identify the target node of a relationship. specifically, it is used to provide either a: Node Template name that can fulfill the target node requirement. Node Type name that the provider will use to select a type-compatible node template to fulfill the requirement at runtime.
relationship	no	string	 The optional reserved keyname used to provide the name of either a: Relationship Template to use to relate the <i>source</i> node to the (capability in the) <i>target</i> node when fulfilling the requirement. Relationship Type that the provider will use to select a type-compatible relationship template to relate the <i>source</i> node to the <i>target</i> node at runtime.
node_filter	no	node filter	The optional filter definition that TOSCA orchestrators or providers would use to select a type-compatible <i>target</i> node that can fulfill the associated abstract requirement at runtime.

2113 The following is the list of recognized keynames for a TOSCA requirement assignment's **relationship**

2114 keyname which is used when Property assignments need to be provided to inputs of declared interfaces

2115 or their operations:

Keyname	Required	Туре	Description
type	no	string	The optional reserved keyname used to provide the name of the Relationship Type for the requirement assignment's relationship keyname.
properties	no	list of interface definitions	The optional reserved keyname used to reference declared (named) interface definitions of the corresponding Relationship Type in order to provide Property assignments for these interfaces or operations of these interfaces.

2116 3.8.2.2 Grammar

2117 Named requirement assignments have one of the following grammars:

2118 **3.8.2.2.1 Short notation:**

The following single-line grammar may be used if only a concrete Node Template for the target node needs to be declared in the requirement:

<<u>requirement name</u>>: <<u>node template name</u>>

2121 This notation is only valid if the corresponding Requirement definition in the Node Template's parent

2122 Node Type declares (at a minimum) a valid Capability Type which can be found in the declared target

2123 Node Template. A valid capability definition always needs to be provided in the requirement declaration of

the *source* node to identify a specific capability definition in the *target* node the requirement will form a

2125 TOSCA relationship with.

2126 3.8.2.2.2 Extended notation:

The following grammar would be used if the requirement assignment needs to provide more information than just the Node Template name:

<requirement_name>:
node: <node template_name> | <node type_name>
relationship: <relationship template_name> | <relationship type_name>
capability: <capability symbolic_name> | <capability_type_name>
node_filter:
 <node filter_definition>
occurrences: [min occurrences, max occurrences]

2129 3.8.2.2.3 Extended grammar with Property Assignments for the relationship's 2130 Interfaces

2131 The following additional multi-line grammar is provided for the relationship keyname in order to provide

2132 new Property assignments for inputs of known Interface definitions of the declared Relationship Type.

```
<requirement_name>:

# Other keynames omitted for brevity

relationship:

type: <<u>relationship template name</u>> | <<u>relationship type name</u>>

properties:

<<u>property assignments</u>>

interfaces:

<<u>interface_assignments</u>>
```

- 2133 Examples of uses for the extended requirement assignment grammar include:
- The need to allow runtime selection of the target node based upon an abstract Node Type rather
 than a concrete Node Template. This may include use of the node_filter keyname to provide
 node and capability filtering information to find the "best match" of a concrete Node Template at
 runtime.
- The need to further clarify the concrete Relationship Template or abstract Relationship Type to use when relating the source node's requirement to the target node's capability.

• The need to further clarify the concrete capability (symbolic) name or abstract Capability Type in the target node to form a relationship between.

• The need to (further) constrain the occurrences of the requirement in the instance model.

2143 In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- **requirement_name:** represents the symbolic name of a requirement assignment as a string.
- ende_template_name: represents the optional name of a Node Template that contains the capability this requirement will be fulfilled by.
- relationship_template_name: represents the optional name of a Relationship Type to be used
 when relating the requirement appears to the capability in the target node.
- capability_symbolic_name: represents the optional ordered list of specific, required capability
 type or named capability definition within the target Node Type or Template.

- node_type_name: represents the optional name of a TOSCA Node Type the associated named requirement can be fulfilled by. This must be a type that is compatible with the Node Type declared on the matching requirement (same symbolic name) the requirement's Node Template is based upon.
 relationship type_name: represents the optional name of a Relationship Type that is
- relationship_type_name: represents the optional name of a Relationship Type that is
 compatible with the Capability Type in the target node.
- property_assignments: represents the optional list of property value assignments for the declared relationship.
- interface_assignments: represents the optional list of interface definitions for the declared
 relationship used to provide property assignments on inputs of interfaces and operations.
- capability_type_name: represents the optional name of a Capability Type definition within the target Node Type this requirement needs to form a relationship with.
- node_filter_definition: represents the optional node filter TOSCA orchestrators would use
 to fulfill the requirement for selecting a target node. Note that this SHALL only be valid if the node
 keyname's value is a Node Type and is invalid if it is a Node Template.
- 2166 **3.8.2.3 Examples**

2167 **3.8.2.3.1 Example 1 – Abstract hosting requirement on a Node Type**

- A web application node template named 'my_application_node_template' of type WebApplication
- 2169 declares a requirement named '**host**' that needs to be fulfilled by any node that derives from the node
- 2170 type WebServer.

Example of a requirement fulfilled by a specific web server node template node_templates:

my_application_node_template:

type: tosca.nodes.WebApplication

•••

requirements:

- host:

node: tosca.nodes.WebServer

- 2171 In this case, the node template's type is **WebApplication** which already declares the Relationship Type
- 2172 HostedOn to use to relate to the target node and the Capability Type of Container to be the specific 2173 target of the requirement in the target node.

2174 3.8.2.3.2 Example 2 - Requirement with Node Template and a custom Relationship 2175 Type

- 2176 This example is similar to the previous example; however, the requirement named 'database' describes
- 2177 a requirement for a connection to a database endpoint (Endpoint.Database) Capability Type in a named
- 2178 node template (my_database). However, the connection requires a custom Relationship Type 2179 (my.types.CustomDbConnection') declared on the keyname 'relationship'.
- (my.types.customubconnection) declared on the keyname relationship?

Example of a (database) requirement that is fulfilled by a node template named # "my_database", but also requires a custom database connection relationship my_application_node_template: requirements:

```
- database:
```

node: my_database
capability: Endpoint.Database
relationship: my.types.CustomDbConnection

2180 **3.8.2.3.3 Example 3 - Requirement for a Compute node with additional selection** 2181 **criteria (filter)**

2182 This example shows how to extend an abstract '**host**' requirement for a Compute node with a filter

definition that further constrains TOSCA orchestrators to include additional properties and capabilitieson the target node when fulfilling the requirement.

```
node templates:
  mysal:
  type: tosca.nodes.DBMS.MySQL
   properties:
      # omitted here for brevity
    requirements:
      - host:
          node: tosca.nodes.Compute
          node filter:
            capabilities:
              - host:
                  properties:
                    - num_cpus: { in_range: [ 1, 4 ] }
                    - mem_size: { greater_or_equal: 512 MB }
              - os:
                  properties:
                     - architecture: { equal: x86 64 }
                     - type: { equal: linux }
                    - distribution: { equal: ubuntu }
              - mytypes.capabilities.compute.encryption:
                  properties:
                     - algorithm: { equal: aes }
                     - keylength: { valid values: [ 128, 256 ] }
```

2185 3.8.3 Node Template

A Node Template specifies the occurrence of a manageable software component as part of an
application's topology model which is defined in a TOSCA Service Template. A Node template is an
instance of a specified Node Type and can provide customized properties, constraints or operations
which override the defaults provided by its Node Type and its implementations.

2190 3.8.3.1 Keynames

2191 The following is the list of recognized keynames for a TOSCA Node Template definition:

Keyname	Required	Туре	Description
type	yes	string	The required name of the Node Type the Node Template is based upon.
description	no	description	An optional description for the Node Template.
metadata	no	map of string	Defines a section used to declare additional metadata information.

Keyname	Required	Туре	Description
directives	no	string[]	An optional list of directive values to provide processing instructions to orchestrators and tooling.
properties	no	list of property assignments	An optional list of property value assignments for the Node Template.
attributes	no	list of attribute assignments	An optional list of attribute value assignments for the Node Template.
requirements	no	list of requirement assignments	An optional <u>sequenced</u> list of requirement assignments for the Node Template.
capabilities	no	list of capability assignments	An optional list of capability assignments for the Node Template.
interfaces	no	list of interface definitions	An optional list of named interface definitions for the Node Template.
artifacts	no	list of artifact definitions	An optional list of named artifact definitions for the Node Template.
node_filter	no	node filter	The optional filter definition that TOSCA orchestrators would use to select the correct target node. This keyname is only valid if the directive has the value of "selectable" set.
сору	no	string	The optional (symbolic) name of another node template to copy into (all keynames and values) and use as a basis for this node template.

2192 3.8.3.2 Grammar

```
<<u>node_template_name>:</u>
  type: <<u>node_type_name</u>>
  description: <<u>node_template_description</u>>
  directives: [<<u>directives</u>>]
  metadata:
     <<u>map</u> of <u>string</u>>
  properties:
     <property_assignments></property_assignments>
  attributes:
     <attribute_assignments>
  requirements:
     - <<u>requirement_assignments</u>>
  capabilities:
     <<u>capability_assignments</u>>
  interfaces:
     <interface definitions>
  artifacts:
```

<artifact_definitions> node_filter: <node_filter_definition> copy: <source node template name>

2193 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

2194 node template name: represents the required symbolic name of the Node Template being 2195 declared. **node_type_name**: represents the name of the Node Type the Node Template is based upon. 2196 ٠ 2197 node template description: represents the optional description string for Node Template. • 2198 directives: represents the optional list of processing instruction keywords (as strings) for use by • 2199 tooling and orchestrators. 2200 property assignments: represents the optional list of property assignments for the Node • 2201 Template that provide values for properties defined in its declared Node Type. 2202 attribute assignments: represents the optional list of attribute assignments for the Node 2203 Template that provide values for attributes defined in its declared Node Type. 2204 requirement assignments: represents the optional sequenced list of requirement assignments • for the Node Template that allow assignment of type-compatible capabilities, target nodes, 2205 2206 relationships and target (node filters) for use when fulfilling the requirement at runtime. 2207 capability assignments: represents the optional list of capability assignments for the Node • 2208 Template that augment those provided by its declared Node Type. 2209 • interface definitions: represents the optional list of interface definitions for the Node 2210 Template that augment those provided by its declared Node Type. 2211 artifact definitions: represents the optional list of artifact definitions for the Node Template • 2212 that augment those provided by its declared Node Type. 2213 node filter definition: represents the optional node filter TOSCA orchestrators would use • 2214 for selecting a matching node template. 2215 • source node template name: represents the optional (symbolic) name of another node 2216 template to copy into (all keynames and values) and use as a basis for this node template. 2217 3.8.3.3 Additional requirements 2218 The node filter keyword (and supporting grammar) SHALL only be valid if the Node Template 2219 has a directive keyname with the value of "selectable" set. 2220 The source node template provided as a value on the copy keyname MUST NOT itself use the • 2221 copy keyname (i.e., it must itself be a complete node template description and not copied from 2222 another node template).

2223 3.8.3.4 Example

```
node_templates:
mysql:
type: tosca.nodes.DBMS.MySQL
properties:
root_password: { get_input: my_mysql_rootpw }
port: { get_input: my_mysql_port }
```

```
requirements:
    - host: db_server
interfaces:
```

Standard:

configure: scripts/my_own_configure.sh

2224 3.8.4 Relationship Template

A Relationship Template specifies the occurrence of a manageable relationship between node templates
 as part of an application's topology model that is defined in a TOSCA Service Template. A Relationship
 template is an instance of a specified Relationship Type and can provide customized properties,
 constraints or operations which override the defaults provided by its Relationship Type and its
 implementations.

2230 3.8.4.1 Keynames

Keyname	Required	Туре	Description
type	yes	string	The required name of the Relationship Type the Relationship Template is based upon.
description	no	description	An optional description for the Relationship Template.
metadata	no	map of string	Defines a section used to declare additional metadata information.
properties	no	list of property assignments	An optional list of property assignments for the Relationship Template.
attributes	no	list of attribute assignments	An optional list of attribute assignments for the Relationship Template.
interfaces	no	list of interface definitions	An optional list of named interface definitions for the Node Template.
сору	no	string	The optional (symbolic) name of another relationship template to copy into (all keynames and values) and use as a basis for this relationship template.

2232 3.8.4.2 Grammar

```
<relationship_template_name>:
```

```
type: <relationship type name>
description: <relationship type description>
metadata:
    <map of string>
properties:
    <property assignments>
attributes:
    <attribute assignments>
interfaces:
```

<interface_definitions> copy: <source_relationship_template_name>

2233 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

2234	• relationship_template_name : represents the required symbolic name of the Relationship
2235	Template being declared.
2236	 relationship_type_name: represents the name of the Relationship Type the Relationship
2237	Template is based upon.
2238	 relationship_template_description: represents the optional description string for the
2239	Relationship Template.
2240	• property_assignments: represents the optional list of property assignments for the Relationship
2241	Template that provide values for properties defined in its declared Relationship Type.
2242	 attribute_assignments: represents the optional list of attribute assignments for the
2243	Relationship Template that provide values for attributes defined in its declared Relationship Type.
2244	• interface_definitions: represents the optional list of interface definitions for the Relationship
2245	Template that augment those provided by its declared Relationship Type.
2246	• source_relationship_template_name: represents the optional (symbolic) name of another
2247	relationship template to copy into (all keynames and values) and use as a basis for this
2248	relationship template.

2249 3.8.4.3 Additional requirements

The source relationship template provided as a value on the copy keyname MUST NOT itself use
 the copy keyname (i.e., it must itself be a complete relationship template description and not
 copied from another relationship template).

2253 3.8.4.4 Example

relationship_templates:
 storage_attachment:
 type: <u>AttachesTo</u>
 properties:
 location: /my mount point

2254 3.8.5 Group definition

A group definition defines a logical grouping of node templates, typically for management purposes, but is separate from the application's topology template.

2257 3.8.5.1 Keynames

2258 The following is the list of recognized keynames for a TOSCA group definition:

Keyname	Required	Туре	Description
type	yes	string	The required name of the group type the group definition is based upon.
description	no	description	The optional description for the group definition.

metadata	no	map of string	Defines a section used to declare additional metadata information.
properties	no	list of property assignments	An optional list of property value assignments for the group definition.
members	no	list of string	The optional list of one or more node template names that are members of this group definition.
interfaces	no	list of interface definitions	An optional list of named interface definitions for the group definition.

2259 3.8.5.2 Grammar

2260 Group definitions have one the following grammars:

```
<group name>:
type: <group type name>
description: <group description>
metadata:
    <map of string>
properties:
    <property assignments>
members: [ <list_of_node_templates> ]
interfaces:
    <<u>interface_definitions</u>>
```

2261 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- group_name: represents the required symbolic name of the group as a string.
- group_type_name: represents the name of the Group Type the definition is based upon.
- group_description: contains an optional description of the group.
- property_assignments: represents the optional list of property assignments for the group
 definition that provide values for properties defined in its declared Group Type.
- 1ist_of_node_templates: contains the required list of one or more node template names
 (within the same topology template) that are members of this logical group.
- **interface_definitions**: represents the optional list of interface definitions for the group definition that augment those provided by its declared Group Type.

2271 3.8.5.3 Additional Requirements

 Group definitions SHOULD NOT be used to define or redefine relationships (dependencies) for an application that can be expressed using normative TOSCA Relationships within a TOSCA topology template.

2275 3.8.5.4 Example

2276 The following represents a group definition:

groups:

my_app_placement_group:

```
type: tosca.groups.Root
```

description: My application's logical component grouping for placement
members: [my_web_server, my_sql_database]

2277 3.8.6 Policy definition

A policy definition defines a policy that can be associated with a TOSCA topology or top-level entity definition (e.g., group definition, node template, etc.).

2280 3.8.6.1 Keynames

2281 The following is the list of recognized keynames for a TOSCA policy definition:

Keyname	Required	Туре	Description
type	yes	string	The required name of the policy type the policy definition is based upon.
description	no	description	The optional description for the policy definition.
metadata	no	map of string	Defines a section used to declare additional metadata information.
properties	no	list of property assignments	An optional list of property value assignments for the policy definition.
targets	no	string[]	An optional list of valid Node Templates or Groups the Policy can be applied to.
triggers	no	list of trigger definitions	An optional list of trigger definitions to invoke when the policy is applied by an orchestrator against the associated TOSCA entity.

2282 **3.8.6.2 Grammar**

2283 Policy definitions have one the following grammars:

```
<policy_name>:
  type: <policy_type_name>
  description: <policy_description>
  metadata:
        <map_of_string>
  properties:
        <property_assignments>
  targets: [<list_of_policy_targets>]
  triggers:
        <list_of_trigger_definitions>
```

- 2284 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- **policy_name**: represents the required symbolic name of the policy as a string.
- **policy_type_name**: represents the name of the policy the definition is based upon.
- **policy_description**: contains an optional description of the policy.

- property_assignments: represents the optional list of property assignments for the policy
 definition that provide values for properties defined in its declared Policy Type.
- list_of_policy_targets: represents the optional list of names of node templates or groups
 that the policy is to applied to.
- **list_of_trigger_definitions**: represents the optional list of trigger definitions for the policy.

2293 3.8.6.3 Example

2294 The following represents a policy definition:

policies:

```
- my_compute_placement_policy:
    type: tosca.policies.placement
    description: Apply my placement policy to my application's servers
    targets: [ my_server_1, my_server_2 ]
    # remainder of policy definition left off for brevity
```

2295 3.8.7 Imperative Workflow definition

A workflow definition defines an imperative workflow that is associated with a TOSCA topology.

2297 3.8.7.1 Keynames

2298 The following is the list of recognized keynames for a TOSCA workflow definition:

Keyname	Required	Туре	Description
description	no	description	The optional description for the workflow definition.
metadata	no	map of string	Defines a section used to declare additional metadata information.
inputs	no	list of property definitions	The optional list of input parameter definitions.
preconditions	no	list of precondition definitions	List of preconditions to be validated before the workflow can be processed.
steps	No	list of step definitions	An optional list of valid Node Templates or Groups the Policy can be applied to.

2299

2300 3.8.7.2 Grammar

2301 Imperative workflow definitions have the following grammar:

```
<workflow_name>:
    description: <workflow_description>
    metadata:
        <map of string>
    inputs:
        <property_definitions>
    preconditions:
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- 2303 workflow_name:
- workflow_description:
- 2305 property_definitions:
- 2306 workflow_precondition_definition:
- 2307 workflow_steps:

2308 **3.8.8 Property mapping**

A property mapping allows to map the property of a substituted node type to a property definition or value (mapped as a constant value property definition) within the topology template.

- A property mapping may refer to an input of the topology, to the property of a node template in the
- topology or be assigned to a constant value.

2313 3.8.8.1 Keynames

- 2314 The following is the list of recognized keynames for a TOSCA property mapping:
- 2315

Keyname	Required	Туре	Description
mapping	no	Array of strings	An array of string with a size from 1 to 3 elements. When size is 1 the string references an input of the topology. When size is 2 the first element refers to the name of a node template in the topology and the second element to a property of the node template. When size is 3 the first element refers to the name of a node template in the topology, the second element to a capability, or a requirement of the given node and the third element to a property of the capability or requirement.
value	no	List of property mappings	This keyname allows to set the value to be assigne to this property definition. This field is mutually exclusive with the mapping keyname.

2316 3.8.8.2 Grammar

2317 The single-line grammar of a **property_mapping** is as follows:

```
2318
```

```
<property_name>: <property_value>
<property_name>: [ <input_name> ]
<property_name>: [ <node_template_name>, <node_template_property_name> ]
<property_name>: [ <node_template_name>, <node_template_capability_name> |
<node template requirement name>, <property name> ]
```

2319 The multi-line grammar is as follows :

```
<property_name>:
mapping: [ < input_name > ]
```

```
<property_name>:
mapping: [ <node_template_name>, <node_template_property_name> ]
<property_name>:
mapping: [ <node_template_name>, <node_template_capability_name> |
<node_template_requirement_name>, <property_name> ]
<property_name>:
value: <property_value>
```

2320

2321 3.8.8.3 Notes

• Single line grammar for a property value assignment is not allowed for properties of list type in order to avoid collision with the mapping single line grammar.

2324 3.8.8.4 Additional constraints

When Input mapping it may be referenced by multiple nodes in the topologies with resulting attributes values that may differ later on in the various nodes. In any situation, the attribute reflecting the property of the substituted type will remain a constant value set to the one of the input at deployment time.

2329 3.8.9 Capability mapping

A capability mapping allows to map the capability of one of the node of the topology template to the capability of the node type the service template offers an implementation for.

2332 3.8.9.1 Keynames

- 2333 The following is the list of recognized keynames for a TOSCA capability mapping:
- 2334

Keyname	Required	Туре	Description
mapping	no	Array of 2 strings	An array of 2 strings, the first one being the name of a node template, the second the name of a capability of the specified node template.
properties	no	List of property assignment	This field is mutually exclusive with the mapping keyname and allow to provide a capability for the template and specify it's related properties.
attributes	no	List of attributes assignment	This field is mutually exclusive with the mapping keyname and allow to provide a capability for the template and specify it's related attributes.

2335

2336 3.8.9.2 Grammar

- 2337 The single-line grammar of a **capability_mapping** is as follows:
- 2338

```
<capability_name>: [ <node_template_name>, <node_template_capability_name> ]
```

2339 The multi-line grammar is as follows :

```
<capability_name>:
  mapping: [ <node_template_name>, <node_template_capability_name> ]
  properties:
     <property_name>: <property_value>
  attributes:
     <attribute_name>: <attribute_value>
```

2340

2341	In the above grammar,	the pseudo value	s that appear in angle	a brackets have the t	following meaning:
2041	in the above granninal,	the pseudo value	s inal appear in anyie		ulowing meaning.

2342 2343 2344	•	capability_name : represents the name of the capability as it appears in the Node Type definition for the Node Type (name) that is declared as the value for on the substitution_mappings' "node_type" key.
2345 2346	•	node_template_name : represents a valid name of a Node Template definition (within the same topology_template declaration as the substitution_mapping is declared).
2347 2348	•	node_template_capability_name : represents a valid name of a capability definition within the <node_template_name> declared in this mapping.</node_template_name>
2349	•	property_name: represents the name of a property of the capability.
2350	•	property_value: represents the value to assign to a property of the capability.
2351	٠	attribute_name: represents the name a an attribute of the capability.
2352	•	attribute_value: represents the value to assign to an attribute of the capability.

2353 3.8.9.3 Additional requirements

• Definition of capability assignment in a capability mapping (through properties and attribute 2355 keynames) SHOULD be prohibited for connectivity capabilities as tosca.capabilities.Endpoint.

2356 3.8.10 Requirement mapping

A requirement mapping allows to map the requirement of one of the node of the topology template to the requirement of the node type the service template offers an implementation for.

2359 3.8.10.1 Keynames

2360 The following is the list of recognized keynames for a TOSCA requirement mapping:

2361

Keyname	Required	Туре	Description
mapping	no	Array of 2 strings	An array of 2 strings, the first one being the name of a node template, the second the name of a requirement of the specified node template.
properties	no	List of property assignment	This field is mutually exclusive with the mapping keyname and allow to provide a requirement for the template and specify it's related properties.
attributes	no	List of attributes assignment	This field is mutually exclusive with the mapping keyname and allow to provide a requirement for the template and specify it's related attributes.

2362

2363 3.8.10.2 Grammar

```
2364 The single-line grammar of a requirement_mapping is as follows:
```

2365

```
<requirement_name>: [ <node_template_name>, <node_template_requirement_name> ]
```

2366 The multi-line grammar is as follows :

```
<requirement_name>:
mapping: [ <node_template_name>, <node_template_requirement_name> ]
properties:
  <property_name>: <property_value>
attributes:
  <attribute name>: <attribute value>
```

2367

```
2368 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
```

```
    requirement_name: represents the name of the requirement as it appears in the Node Type definition for the Node Type (name) that is declared as the value for on the substitution_mappings' "node_type" key.
```

- node_template_name: represents a valid name of a Node Template definition (within the same topology_template declaration as the substitution_mapping is declared).
- node_template_requirement_name: represents a valid name of a requirement definition within
 the <node_template_name> declared in this mapping.
- **property_name:** represents the name of a property of the requirement.
- property_value: represents the value to assign to a property of the requirement.
- **attribute_name:** represents the name a an attribute of the requirement.
- **attribute_value:** represents the value to assign to an attribute of the requirement.

2380 **3.8.10.3 Additional requirements**

• Definition of capability assignment in a capability mapping (through properties and attribute 2382 keynames) SHOULD be prohibited for connectivity capabilities as tosca.capabilities.Endpoint.

2383 **3.8.11 Interface mapping**

An interface mapping allows to map a workflow of the topology template to an operation of the node type the service template offers an implementation for.

2386 3.8.11.1 Grammar

2387 The grammar of an **interface_mapping** is as follows:

2388

<interface_name>:

<operation_name>: <workflow_name>

2389 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- interface_name: represents the name of the interface as it appears in the Node Type definition
 for the Node Type (name) that is declared as the value for on the substitution_mappings'
 "node type" key. Or the name of a new management interface to add to the generated type.
- operation_name: represents the name of the operation as it appears in the
 interface type definition.
- workflow_name: represents the name of a workflow of the template to map to the
 specified operation.

2397 3.8.11.2 Notes

- Declarative workflow generation will be applied by the TOSCA orchestrator after the topology template have been substituted. Unless one of the normative operation of the standard interface is mapped through an interface mapping. In that case the declarative workflow generation will consider the substitution node as any other node calling the create, configure and start mapped workflows as if they where single operations.
- Operation implementation being TOSCA workflows the TOSCA orchestrator replace the usual operation_call activity by an inline activity using the specified workflow.

2405 3.8.12 Substitution mapping

A substitution mapping allows to create a node type out of a given topology template. This allows the consumption of complex systems using a simplified vision.

2408 3.8.12.1 Keynames

Keyname	Required	Туре	Description
node_type	yes	string	The required name of the Node Type the Topology Template is providing an implementation for.
properties	no	List of property mappings	The optional list of properties mapping allowing to map properties of the node_type to inputs, node template properties or values.
capabilities	no	List of capability mappings	The optional list of capabilities mapping.
requirements	no	List of requirement mappings	The optional list of requirements mapping.
interfaces	no	List of interfaces mappings	The optional list of interface mapping allows to map an interface and operations of the node type to implementations that could be either workflows or node template interfaces/operations.

2409

2410 3.8.12.2 Grammar

2411 The grammar of the **substitution_mapping** section is as follows:

```
node_type: <<u>node type_name</u>>
properties:
    <property_mappings>
capabilities:
    <capability_mappings>
requirements:
```

```
<requirement_mappings>
attributes:
    <attribute_mappings>
interfaces:
    <interface_mappings>
```

- 2412 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:
- node_type_name: represents the required Node Type name that the Service Template's topology
 is offering an implementation for.
- **properties**: represents the <optional> list of properties mappings.
- **capability_mappings**: represents the <optional> list of capability mappings.
- **requirement_mappings**: represents the <optional> list of requirement mappings.
- **attributes**: represents the <optional> list of attributes mappings.
- **interfaces:** represents the <optional> list of interfaces mappings.
- 2420 3.8.12.3 Examples
- 2421

2422 3.8.12.4 Additional requirements

• The substitution mapping MUST provide mapping for every property, capability and requirement defined in the specified <node_type>

2425 3.8.12.5 Notes

• The node_type specified in the substitution mapping SHOULD be abstract (does not provide implementation for normative operations).

2428 **3.9 Topology Template definition**

This section defines the topology template of a cloud application. The main ingredients of the topology template are node templates representing components of the application and relationship templates representing links between the components. These elements are defined in the nested **node_templates** section and the nested **relationship_templates** sections, respectively. Furthermore, a topology template allows for defining input parameters, output parameters as well as grouping of node templates.

2434 3.9.1 Keynames

Keyname	Required	Туре	Description
description	no	description	The optional description for the Topology Template.
inputs	no	list of parameter definitions	An optional list of input parameters (i.e., as parameter definitions) for the Topology Template.
node_templates	no	list of node templates	An optional list of node template definitions for the Topology Template.

2435 The following is the list of recognized keynames for a TOSCA Topology Template:

Keyname	Required	Туре	Description
relationship_templates	no	list of relationship templates	An optional list of relationship templates for the Topology Template.
groups	no	list of group definitions	An optional list of Group definitions whose members are node templates defined within this same Topology Template.
policies	no	list of policy definitions	An optional list of Policy definitions for the Topology Template.
outputs	no	list of parameter definitions	An optional list of output parameters (i.e., as parameter definitions) for the Topology Template.
substitution_mappings	no	substitution_mapping	An optional declaration that exports the topology template as an implementation of a Node type. This also includes the mappings between the external Node Types named capabilities and requirements to existing implementations of those capabilities and requirements on Node templates declared within the topology template.
workflows	no	list of imperative workflow definitions	An optional map of imperative workflow definition for the Topology Template.

2436 3.9.2 Grammar

2437 The overall grammar of the **topology_template** section is shown below.–Detailed grammar definitions 2438 of the each sub-sections are provided in subsequent subsections.

```
topology_template:
    description: <template description>
    inputs: <input_parameter_list>
    outputs: <output_parameter_list>
    node_templates: <node_template_list>
    relationship_templates: <relationship_template_list>
    groups: <group_definition_list>
    policies:
        - <policy_definition_list>
    workflows: <workflow_list>
    # Optional declaration that exports the Topology Template
    # as an implementation of a Node Type.
    substitution_mappings:
        <substitution_mappings>
```

2439 In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

2440	•	template_description: represents the optional description string for Topology Template.
2441	٠	input_parameter_list: represents the optional list of input parameters (i.e., as property
2442		definitions) for the Topology Template.
2443	٠	output_parameter_list: represents the optional list of output parameters (i.e., as property
2444		definitions) for the Topology Template.
2445	٠	group_definition_list: represents the optional list of group definitions whose members are
2446		node templates that also are defined within this Topology Template.
2447	•	policy_definition_list: represents the optional sequenced list of policy definitions for the
2448		Topology Template.
2449	٠	<pre>workflow_list: represents the optional list of imperative workflow definitions</pre>
2450		for the Topology Template.
2451	٠	node_template_list: represents the optional list of node template definitions for the Topology
2452		Template.
2453	٠	relationship_template_list: represents the optional list of relationship templates for the
2454		Topology Template.
2455	٠	node_type_name : represents the optional name of a Node Type that the Topology Template
2456		implements as part of the substitution_mappings.
2457	٠	map_of_capability_mappings_to_expose: represents the mappings that expose internal
2458		capabilities from node templates (within the topology template) as capabilities of the Node Type
2459		definition that is declared as part of the substitution_mappings .
2460	٠	<pre>map_of_requirement_mappings_to_expose: represents the mappings of link requirements of</pre>
2461		the Node Type definition that is declared as part of the substitution_mappings to internal
2462		requirements implementations within node templates (declared within the topology template).
2463		

2464 More detailed explanations for each of the Topology Template grammar's keynames appears in the 2465 sections below.

2466 **3.9.2.1 inputs**

The **inputs** section provides a means to define parameters using TOSCA parameter definitions, their allowed values via constraints and default values within a TOSCA Simple Profile template. Input parameters defined in the **inputs** section of a topology template can be mapped to properties of node templates or relationship templates within the same topology template and can thus be used for parameterizing the instantiation of the topology template.

2472

2473 This section defines topology template-level input parameter section.

- Inputs here would ideally be mapped to BoundaryDefinitions in TOSCA v1.0.
- Treat input parameters as fixed global variables (not settable within template)
- If not in input take default (nodes use default)

2477 3.9.2.1.1 Grammar

2478 The grammar of the **inputs** section is as follows:

inputs:

<parameter_definition_list>

2479 3.9.2.1.2 Examples

2480 This section provides a set of examples for the single elements of a topology template.

 TOSCA-Simple-Profile-YAML-v1.2-os

 Standards Track Work Product

 Copyright © OASIS Open 2019. All Rights Reserved.

2481 Simple **inputs** example without any constraints:

```
inputs:
  fooName:
    type: string
    description: Simple string typed property definition with no constraints.
    default: bar
```

2482 Example of **inputs** with constraints:

inputs:

```
SiteName:
  type: string
  description: string typed property definition with constraints
  default: My Site
  constraints:
    - min_length: 9
```

2483 3.9.2.2 node_templates

The **node_templates** section lists the Node Templates that describe the (software) components that are used to compose cloud applications.

2486 3.9.2.2.1 grammar

2487 The grammar of the **node_templates** section is a follows:

```
node_templates:
```

<<u>node_template_defn_1</u>>
....

<node_template_defn_n>

2488 **3.9.2.2.2 Example**

2489 Example of **node_templates** section:

node_templates:

```
my_webapp_node_template:
    type: WebApplication
```

my_database_node_template:
 type: Database

2490 3.9.2.3 relationship_templates

2491 The **relationship_templates** section lists the Relationship Templates that describe the relations

- between components that are used to compose cloud applications.
- 2493

- 2494 Note that in the TOSCA Simple Profile, the explicit definition of relationship templates as it was required 2495 in TOSCA v1.0 is optional, since relationships between nodes get implicitly defined by referencing other
- 2496 node templates in the requirements sections of node templates.

2497 3.9.2.3.1 Grammar

2498 The grammar of the **relationship_templates** section is as follows:

relationship_templates:
 <<u>relationship template defn 1</u>>
 ...
 <<u>relationship template defn n</u>>

2499 3.9.2.3.2 Example

2500 Example of **relationship_templates** section:

relationship_templates:

```
my_connectsto_relationship:
  type: tosca.relationships.ConnectsTo
  interfaces:
     Configure:
     inputs:
     speed: { get_attribute: [ SOURCE, connect_speed ] }
```

2501 **3.9.2.4 outputs**

The **outputs** section provides a means to define the output parameters that are available from a TOSCA Simple Profile service template. It allows for exposing attributes of node templates or relationship templates within the containing **topology_template** to users of a service.

2505 **3.9.2.4.1 Grammar**

2506 The grammar of the **outputs** section is as follows:

2507 3.9.2.4.2 Example

2508 Example of the **outputs** section:

outputs:

server_address:

description: The first private IP address for the provisioned server. value: { get_attribute: [HOST, networks, private, addresses, 0] }

2509 3.9.2.5 groups

The **groups** section allows for grouping one or more node templates within a TOSCA Service Template and for assigning special attributes like policies to the group.

2512 3.9.2.5.1 Grammar

2513 The grammar of the groups section is as follows:

```
groups:
    <group_defn_1>
    ...
    <group_defn_n>
```

2514 **3.9.2.5.2 Example**

The following example shows the definition of three Compute nodes in the **node_templates** section of a **topology template** as well as the grouping of two of the Compute nodes in a group **server group 1**.

```
node_templates:
server1:
  type: tosca.nodes.Compute
  # more details ...
server2:
  type: tosca.nodes.Compute
  # more details ...
server3:
  type: tosca.nodes.Compute
  # more details ...
groups:
  # server2 and server3 are part of the same group
  server_group_1:
   type: tosca.groups.Root
   members: [ server2, server3 ]
```

2517 3.9.2.6 policies

- 2518 The **policies** section allows for declaring policies that can be applied to entities in the topology template.
- 2519 3.9.2.6.1 Grammar
- 2520 The grammar of the **policies** section is as follows:

policies:

- <policy defn 1>

- ...

- <policy_defn_n>

2521 3.9.2.6.2 Example

2522 The following example shows the definition of a placement policy.

policies:

- my_placement_policy:
 - type: mycompany.mytypes.policy.placement
- 2523 3.9.2.7 substitution_mapping
- 2524

2525 **3.9.2.7.1 requirement_mapping**

2526 The grammar of a **requirement_mapping** is as follows:

<requirement_name>: [<node_template_name>, <node_template_requirement_name>]

2527 The multi-line grammar is as follows :

```
<requirement_name>:
mapping: [ <node_template_name>, <node_template_capability_name> ]
properties:
<property_name>: <property_value>
```

- requirement_name: represents the name of the requirement as it appears in the Node Type
 definition for the Node Type (name) that is declared as the value for on the
 substitution_mappings' "node_type" key.
- node_template_name: represents a valid name of a Node Template definition (within the same topology_template declaration as the substitution_mapping is declared).
 - **node_template_requirement_name**: represents a valid name of a requirement definition within the <node_template_name> declared in this mapping.
- 2535

2533

2534

2536 **3.9.2.7.2 Example**

2537 The following example shows the definition of a placement policy.

```
topology_template:
inputs:
    cpus:
    type: integer
    constraints:
    less_than: 2 # OR use "defaults" key
```

```
substitution_mappings:
    node_type: MyService
    properties: # Do not care if running or matching (e.g., Compute node)
      # get from outside? Get from contsraint?
      num_cpus: cpus # Implied "PUSH"
      # get from some node in the topology...
      num_cpus: [ <node>, <cap>, <property> ]
      # 1) Running
      architecture:
        # a) Explicit
        value: { get_property: [some_service, architecture] }
        # b) implicit
        value: [ some_service, <req | cap name>, <property name> architecture ]
        default: "amd"
        # c) INPUT mapping?
        ???
      # 2) Catalog (Matching)
      architecture:
         contraints: equals: "x86"
    capabilities:
      bar: [ some_service, bar ]
    requirements:
      foo: [ some_service, foo ]
  node_templates:
    some_service:
      type: MyService
      properties:
        rate: 100
      capabilities:
        bar:
          • • •
      requirements:
        - foo:
            . . .
```

```
2538
```

2539 3.9.2.8 Notes

- The parameters (properties) that are listed as part of the inputs block can be mapped to
 PropertyMappings provided as part of BoundaryDefinitions as described by the TOSCA v1.0
 specification.
- The node templates listed as part of the node_templates block can be mapped to the list of
 NodeTemplate definitions provided as part of TopologyTemplate of a ServiceTemplate as
 described by the TOSCA v1.0 specification.
- The relationship templates listed as part of the relationship_templates block can be mapped to the list of RelationshipTemplate definitions provided as part of TopologyTemplate of a
 ServiceTemplate as described by the TOSCA v1.0 specification.
- The output parameters that are listed as part of the outputs section of a topology template can be mapped to PropertyMappings provided as part of BoundaryDefinitions as described by the TOSCA v1.0 specification.
- 2552oNote, however, that TOSCA v1.0 does not define a direction (input vs. output) for those2553mappings, i.e. TOSCA v1.0 PropertyMappings are underspecified in that respect and2554TOSCA Simple Profile's inputs and outputs provide a more concrete definition of input2555and output parameters.

2556 **3.10 Service Template definition**

A TOSCA Service Template (YAML) document contains element definitions of building blocks for cloud application, or complete models of cloud applications. This section describes the top-level structural elements (TOSCA keynames) along with their grammars, which are allowed to appear in a TOSCA Service Template document.

2561 **3.10.1 Keynames**

2562 The following is the list of recognized keynames for a TOSCA Service Template definition:

Keyname	Required	Туре	Description
tosca_definitions_version	yes	string	Defines the version of the TOSCA Simple Profile specification the template (grammar) complies with.
namespace	no	URI	<pre># illegalities: not alowed to use "tosca" namespaces (reserve tosca domains), SHOULD be unique (some guidance from XML, look to borrow) # describe this in terms of import, by example) # import brings in other STs into <default namespace=""> # on collision its an error (with local type name or on same name from mult. Imports). # Must use prefix or (full) complete name</default></pre>

Keyname	Required	Туре	Description
metadata	no	map of string	Defines a section used to declare additional metadata information. Domain-specific TOSCA profile specifications may define keynames that are required for their implementations.
description	no	description	Declares a description for this Service Template and its contents.
dsl_definitions	no	N/A	Declares optional DSL-specific definitions and conventions. For example, in YAML, this allows defining reusable YAML macros (i.e., YAML alias anchors) for use throughout the TOSCA Service Template.
repositories	no	list of Repository definitions	Declares the list of external repositories which contain artifacts that are referenced in the service template along with their addresses and necessary credential information used to connect to them in order to retrieve the artifacts.
imports	no	list of Import Definitions	Declares import statements external TOSCA Definitions documents. For example, these may be file location or URIs relative to the service template file within the same TOSCA CSAR file.
artifact_types	no	list of Artifact Types	This section contains an optional list of artifact type definitions for use in the service template
data_types	no	list of Data Types	Declares a list of optional TOSCA Data Type definitions.
capability_types	no	list of Capability Types	This section contains an optional list of capability type definitions for use in the service template.
interface_types	no	list of Interface Types	This section contains an optional list of interface type definitions for use in the service template.
relationship_types	no	list of Relationship Types	This section contains a set of relationship type definitions for use in the service template.
node_types	no	list of Node Types	This section contains a set of node type definitions for use in the service template.
group_types	no	list of Group Types	This section contains a list of group type definitions for use in the service template.
policy_types	no	list of Policy Types	This section contains a list of policy type definitions for use in the service template.
topology_template	no	Topology Template definition	Defines the topology template of an application or service, consisting of node templates that represent the application's or service's components, as well as relationship templates representing relations between the components.

2563 3.10.1.1 Metadata keynames

2564 The following is the list of recognized metadata keynames for a TOSCA Service Template definition:

Keyname	Required	Туре	Description
template_name	no	string	Declares a descriptive name for the template.

Keyname	Required	Туре	Description
template_author	no	string	Declares the author(s) or owner of the template.
template_version	no	string	Declares the version string for the template.

2565 **3.10.2 Grammar**

The overall structure of a TOSCA Service Template and its top-level key collations using the TOSCA Simple Profile is shown below:

```
# Required TOSCA Definitions version string
tosca_definitions_version: <value> # Required, see section 3.1 for usage
namespace: <URI>
                                    # Optional, see section 3.2 for usage
# Optional metadata keyname: value pairs
metadata:
  template_name: <value>
                                    # Optional, name of this service template
  template_author: <value>
                                    # Optional, author of this service template
                                    # Optional, version of this service template
  template version: <value>
  # Optional list of domain or profile specific metadata keynames
# Optional description of the definitions inside the file.
description: <template_type_description>
dsl_definitions:
  # list of YAML alias anchors (or macros)
repositories:
  # list of external repository definitions which host TOSCA artifacts
imports:
  # ordered list of import definitions
artifact_types:
  # list of artifact type definitions
data types:
  # list of datatype definitions
capability_types:
  # list of capability type definitions
interface_types
```

```
# list of interface type definitions
relationship_types:
    # list of relationship type definitions
node_types:
    # list of node type definitions
group_types:
    # list of group type definitions
policy_types:
    # list of policy type definitions
topology_template:
    # topology template definition of the cloud application or service
```

2568 3.10.2.1 Requirements

- The URI value "http://docs.oasis-open.org/tosca", as well as all (path) extensions to it, SHALL be reserved for TOSCA approved specifications and work. That means Service Templates that do not originate from a TOSCA approved work product MUST NOT use it, in any form, when declaring a (default) Namespace.
- The key "tosca_definitions_version" SHOULD be the first line of each Service Template.

2574 3.10.2.2 Notes

TOSCA Service Templates do not have to contain a topology_template and MAY contain simply type definitions (e.g., Artifact, Interface, Capability, Node, Relationship Types, etc.) and be imported for use as type definitions in other TOSCA Service Templates.

2578 **3.10.3 Top-level keyname definitions**

2579 3.10.3.1 tosca_definitions_version

This required element provides a means to include a reference to the TOSCA Simple Profile specification within the TOSCA Definitions YAML file. It is an indicator for the version of the TOSCA grammar that should be used to parse the remainder of the document.

2583 3.10.3.1.1 Keyname

tosca_definitions_version

2584 3.10.3.1.2 Grammar

2585 Single-line form:

tosca_definitions_version: <tosca_simple_profile_version>

2586 **3.10.3.1.3 Examples**:

2587 TOSCA Simple Profile version 1.0 specification using the defined namespace alias (see Section 3.1):

tosca_definitions_version: tosca_simple_yaml_1_0

TOSCA Simple Profile version 1.0 specification using the fully defined (target) namespace (see Section 3.1):

tosca_definitions_version: http://docs.oasis-open.org/tosca/ns/simple/yaml/1.0

2590 **3.10.3.2 metadata**

This keyname is used to associate domain-specific metadata with the Service Template. The metadata keyname allows a declaration of a map of keynames with string values.

2593 **3.10.3.2.1 Keyname**

metadata

2594 3.10.3.2.2 Grammar

metadata:

<map_of_string_values>

2595 **3.10.3.2.3 Example**

metadata:

```
creation_date: 2015-04-14
date_updated: 2015-05-01
status: developmental
```

2596

2597 **3.10.3.3 template_name**

- This optional metadata keyname can be used to declare the name of service template as a single-line string value.
- 2600 3.10.3.3.1 Keyname

template_name

2601 3.10.3.3.2 Grammar

template_name: <name string>

2602 3.10.3.3.3 Example

template_name: My service template

2603 3.10.3.3.4 Notes

Some service templates are designed to be referenced and reused by other service templates.
 Therefore, in these cases, the template_name value SHOULD be designed to be used as a
 unique identifier through the use of namespacing techniques.

2607 3.10.3.4 template_author

- 2608 This optional metadata keyname can be used to declare the author(s) of the service template as a single-2609 line string value.
- 2610 3.10.3.4.1 Keyname

template_author

2611 3.10.3.4.2 Grammar

template_author: <author string>

2612 3.10.3.4.3 Example

template_author: My service template

2613 3.10.3.5 template_version

This optional metadata keyname can be used to declare a domain specific version of the service template as a single-line string value.

2616 3.10.3.5.1 Keyname

template_version

2617 3.10.3.5.2 Grammar

template_version: <<u>version</u>>

2618 3.10.3.5.3 Example

template_version: 2.0.17

2619 3.10.3.5.4 Notes:

Some service templates are designed to be referenced and reused by other service templates
 and have a lifecycle of their own. Therefore, in these cases, a template_version value
 SHOULD be included and used in conjunction with a unique template_name value to enable
 lifecycle management of the service template and its contents.

2624 3.10.3.6 description

This optional keyname provides a means to include single or multiline descriptions within a TOSCA Simple Profile template as a scalar string value.

2627 3.10.3.6.1 Keyname

description

2628 3.10.3.7 dsl_definitions

This optional keyname provides a section to define macros (e.g., YAML-style macros when using the TOSCA Simple Profile in YAML specification).

2631 3.10.3.7.1 Keyname

dsl_definitions

2632 3.10.3.7.2 Grammar

dsl_definitions: <<u>dsl_definition_1</u>> ... <<u>dsl_definition_n</u>>

2633 3.10.3.7.3 Example

dsl_definitions:

ubuntu_image_props: &ubuntu_image_props architecture: x86_64 type: linux distribution: ubuntu os_version: 14.04

redhat_image_props: &redhat_image_props
architecture: x86_64
type: linux
distribution: rhel
os version: 6.6

2634 **3.10.3.8 repositories**

This optional keyname provides a section to define external repositories which may contain artifacts or other TOSCA Service Templates which might be referenced or imported by the TOSCA Service Template definition.

2638 3.10.3.8.1 Keyname

repositories

2639 3.10.3.8.2 Grammar

repositories:

<repository definition 1>
...

<repository_definition_n>

2640 **3.10.3.8.3 Example**

repositories:

my_project_artifact_repo:

description: development repository for TAR archives and Bash scripts

url: http://mycompany.com/repository/myproject/

2641 3.10.3.9 imports

This optional keyname provides a way to import a *block sequence* of one or more TOSCA Definitions documents. TOSCA Definitions documents can contain reusable TOSCA type definitions (e.g., Node Types, Relationship Types, Artifact Types, etc.) defined by other authors. This mechanism provides an effective way for companies and organizations to define normative types and/or describe their software applications for reuse in other TOSCA Service Templates.

2647 3.10.3.9.1 Keyname

imports

2648 3.10.3.9.2 Grammar

imports:

- <<u>import_definition_1</u>>

- ...

- <<u>import_definition_n</u>>

2649 **3.10.3.9.3 Example**

An example import of definitions files from a location relative to the

file location of the service template declaring the import.

imports:

- some_definitions: relative_path/my_defns/my_typesdefs_1.yaml
- file: my_defns/my_typesdefs_n.yaml

repository: my_company_repo

namespace_prefix: mycompany

2650 artifact_types

2651 This optional keyname lists the Artifact Types that are defined by this Service Template.

2652 **3.10.3.9.4 Keyname**

artifact_types

2653 3.10.3.9.5 Grammar

artifact_types: <artifact_type_defn_1> ... <artifact_type_defn_n>

2654 3.10.3.9.6 Example

artifact_types:

mycompany.artifacttypes.myFileType: derived_from: tosca.artifacts.File

2655 3.10.3.10 data_types

2656 This optional keyname provides a section to define new data types in TOSCA.

2657 3.10.3.10.1 Keyname

data_types

2658 3.10.3.10.2 Grammar

data_types:

<<u>tosca_datatype_def_1</u>>

•••

<<u>tosca_datatype_def_n</u>>

2659 3.10.3.10.3 Example

data_types:

A complex datatype definition
simple_contactinfo_type:
 properties:
 name:
 type: string
 email:
 type: string

```
phone:
    type: string
# datatype definition derived from an existing type
full_contact_info:
    derived_from: simple_contact_info
    properties:
        street_address:
            type: string
        city:
            type: string
        state:
            type: string
        postalcode:
            type: string
```

2660 3.10.3.11 capability_types

This optional keyname lists the Capability Types that provide the reusable type definitions that can be used to describe features Node Templates or Node Types can declare they support.

2663 3.10.3.11.1 Keyname

capability_types

2664 3.10.3.11.2 Grammar

```
capability_types:
    <<u>capability_type_defn_1</u>>
    ...
    <<u>capability_type_defn_n</u>>
```

2665 **3.10.3.11.3 Example**

```
capability_types:
```

```
mycompany.mytypes.myCustomEndpoint:
  derived_from: tosca.capabilities.Endpoint
  properties:
    # more details ...
```

```
mycompany.mytypes.myCustomFeature:
```

derived_from: tosca.capabilities.Feature

properties:

more details ...

2666 3.10.3.12 interface_types

This optional keyname lists the Interface Types that provide the reusable type definitions that can be used to describe operations for on TOSCA entities such as Relationship Types and Node Types.

2669 3.10.3.12.1 Keyname

interface_types

2670 3.10.3.12.2 Grammar

interface_types:

<<u>interface type defn 1</u>>

• • •

<<u>interface type_defn_n</u>>

2671 3.10.3.12.3 Example

interface_types:

mycompany.interfaces.service.Signal:

signal_begin_receive:

description: Operation to signal start of some message processing.

signal_end_receive:

description: Operation to signal end of some message processed.

2672 3.10.3.13 relationship_types

- This optional keyname lists the Relationship Types that provide the reusable type definitions that can be used to describe dependent relationships between Node Templates or Node Types.
- 2675 3.10.3.13.1 Keyname

relationship_types

2676 3.10.3.13.2 Grammar

relationship_types:

<relationship type defn 1>

• • •

<relationship type_defn_n>

2677 3.10.3.13.3 Example

relationship_types:

mycompany.mytypes.myCustomClientServerType:

derived_from: tosca.relationships.HostedOn

properties:

```
# more details ...
mycompany.mytypes.myCustomConnectionType:
  derived_from: tosca.relationships.ConnectsTo
  properties:
    # more details ...
```

2678 3.10.3.14 node_types

2679 This optional keyname lists the Node Types that provide the reusable type definitions for software 2680 components that Node Templates can be based upon.

2681 3.10.3.14.1 Keyname

node_types

2682 3.10.3.14.2 Grammar

```
node_types:
    <<u>node_type_defn_1</u>>
    ...
    <<u>node_type_defn_n</u>>
```

2683 3.10.3.14.3 Example

node_types:

```
my_webapp_node_type:
    derived_from: WebApplication
    properties:
        my_port:
        type: integer
```

my_database_node_type: derived_from: Database capabilities: mytypes.myfeatures.transactSQL

2684 **3.10.3.14.4 Notes**

• The node types listed as part of the **node_types** block can be mapped to the list of **NodeType** definitions as described by the TOSCA v1.0 specification.

2687 **3.10.3.15 group_types**

2688 This optional keyname lists the Group Types that are defined by this Service Template.

2689 3.10.3.15.1 Keyname

group_types

2690 3.10.3.15.2 Grammar

group_types:
 <group_type_defn_1>
 ...
 <group_type_defn_n>

2691 **3.10.3.15.3 Example**

group_types:

mycompany.mytypes.myScalingGroup: derived_from: tosca.groups.Root

2692 **3.10.3.16 policy_types**

2693 This optional keyname lists the Policy Types that are defined by this Service Template.

2694 3.10.3.16.1 Keyname

policy_types

2695 3.10.3.16.2 Grammar

policy_types:
 <policy_type_defn_1>
 ...
 <policy_type_defn_n>

2696 3.10.3.16.3 Example

policy_types:

mycompany.mytypes.myScalingPolicy: derived_from: tosca.policies.Scaling

2697 **4 TOSCA functions**

Except for the examples, this section is **normative** and includes functions that are supported for use within a TOSCA Service Template.

2700 4.1 Reserved Function Keywords

The following keywords MAY be used in some TOSCA function in place of a TOSCA Node or
Relationship Template name. A TOSCA orchestrator will interpret them at the time the function would be
evaluated at runtime as described in the table below. Note that some keywords are only valid in the
context of a certain TOSCA entity as also denoted in the table.

2705

Keyword	Valid Contexts	Description
SELF	Node Template or Relationship Template	A TOSCA orchestrator will interpret this keyword as the Node or Relationship Template instance that contains the function at the time the function is evaluated.
SOURCE	Relationship Template only.	A TOSCA orchestrator will interpret this keyword as the Node Template instance that is at the source end of the relationship that contains the referencing function.
TARGET	Relationship Template only.	A TOSCA orchestrator will interpret this keyword as the Node Template instance that is at the target end of the relationship that contains the referencing function.
HOST	Node Template only	A TOSCA orchestrator will interpret this keyword to refer to the all nodes that "host" the node using this reference (i.e., as identified by its HostedOn relationship).
		Specifically, TOSCA orchestrators that encounter this keyword when evaluating the get_attribute or get_property functions SHALL search each node along the "HostedOn" relationship chain starting at the immediate node that hosts the node where the function was evaluated (and then that node's host node, and so forth) until a match is found or the "HostedOn" relationship chain ends.

2706

2707 **4.2 Environment Variable Conventions**

2708 4.2.1 Reserved Environment Variable Names and Usage

2709 TOSCA orchestrators utilize certain reserved keywords in the execution environments that

2710 implementation artifacts for Node or Relationship Templates operations are executed in. They are used to

- 2711 provide information to these implementation artifacts such as the results of TOSCA function evaluation or
- 2712 information about the instance model of the TOSCA application
- 2713
- The following keywords are reserved environment variable names in any TOSCA supported executionenvironment:

Keyword	Valid Contexts	Description
TARGETS	Relationship Template only.	 For an implementation artifact that is executed in the context of a relationship, this keyword, if present, is used to supply a list of Node Template instances in a TOSCA application's instance model that are currently target of the context relationship. The value of this environment variable will be a comma-separated list of identifiers of the single target node instances (i.e., the tosca_id attribute of the node).
TARGET	Relationship Template only.	 For an implementation artifact that is executed in the context of a relationship, this keyword, if present, identifies a Node Template instance in a TOSCA application's instance model that is a target of the context relationship, and which is being acted upon in the current operation. The value of this environment variable will be the identifier of the single target node instance (i.e., the tosca_id attribute of the node).
SOURCES	Relationship Template only.	 For an implementation artifact that is executed in the context of a relationship, this keyword, if present, is used to supply a list of Node Template instances in a TOSCA application's instance model that are currently source of the context relationship. The value of this environment variable will be a comma-separated list of identifiers of the single source node instances (i.e., the tosca_id attribute of the node).
SOURCE	Relationship Template only.	 For an implementation artifact that is executed in the context of a relationship, this keyword, if present, identifies a Node Template instance in a TOSCA application's instance model that is a source of the context relationship, and which is being acted upon in the current operation. The value of this environment variable will be the identifier of the single source node instance (i.e., the tosca_id attribute of the node).

2716

For scripts (or implementation artifacts in general) that run in the context of relationship operations, select properties and attributes of both the relationship itself as well as select properties and attributes of the source and target node(s) of the relationship can be provided to the environment by declaring respective operation inputs.

2721

Declared inputs from mapped properties or attributes of the source or target node (selected via the
SOURCE or TARGET keyword) will be provided to the environment as variables having the exact same name
as the inputs. In addition, the same values will be provided for the complete set of source or target nodes,
however prefixed with the ID if the respective nodes. By means of the SOURCES or TARGETS variables
holding the complete set of source or target node IDs, scripts will be able to iterate over corresponding
inputs for each provided ID prefix.

2728

The following example snippet shows an imaginary relationship definition from a load-balancer node to worker nodes. A script is defined for the **add_target** operation of the Configure interface of the relationship, and the **ip_address** attribute of the target is specified as input to the script:

2732

node_templates:

load_balancer:

type: some.vendor.LoadBalancer

requirements:
- member:
relationship: some.vendor.LoadBalancerToMember
interfaces:
Configure:
add_target:
inputs:
<pre>member_ip: { get_attribute: [TARGET, ip_address] }</pre>
<pre>implementation: scripts/configure_members.py</pre>

The add_target operation will be invoked, whenever a new target member is being added to the loadbalancer. With the above inputs declaration, a member_ip environment variable that will hold the IP address of the target being added will be provided to the configure_members.py script. In addition, the IP addresses of all current load-balancer members will be provided as environment variables with a naming scheme of <target node ID>_member_ip. This will allow, for example, scripts that always just write the complete list of load-balancer members into a configuration file to do so instead of updating existing list, which might be more complicated.

- 2740 Assuming that the TOSCA application instance includes five load-balancer members, node1 through
- **node5**, where **node5** is the current target being added, the following environment variables (plus
- 2742 potentially more variables) would be provided to the script:

```
# the ID of the current target and the IDs of all targets
TARGET=node5
TARGETS=node1,node2,node3,node4,node5
# the input for the current target and the inputs of all targets
member_ip=10.0.0.5
node1_member_ip=10.0.0.1
node2_member_ip=10.0.0.2
node3_member_ip=10.0.0.3
node4_member_ip=10.0.0.4
node5 member ip=10.0.0.5
```

2743 With code like shown in the snippet below, scripts could then iterate of all provided member_ip inputs:

```
#!/usr/bin/python
import os
targets = os.environ['TARGETS'].split(',')
for t in targets:
  target_ip = os.environ.get('%s_member_ip' % t)
  # do something with target_ip ...
```

2744 4.2.2 Prefixed vs. Unprefixed TARGET names

The list target node types assigned to the TARGETS key in an execution environment would have names prefixed by unique IDs that distinguish different instances of a node in a running model Future drafts of this specification will show examples of how these names/IDs will be expressed.

2748 **4.2.2.1 Notes**

- Target of interest is always un-prefixed. Prefix is the target opaque ID. The IDs can be used to find the environment var. for the corresponding target. Need an example here.
- If you have one node that contains multiple targets this would also be used (add or remove target operations would also use this you would get set of all current targets).

2753 4.3 Intrinsic functions

2754 These functions are supported within the TOSCA template for manipulation of template data.

2755 **4.3.1 concat**

2756 The **concat** function is used to concatenate two or more string values within a TOSCA service template.

2757 4.3.1.1 Grammar

concat: [<string_value_expressions_*>]

2758 4.3.1.2 Parameters

Parameter	Required	Туре	Description
<string_value_expressions_*></string_value_expressions_*>	yes	list of string or string value expressions	A list of one or more strings (or expressions that result in a string value) which can be concatenated together into a single string.

2759 4.3.1.3 Examples

```
outputs:
    description: Concatenate the URL for a server from other template values
    server_url:
    value: { concat: [ 'http://',
        get_attribute: [ server, public_address ],
        ':',
        get attribute: [ server, port ] ] }
```

2760 **4.3.2 join**

2761 The **join** function is used to join an array of strings into a single string with optional delimiter.

2762 4.3.2.1 Grammar

```
join: [<list of string_value_expressions_*> [ <delimiter> ] ]
```

2763 4.3.2.2 Parameters

Parameter	Required	Туре	Description
<list string_value_expressions_*></list 	yes	list of string or string value expressions	A list of one or more strings (or expressions that result in a list of string values) which can be joined together into a single string.
<delimiter></delimiter>	no	string	An optional delimiter used to join the string in the provided list.

2764 4.3.2.3 Examples

```
outputs:
    example1:
        # Result: prefix_1111_suffix
        value: { join: [ ["prefix", 1111, "suffix" ], "_" ] }
    example2:
        # Result: 9.12.1.10,9.12.1.20
        value: { join: [ { get_input: my_IPs }, "," ] }
```

2765 **4.3.3 token**

- 2766 The **token** function is used within a TOSCA service template on a string to parse out (tokenize)
- 2767 substrings separated by one or more token characters within a larger string.

2768 4.3.3.1 Grammar

token: [<string_with_tokens>, <string_of_token_chars>, <substring_index>]

2769 4.3.3.2 Parameters

Parameter	Required	Туре	Description
string_with_tokens	yes	string	The composite string that contains one or more substrings separated by token characters.
<pre>string_of_token_chars</pre>	yes	string	The string that contains one or more token characters that separate substrings within the composite string.
substring_index	yes	integer	The integer indicates the index of the substring to return from the composite string. Note that the first substring is denoted by using the '0' (zero) integer value.

2770 4.3.3.3 Examples

1]}

2771 **4.4 Property functions**

These functions are used within a service template to obtain property values from property definitions
declared elsewhere in the same service template. These property definitions can appear either directly in
the service template itself (e.g., in the inputs section) or on entities (e.g., node or relationship templates)
that have been modeled within the template.

2776

Note that the **get_input** and **get_property** functions may only retrieve the static values of property definitions of a TOSCA application as defined in the TOSCA Service Template. The **get_attribute** function should be used to retrieve values for attribute definitions (or property definitions reflected as attribute definitions) from the runtime instance model of the TOSCA application (as realized by the TOSCA orchestrator).

2782 **4.4.1 get_input**

The **get_input** function is used to retrieve the values of properties declared within the **inputs** section of a TOSCA Service Template.

2785 4.4.1.1 Grammar

get_input: <input_property_name>

2786 **4.4.1.2 Parameters**

Parameter	Required	Туре	Description
<input_property_name></input_property_name>	yes	string	The name of the property as defined in the inputs section of the service template.

2787 4.4.1.3 Examples

```
inputs:
    cpus:
      type: integer
node_templates:
    my_server:
    type: tosca.nodes.Compute
    capabilities:
      host:
      properties:
      num_cpus: { get_input: cpus }
```

2788 4.4.2 get_property

The **get_property** function is used to retrieve property values between modelable entities defined in the same service template.

2791 4.4.2.1 Grammar

```
get_property: [ <modelable_entity_name>, <optional_req_or_cap_name>,
<property_name>, <nested_property_name_or_index_1>, ...,
<nested_property_name_or_index_n> ]
```

2792 4.4.2.2 Parameters

Parameter	Required	Туре	Description
<modelable entity<br="">name> SELF SOURCE TARGET HOST</modelable>	yes	string	The required name of a modelable entity (e.g., Node Template or Relationship Template name) as declared in the service template that contains the named property definition the function will return the value from. See section B.1 for valid keywords.
<optional_req_or_c ap_name></optional_req_or_c 	no	string	The optional name of the requirement or capability name within the modelable entity (i.e., the <modelable_entity_name> which contains the named property definition the function will return the value from. Note: If the property definition is located in the modelable entity directly, then this parameter MAY be omitted.</modelable_entity_name>
<property_name></property_name>	yes	string	The name of the property definition the function will return the value from.
<nested_property_n ame_or_index_*></nested_property_n 	no	string integer	Some TOSCA properties are complex (i.e., composed as nested structures). These parameters are used to dereference into the names of these nested structures when needed.
			Some properties represent list types. In these cases, an index may be provided to reference a specific entry in the list (as named in the previous parameter) to return.

2793 **4.4.2.3 Examples**

The following example shows how to use the **get_property** function with an actual Node Template name:

```
node_templates:

mysql_database:

type: tosca.nodes.Database

properties:

name: sql_database1

wordpress:

type: tosca.nodes.WebApplication.WordPress

...

interfaces:

Standard:

configure:

inputs:
```

wp_db_name: { get_property: [mysql_database, name] }

2796 The following example shows how to use the get_property function using the SELF keyword:

```
node templates:
  mysql database:
    type: tosca.nodes.Database
    . . .
    capabilities:
      database_endpoint:
        properties:
          port: 3306
  wordpress:
    type: tosca.nodes.WebApplication.WordPress
    requirements:
      - database_endpoint: mysql_database
    interfaces:
      Standard:
        create: wordpress install.sh
        configure:
          implementation: wordpress_configure.sh
          inputs:
            . . .
            wp_db_port: { get_property: [ SELF, database_endpoint, port ] }
```

2797 The following example shows how to use the get_property function using the TARGET keyword:

```
relationship_templates:
    my_connection:
    type: ConnectsTo
    interfaces:
        Configure:
        inputs:
        targets_value: { get_property: [ TARGET, value ] }
```

2798 **4.5 Attribute functions**

These functions (attribute functions) are used within an instance model to obtain attribute values from instances of nodes and relationships that have been created from an application model described in a service template. The instances of nodes or relationships can be referenced by their name as assigned in the service template or relative to the context where they are being invoked.

2803 **4.5.1 get_attribute**

2804 The **get_attribute** function is used to retrieve the values of named attributes declared by the 2805 referenced node or relationship template name.

2806 **4.5.1.1 Grammar**

```
get_attribute: [ <modelable_entity_name>, <optional_req_or_cap_name>,
<attribute_name>, <nested_attribute_name_or_index_1>, ...,
<nested_attribute_name_or_index_n> ]
```

2807 4.5.1.2 Parameters

Parameter	Required	Туре	Description
<modelable entity<br="">name> SELF SOURCE TARGET HOST</modelable>	yes	string	The required name of a modelable entity (e.g., Node Template or Relationship Template name) as declared in the service template that contains the named attribute definition the function will return the value from. See section B.1 for valid keywords.
<optional_req_or_c ap_name></optional_req_or_c 	no	string	The optional name of the requirement or capability name within the modelable entity (i.e., the <modelable_entity_name> which contains the named attribute definition the function will return the value from. Note: If the attribute definition is located in the modelable entity directly, then this parameter MAY be omitted.</modelable_entity_name>
<attribute_name></attribute_name>	yes	string	The name of the attribute definition the function will return the value from.
<nested_attribute_ name_or_index_*></nested_attribute_ 	no	string integer	Some TOSCA attributes are complex (i.e., composed as nested structures). These parameters are used to dereference into the names of these nested structures when needed.
			Some attributes represent list types. In these cases, an index may be provided to reference a specific entry in the list (as named in the previous parameter) to return.

2808 **4.5.1.3 Examples:**

The attribute functions are used in the same way as the equivalent Property functions described above.
Please see their examples and replace "get property" with "get attribute" function name.

2811 4.5.1.4 Notes

These functions are used to obtain attributes from instances of node or relationship templates by the names they were given within the service template that described the application model (pattern).

These functions only work when the orchestrator can resolve to a single node or relationship
 instance for the named node or relationship. This essentially means this is acknowledged to work
 only when the node or relationship template being referenced from the service template has a
 cardinality of 1 (i.e., there can only be one instance of it running).

2818 4.6 Operation functions

These functions are used within an instance model to obtain values from interface operations. These can be used in order to set an attribute of a node instance at runtime or to pass values from one operation to another.

2822 4.6.1 get_operation_output

The **get_operation_output** function is used to retrieve the values of variables exposed / exported from an interface operation.

2825 4.6.1.1 Grammar

get_operation_output: <modelable_entity_name>, <interface_name>,
<operation_name>, <output_variable_name>

2826 4.6.1.2 Parameters

Parameter	Required	Туре	Description
<modelable entity<br="">name> SELF SOURCE TARGET</modelable>	yes	string	The required name of a modelable entity (e.g., Node Template or Relationship Template name) as declared in the service template that implements the named interface and operation.
<interface_name></interface_name>	Yes	string	The required name of the interface which defines the operation.
<operation_name></operation_name>	yes	string	The required name of the operation whose value we would like to retrieve.
<output_variable_ name></output_variable_ 	Yes	string	The required name of the variable that is exposed / exported by the operation.

2827 4.6.1.3 Notes

If operation failed, then ignore its outputs. Orchestrators should allow orchestrators to continue
 running when possible past deployment in the lifecycle. For example, if an update fails, the
 application should be allowed to continue running and some other method would be used to alert
 administrators of the failure.

2832 **4.7 Navigation functions**

• This version of the TOSCA Simple Profile does not define any model navigation functions.

2834 **4.7.1 get_nodes_of_type**

The **get_nodes_of_type** function can be used to retrieve a list of all known instances of nodes of the declared Node Type.

2837 4.7.1.1 Grammar

get_nodes_of_type: <node_type_name>

2838 4.7.1.2 Parameters

Parameter	Required	Туре	Description
<node_type_name></node_type_name>	yes	string	The required name of a Node Type that a TOSCA orchestrator would use to search a running application instance in order to return all unique, named node instances of that type.

2839 4.7.1.3 Returns

Return Key	Туре	Description
TARGETS	<see above></see 	The list of node instances from the current application instance that match the node_type_name supplied as an input parameter of this function.

2840 4.8 Artifact functions

2841 **4.8.1 get_artifact**

The **get_artifact** function is used to retrieve artifact location between modelable entities defined in the same service template.

2844 4.8.1.1 Grammar

get_artifact: [<modelable_entity_name>, <artifact_name>, <location>, <remove>]

2845 **4.8.1.2 Parameters**

Parameter	Required	Туре	Description
<modelable entity name> SELF SOURCE TARGET HOST</modelable 	yes	string	The required name of a modelable entity (e.g., Node Template or Relationship Template name) as declared in the service template that contains the named property definition the function will return the value from. See section B.1 for valid keywords.
<artifact_name></artifact_name>	yes	string	The name of the artifact definition the function will return the value from.
<location> LOCAL_FILE</location>	no	string	Location value must be either a valid path e.g. '/etc/var/my_file' or 'LOCAL_FILE'. If the value is LOCAL_FILE the orchestrator is responsible for providing a path as the result of the get_artifact call where the artifact file can be accessed. The orchestrator will also remove the artifact from this location at the end of the operation. If the location is a path specified by the user the orchestrator is responsible to copy the artifact to the specified location. The orchestrator will return the path as the value of the get_artifact function and leave the file here after the execution of the operation.

Parameter	Required	Туре	Description
remove	no	boolean	Boolean flag to override the orchestrator default behavior so it will remove or not the artifact at the end of the operation execution.
			If not specified the removal will depends of the location e.g. removes it in case of ' LOCAL_FILE ' and keeps it in case of a path.
			If true the artifact will be removed by the orchestrator at the end of the operation execution, if false it will not be removed.

2846 4.8.1.3 Examples

The following example uses a snippet of a WordPress [WordPress] web application to show how to use the get artifact function with an actual Node Template name:

2849 **4.8.1.3.1 Example: Retrieving artifact without specified location**

```
node_templates:
wordpress:
   type: tosca.nodes.WebApplication.WordPress
   ...
   interfaces:
      Standard:
      configure:
           create:
               implementation: wordpress_install.sh
               inputs
                 wp_zip: { get_artifact: [ SELF, zip ] }
   artifacts:
               zip: /data/wordpress.zip
```

2850 In such implementation the TOSCA orchestrator may provide the **wordpress.zip** archive as

2851 2852

2853

- a local URL (example: file://home/user/wordpress.zip) or
- a remote one (example: http://cloudrepo:80/files/wordpress.zip) where some orchestrator may indeed provide some global artifact repository management features.

2854 **4.8.1.3.2 Example: Retrieving artifact as a local path**

- The following example explains how to force the orchestrator to copy the file locally before calling the operation's implementation script:
- 2857

node_templates:

wordpress:

type: tosca.nodes.WebApplication.WordPress

```
...
interfaces:
Standard:
configure:
create:
    implementation: wordpress_install.sh
    inputs
    wp_zip: { get_artifact: [ SELF, zip, LOCAL_FILE] }
artifacts:
zip: /data/wordpress.zip
```

In such implementation the TOSCA orchestrator must provide the wordpress.zip archive as a local path
 (example: /tmp/wordpress.zip) and will remove it after the operation is completed.

2860 4.8.1.3.3 Example: Retrieving artifact in a specified location

The following example explains how to force the orchestrator to copy the file locally to a specific location before calling the operation's implementation script :

2863

```
node_templates:
wordpress:
   type: tosca.nodes.WebApplication.WordPress
   ...
   interfaces:
      Standard:
      configure:
           create:
               implementation: wordpress_install.sh
                    inputs
                    wp_zip: { get_artifact: [ SELF, zip, C:/wpdata/wp.zip ] }
   artifacts:
        zip: /data/wordpress.zip
```

In such implementation the TOSCA orchestrator must provide the wordpress.zip archive as a local path
 (example: C:/wpdata/wp.zip) and will let it after the operation is completed.

4.9 Context-based Entity names (global)

Future versions of this specification will address methods to access entity names based upon the context in which they are declared or defined.

2869 4.9.1.1 Goals

Using the full paths of modelable entity names to qualify context with the future goal of a more
 robust get_attribute function: e.g., get_attribute(<context-based-entity-name>, <attribute name>)

5 TOSCA normative type definitions 2872

2873 Except for the examples, this section is **normative** and contains normative type definitions which 2874 must be supported for conformance to this specification.

2875

2881

2905

2906

2907

2876 The declarative approach is heavily dependent of the definition of basic types that a declarative

- 2877 container must understand. The definition of these types must be very clear such that the
- 2878 operational semantics can be precisely followed by a declarative container to achieve the effects
- 2879 intended by the modeler of a topology in an interoperable manner.

5.1 Assumptions 2880

- Assumes alignment with/dependence on XML normative types proposal for TOSCA v1.1 •
- 2882 Assumes that the normative types will be versioned and the TOSCA TC will preserve backwards 2883 compatibility.
- 2884 Assumes that security and access control will be addressed in future revisions or versions of this • 2885 specification.

5.2 TOSCA normative type names 2886

2887	Every	normative type has three names declared:
2888	1.	Type URI – This is the unique identifying name for the type.
2889		a. These are reserved names within the TOSCA namespace.
2890	2.	Shorthand Name – This is the shorter (simpler) name that can be used in place of its
2891		corresponding, full Type URI name.
2892		a. These are reserved names within TOSCA namespace that MAY be used in place of the
2893		full Type URI.
2894		b. Profiles of the OASIS TOSCA Simple Profile specifcaition SHALL assure non-collision of
2895		names for new types when they are introduced.
2896		c. TOSCA type designers SHOULD NOT create new types with names that would collide
2897		with any TOSCA normative type Shorthand Name.
2898	3.	Type Qualified Name – This is a modified Shorthand Name that includes the "tosca:"
2899		namespace prefix which clearly qualifies it as being part of the TOSCA namespace.
2900		a. This name MAY be used to assure there is no collision when types are imported from
2901		other (non) TOSCA approved sources.
	E 0 4	Additional requirements

5.2.1 Additional requirements 2902

- Case sensitivity TOSCA Type URI, Shorthand and Type Qualified names SHALL be treated as 2903 • 2904 case sensitive.
 - The case of each type name has been carefully selected by the TOSCA working group 0 and TOSCA orchestrators and processors SHALL strictly recognize the name casing as specified in this specification or any of its approved profiles.

5.3 Data Types 2908

5.3.1 tosca.datatypes.Root 2909

2910 This is the default (root) TOSCA Root Type definition that all complex TOSCA Data Types derive from.

2911 **5.3.1.1 Definition**

2912 The TOSCA Root type is defined as follows:

```
tosca.datatypes.Root:
```

description: The TOSCA root Data Type all other TOSCA base Data Types derive from

2913 **5.3.2 tosca.datatypes.json**

- The json type is a TOSCA data Type used to define a string that containst data in the JavaScript Object Notation (JSON) format.
- 2916

Shorthand Name	json	
Type Qualified Name	tosca:json	
Type URI	tosca.datatypes.json	

2917 **5.3.2.1 Definition**

2918 The json type is defined as follows:

tosca.datatypes.json:
 derived_from: string

2919 5.3.2.2 Examples

2920 5.3.2.2.1 Type declaration example

Simple declaration of an 'event_object' property declared to be a 'json' data type with its associated JSON Schema:

```
properties:
  event_object:
    type: json
    constraints:
      schema: >
        {
          "$schema": "http://json-schema.org/draft-04/schema#",
          "title": "Event",
          "description": "Example Event type schema",
          "type": "object",
          "properties": {
            "uuid": {
              "description": "The unique ID for the event.",
              "type": "string"
            },
            "code": {
```

```
"type": "integer"

},

"message": {

"type": "string"

}

},

"required": ["uuid", "code"]

}
```

2923

2924 5.3.2.2.2 Template definition example

This example shows a valid JSON datatype value for the 'event_object' schema declare in the previous example.

```
# properties snippet from a TOSCA template definition.
properties:
    event_object: <
     {
        "uuid": "cadf:1234-56-0000-abcd",
        "code": 9876
    }</pre>
```

2927 5.3.3 Additional Requirements

• The json datatype SHOULD only be assigned string values that contain valid JSON syntax as defined by the "The JSON Data Interchange Format Standard" (see reference **[JSON-Spec]**).

2930 **5.3.4 tosca.datatypes.xml**

The xml type is a TOSCA data Type used to define a string that containst data in the Extensible Markup Language (XML) format.

Shorthand Name	xml
Type Qualified Name	tosca:xml
Type URI	tosca.datatypes.xml

2933 5.3.4.1 Definition

2934 The xml type is defined as follows:

tosca.datatypes.xml:
 derived_from: string

2935 5.3.4.2 Examples

2936 5.3.4.2.1 Type declaration example

Simple declaration of an 'event_object' property declared to be an 'xml' data type with its associated XMLSchema:

```
properties:
  event_object:
   type: xml
    constraints:
      schema: >
        <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"</pre>
          targetNamespace="http://cloudplatform.org/events.xsd"
          xmlns="http://tempuri.org/po.xsd" elementFormDefault="qualified">
          <xs:annotation>
            <xs:documentation xml:lang="en">
              Event object.
            </xs:documentation>
          </xs:annotation>
          <xs:element name="eventObject">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="uuid" type="xs:string"/>
                <xs:element name="code" type="xs:integer"/>
                <xs:element name="message" type="xs:string" minOccurs="0"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:schema>
```

2939

2940 5.3.4.2.2 Template definition example

This example shows a valid XML datatype value for the 'event_object' schema declare in the previous example.

2943 5.3.5 Additional Requirements

The xml datatype SHOULD only be assigned string values that contain valid XML syntax as defined by the "Extensible Markup Language (XML)" specification" (see reference **[XMLSpec]**).

2946 5.3.6 tosca.datatypes.Credential

The Credential type is a complex TOSCA data Type used when describing authorization credentials used to access network accessible resources.

Shorthand Name	Credential	
Type Qualified Name	tosca:Credential	
Type URI	tosca.datatypes.Credential	

2949 5.3.6.1 Properties

Name	Required	Туре	Constraints	Description
protocol	no	string	None	The optional protocol name.
token_type	yes	string	default: password	The required token type.
token	yes	string	None	The required token used as a credential for authorization or access to a networked resource.
keys	no	map of string	None	The optional list of protocol-specific keys or assertions.
user	no	string	None	The optional user (name or ID) used for non-token based credentials.

2950 **5.3.6.2 Definition**

2951 The TOSCA Credential type is defined as follows:

```
tosca.datatypes.Credential:
  derived_from: tosca.datatypes.Root
  properties:
    protocol:
      type: string
      required: false
    token_type:
      type: string
      default: password
    token:
      type: string
    keys:
      type: map
      required: false
      entry_schema:
```

```
type: string
user:
type: string
required: false
```

2952 5.3.6.3 Additional requirements

 TOSCA Orchestrators SHALL interpret and validate the value of the token property based upon the value of the token_type property.

2955 5.3.6.4 Notes

- Specific token types and encoding them using network protocols are not defined or covered in this specification.
- The use of transparent user names (IDs) or passwords are not considered best practice.

2959 **5.3.6.5 Examples**

5.3.6.5.1 Provide a simple user name and password without a protocol or standardized token format

```
<some_tosca_entity>:
```

properties:

my_credential:

type: Credential

properties:

user: myusername

token: mypassword

2962 **5.3.6.5.2 HTTP Basic access authentication credential**

```
<some_tosca_entity>:
properties:
my_credential: # type: Credential
protocol: http
token_type: basic_auth
# Username and password are combined into a string
# Note: this would be base64 encoded before transmission by any impl.
token: myusername:mypassword
```

2963 5.3.6.5.3 X-Auth-Token credential

```
<some_tosca_entity>:
properties:
my_credential: # type: Credential
```

protocol: xauth
token_type: X-Auth-Token
token encoded in Base64
token: 604bbe45ac7143a79e14f3158df67091

2964 5.3.6.5.4 OAuth bearer token credential

```
<some_tosca_entity>:
  properties:
  my_credential: # type: Credential
    protocol: oauth2
    token_type: bearer
    # token encoded in Base64
    token: 8ao9nE2DEjr1zCsicWMpBC
```

2965 5.3.6.6 OpenStack SSH Keypair

```
<some_tosca_entity>:
    properties:
    my_ssh_keypair: # type: Credential
    protocol: ssh
    token_type: identifier
    # token is a reference (ID) to an existing keypair (already installed)
    token: <keypair_id>
```

2966

2967 5.3.7 tosca.datatypes.TimeInterval

The TimeInterval type is a complex TOSCA data Type used when describing a period of time using the YAML ISO 8601 format to declare the start and end times.

Shorthand Name	TimeInterval	
Type Qualified Name	tosca:TimeInterval	
Type URI	tosca.datatypes.TimeInterval	

2970 5.3.7.1 Properties

Name	Required	Туре	Constraints	Description
start_time	yes	timestamp	None	The inclusive start time for the time interval.
end_time	yes	timestamp	None	The inclusive end time for the time interval.

2971 5.3.7.2 Definition

2972 The TOSCA TimeInterval type is defined as follows:

```
tosca.datatypes.TimeInterval:
  derived_from: tosca.datatypes.Root
  properties:
    start_time:
    type: timestamp
    required: true
  end_time:
    type: timestamp
    required: true
```

2973 5.3.7.3 Examples

2974 5.3.7.3.1 Multi-day evaluation time period

```
properties:
  description:
  evaluation_period: Evaluate a service for a 5-day period across time zones
   type: TimeInterval
   start_time: 2016-04-04-15T00:00:00Z
   end_time: 2016-04-08T21:59:43.10-06:00
```

2975 5.3.8 tosca.datatypes.network.NetworkInfo

2976 The Network type is a complex TOSCA data type used to describe logical network information.

Shorthand Name	NetworkInfo	
Type Qualified Name	tosca:NetworkInfo	
Type URI	tosca.datatypes.network.NetworkInfo	

2977 5.3.8.1 Properties

Name	Туре	Constraints	Description
network_name	string	None	The name of the logical network. e.g., "public", "private", "admin". etc.
network_id	string	None	The unique ID of for the network generated by the network provider.
addresses	string []	None	The list of IP addresses assigned from the underlying network.

2978 5.3.8.2 Definition

2979 The TOSCA NetworkInfo data type is defined as follows:

tosca.datatypes.network.NetworkInfo: derived_from: tosca.datatypes.Root properties:

```
network_name:
  type: string
network_id:
  type: string
addresses:
  type: list
  entry_schema:
    type: string
```

2980 **5.3.8.3 Examples**

2981 Example usage of the NetworkInfo data type:

```
<some_tosca_entity>:
  properties:
    private_network:
    network_name: private
    network_id: 3e54214f-5c09-1bc9-9999-44100326da1b
    addresses: [ 10.111.128.10 ]
```

2982 **5.3.8.4 Additional Requirements**

- It is expected that TOSCA orchestrators MUST be able to map the network_name from the
 TOSCA model to underlying network model of the provider.
- The properties (or attributes) of NetworkInfo may or may not be required depending on usage context.

2987 5.3.9 tosca.datatypes.network.PortInfo

2988 The PortInfo type is a complex TOSCA data type used to describe network port information.

Shorthand Name	PortInfo		
Type Qualified Name	tosca:PortInfo		
Type URI	tosca.datatypes.network.PortInfo		

2989 5.3.9.1 Properties

Name	Туре	Constraints	Description
port_name	string	None	The logical network port name.
port_id	string	None	The unique ID for the network port generated by the network provider.
network_id	string	None	The unique ID for the network.
mac_address	string	None	The unique media access control address (MAC address) assigned to the port.
addresses	string []	None	The list of IP address(es) assigned to the port.

2990 5.3.9.2 Definition

2991 The TOSCA PortInfo type is defined as follows:

```
tosca.datatypes.network.PortInfo:
  derived_from: tosca.datatypes.Root
  properties:
    port_name:
      type: string
    port_id:
      type: string
    network_id:
      type: string
    mac_address:
      type: string
    addresses:
      type: list
      entry_schema:
```

2992 **5.3.9.3 Examples**

2993 Example usage of the PortInfo data type:

type: string

```
ethernet_port:
    properties:
        port_name: port1
        port_id: 2c0c7a37-691a-23a6-7709-2d10ad041467
        network_id: 3e54214f-5c09-1bc9-9999-44100326da1b
        mac_address: f1:18:3b:41:92:1e
        addresses: [ 172.24.9.102 ]
```

2994 **5.3.9.4 Additional Requirements**

- It is expected that TOSCA orchestrators MUST be able to map the port_name from the TOSCA model to underlying network model of the provider.
- The properties (or attributes) of PortInfo may or may not be required depending on usage context.

2998 5.3.10 tosca.datatypes.network.PortDef

2999 The PortDef type is a TOSCA data Type used to define a network port.

Shorthand Name	PortDef
Type Qualified Name	tosca:PortDef
Type URI	tosca.datatypes.network.PortDef

3000 5.3.10.1 Definition

3001 The TOSCA PortDef type is defined as follows:

```
tosca.datatypes.network.PortDef:
  derived_from: integer
  constraints:
     - in_range: [ 1, 65535 ]
```

3002 **5.3.10.2 Examples**

3003 Simple usage of a PortDef property type:

properties:
 listen_port: 9090

3004 Example declaration of a property for a custom type based upon PortDef:

properties:
 listen_port:
 type: PortDef
 default: 9000
 constraints:
 - in_range: [9000, 9090]

3005 5.3.11 tosca.datatypes.network.PortSpec

The PortSpec type is a complex TOSCA data Type used when describing port specifications for a network connection.

Shorthand Name	PortSpec	
Type Qualified Name	tosca:PortSpec	
Type URI	tosca.datatypes.network.PortSpec	

3008 **5.3.11.1 Properties**

Name	Required	Туре	Constraints	Description
protocol	yes	string	default: tcp	The required protocol used on the port.
source	no	PortDef	See PortDef	The optional source port.
source_range	no	range	in_range: [1, 65536]	The optional range for source port.
target	no	PortDef	See PortDef	The optional target port.
target_range	no	range	in_range: [1, 65536]	The optional range for target port.

3009 5.3.11.2 Definition

3010 The TOSCA PortSpec type is defined as follows:

```
tosca.datatypes.network.PortSpec:
  derived_from: tosca.datatypes.Root
  properties:
    protocol:
      type: string
      required: true
      default: tcp
      constraints:
        - valid_values: [ udp, tcp, igmp ]
    target:
      type: PortDef
      required: false
    target range:
      type: range
      required: false
      constraints:
        - in_range: [ 1, 65535 ]
    source:
      type: PortDef
      required: false
    source_range:
      type: range
      required: false
      constraints:
        - in_range: [ 1, 65535 ]
```

3011 5.3.11.3 Additional requirements

3012	•	A valid PortSpec MUST have at least one of the following properties: target, target_range,
3013		source or source_range.
3014	•	A valid PortSpec MUST have a value for the source property that is within the numeric range
3015		specified by the property source_range when source_range is specified.
3016	•	A valid PortSpec MUST have a value for the target property that is within the numeric range

3017 specified by the property target_range when target_range is specified.

3018 **5.3.11.4 Examples**

3019 Example usage of the PortSpec data type:

```
# example properties in a node template
some_endpoint:
```

```
properties:
   ports:
    user_port:
        protocol: tcp
        target: 50000
        target_range: [ 20000, 60000 ]
        source: 9000
        source: 1000, 10000 ]
```

3020 **5.4 Artifact Types**

TOSCA Artifacts Types represent the types of packages and files used by the orchestrator when
 deploying TOSCA Node or Relationship Types or invoking their interfaces. Currently, artifacts are
 logically divided into three categories:

3024 3025

3026

- **Deployment Types**: includes those artifacts that are used during deployment (e.g., referenced on create and install operations) and include packaging files such as RPMs, ZIPs, or TAR files.
- Implementation Types: includes those artifacts that represent imperative logic and are used to implement TOSCA Interface operations. These typically include scripting languages such as Bash (.sh), Chef [Chef] and Puppet [Puppet].
 - **Runtime Types**: includes those artifacts that are used during runtime by a service or component of the application. This could include a library or language runtime that is needed by an application such as a PHP or Java library.

3032 3033

3030

3031

3034 **Note**: Additional TOSCA Artifact Types will be developed in future drafts of this specification.

3035 5.4.1 tosca.artifacts.Root

This is the default (root) TOSCA Artifact Type definition that all other TOSCA base Artifact Types derive from.

3038 **5.4.1.1 Definition**

```
tosca.artifacts.Root:
```

description: The TOSCA Artifact Type all other TOSCA Artifact Types derive from

3039 **5.4.2 tosca.artifacts.File**

This artifact type is used when an artifact definition needs to have its associated file simply treated as a file and no special handling/handlers are invoked (i.e., it is not treated as either an implementation or deployment artifact type).

Shorthand Name	File	
Type Qualified Name	tosca:File	
Type URI	tosca.artifacts.File	

3043 **5.4.2.1 Definition**

tosca.artifacts.File:
 derived from: tosca.artifacts.Root

3044 5.4.3 Deployment Types

3045 5.4.3.1 tosca.artifacts.Deployment

This artifact type represents the parent type for all deployment artifacts in TOSCA. This class of artifacts typically represents a binary packaging of an application or service that is used to install/create or deploy it as part of a node's lifecycle.

3049 **5.4.3.1.1 Definition**

tosca.artifacts.Deployment: derived_from: tosca.artifacts.Root description: TOSCA base type for deployment artifacts

3050 **5.4.3.2 Additional Requirements**

TOSCA Orchestrators MAY throw an error if it encounters a non-normative deployment artifact
 type that it is not able to process.

3053 5.4.3.3 tosca.artifacts.Deployment.Image

This artifact type represents a parent type for any "image" which is an opaque packaging of a TOSCA Node's deployment (whether real or virtual) whose contents are typically already installed and preconfigured (i.e., "stateful") and prepared to be run on a known target container.

Shorthand Name	Deployment.Image	
Type Qualified Name	tosca:Deployment.Image	
Type URI	tosca.artifacts.Deployment.Image	

3057 5.4.3.3.1 Definition

tosca.artifacts.Deployment.Image:

derived_from: tosca.artifacts.Deployment

3058 5.4.3.4 tosca.artifacts.Deployment.Image.VM

This artifact represents the parent type for all Virtual Machine (VM) image and container formatted deployment artifacts. These images contain a stateful capture of a machine (e.g., server) including

- 3061 operating system and installed software along with any configurations and can be run on another
- 3062 machine using a hypervisor which virtualizes typical server (i.e., hardware) resources.

3063 **5.4.3.4.1 Definition**

tosca.artifacts.Deployment.Image.VM:

derived_from: tosca.artifacts.Deployment.Image
description: Virtual Machine (VM) Image

3064 **5.4.3.4.2 Notes**

Future drafts of this specification may include popular standard VM disk image (e.g., ISO, VMI,
 VMDX, QCOW2, etc.) and container (e.g., OVF, bare, etc.) formats. These would include
 consideration of disk formats such as:

3068 **5.4.4 Implementation Types**

3069 5.4.4.1 tosca.artifacts.Implementation

3070 This artifact type represents the parent type for all implementation artifacts in TOSCA. These artifacts are 3071 used to implement operations of TOSCA interfaces either directly (e.g., scripts) or indirectly (e.g., config. 3072 files).

3073 5.4.4.1.1 Definition

tosca.artifacts.Implementation:

derived_from: tosca.artifacts.Root

description: TOSCA base type for implementation artifacts

3074 **5.4.4.2 Additional Requirements**

TOSCA Orchestrators MAY throw an error if it encounters a non-normative implementation artifact type that it is not able to process.

3077 5.4.4.3 tosca.artifacts.Implementation.Bash

3078 This artifact type represents a Bash script type that contains Bash commands that can be executed on 3079 the Unix Bash shell.

Shorthand Name	Bash	
Type Qualified Name	tosca:Bash	
Type URI	tosca.artifacts.Implementation.Bash	

3080 5.4.4.3.1 Definition

tosca.artifacts.Implementation.Bash:

derived_from: tosca.artifacts.Implementation
description: Script artifact for the Unix Bash shell
mime_type: application/x-sh

file_ext: [sh]

3081 **5.4.4.4** tosca.artifacts.Implementation.Python

This artifact type represents a Python file that contains Python language constructs that can be executed within a Python interpreter.

Shorthand Name	Python
Type Qualified Name	tosca:Python
Type URI tosca.artifacts.Implementation.Python	

3084 5.4.4.1 Definition

tosca.artifacts.Implementation.Python: derived_from: <u>tosca.artifacts.Implementation</u> description: Artifact for the interpreted Python language mime_type: application/x-python file_ext: [py]

3085 5.5 Capabilities Types

3086 5.5.1 tosca.capabilities.Root

3087This is the default (root) TOSCA Capability Type definition that all other TOSCA Capability Types derive3088from.

3089 5.5.1.1 Definition

```
tosca.capabilities.Root:
    description: The TOSCA root Capability Type all other TOSCA Capability Types
    derive from
```

3090 **5.5.2 tosca.capabilities.Node**

3091 The Node capability indicates the base capabilities of a TOSCA Node Type.

Shorthand Name Node			
Type Qualified Name	tosca:Node		
Type URI	tosca.capabilities.Node		

3092 5.5.2.1 Definition

tosca.capabilities.Node:

derived_from: tosca.capabilities.Root

3093 5.5.3 tosca.capabilities.Compute

The Compute capability, when included on a Node Type or Template definition, indicates that the node can provide hosting on a named compute resource.

Shorthand Name	Compute	
Type Qualified Name	tosca:Compute	
Type URI	tosca.capabilities.Compute	

3096 5.5.3.1 Properties

Name	Required	Туре	Constraints	Description
name	no	string	None	The otional name (or identifier) of a specific compute resource for hosting.
num_cpus	no	integer	greater_or_equal: 1	Number of (actual or virtual) CPUs associated with the Compute node.
cpu_frequency	no	scalar- unit.frequency	greater_or_equal: 0.1 GHz	Specifies the operating frequency of CPU's core. This property expresses the expected frequency of one (1) CPU as provided by the property "num_cpus".
disk_size	no	scalar- unit.size	greater_or_equal: 0 MB	Size of the local disk available to applications running on the Compute node (default unit is MB).
mem_size	no	scalar- unit.size	greater_or_equal: 0 MB	Size of memory available to applications running on the Compute node (default unit is MB).

3097 **5.5.3.2 Definition**

```
tosca.capabilities.Compute:
  derived_from: tosca.capabilities.Container
  shortname: Compute
  properties:
    name:
      type: string
      required: false
    num_cpus:
      type: integer
      required: false
      constraints:
        - greater_or_equal: 1
    cpu_frequency:
      type: scalar-unit.frequency
      required: false
      constraints:
        - greater_or_equal: 0.1 GHz
```

```
disk_size:
  type: scalar-unit.size
  required: false
  constraints:
    - greater_or_equal: 0 MB
mem_size:
  type: scalar-unit.size
  required: false
  constraints:
    - greater_or_equal: 0 MB
```

3098 5.5.4 tosca.capabilities.Network

The Storage capability, when included on a Node Type or Template definition, indicates that the node can provide addressibility for the resource a named network with the specified ports.

Shorthand Name	Network			
Type Qualified Name	tosca:Network			
Type URI	tosca.capabilities.Network			

3101 5.5.4.1 Properties

Name	Required	Туре	Constraints	Description
name	no	string	None	The otional name (or identifier) of a specific network resource.

3102 5.5.4.2 Definition

```
tosca.capabilities.Network:
  derived_from: tosca.capabilities.Root
  properties:
    name:
    type: string
    required: false
```

3103 5.5.5 tosca.capabilities.Storage

The Storage capability, when included on a Node Type or Template definition, indicates that the node can provide a named storage location with specified size range.

Shorthand Name	Storage			
Type Qualified Name	tosca:Storage			
Type URI	tosca.capabilities.Storage			

3106 **5.5.5.1 Properties**

Name	Required	Туре	Constraints	Description
name	no	string	None	The otional name (or identifier) of a specific storage resource.

3107 5.5.5.2 Definition

tosca.capabilities.Storage:

derived_from: tosca.capabilities.Root

properties:

name:

type: string

required: false

3108 **5.5.6 tosca.capabilities.Container**

The Container capability, when included on a Node Type or Template definition, indicates that the node can act as a container for (or a host for) one or more other declared Node Types.

Shorthand Name	Container			
Type Qualified Name	tosca:Container			
Type URI	tosca.capabilities.Container			

3111 5.5.6.1 Properties

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	N/A

3112 5.5.6.2 Definition

```
tosca.capabilities.Container:
    derived_from: tosca.capabilities.Root
```

3113 **5.5.7 tosca.capabilities.Endpoint**

3114 This is the default TOSCA type that should be used or extended to define a network endpoint capability.

3115 This includes the information to express a basic endpoint with a single port or a complex endpoint with

3116 multiple ports. By default the Endpoint is assumed to represent an address on a private network unless 3117 otherwise specified.

Shorthand Name	Endpoint	
Type Qualified Name	tosca:Endpoint	
Type URI	tosca.capabilities.Endpoint	

3118 5.5.7.1 Properties

Name	Required	Туре	Constraints	Description
protocol	yes	string	default: tcp	The name of the protocol (i.e., the protocol prefix) that the endpoint accepts (any OSI Layer 4-7 protocols)
				Examples: http, https, ftp, tcp, udp, etc.
port	no	PortDef	greater_or_equal: 1 less_or_equal: 65535	The optional port of the endpoint.
secure	no	boolean	default: false	Requests for the endpoint to be secure and use credentials supplied on the ConnectsTo relationship.
url_path	no	string	None	The optional URL path of the endpoint's address if applicable for the protocol.
port_name	no	<mark>string</mark>	None	The optional name (or ID) of the network port this endpoint should be bound to.
network_name	no	string	default: PRIVATE	The optional name (or ID) of the network this endpoint should be bound to. network_name: PRIVATE PUBLIC <network_name> <network_id></network_id></network_name>
initiator	no	string	one of: • source • target • peer default: source	The optional indicator of the direction of the connection.
ports	no	map of PortSpec	None	The optional map of ports the Endpoint supports (if more than one)

3119 5.5.7.2 Attributes

Name	Required	Туре	Constraints	Description
ip_address	yes	string	None	Note: This is the IP address as propagated up by the associated node's host (Compute) container.

3120 5.5.7.3 Definition

tosca.capabilities.Endpoint:

derived_from: tosca.capabilities.Root

properties:

```
protocol:
    type: string
    required: true
    default: tcp
  port:
    type: PortDef
    required: false
  secure:
    type: boolean
    required: false
    default: false
  url path:
    type: string
    required: false
  port_name:
    type: string
    required: false
  network_name:
    type: string
    required: false
    default: PRIVATE
  initiator:
    type: string
    required: false
    default: source
    constraints:
      - valid_values: [ source, target, peer ]
  ports:
    type: <u>map</u>
    required: false
    constraints:
      - min_length: 1
    entry_schema:
      type: PortSpec
attributes:
  ip_address:
    type: <u>string</u>
```

3121 **5.5.7.4 Additional requirements**

Although both the port and ports properties are not required, one of port or ports must be provided in a valid Endpoint.

3124 5.5.8 tosca.capabilities.Endpoint.Public

This capability represents a public endpoint which is accessible to the general internet (and its public IP address ranges).

3127 This public endpoint capability also can be used to create a floating (IP) address that the underlying

3128 network assigns from a pool allocated from the application's underlying public network. This floating

address is managed by the underlying network such that can be routed an application's private addressand remains reliable to internet clients.

Shorthand Name	Endpoint.Public
Type Qualified Name	tosca: Endpoint. Public
Type URI	tosca.capabilities.Endpoint.Public

3131 5.5.8.1 Definition

```
tosca.capabilities.Endpoint.Public:
```

derived_from: tosca.capabilities.Endpoint

properties:

Change the default network_name to use the first public network found network name:

type: <u>string</u>

default: PUBLIC

constraints:

- equal: PUBLIC

floating:

description: >

indicates that the public address should be allocated from a pool of floating IPs that are associated with the network.

type: <u>boolean</u>

default: false

status: experimental

```
dns_name:
    description: The optional name to register with DNS
    type: <u>string</u>
```

required: false status: experimental

3132 5.5.8.2 Additional requirements

- If the network_name is set to the reserved value PRIVATE or if the value is set to the name of network (or subnetwork) that is not public (i.e., has non-public IP address ranges assigned to it) then TOSCA Orchestrators SHALL treat this as an error.
- If a dns_name is set, TOSCA Orchestrators SHALL attempt to register the name in the (local)
 DNS registry for the Cloud provider.

3138 5.5.9 tosca.capabilities.Endpoint.Admin

3139 This is the default TOSCA type that should be used or extended to define a specialized administrator 3140 endpoint capability.

Shorthand Name	Endpoint.Admin
Type Qualified Name	tosca:Endpoint.Admin
Type URI	tosca.capabilities.Endpoint.Admin

3141 5.5.9.1 Properties

Name	Required	Туре	Constraints	Description
None	N/A	N/A	N/A	N/A

3142 **5.5.9.2 Definition**

tosca.capabilities.Endpoint.Admin:

derived_from: tosca.capabilities.Endpoint

Change Endpoint secure indicator to true from its default of false

properties:

secure:

type: boolean

default: true

constraints:

- equal: true

3143 5.5.9.3 Additional requirements

• TOSCA Orchestrator implementations of Endpoint.Admin (and connections to it) **SHALL** assure that network-level security is enforced if possible.

3146 **5.5.10 tosca.capabilities.Endpoint.Database**

- 3147 This is the default TOSCA type that should be used or extended to define a specialized database
- 3148 endpoint capability.

Shorthand Name	Endpoint.Database	
Type Qualified Name	tosca:Endpoint.Database	
Type URI	tosca.capabilities.Endpoint.Database	

3149 5.5.10.1 Properties

Name	Required	Туре	Constraints	Description
None	N/A	N/A	N/A	N/A

3150 **5.5.10.2 Definition**

tosca.capabilities.Endpoint.Database:

derived_from: tosca.capabilities.Endpoint

3151 **5.5.11 tosca.capabilities.Attachment**

This is the default TOSCA type that should be used or extended to define an attachment capability of a (logical) infrastructure device node (e.g., BlockStorage node).

Shorthand Name Attachment		
Type Qualified Name	tosca:Attachment	
Type URI	tosca.capabilities.Attachment	

3154 **5.5.11.1 Properties**

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	N/A

3155 **5.5.11.2 Definition**

tosca.capabilities.Attachment:

derived_from: tosca.capabilities.Root

3156 **5.5.12 tosca.capabilities.OperatingSystem**

This is the default TOSCA type that should be used to express an Operating System capability for a node.

Shorthand Name	OperatingSystem	
Type Qualified Name	tosca:OperatingSystem	
Type URI	tosca.capabilities.OperatingSystem	

3159 5.5.12.1 Properties

Name	Required	Туре	Constraints	Description
architecture	no	string	None	The Operating System (OS) architecture.
				Examples of valid values include: x86_32, x86_64, etc.
type	no	string	None	The Operating System (OS) type.
				Examples of valid values include: linux, aix, mac, windows, etc.
distribution	no	string	None	The Operating System (OS) distribution.
				Examples of valid values for an "type" of "Linux" would include: debian, fedora, rhel and ubuntu.

Name	Required	Туре	Constraints	Description
version	no	version	None	The Operating System version.

3160 5.5.12.2 Definition

```
tosca.capabilities.OperatingSystem:
    derived_from: tosca.capabilities.Root
    properties:
        architecture:
        type: string
        required: false
    type:
        type: string
        required: false
    distribution:
        type: string
        required: false
    version:
        type: version
        required: false
```

3161 5.5.12.3 Additional Requirements

Please note that the string values for the properties architecture, type and distribution
 SHALL be normalized to lowercase by processors of the service template for matching purposes.
 For example, if a "type" value is set to either "Linux", "LINUX" or "linux" in a service template, the
 processor would normalize all three values to "linux" for matching purposes.

3166 **5.5.13 tosca.capabilities.Scalable**

3167 This is the default TOSCA type that should be used to express a scalability capability for a node.

Shorthand Name	Scalable
Type Qualified Name	tosca:Scalable
Type URI	tosca.capabilities.Scalable

3168 5.5.13.1 Properties

Name	Required	Туре	Constraints	Description
min_instances	yes	integer	default: 1	This property is used to indicate the minimum number of instances that should be created for the associated TOSCA Node Template by a TOSCA orchestrator.
max_instances	yes	integer	default: 1	This property is used to indicate the maximum number of instances that should be created for the associated TOSCA Node Template by a TOSCA orchestrator.

Name	Required	Туре	Constraints	Description
default_instances	no	integer	N/A	An optional property that indicates the requested default number of instances that should be the starting number of instances a TOSCA orchestrator should attempt to allocate. Note : The value for this property MUST be in the range between the values set for 'min_instances' and 'max_instances' properties.

3169 5.5.13.2 Definition

```
tosca.capabilities.Scalable:
  derived_from: tosca.capabilities.Root
  properties:
    min_instances:
    type: integer
    default: 1
    max_instances:
    type: integer
    default: 1
    default: 1
    default: instances:
    type: integer
```

3170 5.5.13.3 Notes

The actual number of instances for a node may be governed by a separate scaling policy which conceptually would be associated to either a scaling-capable node or a group of nodes in which it is defined to be a part of. This is a planned future feature of the TOSCA Simple Profile and not currently described.

3175 5.5.14 tosca.capabilities.network.Bindable

A node type that includes the Bindable capability indicates that it can be bound to a logical network association via a network port.

Shorthand Name	network.Bindable
Type Qualified tosca:network.Bindable Name Independent of the section of the s	
Type URI	tosca.capabilities.network.Bindable

3178 5.5.14.1 Properties

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	N/A

3179 5.5.14.2 Definition

tosca.capabilities.network.Bindable:

derived_from: tosca.capabilities.Node

3180 **5.6 Requirement Types**

3181 There are no normative Requirement Types currently defined in this working draft. Typically, 3182 Requirements are described against a known Capability Type

3183 5.7 Relationship Types

3184 5.7.1 tosca.relationships.Root

This is the default (root) TOSCA Relationship Type definition that all other TOSCA Relationship Types derive from.

3187 5.7.1.1 Attributes

Name	Required	Туре	Constraints	Description
tosca_id	yes	string	None	A unique identifier of the realized instance of a Relationship Template that derives from any TOSCA normative type.
tosca_name	yes	string	None	This attribute reflects the name of the Relationship Template as defined in the TOSCA service template. This name is not unique to the realized instance model of corresponding deployed application as each template in the model can result in one or more instances (e.g., scaled) when orchestrated to a provider environment.
state	yes	string	default: initial	The state of the relationship instance. See section "Relationship States" for allowed values.

3188 5.7.1.2 Definition

tosca.relationships.Root:

description: The TOSCA root Relationship Type all other TOSCA base Relationship Types derive from

attributes:

tosca_id:

type: string

tosca_name:

type: string

interfaces:

Configure:

type: tosca.interfaces.relationship.Configure

3189 5.7.2 tosca.relationships.DependsOn

3190 This type represents a general dependency relationship between two nodes.

Shorthand Name	DependsOn			
Type Qualified Name	tosca: Depends On			
Type URI	tosca.relationships.DependsOn			

3191 5.7.2.1 Definition

tosca.relationships.DependsOn: derived_from: tosca.relationships.Root valid_target_types: [tosca.capabilities.Node]

3192 5.7.3 tosca.relationships.HostedOn

3193 This type represents a hosting relationship between two nodes.

Shorthand Name	HostedOn		
Type Qualified Name	tosca:HostedOn		
Type URI	tosca.relationships.HostedOn		

3194 **5.7.3.1 Definition**

tosca.relationships.HostedOn: derived_from: <u>tosca.relationships.Root</u> valid_target_types: [<u>tosca.capabilities.Container</u>]

3195 5.7.4 tosca.relationships.ConnectsTo

3196 This type represents a network connection relationship between two nodes.

Shorthand Name	ConnectsTo			
Type Qualified tosca:ConnectsTo Name tosca:ConnectsTo				
Type URI	tosca.relationships.ConnectsTo			

3197 5.7.4.1 Definition

```
tosca.relationships.ConnectsTo:
    derived_from: tosca.relationships.Root
    valid_target_types: [ tosca.capabilities.Endpoint ]
    properties:
        credential:
        type: tosca.datatypes.Credential
        required: false
```

3198 5.7.4.2 Properties

Name	Required	Туре	Constraints	Description
credential	no	Credential	None	The security credential to use to present to the target endpoint to for either authentication or authorization purposes.

3199 **5.7.5 tosca.relationships.AttachesTo**

This type represents an attachment relationship between two nodes. For example, an AttachesTo relationship type would be used for attaching a storage node to a Compute node.

Shorthand Name	AttachesTo
Type Qualified Name	tosca:AttachesTo
Type URI	tosca.relationships.AttachesTo

3202 5.7.5.1 Properties

Name	Required	Туре	Constraints	Description
location	yes	string	min_length: 1	The relative location (e.g., path on the file system), which provides the root location to address an attached node. e.g., a mount point / path such as '/usr/data' Note: The user must provide it and it cannot be "root".
device	no	string	None	The logical device name which for the attached device (which is represented by the target node in the model). e.g., '/dev/hda1'

3203 5.7.5.2 Attributes

Name	Required	Туре	Constraints	Description
device	no	string	None	The logical name of the device as exposed to the instance. Note: A runtime property that gets set when the model gets instantiated by the orchestrator.

3204 5.7.5.3 Definition

```
tosca.relationships.AttachesTo:
    derived_from: tosca.relationships.Root
    valid_target_types: [ tosca.capabilities.Attachment ]
    properties:
        location:
        type: string
        constraints:
            - min_length: 1
        device:
        type: string
```

required: false

3205 5.7.6 tosca.relationships.RoutesTo

3206 This type represents an intentional network routing between two Endpoints in different networks.

Shorthand Name	RoutesTo
Type Qualified Name	tosca:RoutesTo
Type URI	tosca.relationships.RoutesTo

3207 5.7.6.1 Definition

tosca.relationships.RoutesTo:

derived_from: tosca.relationships.ConnectsTo
valid_target_types: [tosca.capabilities.Endpoint]

3208 **5.8 Interface Types**

Interfaces are reusable entities that define a set of operations that that can be included as part of a Node
 type or Relationship Type definition. Each named operations may have code or scripts associated with
 them that orchestrators can execute for when transitioning an application to a given state.

3212 5.8.1 Additional Requirements

- Designers of Node or Relationship types are not required to actually provide/associate code or
 scripts with every operation for a given interface it supports. In these cases, orchestrators SHALL
 consider that a "No Operation" or "no-op".
- The default behavior when providing scripts for an operation in a sub-type (sub-class) or a
 template of an existing type which already has a script provided for that operation SHALL be
 override. Meaning that the subclasses' script is used in place of the parent type's script.

3219 5.8.2 Best Practices

When TOSCA Orchestrators substitute an implementation for an abstract node in a deployed
 service template it SHOULD be able to present a confirmation to the submitter to confirm the
 implementation chosen would be acceptable.

3223 5.8.3 tosca.interfaces.Root

This is the default (root) TOSCA Interface Type definition that all other TOSCA Interface Types derive from.

3226 5.8.3.1 Definition

tosca.interfaces.Root:

derived_from: tosca.entity.Root

description: The TOSCA root Interface Type all other TOSCA Interface Types derive from

3227 5.8.4 tosca.interfaces.node.lifecycle.Standard

3228 This lifecycle interface defines the essential, normative operations that TOSCA nodes may support.

Shorthand Name	Standard
Type Qualified Name	tosca: Standard
Type URI	tosca.interfaces.node.lifecycle.Standard

3229 5.8.4.1 Definition

```
tosca.interfaces.node.lifecycle.Standard:
  derived_from: tosca.interfaces.Root
  create:
    description: Standard lifecycle create operation.
  configure:
    description: Standard lifecycle configure operation.
  start:
    description: Standard lifecycle start operation.
  stop:
    description: Standard lifecycle stop operation.
  delete:
    description: Standard lifecycle stop operation.
  delete:
    description: Standard lifecycle delete operation.
```

3230 5.8.4.2 Create operation

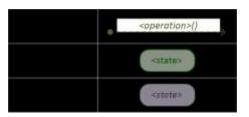
The create operation is generally used to create the resource or service the node represents in the topology. TOSCA orchestrators expect node templates to provide either a deployment artifact or an implementation artifact of a defined artifact type that it is able to process. This specification defines normative deployment and implementation artifact types all TOSCA Orchestrators are expected to be able to process to support application portability.

3236 5.8.4.3 TOSCA Orchestrator processing of Deployment artifacts

- TOSCA Orchestrators, when encountering a deployment artifact on the create operation; will
 automatically attempt to deploy the artifact based upon its artifact type. This means that no
 implementation artifacts (e.g., scripts) are needed on the create operation to provide commands that
- deploy or install the software.
- 3241
- 3242 For example, if a TOSCA Orchestrator is processing an application with a node of type
- 3243 SoftwareComponent and finds that the node's template has a create operation that provides a filename
- 3244 (or references to an artifact which describes a file) of a known TOSCA deployment artifact type such as
- an Open Virtualization Format (OVF) image it will automatically deploy that image into the
 SoftwareComponent's host Compute node.
- · · ·

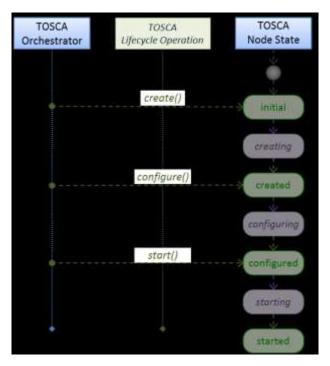
3247 **5.8.4.4 Operation sequencing and node state**

The following diagrams show how TOSCA orchestrators sequence the operations of the Standard lifecycle in normal node startup and shutdown procedures. 3250 The following key should be used to interpret the diagrams:



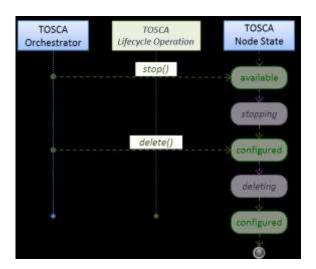
3251 5.8.4.4.1 Normal node startup sequence diagram

The following diagram shows how the TOSCA orchestrator would invoke operations on the Standard lifecycle to startup a node.



3254 5.8.4.4.2 Normal node shutdown sequence diagram

The following diagram shows how the TOSCA orchestrator would invoke operations on the Standard lifecycle to shut down a node.



3257

3258 5.8.5 tosca.interfaces.relationship.Configure

The lifecycle interfaces define the essential, normative operations that each TOSCA Relationship Types may support.

Shorthand Name	Configure
Type Qualified Name	tosca:Configure
Type URI	tosca.interfaces.relationship.Configure

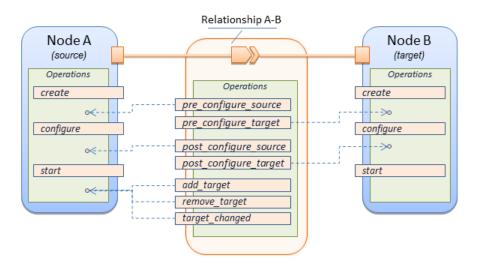
3261 5.8.5.1 Definition

```
tosca.interfaces.relationship.Configure:
  derived_from: tosca.interfaces.Root
  pre_configure_source:
    description: Operation to pre-configure the source endpoint.
  pre_configure_target:
    description: Operation to pre-configure the target endpoint.
  post configure source:
    description: Operation to post-configure the source endpoint.
  post_configure_target:
    description: Operation to post-configure the target endpoint.
  add target:
    description: Operation to notify the source node of a target node being added
via a relationship.
  add source:
    description: Operation to notify the target node of a source node which is
now available via a relationship.
    description:
  target changed:
    description: Operation to notify source some property or attribute of the
target changed
```

```
remove_target:
    description: Operation to remove a target node.
```

3262

3263 **5.8.5.2 Invocation Conventions**



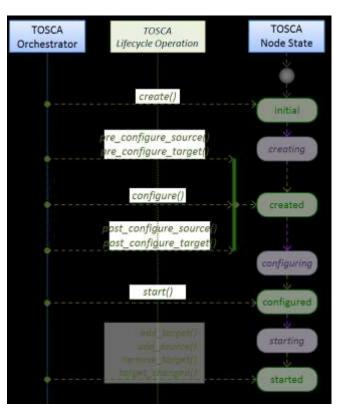
3264 TOSCA relationships are directional connecting a source node to a target node. When TOSCA

3265 Orchestrator connects a source and target node together using a relationship that supports the Configure 3266 interface it will "interleave" the operations invocations of the Configure interface with those of the node's

3267 own Standard lifecycle interface. This concept is illustrated below:

3268 5.8.5.3 Normal node start sequence with Configure relationship operations

The following diagram shows how the TOSCA orchestrator would invoke Configure lifecycle operations in conjunction with Standard lifecycle operations during a typical startup sequence on a node.



3271 **5.8.5.4 Node-Relationship configuration sequence**

- 3272 Depending on which side (i.e., source or target) of a relationship a node is on, the orchestrator will:
 - Invoke either the pre_configure_source or pre_configure_target operation as supplied by the relationship on the node.
 - Invoke the node's **configure** operation.
 - Invoke either the **post_configure_source** or **post_configure_target** as supplied by the relationship on the node.
- 3278 Note that the **pre_configure_xxx** and **post_configure_xxx** are invoked only once per node instance.

3279 5.8.5.4.1 Node-Relationship add, remove and changed sequence

- Since a topology template contains nodes that can dynamically be added (and scaled), removed or
 changed as part of an application instance, the Configure lifecycle includes operations that are invoked
 on node instances that to notify and address these dynamic changes.
- 3283

3273

3274

3275

3276

3277

For example, a source node, of a relationship that uses the Configure lifecycle, will have the relationship operations add_target, or remove_target invoked on it whenever a target node instance is added or removed to the running application instance. In addition, whenever the node state of its target node changes, the target_changed operation is invoked on it to address this change. Conversely, the add_source and remove_source operations are invoked on the source node of the relationship.

3289 5.8.5.5 Notes

- The target (provider) MUST be active and running (i.e., all its dependency stack MUST be fulfilled) prior to invoking add_target
 In other words, all Requirements MUST be satisfied before it advertises its capabilities (i.e.,
 - In other words, all Requirements MUST be satisfied before it advertises its capabilities (i.e., the attributes of the matched Capabilities are available).
 - In other words, it cannot be "consumed" by any dependent node.
 - Conversely, since the source (consumer) needs information (attributes) about any targets (and their attributes) being removed before it actually goes away.
- The remove_target operation should only be executed if the target has had add_target
 executed. BUT in truth we're first informed about a target in pre_configure_source, so if we
 execute that the source node should see remove_target called to cleanup.
 - **Error handling**: If any node operation of the topology fails processing should stop on that node template and the failing operation (script) should return an error (failure) code when possible.

3302 **5.9 Node Types**

3303 5.9.1 tosca.nodes.Root

3304 The TOSCA Root Node Type is the default type that all other TOSCA base Node Types derive from.

3305 This allows for all TOSCA nodes to have a consistent set of features for modeling and management (e.g.,

3306 consistent definitions for requirements, capabilities and lifecycle interfaces).

3307

3293

3294

3295

3296

3300

3301

Shorthand Name	Root
Type Qualified Name	tosca:Root
Type URI	tosca.nodes.Root

3308 5.9.1.1 Properties

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	The TOSCA Root Node type has no specified properties.

3309 5.9.1.2 Attributes

Name	Required	Туре	Constraints	Description
tosca_id	yes	string	None	A unique identifier of the realized instance of a Node Template that derives from any TOSCA normative type.
tosca_name	yes	string	None	This attribute reflects the name of the Node Template as defined in the TOSCA service template. This name is not unique to the realized instance model of corresponding deployed application as each template in the model can result in one or more instances (e.g., scaled) when orchestrated to a provider environment.
state	yes	string	default: initial	The state of the node instance. See section "Node States" for allowed values.

3310 5.9.1.3 Definition

```
tosca.nodes.Root:
  derived_from: tosca.entity.Root
  description: The TOSCA Node Type all other TOSCA base Node Types derive from
  attributes:
    tosca_id:
      type: string
    tosca name:
      type: string
    state:
      type: string
  capabilities:
    feature:
      type: tosca.capabilities.Node
  requirements:
    - dependency:
        capability: tosca.capabilities.Node
        node: tosca.nodes.Root
        relationship: tosca.relationships.DependsOn
        occurrences: [ 0, UNBOUNDED ]
  interfaces:
    Standard:
      type: tosca.interfaces.node.lifecycle.Standard
```

3311 **5.9.1.4 Additional Requirements**

• All Node Type definitions that wish to adhere to the TOSCA Simple Profile **SHOULD** extend from the TOSCA Root Node Type to be assured of compatibility and portability across implementations.

3314 5.9.2 tosca.nodes.Abstract.Compute

- 3315 The TOSCA Abstract.Compute node represents an abstract compute resource without any requirements
- on storage or network resources.
- 3317

Shorthand Name	Abstract.Compute
Type Qualified Name	tosca:Abstract.Compute
Type URI	tosca.nodes.Abstract.Compute

3318 5.9.2.1 Properties

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	N/A

3319 5.9.2.2 Attributes

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	N/A

3320 5.9.2.3 Definition

```
tosca.nodes.Abstract.Compute:
  derived_from: tosca.nodes.Root
  capabilities:
    host:
    type: tosca.capabilities.Compute
    valid_source_types: []
```

3321 5.9.3 tosca.nodes.Compute

The TOSCA Compute node represents one or more real or virtual processors of software applications or
 services along with other essential local resources. Collectively, the resources the compute node
 represents can logically be viewed as a (real or virtual) "server".

Shorthand Name	Compute
Type Qualified Name	tosca:Compute
Type URI	tosca.nodes.Compute

3325 5.9.3.1 Properties

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	N/A

3326 5.9.3.2 Attributes

Name	Required	Туре	Constraints	Description
private_address	no	string	None	The primary private IP address assigned by the cloud provider that applications may use to access the Compute node.
public_address	no	string	None	The primary public IP address assigned by the cloud provider that applications may use to access the Compute node.
networks	no	map of NetworkI nfo	None	The list of logical networks assigned to the compute host instance and information about them.
ports	no	map of PortInfo	None	The list of logical ports assigned to the compute host instance and information about them.

3327 5.9.3.3 Definition

tosca.nodes.Compute:

```
derived_from: tosca.nodes.Abstract.Compute
attributes:
 private_address:
   type: string
 public_address:
   type: string
 networks:
   type: map
    entry schema:
      type: tosca.datatypes.network.NetworkInfo
 ports:
   type: map
    entry schema:
      type: tosca.datatypes.network.PortInfo
requirements:
  - local storage:
      capability: tosca.capabilities.Attachment
      node: tosca.nodes.BlockStorage
      relationship: tosca.relationships.AttachesTo
      occurrences: [0, UNBOUNDED]
capabilities:
 host:
   type: tosca.capabilities.Compute
    valid_source_types: [tosca.nodes.SoftwareComponent]
 endpoint:
   type: tosca.capabilities.Endpoint.Admin
 os:
   type: tosca.capabilities.OperatingSystem
 scalable:
   type: tosca.capabilities.Scalable
 binding:
   type: tosca.capabilities.network.Bindable
```

3328 5.9.3.4 Additional Requirements

The underlying implementation of the Compute node SHOULD have the ability to instantiate
 guest operating systems (either actual or virtualized) based upon the OperatingSystem capability
 properties if they are supplied in the a node template derived from the Compute node type.

3332 5.9.4 tosca.nodes.SoftwareComponent

3333The TOSCA SoftwareComponent node represents a generic software component that can be managed3334and run by a TOSCA Compute Node Type.

Shorthand Name	SoftwareComponent					
Type Qualified Name	tosca:SoftwareComponent					
Type URI	tosca.nodes.SoftwareComponent					

3335 5.9.4.1 Properties

Name	Required	Туре	Constraints	Description
component_version	no	version	None	The optional software component's version.
admin_credential	no	Credential	None	The optional credential that can be used to authenticate to the software component.

3336 5.9.4.2 Attributes

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	N/A

3337 5.9.4.3 Definition

```
tosca.nodes.SoftwareComponent:
    derived_from: tosca.nodes.Root
    properties:
        # domain-specific software component version
        component_version:
        type: version
        required: false
        admin_credential:
        type: tosca.datatypes.Credential
        required: false
    required: false
    requirements:
        - host:
        capability: tosca.capabilities.Compute
        node: tosca.nodes.Compute
        relationship: tosca.relationships.HostedOn
```

3338 5.9.4.4 Additional Requirements

Nodes that can directly be managed and run by a TOSCA Compute Node Type SHOULD extend
 from this type.

3341 5.9.5 tosca.nodes.WebServer

3342 This TOSA WebServer Node Type represents an abstract software component or service that is capable

3343 of hosting and providing management operations for one or more **WebApplication** nodes.

Shorthand Name	hand Name WebServer					
Type Qualified Name	tosca:WebServer					
Type URI	tosca.nodes.WebServer					

3344 5.9.5.1 Properties

Name	Required	Туре	Constraints	Description
None	N/A	N/A	N/A	N/A

3345 **5.9.5.2 Definition**

tosca.nodes.WebServer: derived_from: tosca.nodes.SoftwareComponent capabilities: # Private, layer 4 endpoints data_endpoint: tosca.capabilities.Endpoint admin_endpoint: tosca.capabilities.Endpoint.Admin host: type: tosca.capabilities.Compute valid_source_types: [tosca.nodes.WebApplication]

3346 5.9.5.3 Additional Requirements

• This node **SHALL** export both a secure endpoint capability (i.e., **admin_endpoint**), typically for 3348 administration, as well as a regular endpoint (i.e., **data_endpoint**) for serving data.

3349 **5.9.6 tosca.nodes.WebApplication**

3350The TOSCA WebApplication node represents a software application that can be managed and run by a3351TOSCA WebServer node. Specific types of web applications such as Java, etc. could be derived from3352this type.

Shorthand Name	WebApplication
Type Qualified Name	tosca: WebApplication
Type URI	tosca.nodes.WebApplication

3353 5.9.6.1 Properties

Name	Required	Туре	Constraints	Description
context_root	no	string	None	The web application's context root which designates the application's URL path within the web server it is hosted on.

3354 **5.9.6.2 Definition**

```
tosca.nodes.WebApplication:
    derived_from: tosca.nodes.Root
    properties:
        context_root:
        type: string
    capabilities:
        app_endpoint:
        type: tosca.capabilities.Endpoint
    requirements:
        - host:
        capability: tosca.capabilities.Compute
        node: tosca.nodes.WebServer
        relationship: tosca.relationships.HostedOn
```

3355 5.9.7 tosca.nodes.DBMS

The TOSCA **DBMS** node represents a typical relational, SQL Database Management System software component or service.

3358 **5.9.7.1 Properties**

Name	Required	Туре	Constraints	Description
root_password	no	string	None	The optional root password for the DBMS server.
port	no	integer	None	The DBMS server's port.

3359 **5.9.7.2 Definition**

```
tosca.nodes.DBMS:
  derived_from: tosca.nodes.SoftwareComponent
  properties:
    root_password:
    type: string
    required: false
    description: the optional root password for the DBMS service
    port:
        type: integer
        required: false
        description: the port the DBMS service will listen to for data and requests
    capabilities:
    host:
        type: tosca.capabilities.Compute
        valid_source_types: [ tosca.nodes.Database ]
```

3360 **5.9.8 tosca.nodes.Database**

The TOSCA **Database** node represents a logical database that can be managed and hosted by a TOSCA **DBMS** node.

Shorthand Name	Database				
Type Qualified Name	tosca: Database				
Type URI	tosca.nodes.Database				

3363 5.9.8.1 Properties

Name	Required	Туре	Constraints	Description
name	yes	string	None	The logical database Name
port	no	integer	None	The port the database service will use to listen for incoming data and requests.
user	no	string	None	The special user account used for database administration.
password	no	string	None	The password associated with the user account provided in the 'user' property.

3364 **5.9.8.2 Definition**

```
tosca.nodes.Database:
  derived_from: tosca.nodes.Root
  properties:
    name:
      type: string
      description: the logical name of the database
    port:
      type: integer
      description: the port the underlying database service will listen to for
data
    user:
      type: string
      description: the optional user account name for DB administration
      required: false
    password:
      type: string
      description: the optional password for the DB user account
      required: false
  requirements:
    - host:
        capability: tosca.capabilities.Compute
        node: tosca.nodes.DBMS
```

relationship: tosca.relationships.HostedOn

capabilities:

database_endpoint:

type: tosca.capabilities.Endpoint.Database

3365 5.9.9 tosca.nodes.Abstract.Storage

3366 The TOSCA **Abstract.Storage** node represents an abstract storage resource without any requirements 3367 on compute or network resources.

Shorthand Name	AbstractStorage
Type Qualified Name	tosca:Abstract.Storage
Type URI	tosca.nodes.Abstract.Storage

3368 5.9.9.1 Properties

Name	Required	Туре	Constraints	Description
name	yes	string	None	The logical name (or ID) of the storage resource.
size	no	scalar- unit.size	greater_or_equa l: 0 MB	The requested initial storage size (default unit is in Gigabytes).

3369 5.9.9.2 Definition

```
tosca.nodes.Abstract.Storage:
  derived_from: tosca.nodes.Root
  properties:
    name:
    type: string
    size:
    type: scalar-unit.size
    default: 0 MB
    constraints:
        - greater_or_equal: 0 MB
    capabilities:
    # TBD
```

3370 5.9.10 tosca.nodes.Storage.ObjectStorage

The TOSCA **ObjectStorage** node represents storage that provides the ability to store data as objects (or BLOBs of data) without consideration for the underlying filesystem or devices.

Shorthand Name	ObjectStorage
Type Qualified Name	tosca:ObjectStorage
Type URI	tosca.nodes.Storage.ObjectStorage

3373 5.9.10.1 Properties

Name	Required	Туре	Constraints	Description
maxsize	no	scalar- unit.size	greater_or_equa l: 1GB	The requested maximum storage size (default unit is in Gigabytes).

3374 5.9.10.2 Definition

```
tosca.nodes.Storage.ObjectStorage:
  derived_from: tosca.nodes.Abstract.Storage
  properties:
  maxsize:
    type: scalar-unit.size
    constraints:
    - greater or equal: 0 GB
```

capabilities:

storage_endpoint:

type: tosca.capabilities.Endpoint

3375 **5.9.10.3 Notes:**

- Subclasses of the tosca.nodes.ObjectStorage node type may impose further constraints on properties. For example, a subclass may constrain the (minimum or maximum) length of the 'name' property or include a regular expression to constrain allowed characters used in the 'name' property.
- 3380 5.9.11 tosca.nodes.Storage.BlockStorage
- 3381 The TOSCA **BlockStorage** node currently represents a server-local block storage device (i.e., not 3382 shared) offering evenly sized blocks of data from which raw storage volumes can be created.
- **Note**: In this draft of the TOSCA Simple Profile, distributed or Network Attached Storage (NAS) are not yet considered (nor are clustered file systems), but the TC plans to do so in future drafts.

Shorthand Name	BlockStorage
Type Qualified Name	tosca:BlockStorage
Type URI	tosca.nodes.Storage.BlockStorage

3385

3386 **5.9.11.1 Properties**

Name	Required	Туре	Constraints	Description
size	yes *	scalar- unit.size	greater_or_e qual: 1 MB	 The requested storage size (default unit is MB). * Note: Required when an existing volume (i.e., volume_id) is not available. If volume_id is provided, size is ignored. Resize of existing volumes is not considered at this time.
volume_id	no	string	None	ID of an existing volume (that is in the accessible scope of the requesting application).
snapshot_id	no	string	None	Some identifier that represents an existing snapshot that should be used when creating the block storage (volume).

3387 5.9.11.2 Attributes

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	N/A

3388 5.9.11.3 Definition

```
tosca.nodes.Storage.BlockStorage:
    derived_from: tosca.nodes.Abstract.Storage
    properties:
        volume_id:
        type: string
        required: false
        snapshot_id:
        type: string
        required: false
capabilities:
        attachment:
        type: tosca.capabilities.Attachment
```

3389 5.9.11.4 Additional Requirements

• The **size** property is required when an existing volume (i.e., **volume_id**) is not available. 3391 However, if the property **volume_id** is provided, the **size** property is ignored.

3392 5.9.11.5 Notes

• Resize is of existing volumes is not considered at this time.

- It is assumed that the volume contains a single filesystem that the operating system (that is hosting an associate application) can recognize and mount without additional information (i.e., it is operating system independent).
 - Currently, this version of the Simple Profile does not consider regions (or availability zones) when modeling storage.

3399 5.9.12 tosca.nodes.Container.Runtime

3400 The TOSCA **Container** Runtime node represents operating system-level virtualization technology used 3401 to run multiple application services on a single Compute host.

Shorthand Name	Container.Runtime
Type Qualified Name	tosca:Container.Runtime
Type URI	tosca.nodes.Container.Runtime

3402 **5.9.12.1 Definition**

3397 3398

```
tosca.nodes.Container.Runtime:
  derived_from: <u>tosca.nodes.SoftwareComponent</u>
  capabilities:
    host:
    type: <u>tosca.capabilities.Compute</u>
    valid_source_types: [<u>tosca.nodes.Container.Application</u>]
    scalable:
    type: <u>tosca.capabilities.Scalable</u>
```

3403 5.9.13 tosca.nodes.Container.Application

3404 The TOSCA **Container** Application node represents an application that requires **Container**-level 3405 virtualization technology.

Shorthand Name	Container.Application
Type Qualified Name	tosca:Container.Application
Type URI	tosca.nodes.Container.Application

3406 5.9.13.1 Definition

tosca.nodes.Container.Application:

relationship: tosca.relationships.HostedOn

- storage:

capability: tosca.capabilities.Storage

- network: capability: tosca.capabilities.EndPoint

3407 **5.9.14 tosca.nodes.LoadBalancer**

The TOSCA Load Balancer node represents logical function that be used in conjunction with a Floating
 Address to distribute an application's traffic (load) across a number of instances of the application (e.g.,
 for a clustered or scaled application).

Shorthand Name	LoadBalancer
Type Qualified Name	tosca:LoadBalancer
Type URI tosca.nodes.LoadBalancer	

3411 5.9.14.1 Definition

```
tosca.nodes.LoadBalancer:
  derived_from: tosca.nodes.Root
  properties:
    algorithm:
      type: string
      required: false
      status: experimental
  capabilities:
    client:
      type: tosca.capabilities.Endpoint.Public
      occurrences: [0, UNBOUNDED]
      description: the Floating (IP) client's on the public network can connect
to
  requirements:
    - application:
        capability: tosca.capabilities.Endpoint
        relationship: tosca.relationships.RoutesTo
        occurrences: [0, UNBOUNDED]
        description: Connection to one or more load balanced applications
```

3412 **5.9.14.2 Notes:**

A LoadBalancer node can still be instantiated and managed independently of any applications it would serve; therefore, the load balancer's application requirement allows for zero occurrences.

3416 **5.10 Group Types**

TOSCA Group Types represent logical groupings of TOSCA nodes that have an implied membership
 relationship and may need to be orchestrated or managed together to achieve some result. Some use

- 3419 cases being developed by the TOSCA TC use groups to apply TOSCA policies for software placement
- 3420 and scaling while other use cases show groups can be used to describe cluster relationships.
- 3421
- 3422 Note: Additional normative TOSCA Group Types and use cases for them will be developed in future3423 drafts of this specification.

3424 5.10.1 tosca.groups.Root

This is the default (root) TOSCA Group Type definition that all other TOSCA base Group Types derive from.

3427 5.10.1.1 Definition

tosca.groups.Root:

description: The TOSCA Group Type all other TOSCA Group Types derive from interfaces:

Standard:

type: tosca.interfaces.node.lifecycle.Standard

3428 **5.10.1.2 Notes:**

- Group operations are not necessarily tied directly to member nodes that are part of a group.
- Future versions of this specification will create sub types of the tosca.groups.Root type that will
 describe how Group Type operations are to be orchestrated.

3432 **5.11 Policy Types**

TOSCA Policy Types represent logical grouping of TOSCA nodes that have an implied relationship and
 need to be orchestrated or managed together to achieve some result. Some use cases being developed
 by the TOSCA TC use groups to apply TOSCA policies for software placement and scaling while other
 use cases show groups can be used to describe cluster relationships.

3437 5.11.1 tosca.policies.Root

This is the default (root) TOSCA Policy Type definition that all other TOSCA base Policy Types derive from.

3440 **5.11.1.1 Definition**

```
tosca.policies.Root:
```

description: The TOSCA Policy Type all other TOSCA Policy Types derive from

3441 5.11.2 tosca.policies.Placement

This is the default (root) TOSCA Policy Type definition that is used to govern placement of TOSCA nodes or groups of nodes.

3444 5.11.2.1 Definition

tosca.policies.Placement:

derived_from: tosca.policies.Root

description: The TOSCA Policy Type definition that is used to govern placement of TOSCA nodes or groups of nodes.

3445 5.11.3 tosca.policies.Scaling

- This is the default (root) TOSCA Policy Type definition that is used to govern scaling of TOSCA nodes or groups of nodes.
- 3448 **5.11.3.1 Definition**

tosca.policies.Scaling:

derived_from: tosca.policies.Root

description: The TOSCA Policy Type definition that is used to govern scaling of TOSCA nodes or groups of nodes.

3449 5.11.4 tosca.policies.Update

3450 This is the default (root) TOSCA Policy Type definition that is used to govern update of TOSCA nodes or 3451 groups of nodes.

3452 **5.11.4.1 Definition**

tosca.policies.Update:

derived_from: tosca.policies.Root

description: The TOSCA Policy Type definition that is used to govern update of TOSCA nodes or groups of nodes.

3453 5.11.5 tosca.policies.Performance

This is the default (root) TOSCA Policy Type definition that is used to declare performance requirements for TOSCA nodes or groups of nodes.

3456 **5.11.5.1 Definition**

tosca.policies.Performance:

derived_from: tosca.policies.Root

description: The TOSCA Policy Type definition that is used to declare performance requirements for TOSCA nodes or groups of nodes.

3457

3458 6 TOSCA Cloud Service Archive (CSAR) format

Except for the examples, this section is **normative** and defines changes to the TOSCA archive format relative to the TOSCA v1.0 XML specification.

3461

TOSCA Simple Profile definitions along with all accompanying artifacts (e.g. scripts, binaries,
 configuration files) can be packaged together in a CSAR file as already defined in the TOSCA version 1.0
 specification [TOSCA-1.0]. In contrast to the TOSCA 1.0 CSAR file specification (see chapter 16 in
 [TOSCA-1.0]), this simple profile makes a few simplifications both in terms of overall CSAR file structure
 as well as meta-file content as described below.

3467 6.1 Overall Structure of a CSAR

3468 A CSAR zip file is required to contain one of the following:

- a TOSCA-Metadata directory, which in turn contains the TOSCA.meta metadata file that provides
 entry information for a TOSCA orchestrator processing the CSAR file.
- a yaml (.yml or .yaml) file at the root of the archive. The yaml file being a valid tosca definition
 template that MUST define a metadata section where template_name and template_version are
 required.

The CSAR file may contain other directories with arbitrary names and contents. Note that in contrast to the TOSCA 1.0 specification, it is not required to put TOSCA definitions files into a special "Definitions" directory, but definitions YAML files can be placed into any directory within the CSAR file.

3477 6.2 TOSCA Meta File

3478 The **TOSCA.meta** file structure follows the exact same syntax as defined in the TOSCA 1.0 specification.

3479 However, it is only required to include *block_0* (see section 16.2 in [TOSCA-1.0]) with the Entry-

3480 **Definitions** keyword pointing to a valid TOSCA definitions YAML file that a TOSCA orchestrator should 3481 use as entry for parsing the contents of the overall CSAR file.

3482 Note that it is not required to explicitly list TOSCA definitions files in subsequent blocks of the

3483 **TOSCA.meta** file, but any TOSCA definitions files besides the one denoted by the **Entry-Definitions** 3484 keyword can be found by a TOSCA orchestrator by processing respective **imports** statements in the 3485 entry definitions file (or in recursively imported files).

Note also that any additional artifact files (e.g. scripts, binaries, configuration files) do not have to be
 declared explicitly through blocks in the TOSCA.meta file. Instead, such artifacts will be fully described and
 pointed to by relative path names through artifact definitions in one of the TOSCA definitions files
 contained in the CSAR.

3490 Due to the simplified structure of the CSAR file and **TOSCA.meta** file compared to TOSCA 1.0, the **CSAR**-3491 **Version** keyword listed in *block_0* of the meta-file is required to denote version **1.1**.

3492 6.2.1 Example

3493 The following listing represents a valid **TOSCA.meta** file according to this TOSCA Simple Profile 3494 specification.

> TOSCA-Meta-File-Version: 1.0 CSAR-Version: 1.1 Created-By: OASIS TOSCA TC Entry-Definitions: definitions/tosca_elk.yaml

- 3496 This **TOSCA.meta** file indicates its simplified TOSCA Simple Profile structure by means of the **CSAR**-
- 3497 **Version** keyword with value **1.1**. The **Entry-Definitions** keyword points to a TOSCA definitions
- 3498 YAML file with the name **tosca_elk.yaml** which is contained in a directory called **definitions** within 3499 the root of the CSAR file.
- 3500 6.3 Archive without TOSCA-Metadata
- In case the archive doesn't contains a TOSCA-Metadata directory the archive is required to contains a single YAML file at the root of the archive (other templates may exits in sub-directories).
- This file must be a valid TOSCA definitions YAML file with the additional restriction that the metadata section (as defined in 3.9.3.2) is required and template_name and template_version metadata are also required.
- 3506 TOSCA processors should recognized this file as being the CSAR Entry-Definitions file. The CSAR-3507 Version is defined by the template version metadata section. The Created-By value is defined by the
- 3508 template_author metadata.

3509 6.3.1 Example

The following represents a valid TOSCA template file acting as the CSAR Entry-Definitions file in an archive without TOSCA-Metadata directory.

```
tosca_definitions_version: tosca_simple_yaml_1_1
```

metadata:

template_name: my_template
template_author: OASIS TOSCA TC
template version: 1.0

3512

3513 **7 TOSCA workflows**

TOSCA defines two different kinds of workflows that can be used to deploy (instantiate and start), manage at runtime or undeploy (stop and delete) a TOSCA topology: declarative workflows and imperative workflows. Declarative workflows are automatically generated by the TOSCA orchestrator based on the nodes, relationships, and groups defined in the topology. Imperative workflows are manually specified by the author of the topology and allows the specification of any use-case that has not been planned in the definition of node and relationships types or for advanced use-case (including reuse of existing scripts and workflows).

3521

Workflows can be triggered on deployment of a topology (deploy workflow) on undeployment (undeploy workflow) or during runtime, manually, or automatically based on policies defined for the topology.

3524

3532

Note: The TOSCA orchestrators will execute a single workflow at a time on a topology to guarantee that the defined workflow can be consistent and behave as expected.

3527 7.1 Normative workflows

TOSCA defines several normative workflows that are used to operate a Topology. That is, reserved names of workflows that should be preserved by TOSCA orchestrators and that, if specified in the topology will override the workflow generated by the orchestrator :

- **deploy**: is the workflow used to instantiate and perform the initial deployment of the topology.
 - **undeploy**: is the workflow used to remove all instances of a topology.

3533 7.1.1 Notes

Future versions of the specification will describe the normative naming and declarative generation of additional workflows used to operate the topology at runtime.

- scaling workflows: defined for every scalable nodes or based on scaling policies
- **auto-healing workflows**: defined in order to restart nodes that may have failed

3538 7.2 Declarative workflows

3539 Declarative workflows are the result of the weaving of topology's node, relationships, and groups3540 workflows.

The weaving process generates the workflow of every single node in the topology, insert operations from the relationships and groups and finally add ordering consideration. The weaving process will also take care of the specific lifecycle of some nodes and the TOSCA orchestrator is responsible to trigger errors or warnings in case the weaving cannot be processed or lead to cycles for example.

This section aims to describe and explain how a TOSCA orchestrator will generate a workflow based on the topology entities (nodes, relationships and groups).

3547 7.2.1 Notes

3548 This section details specific constraints and considerations that applies during the weaving process.

3549 7.2.1.1 Orchestrator provided nodes lifecycle and weaving

When a node is abstract the orchestrator is responsible for providing a valid matching resources for the node in order to deploy the topology. This consideration is also valid for dangling requirements (as they represents a quick way to define an actual node).

- The lifecycle of such nodes is the responsibility of the orchestrator and they may not answer to the normative TOSCA lifecycle. Their workflow is considered as "delegate" and acts as a black-box between the initial and started state in the install workflow and the started to deleted states in the uninstall workflow.
- 3557 If a relationship to some of this node defines operations or lifecycle dependency constraint that relies on 3558 intermediate states, the weaving SHOULD fail and the orchestrator SHOULD raise an error.

3559 7.2.2 Relationship impacts on topology weaving

This section explains how relationships impacts the workflow generation to enable the composition of complex topologies.

3562 7.2.2.1 tosca.relationships.DependsOn

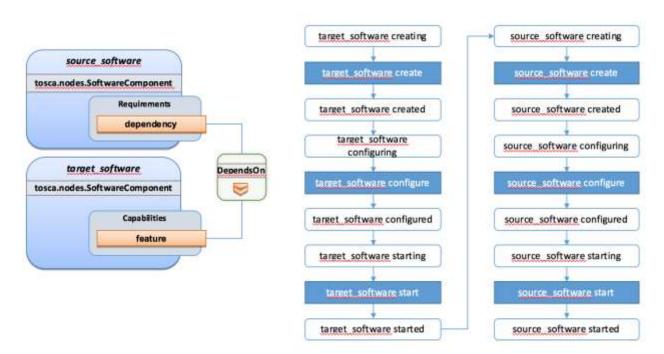
The depends on relationship is used to establish a dependency from a node to another. A source node that depends on a target node will be created only after the other entity has been started.

3565 7.2.2.2 Note

- 3566 DependsOn relationship SHOULD not be implemented. Even if the Configure interface can be
- implemented this is not considered as a best-practice. If you need specific implementation, please have alook at the ConnectsTo relationship.

3569 7.2.2.2.1 Example DependsOn

- This example show the usage of a generic DependsOn relationship between two custom software components.
- 3572



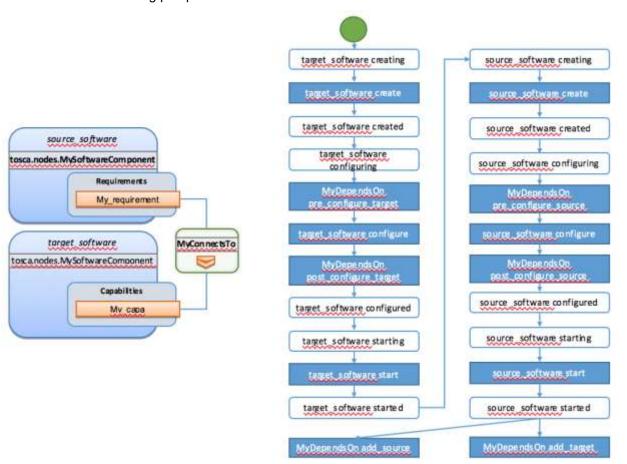
3573

3574 In this example the relationship configure interface doesn't define operations so they don't appear in the 3575 generated lifecycle.

3576 7.2.2.3 tosca.relationships.ConnectsTo

The connects to relationship is similar to the DependsOn relationship except that it is intended to provide an implementation. The difference is more theoretical than practical but helps users to make an actual

3579 distinction from a meaning perspective.



3580

3581 **7.2.2.4 tosca.relationships.HostedOn**

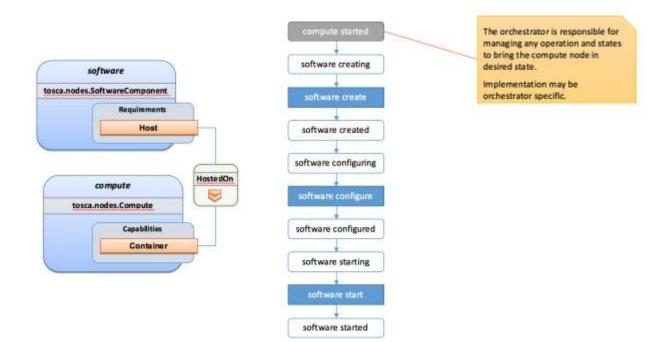
The hosted_on dependency relationship allows to define a hosting relationship between an entity and another. The hosting relationship has multiple impacts on the workflow and execution:

- The implementation artifacts of the source node is executed on the same host as the one of the target node.
- The create operation of the source node is executed only once the target node reach the started state.
- When multiple nodes are hosted on the same host node, the defined operations will not be
 executed concurrently even if the theoretical workflow could allow it (actual generated workflow
 will avoid concurrency).

3591 7.2.2.4.1 Example Software Component HostedOn Compute

This example explain the TOSCA weaving operation of a custom SoftwareComponent on a tosca.nodes.Compute instance. The compute node is an orchestrator provided node meaning that it's lifecycle is delegated to the orchestrator. This is a black-box and we just expect a started compute node to be provided by the orchestrator.

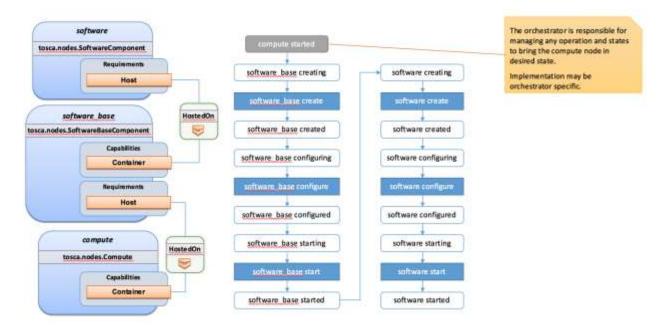
3596 The software node lifecycle operations will be executed on the Compute node (host) instance.



3599 7.2.2.4.2 Example Software Component HostedOn Software Component

Tosca allows some more complex hosting scenarios where a software component could be hosted on another software component.

3602



3603

In such scenarios the software create operation is triggered only once the software_base node hasreached the started state.

3597

3606 7.2.2.4.3 Example 2 Software Components HostedOn Compute

This example illustrate concurrency constraint introduced by the management of multiple nodes on a single compute.

3609 **7.2.3 Limitations**

3610 7.2.3.1 Hosted nodes concurrency

3611 TOSCA implementation currently does not allow concurrent executions of scripts implementation artifacts

3612 (shell, python, ansible, puppet, chef etc.) on a given host. This limitation is not applied on multiple hosts.

3613 This limitation is expressed through the HostedOn relationship limitation expressing that when multiple 3614 components are hosted on a given host node then their operations will not be performed concurrently

3615 (generated workflow will ensure that operations are not concurrent).

3616 7.2.3.2 Dependent nodes concurrency

When a node depends on another node no operations will be processed concurrently. In some situations, especially when the two nodes lies on different hosts we could expect the create operation to be executed concurrently for performance optimization purpose. The current version of the specification will allow to use imperative workflows to solve this use-case. However, this scenario is one of the scenario that we want to improve and handle in the future through declarative workflows.

3622 7.2.3.3 Target operations and get attribute on source

The current ConnectsTo workflow implies that the target node is started before the source node is even created. This means that pre_configure_target and post_configure_target operations cannot use any input based on source attribute. It is however possible to refer to get_property inputs based on source properties. For advanced configurations the add_source operation should be used.

Note also that future plans on declarative workflows improvements aims to solve this kind of issues while it is currently possible to use imperative workflows.

3629 **7.3 Imperative workflows**

Imperative workflows are user defined and can define any really specific constraints and ordering of
 activities. They are really flexible and powerful and can be used for any complex use-case that cannot be
 solved in declarative workflows. However, they provide less reusability as they are defined for a specific
 topology rather than being dynamically generated based on the topology content.

3634 **7.3.1 Defining sequence of operations in an imperative workflow**

- Imperative workflow grammar defines two ways to define the sequence of operations in an imperativeworkflow:
- Leverage the **on_success** definition to define the next steps that will be executed in parallel.
- Leverage a sequence of activity in a step.

3639 7.3.1.1 Using on_success to define steps ordering

- The graph of workflow steps is build based on the values of **on_success** elements of the various defined steps. The graph is built based on the following rules:
- All steps that defines an on_success operation must be executed before the next step can be executed. So if A and C defines an on_success operation to B, then B will be executed only when both A and C have been successfully executed.
- The multiple nodes defined by an **on_success** construct can be executed in parallel.

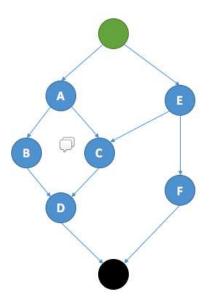
- Every step that doesn't have any predecessor is considered as an initial step and can run in parallel.
- Every step that doesn't define any successor is considered as final. When all the final nodes
 executions are completed then the workflow is considered as completed.

3650 7.3.1.1.1 Example

- 3651 The following example defines multiple steps and the **on_success** relationship between them.
- 3652

```
topology_template:
  workflows:
    deploy:
      description: Workflow to deploy the application
      steps:
        A:
          on_success:
             - B
             - C
        Β:
          on_success:
             - D
        C:
          on_success:
             - D
        D:
        E:
          on success:
             - C
             - F
        F:
```

3653 The following schema is the visualization of the above definition in term of sequencing of the steps.



3655 7.3.1.2 Define a sequence of activity on the same element

3656 The step definition of a TOSCA imperative workflow allows multiple activities to be defined :

3657

<pre>workflows: my_workflow:</pre>
steps:
create_my_node:
target: my_node
activities:
- set_state: creating
 call_operation: tosca.interfaces.node.lifecycle.Standard.create
- set_state: created

- 3658 The sequence defined here defines three different activities that will be performed in a sequential way.
- 3659 This is just equivalent to writing multiple steps chained by an on_success together :

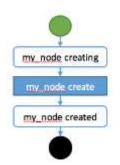
3660 3661

```
workflows:
my_workflow:
steps:
creating_my_node:
target: my_node
activities:
- set_state: creating
on_success: create_my_node
create_my_node:
target: my_node
```

```
activities:
        - call_operation: tosca.interfaces.node.lifecycle.Standard.create
        on_success: created_my_node
        created_my_node:
        target: my_node
        activities:
        - set_state: created
```

3663 In both situations the resulting workflow is a sequence of activities:

3664



3665

3666 7.3.2 Definition of a simple workflow

Imperative workflow allow user to define custom workflows allowing them to add operations that are not
 normative, or for example, to execute some operations in parallel when TOSCA would have performed
 sequential execution.

3670

As Imperative workflows are related to a topology, adding a workflow is as simple as adding a workflows section to your topology template and specifying the workflow and the steps that compose it.

3673 7.3.2.1 Example: Adding a non-normative custom workflow

3674 This sample topology add a very simple custom workflow to trigger the mysql backup operation.

```
topology_template:
  node_templates:
    my_server:
    type: tosca.nodes.Compute
    mysql:
    type: tosca.nodes.DBMS.MySQL
    requirements:
        - host: my_server
        interfaces:
        tosca.interfaces.nodes.custom.Backup:
        operations:
            backup: backup.sh
```

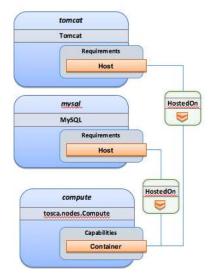
```
workflows:
```

```
backup:
description: Performs a snapshot of the MySQL data.
steps:
    my_step:
    target: mysql
    activities:
        - call_operation: tosca.interfaces.nodes.custom.Backup.backup
```

In such topology the TOSCA container will still use declarative workflow to generate the deploy andundeploy workflows as they are not specified and a backup workflow will be available for user to trigger.

3678 7.3.2.2 Example: Creating two nodes hosted on the same compute in parallel

TOSCA declarative workflow generation constraint the workflow so that no operations are called in
parallel on the same host. Looking at the following topology this means that the mysql and tomcat nodes
will not be created in parallel but sequentially. This is fine in most of the situations as packet managers
like apt or yum doesn't not support concurrency, however if both create operations performs a download
of zip package from a server most of people will hope to do that in parallel in order to optimize throughput.

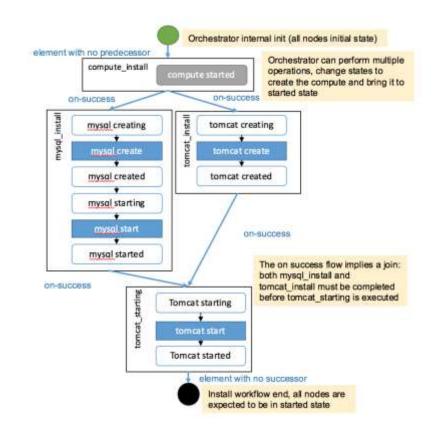


3684

Imperative workflows can help to solve this issue. Based on the above topology we will design a workflow
 that will create tomcat and mysql in parallel but we will also ensure that tomcat is started after mysql is

3687 started even if no relationship is defined between the components:

3688



- 3690
- 3691 To achieve such workflow, the following topology will be defined:

3692

```
topology_template:
  node_templates:
    my_server:
      type: tosca.nodes.Compute
    mysql:
      type: tosca.nodes.DBMS.MySQL
      requirements:
        - host: my_server
    tomcat:
      type: tosca.nodes.WebServer.Tomcat
      requirements:
        - host: my_server
  workflows:
    deploy:
      description: Override the TOSCA declarative workflow with the following.
      steps:
        compute_install
          target: my_server
```

```
activities:
    - delegate: deploy
 on success:
    - mysql_install
    - tomcat install
tomcat install:
 target: tomcat
 activities:
    - set_state: creating
    - call_operation: tosca.interfaces.node.lifecycle.Standard.create
    - set state: created
 on_success:
    - tomcat_starting
mysql_install:
 target: mysql
 activities:
    - set_state: creating
    - call_operation: tosca.interfaces.node.lifecycle.Standard.create
    - set state: created
    - set_state: starting
    - call operation: tosca.interfaces.node.lifecycle.Standard.start
    - set state: started
 on_success:
    - tomcat_starting
tomcat_starting:
 target: tomcat
 activities:
    - set_state: starting
    - call operation: tosca.interfaces.node.lifecycle.Standard.start
    - set_state: started
```

```
3693
```

3694 **7.3.3 Specifying preconditions to a workflow**

Pre conditions allows the TOSCA orchestrator to determine if a workflow can be executed based on the states and attribute values of the topology's node. Preconditions must be added to the initial workflow.

3697 7.3.3.1 Example : adding precondition to custom backup workflow

In this example we will use precondition so that we make sure that the mysql node is in the correct state for a backup.

```
topology template:
  node templates:
    my server:
      type: tosca.nodes.Compute
    mysql:
      type: tosca.nodes.DBMS.MySQL
      requirements:
        - host: my server
      interfaces:
        tosca.interfaces.nodes.custom.Backup:
          operations:
            backup: backup.sh
  workflows:
    backup:
      description: Performs a snapshot of the MySQL data.
      preconditions:
        - target: my server
          condition:
            - assert:
              - state: [{equal: available}]
        - target: mysql
          condition:
            - assert:
              - state: [{valid values: [started, available]}]
              - my_attribute: [{equal: ready }]
      steps:
        my_step:
          target: mysql
          activities:
            - call operation: tosca.interfaces.nodes.custom.Backup.backup
```

When the backup workflow will be triggered (by user or policy) the TOSCA engine will first check that preconditions are fulfilled. In this situation the engine will check that *my_server* node is in *available* state AND that *mysql* node is in *started* OR *available* states AND that *mysql my_attribute* value is equal to *ready*.

3704 **7.3.4 Workflow reusability**

TOSCA allows the reusability of a workflow in other workflows. Such concepts can be achieved thanks to the inline activity.

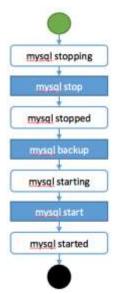
3707 7.3.4.1 Reusing a workflow to build multiple workflows

- 3708 The following example show how a workflow can inline an existing workflow and reuse it.
- 3709

```
topology_template:
  node templates:
    my server:
      type: tosca.nodes.Compute
    mysql:
      type: tosca.nodes.DBMS.MySQL
      requirements:
        - host: my server
      interfaces:
        tosca.interfaces.nodes.custom.Backup:
          operations:
            backup: backup.sh
  workflows:
   start_mysql:
      steps:
        start mysql:
          target: mysql
          activities :
            - set_state: starting
            - call_operation: tosca.interfaces.node.lifecycle.Standard.start
            - set_state: started
    stop_mysql:
      steps:
        stop_mysql:
          target: mysql
          activities:
            - set_state: stopping
            - call_operation: tosca.interfaces.node.lifecycle.Standard.stop
            - set state: stopped
    backup:
      description: Performs a snapshot of the MySQL data.
      preconditions:
        - target: my_server
          condition:
            - assert:
              - state: [{equal: available}]
        - target: mysql
          condition:
            - assert:
              - state: [{valid_values: [started, available]}]
```

```
- my_attribute: [{equal: ready }]
steps:
    backup_step:
    activities:
        inline: stop
        call_operation: tosca.interfaces.nodes.custom.Backup.backup
        inline: start
restart:
    steps:
    backup_step:
    activities:
        inline: stop
        inline: stop
        inline: start
```

- 3711 The example above defines three workflows and show how the start_mysql and stop_mysql workflows
- are reused in the backup and restart workflows.
- Inlined workflows are inlined sequentially in the existing workflow for example the backup workflow wouldlook like this:



3715

3716 7.3.4.2 Inlining a complex workflow

3717 It is possible of course to inline more complex workflows. The following example defines an inlined3718 workflows with multiple steps including concurrent steps:

3719

topology_template: workflows: inlined_wf: steps:

17 January 2019 Page 231 of 354

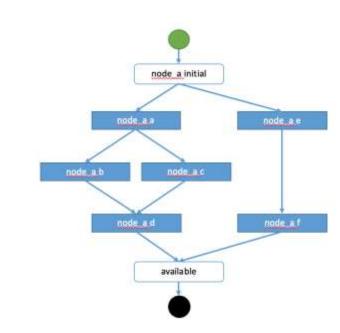
```
A:
      target: node_a
      activities:
        - call_operation: a
      on_success:
        - B
        - C
    B:
      target: node_a
      activities:
        - call_operation: b
      on_success:
        - D
    C:
      target: node_a
      activities:
        - call_operation: c
      on success:
        - D
    D:
      target: node_a
      activities:
        - call_operation: d
    Ε:
      target: node_a
      activities:
        - call_operation: e
      on_success:
        - C
        - F
    F:
      target: node_a
      activities:
        - call_operation: f
main_workflow:
  steps:
    G:
      target: node_a
      activities:
        - set_state: initial
        - inline: inlined_wf
```

- set_state: available

To describe the following workflow:

3720 3721

3722



3723

3724 7.3.5 Defining conditional logic on some part of the workflow

Preconditions are used to validate if the workflow should be executed only for the initial workflow. If a
workflow that is inlined defines some preconditions theses preconditions will be used at the instance level
to define if the operations should be executed or not on the defined instance.

3728

This construct can be used to filter some steps on a specific instance or under some specificcircumstances or topology state.

3731

```
topology_template:
  node_templates:
    my_server:
    type: tosca.nodes.Compute
    cluster:
    type: tosca.nodes.DBMS.Cluster
    requirements:
        - host: my_server
        interfaces:
        tosca.interfaces.nodes.custom.Backup:
        operations:
        backup: backup.sh
```

17 January 2019 Page 233 of 354

```
workflows:
 backup:
    description: Performs a snapshot of the MySQL data.
    preconditions:
      - target: my_server
        condition:
          - assert:
            - state: [{equal: available}]
      - target: mysql
        condition:
          - assert:
            - state: [{valid values: [started, available]}]
            - my_attribute: [{equal: ready }]
    steps:
      backup_step:
        target: cluster
        filter: # filter is a list of clauses. Matching between clauses is and.
          - or: # only one of sub-clauses must be true.
            - assert:
              - foo: [{equals: true}]
            - assert:
              - bar: [{greater_than: 2}, {less_than: 20}]
        activities:
          - call_operation: tosca.interfaces.nodes.custom.Backup.backup
```

3733 7.3.6 Define inputs for a workflow

Inputs can be defined in a workflow and will be provided in the execution context of the workflow. If an
operation defines a get_input function on one of its parameter the input will be retrieved from the workflow
input, and if not found from the topology inputs.

```
3737
```

Workflow inputs will never be configured from policy triggered workflows and SHOULD be used only for user triggered workflows. Of course operations can still refer to topology inputs or template properties or attributes even in the context of a policy triggered workflow.

3741 7.3.6.1 Example

```
topology_template:
node_templates:
    my_server:
        type: tosca.nodes.Compute
    mysql:
        type: tosca.nodes.DBMS.MySQL
```

```
requirements:
        - host: my server
      interfaces:
        tosca.interfaces.nodes.custom.Backup:
          operations:
            backup:
              implementation: backup.sh
              inputs:
                storage_url: { get_input: storage_url }
workflows:
    backup:
      description: Performs a snapshot of the MySQL data.
      preconditions:
        - target: my server
          valid states: [available]
        - target: mysql
          valid states: [started, available]
          attributes:
            my attribute: [ready]
      inputs:
        storage_url:
          type: string
      steps:
        my_step:
          target: mysql
          activities:
            - call_operation: tosca.interfaces.nodes.custom.Backup.backup
```

```
3742
```

To trigger such a workflow, the TOSCA engine must allow user to provide inputs that match the given definitions.

3745 7.3.7 Handle operation failure

- By default, failure of any activity of the workflow will result in the failure of the workflow and will results in stopping the steps to be executed.
- 3748
- Exception: uninstall workflow operation failure SHOULD not prevent the other operations of the workflow to run (a failure in an uninstall script SHOULD not prevent from releasing resources from the cloud).
- 3751

3752 For any workflow other than install and uninstall failures may leave the topology in an unknown state. In

3753 such situation the TOSCA engine may not be able to orchestrate the deployment. Implementation of

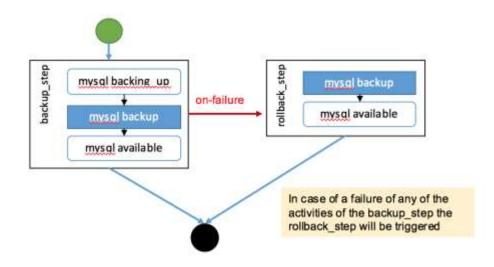
on_failure construct allows to execute rollback operations and reset the state of the affected entities

3755 back to an orchestrator known state.

3756 7.3.7.1 Example

```
topology_template:
  node_templates:
    my_server:
      type: tosca.nodes.Compute
    mysql:
      type: tosca.nodes.DBMS.MySQL
      requirements:
        - host: my_server
      interfaces:
        tosca.interfaces.nodes.custom.Backup:
          operations:
            backup:
              implementation: backup.sh
              inputs:
                storage_url: { get_input: storage_url }
  workflows:
    backup:
      steps:
        backup_step:
          target: mysql
          activities:
            - set_state: backing_up # this state is not a TOSCA known state
            - call_operation: tosca.interfaces.nodes.custom.Backup.backup
            - set_state: available # this state is known by TOSCA orchestrator
          on failure:
            - rollback step
        rollback step:
          target: mysql
          activities:
            - call_operation: tosca.interfaces.nodes.custom.Backup.backup
            - set_state: available # this state is known by TOSCA orchestrator
```

3757



3759

3760 7.4 Making declarative more flexible and imperative more generic

TOSCA simple profile 1.1 version provides the genericity and reusability of declarative workflows that is
 designed to address most of use-cases and the flexibility of imperative workflows to address more
 complex or specific use-cases.

3764

Each approach has some pros and cons and we are working so that the next versions of the specification can improve the workflow usages to try to allow more flexibility in a more generic way. Two non-exclusive leads are currently being discussed within the working group and may be included in the future versions of the specification.

- Improvement of the declarative workflows in order to allow people to extend the weaving logic of
 TOSCA to fit some specific need.
- Improvement of the imperative workflows in order to allow partial imperative workflows to be automatically included in declarative workflows based on specific constraints on the topology elements.
- Implementation of the improvements will be done by adding some elements to the specification and willnot break compatibility with the current specification.

3776 7.4.1.1 Notes

- The weaving improvement section is a Work in Progress and is not final in 1.1 version. The elements in this section are incomplete and may be subject to change in next specification version.
- Moreover, the weaving improvements is one of the track of improvements. As describe improving the reusability of imperative workflow is another track (that may both co-exists in next specifications).

3783 7.4.2 Weaving improvements

3784 Making declarative better experimental option.

3785 7.4.2.1 Node lifecycle definition

3786Node workflow is defined at the node type level. The node workflow definition is used to generate the
declarative workflow of a given node.

The tosca.nodes.Root type defines workflow steps for both the install workflow (used to instantiate or deploy a topology) and the uninstall workflow (used to destroy or undeploy a topology). The workflow is

```
3790 defined as follows:
```

3791

```
node_types:
  tosca.nodes.Root:
    workflows:
      install:
        steps:
          install_sequence:
            activities:

    set state: creating

              - call_operation: tosca.interfaces.node.lifecycle.Standard.create
              - set state: created
              - set state: configuring
              - call operation:
tosca.interfaces.node.lifecycle.Standard.configure
              - set_state: configured
              - set state: starting
              - call operation: tosca.interfaces.node.lifecycle.Standard.start
              - set state: started
      uninstall:
        steps:
          uninstall sequence:
            activities:
              - set state: stopping
              - call operation: tosca.interfaces.node.lifecycle.Standard.stop
              - set state: stopped

    set state: deleting

              - call operation: tosca.interfaces.node.lifecycle.Standard.delete
              - set state: deleted
```

3792

3793 7.4.2.2 Relationship lifecycle and weaving

While the workflow of a single node is quite simple the TOSCA weaving process is the real key element of declarative workflows. The process of weaving consist of the ability to create complex management workflows including dependency management in execution order between node operations, injection of operations to process specific instruction related to the connection to other nodes based the relationships and groups defined in a topology.

3799

3800 This section describes the relationship weaving and how the description at a template level can be 3801 translated on an instance level. relationship types: tosca.relationships.ConnectsTo: workflow: install: # name of the workflow for wich the weaving has to be taken in account source_weaving: # Instruct how to weave some tasks on the source workflow (executed on SOURCE instance) - after: configuring # instruct that this operation should be weaved after the target reach configuring state wait_target: created # add a join from a state of the target activity: tosca.interfaces.relationships.Configure.pre_configure_source - before: configured # instruct that this operation should be weaved before the target reach configured state activity: tosca.interfaces.relationships.Configure.post_configure_source - before: starting wait_target: started # add a join from a state of the target - after: started activity: tosca.interfaces.relationships.Configure.add target target weaving: # Instruct how to weave some tasks on the target workflow (executed on TARGET instance) - after: configuring # instruct that this operation should be weaved after the target reach configuring state after source: created # add a join from a state of the source activity: tosca.interfaces.relationships.Configure.pre_configure_target - before: configured # instruct that this operation should be weaved before the target reach configured state activity: tosca.interfaces.relationships.Configure.post configure target - after: started activity: tosca.interfaces.relationships.Configure.add source

3803 8 TOSCA networking

Except for the examples, this section is **normative** and describes how to express and control the application centric network semantics available in TOSCA.

3806 8.1 Networking and Service Template Portability

TOSCA Service Templates are application centric in the sense that they focus on describing application components in terms of their requirements and interrelationships. In order to provide cloud portability, it is important that a TOSCA Service Template avoid cloud specific requirements and details. However, at the same time, TOSCA must provide the expressiveness to control the mapping of software component connectivity to the network constructs of the hosting cloud.

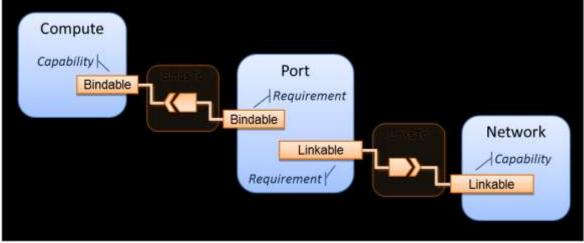
- 3812 TOSCA Networking takes the following approach.
- 38131.The application component connectivity semantics and expressed in terms of Requirements and
Capabilities and the relationships between these. Service Template authors are able to express
the interconnectivity requirements of their software components in an abstract, declarative, and
thus highly portable manner.
- 3817
 2. The information provided in TOSCA is complete enough for a TOSCA implementation to fulfill the
 3818
 application component network requirements declaratively (i.e., it contains information such as
 communication initiation and layer 4 port specifications) so that the required network semantics
 can be realized on arbitrary network infrastructures.
- TOSCA Networking provides full control of the mapping of software component interconnectivity
 to the networking constructs of the hosting cloud network independently of the Service Template,
 providing the required separation between application and network semantics to preserve Service
 Template portability.
- 3825
 4. Service Template authors have the choice of specifying application component networking
 3826
 3826
 3827
 3827
 3828
 3828
 3828
 3828
 3829
 3830
 3830
 3830
 3831
 3831
- 5. Defining a set of network semantics which are expressive enough to address the most common application connectivity requirements while avoiding dependencies on specific network
 technologies and constructs. Service Template authors and cloud providers are able to express unique/non-portable semantics by defining their own specialized network Requirements and Capabilities.

3837 8.2 Connectivity semantics

TOSCA's application centric approach includes the modeling of network connectivity semantics from an
application component connectivity perspective. The basic premise is that applications contain
components which need to communicate with other components using one or more endpoints over a
network stack such as TCP/IP, where connectivity between two components is expressed as a <source
component, source address, source port, target component, target address, target port> tuple. Note that
source and target components are added to the traditional 4 tuple to provide the application centric
information, mapping the network to the source or target component involved in the connectivity.

3846 Software components are expressed as Node Types in TOSCA which can express virtually any kind of 3847 concept in a TOSCA model. Node Types offering network based functions can model their connectivity 3848 using a special Endpoint Capability, tosca.capabilities.Endpoint, designed for this purpose. Node Types

- 3849 which require an Endpoint can specify this as a TOSCA requirement. A special Relationship Type,
- 3850 tosca.relationships.ConnectsTo, is used to implicitly or explicitly relate the source Node Type's endpoint
- 3851 to the required endpoint in the target node type. Since tosca.capabilities.Endpoint and
- 3852 tosca.relationships.ConnectsTo are TOSCA types, they can be used in templates and extended by 3853 subclassing in the usual ways, thus allowing the expression of additional semantics as needed.
- 3854 The following diagram shows how the TOSCA node, capability and relationship types enable modeling
- 3855 the application layer decoupled from the network model intersecting at the Compute node using the 3856
- Bindable capability type.
- 3857 As you can see, the Port node type effectively acts a broker node between the Network node description



3858 and a host Compute node of an application.

8.3 Expressing connectivity semantics 3859

3860 This section describes how TOSCA supports the typical client/server and group communication semantics found in application architectures. 3861

8.3.1 Connection initiation semantics 3862

- 3863 The tosca.relationships.ConnectsTo expresses that requirement that a source application component 3864 needs to be able to communicate with a target software component to consume the services of the target. 3865 ConnectTo is a component interdependency semantic in the most general sense and does not try imply how the communication between the source and target components is physically realized. 3866
- 3867

3868 Application component intercommunication typically has conventions regarding which component(s) 3869 initiate the communication. Connection initiation semantics are specified in tosca.capabilities.Endpoint. 3870 Endpoints at each end of the tosca.relationships.ConnectsTo must indicate identical connection initiation 3871 semantics.

- 3872
- 3873 The following sections describe the normative connection initiation semantics for the 3874 tosca.relationships.ConnectsTo Relationship Type.

3875 8.3.1.1 Source to Target

3876 The Source to Target communication initiation semantic is the most common case where the source component initiates communication with the target component in order to fulfill an instance of the 3877 3878 tosca.relationships.ConnectsTo relationship. The typical case is a "client" component connecting to a 3879 "server" component where the client initiates a stream oriented connection to a pre-defined transport 3880 specific port or set of ports.

- 3882 It is the responsibility of the TOSCA implementation to ensure the source component has a suitable
- 3883 network path to the target component and that the ports specified in the respective
- tosca.capabilities.Endpoint are not blocked. The TOSCA implementation may only represent state of the
 tosca.relationships.ConnectsTo relationship as fulfilled after the actual network communication is enabled
 and the source and target components are in their operational states.
- 3887
- 3888 Note that the connection initiation semantic only impacts the fulfillment of the actual connectivity and does 3889 not impact the node traversal order implied by the tosca.relationships.ConnectsTo Relationship Type.

3890 8.3.1.2 Target to Source

The Target to Source communication initiation semantic is a less common case where the target component initiates communication with the source comment in order to fulfill an instance of the tosca.relationships.ConnectsTo relationship. This "reverse" connection initiation direction is typically required due to some technical requirements of the components or protocols involved, such as the requirement that SSH mush only be initiated from target component in order to fulfill the services required by the source component.

- 3897
- 3898 It is the responsibility of the TOSCA implementation to ensure the source component has a suitable
- 3899 network path to the target component and that the ports specified in the respective
- 3900 tosca.capabilities.Endpoint are not blocked. The TOSCA implementation may only represent state of the 3901 tosca.relationships.ConnectsTo relationship as fulfilled after the actual network communication is enabled
- and the source and target components are in their operational states.
- 3903
- Note that the connection initiation semantic only impacts the fulfillment of the actual connectivity and does not impact the node traversal order implied by the tosca.relationships.ConnectsTo Relationship Type.

3906 8.3.1.3 Peer-to-Peer

- 3907 The Peer-to-Peer communication initiation semantic allows any member of a group to initiate
- 3908 communication with any other member of the same group at any time. This semantic typically appears in 3909 clustering and distributed services where there is redundancy of components or services.
- 3910

3911 It is the responsibility of the TOSCA implementation to ensure the source component has a suitable 3912 network path between all the member component instances and that the ports specified in the respective 3913 tosca.capabilities.Endpoint are not blocked, and the appropriate multicast communication, if necessary,

- enabled. The TOSCA implementation may only represent state of the tosca.relationships.ConnectsTo
- relationship as fulfilled after the actual network communication is enabled such that at least one-member component of the group may reach any other member component of the group.
- 3917
- 3918 Endpoints specifying the Peer-to-Peer initiation semantic need not be related with a
- 3919 tosca.relationships.ConnectsTo relationship for the common case where the same set of component
- instances must communicate with each other.
- 3921
- 3922Note that the connection initiation semantic only impacts the fulfillment of the actual connectivity and does3923not impact the node traversal order implied by the tosca.relationships.ConnectsTo Relationship Type.

3924 8.3.2 Specifying layer 4 ports

TOSCA Service Templates must express enough details about application component
 intercommunication to enable TOSCA implementations to fulfill these communication semantics in the
 network infrastructure. TOSCA currently focuses on TCP/IP as this is the most pervasive in today's cloud

- 3928 infrastructures. The layer 4 ports required for application component intercommunication are specified in
- tosca.capabilities.Endpoint. The union of the port specifications of both the source and target
- tosca.capabilities.Endpoint which are part of the tosca.relationships.ConnectsTo Relationship Template
- are interpreted as the effective set of ports which must be allowed in the network communication.
- 3932
- The meaning of Source and Target port(s) corresponds to the direction of the respective tosca.relationships.ConnectsTo.

3935 8.4 Network provisioning

3936 8.4.1 Declarative network provisioning

TOSCA orchestrators are responsible for the provisioning of the network connectivity for declarative TOSCA Service Templates (Declarative TOSCA Service Templates don't contain explicit plans). This means that the TOSCA orchestrator must be able to infer a suitable logical connectivity model from the Service Template and then decide how to provision the logical connectivity, referred to as "fulfillment", on the available underlying infrastructure. In order to enable fulfillment, sufficient technical details still must be specified, such as the required protocols, ports and QOS information. TOSCA connectivity types, such as tosca.capabilities.Endpoint, provide well defined means to express these details.

3944 8.4.2 Implicit network fulfillment

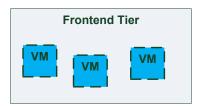
TOSCA Service Templates are by default network agnostic. TOSCA's application centric approach only requires that a TOSCA Service Template contain enough information for a TOSCA orchestrator to infer suitable network connectivity to meet the needs of the application components. Thus Service Template designers are not required to be aware of or provide specific requirements for underlying networks. This approach yields the most portable Service Templates, allowing them to be deployed into any infrastructure which can provide the necessary component interconnectivity.

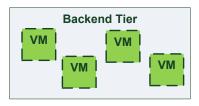
3951 8.4.3 Controlling network fulfillment

- 3952 TOSCA provides mechanisms for providing control over network fulfillment.
- 3953 This mechanism allows the application network designer to express in service template or network 3954 template how the networks should be provisioned.
- 3955

3956 For the use cases described below let's assume we have a typical 3-tier application which is consisting of

FE (frontend), BE (backend) and DB (database) tiers. The simple application topology diagram can be shown below:







	Frontend	Tier
VM	VM	VM

	Backe	nd Tier	
VM	VM	VM	VM

	DB Tier
VM	VM

3959

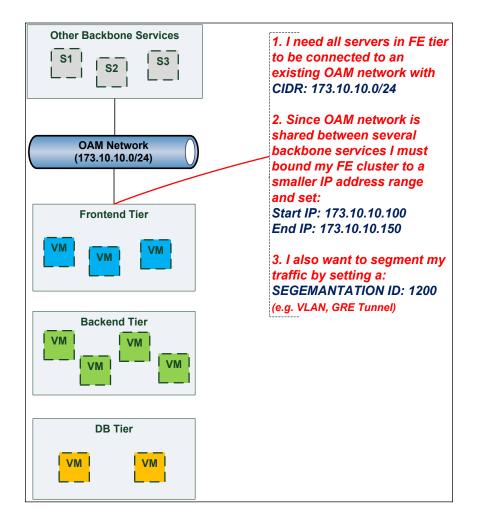
3961

Figure-5: Typical 3-Tier Network

3962 8.4.3.1 Use case: OAM Network

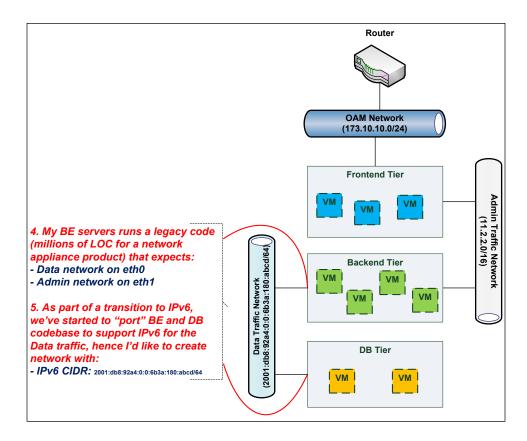
When deploying an application in service provider's on-premise cloud, it's very common that one or more
 of the application's services should be accessible from an ad-hoc OAM (Operations, Administration and
 Management) network which exists in the service provider backbone.

- 3966
- As an application network designer, I'd like to express in my TOSCA network template (which
 corresponds to my TOSCA service template) the network CIDR block, start ip, end ip and segmentation
 ID (e.g. VLAN id).
- The diagram below depicts a typical 3-tiers application with specific networking requirements for its FE
 tier server cluster:



3974 8.4.3.2 Use case: Data Traffic network

The diagram below defines a set of networking requirements for the backend and DB tiers of the 3-tier app mentioned above.



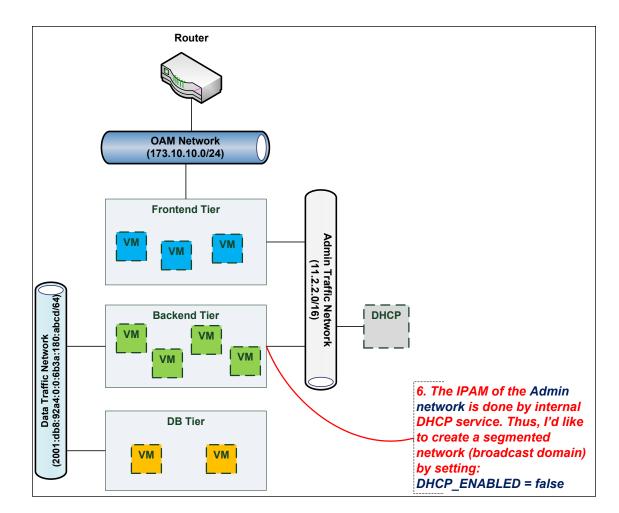
3978 8.4.3.3 Use case: Bring my own DHCP

The same 3-tier app requires for its admin traffic network to manage the IP allocation by its own DHCP which runs autonomously as part of application domain.

3981

3982 For this purpose, the app network designer would like to express in TOSCA that the underlying

3983 provisioned network will be set with DHCP_ENABLED=false. See this illustrated in the figure below:



3985 8.5 Network Types

3986 8.5.1 tosca.nodes.network.Network

3987 The TOSCA **Network** node represents a simple, logical network service.

Shorthand Name	Network			
Type Qualified Name	tosca:Network			
Type URI	tosca.nodes.network.Network			

3988 8.5.1.1 Properties

Name	Required	Туре	Constraints	Description
ip_version	no	integer	valid_values: [4, 6] default: 4	The IP version of the requested network
cidr	no	string	None	The cidr block of the requested network
start_ip	no	string	None	The IP address to be used as the 1 st one in a pool of addresses derived from the cidr block full IP range

Name	Required	Туре	Constraints	Description
end_ip	no	string	None	The IP address to be used as the last one in a pool of addresses derived from the cidr block full IP range
gateway_ip	no	string	None	The gateway IP address.
network_name	no	string	None	 An Identifier that represents an existing Network instance in the underlying cloud infrastructure – OR – be used as the name of the new created network. If network_name is provided along with network_id they will be used to uniquely identify an existing network and not creating a new one, means all other possible properties are not allowed. network_name should be more convenient for using. But in case that network name uniqueness is not guaranteed then one should provide a network_id as well.
network_id	no	string	None	 An Identifier that represents an existing Network instance in the underlying cloud infrastructure. This property is mutually exclusive with all other properties except network_name. Appearance of network_id in network template instructs the Tosca container to use an existing network instead of creating a new one. network_name should be more convenient for using. But in case that network name uniqueness is not guaranteed then one should add a network_id as well. network_name and network_id can be still used together to achieve both uniqueness and convenient.
segmentation_id	no	string	None	A segmentation identifier in the underlying cloud infrastructure (e.g., VLAN id, GRE tunnel id). If the segmentation_id is specified, the network_type or physical_network properties should be provided as well.
network_type	no	string	None	Optionally, specifies the nature of the physical network in the underlying cloud infrastructure. Examples are flat, vlan, gre or vxlan. For flat and vlan types, physical_network should be provided too.
physical_network	no	string	None	Optionally, identifies the physical network on top of which the network is implemented, e.g. physnet1. This property is required if network_type is flat or vlan.
dhcp_enabled	no	boolean	default: true	Indicates the TOSCA container to create a virtual network instance with or without a DHCP service.

8.5.1.2 Attributes

Name	Required	Туре	Constraints	Description
segmentation_i d	no	string	None	The actual <i>segmentation_id</i> that is been assigned to the network by the underlying cloud infrastructure.

3990 8.5.1.3 Definition

```
tosca.nodes.network.Network:
 derived_from: tosca.nodes.Root
 properties:
   ip_version:
      type: integer
      required: false
      default: 4
      constraints:
        - valid_values: [ 4, 6 ]
   cidr:
      type: string
      required: false
    start_ip:
           type: string
      required: false
    end_ip:
           type: string
      required: false
   gateway_ip:
      type: string
      required: false
    network_name:
      type: string
      required: false
    network_id:
      type: string
      required: false
    segmentation id:
      type: string
      required: false
    network_type:
      type: string
      required: false
    physical_network:
      type: string
      required: false
 capabilities:
   link:
      type: tosca.capabilities.network.Linkable
```

3991 8.5.2 tosca.nodes.network.Port

- 3992 The TOSCA **Port** node represents a logical entity that associates between Compute and Network 3993 normative types.
- 3994 The Port node type effectively represents a single virtual NIC on the Compute node instance.

Shorthand Name	Port
Type Qualified Name	tosca:Port
Type URI	tosca.nodes.network.Port

3995 8.5.2.1 Properties

Name	Required	Туре	Constraints	Description
ip_address	no	string	None	Allow the user to set a fixed IP address. Note that this address is a request to the provider which they will attempt to fulfill but may not be able to dependent on the network the port is associated with.
order	no	integer	greater_or_equa I: 0 default: 0	The order of the NIC on the compute instance (e.g. eth2). Note : when binding more than one port to a single compute (aka multi vNICs) and ordering is desired, it is *mandatory* that all ports will be set with an order value and. The <i>order</i> values must represent a positive, arithmetic progression that starts with 0 (e.g. 0, 1, 2,, n).
is_default	no	boolean	default: false	Set is_default =true to apply a default gateway route on the running compute instance to the associated network gateway. Only one port that is associated to single compute node can set as default=true.
ip_range_start	no	string	None	Defines the starting IP of a range to be allocated for the compute instances that are associated by this Port. Without setting this property the IP allocation is done from the entire CIDR block of the network.
ip_range_end	no	string	None	Defines the ending IP of a range to be allocated for the compute instances that are associated by this Port. Without setting this property the IP allocation is done from the entire CIDR block of the network.

3996 8.5.2.2 Attributes

Name	Required	Туре	Constraints	Description
ip_address	no	string	None	The IP address would be assigned to the associated compute instance.

3997 8.5.2.3 Definition

```
tosca.nodes.network.Port:
 derived_from: tosca.nodes.Root
 properties:
    ip address:
      type: string
      required: false
   order:
      type: integer
      required: true
      default: 0
      constraints:
        - greater_or_equal: 0
    is default:
      type: boolean
      required: false
      default: false
   ip_range_start:
      type: string
      required: false
    ip_range_end:
      type: string
      required: false
 requirements:
   - link:
      capability: tosca.capabilities.network.Linkable
      relationship: tosca.relationships.network.LinksTo
   - binding:
      capability: tosca.capabilities.network.Bindable
      relationship: tosca.relationships.network.BindsTo
```

3998 8.5.3 tosca.capabilities.network.Linkable

A node type that includes the Linkable capability indicates that it can be pointed to by a
 tosca.relationships.network.LinksTo relationship type.

Shorthand Name	Linkable
Type Qualified Name	tosca:.Linkable
Type URI	tosca.capabilities.network.Linkable

4001 8.5.3.1 Properties

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	N/A

4002 8.5.3.2 Definition

tosca.capabilities.network.Linkable:

derived_from: tosca.capabilities.Node

4003 8.5.4 tosca.relationships.network.LinksTo

4004 This relationship type represents an association relationship between Port and Network node types.

Shorthand Name	LinksTo		
Type Qualified Name	tosca:LinksTo		
Type URI	tosca.relationships.network.LinksTo		

4005 8.5.4.1 Definition

tosca.relationships.network.LinksTo:

derived_from: tosca.relationships.DependsOn

valid_target_types: [tosca.capabilities.network.Linkable]

4006 8.5.5 tosca.relationships.network.BindsTo

4007 This type represents a network association relationship between Port and Compute node types.

Shorthand Name	network.BindsTo		
Type Qualified Name	tosca:BindsTo		
Type URI	tosca.relationships.network.BindsTo		

4008 8.5.5.1 Definition

tosca.relationships.network.BindsTo:

derived_from: tosca.relationships.DependsOn

valid_target_types: [tosca.capabilities.network.Bindable]

4009 8.6 Network modeling approaches

4010 8.6.1 Option 1: Specifying a network outside the application's Service 4011 Template

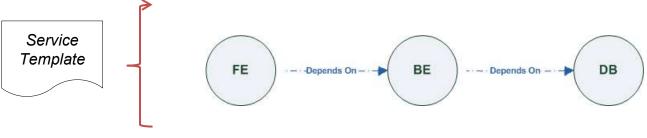
- 4012 This approach allows someone who understands the application's networking requirements, mapping the
- 4013 details of the underlying network to the appropriate node templates in the application.
- 4014

The motivation for this approach is providing the application network designer a fine-grained control on how networks are provisioned and stitched to its application by the TOSCA orchestrator and underlying cloud infrastructure while still preserving the portability of his service template. Preserving the portability means here not doing any modification in service template but just "plug-in" the desired network modeling. The network modeling can reside in the same service template file but the best practice should

- 4020 be placing it in a separated self-contained network template file.
- 4021
- 4022 This "pluggable" network template approach introduces a new normative node type called Port, capability 4023 called *tosca.capabilities.network.Linkable* and relationship type called
- 4024 tosca.relationships.network.LinksTo.
- 4025 The idea of the Port is to elegantly associate the desired compute nodes with the desired network nodes 4026 while not "touching" the compute itself.

4027

- 4028 The following diagram series demonstrate the plug-ability strength of this approach.
- 4029 Let's assume an application designer has modeled a service template as shown in Figure 1 that
- 4030 describes the application topology nodes (compute, storage, software components, etc.) with their
- 4031 relationships. The designer ideally wants to preserve this service template and use it in any cloud
- 4032 provider environment without any change.



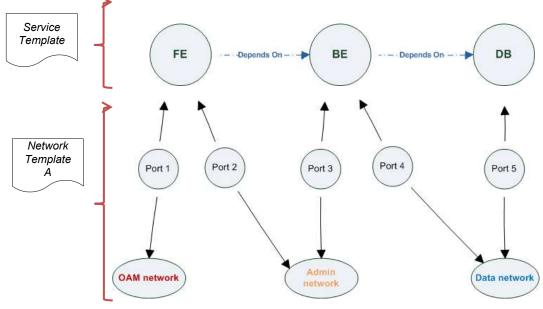
4033 4034

Figure-6: Generic Service Template

4035 When the application designer comes to consider its application networking requirement they typically call 4036 the network architect/designer from their company (who has the correct expertise).

4037The network designer, after understanding the application connectivity requirements and optionally the4038target cloud provider environment, is able to model the network template and plug it to the service

4039 template as shown in Figure 2:

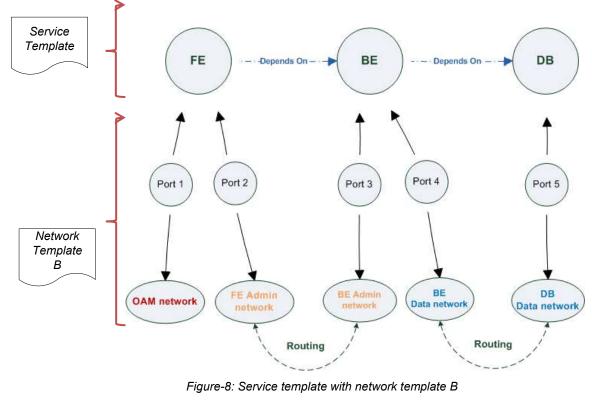


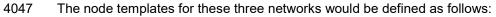


4045 4046

Figure-7: Service template with network template A

4042 When there's a new target cloud environment to run the application on, the network designer is simply 4043 creates a new network template B that corresponds to the new environmental conditions and provide it to 4044 the application designer which packs it into the application CSAR.





node_templates:
 frontend:

17 January 2019 Page 254 of 354

```
type: tosca.nodes.Compute
 properties: # omitted for brevity
backend:
 type: tosca.nodes.Compute
 properties: # omitted for brevity
database:
 type: tosca.nodes.Compute
 properties: # omitted for brevity
oam network:
 type: tosca.nodes.network.Network
 properties: # omitted for brevity
admin network:
 type: tosca.nodes.network.Network
 properties: # omitted for brevity
data_network:
  type: tosca.nodes.network.Network
 properties: # omitted for brevity
# ports definition
fe_oam_net_port:
 type: tosca.nodes.network.Port
 properties:
    is_default: true
    ip_range_start: { get_input: fe_oam_net_ip_range_start }
    ip_range_end: { get_input: fe_oam_net_ip_range_end }
 requirements:
    - link: oam network
    - binding: frontend
fe_admin_net_port:
  type: tosca.nodes.network.Port
  requirements:
    - link: admin network
    - binding: frontend
```

be_admin_net_port:

```
type: tosca.nodes.network.Port
 properties:
     order: 0
 requirements:
    - link: admin_network
    - binding: backend
be_data_net_port:
 type: tosca.nodes.network.Port
 properties:
     order: 1
 requirements:
    - link: data_network
    - binding: backend
db data net port:
 type: tosca.nodes.network.Port
  requirements:
    - link: data network
    - binding: database
```

4048 8.6.2 Option 2: Specifying network requirements within the application's 4049 Service Template

4050 This approach allows the Service Template designer to map an endpoint to a logical network.

The use case shown below examines a way to express in the TOSCA YAML service template a typical 3tier application with their required networking modeling:

```
node_templates:
frontend:
  type: tosca.nodes.Compute
  properties: # omitted for brevity
  requirements:
        - network_oam: oam_network
        - network_admin: oamin_network
    backend:
    type: tosca.nodes.Compute
    properties: # omitted for brevity
    requirements:
        notwork_admin: admin_network
```

- network_admin: admin_network
- network_data: data_network

```
database:
 type: tosca.nodes.Compute
 properties: # omitted for brevity
 requirements:
    - network_data: data_network
oam_network:
 type: tosca.nodes.network.Network
 properties:
    ip_version: { get_input: oam_network_ip_version }
    cidr: { get_input: oam_network_cidr }
    start_ip: { get_input: oam_network_start_ip }
    end_ip: { get_input: oam_network_end_ip }
admin_network:
 type: tosca.nodes.network.Network
 properties:
    ip version: { get input: admin network ip version }
    dhcp_enabled: { get_input: admin_network_dhcp_enabled }
data_network:
 type: tosca.nodes.network.Network
 properties:
    ip_version: { get_input: data_network_ip_version }
    cidr: { get_input: data_network_cidr }
```

9 Non-normative type definitions

This section defines **non-normative** types which are used only in examples and use cases in this
 specification and are included only for completeness for the reader. Implementations of this specification
 are not required to support these types for conformance.

4058 9.1 Artifact Types

4059 This section contains are non-normative Artifact Types used in use cases and examples.

4060 9.1.1 tosca.artifacts.Deployment.Image.Container.Docker

4061 This artifact represents a Docker "image" (a TOSCA deployment artifact type) which is a binary comprised 4062 of one or more (a union of read-only and read-write) layers created from snapshots within the underlying 4063 Docker **Union File System.**

4064 9.1.1.1 Definition

tosca.artifacts.Deployment.Image.Container.Docker: derived_from: tosca.artifacts.Deployment.Image description: Docker Container Image

4065 9.1.2 tosca.artifacts.Deployment.Image.VM.ISO

4066 A Virtual Machine (VM) formatted as an ISO standard disk image.

4067 9.1.2.1 Definition

tosca.artifacts.Deployment.Image.VM.ISO: derived_from: tosca.artifacts.Deployment.Image.VM description: Virtual Machine (VM) image in ISO disk format mime_type: application/octet-stream file_ext: [iso]

4068 9.1.3 tosca.artifacts.Deployment.Image.VM.QCOW2

4069 A Virtual Machine (VM) formatted as a QEMU emulator version 2 standard disk image.

4070 9.1.3.1 Definition

tosca.artifacts.Deployment.Image.VM.QCOW2: derived_from: tosca.artifacts.Deployment.Image.VM description: Virtual Machine (VM) image in QCOW v2 standard disk format mime_type: application/octet-stream file_ext: [qcow2]

4071 **9.2 Capability Types**

4072 This section contains are non-normative Capability Types used in use cases and examples.

4073 9.2.1 tosca.capabilities.Container.Docker

Shorthand Name	Container.Docker
Type Qualified Name	tosca:Container.Docker
Type URI	tosca.capabilities.Container.Docker

4074 The type indicates capabilities of a Docker runtime environment (client).

4075 9.2.1.1 Properties

Name	Required	Туре	Constraints	Description
version	no	version[]	None	The Docker version capability (i.e., the versions supported by the capability).
publish_all	no	boolean	default: false	Indicates that all ports (ranges) listed in the <i>dockerfile</i> using the EXPOSE keyword be published.
publish_ports	no	list of PortSpec	None	List of ports mappings from source (Docker container) to target (host) ports to publish.
expose_ports	no	list of PortSpec	None	List of ports mappings from source (Docker container) to expose to other Docker containers (not accessible outside host).
volumes	no	list of string	None	The <i>dockerfile</i> VOLUME command which is used to enable access from the Docker container to a directory on the host machine.
host_id	no	string	None	The optional identifier of an existing host resource that should be used to run this container on.
volume_id	no	string	None	The optional identifier of an existing storage volume (resource) that should be used to create the container's mount point(s) on.

4076 9.2.1.2 Definition

```
tosca.capabilities.Container.Docker:
    derived_from: tosca.capabilities.Container
    properties:
        version:
        type: list
        required: false
        entry_schema: version
    publish_all:
        type: boolean
        default: false
        required: false
        publish_ports:
        type: list
        entry_schema: PortSpec
```

```
required: false
expose_ports:
  type: list
  entry_schema: PortSpec
  required: false
volumes:
  type: list
  entry_schema: string
  required: false
```

4077 9.2.1.3 Notes

When the expose_ports property is used, only the source and source_range properties of
 PortSpec would be valid for supplying port numbers or ranges, the target and target_range
 properties would be ignored.

4081 **9.3 Node Types**

4082 This section contains non-normative node types referenced in use cases and examples. All additional
4083 Attributes, Properties, Requirements and Capabilities shown in their definitions (and are not inherited
4084 from ancestor normative types) are also considered to be non-normative.

4085 9.3.1 tosca.nodes.Database.MySQL

4086 9.3.1.1 Properties

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	N/A

4087 9.3.1.2 Definition

tosca.nodes.Database.MySQL:

derived_from: tosca.nodes.Database

```
requirements:
```

- host:

node: tosca.nodes.DBMS.MySQL

4088 9.3.2 tosca.nodes.DBMS.MySQL

4089 9.3.2.1 Properties

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	N/A

4090 9.3.2.2 Definition

tosca.nodes.DBMS.MySQL:

```
derived_from: tosca.nodes.DBMS
properties:
    port:
        description: reflect the default MySQL server port
        default: 3306
    root_password:
        # MySQL requires a root_password for configuration
        # Override parent DBMS definition to make this property required
        required: true
capabilities:
    # Further constrain the 'host' capability to only allow MySQL databases
    host:
        valid_source_types: [ tosca.nodes.Database.MySQL ]
```

4091 9.3.3 tosca.nodes.WebServer.Apache

4092 **9.3.3.1 Properties**

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	N/A

4093 9.3.3.2 Definition

tosca.nodes.WebServer.Apache: derived_from: tosca.nodes.WebServer

4094 9.3.4 tosca.nodes.WebApplication.WordPress

4095 This section defines a non-normative Node type for the WordPress [WordPress] application.

4096 9.3.4.1 Properties

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	N/A

4097 9.3.4.2 Definition

```
tosca.nodes.WebApplication.WordPress:
  derived_from: tosca.nodes.WebApplication
  properties:
    admin_user:
    type: string
    admin_password:
    type: string
    db_host:
```

```
type: string
```

```
requirements:
```

- database_endpoint:

```
capability: tosca.capabilities.Endpoint.Database
node: tosca.nodes.Database
relationship: tosca.relationships.ConnectsTo
```

4098 9.3.5 tosca.nodes.WebServer.Nodejs

4099 This non-normative node type represents a Node.js [NodeJS] web application server.

4100 9.3.5.1 Properties

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	N/A

4101 9.3.5.2 Definition

```
tosca.nodes.WebServer.Nodejs:
    derived_from: tosca.nodes.WebServer
    properties:
        # Property to supply the desired implementation in the Github repository
        github_url:
            required: no
            type: string
            description: location of the application on the github.
            default: https://github.com/mmm/testnode.git
        interfaces:
        Standard:
        inputs:
            github_url:
            type: string
```

4102 9.3.6 tosca.nodes.Container.Application.Docker

4103 **9.3.6.1 Properties**

Name	Required	Туре	Constraints	Description
N/A	N/A	N/A	N/A	N/A

4104 9.3.6.2 Definition

tosca.nodes.Container.Application.Docker:

derived_from: tosca.nodes.Containertosca.nodes.Container.Application

requirements:

- host: capability: tosca.capabilities.Container.Docker

4105 **10 Component Modeling Use Cases**

This section is **non-normative** and includes use cases that explore how to model components and their relationships using TOSCA Simple Profile in YAML.

4108 10.1.1 Use Case: Exploring the HostedOn relationship using 4109 WebApplication and WebServer

4110 This use case examines the ways TOSCA YAML can be used to express a simple hosting relationship 4111 (i.e., **HostedOn**) using the normative TOSCA **WebServer** and **WebApplication** node types defined in this

4112 specification.

4113 **10.1.1.1 WebServer declares its "host" capability**

4114 For convenience, relevant parts of the normative TOSCA Node Type for **WebServer** are shown below:

```
tosca.nodes.WebServer
  derived_from: SoftwareComponent
  capabilities:
    ...
    host:
    type: tosca.capabilities.Container
    valid_source_types: [ tosca.nodes.WebApplication ]
```

As can be seen, the WebServer Node Type declares its capability to "contain" (i.e., host) other nodes
using the symbolic name "host" and providing the Capability Type tosca.capabilities.Container. It
should be noted that the symbolic name of "host" is not a reserved word, but one assigned by the type
designer that implies at or betokens the associated capability. The Container capability definition also

- 4119 includes a required list of valid Node Types that can be contained by this, the **WebServer**, Node Type.
- 4120 This list is declared using the keyname of **valid_source_types** and in this case it includes only allowed
- 4121 type WebApplication.

4122 10.1.1.2 WebApplication declares its "host" requirement

The **WebApplication** node type needs to be able to describe the type of capability a target node would have to provide in order to "host" it. The normative TOSCA capability type tosca.capabilities.Container is used to describe all normative TOSCA hosting (i.e., container-containee pattern) relationships. As can be seen below, the WebApplication accomplishes this by declaring a requirement with the symbolic name **host**" with the **capability** keyname set to tosca.capabilities.Container.

4128 Again, for convenience, the relevant parts of the normative WebApplication Node Type are shown below:

```
tosca.nodes.WebApplication:
  derived_from: tosca.nodes.Root
  requirements:
        - host:
        capability: tosca.capabilities.Container
        node: tosca.nodes.WebServer
        relationship: tosca.relationships.HostedOn
```

4129 10.1.1.2.1 Notes

The symbolic name "host" is not a keyword and was selected for consistent use in TOSCA
 normative node types to give the reader an indication of the type of requirement being
 referenced. A valid HostedOn relationship could still be established between WebApplicaton and
 WebServer in a TOSCA Service Template regardless of the symbolic name assigned to either the
 requirement or capability declaration.

4135 **10.1.2 Use Case: Establishing a ConnectsTo relationship to WebServer**

- 4136 This use case examines the ways TOSCA YAML can be used to express a simple connection
- relationship (i.e., ConnectsTo) between some service derived from the SoftwareComponent Node Type,
 to the normative WebServer node type defined in this specification.
- 4139 The service template that would establish a ConnectsTo relationship as follows:

```
node_types:
  MyServiceType:
    derived_from: SoftwareComponent
    requirements:
      # This type of service requires a connection to a WebServer's data_endpoint
      - connection1:
          node: WebServer
          relationship: ConnectsTo
          capability: Endpoint
topology template:
  node templates:
    my web service:
      type: MyServiceType
      . . .
      requirements:
        - connection1:
            node: my web server
    my web server:
      # Note, the normative WebServer node type declares the "data_endpoint"
      # capability of type tosca.capabilities.Endpoint.
      type: WebServer
```

4140 Since the normative WebServer Node Type only declares one capability of type

4141 tosca.capabilties.Endpoint (or Endpoint, its shortname alias in TOSCA) using the symbolic name
4142 data_endpoint, the my_web_service node template does not need to declare that symbolic name on its
4143 requirement declaration. If however, the my_web_server node was based upon some other node type
4144 that declared more than one capability of type Endpoint, then the capability keyname could be used

to supply the desired symbolic name if necessary.

4146 **10.1.2.1 Best practice**

It should be noted that the best practice for designing Node Types in TOSCA should not export two
 capabilities of the same type if they truly offer different functionality (i.e., different capabilities) which

4149 should be distinguished using different Capability Type definitions.

4150 10.1.3 Use Case: Attaching (local) BlockStorage to a Compute node

4151 This use case examines the ways TOSCA YAML can be used to express a simple AttachesTo 4152 relationship between a Compute node and a locally attached BlockStorage node.

4153 The service template that would establish an AttachesTo relationship follows:

```
node templates:
  my server:
    type: Compute
    . . .
    requirements:
      # contextually this can only be a relationship type
      - local storage:
          # capability is provided by Compute Node Type
          node: my block storage
          relationship:
            type: AttachesTo
            properties:
              location: /path1/path2
          # This maps the local requirement name 'local_storage' to the
          # target node's capability name 'attachment'
 my block storage:
    type: BlockStorage
    properties:
      size: 10 GB
```

4154 10.1.4 Use Case: Reusing a BlockStorage Relationship using Relationship 4155 Type or Relationship Template

This builds upon the previous use case (10.1.3) to examine how a template author could attach multiple
Compute nodes (templates) to the same BlockStorage node (template), but with slightly different property
values for the AttachesTo relationship.

- 4159
- 4160 Specifically, several notation options are shown (in this use case) that achieve the same desired result.

4161 10.1.4.1 Simple Profile Rationale

Referencing an explicitly declared Relationship Template is a convenience of the Simple Profile that
allows template authors an entity to set, constrain or override the properties and operations as defined in
its declared (Relationship) Type much as allowed now for Node Templates. It is especially useful when a
complex Relationship Type (with many configurable properties or operations) has several logical

4166 occurrences in the same Service (Topology) Template; allowing the author to avoid configuring these4167 same properties and operations in multiple Node Templates.

4168 **10.1.4.2 Notation Style #1: Augment AttachesTo Relationship Type directly in** 4169 **each Node Template**

- 4170 This notation extends the methodology used for establishing a HostedOn relationship, but allowing
- 4171 template author to supply (dynamic) configuration and/or override of properties and operations.
- 4172
- 4173 **Note:** This option will remain valid for Simple Profile regardless of other notation (copy or aliasing) options
- 4174 being discussed or adopted for future versions.
- 4175

```
node templates:
 my_block_storage:
    type: BlockStorage
   properties:
      size: 10
 my_web_app_tier_1:
    type: Compute
    requirements:
      - local storage:
          node: my block storage
          relationship: MyAttachesTo
            # use default property settings in the Relationship Type definition
 my web app tier 2:
    type: Compute
    requirements:
      - local storage:
          node: my_block_storage
          relationship:
            type: MyAttachesTo
            # Override default property setting for just the 'location' property
            properties:
              location: /some_other_data_location
relationship_types:
 MyAttachesTo:
    derived from: AttachesTo
    properties:
```

```
location: /default_location
interfaces:
   Configure:
    post_configure_target:
    implementation: default_script.sh
```

4176

4177 10.1.4.3 Notation Style #2: Use the 'template' keyword on the Node Templates to 4178 specify which named Relationship Template to use

This option shows how to explicitly declare different named Relationship Templates within the Service
Template as part of a relationship_templates section (which have different property values) and can
be referenced by different Compute typed Node Templates.

```
node templates:
  my block storage:
    type: BlockStorage
    properties:
      size: 10
  my web app tier 1:
    derived from: Compute
    requirements:
      - local_storage:
          node: my_block_storage
          relationship: storage_attachesto_1
  my_web_app_tier_2:
    derived_from: Compute
    requirements:
      - local_storage:
          node: my_block_storage
          relationship: storage_attachesto_2
relationship templates:
  storage attachesto 1:
    type: MyAttachesTo
    properties:
      location: /my_data_location
  storage attachesto 2:
```

```
type: MyAttachesTo
properties:
    location: /some_other_data_location
```

relationship_types:

```
MyAttachesTo:
    derived_from: AttachesTo
    interfaces:
        some_interface_name:
            some_operation:
            implementation: default script.sh
```

4183

4184 10.1.4.4 Notation Style #3: Using the "copy" keyname to define a similar 4185 Relationship Template

How does TOSCA make it easier to create a new relationship template that is mostly the same as one
that exists without manually copying all the same information? TOSCA provides the copy keyname as a
convenient way to copy an existing template definition into a new template definition as a starting point or
basis for describing a new definition and avoid manual copy. The end results are cleaner TOSCA Service
Templates that allows the description of only the changes (or deltas) between similar templates.

4191 The example below shows that the Relationship Template named **storage_attachesto_1** provides 4192 some overrides (conceptually a large set of overrides) on its Type which the Relationship Template

4193 named **storage_attachesto 2** wants to "**copy**" before perhaps providing a smaller number of overrides.

```
node_templates:

my_block_storage:

type: BlockStorage

properties:

size: 10

my_web_app_tier_1:

derived_from: Compute

requirements:

- attachment:

node: my_block_storage

relationship: storage_attachesto_1

my_web_app_tier_2:

derived_from: Compute

requirements:

- attachment:

- attachment:
```

```
node: my_block_storage
          relationship: storage_attachesto_2
relationship_templates:
  storage_attachesto_1:
    type: MyAttachesTo
    properties:
      location: /my_data_location
    interfaces:
      some interface name:
        some_operation_name_1: my_script_1.sh
        some_operation_name_2: my_script_2.sh
        some_operation_name_3: my_script_3.sh
  storage_attachesto_2:
    # Copy the contents of the "storage_attachesto_1" template into this new one
    copy: storage attachesto 1
    # Then change just the value of the location property
    properties:
      location: /some_other_data_location
relationship_types:
 MyAttachesTo:
    derived_from: AttachesTo
    interfaces:
      some_interface_name:
        some_operation:
          implementation: default_script.sh
```

4194 **11 Application Modeling Use Cases**

4195 This section is **non-normative** and includes use cases that show how to model Infrastructure-as-a-

4196 Service (IaaS), Platform-as-a-Service (PaaS) and complete application uses cases using TOSCA Simple 4197 Profile in YAML.

4198 **11.1 Use cases**

- 4199 Many of the use cases listed below can by found under the following link:
- 4200 https://github.com/openstack/heat-translator/tree/master/translator/tests/data

4201 **11.1.1 Overview**

Name	Description
Compute : Create a single Compute instance with a host Operating System	Introduces a TOSCA Compute node type which is used to stand up a single compute instance with a host Operating System Virtual Machine (VM) image selected by the platform provider using the Compute node's properties.
Software Component 1: Automatic deployment of a Virtual Machine (VM) image artifact	Introduces the SoftwareComponent node type which declares software that is hosted on a Compute instance. In this case, the SoftwareComponent declares a VM image as a deployment artifact which includes its own pre-packaged operating system and software. The TOSCA Orchestrator detects this known deployment artifact type on the SoftwareComponent node template and automatically deploys it to the Compute node.
BlockStorage-1: Attaching Block Storage to a single Compute instance	Demonstrates how to attach a TOSCA BlockStorage node to a Compute node using the normative AttachesTo relationship.
BlockStorage-2: Attaching Block Storage using a custom Relationship Type	Demonstrates how to attach a TOSCA BlockStorage node to a Compute node using a custom RelationshipType that derives from the normative AttachesTo relationship.
BlockStorage-3: Using a Relationship Template of type AttachesTo	Demonstrates how to attach a TOSCA BlockStorage node to a Compute node using a TOSCA Relationship Template that is based upon the normative AttachesTo Relationship Type.
BlockStorage-4: Single Block Storage shared by 2-Tier Application with custom AttachesTo Type and implied relationships	This use case shows 2 Compute instances (2 tiers) with one BlockStorage node, and also uses a custom AttachesTo Relationship that provides a default mount point (i.e., location) which the 1 st tier uses, but the 2 nd tier provides a different mount point.
BlockStorage-5: Single Block Storage shared by 2-Tier Application with custom AttachesTo Type and explicit Relationship Templates	This use case is like the previous BlockStorage-4 use case, but also creates two relationship templates (one for each tier) each of which provide a different mount point (i.e., location) which overrides the default location defined in the custom Relationship Type.
BlockStorage-6: Multiple Block Storage attached to different Servers	This use case demonstrates how two different TOSCA BlockStorage nodes can be attached to two different Compute nodes (i.e., servers) each using the normative AttachesTo relationship.
Object Storage 1 : Creating an Object Storage service	Introduces the TOSCA ObjectStorage node type and shows how it can be instantiated.
Network-1: Server bound to a new network	Introduces the TOSCA Network and Port nodes used for modeling logical networks using the LinksTo and BindsTo Relationship Types. In this use case, the template is invoked without an existing network_name as an input property so a new network is created using the properties declared in the Network node.

Network-2: Server bound to an existing network	Shows how to use a network_name as an input parameter to the template to allow a server to be associated with (i.e. bound to) an existing Network .
Network-3: Two servers bound to a single network	This use case shows how two servers (Compute nodes) can be associated with the same Network node using two logical network Ports .
Network-4: Server bound to three networks	This use case shows how three logical networks (Network nodes), each with its own IP address range, can be associated with the same server (Compute node).
WebServer-DBMS-1: WordPress [WordPress] + MySQL, single instance	Shows how to host a TOSCA WebServer with a TOSCA WebApplication, DBMS and Database Node Types along with their dependent HostedOn and ConnectsTo relationships.
WebServer-DBMS-2: Nodejs with PayPal Sample App and MongoDB on separate instances	Instantiates a 2-tier application with Nodejs and its (PayPal sample) WebApplication on one tier which connects a MongoDB database (which stores its application data) using a ConnectsTo relationship.
Multi-Tier-1: Elasticsearch, Logstash, Kibana (ELK)	Shows Elasticsearch , Logstash and Kibana (ELK) being used in a typical manner to collect, search and monitor/visualize data from a running application.
	This use case builds upon the previous Nodejs/MongoDB 2-tier application as the one being monitored. The collectd and rsyslog components are added to both the WebServer and Database tiers which work to collect data for Logstash.
	In addition to the application tiers, a 3 rd tier is introduced with Logstash to collect data from the application tiers. Finally a 4 th tier is added to search the Logstash data with Elasticsearch and visualize it using Kibana .
	<u>Note</u> : This use case also shows the convenience of using a single YAML macro (declared in the dsl_definitions section of the TOSCA Service Template) on multiple Compute nodes.
Container-1: Containers	Minimalist TOSCA Service Template description of 2 Docker containers linked to each other.
using Docker single	Specifically, one container runs wordpress and connects to second mysql database container
Compute instance (Containers only)	both on a single server (i.e., Compute instance). The use case also demonstrates how TOSCA declares and references Docker images from the Docker Hub repository.
	<u>Variation 1</u> : Docker Container nodes (only) providing their Docker Requirements allowing platform (orchestrator) to select/provide the underlying Docker implementation (Capability).

4202 11.1.2 Compute: Create a single Compute instance with a host Operating 4203 System

4204 **11.1.2.1 Description**

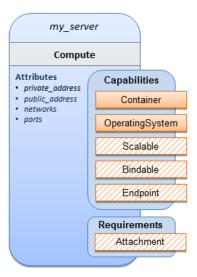
This use case demonstrates how the TOSCA Simple Profile specification can be used to stand up a single Compute instance with a guest Operating System using a normative TOSCA **Compute** node. The TOSCA Compute node is declarative in that the service template describes both the processor and host operating system platform characteristics (i.e., properties declared on the capability named "os" sometimes called a "flavor") that are desired by the template author. The cloud provider would attempt to fulfill these properties (to the best of its abilities) during orchestration.

4211 **11.1.2.2 Features**

- 4212 This use case introduces the following TOSCA Simple Profile features:
- 4213 A node template that uses the normative TOSCA Compute Node Type along with showing an exemplary set of its properties being configured.

4215 Use of the TOSCA Service Template **inputs** section to declare a configurable value the template 4216 user may supply at runtime. In this case, the "host" property named "num cpus" (of type integer) 4217 is declared. 4218 0 Use of a property constraint to limit the allowed integer values for the "num cpus" property to a specific list supplied in the property declaration. 4219 4220 Use of the TOSCA Service Template **outputs** section to declare a value the template user may • request at runtime. In this case, the property named "instance_ip" is declared 4221 4222 The "instance_ip" output property is programmatically retrieved from the Compute 0 4223 node's "public_address" attribute using the TOSCA Service Template-level 4224 get attribute function.

4225 11.1.2.3 Logical Diagram



4226

4227 11.1.2.4 Sample YAML

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: >
  TOSCA simple profile that just defines a single compute instance and selects a
  (guest) host Operating System from the Compute node's properties. Note, this
  example does not include default values on inputs properties.
  topology_template:
    inputs:
      cpus:
      type: integer
      description: Number of CPUs for the server.
      constraints:
           - valid_values: [ 1, 2, 4, 8 ]
      node templates:
```

```
my server:
    type: Compute
    capabilities:
      host:
        properties:
          disk_size: 10 GB
          num_cpus: { get_input: cpus }
          mem size: 1 GB
      os:
        properties:
          architecture: x86 64
          type: Linux
          distribution: ubuntu
          version: 12.04
outputs:
 private ip:
    description: The private IP address of the deployed server instance.
    value: { get attribute: [my server, private address] }
```

4228 11.1.2.5 Notes

• This use case uses a versioned, Linux Ubuntu distribution on the Compute node.

4230 11.1.3 Software Component 1: Automatic deployment of a Virtual Machine 4231 (VM) image artifact

4232 **11.1.3.1 Description**

This use case demonstrates how the TOSCA SoftwareComponent node type can be used to declare
software that is packaged in a standard Virtual Machine (VM) image file format (i.e., in this case QCOW2)
and is hosted on a TOSCA Compute node (instance). In this variation, the SoftwareComponent declares
a VM image as a deployment artifact that includes its own pre-packaged operating system and software.
The TOSCA Orchestrator detects this known deployment artifact type on the SoftwareComponent node
template and automatically deploys it to the Compute node.

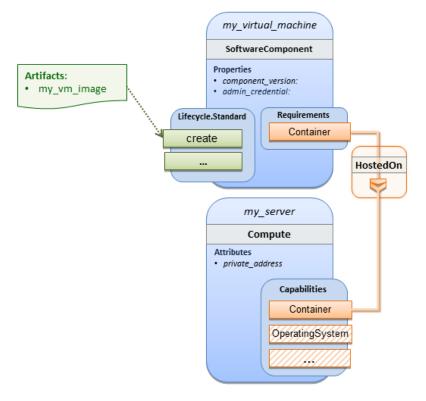
4239 11.1.3.2 Features

- 4240 This use case introduces the following TOSCA Simple Profile features:
- A node template that uses the normative TOSCA SoftwareComponent Node Type along with
 showing an exemplary set of its properties being configured.
- Use of the TOSCA Service Template artifacts section to declare a Virtual Machine (VM) image
 artifact type which is referenced by the SoftwareComponent node template.
- The VM file format, in this case QCOW2, includes its own guest Operating System (OS) and
 therefore does <u>not</u> "require" a TOSCA **OperatingSystem** capability from the TOSCA Compute
 node.

4248 **11.1.3.3 Assumptions**

- 4249 This use case assumes the following:
- That the TOSCA Orchestrator (working with the Cloud provider's underlying management services) is able to instantiate a Compute node that has a hypervisor that supports the Virtual Machine (VM) image format, in this case QCOW2, which should be compatible with many standard hypervisors such as XEN and KVM.
- This is not a "bare metal" use case and assumes the existence of a hypervisor on the machine
 that is allocated to "host" the Compute instance supports (e.g. has drivers, etc.) the VM image
 format in this example.

4257 11.1.3.4 Logical Diagram



4258

4259 11.1.3.5 Sample YAML

tosca_definitions_version: tosca_simple_yaml_1_0

description: >

TOSCA Simple Profile with a SoftwareComponent node with a declared Virtual machine (VM) deployment artifact that automatically deploys to its host Compute node.

topology_template:

node_templates:

my_virtual_machine:

17 January 2019 Page 275 of 354

```
type: SoftwareComponent
    artifacts:
      my_vm_image:
        file: images/fedora-18-x86 64.qcow2
        type: tosca.artifacts.Deployment.Image.VM.QCOW2
    requirements:
      - host: my_server
    # Automatically deploy the VM image referenced on the create operation
    interfaces:
      Standard:
        create: my_vm_image
 # Compute instance with no Operating System guest host
 my server:
    type: Compute
    capabilities:
      # Note: no guest OperatingSystem requirements as these are in the image.
      host:
        properties:
          disk_size: 10 GB
          num_cpus: { get_input: cpus }
          mem size: 4 GB
outputs:
 private ip:
    description: The private IP address of the deployed server instance.
    value: { get_attribute: [my_server, private_address] }
```

4260 11.1.3.6 Notes

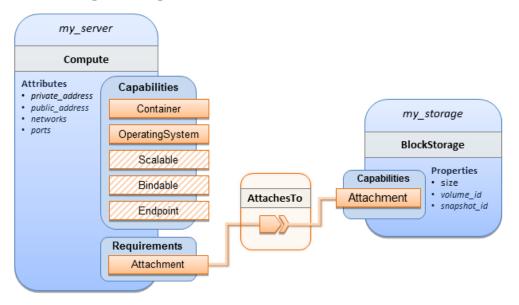
4261	•	The use of the type keyname on the artifact definition (within the my_virtual_machine node
4262		template) to declare the ISO image deployment artifact type (i.e.,
4263		tosca.artifacts.Deployment.Image.VM.ISO) is redundant since the file extension is ".iso"
4264		which associated with this known, declared artifact type.
4265	•	This use case references a filename on the my_vm_image artifact, which indicates a Linux,
4266		Fedora 18, x86 VM image, only as one possible example.

4267 11.1.4 Block Storage 1: Using the normative AttachesTo Relationship Type

4268 **11.1.4.1 Description**

4269 This use case demonstrates how to attach a TOSCA **BlockStorage** node to a **Compute** node using the 4270 normative **AttachesTo** relationship.

4271 11.1.4.2 Logical Diagram



4272

4273 11.1.4.3 Sample YAML

tosca_definitions_version: tosca_simple_yaml_1_0

```
description: >
```

TOSCA simple profile with server and attached block storage using the normative AttachesTo Relationship Type.

```
topology_template:
  inputs:
    cpus:
      type: integer
      description: Number of CPUs for the server.
      constraints:
        - valid_values: [ 1, 2, 4, 8 ]
    storage_size:
      type: scalar-unit.size
      description: Size of the storage to be created.
      default: 1 GB
    storage snapshot id:
      type: string
      description: >
        Optional identifier for an existing snapshot to use when creating
storage.
    storage_location:
```

```
type: string
    description: Block storage mount point (filesystem path).
node_templates:
 my_server:
    type: Compute
    capabilities:
      host:
        properties:
          disk_size: 10 GB
          num_cpus: { get_input: cpus }
          mem size: 1 GB
      os:
        properties:
          architecture: x86_64
          type: linux
          distribution: fedora
          version: 18.0
    requirements:
      - local_storage:
          node: my_storage
          relationship:
            type: AttachesTo
            properties:
              location: { get_input: storage_location }
 my_storage:
   type: BlockStorage
    properties:
      size: { get_input: storage_size }
      snapshot_id: { get_input: storage_snapshot_id }
outputs:
 private_ip:
    description: The private IP address of the newly created compute instance.
    value: { get_attribute: [my_server, private_address] }
 volume id:
    description: The volume id of the block storage instance.
    value: { get_attribute: [my_storage, volume_id] }
```

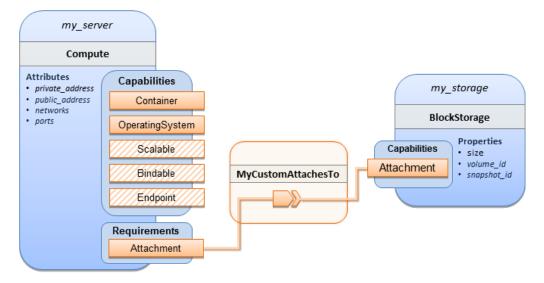
4274 11.1.5 Block Storage 2: Using a custom AttachesTo Relationship Type

4275 **11.1.5.1 Description**

4276 This use case demonstrates how to attach a TOSCA **BlockStorage** node to a **Compute** node using a

4277 custom RelationshipType that derives from the normative AttachesTo relationship.

4278 11.1.5.2 Logical Diagram



4279

4280 11.1.5.3 Sample YAML

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: >
  TOSCA simple profile with server and attached block storage using a custom
AttachesTo Relationship Type.
relationship_types:
  MyCustomAttachesTo:
    derived_from: AttachesTo
topology_template:
    inputs:
    cpus:
      type: integer
      description: Number of CPUs for the server.
      constraints:
        - valid_values: [ 1, 2, 4, 8 ]
    storage_size:
```

```
type: scalar-unit.size
      description: Size of the storage to be created.
      default: 1 GB
    storage_snapshot_id:
     type: string
      description: >
        Optional identifier for an existing snapshot to use when creating
storage.
    storage_location:
      type: string
      description: Block storage mount point (filesystem path).
  node templates:
   my server:
      type: Compute
      capabilities:
        host:
          properties:
            disk_size: 10 GB
            num_cpus: { get_input: cpus }
            mem_size: 4 GB
        os:
          properties:
            architecture: x86_64
            type: Linux
            distribution: Fedora
            version: 18.0
      requirements:
        - local_storage:
            node: my_storage
            # Declare custom AttachesTo type using the 'relationship' keyword
            relationship:
              type: MyCustomAttachesTo
              properties:
                location: { get_input: storage_location }
   my_storage:
      type: BlockStorage
      properties:
        size: { get_input: storage_size }
        snapshot_id: { get_input: storage_snapshot_id }
```

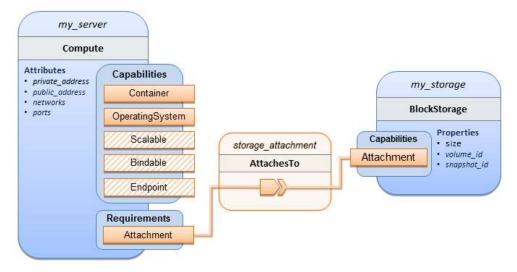
```
outputs:
    private_ip:
        description: The private IP address of the newly created compute instance.
        value: { get_attribute: [my_server, private_address] }
        volume_id:
        description: The volume id of the block storage instance.
        value: { get_attribute: [my_storage, volume_id] }
```

4282 11.1.6 Block Storage 3: Using a Relationship Template of type AttachesTo

4283 **11.1.6.1 Description**

- 4284 This use case demonstrates how to attach a TOSCA **BlockStorage** node to a **Compute** node using a
- 4285 TOSCA Relationship Template that is based upon the normative **AttachesTo** Relationship Type.

4286 11.1.6.2 Logical Diagram



4287

4288 11.1.6.3 Sample YAML

```
tosca_definitions_version: tosca_simple_yaml_1_0

description: >
  TOSCA simple profile with server and attached block storage using a named
Relationship Template for the storage attachment.

topology_template:
    inputs:
    cpus:
        type: integer
        description: Number of CPUs for the server.
```

```
constraints:
      - valid_values: [ 1, 2, 4, 8 ]
  storage_size:
    type: scalar-unit.size
    description: Size of the storage to be created.
    default: 1 GB
  storage_location:
    type: string
    description: Block storage mount point (filesystem path).
node_templates:
 my server:
    type: Compute
    capabilities:
      host:
        properties:
          disk size: 10 GB
          num_cpus: { get_input: cpus }
          mem size: 4 GB
      os:
        properties:
          architecture: x86_64
          type: Linux
          distribution: Fedora
          version: 18.0
    requirements:
      - local_storage:
          node: my_storage
          # Declare template to use with 'relationship' keyword
          relationship: storage_attachment
 my_storage:
    type: BlockStorage
    properties:
      size: { get_input: storage_size }
relationship_templates:
  storage_attachment:
    type: AttachesTo
    properties:
      location: { get_input: storage_location }
```

```
outputs:
    private_ip:
        description: The private IP address of the newly created compute instance.
        value: { get_attribute: [my_server, private_address] }
        volume_id:
        description: The volume id of the block storage instance.
        value: { get_attribute: [my_storage, volume_id] }
```

4290 11.1.7 Block Storage 4: Single Block Storage shared by 2-Tier Application 4291 with custom AttachesTo Type and implied relationships

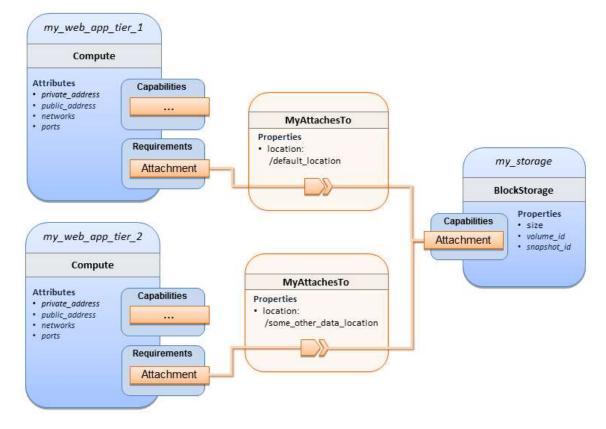
4292 **11.1.7.1 Description**

This use case shows 2 compute instances (2 tiers) with one BlockStorage node, and also uses a custom
 AttachesTo Relationship that provides a default mount point (i.e., **location**) which the 1st tier uses,
 but the 2nd tier provides a different mount point.

4296

4297 Please note that this use case assumes both Compute nodes are accessing different directories within4298 the shared, block storage node to avoid collisions.

4299 11.1.7.2 Logical Diagram



4301 11.1.7.3 Sample YAML

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: >
  TOSCA simple profile with a Single Block Storage node shared by 2-Tier Application with
custom AttachesTo Type and implied relationships.
relationship_types:
  MyAttachesTo:
    derived_from: tosca.relationships.AttachesTo
    properties:
      location:
        type: string
        default: /default_location
topology template:
  inputs:
    cpus:
      type: integer
      description: Number of CPUs for the server.
      constraints:
        - valid values: [ 1, 2, 4, 8 ]
    storage size:
      type: scalar-unit.size
      default: 1 GB
      description: Size of the storage to be created.
    storage_snapshot_id:
      type: string
      description: >
        Optional identifier for an existing snapshot to use when creating
storage.
  node templates:
    my_web_app_tier_1:
      type: tosca.nodes.Compute
      capabilities:
        host:
          properties:
            disk_size: 10 GB
            num_cpus: { get_input: cpus }
            mem_size: 4096 MB
```

```
os:
      properties:
        architecture: x86_64
        type: Linux
        distribution: Fedora
        version: 18.0
  requirements:
    - local_storage:
        node: my_storage
        relationship: MyAttachesTo
my_web_app_tier_2:
  type: tosca.nodes.Compute
  capabilities:
    host:
      properties:
        disk_size: 10 GB
        num_cpus: { get_input: cpus }
        mem size: 4096 MB
    os:
      properties:
        architecture: x86_64
        type: Linux
        distribution: Fedora
        version: 18.0
  requirements:
    - local_storage:
        node: my_storage
        relationship:
          type: MyAttachesTo
          properties:
            location: /some_other_data_location
my_storage:
  type: tosca.nodes.BlockStorage
  properties:
    size: { get_input: storage_size }
    snapshot_id: { get_input: storage_snapshot_id }
```

```
outputs:
```

private_ip_1:

description: The private IP address of the application's first tier.
value: { get_attribute: [my_web_app_tier_1, private_address] }
private_ip_2:
 description: The private IP address of the application's second tier.
 value: { get_attribute: [my_web_app_tier_2, private_address] }
volume_id:
 description: The volume id of the block storage instance.
 value: { get_attribute: [my_storage, volume_id] }

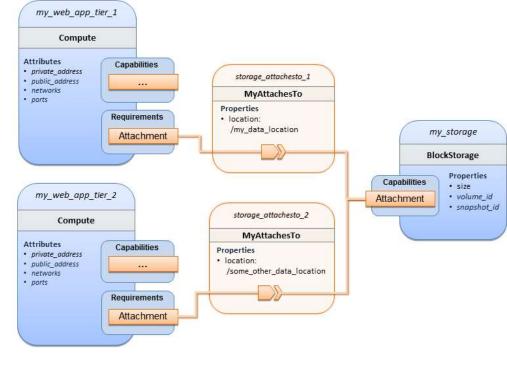
4302 11.1.8 Block Storage 5: Single Block Storage shared by 2-Tier Application 4303 with custom AttachesTo Type and explicit Relationship Templates

4304 **11.1.8.1 Description**

4305 This use case is like the Notation1 use case, but also creates two relationship templates (one for each 4306 tier) each of which provide a different mount point (i.e., **location**) which overrides the default location 4307 defined in the custom Relationship Type.

4308

Please note that this use case assumes both Compute nodes are accessing different directories withinthe shared, block storage node to avoid collisions.



4311 11.1.8.2 Logical Diagram

4312

4313 **11.1.8.3 Sample YAML**

tosca_definitions_version: tosca_simple_yaml_1_0

```
description: >
```

TOSCA simple profile with a single Block Storage node shared by 2-Tier Application with custom AttachesTo Type and explicit Relationship Templates.

```
relationship_types:
```

```
MyAttachesTo:
  derived_from: tosca.relationships.AttachesTo
  properties:
    location:
    type: string
    default: /default_location
```

```
topology_template:
```

```
inputs:
```

cpus:

```
type: integer
```

description: Number of CPUs for the server. constraints:

```
- valid_values: [ 1, 2, 4, 8 ]
```

```
storage_size:
```

type: scalar-unit.size

```
default: 1 GB
```

description: Size of the storage to be created.

```
storage_snapshot_id:
```

type: string

```
description: >
```

Optional identifier for an existing snapshot to use when creating storage.

```
storage_location:
  type: string
  description: >
```

Block storage mount point (filesystem path).

```
node_templates:
```

```
my_web_app_tier_1:
  type: tosca.nodes.Compute
  capabilities:
    host:
    properties:
    disk_size: 10 GB
```

```
num_cpus: { get_input: cpus }
          mem_size: 4096 MB
      os:
        properties:
          architecture: x86_64
          type: Linux
          distribution: Fedora
          version: 18.0
    requirements:
      - local_storage:
          node: my_storage
          relationship: storage_attachesto_1
 my_web_app_tier_2:
   type: tosca.nodes.Compute
   capabilities:
      host:
        properties:
          disk size: 10 GB
          num_cpus: { get_input: cpus }
          mem_size: 4096 MB
      os:
        properties:
          architecture: x86_64
          type: Linux
          distribution: Fedora
          version: 18.0
    requirements:
      - local_storage:
          node: my_storage
          relationship: storage_attachesto_2
 my_storage:
   type: tosca.nodes.BlockStorage
    properties:
      size: { get_input: storage_size }
      snapshot_id: { get_input: storage_snapshot_id }
relationship templates:
  storage_attachesto_1:
    type: MyAttachesTo
```

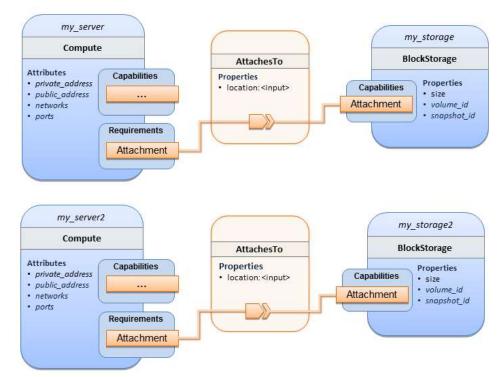
```
properties:
      location: /my_data_location
  storage_attachesto_2:
    type: MyAttachesTo
    properties:
      location: /some_other_data_location
outputs:
 private_ip_1:
    description: The private IP address of the application's first tier.
    value: { get_attribute: [my_web_app_tier_1, private_address] }
 private ip 2:
    description: The private IP address of the application's second tier.
    value: { get_attribute: [my_web_app_tier_2, private_address] }
 volume id:
    description: The volume id of the block storage instance.
    value: { get attribute: [my storage, volume id] }
```

4314 11.1.9 Block Storage 6: Multiple Block Storage attached to different Servers

4315 **11.1.9.1 Description**

- 4316 This use case demonstrates how two different TOSCA **BlockStorage** nodes can be attached to two
- 4317 different **Compute** nodes (i.e., servers) each using the normative **AttachesTo** relationship.





4319

4320 11.1.9.3 Sample YAML

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: >
  TOSCA simple profile with 2 servers each with different attached block storage.
topology_template:
  inputs:
    cpus:
      type: integer
      description: Number of CPUs for the server.
      constraints:
        - valid_values: [ 1, 2, 4, 8 ]
    storage_size:
      type: scalar-unit.size
      default: 1 GB
      description: Size of the storage to be created.
    storage_snapshot_id:
      type: string
```

```
description: >
```

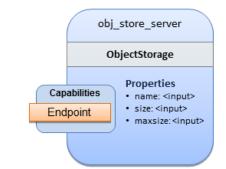
```
Optional identifier for an existing snapshot to use when creating
storage.
    storage_location:
      type: string
      description: >
        Block storage mount point (filesystem path).
  node_templates:
    my server:
      type: tosca.nodes.Compute
      capabilities:
        host:
          properties:
            disk size: 10 GB
            num_cpus: { get_input: cpus }
            mem_size: 4096 MB
        os:
          properties:
            architecture: x86_64
            type: Linux
            distribution: Fedora
            version: 18.0
      requirements:
         - local_storage:
             node: my_storage
             relationship:
               type: AttachesTo
               properties:
                 location: { get_input: storage_location }
    my_storage:
      type: tosca.nodes.BlockStorage
      properties:
        size: { get_input: storage_size }
        snapshot_id: { get_input: storage_snapshot_id }
    my_server2:
      type: tosca.nodes.Compute
      capabilities:
        host:
          properties:
            disk_size: 10 GB
```

```
num_cpus: { get_input: cpus }
          mem size: 4096 MB
      os:
        properties:
          architecture: x86_64
          type: Linux
          distribution: Fedora
          version: 18.0
    requirements:
       - local_storage:
           node: my storage2
           relationship:
             type: AttachesTo
             properties:
               location: { get_input: storage_location }
 my_storage2:
    type: tosca.nodes.BlockStorage
    properties:
      size: { get input: storage size }
      snapshot_id: { get_input: storage_snapshot_id }
outputs:
  server_ip_1:
    description: The private IP address of the application's first server.
    value: { get_attribute: [my_server, private_address] }
 server ip 2:
    description: The private IP address of the application's second server.
    value: { get_attribute: [my_server2, private_address] }
 volume_id_1:
    description: The volume id of the first block storage instance.
    value: { get_attribute: [my_storage, volume_id] }
 volume_id_2:
    description: The volume id of the second block storage instance.
    value: { get_attribute: [my_storage2, volume_id] }
```

4321 11.1.10 Object Storage 1: Creating an Object Storage service

4322 **11.1.10.1 Description**

4323 11.1.10.2 Logical Diagram



4324

4325 11.1.10.3 Sample YAML

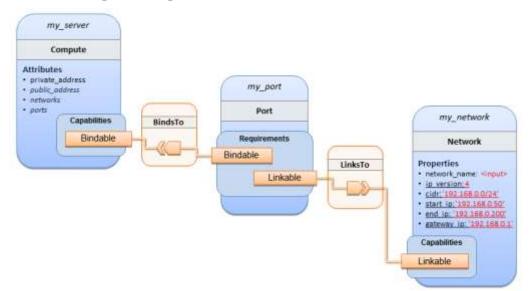
```
tosca_definitions_version: tosca_simple_yaml_1_0
description: >
    Tosca template for creating an object storage service.
topology_template:
    inputs:
    objectstore_name:
    type: string
node_templates:
    obj_store_server:
    type: tosca.nodes.ObjectStorage
    properties:
        name: { get_input: objectstore_name }
        size: 4096 MB
        maxsize: 20 GB
```

4326 **11.1.11 Network 1: Server bound to a new network**

4327 **11.1.11.1 Description**

Introduces the TOSCA Network and Port nodes used for modeling logical networks using the LinksTo and
 BindsTo Relationship Types. In this use case, the template is invoked without an existing network_name
 as an input property so a new network is created using the properties declared in the Network node.

4331 11.1.11.2 Logical Diagram



4332

4333 11.1.11.3 Sample YAML

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: >
  TOSCA simple profile with 1 server bound to a new network
topology_template:
  inputs:
    network_name:
      type: string
      description: Network name
  node_templates:
    my_server:
      type: tosca.nodes.Compute
      capabilities:
        host:
          properties:
            disk_size: 10 GB
            num_cpus: 1
            mem_size: 4096 MB
        os:
          properties:
            architecture: x86_64
```

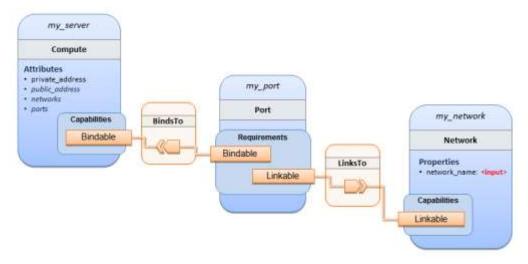
```
type: Linux
        distribution: CirrOS
        version: 0.3.2
my_network:
  type: tosca.nodes.network.Network
  properties:
    network_name: { get_input: network_name }
    ip version: 4
    cidr: '192.168.0.0/24'
    start_ip: '192.168.0.50'
    end_ip: '192.168.0.200'
    gateway_ip: '192.168.0.1'
my_port:
  type: tosca.nodes.network.Port
  requirements:
    - binding: my server
    - link: my network
```

4334 **11.1.12 Network 2: Server bound to an existing network**

4335 **11.1.12.1 Description**

4336 This use case shows how to use a **network_name** as an input parameter to the template to allow a server 4337 to be associated with an existing network.

4338 **11.1.12.2 Logical Diagram**



4339

4340 11.1.12.3 Sample YAML

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: >
  TOSCA simple profile with 1 server bound to an existing network
topology_template:
  inputs:
    network_name:
      type: string
      description: Network name
  node_templates:
    my_server:
      type: tosca.nodes.Compute
      capabilities:
        host:
          properties:
            disk_size: 10 GB
            num cpus: 1
            mem size: 4096 MB
        os:
          properties:
            architecture: x86_64
            type: Linux
            distribution: CirrOS
            version: 0.3.2
    my network:
      type: tosca.nodes.network.Network
      properties:
        network_name: { get_input: network_name }
    my_port:
      type: tosca.nodes.network.Port
      requirements:
        - binding:
            node: my_server
        - link:
            node: my_network
```

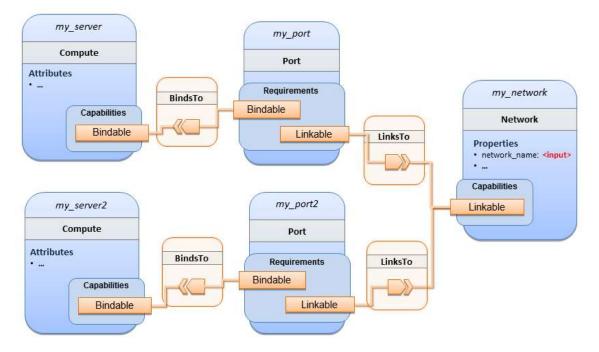
4341 **11.1.13 Network 3: Two servers bound to a single network**

4342 **11.1.13.1 Description**

4343 This use case shows how two servers (**Compute** nodes) can be bound to the same **Network** (node) using

4344 two logical network **Ports**.

4345 11.1.13.2 Logical Diagram



4346

4347 11.1.13.3 Sample YAML

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: >
  TOSCA simple profile with 2 servers bound to the 1 network
topology_template:
  inputs:
    network_name:
    type: string
    description: Network name
    network_cidr:
    type: string
    default: 10.0.0.0/24
    description: CIDR for the network
    network_start_ip:
```

```
type: string
    default: 10.0.0.100
    description: Start IP for the allocation pool
 network_end_ip:
    type: string
    default: 10.0.0.150
    description: End IP for the allocation pool
node_templates:
 my_server:
    type: tosca.nodes.Compute
    capabilities:
      host:
        properties:
          disk_size: 10 GB
          num_cpus: 1
          mem size: 4096 MB
      os:
        properties:
          architecture: x86_64
          type: Linux
          distribution: CirrOS
          version: 0.3.2
 my_server2:
    type: tosca.nodes.Compute
    capabilities:
      host:
        properties:
          disk_size: 10 GB
          num_cpus: 1
          mem_size: 4096 MB
      os:
        properties:
          architecture: x86_64
          type: Linux
          distribution: CirrOS
          version: 0.3.2
 my_network:
    type: tosca.nodes.network.Network
```

```
properties:
```

```
ip_version: 4
cidr: { get_input: network_cidr }
network_name: { get_input: network_name }
start_ip: { get_input: network_start_ip }
end_ip: { get_input: network_end_ip }
```

my_port:

type: tosca.nodes.network.Port

requirements:

- binding: my_server
- link: my_network

my_port2:

type: tosca.nodes.network.Port

requirements:

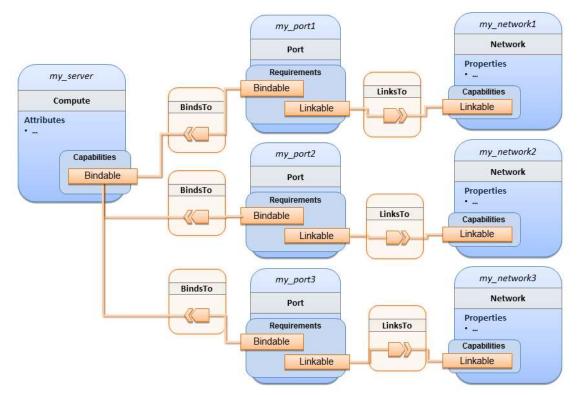
- binding: my_server2
- link: my_network

4348 11.1.14 Network 4: Server bound to three networks

4349 11.1.14.1 Description

This use case shows how three logical networks (Network), each with its own IP address range, can be bound to with the same server (Compute node).

4352 11.1.14.2 Logical Diagram



4353

4354 11.1.14.3 Sample YAML

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: >
  TOSCA simple profile with 1 server bound to 3 networks
topology_template:
  node templates:
    my server:
      type: tosca.nodes.Compute
      capabilities:
        host:
          properties:
            disk_size: 10 GB
            num_cpus: 1
            mem_size: 4096 MB
        os:
          properties:
            architecture: x86_64
```

```
type: Linux
        distribution: CirrOS
        version: 0.3.2
my_network1:
  type: tosca.nodes.network.Network
  properties:
    cidr: '192.168.1.0/24'
    network_name: net1
my_network2:
  type: tosca.nodes.network.Network
  properties:
    cidr: '192.168.2.0/24'
    network_name: net2
my network3:
  type: tosca.nodes.network.Network
  properties:
    cidr: '192.168.3.0/24'
    network_name: net3
my_port1:
  type: tosca.nodes.network.Port
  properties:
    order: 0
  requirements:
    - binding: my_server
    - link: my_network1
my_port2:
  type: tosca.nodes.network.Port
  properties:
    order: 1
  requirements:
    - binding: my_server
    - link: my_network2
my port3:
  type: tosca.nodes.network.Port
  properties:
```

order: 2

requirements:

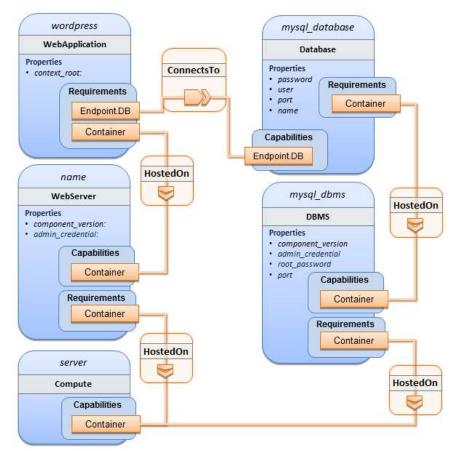
- binding: my_server
- link: my_network3

4355 11.1.15 WebServer-DBMS 1: WordPress + MySQL, single instance

4356 **11.1.15.1 Description**

4357 TOSCA simple profile service showing the WordPress web application with a MySQL database hosted on 4358 a single server (instance).

4359 **11.1.15.2 Logical Diagram**



4360

4361 11.1.15.3 Sample YAML

```
tosca_definitions_version: tosca_simple_yaml_1_0
```

description: >

TOSCA simple profile with WordPress, a web server, a MySQL DBMS hosting the application's database content on the same server. Does not have input defaults or constraints.

```
topology_template:
  inputs:
    cpus:
      type: integer
      description: Number of CPUs for the server.
    db_name:
      type: string
      description: The name of the database.
    db user:
      type: string
      description: The username of the DB user.
    db pwd:
      type: string
      description: The WordPress database admin account password.
    db_root_pwd:
      type: string
      description: Root password for MySQL.
    db port:
      type: PortDef
      description: Port for the MySQL database
  node_templates:
    wordpress:
      type: tosca.nodes.WebApplication.WordPress
      properties:
        context_root: { get_input: context_root }
      requirements:
        - host: webserver
        - database_endpoint: mysql_database
      interfaces:
        Standard:
          create: wordpress install.sh
          configure:
            implementation: wordpress_configure.sh
            inputs:
              wp_db_name: { get_property: [ mysql_database, name ] }
              wp_db_user: { get_property: [ mysql_database, user ] }
              wp db password: { get property: [ mysql database, password ] }
              # In my own template, find requirement/capability, find port
property
```

```
wp_db_port: { get_property: [ SELF, database_endpoint, port ] }
mysql_database:
  type: Database
  properties:
    name: { get_input: db_name }
    user: { get_input: db_user }
    password: { get_input: db_pwd }
    port: { get_input: db_port }
  capabilities:
    database endpoint:
      properties:
        port: { get_input: db_port }
  requirements:
    - host: mysql dbms
  interfaces:
    Standard:
      configure: mysql database configure.sh
mysql_dbms:
  type: DBMS
  properties:
    root_password: { get_input: db_root_pwd }
    port: { get_input: db_port }
  requirements:
    - host: server
  interfaces:
    Standard:
      inputs:
          db_root_password: { get_property: [ mysql_dbms, root_password ] }
      create: mysql_dbms_install.sh
      start: mysql dbms start.sh
      configure: mysql dbms configure.sh
webserver:
  type: WebServer
  requirements:
    - host: server
  interfaces:
    Standard:
      create: webserver_install.sh
```

```
start: webserver_start.sh
  server:
    type: Compute
    capabilities:
      host:
        properties:
          disk_size: 10 GB
          num_cpus: { get_input: cpus }
          mem size: 4096 MB
      os:
        properties:
          architecture: x86_64
          type: linux
          distribution: fedora
          version: 17.0
outputs:
 website url:
    description: URL for Wordpress wiki.
    value: { get_attribute: [server, public_address] }
```

4362 11.1.15.4 Sample scripts

4363 Where the referenced implementation scripts in the example above would have the following contents

4364 11.1.15.4.1 wordpress_install.sh

yum -y install wordpress

4365 11.1.15.4.2 wordpress_configure.sh

sed -i "/Deny from All/d" /etc/httpd/conf.d/wordpress.conf
sed -i "s/Require local/Require all granted/" /etc/httpd/conf.d/wordpress.conf
sed -i s/database_name_here/name/ /etc/wordpress/wp-config.php
sed -i s/username_here/user/ /etc/wordpress/wp-config.php
sed -i s/password_here/password/ /etc/wordpress/wp-config.php
systemctl restart httpd.service

4366 11.1.15.4.3 mysql_database_configure.sh

Setup MySQL root password and create user cat << EOF | mysql -u root --password=db_root_password</pre> CREATE DATABASE name; GRANT ALL PRIVILEGES ON name.* TO "user"@"localhost" IDENTIFIED BY "password"; FLUSH PRIVILEGES; EXIT EOF

4367 11.1.15.4.4 mysql_dbms_install.sh

yum -y install mysql mysql-server
Use systemd to start MySQL server at system boot time
systemctl enable mysqld.service

4368 11.1.15.4.5 mysql_dbms_start.sh

Start the MySQL service (NOTE: may already be started at image boot time)
systemctl start mysqld.service

4369 11.1.15.4.6 mysql_dbms_configure

Set the MySQL server root password mysqladmin -u root password db_root_password

4370 11.1.15.4.7 webserver_install.sh

yum -y install httpd
systemctl enable httpd.service

4371 11.1.15.4.8 webserver_start.sh

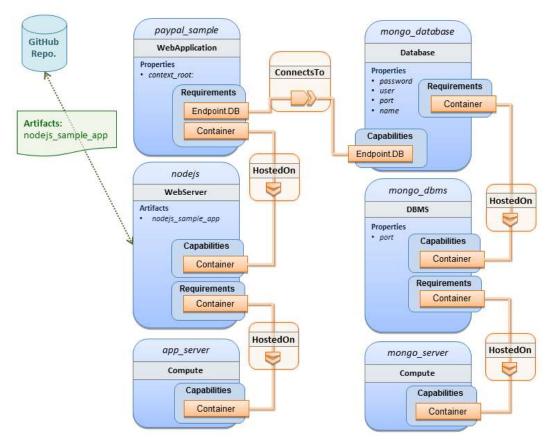
Start the httpd service (NOTE: may already be started at image boot time)
systemctl start httpd.service

4372 11.1.16 WebServer-DBMS 2: Nodejs with PayPal Sample App and MongoDB 4373 on separate instances

4374 **11.1.16.1 Description**

4375 This use case Instantiates a 2-tier application with Nodejs and its (PayPal sample) WebApplication on 4376 one tier which connects a MongoDB database (which stores its application data) using a ConnectsTo 4377 relationship.

4378 11.1.16.2 Logical Diagram



4379

4380 11.1.16.3 Sample YAML

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: >
  TOSCA simple profile with a nodejs web server hosting a PayPal sample
application which connects to a mongodb database.
```

```
imports:
```

- custom_types/paypalpizzastore_nodejs_app.yaml

```
dsl_definitions:
    ubuntu_node: &ubuntu_node
    disk_size: 10 GB
    num_cpus: { get_input: my_cpus }
    mem_size: 4096 MB
    os_capabilities: &os_capabilities
    architecture: x86_64
    type: Linux
```

```
distribution: Ubuntu
      version: 14.04
topology_template:
  inputs:
    my_cpus:
      type: integer
      description: Number of CPUs for the server.
      constraints:
        - valid_values: [ 1, 2, 4, 8 ]
      default: 1
    github url:
       type: string
       description: The URL to download nodejs.
       default: https://github.com/sample.git
  node templates:
    paypal pizzastore:
      type: tosca.nodes.WebApplication.PayPalPizzaStore
      properties:
          github_url: { get_input: github_url }
      requirements:
        - host:nodejs
        - database_connection: mongo_db
      interfaces:
        Standard:
           configure:
             implementation: scripts/nodejs/configure.sh
             inputs:
               github_url: { get_property: [ SELF, github_url ] }
               mongodb_ip: { get_attribute: [mongo_server, private_address] }
           start: scriptsscripts/nodejs/start.sh
    nodejs:
      type: tosca.nodes.WebServer.Nodejs
      requirements:
        - host: app server
      interfaces:
        Standard:
          create: scripts/nodejs/create.sh
```

```
mongo_db:
  type: tosca.nodes.Database
  requirements:
    - host: mongo_dbms
  interfaces:
    Standard:
     create: create_database.sh
mongo_dbms:
  type: tosca.nodes.DBMS
  requirements:
    - host: mongo_server
  properties:
    port: 27017
  interfaces:
    tosca.interfaces.node.lifecycle.Standard:
      create: mongodb/create.sh
      configure:
        implementation: mongodb/config.sh
        inputs:
          mongodb_ip: { get_attribute: [mongo_server, private_address] }
      start: mongodb/start.sh
mongo_server:
  type: tosca.nodes.Compute
  capabilities:
    os:
      properties: *os_capabilities
    host:
      properties: *ubuntu_node
app_server:
  type: tosca.nodes.Compute
  capabilities:
    os:
      properties: *os_capabilities
    host:
      properties: *ubuntu_node
```

```
outputs:
```

nodejs_url: description: URL for the nodejs server, http://<IP>:3000 value: { get_attribute: [app_server, private_address] } mongodb_url: description: URL for the mongodb server. value: { get_attribute: [mongo_server, private_address] }

4381 **11.1.16.4 Notes:**

• Scripts referenced in this example are assumed to be placed by the TOSCA orchestrator in the relative directory declared in TOSCA.meta of the TOSCA CSAR file.

4384 11.1.17 Multi-Tier-1: Elasticsearch, Logstash, Kibana (ELK) use case with 4385 multiple instances

4386 **11.1.17.1 Description**

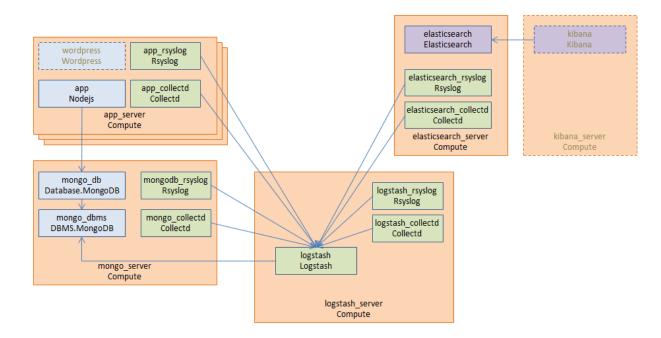
- 4387 TOSCA simple profile service showing the Nodejs, MongoDB, Elasticsearch, Logstash, Kibana, rsyslog4388 and collectd installed on a different server (instance).
- 4389

4392

4393

- 4390 This use case also demonstrates:
- Use of TOSCA macros or dsl_definitions
 - Multiple SoftwareComponents hosted on same Compute node
 - Multiple tiers communicating to each other over ConnectsTo using Configure interface.

4394 11.1.17.2 Logical Diagram



4395

4396 **11.1.17.3 Sample YAML**

4397 11.1.17.3.1 Master Service Template application (Entry-Definitions)

TheThe following YAML is the primary template (i.e., the Entry-Definition) for the overall use case. The imported YAML for the various subcomponents are not shown here for brevity.

```
4400
```

tosca_definitions_version: tosca_simple_yaml_1_0

```
description: >
```

This TOSCA simple profile deploys nodejs, mongodb, elasticsearch, logstash and kibana each on a separate server with monitoring enabled for nodejs server where a sample nodejs application is running. The syslog and collectd are installed on a nodejs server.

imports:

- paypalpizzastore_nodejs_app.yaml
- elasticsearch.yaml
- logstash.yaml
- kibana.yaml
- collectd.yaml
- rsyslog.yaml

```
dsl_definitions:
```

```
host_capabilities: &host_capabilities
    # container properties (flavor)
    disk_size: 10 GB
    num_cpus: { get_input: my_cpus }
    mem_size: 4096 MB
    os_capabilities: &os_capabilities
    architecture: x86_64
    type: Linux
    distribution: Ubuntu
    version: 14.04
```

```
topology_template:
inputs:
  my_cpus:
   type: integer
   description: Number of CPUs for the server.
   constraints:
        - valid_values: [ 1, 2, 4, 8 ]
   github_url:
```

```
type: string
     description: The URL to download nodejs.
     default: https://github.com/sample.git
node_templates:
 paypal_pizzastore:
   type: tosca.nodes.WebApplication.PayPalPizzaStore
    properties:
        github_url: { get_input: github_url }
    requirements:
      - host: nodejs
      - database connection: mongo db
   interfaces:
      Standard:
         configure:
           implementation: scripts/nodejs/configure.sh
           inputs:
             github_url: { get_property: [ SELF, github_url ] }
             mongodb ip: { get attribute: [mongo server, private address] }
         start: scripts/nodejs/start.sh
 nodejs:
   type: tosca.nodes.WebServer.Nodejs
    requirements:
      - host: app_server
   interfaces:
      Standard:
        create: scripts/nodejs/create.sh
 mongo_db:
   type: tosca.nodes.Database
    requirements:
      - host: mongo dbms
   interfaces:
      Standard:
       create: create_database.sh
 mongo dbms:
   type: tosca.nodes.DBMS
    requirements:
      - host: mongo_server
```

```
interfaces:
        tosca.interfaces.node.lifecycle.Standard:
          create: scripts/mongodb/create.sh
          configure:
            implementation: scripts/mongodb/config.sh
            inputs:
              mongodb_ip: { get_attribute: [mongo_server, ip_address] }
          start: scripts/mongodb/start.sh
    elasticsearch:
      type: tosca.nodes.SoftwareComponent.Elasticsearch
      requirements:
        - host: elasticsearch_server
      interfaces:
        tosca.interfaces.node.lifecycle.Standard:
          create: scripts/elasticsearch/create.sh
          start: scripts/elasticsearch/start.sh
    logstash:
      type: tosca.nodes.SoftwareComponent.Logstash
      requirements:
        - host: logstash_server
        - search_endpoint: elasticsearch
          interfaces:
            tosca.interfaces.relationship.Configure:
              pre configure source:
                implementation: python/logstash/configure_elasticsearch.py
                input:
                  elasticsearch_ip: { get_attribute: [elasticsearch_server,
ip_address] }
      interfaces:
        tosca.interfaces.node.lifecycle.Standard:
          create: scripts/lostash/create.sh
          configure: scripts/logstash/config.sh
          start: scripts/logstash/start.sh
    kibana:
      type: tosca.nodes.SoftwareComponent.Kibana
      requirements:
        - host: kibana_server
        - search_endpoint: elasticsearch
      interfaces:
```

```
tosca.interfaces.node.lifecycle.Standard:
          create: scripts/kibana/create.sh
          configure:
            implementation: scripts/kibana/config.sh
            input:
              elasticsearch_ip: { get_attribute: [elasticsearch_server,
ip address] }
              kibana_ip: { get_attribute: [kibana_server, ip_address] }
          start: scripts/kibana/start.sh
    app collectd:
      type: tosca.nodes.SoftwareComponent.Collectd
      requirements:
        - host: app server
        - collectd_endpoint: logstash
          interfaces:
            tosca.interfaces.relationship.Configure:
              pre_configure_target:
                implementation: python/logstash/configure_collectd.py
      interfaces:
        tosca.interfaces.node.lifecycle.Standard:
          create: scripts/collectd/create.sh
          configure:
            implementation: python/collectd/config.py
            input:
              logstash ip: { get attribute: [logstash server, ip address] }
          start: scripts/collectd/start.sh
   app_rsyslog:
      type: tosca.nodes.SoftwareComponent.Rsyslog
      requirements:
        - host: app server
        - rsyslog_endpoint: logstash
          interfaces:
            tosca.interfaces.relationship.Configure:
              pre_configure_target:
                implementation: python/logstash/configure_rsyslog.py
      interfaces:
        tosca.interfaces.node.lifecycle.Standard:
          create: scripts/rsyslog/create.sh
          configure:
```

```
implementation: scripts/rsyslog/config.sh
        input:
          logstash_ip: { get_attribute: [logstash_server, ip_address] }
      start: scripts/rsyslog/start.sh
app_server:
  type: tosca.nodes.Compute
  capabilities:
    host:
      properties: *host_capabilities
    os:
      properties: *os_capabilities
mongo_server:
  type: tosca.nodes.Compute
  capabilities:
    host:
      properties: *host_capabilities
    os:
      properties: *os_capabilities
elasticsearch_server:
  type: tosca.nodes.Compute
  capabilities:
    host:
      properties: *host_capabilities
    os:
      properties: *os_capabilities
logstash_server:
  type: tosca.nodes.Compute
  capabilities:
    host:
      properties: *host_capabilities
    os:
      properties: *os_capabilities
kibana_server:
  type: tosca.nodes.Compute
  capabilities:
    host:
```

```
properties: *host_capabilities
      os:
        properties: *os_capabilities
outputs:
 nodejs_url:
    description: URL for the nodejs server.
    value: { get_attribute: [ app_server, private_address ] }
 mongodb url:
    description: URL for the mongodb server.
    value: { get attribute: [ mongo server, private address ] }
  elasticsearch url:
    description: URL for the elasticsearch server.
    value: { get attribute: [ elasticsearch server, private address ] }
 logstash url:
    description: URL for the logstash server.
    value: { get_attribute: [ logstash_server, private_address ] }
  kibana url:
    description: URL for the kibana server.
    value: { get_attribute: [ kibana_server, private_address ] }
```

4401 **11.1.17.4 Sample scripts**

4402 Where the referenced implementation scripts in the example above would have the following contents

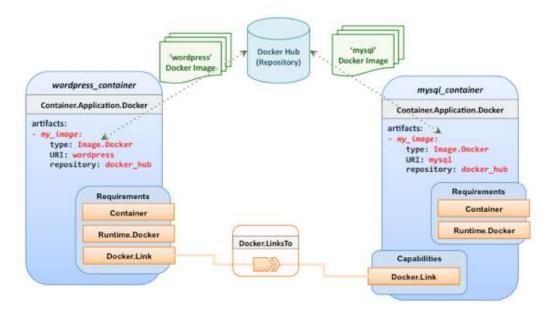
4403 11.1.18 Container-1: Containers using Docker single Compute instance 4404 (Containers only)

4405 **11.1.18.1 Description**

This use case shows a minimal description of two Container nodes (only) providing their Docker
Requirements allowing platform (orchestrator) to select/provide the underlying Docker implementation
(Capability). Specifically, wordpress and mysql Docker images are referenced from Docker Hub.

- 4410 This use case also demonstrates:
- 4411
 Abstract description of Requirements (i.e., Container and Docker) allowing platform to dynamically select the appropriate runtime Capabilities that match.
- Use of external repository (Docker Hub) to reference image artifact.

4414 11.1.18.2 Logical Diagram



4415

4416 11.1.18.3 Sample YAML

4417 11.1.18.3.1 Two Docker "Container" nodes (Only) with Docker Requirements

```
tosca definitions version: tosca simple yaml 1 0
description: >
  TOSCA simple profile with wordpress, web server and mysql on the same server.
# Repositories to retrieve code artifacts from
repositories:
  docker_hub: https://registry.hub.docker.com/
topology_template:
  inputs:
    wp_host_port:
      type: integer
      description: The host port that maps to port 80 of the WordPress container.
    db root pwd:
      type: string
      description: Root password for MySQL.
  node templates:
    # The MYSQL container based on official MySQL image in Docker hub
```

```
mysql_container:
      type: tosca.nodes.Container.Application.Docker
      capabilities:
        # This is a capability that would mimic the Docker -link feature
        database_link: tosca.capabilities.Docker.Link
      artifacts:
        my_image:
          file: mysql
          type: tosca.artifacts.Deployment.Image.Container.Docker
          repository: docker hub
      interfaces:
        Standard:
          create:
            implementation: my_image
            inputs:
              db_root_password: { get_input: db_root_pwd }
    # The WordPress container based on official WordPress image in Docker hub
    wordpress container:
      type: tosca.nodes.Container.Application.Docker
      requirements:
        - database_link: mysql_container
      artifacts:
        my_image:
          file: wordpress
          type: tosca.artifacts.Deployment.Image.Container.Docker
          repository: docker_hub
          <metadata-link> : <topology_artifact_name> # defined outside and linked
to from here
      interfaces:
        Standard:
          create:
            implementation: my_image
            inputs:
              host_port: { get_input: wp_host_port }
```

```
4418
```

4419 **12TOSCA Policies**

This section is **non-normative** and describes the approach TOSCA Simple Profile plans to take for policy
description with TOSCA Service Templates. In addition, it explores how existing TOSCA Policy Types
and definitions might be applied in the future to express operational policy use cases.

4423 12.1 A declarative approach

4424 TOSCA Policies are a type of requirement that govern use or access to resources which can be 4425 expressed independently from specific applications (or their resources) and whose fulfillment is not 4426 discretely expressed in the application's topology (i.e., via TOSCA Capabilities).

4427

TOSCA deems it not desirable for a declarative model to encourage external intervention for resolving
policy issues (i.e., via imperative mechanisms external to the Cloud). Instead, the Cloud provider is
deemed to be in the best position to detect when policy conditions are triggered, analyze the affected
resources and enforce the policy against the allowable actions declared within the policy itself.

4432 **12.1.1 Declarative considerations**

- Natural language rules are not realistic, too much to represent in our specification; however, regular
 expressions can be used that include simple operations and operands that include symbolic names
 for TOSCA metamodel entities, properties and attributes.
- Complex rules can actually be directed to an external policy engine (to check for violation) returns
 true|false then policy says what to do (trigger or action).
- 4438 Actions/Triggers could be:
- Autonomic/Platform corrects against user-supplied criteria
- External monitoring service could be utilized to monitor policy rules/conditions against metrics,
 the monitoring service could coordinate corrective actions with external services (perhaps
 Workflow engines that can analyze the application and interact with the TOSCA instance model).

4443 **12.2 Consideration of Event, Condition and Action**

4444 **12.3 Types of policies**

- 4445 Policies typically address two major areas of concern for customer workloads:
- Access Control assures user and service access to controlled resources are governed by
 rules which determine general access permission (i.e., allow or deny) and conditional access
 dependent on other considerations (e.g., organization role, time of day, geographic location, etc.).
- Placement assures affinity (or anti-affinity) of deployed applications and their resources; that is,
 what is allowed to be placed where within a Cloud provider's infrastructure.
- 4451
 Quality-of-Service (and continuity) assures performance of software components (perhaps captured as quantifiable, measure components within an SLA) along with consideration for scaling and failover.

4454 **12.3.1 Access control policies**

Although TOSCA Policy definitions could be used to express and convey access control policies,
definitions of policies in this area are out of scope for this specification. At this time, TOSCA encourages
organizations that already have standards that express policy for access control to provide their own
guidance on how to use their standard with TOSCA.

4459 12.3.2 Placement policies

4460 There must be control mechanisms in place that can be part of these patterns that accept governance 4461 policies that allow control expressions of what is allowed when placing, scaling and managing the 4462 applications that are enforceable and verifiable in Cloud.

- 4463
- 4464 These policies need to consider the following:
- Regulated industries need applications to control placement (deployment) of applications to different countries or regions (i.e., different logical geographical boundaries).

4467 12.3.2.1 Placement for governance concerns

In general, companies and individuals have security concerns along with general "loss of control" issues
when considering deploying and hosting their highly valued application and data to the Cloud. They want
to control placement perhaps to ensure their applications are only placed in datacenter they trust or
assure that their applications and data are not placed on shared resources (i.e., not co-tenanted).

4472

In addition, companies that are related to highly regulated industries where compliance with government,
 industry and corporate policies is paramount. In these cases, having the ability to control placement of

4475 applications is an especially significant consideration and a prerequisite for automated orchestration.

4476 12.3.2.2 Placement for failover

4477 Companies realize that their day-to-day business must continue on through unforeseen disasters that 4478 might disable instances of the applications and data at or on specific data centers, networks or servers. 4479 They need to be able to convey placement policies for their software applications and data that mitigate 4480 risk of disaster by assuring these cloud assets are deployed strategically in different physical locations. 4481 Such policies need to consider placement across geographic locations as wide as countries, regions, 4482 datacenters, as well as granular placement on a network, server or device within the same physical 4483 datacenter. Cloud providers must be able to not only enforce these policies but provide robust and 4484 seamless failover such that a disaster's impact is never perceived by the end user.

4485 12.3.3 Quality-of-Service (QoS) policies

4486 Quality-of-Service (apart from failover placement considerations) typically assures that software 4487 applications and data are available and performant to the end users. This is usually something that is 4488 measurable in terms of end-user responsiveness (or response time) and often qualified in SLAs established between the Cloud provider and customer. These QoS aspects can be taken from SLAs and 4489 4490 legal agreements and further encoded as performance policies associated with the actual applications 4491 and data when they are deployed. It is assumed that Cloud provider is able to detect high utilization (or 4492 usage load) on these applications and data that deviate from these performance policies and is able to 4493 bring them back into compliance.

4494

4495 **12.4 Policy relationship considerations**

- Performance policies can be related to scalability policies. Scalability policies tell the Cloud provider
 exactly how to scale applications and data when they detect an application's performance policy is
 (or about to be) violated (or triggered).
- Scalability policies in turn are related to placement policies which govern where the application and data can be scaled to.
- There are general "tenant" considerations that restrict what resources are available to applications 4502 and data based upon the contract a customer has with the Cloud provider. This includes other

4503 constraints imposed by legal agreements or SLAs that are not encoded programmatically or 4504 associated directly with actual application or data..

4505 **12.5 Use Cases**

This section includes some initial operation policy use cases that we wish to describe using the TOSCA
metamodel. More policy work will be done in future versions of the TOSCA Simple Profile in YAML
specification.

- 4509 12.5.1 Placement
- 4510 12.5.1.1 Use Case 1: Simple placement for failover

4511 **12.5.1.1.1 Description**

This use case shows a failover policy to keep at least 3 copies running in separate containers. In this simple case, the specific containers to use (or name is not important; the Cloud provider must assure placement separation (anti-affinity) in three physically separate containers.

4515 **12.5.1.1.2 Features**

4516 This use case introduces the following policy features:

- Simple separation on different "compute" nodes (up to discretion of provider).
- Simple separation by region (a logical container type) using an allowed list of region names
 relative to the provider.
- 4520 o Also, shows that set of allowed "regions" (containers) can be greater than the number of containers requested.

4522 **12.5.1.1.3 Logical Diagram**

4523 Sample YAML: Compute separation

failover_policy_1:

type: tosca.policy.placement.Antilocate description: My placement policy for Compute node separation properties: # 3 diff target containers container_type: Compute container_number: 3

4524 **12.5.1.1.4 Notes**

- 4525
 There may be availability (constraints) considerations especially if these policies are applied to "clusters".
- There may be future considerations for controlling max # of instances per container.

4528 12.5.1.2 Use Case 2: Controlled placement by region

4529 **12.5.1.2.1 Description**

- 4530 This use case demonstrates the use of named "containers" which could represent the following:
- Datacenter regions

- Geographic regions (e.g., cities, municipalities, states, countries, etc.)
- Commercial regions (e.g., North America, Eastern Europe, Asia Pacific, etc.)

4534 12.5.1.2.2 Features

- 4535 This use case introduces the following policy features:
- Separation of resources (i.e., TOSCA nodes) by logical regions, or zones.

4537 12.5.1.2.3 Sample YAML: Region separation amongst named set of regions

```
failover_policy_2:
  type: tosca.policy.placement
  description: My failover policy with allowed target regions (logical
  containers)
  properties:
    container_type: region
    container_number: 3
    # If "containers" keyname is provided, they represent the allowed set
    # of target containers to use for placement for .
    containers: [ region1, region2, region3, region4 ]
```

4538 12.5.1.3 Use Case 3: Co-locate based upon Compute affinity

4539 **12.5.1.3.1 Description**

- 4540 Nodes that need to be co-located to achieve optimal performance based upon access to similar
- 4541 Infrastructure (laaS) resource types (i.e., Compute, Network and/or Storage).
- 4542
- This use case demonstrates the co-location based upon Compute resource affinity; however, the same approach could be taken for Network as or Storage affinity as well. :

4545 **12.5.1.3.2 Features**

- 4546 This use case introduces the following policy features:
- Node placement based upon Compute resource affinity.

4548 12.5.1.4 Notes

The concept of placement based upon laaS resource utilization is not future-thinking, as Cloud
 should guarantee equivalent performance of application performance regardless of placement.
 That is, all network access between application nodes and underlying Compute or Storage should
 have equivalent performance (e.g., network bandwidth, network or storage access time, CPU
 speed, etc.).

4554 12.5.1.4.1 Sample YAML: Region separation amongst named set of regions

keep_together_policy:
 type: tosca.policy.placement.Colocate
 description: Keep associated nodes (groups of nodes) based upon Compute
 properties:
 affinity: Compute

4555 **12.5.2 Scaling**

- 4556 12.5.2.1 Use Case 1: Simple node autoscale
- 4557 **12.5.2.1.1 Description**
- 4558 Start with X nodes and scale up to Y nodes, capability to do this from a dashboard for example.

4559 12.5.2.1.2 Features

- 4560 This use case introduces the following policy features:
- Basic autoscaling policy

4562 12.5.2.1.3 Sample YAML

```
my_scaling_policy_1:
  type: tosca.policy.scaling
  description: Simple node autoscaling
  properties:
    min_instances: <integer>
    max_instances: <integer>
    default_instances: <integer>
    increment: <integer>
```

4563 **12.5.2.1.4 Notes**

4564	٠	Assume horizontal scaling for this use case
4565		 Horizontal scaling, implies "stack-level" control using Compute nodes to define a "stack"
4566		(i.e., The Compute node's entire HostedOn relationship dependency graph is considered
4567		part of its "stack")
4568	٠	Assume Compute node has a SoftwareComponent that represents a VM application.
4569	٠	Availability Zones (and Regions if not same) need to be considered in further
4570		use cases.
4571	٠	If metrics are introduced, there is a control-loop (that monitors). Autoscaling is a special concept
4572		that includes these considerations.
4573	٠	Mixed placement and scaling use cases need to be considered:
4574		• <i>Example</i> : Compute1 and Compute2 are 2 node templates. Compute1 has 10 instances, 5
4575		in one region 5 in other region.

4576 13 Artifact Processing and creating portable Service 4577 Templates

TOSCA's declarative modelling includes features that allow service designers to model abstract
components without having to specify concrete implementations for these components. Declarative
modeling is made possible through the use of standardized TOSCA types. Any TOSCA-compliant
orchestrator is expected to know how to deploy these standard types. Declarative modeling ensures
optimal portability of service templates, since any cloud-specific or technology specific implementation
logic is provided by the TOSCA orchestrator, not by the service template.

- 4584
- The examples in the previous chapter also demonstrate how TOSCA allows service designers to extend built-in orchestrator behavior in a number of ways:
- 4587 Service designers can override or extend behavior of built-in types by supplying service-specific
 4588 implementations of lifecycle interface operations in their node templates.
- 4589 Service designers can create entirely new types that define custom implementations of standard
 4590 lifecycle interfaces.
- 4591 Implementations of Interface operations are provided through artifacts. The examples in the previous 4592 chapter showed shell script artifacts, but many other types of artifacts can be used as well. The use of 4593 artifacts in TOSCA service templates breaks pure declarative behavior since artifacts effectively contain 4594 "imperative logic" that is opaque to the orchestrator. This introduces the risk of non-portable templates.
- 4595 Since some artifacts may have dependencies on specific technologies or infrastructure component, the
- 4596 use of artifacts could result in service templates that cannot be used on all cloud infrastructures.
- 4597

4598 The goal of this **non-normative** chapter is to ensure portable and interoperable use of artifacts by 4599 providing a detailed description of how TOSCA orchestrators process artifacts, by illustrating how a 4600 number of standard TOSCA artifact types are expected to be processed, and by describing TOSCA 4601 language features that allow artifact to provide metadata containing artifact-specific processing 4602 instructions. These metadata around the artifact allow the orchestrator to make descisions on the correct Artifact Processor and runtime(s) needed to execute. The sole purpose of this chapter is to show TOSCA 4603 4604 template designers how to best leverage built-in TOSCA capabilities. It is not intended to recommend 4605 specific orchestrator implementations.

4606 13.1 Artifacts Processing

- 4607 Artifacts represent the content needed to realize a deployment or implement a specific management 4608 action.
- 4609
- 4610 Artifacts can be of many different types. Artifacts could be executables (such as scripts or executable 4611 program files) or pieces of data required by those executables (e.g. configuration files, software libraries,
- 4612 license keys, etc). Implementations for some operations may require the use of multiple artifacts.
- 4613
- 4614 Different types of artifacts may require different mechanisms for processing the artifact. However, the 4615 sequence of steps taken by an orchestrator to process an artifcat is generally the same for all types of 4616 artifacts:

4617 13.1.1 Identify Artifact Processor

The first step is to identify an appropriate processor for the specified artifact. A processor is any
executable that knows how to process the artifact in order to achieve the intended management
operation. This processor could be an interpreter for executable shell scripts or scripts written in Python. It
could be a tool such as Ansible, Puppet, or Chef for playbook, manifest, or recipe artifacts, or it could be a

4622 container management or cloud management system for image artifacts such as container images or 4623 virtual machine images.

4624

4625 TOSCA includes a number of standard artifact types. Standard-compliant TOSCA orchestrators are 4626 expected to include processors for each of these types. For each type, there is a correspondent Artifact 4627 Processor that is responsible for processing artifacts of that type.

4628

4629 Note that aside from selecting the proper artifact processor, it may also be important to use the proper
4630 version of the processor. For example, some python scripts may require Python 2.7 whereas other scripts
4631 may require Python 3.4. TOSCA provides metadata to describe service template-specific parameters for
4632 the Artifact Processor. In addition to specifying specific versions, those metadata could also identify

- 4633 repositories from which to retrieve the artifact processor.
- 4634

4635 Some templates may require the use of custom Artifact Processors, for example to process non-standard 4636 artifacts or to provide a custom Artifact Processor for standard artifact types. For such cases, TOSCA 4637 allows service template designers to define Application Processors in service templates as a top-level 4638 entity. Alternatively, service template designers can also provide their own artifact processor by providing 4639 wrapper artifacts of a supported type. These wrapper artifacts could be shell scripts, python scripts, or 4640 artifacts of any other standard type that know how process or invoke the custom artifact.

4641 13.1.2 Establish an Execution Environment

- 4642 The second step is to identify or create a proper execution environment within which to run the artifact 4643 processor. There are generally three options for where to run artifact processors :
- 4644
- 46451.One option is to execute the artifact processor in the topology that is being orchestrated, for
example on a Compute node created by the orchestrator.
- 4647
 4648
 4648
 4648
 4649
 4649
 A second option is to process the artifact in the same environment in which the orchestrator is running (although for security reasons, orchestrators may create sandboxes that shield the orchestrator from faulty or malicious artifacts).
- 46503.The third option is to process the script in a management environment that is external to both the4651orchestrator and the topology being orchestrated. This might be the preferred option for scenarios4652where the environment already exists, but it is also possible for orchestrators to create external4653execution environments.
- It is often possible for the orchestrator to determine the intended execution environment based on the type of the artifact as well as on the topology context in which the the artifact was specified. For example, shell script artifacts associated with software components typically contain the install script that needs to be executed on the software component's host node in order to install that software component. However, other scripts may not need to be run inside the topology being orchestrated. For example, a script that creates a database on a database management system could run on the compute node that hosts the database management system, or it could run in the orchestrator environment and
- 4661 communicate with the DBMS across a management network connection.
- 4662
- 4663 Similarly, there may be multiple options for other types of artifacts as well. For example, puppet artifacts 4664 could get processed locally by a puppet agent running on a compute node in the topology, or they could 4665 get passed to a puppet master that is external to both the orchestrator and the topology.
- 4666

4667 Different orchestrators could make different decisions about the execution environments for various
4668 combinations of node types and artifact types. However, service template designers must have the ability
4669 to specify explicitly where artifacts are intended to be processed in those scneario where correct
4670 operation depends on using a specific execution environment.

4671 Need discussion on how this is done.

4672 13.1.3 Configure Artifact Processor User Account

4673 An artifact processor may need to run using a specific user account in the execution environment to 4674 ensure that the processor has the proper permissions to execute the required actions. Depending on the 4675 artifact, existing user accounts might get used, or the orchestrator might have to create a new user 4676 account specifically for the artifact processor. If new user accounts are needed, the orchestrator may also 4677 have to create a home directory for those users.

4678

4679 Depending on the security mechanisms used in the execution environment, it may also be necessary to 4680 add user accounts to specific groups, or to assign specific roles to the user account.

4681 13.1.4 Deploy Artifact Processor

4682 Once the orchestrator has identified the artifact processor as well as the execution environment, it must 4683 make sure that the artifact processor is deployed in the execution environment:

- If the orchestrator's own environment acts as the execution environment for the artifact
 processor, orchestrator implementors can make sure that a standard set of artifact processors is
 pre-installed in that environment, and nothing further may need to be done.
- When a Compute node in the orchestrated topology is selected as the execution environment,
 typically only the most basic processors (such as bash shells) are pre-installed on that compute
 node. All other execution processors need to be installed on that compute node by the
 orchestrator.
- When an external execution environment is specified, the orchestrator must at the very least be
 able to verify that the proper artifact processor is present in the external execution environment
 and generate an error if it isn't. Ideally, the orchestrator should be able to install the processor if
 necessary.
- 4695 The orchestrator may also take the necessary steps to make sure the processor is run as a specific user 4696 in the execution environment.

4697 13.1.5 Deploy Dependencies

The imperative logic contained in artifacts may in turn install or configure software components that are not part of the service topology, and as a result are opaque to the orchestrator. This means that the orchestrator cannot reflect these components in an instance model, which also means they cannot be managed by the orchestrator.

4702

4703 It is best practice to avoid this situation by explicitly modeling any dependent components that are
4704 required by an artifact processor. When deploying the artifact processor, the orchestrator can then deploy
4705 or configure these dependencies in the execution environment and reflect them in an instance model as
4706 appropriate.

- 4707
- 4708 For artifacts that require dependencies to to be installed, TOSCA provides a generic way in which to 4709 describe those dependencies, which will avoid the use of monolithic scripts.
- 4710
- 4711 Examples of dependent components include the following :
- Some executables may have dependencies on software libraries. For tools like Python, required libraries might be specified in a requirements.txt file and deployed into a virtual environment.
- Environment variables may need to be set.

- 4715
 Configuration files may need to be created with proper settings for the artifact processor. For example, configuration settings could include DNS names (or IP addresses) for contacting a Puppet Master or Chef Server.
- Artifact processors may require valid software licenses in order to run.
- 4719
 Other artifacts specified in the template may need to be deposited into the execution environment.

4721 **13.1.6 Identify Target**

- 4722 Orchestrators must pass information to the artifact processor that properly identifies the target for each4723 artifact being processed.
- In many cases, the target is the Compute node that acts as the host for the node being created or configured. If that Compute node also acts as the execution environment for the artifact processor, the target for the artifacts being processed is the Compute node itself. If that scenario, there is no need for the orchestrator to pass additional target information aside from specifying that all actions are intended to be applied locally.
- When artifact processors run externally to the topology being deployed, they must establish a connection across a management network to the target. In TOSCA, such targets are identified using Endpoint capabilities that contain the necessary addressing information. This addressing information must be passed to the artifact processor
- Note that in addition to endpoint information about the target, orchestrators may also need to pass
 information about the protocol that must be used to connect to the target. For example, some networking
 devices only accept CLI commands across a SSH connection, but others could also accept REST API
 calls. Different python scripts could be used to configure such devices: one that uses the CLI, and one
 that executes REST calls. The artifact must include metadata about which connection mechanism is
 intended to be used, and orchestrators must pass on this information to the artifact processor.
- Finally, artifact processor may need proper credentials to connect to target endpoints. Orchestrators mustpass those credentials to the artifact processor before the artifact can be processed.

4741 **13.1.7 Pass Inputs and Retrieve Results or Errors**

- Orchestrators must pass any required inputs to the artifact processor. Some processors could take inputs
 through environment variables, but others may prefer command line arguments. Named or positional
 command line arguments could be used. TOSCA must be very specific about the mechanism for passing
 input data to processors for each type of artifact.
- 4746
- Similarly, artifact processors must also pass results from operations back to orchestrators so that results
 values can be reflected as appropriate in node properties and attributes. If the operation fails, error codes
 may need to be returned as well. TOSCA must be very specific about the mechanism for returning results
 and error codes for each type of artifact.

4751 **13.1.8 Cleanup**

- 4752 After the artifact has been processed by the artifact processor, the orchestrator could perform optional 4753 cleanup:
- If an artifact processor was deployed within the topology that is being orchestrated, the
 orchestrator could decide to remove the artifact processor (and all its deployed dependencies)
 from the topology with the goal of not leaving behind any components that are not explicitly
 modeled in the service template.
- Alternatively, the orchestrator MAY be able to reflect the additional components/resources
 associated with the Artifact Processor as part of the instance model (post deployment).

- 4760 Artifact Processors that do not use the service template topology as their execution environment do not
- 4761 impact the deployed topology. It is up to each orchestrator implementation to decide if these artifact 4762 processors need to be removed.

4763 **13.2 Dynamic Artifacts**

4764 Detailed Artifacts may be generated on-the-fly as orchestration happens. May be 4765 propagated to other nodes in the topology. How do we describe those?

4766 13.3 Discussion of Examples

- This section shows how orchestrators might execute the steps listed above for a few common artifacttypes, in particular:
- 4769 1. Shell scripts
- 4770 2. Python scripts
- 4771 3. Package artifacts
- 4772 4. VM images
- 4773 5. Container images
- 4774 6. API artifacts
- 4775 7. Non-standard artifacts
- 4776 By illustrating how different types of artifacts are intended to be processed, we identify the information 4777 needed by artifact processors to properly process the artifacts, and we will also identify the components
- 4778 in the topology from which this information is indended to be obtained.

4779 **13.3.1 Shell Scripts**

- 4780 Many artifacts are simple bash scripts that provide implementations for operations in a Node's Lifecycle
 4781 Interfaces. Bash scripts are typically intended to be executed on Compute nodes that host the node with
- 4782 which these scripts are associated.
- 4783
- We use the following example to illustrate the steps taken by TOSCA orchestrators to process shell scriptartifacts.
- 4786

4787

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: Sample tosca archive to illustrate simple shell script usage.
template_name: tosca-samples-shell
template_version: 1.0.0-SNAPSHOT
template_author: TOSCA TC
```

node_types:

```
tosca.nodes.samples.LogIp:
```

```
derived_from: tosca.nodes.SoftwareComponent
```

```
description: Simple linux cross platform create script.
```

attributes:

```
log_attr: { get_operation_output: [SELF, Standard, create, LOG_OUT] }
interfaces:
```

```
Standard:
        create:
          inputs:
            SELF_IP: { get_attribute: [HOST, ip_address] }
          implementation: scripts/create.sh
topology_template:
  node_templates:
    log ip:
      type: tosca.nodes.samples.LogIp
      requirements:
        - host:
            node: compute
            capability: tosca.capabilities.Container
            relationship: tosca.relationships.HostedOn
    # Any linux compute.
    compute:
      type: tosca.nodes.Compute
      capabilities:
        os:
          properties:
            type: linux
```

4789 This example uses the following script to install the LogIP software :

4790

```
#!/bin/bash
```

```
# This is exported so available to fetch as output using the get_operation_output
function
export LOG_OUT="Create script : $SELF_IP"
# Just a simple example of create operation, of course software installation is
better
echo "$LOG_OUT" >> /tmp/tosca_create.log
```

4791
4792
4793 For this simple example, the artifact processing steps outlined above are as follows:
4794
4795

Identify Artifact Processor: The artifact processor for bash shell scripts is the "bash" program.

4796

Establish Execution Environment: The typical execution environment for bash scripts is the 4797

- 4798
 4798
 4799
 4799
 4790
 4800
 4800
 Configure User Account: The bash user account is the default user account created when instantiating the Compute node. It is assumed that this account has been configured with sudo privileges.
- 4801
 4802
 4802
 4802
 4803
 4804
 4804
 4804
 4805
 4805
 4805
 4806
 4806
 4806
 4807
 4807
 4807
 4808
 4808
 4808
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 4809
 <li
- 4803 5. *Deploy Dependencies*: Orchestrators should copy all provided artifacts using a directory
 4804 structure that mimics the directory structure in the original CSAR file containing the artifacts.
 4805 Since no dependencies are specified in the example above, nothing further needs to be done.
- 4806 6. *Identify Target*: The target for bash is the Compute node itself.
- Pass Inputs and Retrieve Outputs: Inputs are passed to bash as environment variables. In the
 example above, there is a single input declared for the create operation called SELF_IP. Before
 processing the script, the Orchestrator creates a corresponding environment variable in the
 execution environment. Similarly, the script creates a single output that is passed back to the
 orchestrator as an environment variable. This environment variable can be accessed elsewhere
 in the service template using the get_operation_output function.
- 4813 13.3.1.1 Progression of Examples
- 4814 The following examples show a number of potential use case variations (not exhaustive) :
- 4815

4816 13.3.1.1.1 Simple install script that can run on all flavors for Unix.

For example, a Bash script called "create.sh" that is used to install some software for a TOSCA Node;
that this introduces imperative logic points (all scripts perhaps) which MAY lead to the creation of "opaque
software" or topologies within the node

- 4820
- 4821

4834

4835

4836

4837

4838

4839

4822 13.3.1.1.1.1 Notes

- Initial examples used would be independent of the specific flavor of Linux.
- 4824 The "create" operation, as part of the normative Standard node lifecycle, has special meaning in 4825 TOSCA in relation to a corresponding deployment artifact; that is, the node is not longer 4826 "abstract" if it either has an impl. Artifact on the create operation or a deployment artifact 4827 (provided on the node).
- 4828 "create.sh" prepares/configures environment/host/container for other software (see below for VM image4829 use case variants).
- 4830 13.3.1.1.1.2 Variants
- 4831 1. "create.sh" followed by a "configure.sh" (or "stop.sh", "start.sh" or a similar variant).
- 4832 2. in Compute node (i.e., within a widely-used, normative, abstract Node Type).
- 4833 3. In non-compute node like WebServer (is this the hello world)?
 - Container vs. Containee "hello worlds"; create is "special"; speaks to where (target) the script is run at! i.e., Compute node does not have a host.
 - What is BEST PRACTICE for compute? Should "create.sh" even be allowed?
 - Luc: customer wanted to use an non-AWS cloud, used shell scripts to cloud API.
 - i. Should have specific Node type subclass for Compute for that other Cloud (OR) a capability that represents that specific target Cloud.

4840 13.3.1.1.2 Script that needs to be run as specific user

4841 For example, a Postgres user

4842 13.3.1.1.3 Simple script with dependencies

- 4843 For example, using example from the meeting where script depends on AWS CLI being installed.
- 4844
- How do you decide whether to install an RPM or python package for the AWS dependency?
- How do we decide whether to install python packages in virtualenv vs. system-wide?

4847 13.3.1.1.4 Different scripts for different Linux flavors

- 4848 For example. run apt-get vs. yum
- The same operation can be implemented by different artifacts depending on the flavor of Linux on which the script needs to be run. We need the ability to specify which artifacts to use based on the target.
- 4852
 How do we extend the "operation" grammar to allow for the selection of one specific artifact out of a number of options?
- 4854 How do we annotate the artifacts to indicate that they require a specific flavor and/or version of
 4855 Linux?

4856 13.3.1.1.4.1 Variants

- 4857
 A variant would be to use different subclasses of abstract nodes, one for each flavor of Linux on 4858
 4859
 4860
 4860
 A variant would be to use different subclasses of abstract nodes, one for each flavor of Linux on 4859
 4860
- 4861 **13.3.1.1.5 Scripts with environment variables**
- Environment variables that may not correspond to input parameters
- 4863 For example, OpenStack-specific environment variables
- How do we specify that these environment variables need to be set?

4865 **13.3.1.1.6 Scripts that require certain configuration files**

- 4866 For example, containing AWS credentials
- 4867 This configuration file may need to be created dynamically (rather than statically inside a CSAR file). How do we specify that these files may need to be created?
- Or does this require template files (e.g. Jinja2)?

4870 **13.3.2 Python Scripts**

4871 A second important class of artifacts are Python scripts. Unlike Bash script artifacts, Python scripts are
4872 more commonly executed within the context of the Orchestrator, but service template designers must also
4873 be able to provide Python scripts artifacts that are intened to be excecuted within the topology being
4874 orchestrated,

4875 13.3.2.1 Python Scripts Executed in Orchestrator

4876

Need a simple example of a Python script executed in the Orchestrator context.

13.3.2.2 Python Scripts Executed in Topology 4877

- 4878 Need a simple example of a Python script executed in the topology being orchestrated.
- 4879

- 4880 The following grammar is provided to allow service providers to specify the execution environment within 4881 which the artifact is intended to be processed :
- 4882 Need to decide on grammar. Likely an additional keyword to the "operation" section of 4883 lifecycle interface definitions.

13.3.2.3 Specifying Python Version 4884

- 4885 Some python scripts conform to Python version 2, whereas others may require version 3. Artifact designers use the following grammar to specify the required version of Python: 4886
- 4887
- 4888 TODO

4889 13.3.2.3.1.1 **Assumptions/Questions**

4890 Need to decide on grammar. Is artifact processer version associated with the processor, with the • 4891 artifact, the artifact type, or the operation implementation?

4892 13.3.2.4 Deploying Dependencies

4893 Most Python scripts rely on external packages that must be installed in the execution environment. Typically, python packages are installed using the 'pip' command. To provide isolation between different 4894 4895 environments, is is considered best practice to create virtual environments. A virtual environment is a tool to keep the dependencies required by different python scripts or projects in separate places, by creating 4896 4897 virtual Python environments for each of them.

- 4898
- 4899 The following example shows a Python script that has dependencies on a number of external packages:
- 4900 TODO
- 4901

4902 13.3.2.4.1.1 Assumptions/Questions

- 4903 Python scripts often have dependencies on a number of external packages (that are referenced 4904 by some package artifcat). How would these be handled?
- 4905 How do we account for the fact that most python packages are available as Linux packages as • 4906 well as pip packages?
- 4907 Does the template designer need to specify the use of virtual environments, or is this up to the 4908 orchestrator implementation? Must names be provided for virtual environments?

13.3.2.4.1.2 4909 Notes

4910 Typically, dependent artifacts must be processed in a specific order. TOSCA grammar must 4911 provide a way to define orders and groups (perhaps by extending groups grammar by allowing 4912 indented sub-lists).

4913 13.3.3 Package Artifacts

4914 Most software components are distributed as software packages that include an archive of files and 4915 information about the software, such as its name, the specific version and a description. These packages 4916 are processed by a package management system (PMS), such as rpm or YUM, that automates the

4917 software installation process.

- 4918
- 4919 Linux packages are maintained in Software Repositories, databases of available application installation
- 4920 packages and upgrade packages for a number of Linux distributions. Linux installations come pre-
- 4921 configured with a default Repository from which additional software components can be installed.
- 4922

4923 While it is possible to install software packages using Bash script artifacts that invoke the appropriate 4924 package installation commands (e.g. using apt or yum), TOSCA provides improved portability by allowing 4925 template designers to specify software package artifacts and leaving it up to the orchestrator to invoke the 4926 appropriate package management system.

4927 13.3.3.1 RPM Packages

- 4928 The following example shows a software component with an RPM package artifact.
- 4929 Need a simple example

4930 13.3.4 Debian Packages

- 4931 The following example shows a software component with Debian package artifact.
- 4932

4933 Need a simple example

- 4934 13.3.4.1.1.1 Notes
- In this scenario, the host on which the software component is deployed must support RPM
 packages. This must be reflected in the software component's host requirement for a target
 container.
- In this scenario, the host on which the software component is deployed must support Debian
 packages. This must be reflected in the software component's host requirement for a target
 container.

4941 13.3.4.2 Distro-Independent Service Templates

Some template designers may want to specify a generic application software topology that can be
deployed on a variety of Linux distributions. Such templates may include software components that
include multiple package artifacts, one for each of the supported types of container platforms. It is up to
the orchestrator to pick the appropriate package depending on the type of container chosed at
deployment time.

- 4947
- 4948 Supporting this use case requires the following:
- Allow multiple artifacts to be expressed for a given lifecycle operation.
- Associate the required target platform for which each of those artfiacts was meant.

4951 13.3.4.2.1.1 Assumptions/Questions

- 4952 How do we specify multiple artifacts for the same operation?
- 4953How we we specify which platforms are support for each artifact? In the artifact itself? In4954the artifact type?

4955 **13.3.5 VM Images**

4956 **13.3.5.1.1.1 Premises**

4957
 VM Images is a popular opaque deployment artifact that may deploy an entire topology that is not declared itself within the service template.

4959 **13.3.5.1.1.2** Notes

4960 • The "create" operation, as part of the normative Standard node lifecycle, has special meaning in 4961 TOSCA in relation to a corresponding deployment artifact; that is, the node is not longer 4962 "abstract" if it either has an impl. Artifact on the create operation or a deployment artifact 4963 (provided on the node).

4964 13.3.5.1.1.3 Assumptions/Questions

- In the future, the image itself could contain TOSCA topological information either in its metadata or externally as an associated file.
 Can these embedded or external descriptions be brought into the TOSCA Service Template
- 4967•Can these embedded or external descriptions be brought into the TOSCA Service Template4968or be reflected in an instance model for management purposes?
- Consider create.sh in conjunction with a VM image deployment artifact
 - VM image only (see below)
 - Create.sh and VM image, both. (Need to address argument that they belong in different nodes).
 - Configure.sh with a VM image.? (see below)
 - Create.sh only (no VM image)

4975 • Implementation Artifact (on TOSCA Operations):

- Operations that have an artifact (implementation).
- 4977 Deployment Artifacts:

4970

4971

4972

4973

4974

4976

4978

4979

4980

4981

4982

4983

4984

4985

4986

4987

- Today: it must appear in the node under "artifacts" key (grammar)
 - In the Future, should it:
 - Appear directly in "create" operation, distinguish by "type" (which indicates processor)?
 - <or> by artifact name (by reference) to artifact declared in service template.
 - What happens if on create and in node (same artifact=ok? Different=what happens? Error?)
 - What is best practice? And why? Which way is clearer (to user?)?
 - Processing order (use case variant) if config file and VM image appear on same node?

4988 **13.3.6 Container Images**

4989 **13.3.7 API Artifacts**

4990 Some implementations may need to be implemented by invoking an API on a remote endpoint. While 4991 such implementations could be provided by shell or python scripts that invoke API client software or use 4992 language-specific bindings for the API, it might be preferred to use generic API artifacts that leave 4993 decisions about the tools and/or language bindings to invoke the API to the orchestrator.

- 4994 To support generic API artifacts, the following is required:
- A format in which to express the target endpoint and the required parameters for the API call
- 4996 A mechanism for binding input parameters in the operation to the appropriate parameters in the
 4997 API call.

- A mechanism for specifying the results and/or errors that will be returned by the API call
- 4999 Moreover, some operations may need to be implemented by making more than one API call. Flexible API 5000 support requires a mechanism for expressing the control logic that runs those API calls.

5001 It should be possible to use a generic interface to describe these various API attributes without being 5002 forced into using specific software packages or API tooling. Of course, in order to "invoke" the API an 5003 orchestrator must launch an API client (e.g. a python script, a Java program, etc.) that uses the appropriate API language bindings. However, using generic API Artifact types, the decision about which 5004 API clients and language bindings to use can be left to the orchestrator. It is up to the API Artifact 5005 5006 Processor provided by the Orchestrator to create an execution environment within which to deploy API 5007 language bindings and associated API clients based on Orchestrator preferences. The API Artifact 5008 Processor then uses these API clients to "process" the API artifact.

5009 13.3.7.1 Examples

- 5010 REST
- 5011 SOAP
- 5012 OpenAPI
- 5013 IoT
- 5014 Serverless

5015 13.3.8 Non-Standard Artifacts with Execution Wrappers

5016 TODO

5017 13.4 Artifact Types and Metadata

- 5018 To unambiguously describe how artifacts need to be processed, TOSCA provides two things:
- 5019 1. Artifact types that define standard ways to process artifacts.
- 5020 2. Descriptive metadata that provide information needed to properly process the artifact.

5021 14Abstract nodes and target node filters matching

5022 This section details the matching or orchestrator's node selection mechanisms that is mentioned and 5023 explained from user point of view in section 2.9 of the specification.

5024

5025 When a user define a service template some of the nodes within the service templates are not 5026 implemented (abstract) and some requirements may define some node filters target rather than actual 5027 abstract node templates. In order to deploy such service templates the orchestrator has to find a valid 5028 fulfillement and implementation available on the deployment target in order to be able to actually 5029 instantiate the various elements of the template.

5030

5031 The goal of this **non-normative** chapter is to give an non-exclusive insight on orchestrator possible 5032 behavior to provide fullfillement to abstract nodes and dangling requirements within a TOSCA template.

503314.1 Reminder on types

5034

5037

5038

5039

5040

5041

5035 TOSCA allows the definition of types that can later be used within templates. Types can be of two nature 5036 on regard of the matching process:

- Abstract types that have no implementation specified and that can be used within a Topology template in order to request the orchestrator to find a valid implementation (for example an abstract tosca.nodes.Compute type can be used to define a template to request a VM from an orchestrator without any specific knowledge on the implementation, allowing that way portability).
 - **Concrete types** that are implemented through TOSCA implementation artifacts (shell scripts, python scripts etc.) or through the mean of a Topology subtitution.
- 5042 5043

5044 Both abstract and concrete types defines properties (and capabilities properties) that can be used for two 5045 different means:

- **Configuration** of the node and of it's behavior (most likely used in concrete types).
 - Matching purpose (most likely used for abstract types).
- 5047 5048

5046

5049 This section will focus on the matching process while configuration properties is mostly related to types 5050 design.

5051 14.2 Orchestrator catalogs

5052Most of orchestrators are likely to have internal catalogs of TOSCA types, pre-defined templates, internal5053implementation of nodes (either through concrete types, substitution mechanisms, potentially supported5054by non-normative workflow definitions etc.) and maybe even running instances (services).

5055

5056 Theses catalogs are not normative and it is up to the TOSCA implementation to support some or all of 5057 them. During matching the TOSCA orchestrator may find a valid match for a template within any of it's 5058 internal catalogs or through any other mean.

5059

5064

5060 This section will consider and provide examples based on the three following catalogs (thay may or may 5061 not be used in actual implementations):

- **Type catalog:** Basic internal catalog but not the most intuitive candidate for node matching. It contains:
 - abstract node types

 TOSCA-Simple-Profile-YAML-v1.2-os

 Standards Track Work Product

 Copyright © OASIS Open 2019. All Rights Reserved.

5065	 concrete node types implemented through implementation artifacts.
5066	 concrete node types implemented through topology substitution.
5067	• Pre-defined node template catalog: This is the catalog that is the most likely to be used for
5068	matching, it may contains:
5069	 Orchestrator Provider pre-defined node templates offered to it's user eventually backed
5070	up with orchestrator specific implementations (that may delegate to non-tosca internal
5071	components).
5072	 User defined node templates implemented through implementation artifacts.
5073	 User defined node templates implemented through topology substitution.
5074	Running instance/Services catalog: Catalog of already running services available for matching
5075	that contains some definition of TOSCA instances.
5076	14.3 Abstract node template matching
5077	A TOSCA topology template as defined by a user will probably define some abstract node templates. A
5078	node template is considered abstract if it is based on an abstract type and does not provides
5079	implementation at the template level. As instanciating an abstract node can not be done by an
5080	orchestrator, the orchestrator will have to perform internally the replacement of the defined abstract node
5081	template's types by a matching implementation of the type.
5082	
5083	A type is considered as a valid matching implementation if it fullfills all of the following conditions:
5084	 The matching node derives from the type specified in the template
5085	• Every property defined in the matching node is matching the constraint specified on the node
5086	template's properties or capability properties given the following rules:
5087	• A property that is defined in the node template (either through a value at the template
5088	level or through a default property value at the type level) should be match the constraint
5089	defined on the matching node type property.
5090	 A property that is not defined in the node template may have no or any value (matching
5091	the node type property definition constraints) in the orchestrator matched node.
5092	
5093 5094	A pre-defined template is considered as a valid matching implementation if it fullfills all of the following conditions:
5095	The orchestrator pre-defined matching node derives from the type specified in the topology
5096	template's node
5097	• Every property defined in the orchestrator pre-defined matching node is matching the constraint
5098	specified on the node template's properties or capability properties given the following rules:
5099	• A property that is defined in the node template (either through a value at the template
5100	level or through a default property value at the type level) should be matched by an
5101	equality constraint
5102	• A property that is not defined in the node template may have no or any value (matching
5103	the node type property definition constraints) in the orchestrator matched node.
5104	
5105	A running instance (service) is considered as a valid matching implementation if it fullfills all of the
5106	following conditions:
5107	 The node instance has a type that equals or derives from the type specified in the topology
5108	template's node
5109	Every attribute defined in the orchestrator instance node is matching the constraint specified on
5110	the node template's properties or capability properties given the following rules:

- 5111oA property that is defined in the node template (either through a value at the template5112level or through a default property value at the type level) should be matched by an5113equality constraint against the attribute value.
- 5114 o A property that is not defined in the node template may have no or any value (matching 5115 the node type property definition constraints) in instance node.
- 5116 Note that the node instance that defines the running instance/service can be actually a full topology that 5117 propose a node abstraction through the topology substitution mechanism.
- 5118

5119 **Multiple valid matches:** If the orchestrator has more than one valid match in it's catalog(s) he is

- responsible for either choosing automatically a node or providing a mean for users to specify the nodethey want to select.
- 5122

5123 **No match:** If the orchestrator does not find any valid match he could propose alternative that he consider 5124 valid but should not automatically deploy the topology without an explicit user approval.

5125

5126 Note: Theses rules are the basic matching rules of TOSCA, however if an orchestrator has a UI and want 5127 to propose other matching nodes that does not fullfill all of these constraints he can still do that even if he 5128 should warn the user that the deployed template will not be the same template as defined. For example 5129 an orchestrator could propose a node with greater than CPU rather than an equal match, or propose an 5130 equivalent node (with different type) that has the same capabilities as the ones connected by the node in 5131 the topology.

5132

5133 Note: Support of instances matching may impact the TOSCA workflow and lifecycle as their operations 5134 will not be included in the workflow (instances are already created).

5135

5136 **14.3.1 Examples**

Let's consider a few examples of abstract node templates and how they can be matched against an
orchestrator catalog(s). Note that the type catalog is not the only catalog in which to find implementation.
Most orchestrator will probably have an internal provider templates catalog that includes pre-defined
templates. None of the catalog is required to be a valid TOSCA implementation and the following are just

- 5141 examples for orchestrator implementers but is not required to be implemented.
- 5142 14.3.1.1 Matching from a type catalog
- 5143 Let's consider the following node types in an orchestrator internal type catalog.
- 5144

tosca_definitions_version: tosca_simple_yaml_1_0

node_types: tosca.samples.nodes.MyAbstractNode:

derived from: tosca.nodes.Root

properties:

str_prop:

type: string

nbr_prop:

type: integer

5145 MyAbstractNode is an abstract type as Root does not define any implementation and the defined node

neither. 5146

5147

5148

5149

node_types:
tosca.samples.nodes.MyNodeImpl1:
derived_from: tosca.samples.nodes.MyAbstractNode
properties:
nbr_prop :
constraints:
- greater_or_equal: 1
interfaces:
standard:
create: test.sh
lyNodeImpl1 is an implementation (through the test.sh script) of MyAbstractNode that requires the br_prop property to be higher than 1.
<pre>tosca_definitions_version: tosca_simple_yaml_1_0</pre>
node_types:
tosca.samples.nodes.MyNodeImpl2:
derived_from: tosca.samples.nodes.MyAbstractNode

5150

```
properties:
nbr_prop :
 constraints:
   - greater_or_equal: 25
interfaces:
 standard:
  create: test2.sh
```

5151 MyNodeImpl2 is an implementation (through the test2.sh script) of MyAbstractNode that requires the 5152 nbr_prop property to be higher than 25.

5153

- 5154 Let's consider the following topology template that a user want to deploy:
- 5155

tosca_definitions_version: tosca_simple_yaml_1_0

topology_template: node templates:

my node:

type: tosca.samples.MyAbstractNode

properties:

str_prop: standard nbr_prop: 10

5156 The specified node template (my_node) is an abstract node template as it's type is abstract and it does 5157 not add any implementation. Before being able to deploy this template a TOSCA orchestrator will have to 5158 find a valid match for this node. In order to do so it will look into it's catalog (in this example the type 5159 catalog) and try to find nodes that matches the definition.

5160 In this example while both MyNodeImpl1 and MyNodeImpl2 have a valid type as they derive from 5161 MyAbstractNode only MyNodeImpl1 is a valid match as the constraint defined on the nbr_prop property of 5162 the MyNodeImpl2 node type (greater_or_equal: 25) is not matching the property value defined in the 5163 requested node template (10).

5164

5165 14.3.1.2 Matching from a pre-defined template catalog

5166 This example details how a tosca.nodes.Compute abstract node can be matched to a specific pre-defined 5167 template that an orchestrator may have. First of all the orchestrator will probably define a concrete

- 5168 implementation of the Compute node. So let's consider the following example type
- 5169

tosca_definitions_version: tosca_simple_yaml_1_0

node_types:

tosca.samples.nodes.MyCloudCompute:

derived_from: tosca.nodes.Compute

properties: image_id: type: string required: true flavor_id: type: string required: true

interfaces: standard:

create: create.py

- 5170 This type add two properties to the Compute node so the orchestrator knows which image_id and 5171 flavor_id are used to instanciate the VM. Implementation is simplified here and just a single python script
- 5172 is enough.

5173

5174 Note: an orchestrator provider can define internally some non-portable implementations of types that will 5175 be supported only by the latter. As the user defines an abstract node it's template is portable even if the 5176 execution is specific to the orchestrator.

5177

5178 Let's now consider that the orchestrator has defined some internal node template in it's own pre-defined 5179 templates or provider catalog (note that this is orchestrator specific and this specification has no intent on

- 5180 defining how the orchestrator should manage, import or support it's internal catalogs).
- 5181

```
tosca_definitions_version: tosca_simple_yaml_1_0
node templates:
  small_ubuntu:
   type: tosca.samples.nodes.MyCloudCompute
   properties:
    image id: ubuntu
    flavor_id: small
   capabilities:
    host:
           num_cpus: 1
           cpu_frequency: 1 GHz
           disk size: 15 GiB
           mem_size: 2 GiB
    os:
      type: linux
      distribution: ubuntu
  large ubuntu:
   type: tosca.samples.nodes.MyCloudCompute
   properties:
    image_id: ubuntu
    flavor_id: small
   capabilities:
    host:
           num cpus: 4
           cpu_frequency: 2 GHz
           disk_size: 15 GiB
           mem_size: 8 GiB
    OS:
      type: linux
      distribution: ubuntu
  large_windows:
   type: tosca.samples.nodes.MyCloudCompute
   properties:
    image_id: ubuntu
    flavor_id: small
   capabilities:
    host:
           num_cpus: 4
           cpu_frequency: 2 GHz
           disk_size: 15 GiB
```

mem_size: 8 GiB

os: type: windows distribution: server

5182

5183 If a user defines the following template:

5184

5207

tosca_definitions_version: tosca_simple_yaml_1_0

topology_template: node_templates: my_node: type: tosca.nodes.Compute capabilities: host: num_cpus: 1 mem_size: 2 GiB os:

distribution: Ubuntu

5185 The orchestrator will select the small_ubuntu pre-defined template as a valid match. The image_id and 5186 flavor_id properties are internal to the orchestrator.

5187 **14.4 Target node filter matching**

5188 In addition to matching abstract nodes, an orchestrator also has to find matches for dangling 5189 requirements. Target node filter (also reffered as dangling requirements) matching provides loose coupling as you may specify a request on any node that provides a capability rather than a specific node. 5190 5191 5192 A dangling requirement is defined on the requirement section of a node template, it instruct the 5193 orchestrator how to find a valid node template to add and connect in the topology. The node added by the 5194 orchestrator as a relationship target is matched based on the following rules. 5195 5196 A type is considered as a valid matching implementation if it fullfills all of the following conditions: 5197 The selected node must define a capability with the same type as specified by the dangling requirement or with a type that derive from the specified type. 5198 5199 If the *node* property is specified on the dangling requirement, then the type of the matched node • 5200 must derive from the requested type 5201 The node filter constraints defined on the dangling requirement are compatible with the candidate • 5202 node type properties constraints and default values. 5203 5204 A pre-defined template is considered as a valid matching implementation if it fullfills all of the following 5205 conditions: 5206 The orchestrator pre-defined node defines a capability with the same type as specified by the

dangling requirement or with a type that derive from the specified type.

5208 If the *node* property is specified on the dangling requirement, then the type of the orchestrator 5209 pre-defined node must derive from the requested type 5210 The node filter constraints defined on the dangling requirement are matched by the pre-defined • 5211 template properties values. 5212 5213 A running instance (service) is considered as a valid matching implementation if it fullfills all of the 5214 following conditions: 5215 The orchestrator pre-defined node defines a capability with the same type as specified by the 5216 dangling requirement or with a type that derive from the specified type. If the node property is specified on the dangling requirement, then the type of the node instance 5217 • 5218 must derive from the requested type 5219 • The node filter constraints defined on the dangling requirement are matched by the node instance current attribute values 5220

5221

5222 A property that is not defined in the node template may have no or any value (matching the node type 5223 property definition constraints) in instance node.

5224 **14.4.1 Examples**

5225 14.4.1.1 Matching a node filter target against a type catalog

- 5226 Let's consider the following nodes in a type catalog:
- 5227

```
tosca definitions version: tosca simple yaml 1 0
capability_types:
  tosca.samples.capabilities.MyMessagingEndpoint :
    derived_from: tosca.capabilities.Endpoint
    properties:
      throughput :
        type: integer
        required: true
  tosca.samples.capabilities.MyLimitedMessagingEndpoint :
    derived from: tosca.samples.capabilities.MyMessagingEndpoint
    properties:
      throughput :
        type: integer
        required: true
        constraints:
          - lower than: 5
node types:
  tosca.samples.nodes.MyNode :
```

```
derived_from: tosca.nodes.Root
  requirements: tosca.samples.capabilities.MyMessagingEndpoint
 interfaces:
    standard:
      create: install.sh
tosca.samples.nodes.MyAbstractMessagingSystem:
 derived_from: tosca.nodes.Root
 properties:
    scaling:
      type: string
      required: true
      constraints:
        - valid_values: [ "auto", "manual", "none" ]
    highly available :
      type: boolean
      required: true
  capabilities:
    messaging: tosca.samples.capabilities.MyMessagingEndpoint
tosca.samples.nodes.MyMessagingServiceSystem:
 derived_from: tosca.samples.nodes.MyAbstractMessagingSystem
 properties:
    scaling :
      type: string
      required: true
      constraints:
        - valid_values: [ "manual"]
    highly_available:
      constraints:
        - equal: true
 interfaces:
    standard:
      create: create.py
tosca.samples.nodes.MyMessagingSystem:
  derived_from: tosca.samples.nodes.MyAbstractMessagingSystem
 properties:
    scaling :
      type: string
      required: true
      constraints:
        - valid_values: [ "none"]
    highly_available:
```

```
constraints:
        - equal: false
capabilities:
    messaging: tosca.samples.capabilities.MyLimitedMessagingEndpoint
interfaces:
    standard:
    create: install.sh
    start: start.sh
```

5229 And the following user template to deploy:

5230

```
tosca definitions version: tosca simple yaml 1 0
topology_template:
  node_templates:
    my node:
      type: tosca.samples.nodes.MyNode
      requirements:
        - messaging:
            node: tosca.samples.nodes.MyAbstractMessagingSystem
            node filter:
              properties:
                - scaling: { valid values: [manual, auto] }
                - highly available: { equal: true }
              capabilities:
                - tosca.samples.capabilities.MyMessagingEndpoint:
                    properties:
                      - throughput: { greater_than: 10 }
```

In order to fulfill the messaging endpoint target the orchestrator will have to add a node template from a
type that derives from MyAbstractMessagingSystem (as specified within the node filter node property)
and that defines constraints that are compatible with the ones specified on the node filter.

In the defined type catalog the only type that fulfill all constraints is the MyMessagingServiceSystem node.

5236

5237 14.4.1.2 Matching a node filter target against a type catalog with substitution

5238 TOSCA allows the definition of a type implementation through a substitution template. In this case the 5239 specified topology templates becomes a type in the catalog. From this type an orchestrator may define 5240 some pre-defined templates or even running services if instanciated. In the following example we will 5241 consider the same user template as in the previous example as well as the same abstract types. However 5242 the implemented type will be defined through the following topology template:

5243

```
tosca_definitions_version: tosca_simple_yaml_1_0
topology_template:
  inputs:
    # Nodes in this topology can be configured to enable auto-scaling or not
    scaling_input :
        type: string
        required: true
        constraints:
          - valid_values: [ "auto", "none" ]
  substitution_mappings:
    node_type: tosca.samples.nodes.MyAbstractMessagingSystem
    properties:
      scaling: [ scaling input ]
      highly available: true
    capabilities:
      messaging : [ my_load_balancer, load_balanced_messaging_endpoint]
  node_templates:
    my_load_balancer:
      type: tosca.samples.nodes.MyLoadBalancer
      capability:
        load balanced_messaging_endpoint:
tosca.samples.capabilities.MyMessagingEndpoint
    my_other_node_that_trigger_a_service_somewhere:
      type: org.custom.Type
      properties:
        my_scaling_info: get_input { scaling }
    my other node:
      type: org.something.Type:
      properties:
        my_other_scaled_prop: get_input { scaling }
        another_prop: value
... other nodes templates
```

5245

5246 This template from a substitution boundaries point of view would be equivalent to the following node type: 5247

tosca_definitions_version: tosca_simple_yaml_1_0

```
node_type:
my_node_resulting_from_topology
# From topology_template -> substitution_mappings -> node_type
derived_from: tosca.samples.nodes.MyAbstractMessagingSystem
properties:
    scaling :
        constraints:
            - valid_values: [ "auto", "none" ]
    highly_available:
        default: true
        constraints:
            - equal: true
    # Equivalent:
    # implementation: The topology specified above
```

5249 In this example the orchestrator can select the topology template specified above as a valid match for the 5250 requested target node filter.

5251 14.5 Post matching properties

It is possible that, even after matching, some properties have unset values, moreover some properties
may be added by the type that is selected by the orchestrator and derives from the user requested type.
In any case an orchestrator should not deploy a node that has some required properties undefined.

Based on the orchestrator capabilities it could be possible to assign values to the properties (either
required or not required) of the node after the matching, including properties added by the selected
implementation node. Note that theses capabilities are not mandatory and that as properties depends
from the actual result of the matching it is not possible to ship them with the template. Therefore there is
no standard for defining theses additional properties and the mean of providing them will be specific to
the orchestrator implementation.

5261 **15Conformance**

5262 15.1 Conformance Targets

- 5263 The implementations subject to conformance are those introduced in Section 11.3 "Implementations". 5264 They are listed here for convenience:
- 5265 TOSCA YAML service template
- 5266 TOSCA processor
- TOSCA orchestrator (also called orchestration engine)
- TOSCA generator
- 5269 TOSCA archive

5270 15.2 Conformance Clause 1: TOSCA YAML service template

- 5271 A document conforms to this specification as TOSCA YAML service template if it satisfies all the 5272 statements below:
- (a) It is valid according to the grammar, rules and requirements defined in section 3 "TOSCA Simple
 Profile definitions in YAML".
- 5275 (b) When using functions defined in section 4 "TOSCA functions", it is valid according to the grammar 5276 specified for these functions.
- (c) When using or referring to data types, artifact types, capability types, interface types, node types,
 relationship types, group types, policy types defined in section 5 "TOSCA normative type
 definitions", it is valid according to the definitions given in section 5.

5280 15.3 Conformance Clause 2: TOSCA processor

5281 A processor or program conforms to this specification as TOSCA processor if it satisfies all the 5282 statements below:

- (a) It can parse and recognize the elements of any conforming TOSCA YAML service template, and
 generates errors for those documents that fail to conform as TOSCA YAML service template
 while clearly intending to.
- 5286(b) It implements the requirements and semantics associated with the definitions and grammar in5287section 3 "TOSCA Simple Profile definitions in YAML", including those listed in the "additional5288requirements" subsections.
 - (c) It resolves the imports, either explicit or implicit, as described in section 3 "TOSCA Simple Profile definitions in YAML".
- 5291(d) It generates errors as required in error cases described in sections 3.1 (TOSCA Namespace URI5292and alias), 3.2 (Parameter and property type) and 3.6 (Type-specific definitions).
- 5293 (e) It normalizes string values as described in section 5.4.9.3 (Additional Requirements)
- 5294

5289 5290

5295 **15.4 Conformance Clause 3: TOSCA orchestrator**

5296 A processor or program conforms to this specification as TOSCA orchestrator if it satisfies all the 5297 statements below:

- 5298 (a) It is conforming as a TOSCA Processor as defined in conformance clause 2: TOSCA Processor.
- 5299 (b) It can process all types of artifact described in section 5.3 "Artifact types" according to the rules 5300 and grammars in this section.
- (c) It can process TOSCA archives as intended in section 6 "TOSCA Cloud Service Archive (CSAR)
 format" and other related normative sections.

- 5303(d) It can understand and process the functions defined in section 4 "TOSCA functions" according to
their rules and semantics.
- (e) It can understand and process the normative type definitions according to their semantics and
 requirements as described in section 5 "TOSCA normative type definitions".
- 5307 (f) It can understand and process the networking types and semantics defined in section 7 "TOSCA 5308 Networking".
- 5309(g) It generates errors as required in error cases described in sections 2.10 (Using node template
substitution for chaining subsystems), 5.4 (Capabilities Types) and 5.7 (Interface Types).).

5311 15.5 Conformance Clause 4: TOSCA generator

- 5312 A processor or program conforms to this specification as TOSCA generator if it satisfies at least one of 5313 the statements below:
- (a) When requested to generate a TOSCA service template, it always produces a conforming
 TOSCA service template, as defined in Clause 1: TOSCA YAML service template,
- (b) When requested to generate a TOSCA archive, it always produces a conforming TOSCA archive,as defined in Clause 5: TOSCA archive.

5318 **15.6 Conformance Clause 5: TOSCA archive**

- 5319 A package artifact conforms to this specification as TOSCA archive if it satisfies all the statements below:
- (a) It is valid according to the structure and rules defined in section 6 "TOSCA Cloud Service Archive
 (CSAR) format".

5322 Appendix A. Known Extensions to TOSCA v1.0

5323 The following items will need to be reflected in the TOSCA (XML) specification to allow for isomorphic 5324 mapping between the XML and YAML service templates.

5325 A.1 Model Changes

5330

5331

5341

- The "TOSCA Simple 'Hello World'" example introduces this concept in Section 2. Specifically, a VM image assumed to accessible by the cloud provider.
- Introduce template Input and Output parameters
- The "Template with input and output parameter" example introduces concept in Section 2.1.1.
 - "Inputs" could be mapped to BoundaryDefinitions in TOSCA v1.0. Maybe needs some usability enhancement and better description.
- "outputs" are a new feature.
- 5333 Grouping of Node Templates
- This was part of original TOSCA proposal, but removed early on from v1.0 This allows grouping
 of node templates that have some type of logically managed together as a group (perhaps to
 apply a scaling or placement policy).
- Lifecycle Operation definition independent/separate from Node Types or Relationship types (allows reuse). For now, we added definitions for "node.lifecycle" and "relationship.lifecycle".
- Override of Interfaces (operations) in the Node Template.
- 5340 Service Template Naming/Versioning
 - Should include TOSCA spec. (or profile) version number (as part of namespace)
- Allow the referencing artifacts using a URL (e.g., as a property value).
- Repository definitions in Service Template.
- Substitution mappings for Topology template.
- Addition of Group Type, Policy Type, Group def., Policy def. along with normative TOSCA base types for policies and groups.
- Addition of Artifact Processors (AP) as first class citizens

5348 A.2 Normative Types

Standards Track Work Product

5349	 Types 	es / Property / Parameters	
5350	0	list, map, range, scalar-unit types	
5351	0	Includes YAML intrinsic types	
5352	0	 NetworkInfo, PortInfo, PortDef, PortSpec, Credential 	
5353	0	TOSCA Version based on Maven	
5354	0	 JSON and XML types (with schema constraints) 	
5355	Cons	straints	
5356	0	constraint clauses, regex	
5357	0	External schema support	
5358	Node	e	
5359	0	Root, Compute, ObjectStorage, BlockStorage, Network, Port, SoftwareCol	mponent,
5360		WebServer, WebApplicaton, DBMS, Database, Container, and others	
5361	 Relat 	tionship	
5362	0	Root, DependsOn, HostedOn, ConnectsTo, AttachesTo, RoutesTo, Binds	To, LinksTo
5363		and others	
5364	Artifa	act	
5365	0	Deployment: Image Types (e.g., VM, Container), ZIP, TAR, etc.	
	TOSCA-Simple-F	Profile-YAML-v1.2-os	17 January 2

5366		 Implementation : File, Bash, Python, etc.
5367		
5368	•	Artifact Processors
5369		 New in v1.2 as "first class" citizen
5370	•	Requirements
5371		o None
5372	٠	Capabilities
5373		 Container, Endpoint, Attachment, Scalable, …
5374	٠	Lifecycle
5375		 Standard (for Node Types)
5376		 Configure (for Relationship Types)
5377	•	Functions
5378		 get_input, get_attribute, get_property, get_nodes_of_type, get_operation_output and
5379		others
5380		o concat, token
5381		○ get_artifact
5382		○ from (file)
5383	٠	Groups
5384		o Root
5385	•	Policies
5386		 Root, Placement, Scaling, Update, Performance
5387	•	Workflow
5388		 Complete declarative task-based workflow grammar.
5389	•	Service Templates
5390		 Advanced "import" concepts
5391		 Repository definitions
5392	٠	CSAR
5393		• Allow multiple top-level Service Templates in same CSAR (with equivalent functionality)

5395 Appendix B. Acknowledgments

5396 The following individuals have participated in the creation of this specification and are gratefully 5397 acknowledged:

	5
5398	Contributors:
5399	Alex Vul, (alex.vul@intel.com), Intel
5400	Avi Vachnis (avi.vachnis@alcatel-lucent.com), Alcatel-Lucent
5401	Chris Lauwers (lauwers@ubicity.com)
5402	Claude Noshpitz (claude.noshpitz@att.com), AT&T
5403	Derek Palma (dpalma@vnomic.com), Vnomic
5404	Frank Leymann (Frank.Leymann@informatik.uni-stuttgart.de), Univ. of Stuttgart
5405	Gerd Breiter (gbreiter@de.ibm.com), IBM
5406	Hemal Surti (hsurti@cisco.com), Cisco
5407	Ifat Afek (ifat.afek@alcatel-lucent.com), Alcatel-Lucent
5408	Idan Moyal, (idan@gigaspaces.com), Gigaspaces
5409	Jacques Durand (jdurand@us.fujitsu.com), Fujitsu
5410	Jin Qin, (chin.qinjin@huawei.com), Huawei
5411	Jeremy Hess, (jeremy@gigaspaces.com), Gigaspaces
5412	John Crandall, (mailto:jcrandal@brocade.com), Brocade
5413	Juergen Meynert (juergen.meynert@ts.fujitsu.com), Fujitsu
5414	Kapil Thangavelu (kapil.thangavelu@canonical.com), Canonical
5415	Karsten Beins (karsten.beins@ts.fujitsu.com), Fujitsu
5416	Kevin Wilson (kevin.l.wilson@hp.com), HP
5417	Krishna Raman (kraman@redhat.com), Red Hat
5418	Luc Boutier (luc.boutier@fastconnect.fr), FastConnect
5419	Luca Gioppo, (luca.gioppo@csi.it), CSI-Piemonte
5420	Matt Rutkowski (mrutkows@us.ibm.com), IBM
5421	Moshe Elisha (moshe.elisha@alcatel-lucent.com), Alcatel-Lucent
5422	Nate Finch (nate.finch@canonical.com), Canonical
5423	Nikunj Nemani (nnemani@vmware.com), Wmware
5424	Priya TG (priya.g@netcracker.com) NetCracker
5425	Richard Probst (richard.probst@sap.com), SAP AG
5426	Sahdev Zala (spzala@us.ibm.com), IBM
5427	Shitao li (lishitao@huawei.com), Huawei
5428	Simeon Monov (sdmonov@us.ibm.com), IBM
5429	Sivan Barzily, (sivan@gigaspaces.com), Gigaspaces
5430	Sridhar Ramaswamy (sramasw@brocade.com), Brocade
5431	Stephane Maes (stephane.maes@hp.com), HP
5432	Steve Baillargeon (steve.baillargeon@ericsson.com), Ericsson
5433	Thinh Nguyenphu (thinh.nguyenphu@nokia.com), Nokia
5434	Thomas Spatzier (thomas.spatzier@de.ibm.com), IBM
5435	Ton Ngo (ton@us.ibm.com), IBM

- 5436 Travis Tripp (travis.tripp@hp.com), HP
- 5437 Vahid Hashemian (vahidhashemian@us.ibm.com), IBM
- 5438 Wayne Witzel (wayne.witzel@canonical.com), Canonical
- 5439 Yaron Parasol (yaronpa@gigaspaces.com), Gigaspaces

5440 Appendix C. Revision History

Revision	Date	Editor	Changes Made
WDO2, Rev01	2017-09-12	Luc Boutier	Initial WD02, Revision 01 baseline for TOSCA Simple Profile in YAML v1.2
WDO2, Rev02	2017-10-03	Matt Rutkowski	 Developed Abstract.Compute and Abstract.Storage node types and inserted it into normative type hierarchy. Reveresed the inheritance of tosca.capabilities.Compute and tosca.capabilities.Container to make Container the parent (abstract) Capability Type. Adjusted all Node types that had "host" capability or requirement defns. to reflect this change. Changed TOSCA namespace URI to reflect v1.2.
WD02, Rev03	2017-10-12	Luc Boutier	 Updated policy trigger condition to leverage the constraint clause definition introduced with workflows in 1.1 Added simplified definition for constraint clause. Added Priya TG to the acknowledgements as she pushed this change/proposal.

5441