TOSCA Simple Profile in YAML Version 1.0

Committee Specification Draft 02

11 December 2014

Specification URIs

This version:

http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csd02/TOSCA-Simple-Profile-YAML-v1.0-csd02.pdf (Authoritative)
http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csd02/TOSCA-Simple-Profile-YAML-v1.0-csd02.html
http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csd02/TOSCA-Simple-Profile-YAML-v1.0-csd02.doc

Previous version:

http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csd01/TOSCA-Simple-Profile-YAML-v1.0-csd01.pdf (Authoritative)
http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csd01/TOSCA-Simple-Profile-YAML-v1.0-csd01.html
http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csd01/TOSCA-Simple-Profile-YAML-v1.0-csd01.doc

Latest version:

http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.pdf (Authoritative)
http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.html
http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.doc

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:

Derek Palma (dpalma@vnomic.com), Vnomic
Matt Rutkowski (mrutkows@us.ibm.com), IBM
Thomas Spatzier (thomas.spatzier@de.ibm.com), IBM

Related work:

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. Latest version: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html.

Declared XML namespaces:

- http://docs.oasis-open.org/tosca/ns/simple/yaml/1.0

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 Topology and Orchestration Specification for Cloud Applications (TOSCA) TC 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.oasis-open.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/.

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 Technical Committee web page (https://www.oasis-open.org/committees/tosca/ipr.php).

Citation format:

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

[TOSCA-Simple-Profile-YAML-v1.0]

TOSCA Simple Profile in YAML Version 1.0. Edited by Derek Palma, Matt Rutkowski, and Thomas Spatzier. 11 December 2014. OASIS Committee Specification Draft 02. http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csd02/TOSCA-Simple-Profile-YAML-v1.0-csd02.html. Latest version: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.html.

Notices

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

Table of Figures

1 Objective

The TOSCA Simple Profile in YAML specifies a rendering of TOSCA which aims to provide a more 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 cloud applications.

This proposal describes a YAML rendering for TOSCA. YAML is a human friendly data serialization standard (http://yaml.org/) with a syntax much easier to read and edit than XML. As there are a number of DSLs encoded in YAML, a YAML encoding of the TOSCA DSL makes TOSCA more accessible by these communities.

This proposal prescribes an isomorphic rendering in YAML of a subset of the TOSCA v1.0 ensuring that TOSCA semantics are preserved and can be transformed from XML to YAML or from YAML to XML. Additionally, in order to streamline the expression of TOSCA semantics, the YAML rendering is sought to be more concise and compact through the use of the YAML syntax.

2 Summary of key TOSCA concepts

The TOSCA metamodel uses the concept of service templates to describe cloud workloads as a topology 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 system of node types to describe the possible building blocks for constructing a service template, as well as relationship type to describe possible kinds of relations. Both node and relationship types may define lifecycle operations to implement the behavior an orchestration engine can invoke when instantiating a service template. For example, a node type for some software product might provide a ‘create’ operation to handle the creation of an instance of a component at runtime, or a ‘start’ or ‘stop’ operation to handle a start or stop event triggered by an orchestration engine. Those lifecycle operations are backed by 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 ‘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 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 (see Appendix C). 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 existing types, for example by providing a customized ‘create’ script for some software.

3 A “hello world” template for TOSCA Simple Profile in YAML

As mentioned before, the TOSCA simple profile assumes the existence of a base set of node types (e.g., a ‘Compute’ node) and other types for creating TOSCA Service Templates. It is envisioned that many additional node types for building service templates will be created by communities. Consequently, a most basic TOSCA template for deploying just a single server would look like the following:

Example 1 - TOSCA Simple "Hello World"

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

description: Template for deploying a single server with predefined properties.

topology_template:
  node_templates:
    my_server:
      type: tosca.nodes.Compute
      properties:
        # Compute properties
        num_cpus: 2
        disk_size: 10 GB
        mem_size: 4 MB
      capabilities:
        os:
          properties:
            # host Operating System image properties
            architecture: x86_64
            type: linux
            distribution: rhel
            version: 6.5
```

The template above contains a very simple topology template with only the definition of one single ‘Compute’ node template with predefined (hardcoded) values for number of CPUs, memory size, etc. When instantiated in a provider environment, the provider would allocate a physical or virtual server that meets those specifications. The set of properties of any node type, as well as their schema definition, is defined by the respective node type definitions, which a TOSCA orchestration engine can resolve to validate the properties provided in a template. The Compute node also has built-in TOSCA Capabilities; one is named “**os**”, which is used to provide values to indicate what host operating system the Compute node should have when it is instantiated.

3.1 Requesting input parameters and providing output

Typically, one would want to allow users to customize deployments by providing input parameters instead 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 IP address of the deployed server). A refined service template with corresponding **inputs** and **outputs** sections is shown below.

Example 2 - Template with input and output parameter sections

```
tosca_definitions_version: tosca_simple_yaml_1_0_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
      properties:
        # Compute properties
        num_cpus: { get_input: cpus }
        mem_size: 4 MB
        disk_size: 10 GB

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

The **inputs** and **outputs** sections are contained in the **topology_template** element of the TOSCA template, meaning that they are scoped to node templates within the topology template. Input parameters 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 containing topology template.

Note that the **inputs** section of a TOSCA template allows for defining optional constraints on each input parameter to restrict possible user input. Further note that TOSCA provides for a set of intrinsic functions like **get_input**, **get_property** or **get_attribute** to reference elements within the template or to retrieve runtime values.

4 TOSCA template for a simple software installation

Software installations can be modeled in TOSCA as node templates that get related to the node template for a server on which the software shall be installed. With a number of existing software node types (e.g. either created by the TOSCA work group or a community) template authors can just use those node types for writing service templates as shown below.

Example 3 - Simple (MySQL) software installation on a TOSCA Compute node

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

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

topology_template:
  inputs:
    # omitted here for sake of brevity

  node_templates:
    mysql:
      type: tosca.nodes.DBMS.MySQL
      properties:
        dbms_root_password: { get_input: my_mysql_rootpw }
        dbms_port: { get_input: my_mysql_port }
      requirements:
        - host: db_server

    db_server:
      type: tosca.nodes.Compute
      properties:
        # omitted here for sake of 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 **dbms_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 **dbms_root_password** property. The same is true for the **dbms_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 node type defines a requirement called **host**, which needs to be fulfilled by pointing to a node template of type **tosca.nodes.Compute**.

Within the **requirements** section, all entries contain the name of a requirement as key and the identifier of the fulfilling entity as value, expressing basically a named reference to some other node. In the example above, the **host** requirement is fulfilled by referencing the **db_server** node template.

5 Overriding behavior of predefined node types

Node types in TOSCA have associated implementations that provide the automation (e.g. in the form of scripts or Chef recipes) for lifecycle operations of a node. For example, the node type implementation for MySQL will provide the scripts to configure, start, or stop MySQL at runtime.

If it is desired to use a custom script for one of the operation defined by a node type in the context of a specific template, the default implementation can be easily overridden by providing a reference to the own automation in the template as shown in the following example:

Example 4 - Node Template overriding its Node Type's "configure" interface

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

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

topology_template:
  inputs:
    # omitted here for sake of brevity

  node_templates:
    mysql:
      type: tosca.nodes.DBMS.MySQL
      properties:
        dbms_root_password: { get_input: my_mysql_rootpw }
        dbms_port: { get_input: my_mysql_port }
      requirements:
        - host: db_server
      interfaces:
        Standard:
          configure: scripts/my_own_configure.sh

    db_server:
      type: tosca.nodes.Compute
      properties:
        # omitted here for sake of brevity
```

In the example above, an own script for the **configure** operation of the MySQL node type's lifecycle interface is provided. The path given in the example above is interpreted relative to the template file, but it would also be possible to provide an absolute URI to the location of the script.

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).

6 TOSCA template for database content deployment

In the example shown in section 4 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.

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

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

description: Template for deploying MySQL and database content.

topology_template:
  inputs:
    # omitted here for sake of brevity

  node_templates:
    my_db:
      type: tosca.nodes.Database.MySQLDatabase
      properties:
        db_name: { get_input: database_name }
        db_user: { get_input: database_user }
        db_password: { get_input: database_password }
        db_port: { get_input: database_port }
      artifacts:
        - db_content: files/my_db_content.txt
          type: tosca.artifacts.File
      requirements:
        - host: mysql

    mysql:
      type: tosca.nodes.DBMS.MySQL
      properties:
        dbms_root_password: { get_input: mysql_rootpw }
        dbms_port: { get_input: mysql_port }
      requirements:
        - host: db_server

    db_server:
      type: tosca.nodes.Compute
      properties:
        # omitted here for sake of 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. In its **artifacts** section, the node template points to a text file (i.e., `my_db_content.txt`) which can be used to help create the database content during deployment time. The **requirements** section of the **my_db** node template expresses that the database is hosted on a MySQL DBMS represented by the **mysql** node.

Note that while it would be possible to define one node type and corresponding node templates that represent both the DBMS middleware and actual database content as one entity, TOSCA distinguishes between middleware node types and application layer node types. This allows at the one hand to have better re-use of generic middleware node types without binding them to content running on top, and on the other hand this allows for better substitutability of, for example, middleware components during the deployment of TOSCA models.

7 TOSCA template for a two-tier application

The definition of multi-tier applications in TOSCA is quite similar to the example shown in section 4, 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.

Example 6 - Basic two-tier application (web application and database server tiers)

```
tosca_definitions_version: tosca_simple_yaml_1_0_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

  node_templates:
    wordpress:
      type: tosca.nodes.WebApplication.WordPress
      properties:
        admin_user: { get_input: wp_admin_username }
        admin_password: { get_input: wp_admin_password }
        db_host: { get_property: [ db_server, ip_address ] }
      requirements:
        - host: apache
        - database_endpoint: wordpress_db
      interfaces:
        Standard:
          inputs:
            db_host: { get_property: [ db_server, ip_address ] }
            db_port: { get_property: [ wordpress_db, db_port ] }
            db_name: { get_property: [ wordpress_db, db_name ] }
            db_user: { get_property: [ wordpress_db, db_user ] }
            db_password: { get_property: [ wordpress_db, db_password ] }

    apache:
      type: tosca.nodes.WebServer.Apache
      properties:
        # omitted here for sake of brevity
      requirements:
        - host: web_server

    web_server:
      type: tosca.nodes.Compute
      properties:
        # omitted here for sake of brevity

    wordpress_db:
      type: tosca.nodes.Database.MySQL
      properties:
        db_name: { get_input: wp_db_name }
        db_user: { get_input: wp_db_user }
        db_password: { get_input: wp_db_password }
        db_port: { get_input: wp_db_port }
      requirements:
        - host: mysql

    mysql:
      type: tosca.nodes.DBMS.MySQL
      properties:
        dbms_root_password: { get_input: mysql_rootpw }
        dbms_port: { get_input: mysql_port }
      requirements:
        - host: db_server
```

```
    db_server:
      type: tosca.nodes.Compute
      properties:
        # omitted here for sake of brevity
```

The web application stack consists of the **wordpress**, the **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 **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 **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 respective input parameters (as defined for the configure operation of node type **tosca.nodes.WebApplication.WordPress** – see section 6) are mapped to properties of the **wordpress_db** node via the **get_property** function.

Note: besides the **configure** lifecycle operation (i.e., from the **tosca.interfaces.node.lifecycle.Standard** interface) of the **wordpress** node template, more operations would be listed in a complete TOSCA template. Those other operations have been omitted for the sake of brevity.

8 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 executed at runtime as shown in the following example.

Example 7 – Providing a custom script to establish a connection

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

description: Template for deploying a two-tier application on two servers.

topology_template:
  inputs:
    # omitted here for sake of brevity

  node_templates:
    wordpress:
      type: tosca.nodes.WebApplication.WordPress
      properties:
        # omitted here for sake of brevity
      requirements:
        - host: apache
        - database_endpoint:
            node: wordpress_db
            relationship:
              interfaces:
                tosca.interfaces.relationships.Configure:
                  pre_configure_source: scripts/wp_db_configure.sh

    wordpress_db:
      type: tosca.nodes.Database.MySQL
      properties:
        # omitted here for the sake of brevity
```

```
     requirements:
       - host: mysql

  # 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 a **DatabaseEndpoint** Capability Type using a ConnectsTo relationship. The **wordpress_db** node template's underlying **MySQL** type definition indeed provides the **DatabaseEndpoint** 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 C.5.4) 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.

9 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.

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

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

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

description: Template for deploying a two-tier application on two servers.

topology_template:
  inputs:
    # omitted here for sake of brevity

  node_templates:
    wordpress:
      type: tosca.nodes.WebApplication.WordPress
      properties:
        # omitted here for sake of brevity
      requirements:
        - host: apache
        - database_endpointase:
            node: wordpress_db
            relationship: my.types.WordpressDbConnection

    wordpress_db:
      type: tosca.nodes.Database.MySQL
      properties:
        # omitted here for the sake of 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.

9.1 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).

Example 9 - Defining a custom relationship type

```
tosca_definitions_version: tosca_simple_yaml_1_0_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.

10 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 C.7.1).

Example 10 - Simple dependency relationship between two nodes

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

description: Template with a generic dependency between two nodes.

topology_template:
  inputs:
    # omitted here for sake of brevity

  node_templates:
    my_app:
      type: my.types.MyApplication
      properties:
        # omitted here for sake of brevity
      requirements:
        - dependency: some_service

    some_service:
      type: some.type.SomeService
      properties:
        # omitted here for sake of 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 8), the template author who knows that there is a timing dependency and can use the generic **dependency** requirement to express that constraint using the very same syntax as used for all other references.

11 Defining requirements on the hosting infrastructure for a software installation

Instead of defining software installations and the hosting infrastructure (the servers) in the same template, it is also 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 and place software components on top.

The following example shows how such generic hosting requirements can be expressed in the **requirements** section of node templates.

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

description: Template with requirements against hosting infrastructure.

topology_template:
  inputs:
    # omitted here for sake of brevity

  node_templates:
    mysql:
      type: tosca.nodes.DBMS.MySQL
      properties:
        # omitted here for sake of brevity
      requirements:
        - host:
            node: tosca.nodes.Compute
            target_filter:
              properties:
                - num_cpus: { in_range: [ 1, 4 ] }
                - mem_size: { greater_or_equal: 2 }
              capabilities:
                - os:
                    properties:
                      - architecture: x86_64
                      - type: linux
                      - distribution: ubuntu
```

In the example above, it is expressed that the **mysql** component requires a host of type **Compute**. In contrast to previous examples, there is no reference to any node template but just a specification of the type of required node. At deployment time, the provider will thus have to allocate or assign a resource of the given type.

In the **`constraints`** section, the characteristics of the required compute node can be narrowed down by defining boundaries for the memory size, number of CPUs, etc. Those constraints can either be expressed by means of concrete values (e.g. for the **`architecture`** attribute) which will require a perfect match, or by means of qualifier functions such as **`greater_or_equal`**.

12 Defining requirements on a database for an application

In the same way requirements can be defined on the hosting infrastructure 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.

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

description: Template with a database requirement.

topology_template:
  inputs:
    # omitted here for sake of 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, db_endpoint_url ] }
      requirements:
        - database_endpoint:
            node: tosca.nodes.DBMS.MySQL
            target_filter:
              properties:
                - mysql_version: { greater_or_equal: 5.5 }
```

In the example above, the application **my_app** needs a MySQL database, where the version of MySQL must be 5.5 or higher. The example shows an additional feature of referencing a property of the database to get the database connection endpoint URL at runtime via the **get_property** intrinsic function. The **get_property** function allows for getting a property via a reference expressed in the **requirements** section. The first argument is a keyword (SELF) to indicate the requirement is in the current node, the second parameter is the name of a reference to another node, in this case as described by the requirement named **database** in the example above – and the last argument is the name of the property of the referenced node, which must be defined by the respective node type **tosca.nodes.DBMS.MySQL**.

13 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.

13.1 Understanding node template instantiation through a TOSCA Orchestrator

When a topology template is instantiated by a TOSCA Orchestrator, the orchestrator has to look for realizations of the single node templates according to the node types specified for each 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 get “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.

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.

13.2 Definition of the top-level service template

The following sample defines a web application **web_app** connected to a database **db**. In this example, the complete hosting stack for the application is defined within the same topology template: the web application is hosted on a web server **web_server**, which in turn is installed (hosted) on a compute node **server**.

The hosting stack for the database **db**, in contrast, is not defined within the same file but only the database is represented as a node template of type **tosca.nodes.Database**. The underlying hosting stack for the database is defined in a separate template file, which is shown later in this section. Within the current template, only a number of properties (**db_user**, **db_password**, **db_name**) are assigned to the database using hardcoded values in this simple example.

Note that in contrast to the use case described in section 12 where a database was abstractly referred 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. For example, if multiple components shall use the same database (or any other sub-system of the overall service), this can be expressed by means of normal relations between node templates, whereas such modeling would not be possible in **requirements** sections of disjoint node templates.

```
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

  db:
    type: tosca.nodes.Database
    properties:
      db_user: my_db_user
      db_password: secret
      db_name: my_db_name
```

13.3 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.

In short, the substitution_mappings section provides the following information:

1. It defines what node templates, i.e. node templates of which type, can be substituted by the topology template.

2. It defines how capabilities of the substituted node (or the capabilities defined by the node type of the substituted node template, respectively) are bound to capabilities of node templates defined in the topology template.

3. It defines how requirements of the substituted node (or the requirements defined by the node type of the substituted node template, respectively) are bound to requirements of node templates defined in the topology template.

```
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 sake of brevity

  substitution_mappings:
    node_type: tosca.nodes.Database
    capabilities:
      database_endpoint: [ database, database_endpoint ]

  node_templates:
    database:
      type: tosca.nodes.Database
      properties:
        db_user: { get_input: db_user }
        # other properties omitted for sake of brevity
      requirements:
        - host: dbms

    dbms:
      type: tosca.nodes.DBMS
      # details omitted for sake of brevity

    server:
      type: tosca.nodes.Compute
      # details omitted for sake of brevity
```

The **substitution_mappings** section in the sample above 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** capabilities of the **database** node contained in the topology template.

Note that the **substitution_mappings** section does not define any mappings for requirements of the Database node type, since all requirements are fulfilled by other nodes templates in the current topology template. In cases where a requirement of a substituted node is bound in the top-level service template 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 have to match the 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.

14 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.

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.

Example 11 - Grouping Node Templates with same scaling policy

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

description: Template for a scaling web server.

topology_template:
  inputs:
    # omitted here for sake of brevity

  node_templates:
    apache:
      type: tosca.nodes.WebServer.Apache
      properties:
        http_port: 8080
        https_port: 8443
      requirements:
        - host: server

    server:
      type: tosca.nodes.Compute
      properties:
        # omitted here for sake of brevity

  group:
    webserver_group:
      members: [ apache, server ]
      policies:
        - my_scaling_policy:
            # Specific policy definitions are considered domain specific and
            # are not included here
```

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.

Furthermore, a scaling policy is defined for the group to express that the group as a whole (i.e. pairs of **server** node and the **apache** component installed on top) should scale up or down under certain 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 11, a provider could decide to place software components on the same host if their hosting requirements match, or to place them onto different hosts.

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

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

description: Template hosting requirements and placement policy.

topology_template:
  inputs:
    # omitted here for sake of brevity

  node_templates:
    wordpress:
      type: tosca.nodes.WebApplication.Wordpress
      properties:
        # omitted here for sake of brevity
      requirements:
        - host:
            node: tosca.nodes.Compute
            target_filter:
              properties:
                - mem_size: { greater_or_equal: 2 MB }
              capabilities:
                - os:
                    properties:
                      - architecture: x86_64
                      - type: linux

    mysql:
      type: tosca.nodes.DBMS.MySQL
      properties:
        # omitted here for sake of brevity
      requirements:
        - host:
            node: tosca.nodes.Compute
            target_filter:
              properties:
                - disk_size: { greater_or_equal: 10 }
              capabilities:
                - os:
                    properties:
                      - architecture: x86_64
                      - type: linux

  groups:
    my_collocation_group:
      members: [ wordpress, mysql ]
      policies:
        - my_anti_collocation_policy:
             # Specific policy definitions are considered domain specific and
             # are not included here
```

In the example above, both software components **`wordpress`** and **`mysql`** have identical hosting requirements. Therefore, a provider could decide to put both on the same server. By defining a group of the two components and attaching an anti-collocation policy to the group it can be made sure, though, that both components are put onto different hosts at deployment time.

15 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.

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 “**dsl_definitions**” section of the TOSCA Service Template and reference the anchor name as an alias in any **Compute** node templates where these properties may need to be reused. For example:

```
tosca_definitions_version: tosca_simple_yaml_1_0_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: 4096 KB

topology_template:
  node_templates:
    my_server:
      type: Compute
      properties: *my_compute_node_props

    my_database:
      type: Compute
      properties: *my_compute_node_props
```

16 Passing information as inputs to Nodes and Relationships

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

16.1 Example: declaring input variables for all operations in all interfaces

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

16.2 Example: declaring input variables for all operations on a single 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 ] }
```

16.3 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 ] }
```

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.

16.4 Example: setting output variables to an attribute

```
node_templates:
  frontend:
    type: tosca.nodes.WebApplication.WordPress
    attributes:
      url: { get_operation_output: [ SELF, Standard, create, generated_url ] }
    interfaces:
      Standard:
        create:
          implementation: scripts/frontend/create.sh
```

In this example, the Standard create operation exposes / exports an environment variable named "**generated_url**" attribute which will be assigned to the WordPress node's **url** attribute.

16.5 Example: passing output variables between operations

```
node_templates:
  frontend:
    type: tosca.nodes.WebApplication.WordPress
    interfaces:
      Standard:
        create:
          implementation: scripts/frontend/create.sh
        configure:
          implementation: scripts/frontend/configure.sh
          inputs:
            data_dir: { get_operation_output: [ SELF, Standard, create, data_dir ]
}
```

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

17 Topology Template Model versus Instance Model

A TOSCA service template contains a **topology template,** which models the components of an application, their relationships and dependencies (a.k.a., a topology model) that get interpreted and instantiated by TOSCA Orchestrators. The actual node and relationship instances that are created represent a set of resources distinct from the template itself, called a **topology instance (model).** The direction of this specification is to provide access to the instances of these resources for management 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 fashion, That is, the orchestrator can choose the order of resources to instantiate (i.e., establishing a partial set of node and relationship instances) and have the ability, as they are being created, to access them in order to facilitate instantiating the remaining resources of the complete topology template.

18 Using attributes implicitly reflected from properties

Most entity types in TOSCA (e.g., Node, Relationship, Requirement and Capability Types) 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.

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

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.

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

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

node_types:
  ServerNode:
    derived_from: SoftwareComponent
    properties:
      notification_port:
        type: integer
    capabilities:
      # omitted here for sake of brevity

  ClientNode:
    derived_from: SoftwareComponent
    properties:
      # omitted here for sake of brevity
    requirements:
      - server:
          node: ServerNode
          relationship:
            type: ConnectsTo
            # Augment resulting Relationship’s interfaces by providing inputs
            interfaces:
              Configure:
                inputs:
                  targ_notify_port: { get_attribute: { [ TARGET, notification_port }
                                          }
          # other operation definitions omitted here for sake of brevity

topology_template:
  node_templates:

    my_server:
      type: ServerNode
      properties:
        notification_port: 8000

    my_client:
      type: ClientNode
      requirements:
        - server: my_server
```

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

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 “runtime” value would use the **get_attribute** function instead of the **get_property** function.

Appendix A. TOSCA Simple Profile definitions in YAML

This section describes all of the YAML block structure for all keys and mappings that are defined for the TOSCA Version 1.0 Simple Profile specification that are needed to describe a TOSCA Service Template (in YAML).

A.1 TOSCA namespace and alias

The following table defines the namespace alias and (target) namespace values that SHALL be used when referencing the TOSCA Simple Profile version 1.0 specification.

Alias	Target Namespace	Specification Description
tosca_simple_yaml_1_0_0	http://docs.oasis-open.org/tosca/ns/simple/yaml/1.0	The TOSCA Simple Profile v1.0 (YAML) target namespace and namespace alias.

A.1.1 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 and service templates can be "nested" rules are needed so that TOSCA Orchestrators know how to avoid collisions and apply their own namespaces when import and nesting occur.

The following cases are considered:

- Duplicate property names within same entity (e.g., Node Type, Node Template, Relationship Type, etc.)
- Duplicate requirement names within same entity (e.g., Node Type, Node Template, Relationship Type, etc.)
- Duplicate capability names within same entity (e.g., Node Type, Node Template, Relationship Type, etc.)
- Collisions that occurs from "import" for any Type or Template.
- Collision that occurs from "nesting" for any Type or Template.

A.2 Parameter and property types

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

A.2.1 Referenced YAML Types

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

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
null	tag:yaml.org,2002:null

A.2.1.1 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.

A.2.2 TOSCA base types

This specification defines the following types that may be used when defining properties or parameters.

A.2.2.1 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. Therefore, the TOSCA TC intends to provide a normative version type (string) for this purpose in future Working Drafts of this specification.

A.2.2.1.1 *Grammar*

TOSCA version strings have the following grammar:

```
<major_version>.<minor_version>.<fix_version>[.<qualifier>[-<build_version] ]
```

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 a required 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**.

A.2.2.1.2 *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.

A.2.2.1.3 *Examples*

Example of a version with

```
# basic version string
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
```

A.2.2.1.4 *Notes*

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

A.2.2.2 TOCSA 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.

A.2.2.2.1 *Grammar*

TOSCA range values have the following grammar:

```
[<lower_bound>, <upper_bound>]
```

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**.

A.2.2.2.2 *Examples*

Example of a node template property with a range value:

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

A.2.2.3 TOCSA 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.

A.2.2.3.1 *Recognized Units*

Unit	Usage	Description
B	size	byte
kB	size	kilobyte (1000 bytes)
MB	size	megabyte (1000000 bytes)
GB	size	gigabyte (1000000000 bytes)
TB	size	terabyte (1000000000000 bytes)

A.2.2.3.2 *Grammar*

TOSCA scalar-unit typed values have the following grammar:

```
<scalar> <unit>
```

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

- **`scalar`**: is a required scalar value.
- **`unit`**: is a required unit value. The unit value MUST be type-compatible with the scalar.

A.2.2.3.3 *Examples*

```
Storage size in Gigabytes
properties:
  storage_size: 10 GB
```

A.2.2.3.4 *Additional requirements*

- **<u>Whitespace</u>**: any number of spaces (including zero or none) is allowed between the **`scalar`** value and the **`unit`** value.
- When performing constraint clause evaluation on values of the scalar-unit type, both the scalar value portion and unit value portion MUST 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:
    - `storage_size: in_range { 4 GB, 20 GB }`

  where **`storage_size`**'s range would be evaluated using both the numeric and unit values (combined together), in this case '4 GB' and '20 GB'.

A.2.2.3.5 *Notes*

- The unit values recognized by TOSCA Simple Profile for size-type units are based upon a subset of those defined by GNU at http://www.gnu.org/software/parted/manual/html_node/unit.html , which is a non-normative reference to this specification.

A.2.2.4 TOSCA list type

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 using the list data type.

Note that entries in a list 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.

A.2.2.4.1 *Grammar*

TOSCA lists normal YAML lists with the following grammars:

A.2.2.4.2 *Square bracket notation*

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

A.2.2.4.3 *Bulleted list notation*

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

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

- **<list_entry_*>**: represents one entry of the list

A.2.2.4.4 *Examples*

Example of node template property with a list value:

A.2.2.4.5 *Square bracket notation*

```
listen_ports: [ 80, 8080 ]
```

A.2.2.4.6 *Bulleted list notation*

```
listen_ports:
  - 80
  - 8080
```

A.2.2.5 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.

A.2.2.5.1 *Grammar*

TOSCA maps are normal YAML dictionaries with following grammar:

A.2.2.5.2 *Single-line grammar*

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

A.2.2.5.3 *Multi-line grammar*

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

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

A.2.2.5.4 *Examples*

Example of a node template property with a map value:

A.2.2.5.5 *Single-line notation*

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

A.2.2.5.6 *Multi-line notation*

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

In the examples above, two notation options are given: option 1 is using a notation where each map entry is one 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. Option 2 is using a notation that is useful for only short maps with simple entries.

A.3 Normative values

A.3.1 Node States

As components (i.e., nodes) of TOSCA applications are deployed, instantiated and orchestrated over their lifecycle using normative lifecycle operations (see section C.6 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.

The following table provides the list of recognized node states for TOSCA Simple Profile that would be set by the orchestrator to describe a node instance’s 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.
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.

A.3.1.1 Additional requirements

- None

A.4 TOSCA entity and element definitions (meta-model)

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

A.4.1 Description element

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

A.4.1.1 Keyname

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

```
description
```

A.4.1.2 Grammar

The description element is a YAML string.

```
description: <string>
```

A.4.1.3 Examples

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

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

The YAML “folded” style may also be used for multi-line descriptions which “folds” line breaks as space 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.
```

A.4.1.4 Notes

- Use of “folded” style is discouraged for the YAML string type apart from when used with the **`description`** keyname.

A.4.2 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.

A.4.2.1 Operator keynames

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

Operator	Type	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.

Operator	Type	Value Type	Description
in_range	dual scalar	comparable	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	Constrains the property or parameter to a value of a given length.
min_length	scalar	string	Constrains the property or parameter to a value to a minimum length.
max_length	scalar	string	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.

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

A.4.2.2 Grammar

Constraint clauses take one of the following forms:

```
# 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>
```

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

- **`operator`**: represents a required operator from the specified list shown above (see section A.4.2.1 "Operator keynames").
- **`scalar_value, scalar_value_*`**: represents a required scalar (or atomic quantity) that can hold only one value at a time. This will be a value of a primitive type, such as an integer or string that is allowed by this specification.
- **`value_*`**: represents a required value of the operator that is not limited to scalars.
- **`reqular_expression_value`**: represents a regular expression (string) value.

A.4.2.3 Examples

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
```

A.4.2.4 Notes

- 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.

A.4.3 Constraints element

The Constraints element specifies a sequenced list of constraints on one or more of the Service Template’s properties, parameters or other typed elements of the TOSCA Simple Profile. A constraints element is represented as a YAML block collection that contains a sequenced list of nested constraint clauses.

A.4.3.1 Keyname

The following keyname is used to provide a list of constraints within the TOSCA Simple Profile specification:

```
constraints
```

A.4.3.2 Grammar

The constraints element is described as a YAML block collection that contains a sequence of constraint clauses:

```
<some_typed_property_name>:
  constraints:
    - <constraint_clause_1>
    - ...
    - <constraint_clause_n>
```

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

- **some_typed_property_name**: represents the required symbolic name of a typed property definition, as a string, which can be associated to a TOSCA entity.
  - For example, a property (definition) can be declared as part of a Node Type or Node Template definition or it can be used to define an input or output property (parameter) for a Service Template's.
- **constraint_clause_***: represents constraint clauses for the associated property or parameter.

A.4.3.3 Examples

Constraint on an integer-typed parameter definition:

```
# An example input parameter that represents a number of CPUs
# and constrains its value to a specific range.
inputs:
  num_cpus:
    type: integer
    constraints:
      - in_range: [ 2, 4 ]
```

Constraints on a string-typed parameter definition:

```
# An example input parameter that represents a user ID and constrains its length.
inputs:
  user_id:
    type: string
    constraints:
      - min_length: 8
      - max_length: 16
```

A.4.3.4 Notes

- Constraints of properties or parameters SHOULD be type-compatible with the type defined for that property or parameter.
- 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.

A.4.4 Property definition

A property definition defines a named, typed value and related data that can be associated with an entity defined in this specification. It is used to provide a transparent property or characteristic of that entity which can either be set on or retrieved from it. Properties are used by template authors to provide the “desired state”, as input to TOSCA entities for use when they are instantiated. The value of a property can be retrieved using the **get_property** function within TOSCA Service Templates.

A.4.4.1 Keynames

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

Keyname	Required	Type	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**). If this key is not declared for property definition, then the property SHALL be considered required by default.
default	no	<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.
constraints	no	constraints	None	The optional list of sequenced constraints for the property.
status	no	string	default: supported	The optional status of the property relative to the specification or implementation. See table below for valid values.
entry_schema	no	schema	None	The optional key that is used to declare the schema definition for entries of “container” types such as the TOSCA list or map.

A.4.4.2 Status values

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	Indicates the property is not supported.

Value	Description
experimental	Indicates the property is experimental and has no official standing.
deprecated	Indicates the property has been deprecated by a new specification version.

A.4.4.3 Grammar

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:
    <schema_definition>
```

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

- **property_name**: represents the required symbolic name of the property as a string.
- **property_type**: represents the required data type of the property.
- **property_description**: represents the optional description 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 (as shown in the constraints element) on the property definition.
- **schema_definition**: represents the optional entry schema used to declare the (anonymous type) schema for set types (e.g., list).

A.4.4.4 Example

The following represents a required property definition:

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

A.4.4.5 Additional Requirements

- 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.4.4.6 Notes

- 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.

A.4.5 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 Type or Relationship Type). Specifically, it is used to expose the "actual state" of some property of a TOSCA entity (set by the orchestrator) after it has been deployed and instantiated. Attribute values can be retrieved via the **`get_attribute`** function from the instance model and used as inputs to other entities within TOSCA Service Templates.

A.4.5.1 Keynames

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

Keyname	Required	Type	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>	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.

A.4.5.2 Grammar

Named attribute definitions have the following grammar:

```
<attribute_name>:
  type: <attribute_type>
  description: <attribute_description>
  default: <default_value>
  status: <status_value>
```

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.
- **`default_value`**: contains a type-compatible value that may be used as a default if not provided by another means.

A.4.5.3 Example

The following represents a required attribute definition:

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

A.4.5.4 Notes

- 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.
- 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).

A.4.6 Parameter definition

A parameter definition is map used to declare a name for a parameter along with its value to be used as inputs for operations. This value can either be a fixed value or one that is evaluated from a function or expression.

A.4.6.1 Keynames

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

Keyname	Required	Type	Description
N/A	N/A	N/A	N/A

A.4.6.2 Grammar

Named property definitions have the following grammar:

```
<parameter_name> : <value> | { <value_expression> }
```

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

- **parameter_name**: represents the required symbolic name of the parameter as a string.
- **value**: represents the required value to associate with the parameter name.
- **value_expression**: represents an expression, that when evaluated, provides the required value to associate with the parameter name.

A.4.6.3 Example

The following represents a required property definition:

```
...
  interfaces:
    Standard:
      create:
        inputs:
          # Parameter definition
          compute_memory: { get_property: [ my_host, mem_size ] }
```

A.4.6.4 Additional Requirements

- 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.4.6.5 Notes

- 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.

A.4.7 Operation definition

An operation definition defines a named function or procedure that can be bound to an implementation artifact (e.g., a script).

A.4.7.1 Keynames

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

Keyname	Required	Type	Description
description	no	description	The optional description string for the associated named operation.
implementation	no	string	The optional implementation artifact name (e.g., a script file name within a TOSCA CSAR file).
inputs	no	list of parameter definitions	The optional list of input parameter definitions.

A.4.7.2 Grammar

Named operation definitions have the following grammars:

A.4.7.2.1 *Short notation*

The following single-line grammar may be used when only an operation's implementation artifact is needed:

```
<operation_name>: <implementation_artifact_name>
```

A.4.7.2.2 *Extended notation*

The following multi-line grammar may be used when additional information about the operation is needed:

```
<operation_name>:
   description: <operation_description>
   implementation: <implementation_artifact_name>
   inputs:
     <parameter_definitions>
```

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.
- **operation_description**: represents the optional description string for the corresponding **operation_name**.
- **implementation_artifact_name**: represents the optional name (string) of an 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).
- **parameter_definitions**: represents the optional list of parameter definitions which the TOSCA orchestrator would make available (i.e., or pass) to the corresponding implementation artifact during its execution.

A.4.7.3 Notes

- Implementation artifact file names (e.g., script filenames) may include file directory path names that are relative to the TOSCA service template file itself when packaged within a TOSCA Cloud Service ARchive (CSAR) file.

A.4.8 Interface definition

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

A.4.8.1 Keynames

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

Keyname	Type	Description
inputs	list of parameter definitions	The optional list of input parameter definitions.

A.4.8.2 Grammar

The following keyname is used to provide a list of properties within the TOSCA Simple Profile specification:

```
<interface_definition_name>:
  inputs:
    <parameter_definitions>
  <operation_definition_1>
  ...
  <operation_definition_n>
```

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

- **interface_definition_name:** represents the required symbolic name of the interface definition as a string.
- **parameter_definitions**: represents the optional list of parameter definitions 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_definition_*:** represents the required name of one or more operation definitions.

A.4.8.3 Example

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

```
mycompany.mytypes.myinterfaces.MyConfigure:
  configure_service_A:
    description: My application’s custom configuration interface for service A.
  configure_service_B:
    description: My application’s custom configuration interface for service B.
```

A.4.9 Artifact definition

An artifact definition defines a named, typed file that can be associated with Node Type or Node Template and used by orchestration engine to facilitate deployment and implementation of interface operations.

A.4.9.1 Keynames

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

Keyname	Type	Required	Description
type	string	no	The optional data type for the artifact definition.
description	description	no	The optional description for the artifact definition.
mime_type	string	no	The optional Mime type for finding the correct artifact definition when it is not clear from the file extension.
deploy_path	string	no	The file path the associated file would be deployed into within the target node’s container.

A.4.9.2 Grammar

Named artifact definitions have the following grammars:

A.4.9.2.1 *Short notation*

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

```
<artifact_name>: <artifact_file_URI>
```

A.4.9.2.2 ***Extended notation:***

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

```
<artifact_name>:
  implementation: <artifact_file_URI>
  type: <artifact_type_name>
  description: <artifact_description>
  mime_type: <artifact_mime_type_name>
```

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

- **artifact_name**: represents the required symbolic name of the artifact definition as a string.
- **artifact_file_URI**: represents the required URI string (relative or absolute) which can be used to locate the artifact's file.
- **artifact_type_name**: represents the required artifact type the artifact definition is based upon.
- **artifact_description**: represents the optional description string for the corresponding **artifact_name**.
- **artifact_mime_type_name**: represents the optional, explicit Mime Type (as a string) for the associated artifact definition when it is not clear from the file description.

A.4.9.3 Example

The following represents an artifact definition:

```
my_file_artifact: ../my_apps_files/operation_artifact.txt
```

A.4.10 Artifacts element

The Artifacts element is used to associate one or more typed artifact definitions with a TOSCA Node Type or Node Template.

A.4.10.1 Keynames

The following keyname is used to declare a list of artifacts within the TOSCA Simple Profile specification:

```
artifacts
```

A.4.10.2 Grammar

The artifacts element is described by a YAML block collection that contains a ***sequenced*** list of artifact definitions:

```
<some_typed_entity_name>:
  artifacts:
    - <artifact_definition_1>
    - ...
    - <artifact_definition_n>
```

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

- **some_typed_entity_name:** represents the symbolic name (string) of a typed TOSCA entity (e.g., a Node Type, Node Template) that has, as part of its definition, a list of artifacts.
- **artifact_definition_*:** represents one or more Artifact definitions for the associated entity.

A.4.10.3 Examples

The following examples show artifact definitions in both simple and full forms being associated to Node Types:

```
TBD
```

A.4.11 Interfaces element

The Interfaces element describes a list of one or more interface definitions for a modelable entity (e.g., a Node or Relationship Type) as defined within the TOSCA Simple Profile specification. Each interface definition contains one or more interfaces for operations that can be invoked on the associated entity.

A.4.11.1 Keyname

The following keyname is used to declare a list of interfaces definition names within the TOSCA Simple Profile specification:

```
interfaces
```

The following is the list of recognized keynames recognized for a TOSCA interfaces element:

Keyname	Type	Required	Description
inputs	list of parameter definitions	no	The optional list of input parameter definitions

A.4.11.2 Grammar

A.4.11.2.1 *Short notation*

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

```
# Declaration of valid interface (type) names
interfaces: [ <interface_defn_name_1>, ..., <interface_defn_name_n> ]
```

A.4.11.2.2 *Extended notation*

The following multi-line grammar may be used when interface definitions, along with any additional input parameter information, are needed to define a set of interfaces:

```
interfaces:
  inputs:
    <parameter_definitions>
  <interface_defn_1>
  ...
  <interface_defn_n>
```

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

- **parameter_definitions**: represents one or more names of valid TOSCA parameter definitions.
- **interface_defn_***: represents one or more valid TOSCA interface definitions.

A.4.11.3 Examples

A.4.11.3.1 *Declaration of valid interface type names*

```
interfaces: [ mytypes.myinterfaces.myOperationsDefn ]
```

A.4.11.3.2 *Declaration of interfaces*

```
interfaces:
   mytypes.myinterfaces.my_node_interfaces:
     my_interface_1:
       # Additional details omitted for brevity
       ...
     my_interface_2:
       # Additional details omitted for brevity
       ...
```

A.4.12 Properties element

The Properties element describes one or more typed Property definitions that can be associated with modelable TOSCA entities (e.g., Node Types, Node Templates, Relationship Types, Artifact Types, etc.). Properties are used by the author to declare the "desired state" of that entity when initially deployed. 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).

A.4.12.1 Keyname

The following keyname is used to declare a list of properties within the TOSCA Simple Profile specification:

```
properties
```

A.4.12.2 Grammar

The properties element is described as a YAML block collection that contains a list of property definitions:

```
<some_typed_entity_name>:
  properties:
    <property_defn_1>
    ...
    <property_defn_n>
```

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

- **some_typed_entity_name**: represents the name of a typed TOSCA entity (e.g., a Node Type, Node Template, Relationship Type, etc.) that has, as part of its definition, a list of properties.
- **property_defn_***: represents one or more property definitions for the associated entity.

A.4.12.3 Example

The following example shows property definitions being associated to a Node Type:

```
my_app_node_type:
  derived_from: tosca.nodes.Root
  properties:
    stylesheet:
      type: string
      default: basic.css
    max_connections:
      type: integer
      required: false
```

A.4.13 Attributes element

The Attributes element describes one or more typed Attribute definitions that can be associated with a modelable TOSCA entity (e.g., Node Types, Relationship Types, etc.). Attributes are used by the author to provide access the “actual state” of certain properties of TOSCA entities at any point in their lifecycle once instantiated (i.e., post deployment). TOSCA orchestrators automatically create Attribute definitions for any Property definitions declared on the same TOSCA entity in order to make accessible the actual (i.e., the current state) value from the running instance of the entity.

A.4.13.1 Keyname

The following keyname is used to declare a list of attributes within the TOSCA Simple Profile specification:

```
attributes
```

A.4.13.2 Grammar

The attributes element is described as a YAML block collection that contains a list of attribute definitions:

```
<some_typed_entity_name>:
  attributes:
    <attribute_defn_1>
    ...
    <attribute_defn_n>
```

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

- **some_typed_entity_name**: represents the name of a typed TOSCA entity (e.g., a Node Type, Relationship Type, etc.) that has, as part of its definition, a list of attributes.
- **attribute_defn_***: represents one or more attribute definitions for the associated entity.

A.4.13.3 Example

The following example shows attribute definitions being associated to a Node Type:

```
my_app_node_type:
  derived_from: tosca.nodes.Root
  attributes:
    instanceId:
      type: string
    max_connections:
      type: integer
```

A.4.14 Schema definition

A schema definition defines the schema for a new named or anonymous type in TOSCA. The schema can be derived from an existing type and may provide additional properties or constraints.

A.4.14.1 Keynames

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

Keyname	Required	Type	Description
description	no	string	The optional description for the schema.
type	no	string	The optional key used when the schema is based upon, but not extend, an existing TOSCA type.
derived_from	no	string	The optional key used when a schema is derived from an existing TOSCA type and will be extended with additional properties.
constraints	no	constraints	The optional list of sequenced constraints for the schema type.
properties	no	properties	The required key used when the schema definition is used to declare a complex type and comprised of a set of valid property definitions.

A.4.14.2 Grammar

Schema definitions have the following grammar:

```
description: <schema_description>
type: <existing_type_name>
```

```
derived_from: <existing_type_name>
constraints:
  <type_constraints>
properties:
  <property_definitions>
```

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

- **schema_description:** represents the optional description for the schema.
- **existing_type_name:** represents the optional name of a valid TOSCA type declaration this new schema would be based or derive from.
- **type_constraints**: represents the optional sequenced list of one or more constraint clauses that restrict the schema's declared type.
- **property_definitions**: represents one or more property definitions that (together) comprise the schema for the schema definition.

A.4.14.3 Additional Requirements

- Schema definitions MAY have either the **type** keyname or **derived_from** keyname, but not both.
- Any **constraint** clauses SHALL be type-compatible with the type declared by the schema's **type** or **derived_from** keynames.
- The **properties** keyname SHALL only be used in conjunction with the **derived_from** keyname.
- If a **properties** keyname is provided, it SHALL contain one or more valid property definitions.

A.4.14.4 Examples

A.4.14.4.1 *Entry schema based upon a simple type*

The following example represents a map entry schema definition based upon an existing string type:

```
# Example: Entry schema for a list of emails using an existing string type
<some_entity>:
  ...
  properties:
    emails:
      type: map
      entry_schema:
        description: basic email
        type: string
        constraints:
          - max_length: 128
```

A.4.14.4.2 *Complex entry schema example*

The following example represents a list's entry schema definition for contact information:

```
# Example: Contact information described as a complex entry schema
```

```
<some_entity>:
  ...
  properties:
    contacts:
      type: list
      entry_schema:
        description: simple contact information
        properties:
          contact_name:
            type: string
          contact_email:
            type: string
          contact_phone:
            type: string
            required: false
```

A.4.15 Datatype definition

A datatype definition defines the schema for a new named datatype in TOSCA.

A.4.15.1 Grammar

Datatype definitions have the following grammar:

```
<datatype_name>:
  <schema_definition>
```

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

- **datatype_name**: represents the required symbolic name of the datatype being declared.
- **schema_definition**: represents the required schema definition for the datatype.

A.4.15.2 Additional Requirements

- None

A.4.15.3 Examples

The following example represents a datatype definition based upon an existing string type:

A.4.15.3.1 *Defining a complex datatype*

```
# define a new complex datatype
mytypes.phone.number:
  properties:
    countrycode:
      type: int
    areacode:
      type: int
    number:
      type: int
```

A.4.15.3.2 *Defining a datatype derived from an existing datatype*

```
# define a new datatype that derives from existing type and extends it
mytypes.phone.entry:
  derived_from: phonenumber
  properties:
    phone_description:
      type: string
      constraints:
        - max_length: 128
```

A.4.16 Capability definition

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

A.4.16.1 Keynames

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

Keyname	Type	Description
type	string	The required name of the Capability Type the capability definition is based upon.
description	description	The optional description of the Capability Type.
properties	properties	An optional list of property definitions for the capability definition.

A.4.16.2 Grammar

Named capability definitions have one of the following grammars:

A.4.16.2.1 *Short notation*

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

```
<capability_defn_name>: <capability_type>
```

A.4.16.2.2 *Extended notation*

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

```
<capability_defn_name>:
   type: <capability_type>
   description: <capability_defn_description>
```

```
  properties:
    <property_definitions>
  attributes:
    <attribute_definitions>
```

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

- **capability_defn_name:** represents the name of a capability definition as a string.
- **`capability_type`**: represents the required capability type the capability definition is based upon.
- **`capability_defn_description`**: represents the optional description of the capability definition.
- **`property_definitions`**: represents the optional list of property definitions for the capability definition.
- **`attribute_definitions`**: represents the optional list of attribute definitions for the capability definition.

A.4.16.3 Examples

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

A.4.16.3.1 *Simple notation example*

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

A.4.16.3.2 *Full notation example*

```
# Full notationnotation, augmenting properties of the referenced capability type
some_capability:
  type: mytypes.mycapabilities.MyCapabilityTypeName
  properties:
    limit: 100
```

A.4.16.4 Notes

- `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.

A.4.17 Capabilities element

The Capabilities element is used to associate one or more typed Capability definitions with a TOSCA Node Type or Node Template.

A.4.17.1 Keyname

The following keyname is used to declare a list of capabilities within the TOSCA Simple Profile specification:

```
capabilities
```

A.4.17.2 Grammar

The capabilities element is described by a YAML block collection that contains a list of capability definitions:

```
<some_typed_entity_name>:
  capabilities:
    <capability_definition_1>
    ...
    <capability_definition_n>
```

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

- **some_typed_entity_name:** represents the name of a typed TOSCA entity (e.g., a Node Type, Node Template) that has, as part of its definition, a list of capabilities.
- **capability_definition_*:** represents one or more Capability definitions for the associated entity.

A.4.17.3 Example

The following examples show capability definitions in both simple and full forms being associated to Node Types:

```
my_node_type_1:
  # Other keys omitted here for sake of brevity
  capabilities:
    app_container: mytypes.mycapabilities.AppContainer
    app_endpoint:
      type: mytypes.mycapabilities.Endpoint
      properties:
        timeout: 300
```

A.4.17.4 Notes

- This element directly maps to the **Capabilities** element defined as part of the schema for the Node Template entity as defined in the TOSCA v1.0 specification.

A.4.18 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 orchestrator can use to fulfil the capability at runtime (implicitly).

A.4.18.1 Keynames

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

Keyname	Type	Required	Description
node	string	no	The optional reserved keyname used to provide the name of a Node Type or Node Template that can fulfil the target node requirement.
capability	string	no	The optional reserved keyname used to provide the name of the capability within target node of the relationship that the associated requirement fulfills.
relationship	string	no	The optional reserved keyname used to provide a named Relationship Type to use when fulfilling the associated named requirement. Please note that this is the “simple form” for the relationship portion of the requirement. If the relationship needs to be further described or augmented, then the extended form of relationship (described below) MUST be used.
target_filter	node filter	no	The optional filter definition that TOSCA orchestrators would use to select the correct target node to fulfill the associated requirement.

A.4.18.2 Extended relationship grammar for the requirement definition

The following are recognized keynames that may be used when the relationship keyname of the requirement definition needs to provide extended information (i.e., it cannot be expressed in a simple, one-line grammar):

Keyname	Type	Required	Description
type	string	no	The optional reserved keyname used to provide the name of a Relationship Type that should be used to fulfil the target node requirement.
interfaces	N/A	no	The optional reserved keyname used to declare or augment relationship interfaces, their operations, implementations and properties.

A.4.18.3 Grammar

Named requirement definitions have one of the following grammars:

A.4.18.3.1 *Short notation (node only):*

The following single-line grammar may be used when only a target node is needed to describe the requirement:

```
<requirement_name>: <node_type_or_template_name>
```

A.4.18.3.2 *Short notation (with relationship or capability):*

The following grammar would be used if either a relationship or capability is needed to describe the requirement:

```
<requirement_name>:
  node: <node_type_or_template_name>
  capability: <capability_type_or_template_name>
  relationship: <relationship_type_or_template_name>
```

A.4.18.3.3 *Extended notation:*

The following grammar would be used if additional target node filtering would be needed to further clarify the requirement or if additional information would need to be provided to the relationship.

```
<requirement_name>:
  node: <node_type_or_template_name>
  capability: <capability_type_or_template_name>
  relationship:
    type: <relationship_type_or_template_name>
    interfaces:
      <interface_settings>
  target_filter:
    <target_filter_definition>
```

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

- **`requirement_name:`** represents the name of a requirement definition as a string.
- **`node_type_or_template_name:`** represents the name (a string) of a TOSCA Node Type or Node Template that either, by its definition, has an implicit set of capabilities or contains an explicit Capability Type definition that the associated named requirement can be fulfilled by.
- **`relationship_type_or_template_name`**: represents the optional name of an explicit, Relationship Type or Relationship Template definition to be used when relating the node the requirement appears in to another node.
- **`interface_settings`**: represents any optional property (input) settings that would be provided to the interfaces on the relationship when it is realized by the orchestrator.
- **`capability_type_or_template_name`**: represents the optional name of a specific capability type or template within the target Node Type or Template identified by **`node_type_or_template_name`** value to be used when relating it to the source node the requirement appears in.
    - **Note**: This key can be used to assure that when the target node exports more than one capability that could fulfill the requirement the correct capability will be selected by the orchestrator using this name.
- **`target_filter_definition`**: represents the optional node filter TOSCA orchestrators would use to fulfill the requirement for selecting a target node.

A.4.18.4 Requirement definition is a tuple

A requirement definition allows fulfillment to be described using three levels of specificity.

1. Node Type or Node Template
2. Capability Type
3. Relationship Type or Template

The first level allows selection, as shown in both the simple or complex grammar, simply providing the node's type or template name using the **`node`** keyname. The second level provides the ability to name the specific capability on the target node that the requirement is seeking using the **`capability`** keyname. Finally, the third level is specification of the relationship (type or template) to use when connecting the requirement to the capability using the **`relationship`** keyname.

In addition, a filter, with the keyname **target_filter**, may be provided to allow a flexible description of matching criteria against potential target nodes' properties, capabilities and capabilities' properties. This allows TOSCA orchestrators to help find the "best fit" when selecting among multiple potential target nodes for the expressed requirements.

A.4.18.5 Examples

A.4.18.5.1 *Example 1 – Explicit hosting requirement on a Node Template*

A web application node template named '**my_webapp_node_template**' declares a requirement named '**host**' that needs to be fulfilled by the same named capability on a web server node template named '**my_webserver_node_template**' in the same TOSCA Service Template.

```
# Example of a requirement fulfilled by a specific named node template
node_templates:
  my_webapp_node_template:
    ...
    requirements:
      - host: my_webserver_node_template

  my_webserver_node_template:
    ...
    capabilities:
      host:
        type: tosca.capabilities.Container
```

Please note that in this example, TOSCA orchestrators would relate these two nodes using an implied HostedOn relationship.

A.4.18.5.2 *Example 2 – Abstract hosting requirement on a Node Type*

A web application node template named '**my_webapp_node_template**' declares a requirement named '**host**' that needs to be fulfilled by any node that derives from the node type **WebServer**.

```
# Example of a requirement fulfilled by a specific web server node template
node_templates:
  my_webapp_node_template:
    ...
    requirements:
      - host: tosca.nodes.WebServer
```

A.4.18.5.3 *Example 3 - Requirement on a Capability Type from any node*

A web application node template named '**my_webapp_node_template**' declares a requirement named 'database' that needs to be fulfilled by any node that declares a capability of (or derives from) type **DatabaseEndpoint**.

```
node_templates:
  my_webapp_node_template:
    requirements:
      - database:
          capability: tosca.capabilities.DatabaseEndpoint
```

Please note that in this example, TOSCA orchestrators would relate the two nodes with an implied ConnectsTo relationship type which supports any connections what derive from the `Endpoint` capability type such as DatabaseEndpoint.

A.4.18.5.4 *Example 4:Requirement with Node Template and a custom Relationship Type*

This example is similar to the previous example; however, the connection between the web application and a named database node template (**`my_database`**) and further requires a custom relationship designated by the keyword '**`relationship`**' and having the custom relationship type definition name of '**`my.types.CustomDbConnection`**'.

```
# Example of a (database) requirement that is fulfilled by a node template named
# "my_database", but also requires a custom database connection relationship
my_webapp_node_template:
  requirements:
    - database:
        node: my_database
        capability: DatabaseEndpoint
        relationship: my.types.CustomDbConnection
```

A.4.18.5.5 *Example 5:Requirement for a Compute node with additional selection criteria (filter)*

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 capabilities on the target node when fulfilling the requirement.

```
node_templates:
  mysql:
   type: tosca.nodes.DBMS.MySQL
    properties:
      # omitted here for sake of brevity
    requirements:
      - host: tosca.nodes.Compute
          target_filter:
            properties:
              - num_cpus: { in_range: [ 1, 4 ] }
              - mem_size: { greater_or_equal: 2 }
            capabilities:
              - 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 ] }
```

A.4.18.6 Additional Requirements

- TBD

A.4.18.7 Notes

- This element directly maps to the **`RequirementsDefinition`** of the Node Type entity as defined in the TOSCA v1.0 specification.

A.4.19 Requirements element

The Requirements element is used to associate one or more named Requirement definitions with a TOSCA Node Type or Node Template.

A.4.19.1 Keynames

The following keyname is used to declare a list of requirements within the TOSCA Simple Profile specification:

```
requirements
```

A.4.19.2 Grammar

The requirements element is described by a YAML block collection that contains a ***sequenced*** list of requirement definitions:

```
<some_typed_entity_name>:
  requirements:
    - <requirement_definition_1>
    - ...
    - <requirement_definition_n>
```

A.4.19.3 Example

```
# Example a node template with two named requirements
node_templates:
  my_software_node:
    requirements:
      # Short notation used here for brevity
      - host: tosca.nodes.Compute
      - database: tosca.nodes.Database
      - ...
```

A.4.19.4 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.
    - Note: TOSCA Orchestrators, having a full view of the complete application 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.

A.4.19.5 Notes

- This element directly maps to the **`Requirements`** element defined as part of the schema for the Node Templates entity (as part of a Service Template's Topology Template as defined in the TOSCA v1.0 specification.

A.4.20 Property Filter definition

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

A.4.20.1 Grammar

Property filter definitions have the following grammar:

A.4.20.1.1 *Short notation:*

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

```
<property_name>: <property_constraint_clause>
```

A.4.20.1.2 *Extended notation:*

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>
```

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).

A.4.20.2 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.

A.4.21 Node Filter definition

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

A.4.21.1 Keynames

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

Keyname	Required	Type	Description
properties	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 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.
<capability name_or_type> properties	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.

A.4.21.2 Grammar

Node filter definitions have the 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>
```

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.
- **property_constraint_clause_*:** represents constraint clause(s) that would be used to filter entities based upon property 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.

A.4.21.3 Additional Requirements

- None

A.4.21.4 Example

The following example is a filter that would be used to select a TOSCA Compute node based upon the values of its properties and also values on 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). In addition, the Compute node must support an encryption capability of type **mytypes.capabilities.compute.encryption** which has properties that support a specific (aes) encryption **algorithm** and **keylength** (128).

```
target_filter:
  properties:
    - num_cpus: { in_range: [ 1, 4 ] }
    - mem_size: { greater_or_equal: 2 }
  capabilities:
    - mytypes.capabilities.compute.encryption:
        properties:
          - algorithm: { equal: aes }
          - keylength: { valid_values: [ 128, 256 ] }
```

A.4.22 Artifact Type

An Artifact Type is a reusable entity that defines the type of one or more files which Node Types or Node Templates can have dependent relationships and used during operations such as during installation or deployment.

A.4.22.1 Keynames

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

Keyname	Definition/Type	Description
derived_from	string	An optional parent Artifact Type name the Artifact Type derives from.
description	description	An optional description for the Artifact Type.

Keyname	Definition/Type	Description
mime_type	string	The required mime type property for the Artifact Type.
file_ext	string[]	The required file extension property for the Artifact Type.
properties	properties	An optional list of property definitions for the Artifact Type.

A.4.22.2 Grammar

```
<artifact_type_name>:
  derived_from: <parent_artifact_type_name>
  description: <artifact_description>
  mime_type: <mime_type_string>
  file_ext: [ <file_extension_1>, ..., <file_extension_n> ]
  properties:
    <property_definitions>
```

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

- **artifact_type_name**: represents the name of the Artifact Type being declared as a string.
- **parent_artifact_type_name**: represents the name of the Artifact Type this Artifact Type definition derives from (i.e., its "parent" type).
- **artifact_description**: represents the optional description string for the corresponding **artifact_type_name**.
- **mime_type_string**: represents the Multipurpose Internet Mail Extensions (MIME) standard string value that describes the file contents for this type of artifact as a string.
- **file_extension_***: represents one or more recognized file extensions for this type of artifact as strings.
- **property_definitions**: represents the optional list of property definitions for the artifact type.

A.4.22.3 Examples

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

A.4.23 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.

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

A.4.24 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 other node.

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

Keyname	Definition/Type	Description
derived_from	string	An optional parent capability type name this new capability type derives from.
description	description	An optional description for the capability type.
properties	properties	An optional list of property definitions for the capability type.

A.4.24.1 Grammar

```
<capability_type_name>:
  derived_from: <parent_capability_type_name>
  description: <capability_description>
  properties:
    <property_definitions>
```

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

- **capability_type_name**: represents the required symbolic 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).
- **capability_description**: represents the optional description string for the corresponding **capability_type_name**.
- **property_definitions**: represents an optional list of property definitions that the capability type exports.

A.4.24.2 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
```

A.4.25 Relationship Type

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

A.4.25.1 Keynames

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

Keyname	Definition/Type	Description
derived_from	string	An optional parent Relationship Type name the Relationship Type derives from.
description	description	An optional description for the Relationship Type.
properties	properties	An optional list of property definitions for the Relationship Type.
attributes	attributes	An optional list of attribute definitions for the Relationship Type.
interfaces	interfaces	An optional list of named interfaces for the Relationship Type.
valid_targets	string[]	A required list of one or more valid target entities or entity types (i.e., a Node Types or Capability Types)

A.4.25.2 Grammar

```
<relationship_type_name>:
  derived_from: <parent_relationship_type_name>
  description: <relationship_description>
  properties:
    <property_definitions>
  attributes:
    <attribute_definitions>
  interfaces: <interface_definitions>
  valid_targets: [ <entity_name_or_type_1>, ..., <entity_name_or_type_n> ]
```

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**.
- **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 named interface definitions supported by the Relationship Type.
- **entity_name_or_type_***: represents one or more valid target (types) for the relationship (e.g., Node Types, Capability Types, etc.).

A.4.25.3 Best Practices

- The TOSCA Root relationship type (tosca.relationships.Root) provides a standard configuration interface (tosca.interfaces.relationship.Configure) that SHOULD be used where possible when defining new relationships types.

A.4.25.4 Examples

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

A.4.26 Relationship Template definition

A Relationship Template specifies the occurrence of a manageable relationship between node templates as part of an application's topology model which 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.

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

Keyname	Definition/Type	Description
type	string	The required name of the Relationship Type the Relationship Template is based upon.
alias	string	The optional name of a different Relationship Template definition whose values are (effectively) copied into the definition for this Relationship Template (prior to any other overrides).
description	description	An optional description for the Relationship Template.
properties	properties	An optional list of property definitions for the Relationship Template.
interfaces	interfaces	An optional list of named interfaces for the Node Template.

A.4.26.1 Grammar

```
<relationship_template_name>:
  type: <relationship_type_name>
  description: <relationship_type_description>
  properties:
    <property_definitions>
  attributes:
    <attribute_definitions>
  interfaces:
    <interface_definitions>
```

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

- **relationship_template_name**: represents the required symbolic name of the Relationship Template being declared.
- **relationship_type_name**: represents the name of the Relationship Type the Relationship Template is based upon.
- **relationship_template_description**: represents the optional description string for the Relationship Template.
- **property_definitions**: represents the optional list of property definitions for the Relationship Template that augment those provided by its declared Relationship Type.

- **interface_definitions**: represents the optional list of interface definitions for the Relationship Template that augment those provided by its declared Relationship Type.

A.4.26.2 Example

```
relationship_templates:
```

A.4.27 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.*

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

Keyname	Definition/Type	Description
derived_from	string	An optional parent Node Type name this new Node Type derives from.
description	description	An optional description for the Node Type.
properties	properties	An optional list of property definitions for the Node Type.
attributes	attributes	An optional list of attribute definitions for the Node Type.
requirements	requirements	An optional *sequenced* list of requirement definitions for the Node Type.
capabilities	capabilities	An optional list of capability definitions for the Node Type.
interfaces	interfaces	An optional list of named interfaces for the Node Type.
artifacts	artifacts	An optional *sequenced* list of named artifact definitions for the Node Type.

A.4.27.1 Grammar

```
<node_type_name>:
  derived_from: <parent_node_type_name>
  description: <node_type_description>
  properties:
    <property_definitions>
  attributes:
    <attribute_definitions>
  requirements:
    <requirement_definitions>
  capabilities:
    <capability_definitions>
  interfaces: <interface_definitions>
  artifacts:
    <artifact_definitions>
```

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

- **node_type_name**: represents the required symbolic name of the Node Type being declared.

- **`parent_node_type_name`**: represents the name (string) of the Node Type this Node Type definition derives from (i.e., its “parent” type).
- **`node_type_description`**: represents the optional description string for the corresponding **`node_type_name`**.
- **`property_definitions`**: represents the optional list of property definitions for the Node Type.
- **`attribute_definitions`**: represents the optional list of attribute definitions for the Node Type.
- **`requirement_definitions`**: represents the optional *sequenced* list of requirement definitions for the Node Type.
- **`capability_definitions`**: represents the optional list of capability definitions for the Node Type.
- **`interface_definitions`**: represents the optional list of one or more named interface definitions supported by the Node Type.
- **`artifact_definitions`**: represents the optional list of artifact definitions for the Node Template that augment those provided by its declared Node Type.

A.4.27.2 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.

A.4.27.3 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
    my_app_port:
      type: integer
      description: application port number
  requirements:
    - host: tosca.nodes.Compute
  interfaces: [ Standard ]
```

A.4.28 Node Template definition

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.

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

Keyname	Definition/Type	Description
type	string	The required name of the Node Type the Node Template is based upon.
description	description	An optional description for the Node Template.
properties	properties	An optional list of property definitions for the Node Template.
attributes	attributes	An optional list of attribute definitions for the Node Template.
requirements	requirements	An optional *sequenced* list of requirement definitions for the Node Template.
capabilities	capabilities	An optional list of capability definitions for the Node Template.
interfaces	interfaces	An optional list of named interfaces for the Node Template.
artifacts	artifacts	An optional *sequenced* list of named artifact definitions for the Node Template.

A.4.28.1 Grammar

```
<node_template_name>:
  type: <node_type_name>
  description: <node_template_description>
  properties:
    <property_definitions>
  attributes:
    <attribute_definitions>
  requirements:
    <requirement_definitions>
  capabilities:
    <capability_definitions>
  interfaces:
    <interface_definitions>
  artifacts:
    <artifact_definitions>
```

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

- **node_template_name**: represents the required symbolic name of the Node Template being declared.
- **node_type_name**: represents the name of the Node Type the Node Template is based upon.
- **node_template_description**: represents the optional description string for Node Template**.**
- **property_definitions**: represents the optional list of property definitions for the Node Template that augment those provided by its declared Node Type.
- **requirement_definitions**: represents the optional *sequenced* list of requirement definitions for the Node Template that augment those provided by its declared Node Type.
- **capability_definitions**: represents the optional list of capability definitions for the Node Template that augment those provided by its declared Node Type.
- **interface_definitions**: represents the optional list of interface definitions for the Node Template that augment those provided by its declared Node Type.
- **artifact_definitions**: represents the optional list of artifact definitions for the Node Template that augment those provided by its declared Node Type.

A.4.28.2 Example

```
node_templates:
  mysql:
    type: tosca.nodes.DBMS.MySQL
    properties:
      dbms_password: { get_input: my_mysql_rootpw }
      dbms_port: { get_input: my_mysql_port }
    requirements:
      - host: db_server
    interfaces:
      Standard:
        configure: scripts/my_own_configure.sh
```

A.5 Service Template

A TOSCA Definitions 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 (i.e., YAML keys), which are allowed to appear in a TOSCA Definitions YAML document.

A.5.1 Keynames

A TOSCA Definitions file contains the following element keynames:

Keyname	Required	Description
tosca_definitions_version	yes	Defines the version of the TOSCA Simple Profile specification the template (grammar) complies with.
tosca_default_namespace	no	Defines the namespace of the TOSCA schema to use for validation.
template_name	no	Declares the name of the template.
template_author	no	Declares the author(s) of the template.
template_version	no	Declares the version string for the template.
description	no	Declares a description for this Service Template and its contents.
imports	no	Declares import statements external TOSCA Definitions documents (files).
dsl_defintions	no	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.
datatype_definitions	no	Declares a list of optional TOSCA datatype definitions.
topology_template	no	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.
node_types	no	This section contains a set of node type definitions for use in service templates. Such type definitions may be used within the node_templates section of the same file, or a TOSCA Definitions file may also just contain node type definitions for use in other files.

Keyname	Required	Description
relationship_types	no	This section contains a set of relationship type definitions for use in service templates. Such type definitions may be used within the same file, or a TOSCA Definitions file may also just contain relationship type definitions for use in other files.
capability_types	no	This section contains an optional list of capability type definitions for use in service templates. Such type definitions may be used within the same file, or a TOSCA Definitions file may also just contain capability type definitions for use in other files.
artifact_types	no	This section contains an optional list of artifact type definitions for use in service templates. Such type definitions may be used within the same file, or a TOSCA Definitions file may also just contain capability type definitions for use in other files.

1273 A.5.2 Grammar

1274 The overall structure of a TOSCA Service Template and its top-level key collations using the TOSCA
1275 Simple Profile is shown below:

```
tosca_definitions_version: # Required TOSCA Definitions version string
tosca_default_namespace:   # Optional. default namespace (schema, types version)
template_name:             # Optional name of this service template
template_author:           # Optional author of this service template
template_version:          # Optional version of this service template

description: A short description of the definitions inside the file.

imports:
  # list of import statements for importing other definitions files

dsl_definitions:
  # list of YAML alias anchors (or macros)

datatype_definitions:
  # list of TOSCA datatype definitions

topology_template:
  # topology template definition of the cloud application or service

node_types:
  # list of node type definitions

capability_types:
  # list of capability type definitions
```

```
relationship_types:
  # list of relationship type definitions

artifact_types:
  # list of artifact type definitions
```

A.5.2.1 Notes

- None

A.5.3 Top-level key definitions

A.5.3.1 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.

A.5.3.1.1 *Keyword*

```
tosca_definitions_version
```

A.5.3.1.2 *Grammar*

Single-line form:

```
tosca_definitions_version: <tosca_simple_profile_version>
```

A.5.3.1.3 *Examples:*

TOSCA Simple Profile version 1.0 specification using the defined namespace alias (see Section A.1):

```
tosca_definitions_version: tosca_simple_yaml_1_0_0
```

TOSCA Simple Profile version 1.0 specification using the fully defined (target) namespace (see Section A.1):

```
tosca_definitions_version: http://docs.oasis-open.org/tosca/simple/1.0
```

A.5.3.2 template_name

This optional element declares the optional name of service template as a single-line string value.

A.5.3.2.1 *Keyword*

```
template_name
```

A.5.3.2.2 *Grammar*

```
template_name: <name string>
```

A.5.3.2.3 *Example*

```
template_name: My service template
```

A.5.3.2.4 *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.

A.5.3.3 template_author

This optional element declares the optional author(s) of the service template as a single-line string value.

A.5.3.3.1 *Keyword*

```
template_author
```

A.5.3.3.2 *Grammar*

```
template_author: <author string>
```

A.5.3.3.3 *Example*

```
template_author: My service template
```

A.5.3.4 template_version

This element declares the optional version of the service template as a single-line string value.

A.5.3.4.1 *Keyword*

```
template_version
```

A.5.3.4.2 *Grammar*

```
template_version: <version>
```

A.5.3.4.3 *Example*

```
template_version: 2.0.17
```

A.5.3.4.4 *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.

A.5.3.5 description

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

A.5.3.5.1 *Keyword*

```
description
```

A.5.3.6 imports

This optional element 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.

A.5.3.6.1 *Keyword*

```
imports
```

A.5.3.6.2 *Grammar*

```
imports:
   - <tosca_definitions_file_1>
   - ...
   - <tosca_definitions_file_n>
```

A.5.3.6.3 *Example*

```
# An example import of definitions files from a location relative to the
# file location of the service template declaring the import.

imports:
  - relative_path/my_defns/my_typesdefs_1.yaml
  - ...
  - relative_path/my_defns/my_typesdefs_n.yaml
```

A.5.3.7 dsl_definitions

This optional element provides a section to define macros (e.g., YAML-style macros when using the TOSCA Simple Profile in YAML specification).

A.5.3.7.1 *Keyword*

```
dsl_definitions
```

A.5.3.7.2 *Grammar*

```
dsl_definitions:
   <dsl_definitions_1>
   ...
   <dsl_definitions_n>
```

A.5.3.7.3 *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
```

A.5.3.8 datatype_definitions

This optional element provides a section to define new datatypes in TOSCA.

A.5.3.8.1 *Keyword*

```
datatype_definitions
```

A.5.3.8.2 *Grammar*

```
datatype_definitions:
   <tosca_datatype_def_1>
   ...
   <tosca_datatype_def_n>
```

A.5.3.8.3 *Example*

```
datatype_definitions:
  # 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
```

A.5.3.9 node_types

This element lists the Node Types that provide the reusable type definitions for software components that Node Templates can be based upon.

A.5.3.9.1 *Keyword*

```
node_types
```

A.5.3.9.2 *Grammar*

```
node_types:
  <node_types_defn_1>
  ...
  <node_type_defn_n>
```

A.5.3.9.3 *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
```

A.5.3.9.4 *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.

A.5.3.10 relationship_types

This element lists the Relationship Types that provide the reusable type definitions that can be used to describe dependent relationships between Node Templates or Node Types.

A.5.3.10.1 *Keyword*

```
relationship_types
```

A.5.3.10.2 *Grammar*

```
relationship_types:
  <relationship_type_defn_1>
  ...
  <relationship type_defn_n>
```

A.5.3.10.3 *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 ...
```

A.5.3.11 capability_types

This element 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.

A.5.3.11.1 *Keyword*

```
capability_types
```

A.5.3.11.2 *Grammar*

```
capability_types:
  <capability_type_defn_1>
  ...
  <capability type_defn_n>
```

A.5.3.11.3 *Example*

```
capability_types:
  mycompany.mytypes.myCustomEndpoint:
    derived_from: tosca.capabilities.Endpoint
    properties:
      # more details ...

  mycompany.mytypes.myCustomFeature:
    derived_from: tosca.capabilites.Feature
    properties:
      # more details ...
```

A.6 topology_template

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 elements.

A.6.1 Grammar

The overall grammar of the **topology_template** section is shown below.–Detailed grammar definitions of the each sub-sections are provided in subsequent subsections.

```
topology_template:
  description:
    # a description of the topology template

  inputs:
    # definition of input parameters for the topology template

  node_templates:
    # definition of the node templates of the topology

  relationship_templates:
    # definition of the relationship templates of the topology

  outputs:
    # definition of output parameters for the topology template
```

A.6.1.1 inputs

The **inputs** section provides a means to define parameters, 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.

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)

A.6.1.1.1 *Grammar*

The grammar of the **inputs** section is as follows:

```
inputs:
  <property_definition_1>
  ...
  <property_definition_n>
```

A.6.1.1.2 *Examples*

This section provides a set of examples for the single elements of a topology template.

Simple **inputs** example without any constraints:

```
inputs:
  fooName:
    type: string
    description: Simple string typed property definition with no constraints.
    default: bar
```

Example of **inputs** with constraints:

```
inputs:
  SiteName:
    type: string
    description: string typed property definition with constraints
    default: My Site
    constraints:
      - min_length: 9
```

A.6.1.2 node_templates

The **node_templates** section lists the Node Templates that describe the (software) components that are used to compose cloud applications.

A.6.1.2.1 *grammar*

The grammar of the **node_templates** section is a follows:

```
node_templates:
  <node_template_defn_1>
  ...
  <node_template_defn_n>
```

A.6.1.2.2 *Example*

Example of **node_templates** section:

```
node_templates:
  my_webapp_node_template:
    type: WebApplication

  my_database_node_template:
    type: Database
```

A.6.1.3 relationship_templates

The **relationship_templates** section lists the Relationship Templates that describe the relations between components that are used to compose cloud applications.

Note that in the TOSCA Simple Profile, the explicit definition of relationship templates as it was required in TOSCA v1.0 is optional, since relationships between nodes get implicitly defined by referencing other node templates in the requirements sections of node templates.

A.6.1.3.1 *Grammar*

The grammar of the **relationship_templates** section is as follows:

```
relationship_templates:
  <relationship_template_defn_1>
  ...
  <relationship_template_defn_n>
```

A.6.1.3.2 *Example*

Example of **relationship_templates** section:

```
relationship_templates:
```

A.6.1.4 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.

A.6.1.4.1 *Grammar*

The grammar of the **outputs** section is as follows:

```
outputs:
  <property_definitions>
```

A.6.1.4.2 *Example*

Example of **ouputs** section:

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

A.6.1.5 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.

A.6.1.5.1 *Grammar*

The grammar of the **groups** section is as follows:

```
groups:
  <group_name_A>:
    <node_template_defn_A_1>
    ...
    <node_template_defn_A_n>

  <group_name_B>
    <node_template_defn_B_1>
    ...
    <node_template_defn_B_n>
```

A.6.1.5.2 *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:
  server_group_1:
    members: [ server1, server2 ]
    policies:
      - anti_collocation_policy:
          # specific policy declarations omitted, as this is not yet specified
```

A.6.2 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.
    - Note, however, that TOSCA v1.0 does not define a direction (input vs. output) for those mappings, i.e. TOSCA v1.0 **PropertyMappings** are underspecified in that respect and TOSCA Simple Profile's **inputs** and **outputs** provide a more concrete definition of input and output parameters.

Appendix B. Functions

This section includes functions that are supported for use within a TOSCA Service Template.

B.1 Reserved Function Keywords

The following keywords MAY be used in some TOSCA function in place of a TOSCA Node or Relationship Template name. They will be interpreted by a TOSCA orchestrator 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.

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.

B.2 Environment Variable Conventions

B.2.1 Reserved Environment Variable Names and Usage

TOSCA orchestrators utilize certain reserved keywords in the execution environments that implementation artifacts for Node or Relationship Templates operations are executed in. They are used to provide information to these implementation artifacts such as the results of TOSCA function evaluationl or information about the instance model of the TOSCA application

The following keywords are reserved environment variable names in any TOSCA supported execution environment:

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.
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.
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.
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.

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.

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.

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:

```
node_templates:
  load_balancer:
    type: some.vendor.LoadBalancer
    requirements:
      - member:
          relationship: some.vendor.LoadBalancerToMember
```

```
            interfaces:
              tosca.interfaces.relationships.Configure:
                add_target:
                  inputs:
                    member_ip: { get_attribute: [ TARGET, ip_address ] }
                  implementation: scripts/configure_members.py
```

The **add_target** operation will be invoked, whenever a new target member is being added to the load-balancer. 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.

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 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
```

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 ...
```

B.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.

B.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).

B.3 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.

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).

B.3.1 get_input

The **`get_input`** function is used to retrieve the values of properties declared within the **`inputs`** section of a TOSCA Service Template.

B.3.1.1 Grammar

```
get_input: <input_property_name>
```

B.3.1.2 Parameters

Parameter	Required	Type	Description
<input_property_name>	yes	string	The name of the property as defined in the **`inputs`** section of the service template.

B.3.1.3 Examples

```
inputs:
  cpus:
    type: integer

node_templates:
  my_server:
    type: tosca.nodes.Compute
    properties:
      num_cpus: { get_input: cpus }
```

B.3.2 get_property

The **get_property** function is used to retrieve property values between modelable entities defined in the same service template.

B.3.2.1 Grammar

```
get_property: <modelable_entity_name>, [<req_or_cap_name>], <property_name> [,
<nested_property_name_1>, ..., <nested_property_name_*> ]
```

B.3.2.2 Parameters

Parameter	Required	Type	Description
<modelable entity name> | SELF | SOURCE | TARGET | HOST	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.
<req_or_cap_name>	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.
<property_name>	yes	string	The name of the property definition the function will return the value from.
<nested_property_name_1> | nested_property_index_1,, ..., <nested_property_name_m> | nested_property_index_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.

B.3.2.3 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:
      db_name: sql_database1
  wordpress:
    type: tosca.nodes.WebApplication.WordPress
    ...
    interfaces:
      Standard:
        configure:
          inputs:
            wp_db_name: { get_property: [ mysql_database, db_name ] }
```

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 ] }
```

The following example shows how to use the get_property function using the TARGET keyword:

```
TBD
```

B.4 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.

B.4.1 get_attribute

The **`get_attribute`** function is used to retrieve the values of named attributes declared by the referenced node or relationship template name.

B.4.1.1 Grammar

```
get_attribute: <modelable_entity_name>, [<req_or_cap_name>], <attribute_name> [,
<nested_attribute_name_1>, ..., <nested_attribute_name_x> ]
```

B.4.1.2 Parameters

Parameter	Required	Type	Description
`<modelable entity name> | SELF | SOURCE | TARGET | HOST`	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.
`<req_or_cap_name>`	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.
`<attribute_name>`	yes	string	The name of the attribute definition the function will return the value from.
`<nested_attribute_name_1> | nested_attribute_index_1, ..., <nested_attribute_name_x> | <nested_attribute_index_x>`	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.

B.4.1.3 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.

B.4.2 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).

Notes:

- 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).

B.5 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.

B.5.1 get_operation_output

The **`get_operation_output`** function is used to retrieve the values of variables exposed / exported from an interface operation.

B.5.1.1 Grammar

```
get_operation_output: <modelable_entity_name>, <interface_name>, <operation_name>,
<output_variable_name>
```

B.5.1.2 Parameters

Parameter	Required	Type	Description
`<modelable entity name> | SELF | SOURCE | TARGET`	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>`	Yes	string	The required name of the interface which defines the operation.
`<operation_name>`	yes	string	The required name of the operation whose value we would like to retrieve.
`<output_variable_name>`	Yes	string	The required name of the variable that is exposed / exported by the operation.

B.5.1.3 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.

B.6 Navigation functions

- This version of the TOSCA Simple Profile does not define any model navigation functions.

B.6.1 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.

B.6.1.1 Grammar

```
get_nodes_of_type: <node_type_name>
```

B.6.1.2 Parameters

Parameter	Required	Type	Description
<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.

B.6.1.3 Returns

B.6.1.4

Return Key	Type	Description
TARGETS	<see above>	The list of node instances from the current application instance that match the **node_type_name** supplied as an input parameter of this function.

B.7 Context-based Entity name (global)

TBD

Goal:

- 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>)

Appendix C. TOSCA normative type definitions

The declarative approach is heavily dependent of the definition of basic types that a declarative container must understand. The definition of these types must be very clear such that the operational semantics can be precisely followed by a declarative container to achieve the effects intended by the modeler of a topology in an interoperable manner.

C.1 Assumptions

- Assumes alignment with/dependence on XML normative types proposal for TOSCA v1.1
- Assumes that the normative types will be versioned and the TOSCA TC will preserve backwards compatibility.
- Assumes that security and access control will be addressed in future revisions or versions of this specification.

C.2 Data Types

C.2.1 tosca.datatypes.network.NetworkInfo

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

C.2.1.1 Properties

Name	Type	Constraints	Description
network_name	string	None	The name of the logical network. e.g., public private admin
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.

C.2.1.2 Definition

The TOSCA NetworkInfo data type is defined as follows:

```
tosca.datatypes.network.NetworkInfo:
  properties:
    network_name:
      type: string
    network_id:
      type: string
    addresses:
      type: list
      entry_schema:
        type: string
```

C.2.1.3 Examples

Example usage of the NetworkInfo data type:

```
private_network:
   network_name: private
   network_id: 3e54214f-5c09-1bc9-9999-44100326da1b
   addresses: [ 10.111.128.10 ]
```

C.2.1.4 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.

C.2.2 tosca.datatypes.network.PortInfo

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

C.2.2.1 Properties

Name	Type	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.

C.2.2.2 Definition

The TOSCA Port type is defined as follows:

```
tosca.datatypes.network.PortInfo:
  properties:
    port_name:
      type: string
    port_id:
      type: string
    network_id:
      type: string
    network_id:
      type: string
    mac_address:
      type: string
    addresses:
      type: list
      entry_schema:
        type: string
```

C.2.2.3 Examples

Example usage of the PortInfo data type:

```
ethernet_port:
   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 ]
```

C.2.2.4 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 NetworkInfo may or may not be required depending on usage context.

C.2.3 tosca.datatypes.network.PortDef

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

C.2.3.1 Definition

The TOSCA PortDef type is defined as follows:

```
tosca.datatypes.network.PortDef:
  type: integer
  constraints:
    - in_range: [ 1, 65535 ]
```

C.2.3.2 Examples

Example use of a PortDef property type:

```
listen_port:
   type: PortDef
   default: 9000
   constraints:
     - in_range [ 9000, 9090 ]
```

C.2.4 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

C.2.4.1 Properties

Name	Required	Type	Constraints	Description
protocol	yes	string	default: tcp	The required protocol used on the port.
source	no	list of integer	integer entries in_range: [1, 65536]	The optional list of source ports.
source_range	no	range	in_range: [1, 65536]	The optional range for source ports.
target	no	list of integer	integer entries in_range: [1, 65536]	The optional list of target ports.
target_range	no	range	in_range: [1, 65536]	The optional range for target ports.

C.2.4.2 Definition

The TOSCA PortSpec type is defined as follows:

```
tosca.datatypes.network.PortSpec:
  properties:
    protocol:
      type: string
      required: true
      default: tcp
      constraints:
        - valid_values: [ udp, tcp, igmp ]
    target:
      type: list
      entry_schema:
        type: PortDef
    target_range:
      type: range
      constraints:
        - in_range: [ 1, 65535 ]
    source:
      type: list
      entry_schema:
        type: PortDef
    source_range:
      type: range
      constraints:
        - in_range: [ 1, 65535 ]
```

C.2.4.3 Additional requirements

- A valid PortSpec must have at least one of the following properties: **`target`**, **`target_range`**, **`source`** or **`source_range`**.

C.2.4.4 Examples

Example usage of the PortSpec data type:

```
# example properties in a node template
some_endpoint:
  properties:
    ports:
      user_port:
        ip_proto: tcp
        target: 50000
        target_range: [ 20000, 60000 ]
        source: 9000
        source_range: [ 1000, 10000 ]
```

C.2.5 tosca.datatypes.network.Credential

The PortSpec 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.network.Credential

C.2.5.1 Properties

Name	Required	Type	Constraints	Description
protocol	yes	string	None	The required protocol name.
token_type	yes	string	None	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.

C.2.5.2 Definition

The TOSCA Credential type is defined as follows:

```
tosca.datatypes.network.Credential:
  properties:
    protocol:
      type: string
    token_type:
      type: string
    token:
      type: string
    keys:
      type: map
      entry_schema:
        type: string
```

C.2.5.3 Notes

- Specific token types and encoding them using network protocols are not defined or covered in this specification.

C.2.5.4 Examples

Example usage of the Credential data type:

C.2.5.4.1 *HTTP Basic access authentication credential*

```
<some_tosca_entity>:
  properties:
    my_credential:
      type: Credential
        properties:
          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
```

C.2.5.4.2 *X-Auth-Token credential*

```
<some_tosca_entity>:
  properties:
    my_credential:
      type: Credential
        properties:
          protocol: xauth
          token_type: X-Auth-Token
          # token encoded in Base64
          token: 604bbe45ac7143a79e14f3158df67091
```

C.2.5.4.3 *OAuth bearer token credential*

```
<some_tosca_entity>:
  properties:
    my_credential:
      type: Credential
        properties:
          protocol: oauth2
          token_type: bearer
          # token encoded in Base64
          token: 8ao9nE2DEjr1zCsicWMpBC
```

C.3 Capabilities Types

C.3.1 tosca.capabilities.Root

This is the default (root) TOSCA Capability Type definition that all other TOSCA Capability Types derive from.

C.3.1.1 Definition

```
tosca.capabilities.Root:
```

C.3.2 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

C.3.2.1 Properties

Name	Type	Constraints	Description
valid_node_types	NodeType[]	None	A list of one or more names of Node Types that are supported as containees that declare the Container type as a Capability.

C.3.2.2 Definition

```
tosca.capabilities.Container:
  derived_from: tosca.capabilities.Root
  properties:
    valid_node_types: [ <node_type_name_1>,..., <node_type_name_n> ]
```

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

- **node_type_name_*:** represents the name of a Node Type definition as a string.

C.3.3 tosca.capabilities.Endpoint

This is the default TOSCA type that should be used or extended to define a network endpoint capability. This includes the information to express a basic endpoint with a single port or a complex endpoint with multiple ports.

Shorthand Name	Endpoint
Type Qualified Name	tosca:Endpoint
Type URI	tosca.capabilities.Endpoint

C.3.3.1 Properties

Name	Required	Type	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	yes	integer	greater_or_equal: 1 less_or_equal: 65535	The port of the endpoint.
secure	no	boolean	default: false	Indicates if the endpoint is a secure endpoint.

Name	Required	Type	Constraints	Description
url_path	no	string	None	The optional URL path of the endpoint's address if applicable for the protocol.
port_name	no	string	None	The optional name (or ID) of the network port this endpoint should be bound to.
network_name	no	string	None	The optional name (or ID) of the network this endpoint should be bound to.
initiator	no	string	one of: • source • target • peer default: source	Indicates the direction of the connection.
ports	yes	map of PortSpec	TBD	TBD look to change to "virtual network interface" (vnic) or something else.

C.3.3.2 Attributes

Name	Required	Type	Constraints	Description
ip_address	yes	string	None	Note: This is the IP address as propagated up by the associated node's host (Compute) container.

C.3.3.3 Definition

```
tosca.capabilities.Endpoint:
  derived_from: tosca.capabilities.Root
  properties:
    protocol:
      type: string
      default: tcp
    port:
      type: integer
      constraints:
        - greater_or_equal: 1
        - less_or_equal: 65535
    secure:
      type: boolean
      default: false
    url_path:
      type: string
      required: false
    port_name:
      type: string
      required: false
    network_name
      type: string
      required: false
    initiator:
      type: string
      default: source
      constraints:
        - valid_values: [ source, target, peer ]
    ports:
      type: map
      required: true
      constraints:
        - min_length: 1
      entry_schema:
        type: PortSpec
  attributes:
    ip_address:
      type: string
```

C.3.4 tosca.capabilities.DatabaseEndpoint

This is the default TOSCA type that should be used or extended to define a specialized database endpoint capability.

Shorthand Name	DatabaseEndpoint
Type Qualified Name	tosca:DatabaseEndpoint
Type URI	tosca.capabilities.DatabaseEndpoint

C.3.4.1 Properties

Name	Required	Type	Constraints	Description
None	N/A	N/A	N/A	N/A

C.3.4.2 Definition

```
tosca.capabilities.DatabaseEndpoint:
  derived_from: tosca.capabilities.Endpoint
```

C.3.5 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

C.3.5.1 Properties

Name	Required	Type	Constraints	Description
N/A	N/A	N/A	N/A	N/A

C.3.5.2 Definition

```
tosca.capabilities.Attachment:
  derived_from: tosca.capabilities.Root
```

C.3.6 tosca.capabilities.OperatingSystem

This is the default TOSCA type that should be used to express a scalability capability for a node.

Shorthand Name	OperatingSystem
Type Qualified Name	tosca:OperatingSystem
Type URI	tosca.capabilities.OperatingSystem

C.3.6.1 Properties

Name	Required	Type	Constraints	Description
architecture	yes	string	None	The Operating System (OS) architecture. Examples of valid values include: x86_32, x86_64, etc.
type	yes	string	None	The Operating System (OS) type. Examples of valid values include: linux, aix, mac, windows, etc.
distribution	no	string	None	TheOperating System (OS) distribution. Examples of valid values for an “type” of “Linux” would include: debian, fedora, rhel and ubuntu.
version	no	string	None	The Operating System version.

C.3.6.2 Definition

```
tosca.capabilities.OperatingSystem:
  derived_from: tosca.capabilities.Root
  properties:
    min_intances:
      type: integer
      default: 1
    max_intances:
      type: integer
      default: 1
    default_instances:
      type: integer
```

C.3.6.3 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.

C.3.6.4 Notes

- None

C.3.7 tosca.capabilities.Scalable

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

C.3.7.1 Properties

Name	Required	Type	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.
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.

C.3.7.2 Definition

```
tosca.capabilities.Scalable:
  derived_from: tosca.capabilities.Root
  properties:
    min_intances:
      type: integer
      default: 1
    max_intances:
      type: integer
      default: 1
    default_instances:
      type: integer
```

C.3.7.3 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.

C.3.8 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 Name	tosca:network.Bindable
Type URI	tosca.capabilities.network.Bindable

C.3.8.1 Properties

Name	Required	Type	Constraints	Description
N/A	N/A	N/A	N/A	N/A

C.3.8.2 Definition

```
tosca.capabilities.network.Bindable:
  derived_from: tosca.capabilities.Root
```

C.4 Requirement Types

There are no normative Requirement Types currently defined in this working draft. Typically, Requirements are described against a known Capability Type

C.5 Relationship Types

C.5.1 tosca.relationships.Root

This is the default (root) TOSCA Relationship Type definition that all other TOSCA Relationship Types derive from.

C.5.1.1 Attributes

Name	Required	Type	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.

C.5.1.2 Definition

```
tosca.relationships.Root:
  # The TOSCA root relationship type has no property mappings
  interfaces: [ tosca.interfaces.relationship.Configure ]
  valid_targets: [ tosca.capabilities.Root ]
  attributes:
    tosca_id:
      type: string
    tosca_name:
      type: string
```

C.5.2 tosca.relationships.DependsOn

This type represents a general dependency relationship between two nodes.

Shorthand Name	DependsOn
Type Qualified Name	tosca:DependsOn
Type URI	tosca.relationships.DependsOn

C.5.2.1 Definition

```
tosca.relationships.DependsOn:
  derived_from: tosca.relationships.Root
```

C.5.3 tosca.relationships.HostedOn

This type represents a hosting relationship between two nodes.

Shorthand Name	HostedOn
Type Qualified Name	tosca:HostedOn
Type URI	tosca.relationships.HostedOn

C.5.3.1 Definition

```
tosca.relationships.HostedOn:
  derived_from: tosca.relationships.DependsOn
  valid_targets: [ tosca.capabilities.Container ]
```

C.5.4 tosca.relationships.ConnectsTo

This type represents a network connection relationship between two nodes.

Shorthand Name	ConnectsTo
Type Qualified Name	tosca:ConnectsTo
Type URI	tosca.relationships.ConnectsTo

C.5.4.1 Definition

```
tosca.relationships.ConnectsTo:
  derived_from: tosca.relationships.Root
  valid_targets: [ tosca.capabilities.Endpoint ]
```

C.5.4.2 Properties

Name	Required	Type	Constraints	Description
username	no	string		
password	no	string		

C.5.5 tosca.relationships.AttachTo

This type represents an attachment relationship between two nodes. For example, an AttachTo relationship type would be used for attaching a storage node to a Compute node.

Shorthand Name	AttachTo
Type Qualified Name	tosca:AttachTo
Type URI	tosca.relationships.AttachTo

C.5.5.1 Properties

Name	Required	Type	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.
device	no	string	None	The logical device name which for the attached device (which is represented by the target node in the model).

C.5.5.2 Attributes

Name	Required	Type	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.

C.5.5.3 Definition

```
tosca.relationships.AttachTo:
  derived_from: tosca.relationships.Root
  valid_targets: [ tosca.capabilities.Attachment ]
  properties:
    location:
      type: string
      constraints:
        - min_length: 1
    device:
      type: string
      required: false
```

C.6 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.

C.6.1 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”.
- Template designers MAY provide or override code or scripts provided by a type for a specified interface defined for the type (even if the type itself does not provide a script for that operation).

C.6.2 tosca.interfaces.node.lifecycle.Standard

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

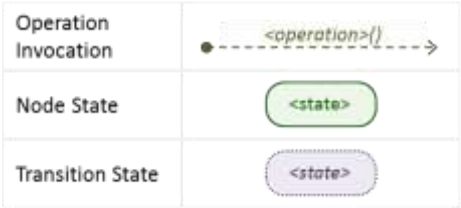
C.6.2.1 Definition

```
tosca.interfaces.node.lifecycle.Standard:
  create:
    description: Standard lifecycle create operation.
  configure:
    description: Standard lifecycle configure operation (pre-start).
  start:
    description: Standard lifecycle start operation.
  stop:
    description: Standard lifecycle stop operation.
  delete:
    description: Standard lifecycle delete operation.
```

C.6.2.2 Operation sequencing and node state

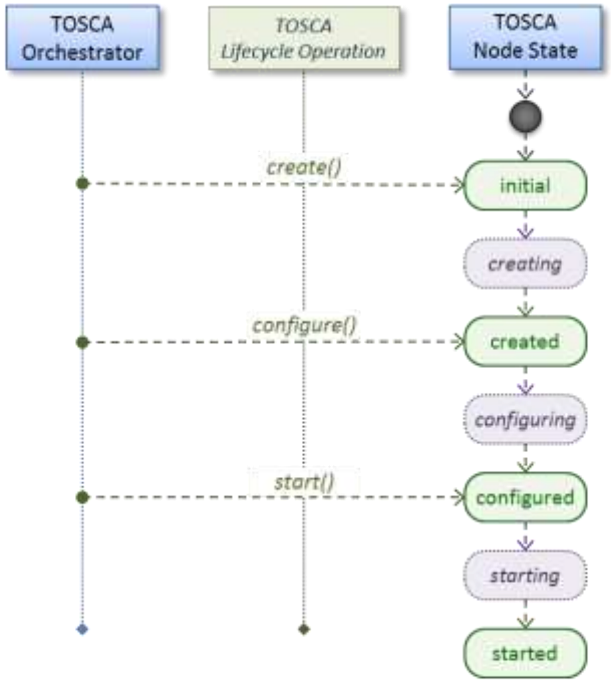
The following diagrams show how TOSCA orchestrators sequence the operations of the Standard lifecycle in normal node startup and shutdown procedures.

The following key should be used to interpret the diagrams:



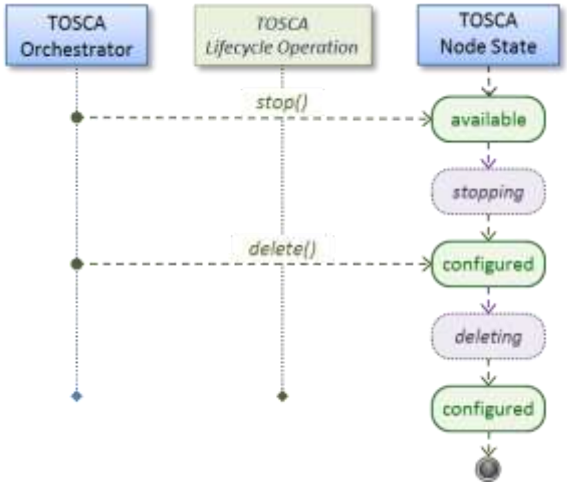
C.6.2.2.1 *Normal node startup sequence diagram*

The following diagram shows how the TOSCA orchestrator would invoke operations on the Standard lifecycle to shut down a node.



C.6.2.2.2 *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.



C.6.3 tosca.interfaces.node.lifecycle.Simple

This interface defines the simplest, normative lifecycle operations that TOSCA nodes may support. It can be used when nodes are able to perform **`create`**, **`configure`**, **`start`** and **`postconfigure`** operations as defined in the Standard lifecycle as a single **`deploy`** operation.

Shorthand Name	Simple
Type Qualified Name	tosca:Simple
Type URI	tosca.interfaces.node.lifecycle.Simple

C.6.3.1 Definition

```
tosca.interfaces.node.lifecycle.Simple:
  deploy:
    description: Simple lifecycle deploy operation. This single operation would be
used to implement the Standard lifecycle operations of create, configure, start and
postconfigure.
  start:
    description: Simple lifecycle start operation.
  stop:
    description: Simple lifecycle stop operation.
  delete:
    description: Simple lifecycle delete operation.
```

C.6.3.2 Requirements

- Following the execution of the **deploy** operation; the node MUST be in an **active** node instance state.
- Implementers of the Simple lifecycle interfaces SHALL code valid **start** and **stop** operation implementations.

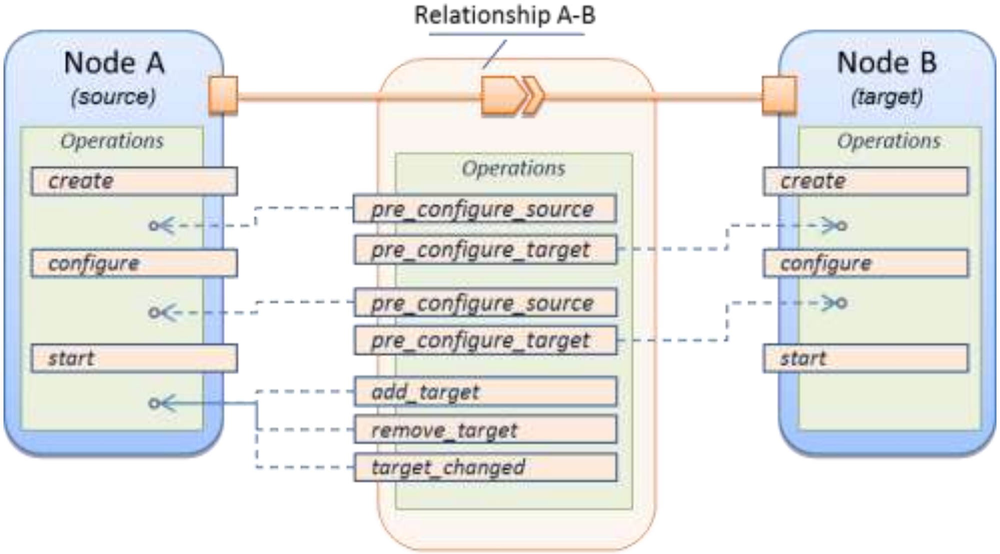
C.6.4 tosca.interfaces.relationship.Configure

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

C.6.4.1 Definition

```
tosca.interfaces.relationship.Configure:
  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:
```



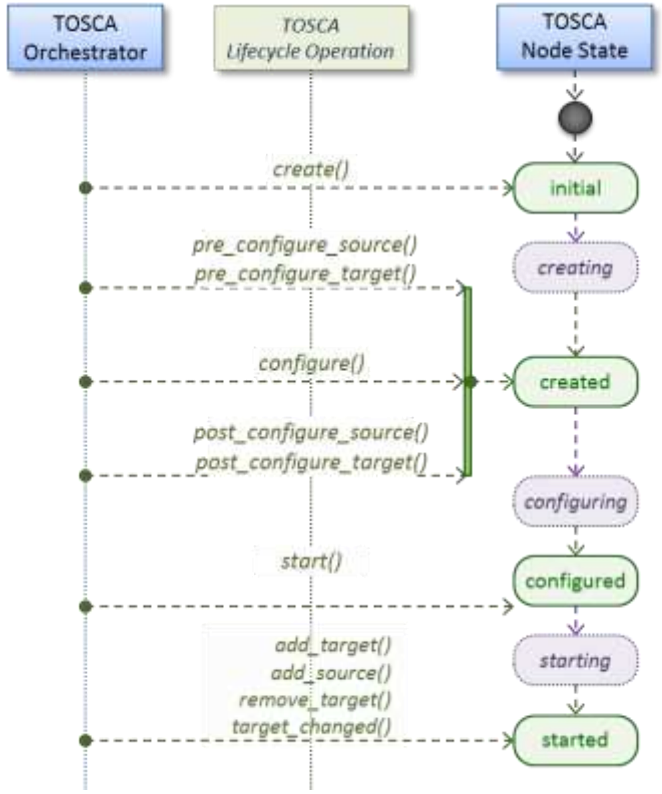
```
    description: Operation to post-configure the target endpoint.
  add_target:
    description: Operation to notify the source node of a target node being added
via a relationship.
  add_source:
    description: Operation to notify the target node of a source node which is now
available via a relationship.
    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.
```

C.6.4.2 Invocation Conventions

TOSCA relationships are directional connecting a source node to a target node. When TOSCA Orchestrator connects a source and target node together using a relationship that supports the Configure interface it will “interleave” the operations invocations of the Configure interface with those of the node’s own Standard lifecycle interface. This concept is illustrated below:

C.6.4.3 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.



C.6.4.3.1 *Node-Relationship configuration sequence*

Depending on which side (i.e., source or target) of a relationship a node is on, the orchestrator will:

1. Invoke either the **pre_configure_source** or **pre-configure_target** operation as supplied by the relationship on the node.
2. Invoke the node's **configure** operation.
3. Invoke either the **post_configure_source** or **post_configure_target** as supplied by the relationship on the node.

Note that the The **pre_configure_xxx** and **post_configure_xxx** are invoked only once per node instance.

C.6.4.3.2 *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.

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.

C.6.4.4 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., 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 a failure code when possible.

C.7 Node Types

C.7.1 tosca.nodes.Root

The TOSCA **`Root`** Node Type is the default type that all other TOSCA base Node Types derive from. This allows for all TOSCA nodes to have a consistent set of features for modeling and management (e.g., consistent definitions for requirements, capabilities and lifecycle interfaces).

C.7.1.1 Properties

Name	Required	Type	Constraints	Description
N/A	N/A	N/A	N/A	The TOSCA Root Node type has no specified properties.

C.7.1.2 Attributes

Name	Required	Type	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 xxx for allowed values.

C.7.1.3 Definition

```
tosca.nodes.Root:
  description: The TOSCA Node Type all other TOSCA base Node Types derive from
  attributes:
    tosca_id:
```

```
    type: string
  tosca_name:
    type: string
requirements:
  - dependency:
      node: tosca.capabilities.Root
interfaces: [ tosca.interfaces.node.lifecycle.Standard |
              tosca.interfaces.node.lifecycle.Simple ]
```

C.7.1.4 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.
- Valid Nodes Types or Node Templates MUST implement either the Standard or Simple lifecycle interfaces, but not both.

C.7.2 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

C.7.2.1 Properties

Name	Required	Type	Constraints	Description
num_cpus	no	integer	greater_or_equal: 1	Number of (actual or virtual) CPUs associated with the Compute node.
disk_size	no	scalar-unit	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	greater_or_equal: 0 MB	Size of memory available to applications running on the Compute node (default unit is MB).

C.7.2.2 Attributes

Name	Required	Type	Constraints	Description
ip_address	no	string	status: **deprecated**	The primary IP address assigned by the cloud provider that applications may use to access the Compute node. **Note**: This is used by the platform provider to convey the primary address used to access the compute node. Future working drafts will address implementations that support floating or multiple IP addresses.

Name	Required	Type	Constraints	Description
networks	no	map(string) of NetworkInfo	None	The list of logical networks assigned to the compute host instance and information about them.
ports	no	map(string) of PortInfo	None	The list of logical ports assigned to the compute host instance and information about them.

C.7.2.3 Definition

```
tosca.nodes.Compute:
  derived_from: tosca.nodes.Root
  properties:
    # compute properties
    num_cpus:
      type: integer
      constraints:
        - greater_or_equal: 1
    disk_size:
      type: scalar-unit
      constraints:
        - greater_or_equal: 0 MB
    mem_size:
      type: scalar-unit
      constraints:
        - greater_or_equal: 0 MB

  attributes:
    # DEPRECATED: Compute node’s primary IP address
    ip_address:
      type: string
      status: deprecated

    networks:
      type: map
      entry_schema:
        type: tosca.datatypes.network.NetworkInfo

    ports:
      type: map
      entry_schema:
        type: tosca.datatypes.network.PortInfo
  capabilities:
    host:
      type: tosca.capabilities.Container
      properties:
        valid_node_types: [tosca.nodes.SoftwareComponent]

    endpoint:
      type: tosca.capabilities.Endpoint

    os:
      type: tosca.capabilites.OperatingSystem

    scalable:
      type: tosca.capabilities.Scalable

    binding:
      type: tosca.capabilities.network.Bindable
```

1820 C.7.3 tosca.nodes.SoftwareComponent

1821 The TOSCA **SoftwareComponent** node represents a generic software component that can be managed 1822 and run by a TOSCA **Compute** Node Type.

Shorthand Name	SoftwareComponent
Type Qualified Name	tosca:SoftwareComponent
Type URI	tosca.nodes.SoftwareComponent

1823 C.7.3.1 Properties

Name	Required	Type	Constraints	Description
component_version	no	version	None	The software component’s version.

1824 C.7.3.2 Attributes

Name	Required	Type	Constraints	Description
ip_address	no	string	**status: deprecated**	The first public IP address assigned to the host Compute node. Default: get_attribute (SELF, host, ip_address)

1825 C.7.3.3 Definition

```
tosca.nodes.SoftwareComponent:
  derived_from: tosca.nodes.Root
```

```
properties:
  # domain-specific software component version
  component_version:
    type: version
    required: false
attributes:
  # Deprecated
  ip_address:
    type: string
    status: deprecated
    default: { get_attribute: [ SELF, host, ip_address ] }

requirements:
  - host:
      node: tosca.nodes.Compute
      relationship: tosca.relationships.HostedOn
```

C.7.3.4 Additional Requirements

- Nodes that can directly be managed and run by a TOSCA **Compute** Node Type SHOULD extend from this type.

C.7.4 tosca.nodes.WebServer

This TOSA **WebServer** Node Type represents an abstract software component or service that is capable of hosting and providing management operations for one or more **WebApplication** nodes.

Shorthand Name	WebServer
Type Qualified Name	tosca:WebServer
Type URI	tosca.nodes.WebServer

C.7.4.1 Properties

Name	Required	Type	Constraints	Description
None	N/A	N/A	N/A	N/A

C.7.4.2 Definition

```
tosca.nodes.WebServer:
  derived_from: tosca.nodes.SoftwareComponent
  capabilities:
    # Private, layer 4 endpoints
    app_endpoint: tosca.capabilites.Endpoint
    secure_endpoint: tosca.capabilities.Endpoint
    host:
      type: tosca.capabilities.Container
      properties:
        valid_node_types: [ tosca.nodes.WebApplication ]
```

C.7.4.3 Notes and Additional Requirements

- This node exports both a secure endpoint capability (i.e., **`secure_endpoint`**), typically for administration, as well as a regular endpoint (i.e., app**`_endpoint`**)

C.7.5 tosca.nodes.WebApplication

The TOSCA **`WebApplication`** node represents a software application that can be managed and run by a TOSCA **`WebServer`** node. Specific types of web applications such as Java, etc. could be derived from this type.

Shorthand Name	WebApplication
Type Qualified Name	tosca: WebApplication
Type URI	tosca.nodes.WebApplication

C.7.5.1 Properties

Name	Required	Type	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.

C.7.5.2 Definition

```
tosca.nodes.WebApplication:
  derived_from: tosca.nodes.Root
  properties:
    context_root:
      type: string
  capabilities:
    app_endpoint: tosca.capabilities.Endpoint
  requirements:
    - host:
        node: tosca.nodes.WebServer
        relationship: tosca.relationships.HostedOn
```

C.7.5.3 Additional Requirements

- None

C.7.6 tosca.nodes.DBMS

The TOSCA **DBMS** node represents a typical relational, SQL Database Management System software component or service.

C.7.6.1 Properties

Name	Required	Type	Constraints	Description
dbms_root_password	yes	string	None	The DBMS server’s root password.
dbms_port	no	integer	None	The DBMS server’s port.

C.7.6.2 Definition

```
tosca.nodes.DBMS:
  derived_from: tosca.nodes.SoftwareComponent
  properties:
    dbms_root_password:
      type: string
      description: the root password for the DBMS service
    dbms_port:
      type: integer
      description: the port the DBMS service will listen to for data and requests
  capabilities:
    host:
      type: tosca.capabilities.Container
      properties:
        valid_node_types: [ tosca.nodes.Database ]
```

C.7.6.3 Additional Requirements

- None

C.7.7 tosca.nodes.Database

Base type for the schema and content associated with a DBMS.

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

C.7.7.1 Properties

Name	Required	Type	Constraints	Description
db_user	yes	string	None	The special user account used for database administration.
db_password	yes	string	None	The password associated with the user account provided in the ‘db_user’ property.
db_port	yes	integer	None	The port the database service will use to listen for incoming data and requests.
db_name	yes	string	None	The logical database Name

C.7.7.2 Definition

```
tosca.nodes.Database:
  derived_from: tosca.nodes.Root
  properties:
    db_user:
      type: string
      description: user account name for DB administration
    db_password:
      type: string
      description: the password for the DB user account
    db_port:
      type: integer
      description: the port the underlying database service will listen to data
    db_name:
      type: string
      description: the logical name of the database
  requirements:
    - host:
        node: tosca.nodes.DBMS
        relationship: tosca.relationships.HostedOn
  capabilities:
    database_endpoint: tosca.capabilities.DatabaseEndpoint
```

C.7.7.3 Additional Requirements

- None

C.7.8 tosca.nodes.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.ObjectStorage

C.7.8.1 Properties

Name	Required	Type	Constraints	Description
store_name	yes	string	None	The logical name of the object store (or container).
store_size	no	scalar-unit	greater_or_equal : 0 GB	The requested initial storage size (default unit is in Gigabytes).
store_maxsize	no	scalar-unit	greater_or_equal : 0 GB	The requested maximum storage size (default unit is in Gigabytes).

C.7.8.2 Definition

```
tosca.nodes.ObjectStorage:
  derived_from: tosca.nodes.Root
  properties:
    store_name:
       type: string
    store_size:
       type: scalar-unit
       constraints:
         - greater_or_equal: 0 GB
    store_maxsize:
       type: scalar-unit
       constraints:
         - greater_or_equal: 0 GB
```

C.7.8.3 Additional Requirements

- None

C.7.8.4 Notes:

- Subclasses of the ObjectStorage node may impose further constraints on properties such as **store_name**, such as minimum and maximum lengths or include regular expressions to constrain allowed characters.

C.7.9 tosca.nodes.BlockStorage

The TOSCA **`BlockStorage`** node currently represents a server-local block storage device (i.e., not 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.BlockStorage

C.7.9.1 Properties

Name	Required	Type	Constraints	Description
size	yes *	scalar-unit	`greater_or_equal: 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).

C.7.9.2 Attributes

Name	Required	Type	Constraints	Description
volumeId	no	string	None	ID provided by the orchestrator for newly created volumes

C.7.9.3 Definition

```
tosca.nodes.BlockStorage:
  derived_from: tosca.nodes.Root
  properties:
    size:
      type: scalar-unit
      constraints:
        - greater_or_equal: 1 MB
    volume_id:
      type: string
      required: false
    snapshot_id:
      type: string
      required: false
  attributes:
    volumeId:
      type: string
  capabilities:
    attachment: tosca.capabilities.Attachment
```

C.7.9.4 Additional Requirements

- The **size** property is required when an existing volume (i.e., **volume_id**) is not available. However, if the property **volume_id** is provided, the **size** property is ignored.

C.7.9.5 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.

C.7.10 tosca.nodes.Container

The TOSCA **Container** node represents operating system-level virtualization technology used to run multiple application services on a single Compute host.

Shorthand Name	Container
Type Qualified Name	tosca:Container
Type URI	tosca.nodes.Container

C.7.10.1 Properties

Name	Required	Type	Constraints	Description
ContainerPort	yes	integer	TBD	The network port the container wishes to be addressed at.
hostPort	yes	integer	TBD	The network port of the host.

C.7.10.2 Attributes

Name	Required	Type	Constraints	Description
TBD				

C.7.10.3 Definition

```
tosca.nodes.Container:
  derived_from: tosca.nodes.Root
  properties:
    # TBD
  attributes:
    # TBD

  capabilities:
    host:
      type: tosca.capabilities.Container
      properties:
        valid_node_types: [tosca.nodes.ContainerApp]
    os:
      type: tosca.capabilities.OperatingSystem

    scalable:
      type: tosca.capabilities.Scalable

    endpoint:
      type: tosca.capabilities.Endpoint
```

C.7.10.4 Additional Requirements

- None

C.8 Artifact Types

TOSCA Artifacts represent the packages and imperative used by the orchestrator when invoking TOSCA Interfaces on Node or Relationship Types. Currently, artifacts are logically divided into three categories:

- **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 and 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.

Note: Normative TOSCA Artifact Types will be developed in future drafts of this specification.

C.8.1 tosca.artifacts.Root

This is the default (root) TOSCA Artifact Type definition that all other TOSCA base Artifact Types derive from.

C.8.1.1 Definition

```
tosca.artifacts.Root:
  description: The TOSCA Artifact Type all other TOSCA Artifact Types derive from
```

C.8.2 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.

C.8.2.1 Definition

```
tosca.artifacts.File:
  derived_from: tosca.artifacts.Root
```

C.8.3 Implementation Types

C.8.3.1 Script Types

C.8.3.1.1 *tosca.artifacts.impl.Bash*

This artifact type represents a Bash script type that contains Bash commands that can be executed on the Unix Bash shell.

C.8.3.2 Definition

```
tosca.artifacts.impl.Bash:
  derived_from: tosca.artifacts.Root
  description: Script artifact for the Unix Bash shell
  properties:
    mime_type: application/x-sh
    file_ext: [ sh ]
```

Appendix D. Non-normative type definitions

This section defines non-normative types used in examples or use cases within this specification.

D.1 Capability Types

D.1.1 tosca.capabilities.DatabaseEndpoint.MySQL

This type defines a custom MySQL database endpoint capability.

D.1.1.1 Properties

Name	Required	Type	Constraints	Description
None	N/A	N/A	N/A	N/A

D.1.1.2 Definition

```
tosca.capabilities.DatabaseEndpoint.MySQL:
  derived_from: tosca.capabilities.DatabaseEndpoint
```

D.2 Node Types

D.2.1 tosca.nodes.Database.MySQL

D.2.1.1 Properties

Name	Required	Type	Constraints	Description
None	N/A	N/A	N/A	N/A

D.2.1.2 Definition

```
tosca.nodes.Database.MySQL:
  derived_from: tosca.nodes.Database
  requirements:
    - host: tosca.nodes.DBMS.MySQL
  capabilities:
    database_endpoint: tosca.capabilities.DatabaseEndpoint.MySQL
```

D.2.2 tosca.nodes.DBMS.MySQL

D.2.2.1 Properties

Name	Required	Type	Constraints	Description
None	N/A	N/A	N/A	N/A

D.2.2.2 Definition

```
tosca.nodes.DBMS.MySQL:
  derived_from: tosca.nodes.DBMS
  properties:
    dbms_port:
      description: reflect the default MySQL server port
      default: 3306
  capabilities:
    host:
      type: Container
      properties:
        valid_node_types: [ tosca.nodes.Database.MySQL ]
```

D.2.3 tosca.nodes.WebServer.Apache

D.2.3.1 Properties

Name	Required	Type	Constraints	Description
None	N/A	N/A	N/A	N/A

D.2.3.2 Definition

```
tosca.nodes.WebServer.Apache:
  derived_from: tosca.nodes.WebServer
```

D.2.4 tosca.nodes.WebApplication.WordPress

D.2.4.1 Properties

Name	Required	Type	Constraints	Description
None	N/A	N/A	N/A	N/A

D.2.4.2 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: tosca.nodes.Database
  interfaces:
    Standard:
      inputs:
        db_host: string
        db_port: integer
        db_name: string
        db_user: string
        db_password: string
```

D.2.5 tosca.nodes.WebServer.Nodejs

D.2.5.1 Properties

Name	Required	Type	Constraints	Description
TBD	N/A	N/A	N/A	N/A

D.2.5.2 Definition

```
tosca.nodes.WebServer.Nodejs:
  derived_from: tosca.nodes.WebServer
  properties:
    github_url:
      required: no
      type: string
      description: location of the application on the github.
      default: https://github.com/mmm/testnode.git
  requirements:
    - database_endpoint:
        node: tosca.nodes.Database
        relationship:
          type: tosca.relationships.ConnectsTo
          interfaces:
            tosca.interfaces.relationship.Configure:
              pre_configure_source:
                implementation:
                  type: string
                input:
                  host:
                    type: string
                  port:
                    type: integer
  interfaces:
    tosca.interfaces.node.Lifecycle:
      input:
        github_url:
          type: string
```

Appendix E. Networking

This describes how to express and control the application centric network semantics available in TOSCA.

E.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.

TOSCA Networking takes the following approach.

1. 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.
2. The information provided in TOSCA is complete enough for a TOSCA implementation to fulfill the 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.
3. 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.
4. Service Template authors have the choice of specifying application component networking requirements in the Service Template or completely separating the application component to network mapping into a separate document. This allows application components with explicit network requirements to express them while allowing users to control the complete mapping for all software components which may not have specific requirements. Usage of these two approaches is possible simultaneously and required to avoid having to re-write components network semantics as arbitrary sets of components are assembled into Service Templates.
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.

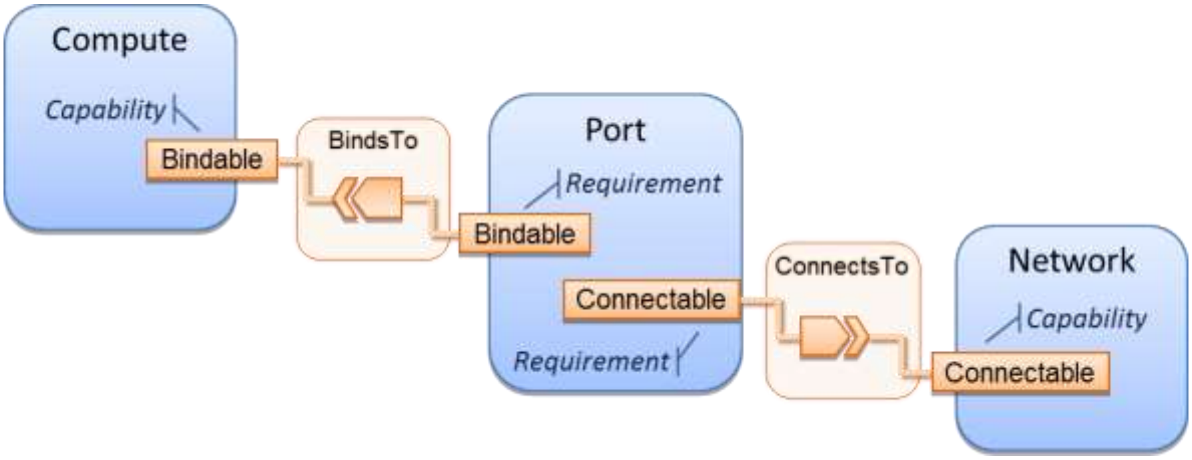
E.2 Connectivity Semantics

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.

Software components are expressed as Node Types in TOSCA which can express virtually any kind of concept in a TOSCA model. Node Types offering network based functions can model their connectivity using a special Endpoint Capability. tosca.capabilities.Endpoint, designed for this purpose. Node Types which require an Endpoint can specify this as a TOSCA requirement. A special Relationship Type, tosca.relationships.ConnectsTo, is used to implicitly or explicitly relate the source Node Type's endpoint to the required endpoint in the target node type. Since tosca.capabilities.Endpoint and tosca.relationships.ConnectsTo are TOSCA types, they can be used in templates and extended by subclassing in the usual ways, thus allowing the expression of additional semantics as needed.

The following diagram shows how the TOSCA node, capability and relationship types enable modeling the application layer decoupled from the network model intersecting at the Compute node using the Bindable capability type.

As you can see, the Port node type effectively acts a broker node between the Network node



description and a host Compute node of an application.

E.3 Expressing connectivity semantics

This section describes how TOSCA supports the typical client/server and group communication semantics found in application architectures.

E.3.1 Connection initiation semantics

The tosca.relationships.ConnectsTo expresses that requirement that a source application component needs to be able to communicate with a target software component to consume the services of the target. 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.

Application component intercommunication typically has conventions regarding which component(s) initiate the communication. Connection initiation semantics are specified in tosca.capabilities.Endpoint. Endpoints at each end of the tosca.relationships.ConnectsTo must indicate identical connection initiation semantics.

The following sections describe the normative connection initiation semantics for the tosca.relationships.ConnectsTo Relationship Type.

E.3.1.1 Source to Target

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 tosca.relationships.ConnectsTo relationship. The typical case is a “client” component connecting to a “server” component where the client initiates a stream oriented connection to a pre-defined transport specific port or set of ports.

It is the responsibility of the TOSCA implementation to ensure the source component has a suitable 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.

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.

E.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.

It is the responsibility of the TOSCA implementation to ensure the source component has a suitable 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.

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.

E.3.1.3 Peer-to-Peer

The Peer-to-Peer communication initiation semantic allows any member of a group to initiate communication with any other member of the same group at any time. This semantic typically appears in clustering and distributed services where there is redundancy of components or services.

It is the responsibility of the TOSCA implementation to ensure the source component has a suitable network path between all the member component instances and that the ports specified in the respective 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.

Endpoints specifying the Peer-to-Peer initiation semantic need not be related with a tosca.relationships.ConnectsTo relationship for the common case where the same set of component instances must communicate with each other.

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.

E.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 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.

The meaning of Source and Target port(s) corresponds to the direction of the respective tosca.relationships.ConnectsTo.

E.4 Network provisioning

E.4.1 Declarative network provisioning

TOSCA orchestrators are responsible for the provisioning of the network connectivity for declarative TOCSA Service Templates (Declarative TOCSA 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.

E.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.

E.4.3 Controlling network fulfillment

TOSCA provides mechanisms for providing control over network fulfillment.

This mechanism allows the application network designer to express in service template or network template how the networks should be provisioned.

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:

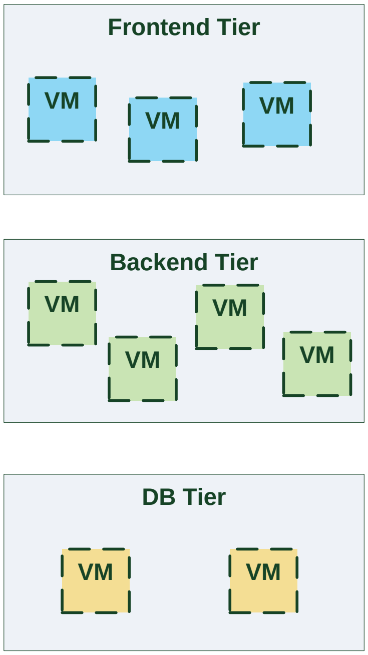


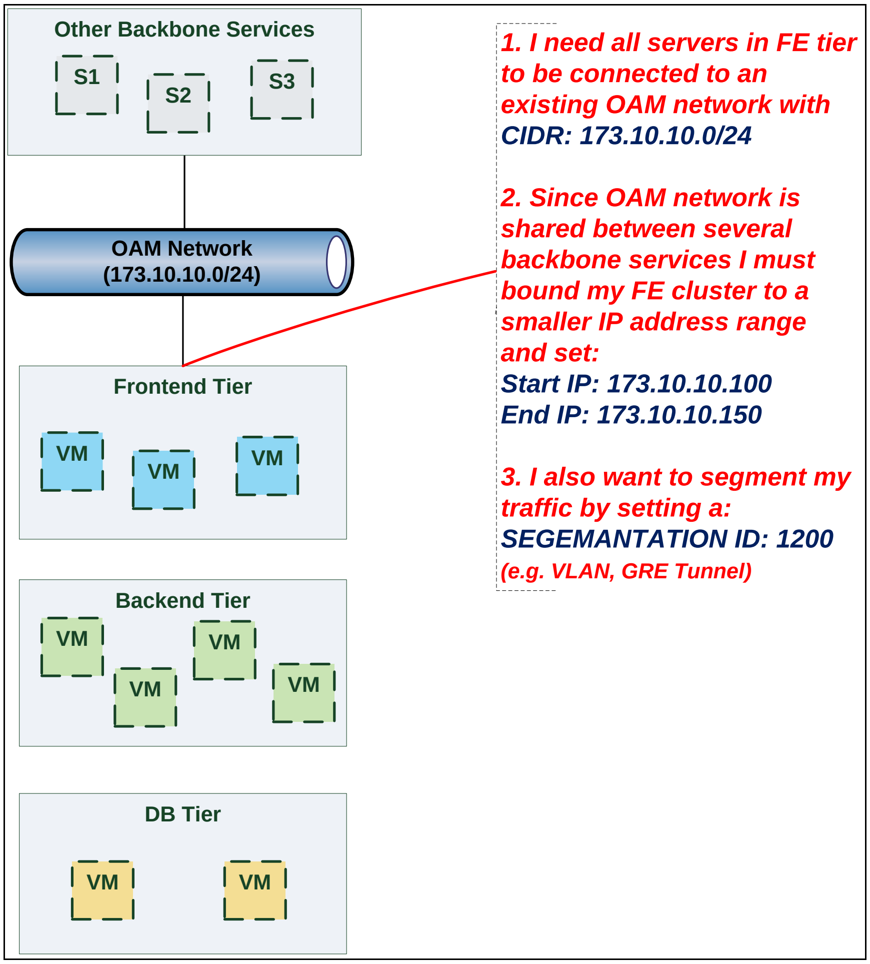
Figure-1: Typical 3-Tier Network

E.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.

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:



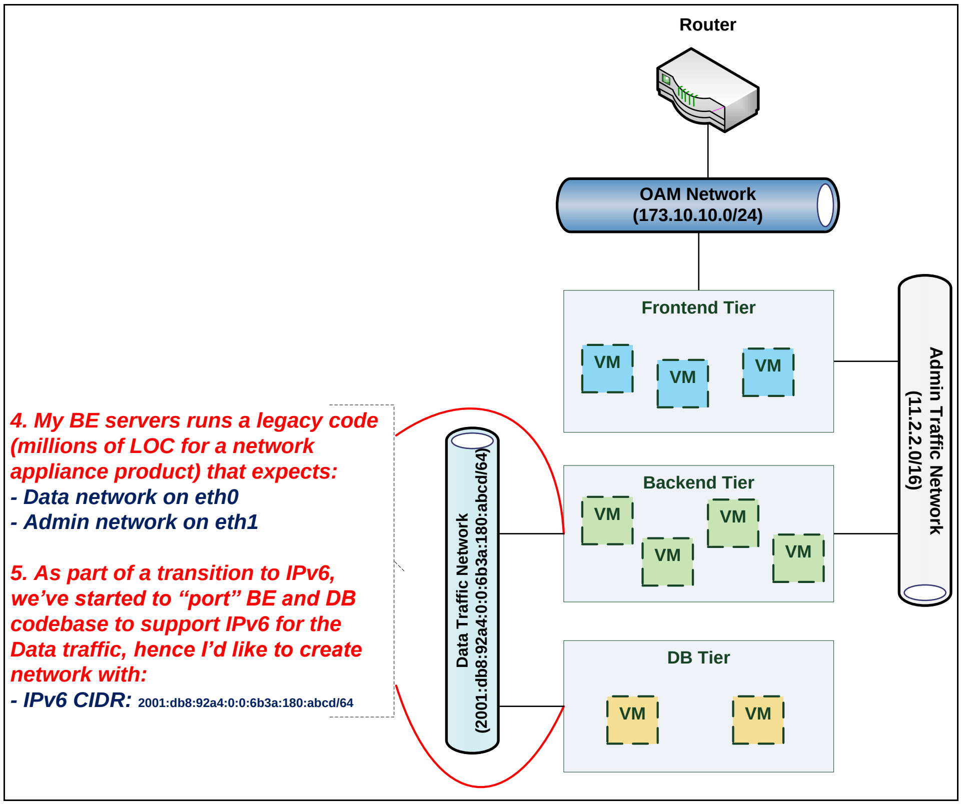
1. I need all servers in FE tier to be connected to an existing OAM network with CIDR: 173.10.10.0/24

2. Since OAM network is shared between several backbone services I must bound my FE cluster to a smaller IP address range and set:
Start IP: 173.10.10.100
End IP: 173.10.10.150

3. I also want to segment my traffic by setting a:
SEGEMANTATION ID: 1200
(e.g. VLAN, GRE Tunnel)

E.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.



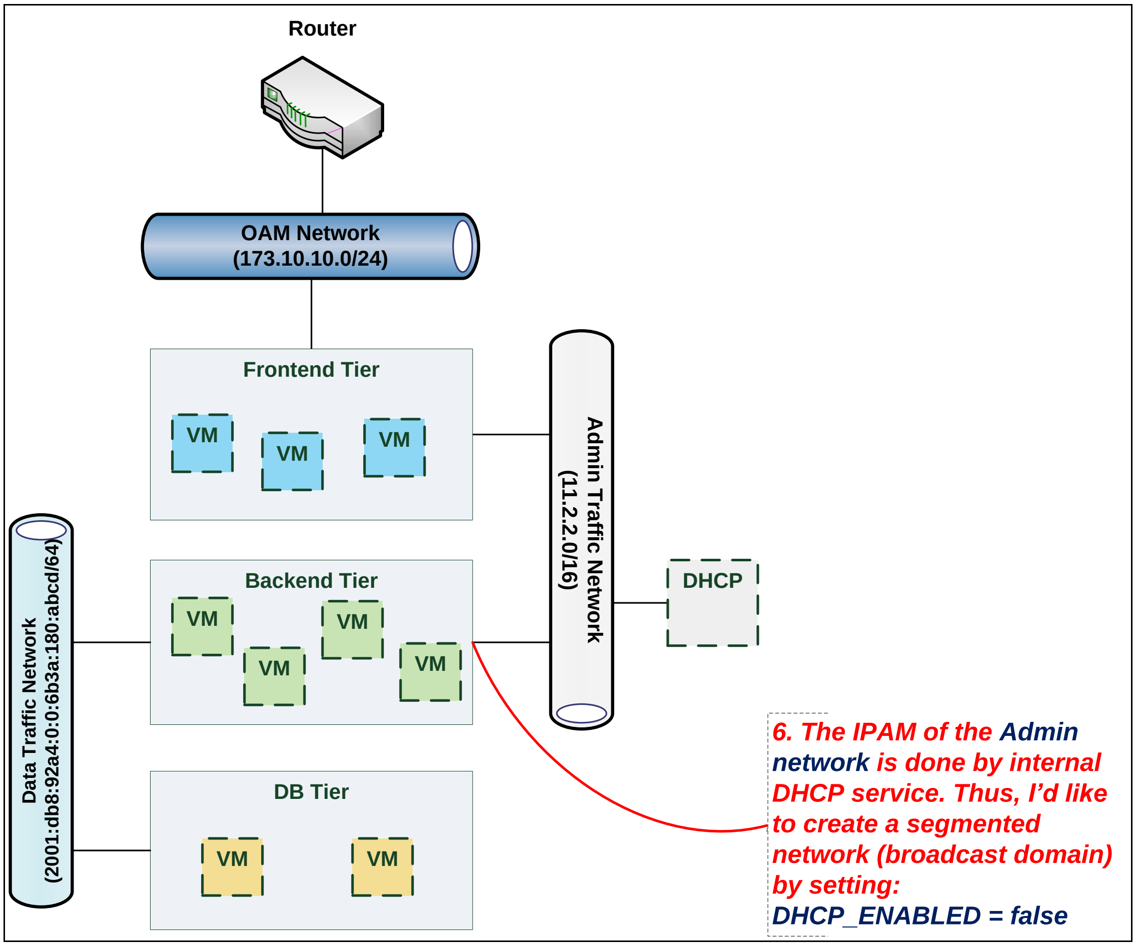
4. My BE servers runs a legacy code (millions of LOC for a network appliance product) that expects:
- Data network on eth0
- Admin network on eth1

5. As part of a transition to IPv6, we've started to "port" BE and DB codebase to support IPv6 for the Data traffic, hence I'd like to create network with:
- IPv6 CIDR: **2001:db8:92a4:0:0:6b3a:180:abcd/64**

E.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.

For this purpose, the app network designer would like to express in TOSCA that the underlying provisioned network will be set with DHCP_ENABLED=false. See this illustrated in the figure below:



6. The IPAM of the Admin network is done by internal DHCP service. Thus, I'd like to create a segmented network (broadcast domain) by setting: DHCP_ENABLED = false

E.5 Network Types

E.5.1 tosca.nodes.Network

The TOSCA **Network** node represents a simple, logical network service.

Shorthand Name	Network
Type Qualified Name	tosca:Network
Type URI	tosca.nodes.Network

E.5.1.1 Properties

Name	Required	Type	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 1st one in a pool of addresses derived from the cidr block full IP range
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

Name	Required	Type	Constraints	Description
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.
dhcp_enabled	no	boolean	default: true	Indicates the TOSCA container to create a virtual network instance with or without a DHCP service.

E.5.1.2 Attributes

Name	Required	Type	Constraints	Description
segmentation_id	no	string	None	The actual *segmentation_id* that is been assigned to the network by the underlying cloud infrastructure.

E.5.1.3 Definition

```
tosca.nodes.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
  capabilities:
    connection:
      type: tosca.capabilities.network.Connectable
```

E.5.1.4 Additional Requirements

- None

E.5.2 tosca.nodes.Port

The TOSCA **Port** node represents a logical entity that associates between Compute and Network normative types.

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.Port

E.5.2.1 Properties

Name	Required	Type	Constraints	Description
ip_address	no	string	None	Allow the user to set a static IP.

Name	Required	Type	Constraints	Description
order	no	integer	greater_or_equal: 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 *order* 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.

E.5.2.2 Attributes

Name	Required	Type	Constraints	Description
ip_address	no	string	None	The IP address which is being assigned to the associated compute instance.

E.5.2.3 Definition

```
tosca.nodes.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: string
      required: false
```

```
        default: false
      ip_range_start:
        type: string
        required: false
      ip_range_end:
        type: string
        required: false
    attributes:
      ip_address:
        type: string
    requirements:
      binding:
        type: tosca.capabilities.network.Bindable
      connection:
        type: tosca.capabilities.network.Connectable
```

E.5.2.4 Additional Requirements

- None

E.5.3 tosca.capabilities.network.Connectable

A node type that includes the Connectable capability indicates that it can be pointed by tosca.relationships.network.ConnectsTo relationship type.

Shorthand Name	network.Connectable
Type Qualified Name	tosca:network.Connectable
Type URI	tosca.capabilities.network.Connectable

E.5.3.1 Properties

Name	Required	Type	Constraints	Description
N/A	N/A	N/A	N/A	N/A

E.5.3.2 Definition

```
tosca.capabilities.network.Connectable:
  derived_from: tosca.capabilities.Root
```

E.5.4 tosca.relationships.network.ConnectsTo

This relationship type represents an association relationship between Port and Network node types.

Shorthand Name	network.ConnectsTo
Type Qualified Name	tosca:network.ConnectsTo
Type URI	tosca.relationships.network.ConnectsTo

E.5.4.1 Definition

```
tosca.relations.network.ConnectsTo:
  derived_from: tosca.relationships.DependsOn
  valid_targets: [ tosca.capabilities.network.Connectable ]
```

E.5.5 tosca.relationships.network.BindTo

This type represents a network association relationship between Port and Compute node types.

Shorthand Name	network.BindTo
Type Qualified Name	tosca:network.BindTo
Type URI	tosca.relationships.network.BindTo

E.5.5.1 Definition

```
tosca.relations.network.BindTo:
  derived_from: tosca.relationships.DependsOn
  valid_targets: [ tosca.capabilities.network.Bindable ]
```

E.6 Network modeling approaches

E.6.1 Option 1: Specifying a network outside the application's Service Template

This approach allows someone who understands the application's networking requirements, mapping the details of the underlying network to the appropriate node templates in the application.

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 be placing it in a separated self-contained network template file.

This "pluggable" network template approach introduces a new normative node type called Port, capability called *tosca.capabilities.network.Connectable* and relationship type called *tosca.relationships.network.ConnectsTo*.

The idea of the Port is to elegantly associate the desired compute nodes with the desired network nodes while not "touching" the compute itself.

The following diagram series demonstrate the plug-ability strength of this approach.

Let’s assume an application designer has modeled a service template as shown in Figure 1 that describes the application topology nodes (compute, storage, software components, etc.) with their relationships. The designer ideally wants to preserve this service template and use it in any cloud provider environment without any change.

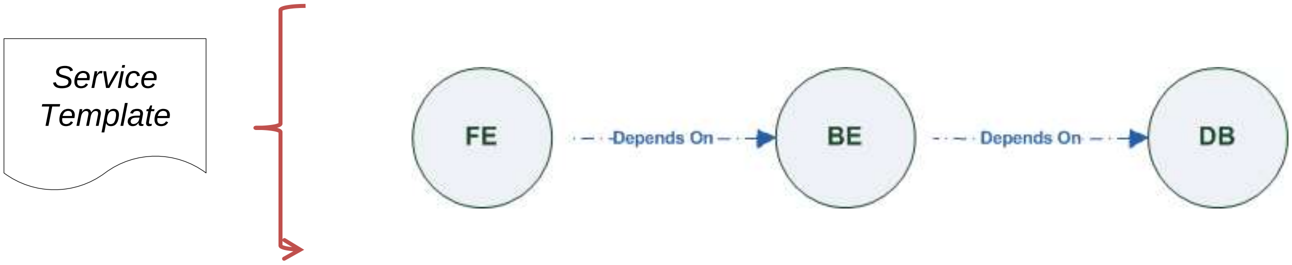


Figure-2: Generic Service Template

When the application designer comes to consider its application networking requirement they typically call the network architect/designer from their company (who has the correct expertise).

The network designer, after understanding the application connectivity requirements and optionally the target cloud provider environment, is able to model the network template and plug it to the service template as shown in Figure 2:

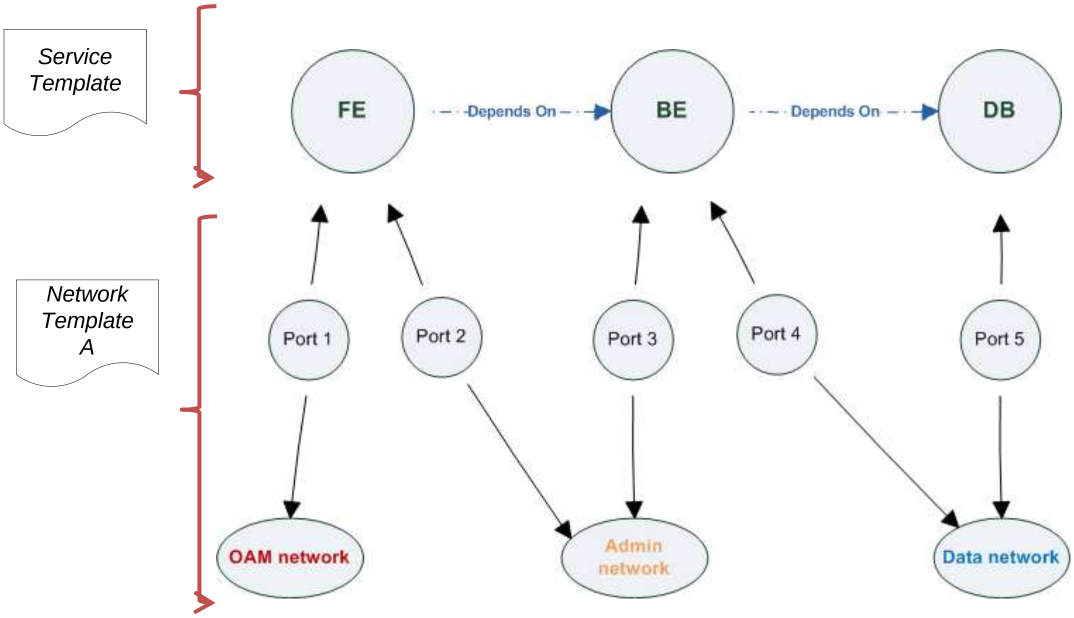
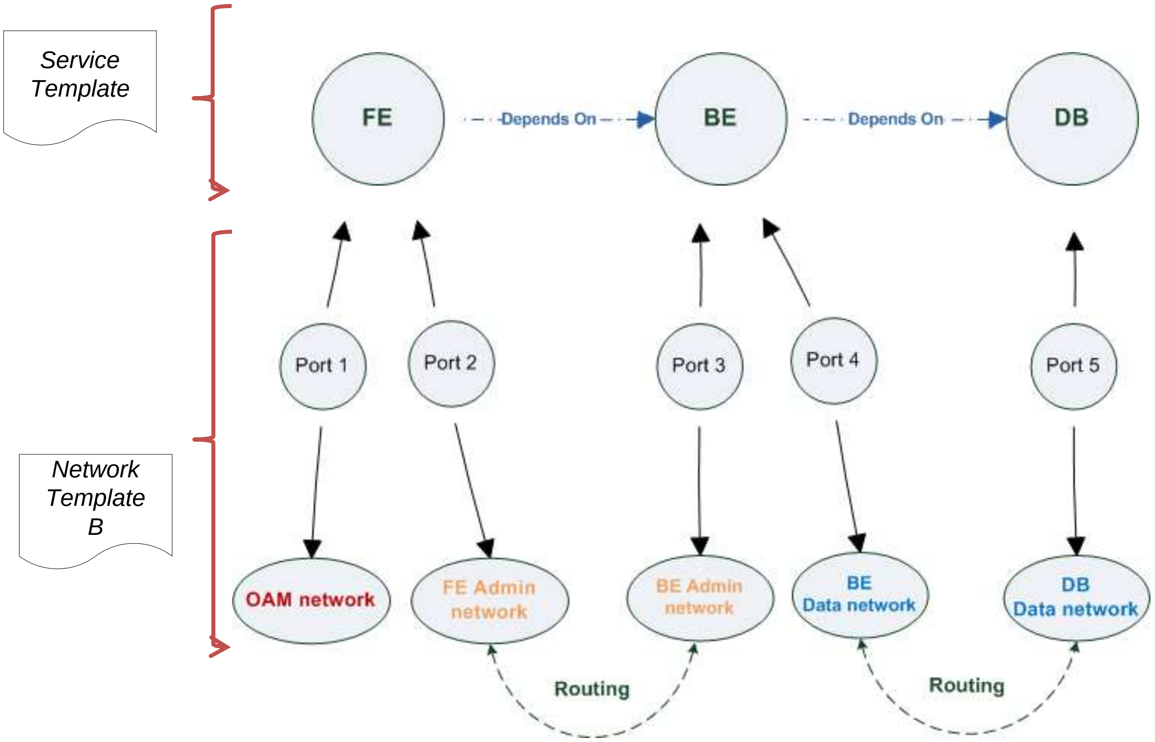


Figure-3: Service template with network template A

When there’s a new target cloud environment to run the application on, the network designer is simply creates a new network template B that corresponds to the new environmental conditions and provide it to the application designer which packs it into the application CSAR.



2195

2196 *Figure-4: Service template with network template B*

2197 The node templates for these three networks would be defined as follows:

```
node_templates:
  frontend:
    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
    properties: # omitted for brevity

  admin_network:
    type: tosca.nodes.Network
    properties: # omitted for brevity
```

```
data_network:
  type: tosca.nodes.Network
  properties: # omitted for brevity

# ports definition
fe_oam_net_port:
  type: tosca.nodes.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:
    - binding: frontend
    - connection: oam_network

fe_admin_net_port:
  type: tosca.nodes.Port
  requirements:
    - binding: frontend
    - connection: admin_network

be_admin_net_port:
  type: tosca.nodes.Port
  properties:
     order: 0
  requirements:
    - binding: backend
    - connection: admin_network

be_data_net_port:
  type: tosca.nodes.Port
  properties:
     order: 1
  requirements:
    - binding: backend
    - connection: data_network

db_data_net_port:
  type: tosca.nodes.Port
  requirements:
    - binding: database
```

```
      - connection: data_network
```

E.6.2 Option 2: Specifying network requirements within the application’s Service Template

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 3-tier application with their required networking modeling:

```
node_templates:
  frontend:
    type: tosca.nodes.Compute
    properties: # omitted for brevity
    requirements:
      -   network_oam: oam_network

      -   network_admin: admin_network

  backend:
    type: tosca.nodes.Compute
    properties: # omitted for brevity
    requirements:
      -   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
    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
    properties:
      ip_version:  { get_input: admin_network_ip_version }
      dhcp_enabled: { get_input: admin_network_dhcp_enabled }
```

```
data_network:
   type: tosca.nodes.Network
   properties:
     ip_version:  { get_input: data_network_ip_version }
      cidr: { get_input: data_network_cidr }
```

Appendix F. Component Modeling Use Cases

F.1.1 Use Case: Establishing a HostedOn relationship using WebApplication and WebServer

This use case examines the ways TOSCA YAML can be used to express a simple hosting relationship (i.e., HostedOn) using the normative WebServer and WebApplication node types defined in this specification.

For convenience, relevant parts of the normative Node Type for Web Server are shown below:

```
tosca.nodes.WebServer
  derived_from: SoftwareComponent
  capabilities:
    ...
    host:
      type: tosca.capabilities.Container
      properties:
        valid_node_types: [ tosca.nodes.WebApplication ]
```

As can be seen, the **WebServer** Node Type declares its capability to “contain” other nodes using the logical name “**host**” and providing the Capability Type tosca.capabilities.Container using its alias **Container**. It should be noted that the logical 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 includes a required list of valid Node Types that can be contained by this, the **WebServer**, Node Type. It is given the property name of **valid_node_types** and in this case it includes only the type **WebApplication**.

If we wish to establish a HostedOn relationship between a source WebApplication NodeType to a target WebServer Node Type we need to be able to declare a requirement from the source WebApplication that either explicitly declares the relationship or one that allows the relationship to be unambiguously inferred. We will examine three options for declaring this relationship below.

F.1.1.1 Option A: Inferred HostedOn relationship via logical name matching

In this option, the target **WebApplication** declares a requirement with the logical name “**host**” which matches the logical name for the declared capability in the **WebServer** Node Type, also named “**host**”. By virtue of the logical names matching (via the type designers), the HostedOn Relationship Type can be inferred by an orchestrator.

```
tosca.nodes.WebApplication:
  derived_from: tosca.nodes.Root
  requirements:
    - host: tosca.nodes.WebServer
```

F.1.1.1.1 *Notes*

- The logical name “host” is not a keyword and was selected for us in TOSCA normative types to give the reader an indication of the type of requirement being referenced.

F.1.1.2 Option B: Explicit HostedOn relationship via ‘relationship’ keyword

In this option, the target **`WebApplication`** declares a requirement with the logical name “**`host`**” (as in Option A), but also uses the **`relationship`** keyword to explicitly declare the Relationship Type HostedOn.

```
tosca.nodes.WebApplication:
  derived_from: tosca.nodes.Root
  requirements:
    - host:
        node: tosca.nodes.WebServer
        relationship: HostedOn
```

F.1.1.3 Option C: Explicit HostedOn relationship with capability keyword

In this option, let us instead declare a different Node Type called **`CustomWebApplication`** which declares a requirement with the logical name “**`bar`**” for a **`WebServer`** Node Type and also uses the **`type`** keyword to explicitly declare the Relationship Type HostedOn.

Since there is no implicit logical name match between “**`host`**” capability in the **`WebServer`** and “**`bar`**” requirement in **`CustomWebApplication`**, the type designer MUST use the capability keyword on the requirement to indicate to the orchestrator the exact name (i.e., “**`host`**”) of the capability in **`WebServer`** it should use to create the HostedOn relationship with.

```
tosca.nodes.WebApplication:
  derived_from: tosca.nodes.Root
  requirements:
    - bar:
        node: tosca.nodes.WebServer
        relationship: HostedOn
        capability: host
```

The service template that would reference the hosted on relationship would appear as follows:

```
TBD
```

F.1.2 Use Case: Establishing a ConnectsTo relationship to WebServer

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.

The service template that would establish a ConnectsTo relationship as follows:

```
node_types:
  MyServiceType:
    derived_from: SoftwareComponent
    requirements:
      - connection1:
          node: WebServer
          relationship: ConnectsTo
          capability: https_endpoint

node_templates:
   my_web_app:
      type: MyServiceType
      ...
      requirements:
        - connection1: my_web_server

  my_web_server:
    type: WebServer
```

F.1.2.1 Issues

4. How do we know that the requirement labeled “host” is a HostedOn relationship?
    1. versus a general “DependsOn” relationship? For example, what if the WebApplication had a different dependency on a WebServer node in addition to a hosting (i.e., HostedOn) dependency?
    2. Currently, our normative node uses the same named slot “host” on the requirement and capabilities side. If this were not the case, an ambiguity exists.
    3. Should Nodes be restricted to one Container requirement?
5. What capability does the “http_endpoint” export versus the “https_endpoint” from the WebServer?
    4. How does a WebApplication provide a Requirement to (one or the other of) them?
6. How do we list additional (sub) capabilities on the WebServer node that are NOT types?
    5. How do we reference them as additional requirements from the WebApplication?

F.1.3 Use Case: Attaching (local) BlockStorage to a Compute node

This use case examines the ways TOSCA YAML can be used to express a simple AttachTo relationship between a Compute node and a locally attached BlockStorage node.

The service template that would establish an AttachTo relationship follows:

```
node_templates:
   my_server:
    type: Compute
    ...
    requirements:
     # contextually this can only be a relationship type
     - persistant_storage:
         node: my_block_storage
         relationship: AttachTo
         # This maps the local requirement name ‘persistent_storage’ to the
         # target node’s capability name ‘attachment’
         capability: attachment
          properties:
           location: /path1/path2

  my_block_storage:
   type: BlockStorage
   properties:
    size: 10
```

F.1.3.1 Issues

- TBD

F.1.4 Use Case: Reusing a BlockStorage Relationship using Relationship Type or Relationship Template

This builds upon the previous use case (F.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 AttachTo relationship.

Specifically, several notation options are shown (in this use case) that achieve the same desired result.

F.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 occurrences in the same Service (Topology) Template; allowing the author to avoid configuring these same properties and operations in multiple Node Templates.

F.1.4.2 Notation Style #1: Augment AttachTo Relationship Type directly in each Node Template

This notation extends the methodology used for establishing a HostedOn relationship (see previous example, F.1.1.2), but allowing template author to supply (dynamic) configuration and/or override of properties and operations.

Note: This option will remain valid for Simple Profile regardless of other (following) notation (or aliasing) options being discussed or adopted.

```
node_templates:

  my_block_storage:
    type: BlockStorage
    properties:
      size: 10

  my_web_app_tier_1:
    type: Compute
    requirements:
      - attachment:
          node: my_block_storage
          relationship: MyAttachTo
            # use default property settings in the Relationship Type definition

  my_web_app_tier_2:
    type: Compute
    requirements:
      - attachment:
          node: my_block_storage
          relationship: MyAttachTo
            # Override default property setting for just the ‘location’ property
            properties:
              location: /some_other_data_location

relationship_types:

  MyAttachTo:
    derived_from: AttachTo
    properties: # follows the syntax of property definitions
      location:
        default: /default_location
    interfaces:
      some_interface_name:
        some_operation:
          implementation: default_script.sh
```

F.1.4.3 Notation Style #2: Use the ‘template’ keyword on the Node Templates to 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:
      - attachment:
          node: my_block_storage
          relationship: storage_attachto_1

  my_web_app_tier_2:
    derived_from: Compute
    requirements:
      - attachment:
          node: my_block_storage
          relationship: storage_attachto_2

relationship_templates:
  storage_attachto_1:
    type: MyAttachTo
    properties:
      location: /my_data_location



  storage_attachto_2:
    type: MyAttachTo
    properties:
      location: /some_other_data_location

relationship_types:

  MyAttachTo:
    derived_from: AttachTo
    properties: # follows the syntax of property definitions
      location:
        default: /default_location
    interfaces:
      some_interface_name:
        some_operation:
          implementation: default_script.sh
```

F.1.4.4 Notation Style #3: Using an alias which named Relationship Template to use

This option shows a way to alias an existing template from another template to further simplify the definition of named Relationship Templates using aliases to effectively “copy” an existing definition in to avoid repetition.

The example below shows that the Relationship Template named **storage_attachto_1** provides some overrides (conceptually a large set of overrides) on its Type which the Relationship Template named **storage_attachto_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_attachto_1

  my_web_app_tier_2:
    derived_from: Compute
    requirements:
      - attachment:
          node: my_block_storage
          relationship: storage_attachto_2
```

```
relationship_templates:
  storage_attachto_1:
    type: MyAttachTo
    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_attachto_2:
    alias: storage_attachto_1
    properties:
      location: /some_other_data_location

relationship_types:

  MyAttachTo:
    derived_from: AttachTo
    properties: # follows the syntax of property definitions
      location:
        default: /default_location
    interfaces:
      some_interface_name:
        some_operation:
          implementation: default_script.sh
```

For reference, here are is the BlockStorage, AttachTo and Attachment definitions:

```
tosca.capabilities.Attachment:
  derived_from: tosca.capabilities.Root

tosca.relationships.AttachTo:
  derived_from: tosca.relationships.Root
  valid_targets: [ tosca.capabilities.Attachment ]
  properties:
    location:
      type: string
      constraints:
        - min_length: 1
    device:
      type: string
      required: false

type: tosca.nodes.BlockStorage
  derived_from: tosca.nodes.Root
  properties:
    size:
      type: integer
      constraints:
        - greater_or_equal: 1
    volumeId:
      type: string
      required: false
  attributes:
    volumeId:
      type: string
  capabilities:
    - attachment: tosca.capabilities.Attachment

type: tosca.nodes.Compute
  derived_from: tosca.nodes.Root
  properties:
    ...
  capabilities:
    host:
      type: Container
        properties:
          valid_node_types: [tosca.nodes.SoftwareComponent]
```

F.1.5 Usage of add_target, target_changed, remove_target

TODO

Notes:

- These examples would apply the corresponding **`add_source`**, **`remove_source`** operations as well but in the reverse direction.

Appendix G. Application Modeling Use Cases

G.1 Application Modeling Use Cases:

Short description	Interesting Feature	Description
Virtual Machine (VM), single instance	• Introduces the TOSCA base Node Type for “Compute”.	TOSCA simple profile demonstrates how to stand up a single instance of a Virtual Machine (VM) image using a normative TOSCA Compute node.
WordPress + MySQL, single instance	• Introduces the TOSCA base Node Types of: “WebServer”, “WebApplication”, “DBMS” and “Database” along with their dependent hosting and connection relationships.	TOSCA simple profile service showing the WordPress web application with a MySQL database hosted on a single server (instance).
WordPress + MySQL + Object Storage, single instance	• Introduces the TOSCA base Node Type for “ObjectStorage”.	TOSCA simple profile service showing the WordPress web application hosted on a single server (instance) with attached (Object) storage.
WordPress + MySQL + Block Storage, single instance	• Introduces the TOSCA base Node Type for “BlockStorage” (i.e., for Volume-based storage).	TOSCA simple profile service showing the WordPress web application hosted on a single server (instance) with attached (Block) storage.
WordPress + MySQL, each on separate instances	• Instantiates 2 tiers, 1 for WordPress, 1 for DBMS and coordinates both.	Template installs two instances: one running a WordPress deployment and the other using a specific (local) MySQL database to store the data.
WordPress + MySQL + Network, single instance	• Introduces the TOSCA base Node Type for a simple “Network”.	TOSCA simple profile service showing the WordPress web application and MySQL database hosted on a single server (instance) along with demonstrating how to define associate the instance to a simple named network.
WordPress + MySQL + Floating IPs, single instance	• Connects to an external (relational) DBMS service	TOSCA simple profile service showing the WordPress web application and MySQL database hosted on a single server (instance) along with demonstrating how to create a network for the application with Floating IP addresses.

G.1.1 Virtual Machine (VM), single instance

G.1.1.1 Description

This use case demonstrates how the TOSCA Simple Profile specification can be used to stand up a single instance of a Virtual Machine (VM) image 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) that are desired by the template author. The cloud provider would attempt to fulfill these properties (to the best of its abilities) during orchestration.

G.1.1.2 Features

This use case introduces the following TOSCA Simple Profile features:

- A node template that uses the normative TOSCA **`Compute`** Node Type along with showing an exemplary set of its properties being configured.
- Use of the TOSCA Service Template **`inputs`** section to declare a configurable value the template user may supply at runtime. In this case, the property named “cpus” (of type integer) is declared.

    - Use of a property constraint to limit the allowed integer values for the “cpus” property to a specific list supplied in the property declaration.
- 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
    - The “instance_ip” output property is programmatically retrieved from the **Compute** node’s “ip_address” property using the TOSCA Service Template-level **get_property** function.

G.1.1.3 Logical Diagram

TBD

G.1.1.4 Sample YAML

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

description: >
  TOSCA simple profile that just defines a single compute instance. 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
      properties:
        # compute properties
        disk_size: 10 GB
        num_cpus: { get_input: cpus }
        mem_size: 4 MB
      capabilities:
        os:
          properties:
            architecture: x86_64
            type: Linux
            distribution: ubuntu
            version: 12.04
  outputs:
    instance_ip:
      description: The IP address of the deployed instance.
      value: { get_attribute: [my_server, ip_address] }
```

G.1.1.5 Notes

- This use case uses a versioned, Linux Ubuntu distribution on the Compute node.

G.1.2 WordPress + MySQL, single instance

G.1.2.1 Description

TOSCA simple profile service showing the WordPress web application with a MySQL database hosted on a single server (instance).

This use case is built upon the following templates from, OpenStack Heat’s Cloud Formation (CFN) template and from an OpenStack Heat-native template:

- https://github.com/openstack/heat-templates/blob/master/cfn/F17/WordPress_With_RDS.template
- https://github.com/openstack/heat-templates/blob/master/hot/F18/WordPress_Native.yaml

However, where the CFN template simply connects to an existing Relational Database Service (RDS) our template below will also install a MySQL database explicitly and connect to it.

G.1.2.2 Logical Diagram

TBD

G.1.2.3 Sample YAML

```
tosca_definitions_version: tosca_simple_yaml_1_0_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: integer
      description: Port for the MySQL database

  node_templates:
    wordpress:
      type: tosca.nodes.WebApplication.WordPress
      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, db_name ] }
              wp_db_user: { get_property: [ mysql_database, db_user ] }
              wp_db_password: { get_property: [ mysql_database, db_password ] }
              # goto requirement, goto capability, goto port property
              wp_db_port: { get_property: [ SELF, database_endpoint, port ] }

    mysql_database:
      type: Database
      properties:
        db_name: { get_input: db_name }
        db_user: { get_input: db_user }
        db_password: { get_input: db_pwd }
      capabilities:
        database_endpoint:
          properties:
            port: { get_input: db_port }
      requirements:
        - host: mysql_dbms
      interfaces:
        Standard:
          postconfigure: mysql_database_postconfigure.sh

    mysql_dbms:
      type: DBMS
      properties:
        dbms_root_password: { get_input: db_root_pwd }
        dbms_port: { get_input: db_port }
      requirements:
        - host: server
      interfaces:
        Standard:
          create: mysql_dbms_install.sh
          start: mysql_dbms_start.sh
          configure: mysql_dbms_configure.sh
            inputs:
              db_root_password: { get_property: [ mysql_dbms, dbms_root_password ] }

    webserver:
      type: WebServer
      requirements:
        - host: server
      interfaces:
        Standard:
          create: webserver_install.sh
          start: webserver_start.sh

    server:
      type: Compute
      properties:
        # compute properties (flavor)
        disk_size: 10
        num_cpus: { get_input: cpus }
        mem_size: 4096
      capabilities:
        os:
          properties:
            architecture: x86_64
            type: linux
            distribution: fedora
            version: 17

  outputs:
    website_url:
      description: URL for Wordpress wiki.
      value: { get_attribute: [server, ip_address] }
```

G.1.2.4 Sample scripts

Where the referenced implementation scripts in the example above would have the following contents

G.1.2.4.1 *wordpress_install.sh*

```
yum -y install wordpress
```

G.1.2.4.2 *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/db_name/ /etc/wordpress/wp-config.php
sed -i s/username_here/db_user/ /etc/wordpress/wp-config.php
sed -i s/password_here/db_password/ /etc/wordpress/wp-config.php
systemctl restart httpd.service
```

G.1.2.4.3 *mysql_database_postconfigure.sh*

```
# Setup MySQL root password and create user
cat << EOF | mysql -u root --password=db_rootpassword
CREATE DATABASE db_name;
GRANT ALL PRIVILEGES ON db_name.* TO "db_user"@"localhost"
IDENTIFIED BY "db_password";
FLUSH PRIVILEGES;
EXIT
EOF
```

G.1.2.4.4 *mysql_dbms_install.sh*

```
yum -y install mysql mysql-server
# Use systemd to start MySQL server at system boot time
systemctl enable mysqld.service
```

G.1.2.4.5 *mysql_dbms_start.sh*

```
# Start the MySQL service (NOTE: may already be started at image boot time)
```

```
systemctl start mysqld.service
```

G.1.2.4.6 *mysql_dbms_configure*

```
# Set the MySQL server root password
mysqladmin -u root password db_rootpassword
```

G.1.2.4.7 *webserver_install.sh*

```
yum -y install httpd
systemctl enable httpd.service
```

G.1.2.4.8 *webserver_start.sh*

```
# Start the httpd service (NOTE: may already be started at image boot time)
systemctl start httpd.service
```

G.1.3 WordPress + MySQL + Object Storage, single instance

G.1.3.1 Description

This use case shows a WordPress application that makes use of an Object Storage service to application artifacts.

Note: Future drafts of this specification will detail this use case

G.1.3.2 Logical Diagram

TBD

G.1.3.3 Sample YAML

```
TBD
```

G.1.4 WordPress + MySQL + Block Storage, single instance

G.1.4.1 Description

This use case is based upon OpenStack Heat's Cloud Formation (CFN) template:

- https://s3.amazonaws.com/cloudformation-templates-us-east-1/Gollum_Single_Instance_With_EBS_Volume.template
- https://github.com/openstack/heat-templates/blob/master/cfn/F17/WordPress_Single_Instance_With_EBS.template

Note: Future drafts of this specification will detail this use case.

G.1.4.2 Logical Diagram

TBD

G.1.4.3 Sample YAML: Variant 1: Using the normative AttachTo Relationship Type

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

description: >
  TOSCA simple profile with server and attached block storage using the normative
AttachTo 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: string
      description: Size of the storage to be created.
      default: 1 GB
    storage_location:
      type: string
      description: Storage mount path.

  node_templates:
    server:
      type: Compute
      properties:
        # compute properties (flavor)
        disk_size: 10
        num_cpus: { get_input: cpus }
        mem_size: 4096
        # host image properties
      capabilities:
        os:
          properties:
            architecture: x86_64
            type: linux
            distribution: fedora
            version: 18
      requirements:
       - persistant_storage:
           node: storage
           # Clarify the requirement as an ‘AttachTo’ Relationship Type
           relationship: AttachTo
             properties:
               location: { get_input: storage_location }

    storage:
      type: BlockStorage
      properties:
        size: { get_input: storage_size }

  outputs:
    public_ip:
      description: Public IP address of the newly created compute instance.
      value: { get_attribute: [server, ip_address] }
```

G.1.4.4 Sample YAML: Variant 2: Using a custom AttachTo Relationship Type

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

description: >
  TOSCA simple profile with server and attached block storage using a custom
AttachTo 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: string
      description: Size of the storage to be created.
      default: 1 GB
    storage_location:
      type: string
      description: Storage mount path.

  node_templates:
    server:
      type: Compute
      properties:
        # compute properties (flavor)
        disk_size: 10
        num_cpus: { get_input: cpus }
        mem_size: 4096
      capabilities:
        os:
          properties:
            architecture: x86_64
            type: Linux
            distribution: Fedora
            version: 18

      requirements:
        - persistant_storage:
            node: storage
            # Declare custom AttachTo type using the ‘type’ keyword
            relationship: MyCustomAttachToType

    storage:
      type: BlockStorage
      properties:
        size: { get_input: storage_size }

outputs:
  public_ip:
    description: Public IP address of the newly created compute instance.
    value: { get_attribute: [server, ip_address] }
```

G.1.4.5 Sample YAML: Variant 3: using a Relationship Template

```
tosca_definitions_version: tosca_simple_yaml_1_0_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: string
      description: Size of the storage to be created.
      default: 1 GB
    storage_location:
      type: string
      description: Storage mount path.

  node_templates:
    server:
      type: Compute
      properties:
        # compute properties (flavor)
        disk_size: 10
        num_cpus: { get_input: cpus }
        mem_size: 4096
      capabilities:
        os:
          properties:
            architecture: x86_64
            type: Linux
            distribution: Fedora
            version: 18
      requirements:
        - persistant_storage:
            node: storage
            # Declare template to use with ‘relationship’ keyword
            relationship: storage_attachment

    storage:
      type: BlockStorage
      properties:
        size: { get_input: storage_size }

    relationship_templates:
      storage_attachment:
        type: AttachTo
        properties:
          location: { get_input: storage_location }

  outputs:
    public_ip:
      description: Public IP address of the newly created compute instance.
      value: { get_attribute: [server, ip_address] }

relationship_types:
  MyCustomAttachToType:
     derived_from: AttachTo
     properties:
       location: { get_input: storage_location }
```

G.1.5 WordPress + MySQL, each on separate instances

G.1.5.1 Description

TOSCA simple profile service showing the WordPress web application hosted on one server (instance) and a MySQL database hosted on another server (instance).

This is based upon OpenStack Heat's Cloud Formation (CFN) template:

- https://github.com/openstack/heat-templates/blob/master/cfn/F17/WordPress_2_Instances.template

Note: Future drafts of this specification will detail this use case.

G.1.5.2 Logical Diagram

TBD

G.1.5.3 Sample YAML

```
TBD
```

G.1.6 WordPress + MySQL + Network, single instance

G.1.6.1 Description

This use case is based upon OpenStack Heat's Cloud Formation (CFN) template:

- https://github.com/openstack/heat-templates/blob/master/cfn/F17/WordPress_Single_Instance_With_Quantum.template

Note: Future drafts of this specification will detail this use case.

G.1.6.2 Logical Diagram

TBD

G.1.6.3 Sample YAML

```
TBD
```

G.1.7 WordPress + MySQL + Floating IPs, single instance

G.1.7.1 Description

This use case is based upon OpenStack Heat's Cloud Formation (CFN) template:

- https://github.com/openstack/heat-templates/blob/master/cfn/F17/WordPress_Single_Instance_With_EIP.template

Note: Future drafts of this specification will detail this use case.

G.1.7.2 Logical Diagram

TBD

G.1.7.3 Sample YAML

```
TBD
```

G.1.7.4 Notes

- The Heat/CFN use case also introduces the concept of "Elastic IP" (EIP) addresses which is the Amazon AWS term for floating IPs.
- The Heat/CFN use case provides a "key_name" as input which we will not attempt to show in this use case as this is a future security/credential topic.
- The Heat/CFN use case assumes that the "image" uses the "yum" installer to install Apache, MySQL and Wordpress and installs, starts and configures them all in one script (i.e., under Compute). In TOSCA we represent each of these software components as their own Nodes each with independent scripts.

G.1.8 BlockStorage + Compute

G.1.8.1.1 *Description*

TOSCA simple profile service showing single BlockStorage attached to Computer node as well as multiple BlockStorage attached to Compute node.

This use case is built upon the following templates from, OpenStack Heat's Cloud Formation (CFN) template and from an OpenStack Heat-native template:

- https://github.com/openstack/heat-templates/blob/master/hot/F18/NovaInstanceWithCinderVolume_Native.yaml

G.1.8.2 Logical Diagram

TBD

G.1.8.3 Sample YAML

G.1.8.3.1 *G1.8.3.1 Single BlockStorage*

```
tosca_definitions_version: tosca_simple_1.0

description: >
  TOSCA simple profile with server and 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: integer
      default: 1 GB
      description: Size of the storage to be created.
    storage_snapshot_id:
      type: string
      description: Some identifier that represents an existing snapshot that should
be used when creating the block storage.
    storage_location:
      type: string
      description: The relative location (e.g., path on the file system), which
provides the root location to address an attached node.

  node_templates:
    my_server:
      type: tosca.nodes.Compute
      properties:
        # compute properties (flavor)
        disk_size: 10
        num_cpus: { get_input: cpus }
        mem_size: 4096
        # host image properties
      capabilities:
        os:
          properties:
            architecture: x86_64
            type: linux
      distribution: fedora
      version: 18
   requirements:
    - attachment:
      node: my_storage
      relatonship: AttachTo
       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 }

 outputs:
  public_ip:
   description: Public IP address of the newly created compute instance.
   value: { get_attribute: [server, ip_address] }
```

G.1.8.3.2 *G1.8.3.2 Multiple BlockStorage*

```
tosca_definitions_version: tosca_simple_1.0

description: >
 TOSCA simple profile with server and 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: integer
   default: 1 GB
   description: Size of the storage to be created.
  storage_snapshot_id:
   type: string
   description: Some identifier that represents an existing snapshot that should
be used when creating the block storage.
  storage_location:
      type: string
      description: The relative location (e.g., path on the file system), which
provides the root location to address an attached node.

  node_templates:
    my_server:
      type: tosca.nodes.Compute
      properties:
        # compute properties (flavor)
        disk_size: 10
        num_cpus: { get_input: cpus }
        mem_size: 4096
      capabilities:
        os:
          properties:
            architecture: x86_64
            type: linux
            distribution: fedora
            version: 18
      requirements:
        - attachment:
            node: my_storage
            relationship: AttachTo
              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
      properties:
        # compute properties (flavor)
        disk_size: 10
        num_cpus: { get_input: cpus }
        mem_size: 4096
      capabilities:
        os:
          properties:
```

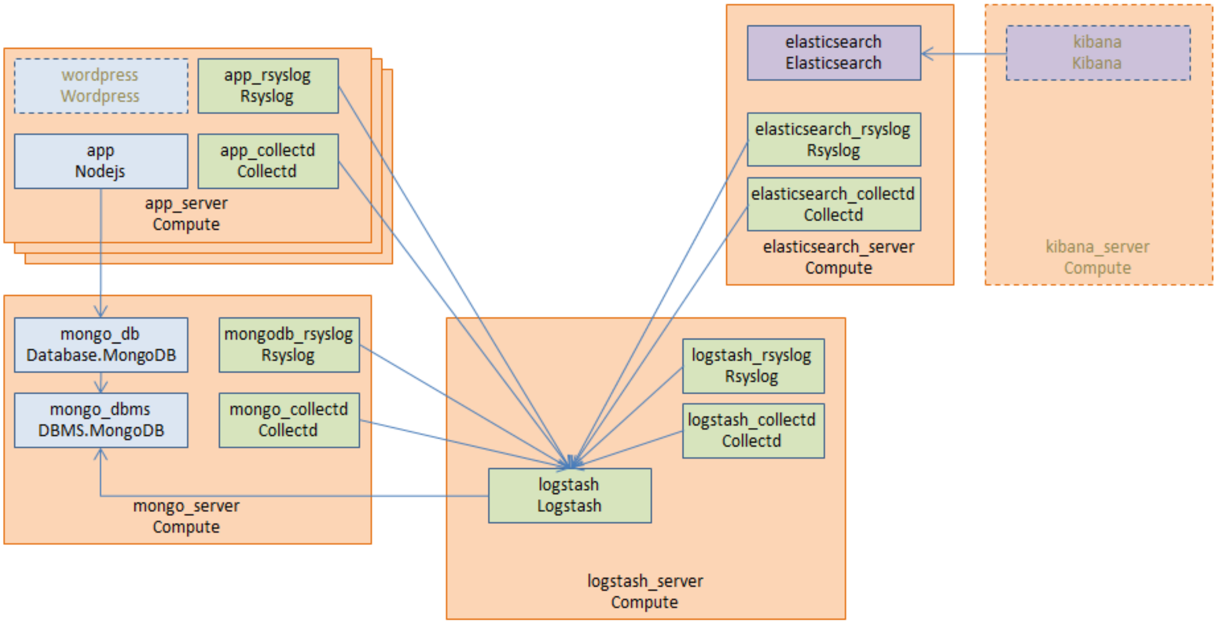
```
          architecture: x86_64
          type: Linux
          distribution: Fedora
          version: 18
    requirements:
      - attachment:
          node: my_storage2
          relationship: AttachTo
            properties:
              location: { get_input: storage_location }
  my_storage2:
    type: tosca.nodes.BlockStorage
    properties:
      size: { get_input: storage_size }
      snapshot_id: { get_input: storage_snapshot_id }

outputs:
  public_ip:
    description: Public IP address of the newly created compute instance.
    value: { get_attr: [server, ip_address] }
```

G.1.9 Monitoring use case with multiple instances

G.1.9.1 Description

TOSCA simple profile service showing the nodejs, mongodb, elaasticsearch, logstash, kibana, rsyslog and collectd installed on a different server (instance). This use case also demonstrates a use of TOSCA macros or dsl_definitions. It is a work in progress…

G.1.9.2 Logical Diagram



G.1.9.3 Sample YAML for application server

```
tosca_definitions_version: tosca_simple_1.0

description: >
  TOSCA simple profile with nodejs mongodb, elaasticsearch, logstash, kibana,
rsyslog and collectd.

imports:
  - custom_types/nodejs.yaml

dsl_definitions:
    ubuntu_node: &ubuntu_node
      # compute properties (flavor)
      disk_size: 10
      num_cpus: { get_input: my_cpus }
      mem_size: 4096
      # host image properties
    capabilities:
      os:
        properties:
          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/mmm/testnode.git

  node_templates:
    nodejs:
      type: tosca.nodes.Nodejs
      properties:
        github_url: { get_input: github_url }
      requirements:
        - host: app_server
        - database_endpoint:
            node: mongo_db
              interfaces:
                tosca.interfaces.relationship.Configure:
                  pre_configure_source:
                    implementation: nodejs/pre_configure_source.sh
                    input:
                      host: { get_attribute: [ TARGET, ip_address ] }
                      port: { get_property: [mongo_dbms, dbms_port] }
      interfaces:
        tosca.interfaces.node.Lifecycle:
          create: nodejs/create.sh
          configure:
            implementation: nodejs/config.sh
            input:
              github_url: { get_property: [ SELF, github_url ] }
          start: nodejs/start.sh

    mongo_db:
      type: tosca.nodes.Database
      requirements:
        - host: mongo_dbms

    mongo_dbms:
      type: tosca.nodes.DBMS
      requirements:
        - host: mongo_server
      properties:
        dbms_port: 27017
      interfaces:
        tosca.interfaces.node.Lifecycle:
          create: mongodb/create.sh
          configure: mongodb/config.sh
          start: mongodb/start.sh

    mongo_server:
      type: tosca.nodes.Compute
      properties: *ubuntu_node

    app_server:
      type: tosca.nodes.Compute
      properties: *ubuntu_node

  outputs:
    nodejs_url:
      description: URL for the nodejs server.
      value: { get_attribute: [app_server, ip_address] }
    mongodb_url:
      description: URL for the mongodb server.
      value: { get_attribute: [mongo_server, ip_address] }
    mongodb_port:
      description: Port for the mongodb server.
      value: { get_property: [mongo_dbms, dbms_port] }
```

G.1.9.4 Sample scripts

Where the referenced implementation scripts in the example above would have the following contents

G.1.9.4.1 *nodejs_install.sh*

```
#!/bin/bash
add-apt-repository ppa: chris-lea/node.js
```

```
apt-get update
apt-get install -y nodejs build-essential curl git npm
```

G.1.9.4.2 *nodejs_configure.sh*

```
#!/bin/bash
export app_dir=/opt/app
git clone $github_url /opt/app
if [ -f /opt/app/package.json ]
  cd  /opt/app/ && npm install
fi

cat > /etc/init/nodeapp.conf <<EOS
description "node.js app"

start on (net-device-up
and local-filesystems
and runlevel [2345])
stop on runlevel [!2345]

expect fork
respawn

script
export HOME=/
export NODE_PATH=/usr/lib/node
exec /usr/bin/node ${app_dir}/server.js >> /var/log/nodeapp.log 2>&1 &
end script
EOS
```

G.1.9.4.3 *nodejs_start.sh*

```
#!/bin/bash
start nodeapp
```

G.1.9.4.4 *nodejs_preconfigure.sh*

```
#!/bin/bash

cat > /opt/node/config.js<<EOF
{
  "host": "${host}"
  , "port": ${port}
}
EOF
```

G.1.9.4.5 ***mongodb_install.sh***

```
#!/bin/bash
apt-get install -y mongodb
```

G.1.9.4.6 ***mongodb_start.sh***

```
#!/bin/bash
start mongodb
```

Appendix H. References

H.1 Known Extensions to TOSCA v1.0

The following items will need to be reflected in the TOSCA (XML) specification to allow for isomorphic mapping between the XML and YAML service templates.

H.1.1 Model Changes

- The “TOSCA Simple ‘Hello World’” example introduces this concept in Section 3. 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 3.1.
  - “Inputs” could be mapped to BoundaryDefinitions in TOSCA v1.0. Maybe needs some usability enhancement and better description.
  - “outputs” are a new feature.
- 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.
- 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).

H.1.2 Normative Types

- Constraint (addresses TOSCA-117)
  - constraint clauses, regex
- Types / Property / Parameters
  - list, map, range, scalar-unit
  - anonymous types (entity_schema)
  - Includes YAML intrinsic types
  - NetworkInfo, PortInfo
- Node
  - Root, Compute, ObjectStorage, BlockStorage, Network, SoftwareComponent, WebServer, WebApp, DBMS, Database, …
- Relationship
  - Root, DependsOn, HostedOn, ConnectsTo, AttachTo, …
- Artifact
  - Deployment: Bash (for WD01)
- Requirements
  - None
- Capabilities
  - Container, Endpoint, Attachment, Scalable
- Lifecycle

  - (node) Standard, Simple
  - (Relationship) Configure
- Functions
  - get_input, get_attribute, get_property, get_nodes_of_type, get_operation_output
- Resource
  - In HEAT they have concept of key pairs (an additional resource type in the template).

H.2 Terminology

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

[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

.

H.3 Normative References

[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

H.4 Non-Normative References

[AWS-CFN]	Amazon Cloud Formation (CFN), http://aws.amazon.com/cloudformation/
[Chef]	Chef, https://wiki.opscode.com/display/chef/Home
[OS-Heat]	OpenStack Project Heat, https://wiki.openstack.org/wiki/Heat
[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

H.5 Glossary

The following terms are used throughout this specification and have the following definitions when used in 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 *Relationship Template* 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 ***reusable pattern*** and includes all details necessary to accomplish it.

Appendix I. Acknowledgments

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Contributors:

Avi Vachnis (avi.vachnis@alcatel-lucent.com), Alcatel-Lucent

Chris Lauwers (lauwers@ubicity.com)

Derek Palma (dpalma@vnomic.com), Vnomic

Frank Leymann (Frank.Leymann@informatik.uni-stuttgart.de), Univ. of Stuttgart

Gerd Breiter (gbreiter@de.ibm.com), IBM

Hemal Surti (hsurti@cisco.com), Cisco

Idan Moyal, (idan@gigaspaces.com), Gigaspaces

Jacques Durand (jdurand@us.fujitsu.com), Fujitsu

Juergen Meynert (juergen.meynert@ts.fujitsu.com), Fujitsu

Karsten Beins (karsten.beins@ts.fujitsu.com), Fujitsu

Kevin Wilson (kevin.l.wilson@hp.com), HP

Krishna Raman (kraman@redhat.com) , Red Hat

Luc Boutier (luc.boutier@fastconnect.fr), FastConnect

Matt Rutkowski (mrutkows@us.ibm.com), IBM

Moshe Elisha (moshe.elisha@alcatel-lucent.com), Alcatel-Lucent

Nikunj Nemani (nnemani@vmware.com), WMware

Richard Probst (richard.probst@sap.com), SAP AG

Sahdev Zala (spzala@us.ibm.com), IBM

Stephane Maes (stephane.maes@hp.com), HP

Thomas Spatzier (thomas.spatzier@de.ibm.com), IBM

Ton Ngo (ton@us.ibm.com), IBM

Travis Tripp (travis.tripp@hp.com), HP

Yaron Parasol (yaronpa@gigaspaces.com), Gigaspaces

2549

Appendix J. Revision History

Revision	Date	Editor	Changes Made
WD04, Rev. 01	2014-10-13	Matt Rutkowski, IBM	• Initial WD04, Revision 01 baseline. • Imported Ch. 13 “Nested templates” from Thomas Spatzier.
WD04, Rev 02	2014-10-20	Matt Rutkowski, IBM	• Merged updates to Ch 13 “nested templates” which update the examples/prose to wd03 normative type definitions. • Merged in Block Storage and Monitoring use cases from Sahdev Zala
WD04, Rev03	2014-10-21	Matt Rutkowski, IBM	• Broke out Requirement definition grammar from requirements element • Split up 4 different examples of requirement definition usage • Added target_filter keyword on requirement defn. and added an example for it.
WD04, Rev04	2014-10-22	Matt Rutkowski, IBM	• More work clarifying requirements and grammar for the Schema definition. • Re-authored requirement definition grammar and section (and examples). • Added datatype definitions • Added dsl_definitions, datatype_definitions and implements keywords for service template • Added placeholder for filter definition.
WD04, Rev05	2014-10-29	Matt Rutkowski, IBM	• Requirement definitions grammar chance and cascade change to ALL examples in document • Addition of property and node filter grammar, also cascaded to ALL examples in document
WD04, Rev06	2014-11-08	Matt Rutkowski, IBM	• Added the OperatingSystem Capability Type. changed Compute node to use it and fixed all examples in document. • Assured all references to target_filter on Requirement definitions where correct in all examples • Added text to property and attribute (definitions and elements) to better explain usage to retrieve desire or actual state of a Tosca entity’s transparent properties using get_xxx functions. • Better explained how scalar-unit type is used with constraints, updated properties in Compute, ObjectStorage, etc. to change some memory/storage size values to this type instead of integer and updated their definitions. • Added a status flag to Property definition with allowed values.
WD04, Rev07	2014-11-11	Matt Rutkowski, IBM	• Fixed one case where type “number” appeared • Added clarification to HOST function keyword to indicate entity hosted on chain is to be searched. Updated get_xxx grammars to include HOST. • Added Credential datatype • changed namespace for datatypes, to say “datatypes” instead of just “type”
WD04, Rev08	2014-11-17	Matt Rutkowski, IBM	• Updated issues lost based upon half-day work group review on 2014-11-13 • Fixed a few examples that used old namespaces. • Fixed prose of one example that used an old keyname “type” for a requirement definition. • Merged properties from MultiportEndpoint capability to Endpoint capability.
WD04, Rev09	2014-11-17	Thomas Spatzier, IBM	• Initial draft for nested template changes (TOSCA-186) in section 13; just edited sample snippets as base for discussion; overall text changes still to be done once discussions settle

WD04, Rev10	2014-12-03	Matt Rutkowski, IBM	• Fixed type defn. typos in some examples. • Cleaned up closed and deferred Mantis issue references • Fixed Port Node Type to have proper Requirements for “link” and “connection” otherwise they could not be wired together. • Showed simplified Req. Def. grammar versus full (i.e., no need to use “type” keyname in simple cases). ○ Opened issue TOSCA-217 for simplified grammar
WD04, Rev11	2014-12-05	Matt Rutkowski, IBM	• Fixed type defn. typos in some examples. • Added additional transitional states, removed unnecessary states (stopped, deleted). • Added diagrams to show normal startup/shutdown sequencing of Standard Lifecycle operations and resulting Node States. • Created TOSCA-218 to address agreements on how to handle valid_node_types as a part of Capability metamodel and tagged all relevant code/comments with this issue number.
WD04, Rev12	2014-12-08	Thomas Spatzier, IBM	• Introduction of topology_template throughout the document (except appendixes F and G). • Rewrite of “nested templates” use case in section 13. • Introduction of substitution_mapping section for topology_template.
WD04, Rev13	2014-12-10	Matt Rutkowski, IBM	• Added topology template key for all use cases in Appendix G. Fixed indentation on all use cases. • Fixed some errors in examples in section 13 as identified during YAML WG review. • Removed post_configure operation from Standard lifecycle and adjusted diagrams • Added diagrams showing typical startup and shutdown sequence for Standard lifecycle • Added illustration to show interleaving of relationship Configure lifecycle and Standard node lifecycle. • Added diagram to show how Compute-Port-Network relationship is established using new network caps., reqs. and relationship types. Added prose to explain diagram • After consulting Network WG co-chairs changed network node/Cap/Rel. type names as follows: ○ Capabilities.Linkable -> tosca.capabilities.network.Bindable ○ Relationships.LinkedTo -> tosca.relationships.network.BindsTo ○ Relationships.NetworkTo -> tosca.relationships.network.ConnectsTo ○ Capabilities.Connectivity -> tosca.capabilities.network.Connectable ○ This allows more app-centric naming for “network binding” and follows the naming paradigms established for Caps and Rels. ○ Updated all examples and grammar to reflect these name changes.

2550
Appendix K. Issues List

Issue #	Target	Status	Owner	Title	Notes
TOSCA-132	CSD03	Open	Palma	Use "set_property" methods to "push" values from template inputs to nodes	Feature. Needs new owner.
TOSCA-135	CSD02	Open	Rutkowski	Define/reference a Regex language (or subset) we wish to support for constraints	Feature, Reference a Perl subset.
TOSCA-136	CSD03	Open	Spatzier	Need rules to assure non-collision (uniqueness) of requirement or capability names	None
TOSCA-137	CSD03	Defer	Palma	Need to address "optional" and "best can" on node requirements (constraints) for matching/resolution	None
TOSCA-138	CSD02	Review/ Action	Palma	Define a Network topology for L2 Networks along with support for Gateways, Subnets, Floating IPs and Routers	Luc Boutier has rough proposal in MS Word format.
TOSCA-140	CSD03	Review	Palma	Constraining the capabilities of multiple node templates	
TOSCA-141	CSD03	Review	Palma	Specifying Environment Constraints for Node Templates (Policy related)	
TOSCA-142	CSD02	Review	Spatzier / Rutkowski	Define normative Artifact Types (including deployment/packages, impls., and runtime types)	
TOSCA-143	CSD02	Review	Rutkowski	Define normative tosca.nodes.Network Node Type (for simple networks)	Separate use case as what Luc proposes in TOSCA-138.
TOSCA-148	CSD03	Open	Palma	Need a means to express cardinality on relationships (e.g., number of connections allowed)	
TOSCA-151	CSD03	Defer	Rutkowski	Resolve spec. behavior if name collisions occur on named Requirements	subtask of TOSCA-148
TOSCA-152	CSD03	Open	Palma	Extend Requirement grammar to support "Optional/Best Can" Capability Type matching	subtask of TOSCA-137
TOSCA-153	CSD03	Open	Rutkowski	Define grammar and usage of Service Template keyname (schema namespace) "tosca_default_namespace"	
TOSCA-154	CSD03	Defer	Palma	Decide how security/access control work with Nodes, update grammar, author descriptive text/examples	
TOSCA-155	CSD03	Open	Rutkowski	How do we provide constraints on properties declared as simple YAML lists (sets)	Need to define constraints for “set” types
TOSCA-156	CSD03	Defer	Palma	Are there IPv6 considerations (e.g., new properties) for tosca.capabilities.Endpoint	
TOSCA-158	CSD03	Defer		Provide prose describing how Feature matching is done by orchestrators	Subtask of TOSCA-137
TOSCA-161	CSD03	Defer	Spatzier	Need examples of using the built-in feature (Capability) and dependency (Requirement) of tosca.nodes.Root	
TOSCA-162	CSD03	Defer	Rutkowski	Provide recognized values for tosca.nodes.compute properties: os_arch	
TOSCA-163	CSD03	Defer	Vachnis	Provide recognized values for tosca.nodes.BlockStorage: store_fs_type	
TOSCA-165	CSD03	Defer	Need new owner	New use case / example: Selection/Replacement of web server type (e.g. Apache, NGinx, Lighttpd, etc.)	
TOSCA-166	CSD03	Defer	Unassigned	New use case / example: Web Server with (one or more) runtimes environments (e.g., PHP, Java, etc.)	
TOSCA-167	CSD03	Defer	Unassigned	New use case / example: Show abstract substitution	

				of Compute node OS with different Node Type Impls.	
TOSCA-168	CSD03	Defer	Unassigned	New use case / example: Show how substitution of IaaS can be accomplished.	
TOSCA-170	CSD02	Open	Elisha	WD02 - Explicit textual mention, and grammar support, for adding (extending) node operations	
TOSCA-172	CSD02	Review	Lipton	2014 March - Public Comment Questions (Plans, Instance Counts, and linking SW Nodes)	
TOSCA-176	CSD03	Defer	Elisha	Add connectivity ability to Compute	
TOSCA-179	CSD03	Defer	Elisha	Add "timeout" and "retry" keynames to an operation	
TOSCA-180	CSD02	Open / In-progress	Elisha / Rutkowski	Support of secured repositories for artifacts	
TOSCA-181	CSD03	Open	Boutier	Dependency requirement type should match any target node. Dependency requirement type should match any target node.	Subtask of TOSCA-161
TOSCA-182	CSD02	Open	Palma	Document parsing conventions	
TOSCA-183	CSD02	Open	Palma	Composition across multiple yaml documents	
TOSCA-184	CSD02	Open	Palma	Pushing (vs pulling) inputs to templates	Subtask of TOSCA-132
TOSCA-185	CSD03	Review / Defer	Durand	Instance model	
TOSCA-186	CSD03	Defer	Spatzier	model composition	
TOSCA-189	CSD03	Defer	Shtilman	Application Monitoring - Proposal	Fixed
TOSCA-191	CSD02	Open In-Progress	Rutkowski	Document the “augmentation” behavior after relationship is selected in a requirement	
TOSCA-193	CSD02	Open	Spatzier	“implements” keyword needs its own section/grammar/example in A.5.2	Subtask of TOSCA-186
TOSCA-194	CSD02	Open	Lauwers	Nested Service Templates should be able to define additional operations	Subtask of TOSCA-186
TOSCA-196	CSD02	Open	Lauwers	Enhance "capabilities" section in nested templates	Subtask of TOSCA-186
TOSCA-197	CSD02	Open	Lauwers	Add "requirements" section in nested templates	Subtask of TOSCA-186
TOSCA-198	CSD02	Open	Lauwers / Rutkowski	Simplify “schema” specification	
TOSCA-200	CSD03	Open	Parasol	Query based upon capability	
TOSCA-201	CSD03	Open	Lauwers	Harmonize Properties and Capabilities in Node Types	
TOSCA-202	CSD03	Open	Boutier	Cardinalities for capabilities and requirements	Subtask of TOSCA-148
TOSCA-204	CSD03	Open	Boutier	Parameter definitions on operations should be closer to property definitions	
TOSCA-205	CSD03	Open	Boutier	Add interface type.	
TOSCA-206	CSD03	Open	Boutier	lifecycle.Simple interface and plan/workflow management...	
TOSCA-207	CSD02	Open	Boutier	postconfigure operation on Standard operation should be renamed in poststart	
TOSCA-208	CSD03	Open	Boutier	Add conditional capabilities (enable/disable capabilities on a node)	
TOSCA-209	CSD02	Open	Rutkowski	Fix Grouping example to use correct parameter for WebServer	
TOSCA-210	CSD02	Open	Rutkowski	Need example on get_xxx functions using HOST keyword	
TOSCA-211	CSD02	Open	Rutkowski	Need version on TOSCA Types (Node, Relationship, etc.)	

TOSCA-212	CSD03	Open	Boutier	Allow String concatenation for get_attributes/ properties to create aggregated props/outputs	
TOSCA-213	CSD03	Open	Lauwers	Clarify distinction between declaring properties and assigning property values	
TOSCA-214	CSD02	Open	Vachnis / Rutkowski	New functions for accessing the instance model	
TOSCA-217	CSD02	Open	Spatzier / Rutkowski	Add new simplified, single-line list notation / grammar for Requirement Def.	