Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0

Committee Note Draft 01

31 January 2013

Specification URIs

This version:

http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.doc (Authoritative)
http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html
http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.pdf

Previous version:

N/A

Latest version:

http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.doc (Authoritative)
http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.html
http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.pdf

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:

Frank Leymann (Frank.Leymann@informatik.uni-stuttgart.de), IBM
Matt Rutkowski (mrutkows@us.ibm.com), IBM
Adolf Hohl (Adolf.Hohl@netapp.com), NetApp
Marvin Waschke (marvin.waschke@ca.com), CA Technologies
Paul Zhang (paul.zhangyi@huawei.com), Huawei

Related work:

This document is related to:

- *Topology and Orchestration Specification for Cloud Applications Version 1.0*. Latest version. http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html.

Abstract:

This document provides an introduction to the Topology and Orchestration Specification for Cloud Applications (TOSCA).

Status:

This document was last revised or approved by the OASIS 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.

Technical Committee members should send comments on this document to the Technical Committee's email list. Others should send comments to the Technical Committee by using the "Send A Comment" button on the Technical Committee's web page at http://www.oasis-open.org/committees/tosca/.

Citation format:

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

[TOSCA-Primer]

Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0. 31 January 2013. OASIS Committee Note Draft 01. http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Statement of Purpose

Cloud computing offers a compelling cost-effective model for businesses that wish to host their applications and services in an environment where it can scale to meet their customer demands while reducing their need in maintaining the overhead of large datacenters and their operations.

However, these same customers, until TOSCA, lacked a standard means to describe the topology of their applications along with their dependent environments, services and artifacts inside a single service template which would enable them to deploy and manage them against the capabilities offered by any cloud provider, regardless of their infrastructure or service model.

This document seeks to provide a practical introduction to the TOSCA meta-model as defined within the TOSCA version 1.0 draft specification. It is intended to guide application architects and developers, as well as cloud providers and tool vendors, through the process of modeling and representing some basic applications as TOSCA service templates. Its purpose is to make you, regardless of your role, productive using TOSCA as soon as possible.

Each scenario is authored in a way to highlight the considerations and activities each role involved in the process of creating a cloud-based application would approach their task using different aspects and concepts from TOSCA.

The authors of this primer realize that many of the sample applications presented in this first version of the primer are quite simple compared to the possible “real world” applications many readers may be concerned with. It may seem even seem that using a modeling language like TOSCA might seem like “overkill” for such cases when compared to some domain-specific alternatives that are available. However, it is our hope, that through careful explanation of the thinking behind TOSCA modeling (even using the basic “hello world” examples included) the readers will come to appreciate the power, flexibility and benefits of TOSCA to handle more complex cases over other, more narrowly-scoped alternatives.

1.2 Scope of this Document

The aim of this document is to provide a quick start for cloud service developers to describe operational procedures using TOSCA. The following of this document is written primarily for cloud service developers and touches upon the view of other roles only. It is not meant to be a reference – it provides an answer for the most urgent questions to get familiar and leverage TOSCA from the chosen role.

1.3 References (non-normative)

1.3.1 Standards

[DMTF-CIMI]

Cloud Infrastructure Management Interface (CIMI) Model and REST Interface over HTTP Specification, Version 1.0, a Distributed Management Task Force (DMTF) Standard (DSP0263), 30 October 2012, http://dmtf.org/sites/default/files/standards/documents/DSP0263_1.0.1.pdf

[TOSCA-CSD-1.0]

OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA), Version 1.0, Committee Specification Draft (CSD) 06 / Public Review Draft 01, 29 November 2012, http://docs.oasis-open.org/tosca/TOSCA/v1.0/csprd01/TOSCA-v1.0-csprd01.pdf

1.3.2 Software and Services

[Amazon-EC2]

Amazon Elastic Compute Cloud (EC2), http://aws.amazon.com/ec2/.

[Apache-HTTP-Server]

Apache HTTP Server Project, http://httpd.apache.org/.

[MySQL-DB]

MySQL Community Server, http://dev.mysql.com/downloads/mysql/5.1.html.

[OpenStack]

OpenStack – Open Source Cloud Computing Software, http://www.openstack.org/.

[SugarCRM-CE]

SugarCRM Community Edition (CE), http://www.sugarforge.org/projects/sugarcrm/.

2 What Everybody Should Know About TOSCA

2.1 An overview of TOSCA

TOSCA formalizes a significant amount of interactions between somebody who develops IT delivered services and the one who actually operates them. In this chapter, we outline how TOSCA roles map to real-world IT business actors and what value it brings for each individual actor. However, it should be noted, that the majority of this document is targeted for people or organizations acting in a “Cloud Service Developer” role.

2.1.1 Bringing Cloud Services to Market – TOSCA Roles

TOSCA is a specification which adds value to the relationships between users, providers and developers of IT provided services. The roles are oriented on a model where a cloud service developer provides services which they distribute via further channels, primarily cloud service providers and which are eventually offered to service consumers.

The roles that we will reference in this document are briefly defined here:

- **Cloud Service Consumer**: A service consumer leverages IT provided services to run their business. These persons benefit from TOSCA, but not from direct contact with the specification.
- **Cloud Service Developer**: The main business of a cloud service developer is developing cloud services that will rely upon the operational and support services of from a Cloud Service Provider offers. The cloud service developer uses TOSCA to express how to instantiate and operate the services they developed.
- **Cloud Service Provider**: The main business of a cloud service provider is operating services developed by cloud service developers. Persons in this role use TOSCA to map request of a new service consumer to their infrastructure.

Of course, roles typically apply to separate market actors but one actor may also serve in multiple roles.

2.1.2 TOSCA Value Statement

TOSCA provides a compelling value statement for each role and its corresponding actor. In this section, we would like to highlight the reason why it makes sense to use the TOSCA specification for those who develop cloud services and those who deploy and operate them. Furthermore, there is an incentive for service consumers to choose services deployed and operated using TOSCA.

Although the majority of this document will be from the view of the TOSCA role of a Cloud Service Developer, the following table shows the benefits of TOSCA for each service role:

TABLE 1 - TOSCA BENEFITS BY SERVICE ROLE

TOSCA Service Roles		
Cloud Service Consumer	**Cloud Service Developer**	**Cloud Service Provider**
Cloud Service Consumers benefit indirectly from the standardization which TOSCA brings to the Cloud Service Developer and Cloud Service Provider. These benefits include: • More choices and flexibility in Cloud Provider. • Lower set-up and operational costs from TOSCA automation.	A Cloud Service Developer uses TOSCA as the standard to get their developed services at Cloud Service Providers in place. These persons: • Leverage the operational expertise of Cloud Service Providers • May further leverage the business/distribution network of Cloud Service Providers • Are able to choose from a wider range of cloud service providers. • Can work with multiple Cloud Service Providers in different legal environments with reasonable effort.	A Cloud Service Provider uses TOSCA to rapidly offer and deploy cloud services developed by Cloud Service Developers for Cloud Service Consumers. Provides: • Act as resellers for services developed by cloud service developer. • Can extend service offerings and revenue chances. • Optimize deployment and operational procedures and expenses. • Optimize the time to market for services

2.1.3 TOSCA Processing Environment

A TOSCA environment, operated by a Cloud Service Provider, might include various features that would be used to process TOSCA definitions according to the specification. These features may in turn be grouped into and provided as components that can be described as parts of cloud architectures. Many different ways of grouping these features into components and arranging these components into architecture exist. In addition, each vendor may decide which set of features to support and how they would be provided within their specific architecture. The figure below shows an example of a complete TOSCA environment in order to better help the reader comprehend the conceptual modeling and processing steps behind the TOSCA specification (see Figure 1).

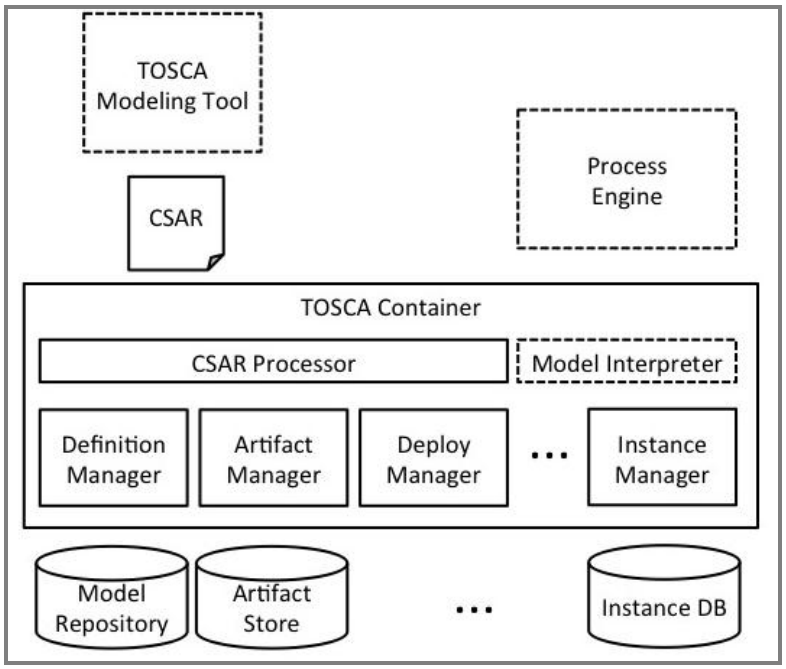

FIGURE 1 - SAMPLE ARCHITECTURE OF A TOSCA ENVIRONMENT

Cloud applications are typically packaged in TOSCA using *Cloud Service Archive* (or "CSAR") files. While the format of a CSAR file is defined by TOSCA, the way in which such files are created is out of scope of the specification. These archives could be created manually, but the TOSCA environment suggests that (graphical) TOSCA *Modeling Tools* would be offered to *Cloud Service Developers* to ease the creation of CSAR files. Nevertheless, the *Modeling Tool* is optional in a TOSCA environment, which is indicated by rendering it via a dashed box. Similarly, the *Process Engine* and the *Model Interpreter* are optional components as well, which is why they are also depicted by dashed boxes. However, when processing a CSAR in an imperative manner (see Section 2.2.3.1), a *Process Engine* is a mandatory component, but the *Model Interpreter* is still optional. Conversely, when processing a CSAR in a declarative manner (see Section 2.2.3.1) the *Model Interpreter* is mandatory and the *Process Engine* may be omitted.

During normal processing, CSAR files would be passed to the TOSCA *Container* within the environment: the TOSCA *Container* (or simply *container* for short) understands all the steps necessary to deploy, manage and decommission the cloud application over its lifecycle according to its definition.

As its first action, the container forwards a CSAR to the *CSAR Processor* which is the component in charge of processing the CSAR file in such a way that it can be initially deployed (step 1 in Figure 2). For this purpose, the *CSAR Processor* may interact with a *Model Interpreter* component (step 2 in Figure 2). Such an interaction may be necessary in case the cloud application packaged into the CSAR is processed declaratively.

The *CSAR Processor* will extract the definitions from the CSAR and pass them to the *Definition Manager* (step 3 in Figure 2*)*. The *Definitions Manager* is in charge of storing the definitions into the *Model Repository* such that all the definitions are available later on (step 4 in Figure 2).

Furthermore, the *CSAR Processor* would also extract all implementation artifacts and deployment artifacts from the CSAR and passes them to the *Artifact Manager* (step 5 in Figure 2*)*. The *Artifact Manager* component stores the artifacts in appropriate artifact stores (step 6 in Figure 2). This may include storing virtual images into image libraries available in the environment, storing scripts in subdirectories, etc. Next, the *Deploy Manager* is in charge to deploy all implementation artifacts into the environment (step 7 in Figure 2). Once this is done, the executables of all operations of node types and relationship types used in the topology template of the cloud application are available in the environment and ready for use.

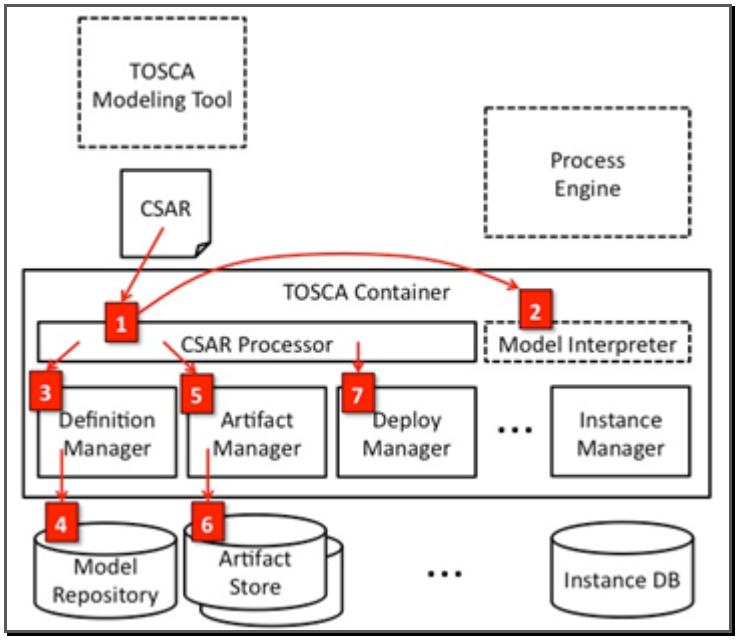

FIGURE 2 - SAMPLE "DECLARATIVE" PROCESSING SEQUENCE WHEN IMPORTING A CSAR

In an alternative processing flow (Figure 3), the *TOSCA Modeling Tool* (step 1 in Figure 3), instead of the *container*, may interact with a *Model Interpreter* component (step 2 in Figure 3) to transform a declarative model specified by an *application architect* into a completely imperatively processable model (step 3 in Figure 3) before it is packaged into a CSAR. In that case, the CSAR will always contain definitions that can be imperatively processed freeing the container from dealing with the declarative processing model at all.

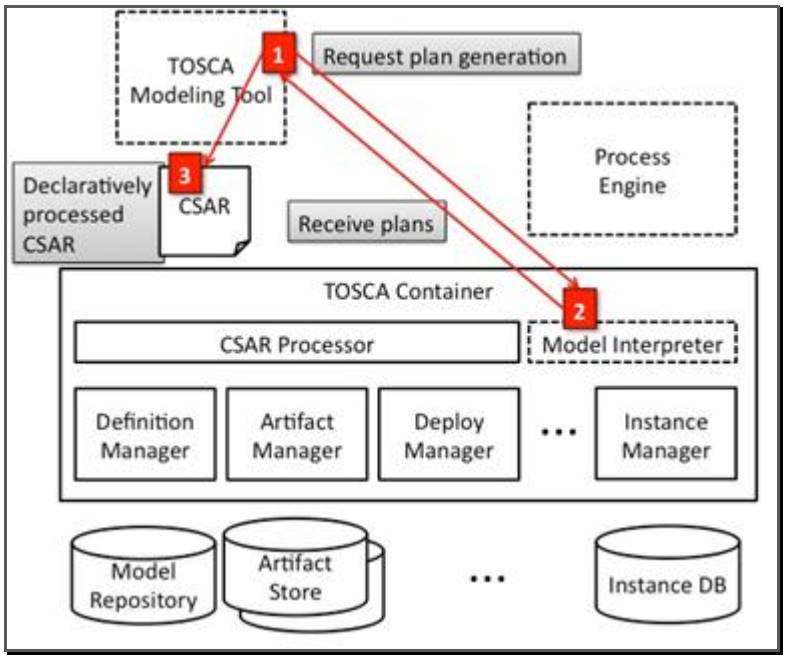

FIGURE 3 - SAMPLE EXTENSION OF A CSAR FOR "IMPERATIVE" PROCESSING

In the case where the TOSCA service template contains plans, the TOSCA container would perform additional imperative processing steps that continue from those shown in Figure 2 for the declarative case. The *Deploy Manager* of the container will deploy each plan into the *Process Engine* (step 7 in Figure 4). Deploying a plan into the *Process Engine* includes binding the tasks of the plans to the formerly deployed implementation artifacts: a task in a plan may refer to the operations of node types and relationship types, and the implementation artifacts of these operations are now known because they have been deployed at concrete endpoints in the environment before.

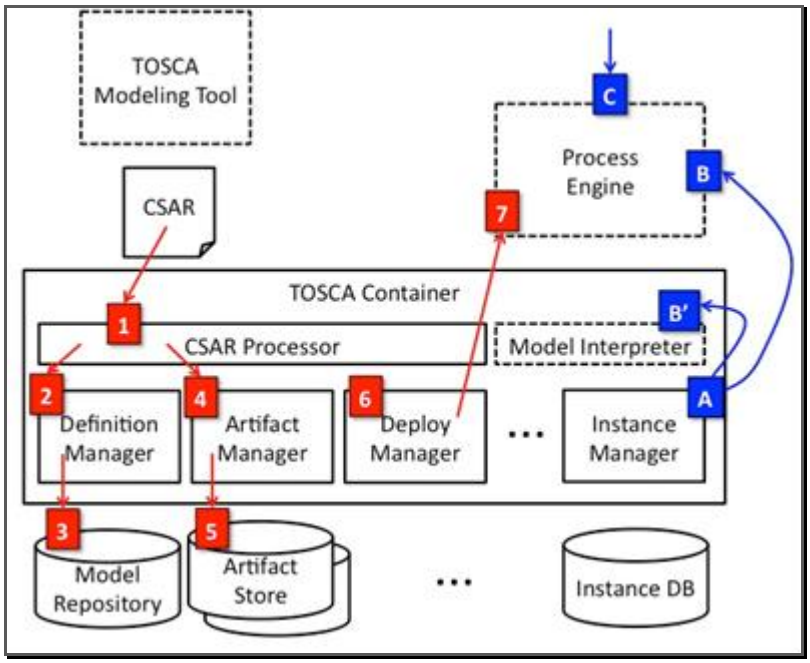

FIGURE 4 - SAMPLE "IMPERATIVE" PROCESSING SEQUENCE WHEN IMPORTING A CSAR

After these steps, the *environment* is set up for managing the cloud application represented by the CSAR and instances of the cloud application can be created. Creation of an instance of the cloud application is performed by the *Instance Manager* (step A in Figure 4). An instance of a cloud application is created by either invoking the build plan of the cloud application (i.e. imperative processing as shown in step B in Figure 4) that is executed by the *Process Engine*, or by interacting with the *Model Interpreter* (i.e. declarative processing as shown in step B' of Figure 4). Once a cloud application has been created it can be managed by invoking the other plans defined in the service template of the cloud application; these plans are executed by the process engine (step C in Figure 4). Finally, when the cloud application is no longer needed, it is decommissioned (by means of a plan performed by the *Process Engine*, or via the *Model Interpreter*).

> Note: The *Process Engine* is shown above as an optional component of the environment. In the case where a cloud application will be processed in a declarative way, i.e. without any plans, a process engine is not needed.

2.2 Roles Involved in Modeling a Cloud Application

In this document, we attempt to author sections that are targeted to the possible roles that may be involved with developing or interacting with cloud applications modeled using TOSCA:

The technical roles this document primarily addresses include the *type architect*, the *artifact developer* and the *application architect* and are profiled in this section. Each of these roles is a specialization of the generic role *cloud application service developer*. Depending on your company, the same person may fulfill more than one of these specialized roles. Concerted actions of these three roles are required to create a TOSCA service template and a corresponding TOSCA Cloud Service Archive (CSAR).

There are two other roles that may be concerned with TOSCA modeled cloud applications: the *cloud service consumer* and the *cloud service provider*. The cloud service consumer makes use of a modeled cloud application service, e.g. by following the self-service paradigm, browsing a cloud service catalogue and deciding to subscribe to a particular cloud service. The cloud service provider offers an environment (which will typically encompass a TOSCA container) in which cloud services can be run, especially provisioned, managed, and decommissioned.

Both these last two roles are typically not involved in the details of creating a service template, its types, artifacts, or the CSAR packaging of a cloud service. However, if you have a *cloud service provider* role and want to understand more about deploying TOSCA service templates, please read Section 5.

> Note: In the subsequent sections, we use the notion of cloud *services* and cloud *applications* synonymously.

2.2.1 Type Architect Role

The *type architect* is an expert for the types of components required by applications as well as the various types of connections between these components. This especially includes knowledge of the local management behavior of such components and connections, which is defined as the operations of these components and connections. Types of components are defined as TOSCA *Node Types*, and types of connections are defined as TOSCA *Relationship Types*.

For example, web applications based on the Java programming language may consist of "servlets" (or granular services) that run in a web server. These servlets may require a relational Database Management System (DBMS) for managing persistent data. A type architect will then define a "Servlet" node type, a "Web Server" node type, as well as a "DBMS" node type.

Since servlets are deployed on web servers, the type architect might define a relationship type "HostedOn" that can be used to express the corresponding relationship between a servlet and its web server. Similarly, a relationship "ConnectsToDBMS" will be defined to express the servlet's requirement for accessing a DBMS. The local management behavior of a specific DBMS includes operations for starting and stopping the database system, for backup and recovery of particular database content and other tasks.

Often, type architects are employees of vendors offering components or the ability to connect such components for use by application architects to define their cloud applications. For example, the vendor of a specific relational database system may define a node type that defines the *properties* of that relational database system as well as its local management behavior. For this purpose, the type architect may inherit from the "DBMS" node type mentioned above; this new, derived node type may define additional *properties* and include local management behavior (as additional *operations*) that is common for all relational database systems, independent of the specific product of a particular vendor. Such product-independent node types (as well as relationship types) may be defined by vendor consortia, for example, to ease the definition of product-specific types and to ensure within a cloud application the exploitation of different products in different environments.

> Note: The entire cloud application itself (i.e. its own service template) may be treated as a single set of services and defined as another TOSCA *Node Type*.
>
> For example, a cloud application representing a clustered application server (perhaps consisting of an HTTP server, a cluster manager, several cluster members, etc.) may be grouped together and defined as a new node type called "ScalableAppServer". This can be achieved by exporting relevant properties and operations of the enclosed node templates and relationship templates to "the boundary" of the service template (by means of the `BoundaryDefinition` element). In such a way, node types can be "implemented" by means of the language provided by TOSCA itself, supporting a recursive model of specifying increasingly complex cloud applications.

If you hold the role of a *type architect*, we recommend reading the contents of Section 3.

2.2.2 Artifact Developers Role

While the *type architect* is a specialist on the enablement of the management behavior of cloud applications as well as providing components of cloud applications and their connections, the *artifact developer* is an expert in code artifacts. They are in charge of providing and describing the installables and executables required to instantiate and manage a cloud application. For this purpose, the artifact developer defines the corresponding TOSCA *Node Type Implementations* and *Relationship Type Implementations*.

TOSCA supports two kinds of such code artifacts, namely *Implementation Artifacts* and *Deployment Artifacts*. *Implementation artifacts* are the executables implementing the operations of the interfaces of node types and relationship types. *Deployment artifacts* are the installables of the components that make up a cloud application; these components are defined by means of node types.

Each artifact has a TOSCA *Artifact Type*. An artifact type defines the kind of an artifact (e.g. a Java Archive or "JAR" file or an RPM package), as well as the kind of information required to correctly process the corresponding artifact. For example, an artifact type named "JARfile" may specify that a JAR file has a file name and a version number.

A TOSCA *Artifact Template* represents a reusable artifact (e.g. a particular RPM package, or a particular JAR file) and provides the actual information required to cope with the artifact. An example would be a JAR file, which provides a service for registering customers, and has the file name "RegisterCustomer" and has the version "1.0". An artifact template points to the actual code it represents, for example by pointing to a CSAR or to an FTP address.

All implementation artifacts and deployment artifacts required to install, run, and manage an instance of a node type or a relationship type is defined as a TOSCA *Node Type Implementation* or TOSCA *Relationship Type Implementation*, respectively. A node type implementation or a relationship type implementation refers to the artifact templates needed in order to bundle appropriate implementation artifacts and deployment artifacts. An implementation artifact or a deployment artifact may add further information about the artifact used, which is dependent on its usage context; for example, authentication information might be required to process an RPM package.

In order to provide components that are useful in different kinds of cloud applications, the corresponding node types and associated node type implementations must be defined. The same is true for connections between components of cloud applications: they are defined by *relationship types* and *relationship type implementations*. Thus*, type architects* provide the type definitions that *artifact developers* take as basis for creating their implementations.

Related type definitions and implementations might then be packaged by vendors into CSARs. These packages may then be used by *application architects* to import the corresponding types and implementations allowing them to reuse the types and implementation in building their own cloud applications.

If you hold the role of *artifact developer*, we recommend reading the contents of Section 4.

2.2.3 Application Architect Role

The *application architect* is an expert in both, the overall structure of a cloud application, its composite types and artifacts, as well as its global management behavior covering its complete lifecycle. The structure of a cloud application is specified by means of a *topology template*. Thus, the application architect identifies and defines the node templates making up a cloud application as well as the relationship templates wiring the collection of node templates into the topology of the cloud application. For this purpose, the application architect relies on node types and relationship types available at their disposal, and that have already been defined by a *type architect* previously. However, some application architects may also need to define their own node and artifact types when ready-made types have not been made available to them by some type architect. In these cases, the *application architect* would need to understand the functions described for the *type architect* and a*rtifact developer* roles already discussed in this chapter.

The global management behavior covering the complete lifecycle of a cloud application is defined by means of plans. A *plan* is a workflow that specifies the sequencing in which individual management operations offered by the node templates and relationship templates making up the cloud application have to be executed. The management operations available for the node templates and relationship templates are exactly the operations defined by the type architect when specifying the node types and relationship types of the corresponding templates. Furthermore, a plan may include any other task required to define overall management behavior. For example, acquiring a license for a cloud application might be realized by a plan. This plan may use operations for acquiring licenses of each individual component of the cloud application. Corresponding operations may have been defined by the type architect of the node types and relationship types of the affected templates.

A topology template and its corresponding plans are referred to collectively as a *service template*. A service template is the definition of the types of components that make up a cloud application. The executables required to actually instantiate, run and manage the cloud application are packaged with the service template into a *CSAR* (i.e. a *Cloud Service ARchive*). Typically, a CSAR is a self-contained and portable representation of a cloud application that can be deployed and managed in an environment that supports TOSCA.

If you hold the role of an *application architect*, we recommend reading the contents of Section 6.

2.2.3.1 A Note on Imperative Processing and Declarative Processing of Service Templates

TOSCA supports processing of service templates in two different flavors. The first, referred to as *imperative processing* of a service template, requires the complete topology of a cloud application to be explicitly defined, as well as all of its management behavior by means of plans. The second flavor, referred to as *declarative processing* of a service template, is based on the assumption that it is possible to infer the management behavior of the corresponding cloud application (e.g. how to deploy the application); this typically requires the precise definition of the semantics of node types and relationship types and their correct interpretation within a TOSCA *environment*.

In other words: the *imperative processing* flavor specifies precisely *how* a cloud application is structured and managed, while the *declarative processing* flavor specifies *what* structural elements of a cloud application are needed and what management behavior is to be realized. While the TOSCA specification already provides hooks (such as TOSCA *plans*) for defining models that encompass imperative processing of service templates, its current focus in the first version of the specification is on declarative processing.

As an example, you want to model a cloud application that consists of a customer database that is managed by a relational database system that makes use of block storage. The topology of the corresponding cloud application consists of three nodes (see Figure 5 below): the “CustomerDB” node, the “MyDBMS” node, and the “MyBlockStore” node.

These three nodes are connected by two relationships: the “CustomerDB_on_MyDBMS” relationship connects the “CustomerDB” node and the “MyDBMS” node; the “MyBlockStore_for_MyDBMS” relationship connects the “MyDBMS” node and the “MyBlockStore” node. Furthermore, the figure also depicts the operations of the nodes and the relationships that are offered to manage the overall cloud application. In this example, the “MyDBMS” node offers the “Install()” operation used to install in instance of the database system in the environment. This explicitly defines the topology of your cloud application.

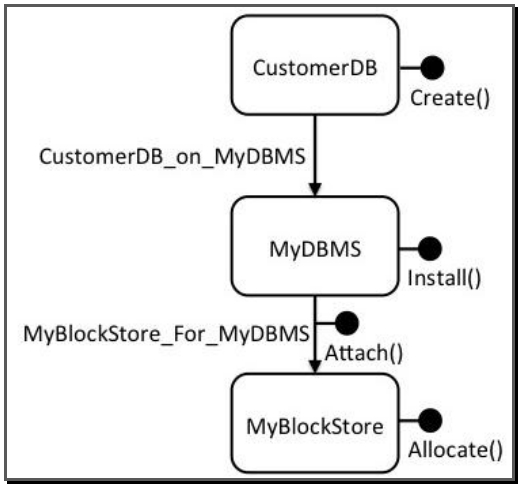

FIGURE 5 - TOPOLOGY OF A SIMPLE CLOUD APPLICATION

> Note: In order to ease comprehension of the basic mechanisms behind TOSCA we are bit lax in this section when using TOSCA terminology. For example, we do not distinguish between node types and node templates, or relationship types and relationship templates, respectively, but we just speak about nodes and relationships. In the other sections of this primer, we use the precise language established by TOSCA.
>
> For example, the precise terminology would be "*the CustomerDB node template refers to the DBMS node type*" etc. We will explain the differences between types and templates in the following sections.

In case you focus on deployment and decommissioning of your cloud application, you don't have to provide additional plans to the topology. This is because the *Model Interpreter* (see Section 2.1.3) of your TOSCA container can infer how to initially deploy and finally decommission the cloud application. The model as depicted in Figure 2 is sufficient for declarative processing of the sample cloud application. For example, the *Model Interpreter* can (in a nutshell) simply navigate the topology "from the leaves to the root". For each node reached it will invoke the operation distinguished as the one to use for creating an instance of the node, and similar for the relationships.

For this purpose, each node and relationship must support basic lifecycle operations such as an operation to create and delete instances, starting and stopping instances and so on. When initially provisioning a new instance of a cloud application, the *Model Interpreter* will determine the corresponding operation of each node (or relationship, respectively, if necessary). Also, the order in which relations have to be considered must be clear. For example, when provisioning an instance of the sample cloud application, the "Attach()" operation of the "MyBlockStorage_For_MyDBMS" can only meaningfully be performed once the "MyBlockStore" is allocated and the "MyDBMS" is installed.

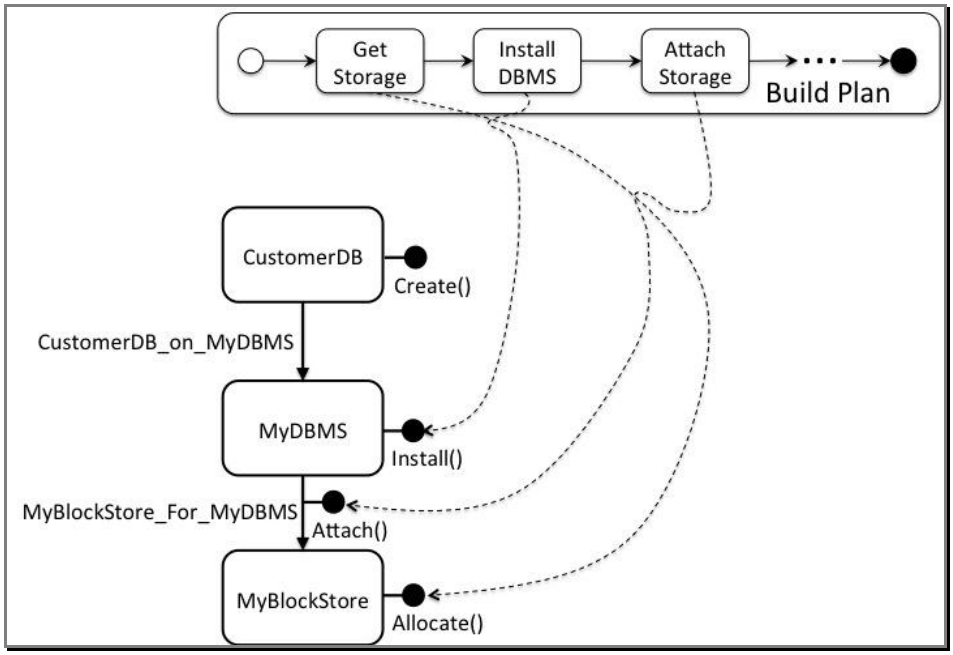

FIGURE 6 - SERVICE TEMPLATE OF A SAMPLE APPLICATION INCLUDING A BUILD PLAN

In contrast to declarative processing that relies on a *Model Interpreter* available in the environment, imperative processing of the cloud application extends the topology by explicitly modeled plans that specify how to manage the cloud application. For example, Figure 6 shows the plan that ensures the initial deployment of the cloud application – such kind of plan is called a *build plan*. This plan will first perform the Get Storage task that is bound to the "Allocate()" operation of the "MyBlockStore" node. Next, the "Install DBMS" task will invoke the "Install()" operation of the "MyDBMS" node, the "Attach Storage" task will invoke the "Attach()" operation of the "MyBlockStore_For_MyDBMS" relationship, and so on. This explicitly defines how to set up the environment for your "CustomerDB" cloud application. Recall (as sketched in Section 2.1.3), that the implementation artifacts of these operations will be deployed before by the Deployment Manager of the TOSCA container, i.e. they are available in your environment and can be bound to the tasks of the plan during deployment of the plan itself.

2.2.3.2 Note on Advanced Processing of Service Templates

Typically, the topology will be same whether it is processed imperatively or declaratively. But the flavor of declarative processing of a service template may get more advanced making modeling of cloud applications even easier. As describe before, declarative processing infers plans for provisioning and decommissioning of instances of the specified topologies. More advanced declarative processing may be based on nodes of the topology that simply declare requirements on their "downstream" topology. In a supporting TOSCA environment, nodes have been made available that specify their capabilities, allowing the Model Interpreter of the environment to match requirements and capabilities. As a result, topology models including requirements and capabilities will be "auto-completed".

For example, the modified topology of your cloud application may specify just two nodes (see Figure 7, left side): the "CustomerDB" node and the "MyDBMS" node. The latter node is associated with an explicit *requirement* for "BlockStorage", called "BlockStorage_Needed" in the figure. The environment may be aware of a node that explicitly declares a matching *capability*: in the figure the "BlockStore" node is show that declares the "BlockStorage_Offered" capability (and that has been defined to correspond to the "BlockStorage_Needed" requirement). The *Model Interpreter* of a TOSCA environment will then match the requirement and the capability. Thus, it will automatically extend the specified topology accordingly: the topology on the right side of Figure 7 results. Furthermore, it will infer the corresponding build and decommissioning plans as sketched before.

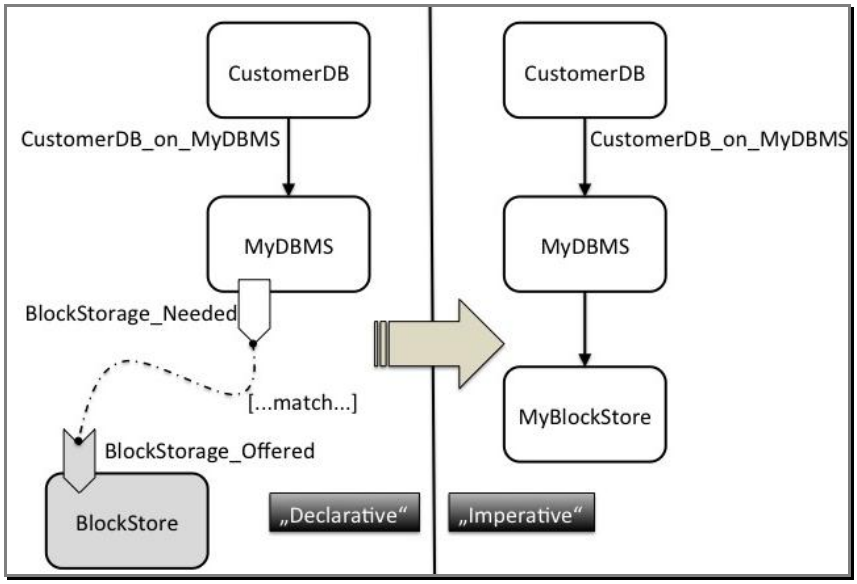

FIGURE 7 - SERVICE TEMPLATE THAT MAKES USE OF REQUIREMENTS AND CAPABILITIES

Declarative processing of service templates that supports derivation and the addition of plans requires a very crisp and precise specification of node types and relationship types by type architects. Also, type architects must follow certain conventions. For example, node types must provide a lifecycle interface that encompasses operations for starting, stopping, etc. instances of the node type. Relationship types must define the order in which their sources and targets must be manipulated. The order in which relationship types have to be considered when "generating" a plan must be defined; for example, the relationship type "DependsOn" must take preference of the relationship type "ConnectsTo". In order to make use of auto-completion features for topologies the type architect must specify requirements and capabilities as well as their matching.

The declarative approach puts much more burden on the type architect, while the imperative approach puts much more burden on the application architect. In practice, more complex cloud applications will be modeled by a combination of both modeling approaches. For example, while the behavior for initial deployment and decommissioning may be inferred in many cases from the topology of a cloud application, plans for granting access rights and establishing security policies can typically not be inferred, i.e. they have to be modeled explicitly. Similarly, defining

precedence rules for relationship types might get complex when the number of such types increase, and a TOSCA *environment* (Section 2.1.3) may not automatically support all such predefined relationship types and the preference rules. In any case, the imperative approach will work because the application architect precisely instructs the TOSCA *environment* how to manage a cloud application (including initial deployment and decommissioning) – of course, interoperability can only be achieved if the plans specified by the application architect only make use of operations defined as part of the associated topology, and no operations are used that are proprietary to a specific environment.

3 What Type Architects Should Know About TOSCA

The *type architect* is an expert for the types of components required by applications as well as the various types of connections between these components. By defining such components (i.e. TOSCA *Node Types*) and connections (i.e. TOSCA *Relationship Types*) the type architect enables an application architect to specify the topology of a cloud application.

Furthermore, the *type architect* specifies types of requirements and capabilities. Such requirement types and capability types are the basis for defining requirements and capabilities of individual node types. In turn, requirements and capabilities of node types eases the correct specification of cloud applications, thus, supporting application architects: for example, graphical modeling tools may check the validity of relationship types used to connect two node templates; or a tool may suggest node types that may be connected to an already chosen node type; or an environment may bind node types to a node of a topology the requirements of which have not been fulfilled yet.

Finally, the *type architect* defines the interfaces of a node type, as well as the source interfaces and target interfaces of relationship types. This is a perquisite for defining all management behavior by application architects: the declarative flavor of processing service templates depends on the existence of certain lifecycle operations for provisioning and decommissioning, and the imperative flavor depends on proper interfaces by binding tasks of management plans to the operations of these interfaces.

3.1 Providing Node Types and Relationship Types

3.1.1 Vendor Perspective

A vendor of components that should become the basis for cloud applications will typically render these components as TOSCA *Node Types*. For example, a vendor that wants to offer their database management system and web server as components to be used by their customers to build cloud applications will define corresponding node types (e.g. a “MyDBMS” node type and a “MyWebServer” node type, respectively). Furthermore, the vendor will also define a corresponding relationship type representing the ability to connect their database management system and their web server. For example, the “MyWebServer_ConnectsTo_MyDBMS” relationship type supports enables the construction of a connection between the DBMS and the web server of the vendor, as well as later destroying an existing connection at decommissioning time.

Often, different vendors offer products of the same type. For example, ”vendor X” as well as “vendor Y” may offer a web server. Both products may have vendor specific aspects like vendor specific operations or properties; thus, two different vendor specific node types will result (e.g. the “XWebServer” and the “YWebServer” node types). Nevertheless, both node types will typically have a lot of commonalities like properties as “ServerName” or “IPAddress”, or the lifecycle interface; thus, a common node type “WebServer” can be defined specifying the

commonalities of all web servers of the different vendors. The vendor specific node types "XWebServer" and "YWebServer" will then inherit from the common "WebServer" node type, which will only define vendor specifics on top of the "WebServer" node type.

3.1.2 Node Types

The following `NodeType` element defines the "ApacheWebServer" node type. It inherits from the "WebServer" node type as specified in the nested `DerivedFrom` element. The Apache-specific properties are defined in the `PropertiesDefinition` element; these properties will typically add new properties compared to the inherited "WebServer" properties but it may also override definitions specified for the "WebServer" node type. The "ApacheWebServer" defines two capabilities defined by separate `CapabilityDefinition` elements. The first capability specifies that an "ApacheWebServer" node may contain any number of Web Application nodes (as indicated by the value "0" of the `lowerBound` attribute and the "unbounded" value of the `upperBound` attribute of the corresponding `CapabilityDefinition` element). Similarly, the second capability specifies that an "ApacheWebServer" may contain any number of modules. Finally, the "ApacheWebServer" defines a single interface, which is the lifecycle interface (see below).

```
<NodeType name="ApacheWebServer">
  <documentation>Apache Web Server</documentation>
  <DerivedFrom typeRef="ns1:WebServer"/>
  <PropertiesDefinition element="tns:ApacheWebServerProperties"/>
  <CapabilityDefinitions>
    <CapabilityDefinition
      capabilityType="tns:ApacheWebApplicationContainerCapability"
      lowerBound="0" name="webapps" upperBound="unbounded"/>
    <CapabilityDefinition
      capabilityType="tns:ApacheModuleContainerCapability"
      lowerBound="0" name="modules" upperBound="unbounded"/>
  </CapabilityDefinitions>
  <Interfaces>
    <Interface name="http://www.example.com/interfaces/lifecycle">
      <Operation name="install"/>
      <Operation name="configure"/>
      <Operation name="start"/>
      <Operation name="stop"/>
      <Operation name="uninstall"/>
    </Interface>
  </Interfaces>
</NodeType>
```

3.1.3 The Lifecycle Interface

The lifecycle interface is defined by the following `Interface` element. Note, that this definition is non-normative. The lifecycle interface defines five operations: the "install" operation will be invoked to install (and, thus, instantiate) an instance of the node type containing this interface. Configuration of the instance is achieved by invoking the "configure" operation. Starting and stopping an instance is done by means of the "start" and "stop" operations. Decommissioning is achieved by invoking the "uninstall" operation.

```
<Interface name="http://www.example.com/interfaces/lifecycle">
```



```
   <Operation name="install"/>
   <Operation name="configure"/>
   <Operation name="start"/>
   <Operation name="stop"/>
   <Operation name="uninstall"/>
</Interface>
```

Note: These lifecycle operations are defined without any input or output parameters. This means that each lifecycle operation "just" defines the effects that can be achieved by it, i.e. the name of a lifecycle operation indicates its semantics. It is expected that every implementation of a lifecycle operation of a particular name understands this semantics and faithfully realizes it. For example, an implementation of the start operation of a web server will start the web server (and not stop it).

Different implementations of an operation of a particular name may expect different input parameters and may produce different output parameters. For example, the configure operation used by a DBMS node type will require different parameters than the "configure" operation of a web server. It is expected that the user of a concrete implementation of a given operation understands which parameters are to be exchanged. The knowledge of the actual parameters exchanged may come from various sources, e.g. via documentation that comes with the implementation.

The actual implementations of the operations of a `NodeType` are specified by a node type implementation (i.e. by a `NodeTypeImplementation` element) provided for each node type. The concrete executable of an operation of a node type is defined by a corresponding `ImplementationArtifact` element of the node type implementation, which in turn references an `ArtifactTemplate` element. This artifact template points to the executable implementing the operation for the node type. More details about these relations are given in Section 4; for a quick overview on these relations see Figure 8.

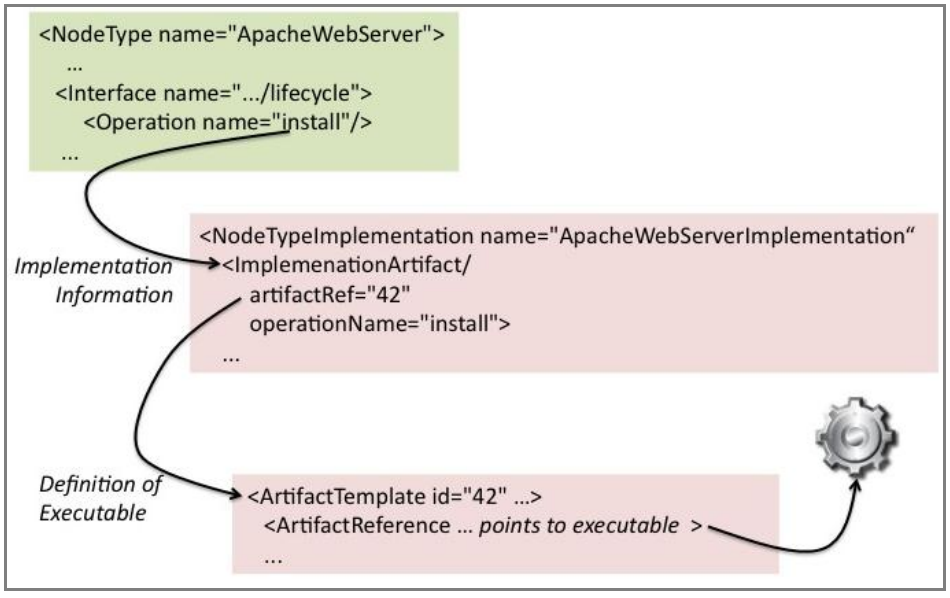

FIGURE 8 – DEFINING INTERFACES AND THEIR IMPLEMENTATIONS FOR PARTICULAR NODE TYPES

Note: Node Types may define additional interfaces that provide operations that go beyond lifecycle capabilities. For example, a DBMS node type may define an

interface that supports the management of data stored by the DBMS; such an interface may include operations for backup and restore tables, creating and dropping indexes and so on.

3.1.4 Relationship Types

The following relationship type defines the ability to establish a connection to a SQL database. This relationship type inherits from the "ConnectsTo" relationship type as specified in the nested DerivedFrom element. The database connection can be established between any node type that requires a connection to a SQL database and any node type that provides the capability of being an SQL database: the `ValidSource` element defines possible source node types (by means of a corresponding requirement type) and the `ValidTarget` element defines the potential target node types (by means of a corresponding `CapabilityDefinition` type). The "MySQLDatabaseConnection" relationship type also defines operations that can be used to act on the source of the relationship: the `SourceInterface` element specifies the "…/ConnectsTo" interface with the "connectTo" operation.

```
<RelationshipType name="MySQLDatabaseConnection">
  <documentation>Connects on</documentation>
  <DerivedFrom typeRef="ns1:ConnectsTo"/>
  <SourceInterfaces>
    <Interface name="http://www.example.com/ToscaBaseTypes/ConnectsTo">
      <Operation name="connectTo"/>
    </Interface>
  </SourceInterfaces>
  <ValidSource typeRef="tns:MySQLDatabaseEndpointRequirement"/>
  <ValidTarget typeRef="tns:MySQLDatabaseEndpointCapability"/>
</RelationshipType>
```

3.2 Using Inheritance

The use of inheritance is well-established and proven as very useful. To support inheritance in TOSCA, each language element that allows defining types (i.e. node types, relationship types, artifact types, requirement types, and capability types) has a nested `DerivedFrom` element that allows supporting super types. In a non-normative manner, hierarchies for all these types have been defined in order to achieve interoperability between TOSCA environments. Figure 9 (below) sketches these hierarchies.

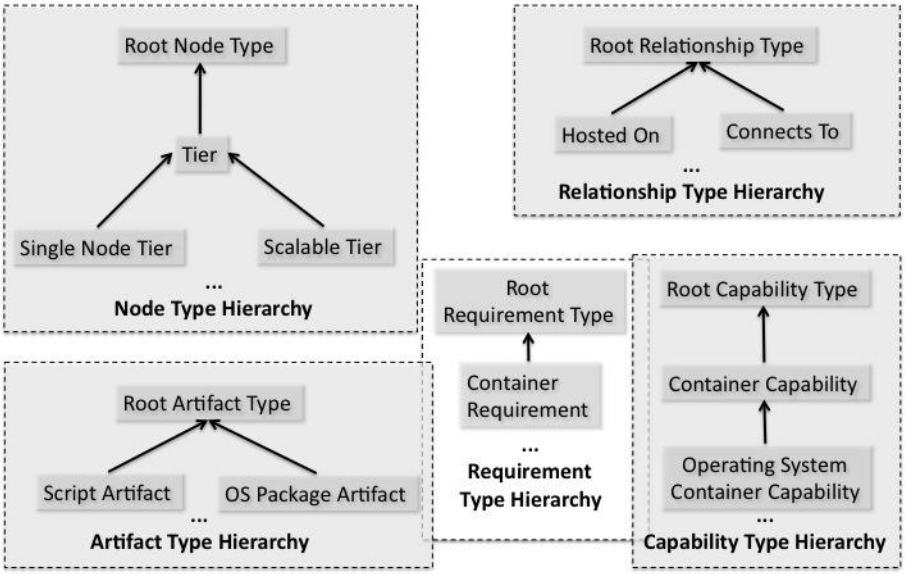

FIGURE 9 – NODE, ARTIFACT, RELATIONSHIP, REQUIREMENTS AND CAPABILITIES TYPE HIERARCHIES

The following "ApacheWebServer" node type inherits from the "WebServer" node type. This is defined by means of a `DerivedFrom` element with the value "WebServer" specified for the `typeRef` attribute.

```
<NodeType name="ApacheWebServer">
  <DerivedFrom typeRef="ns1:WebServer"/>
...
</NodeType>
```

The "WebServer" node type in turn inherits from the "RootNodeType". Since the "ApacheWebServer" node type inherits from the "WebServer" node type, an "ApacheWebServer" node type has both, a "SoftwareContainerRequirement" as well as a "WebApplicationContainerCapability" because this requirement and this capability has been defined with the "WebServer" node type by corresponding nested `RequirementDefinition` and `CapabilityDefinition` elements.

```
<NodeType name="WebServer">
  <documentation>Web Server</documentation>
  <DerivedFrom typeRef="tns:RootNodeType"/>
  <RequirementDefinitions>
    <RequirementDefinition lowerBound="1" name="container"
      requirementType="tns:SoftwareContainerRequirement" upperBound="1"/>
  </RequirementDefinitions>
  <CapabilityDefinitions>
    <CapabilityDefinition
      capabilityType="tns:WebApplicationContainerCapability"
      lowerBound="0" name="webapps" upperBound="unbounded"/>
  </CapabilityDefinitions>
</NodeType>
```

The next two code snippets show inheritance between relationship types (via the `RelationshipType` element). The "MySQLDatabaseConnection" relationship type inherits

from the “ConnectsTo” relationship type, which in turn inherits from the “RootRelationshipType” relationship type.

```
<RelationshipType name="MySQLDatabaseConnection">
  <DerivedFrom typeRef="ns1:ConnectsTo"/>
...
</RelationshipType>
```

Here is the definition of the “ConnectsTo” relationship type:

```
<RelationshipType name="ConnectsTo">
  <documentation>ConnectsTo</documentation>
  <DerivedFrom typeRef="tns:RootRelationshipType"/>
  <ValidSource typeRef="tns:EndpointRequirement"/>
  <ValidTarget typeRef="tns:EndpointCapability"/>
</RelationshipType>
```

The next section will show examples of the use of inheritance for both, requirement types and capability types. Section 4.1 gives examples of inheritance between artifact types.

3.3 Providing Requirement Types and Capability Types

In TOSCA, requirements and capabilities allow to define dependencies between node types. For example, the following “ApacheWebApplicationContainerCapability” capability type allows to express the capability of a node type to serve as a runtime container for an Apache web application; note, that the capability type inherits from the “WebApplicationContainerCapability”. Each node type that includes a `CapabilityDefinition` of this type (as the “ApacheWebServer” node defined above) warrants that it can serve as a container for Apache web applications.

```
<CapabilityType name="ApacheWebApplicationContainerCapability">
  <documentation>Apache Web Application
                 Container Capability</documentation>
  <DerivedFrom typeRef="ns1:WebApplicationContainerCapability"/>
</CapabilityType>
```

In case of the “ApacheWebServer” node type mentioned earlier, the node type actually refines the generic “WebApplicationContainerCapability” inherited from node type “WebServer” by specifying the specialized “ApacheWebApplicationContainerCapability”. It thereby restricts its container capabilities to Apache web applications only, meaning that general web applications may not necessarily run on instances of the node type.

```
<NodeType name="ApacheWebServer">
  <DerivedFrom typeRef="ns1:WebServer"/>
  ...
  <CapabilityDefinitions>
    <CapabilityDefinition
      capabilityType="tns:ApacheWebApplicationContainerCapability"
      lowerBound="0" name="webapps" upperBound="unbounded"/>
    ...
  </CapabilityDefinitions>
```



```
  ...
</NodeType>
```

The XML snippet above shows the refinement of the “webapps” capability inherited from the “WebServer” node type. The capability type is refined to the specialized “ApacheWebApplicationContainerCapability” type which is derived from the more generic capability type “WebApplicationContainerCapability”.

When associating a node type with other node types (i.e. establishing relationship between them) each *requirement* of the source of these associations should be matched by a *capability* of one target of one of the associations. This way, requirements and capabilities support fulfillment of dependencies and help to ensure correctness of the topology of cloud applications. Note, that requirements and capabilities are not intended to express quality-of-services (like availability classes, etc.). Such non-functional properties should be expressed by means of policies (which are currently beyond the scope of this document).

The next “MySQLDatabaseEndpointRequirement” requirement type supports to express that a certain node type requires a database endpoint that provides features of an SQL database system. It inherits from the general requirement for database features. Note, that the requirement type explicitly specifies by which capability type it can be satisfied by means of the `requiredCapabilityType` attribute: this attribute refers to the satisfying `CapabilityType`.

```
<RequirementType name="MySQLDatabaseEndpointRequirement"
           requiredCapabilityType="tns:MySQLDatabaseEndpointCapability">
  <documentation>MySQL Database Endpoint Requirement</documentation>
  <DerivedFrom typeRef="ns1:DatabaseEndpointRequirement"/>
</RequirementType>
```

The following `CapabilityType` definition satisfies the former `RequirementType`:

```
<CapabilityType name="MySQLDatabaseEndpointCapability">
  <documentation>MySQL Database Endpoint Capability</documentation>
  <DerivedFrom typeRef="ns1:DatabaseEndpointCapability"/>
</CapabilityType>
```

> Remark: The semantics of requirement types and capability types are documented together with their formal XML definition. It is assumed that type architects understand this semantics when claiming capabilities of the node types they define; similarly, artifact developers will have to choose appropriate implementations to ensure the warrants made by a type architect about the node type they implement.

Typically, requirement types and capability types are very generic in nature. Thus, it is expected that only few type architects will have to define their own requirement types and capability types but that independent groups like vendor consortia or standardization bodies will define these types.

3.4 Green Field Perspective

It is assumed that for popular application domains all types necessary to define cloud applications in such a domain will be defined by interest groups, consortia, or standardization bodies, for example. Thus, it will often be the case that the types you need will have already been defined and you can use them immediately.

In other cases, you will have to define the necessary types. For example, if you are a vendor of complex applications it is likely that not all types you need to model your application are already defined by others. Or if you want to build cloud applications based on rarely used components, the types required for these components (i.e. node types, requirement types, or capability types) as well the relationship types used to connect your newly defined node types with other node types must be defined by yourself.

3.5 Modular Design of Service Templates

The TOSCA `Definitions` element has a `targetNamespace` attribute. As usual, this allows to scope names of definitions for two main purposes:

1. Avoiding name clashes, and
2. Reusing existing definitions by referring to them by name in their namespace.

In order to define new elements in a particular namespace you create a *Definitions* document, set the value of its `targetNamespace` attribute to this particular value and, then, you specify the new elements in this `Definitions` element. For example, in Figure 10 - Making Use of Imports", the service template defined in the *Definitions* document shown is put into the namespace set by the `Definitions` element.

> Note: that selective TOSCA language elements like service templates, node type, and others, have their own `targetNamespace` attribute to support local specification of namespaces for the names of these elements, but this is an advanced usage that we will not discuss in this document.

The *Definitions* document is then stored somewhere. Typically, it will be made available via a dereferencable URI. This URI is then used as value of the `location` attribute of an `Import` element. A TOSCA *processor* will then make all the definitions of the imported document available in scope of the importing *Definitions* document. This way, the TOSCA `Import` element supports reuse of definitions.

> Note: Not only can TOSCA *Definitions* documents be imported in this manner, but also any other types of definitions can be imported via the same mechanism. To ease processing of imported documents, the `Import` element has an `importType` attribute the value of which indicates the kind of information imported (e.g. an XML schema document, a BPMN document, along with a TOSCA *Definitions* document).

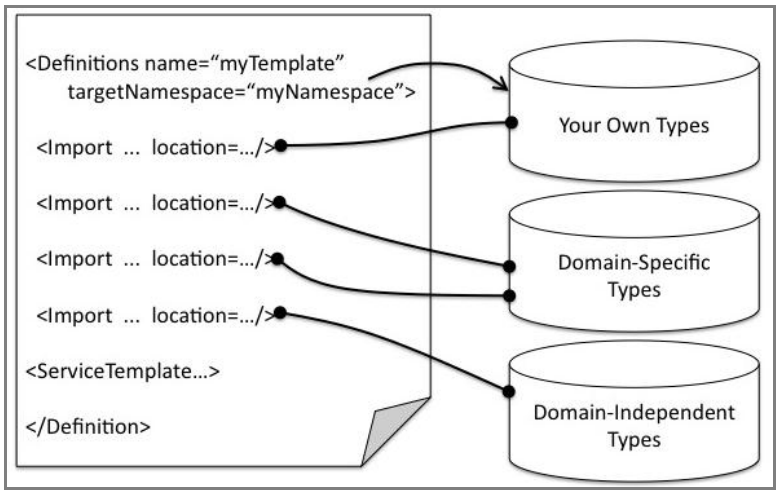

FIGURE 10 - MAKING USE OF IMPORTS

In Figure 10, the *Definitions* document imports types that have been already defined in the own namespace of the *Definitions* document. This way, TOSCA supports a modular approach to define service templates: different kinds of types may be specified in different *Definitions* documents. For example, node types of your application will be defined in a different document than the capability types of your application. This allows type architects with different focus to define their own *Definition* documents and make them available for import into other *Definitions* document. The figure also indicates that all elements newly defined in the *Definitions* document shown will be in the same namespace that contains the own type definitions.

Similarly, the document imports two documents that contain definitions that are domain-specific (for example, node types and relationship types that are useful for web applications) as well as definitions that are domain-independent (for example, a "Server" node type and an "OperatingSystem" node type). Both, the domain-specific definitions and the domain independent definitions have been defined by other groups like interest groups, industry consortia, standardization bodies, or vendors.

Finally, not only types can be defined in particular namespaces and be made available in separate files but also node type implementations and relationship implementations. This allows a vendor to make use of types defined elsewhere to provide the implementations of these types based on their products. For example, "vendor X" may use the "DBMS" node type and provide a definitions document for the "X_DBMS" node type implementation by importing the document containing the "DBMS" node type and adding the vendor specific information to the corresponding node type implementation.

3.6 Simplifying Application Modeling: Composable Service Templates

Remark: This section describes an advanced feature of TOSCA; thus, you may choose to skip this section at a first pass through the document. When you need to express more complex applications using TOSCA as composable service templates, you may want to return here to understand this concept.

3.6.1 Turning Service Templates into Composables

Often, not only individual types can be reused but complete topologies are meaningful in many situations. For example, in most cases a Web server requires persistency by means of an SQL Database system; the connection between the corresponding two node types is achieved via an appropriate relationship type. TOSCA supports to model a corresponding service template and turn this service template into a substitutable for a node type. This is achieved by means of the `BoundaryDefinitions` element: This element contains nested elements that can refer to the constituencies (like node templates etc.) of the service template and "export" corresponding definitions "to the boundary" of the service template. This way, the service template "looks like a node type" and can be used as such in another service template.

In the following code snippet, the `BoundaryDefinitions` element defines the properties of the service template rendered as a node type: each of these properties is defined by a mapping prescription of properties of node templates or relationship templates of making up the topology of the service template. Similarly, individual operations of nodes can be referenced to be visible as operations of the service template rendered as node type.

```
<ServiceTemplate name="WebAppInfrastructure" ...>

  <BoundaryDefinitions>
    <Properties>
      <PropertyMappings>
        <PropertyMapping .../>
    </Properties>

    <Interfaces>
      <Interface name=...>
        <Operation name=...>
          <NodeOperation nodeRef=...
                         interfaceName=...
                         operationName=.../>
        </Operation>
      </Interface>
    </Interfaces>
  </BoundaryDefinitions>

</ServiceTemplate>
```

3.6.2 Using Abstract Node Types

A node types may be defined as abstract by setting the value "yes" for its `abstract` attribute. A node template that is defined based on an abstract node type cannot be instantiated. Thus, in order to be able to instantiate a node template, its node type must be concrete.

One way to turn an abstract node type into a node type that can be used as the basis for an instantiatable node template is to define another node type that is derived from the abstract node type and that is not an abstract node type. This way, abstract node types define common knowledge or best practices about the component services that make up cloud applications and that would need to be refined by more concrete definitions.

3.6.3 Substitution of Abstract Node Types by Service Templates

One way abstract node types can be made concrete is by substituting them by a service template that has been defined as being a substitute for another node type by proper boundary definitions.

A node type can be substituted by a service template. For this purpose, a service template must be turned into a substituting (or composable) service template by defining a `BoundaryDefinitions` element for it (see section 3.6.1). Figure 11 depicts a service template "ST" that exposes selective properties, operations, capabilities, and requirements of some of its ingredients to the outside. This is done by corresponding boundary definitions like `PropertyMappings` for defining the properties visible at the boundary of the service template "ST", or `NodeOperation` elements for exposing operations of interfaces available at the boundary of "ST".

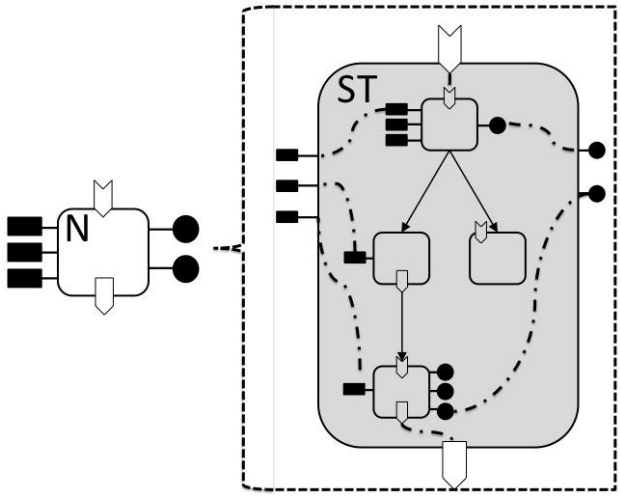

FIGURE 11 - SERVICE TEMPLATE SUBSTITUTING A NODE TYPE

Service template "ST" may substitute node type "N" because the boundary of "ST" matches all defining elements of "N": all properties, operations, requirements and capabilities of "ST" match exactly those of "N". From this perspective, "N" and "ST" are undistinguishable, i.e. the service template "ST" may substitute node type "N". Especially, an abstract node type may be substituted by a service template that matches all the characteristics of the abstract node type. To ease the matchmaking of a service template and a node type the `ServiceTemplate` element may explicitly set its `substitutableNodeType` attribute to the name of the node type it can substitute.

4 What Artifact Developers Should Know About TOSCA

As discussed before, the *artifact developer* is an expert in code artifacts: he is in charge of providing the installables and executables required to instantiate and manage a cloud application. The upper half of Figure 12 depicts the definitions an *artifact developer* has to provide in order to deliver an implementation of the "ApacheWebServer" node type. The figure also shows how these definitions relate to each other.

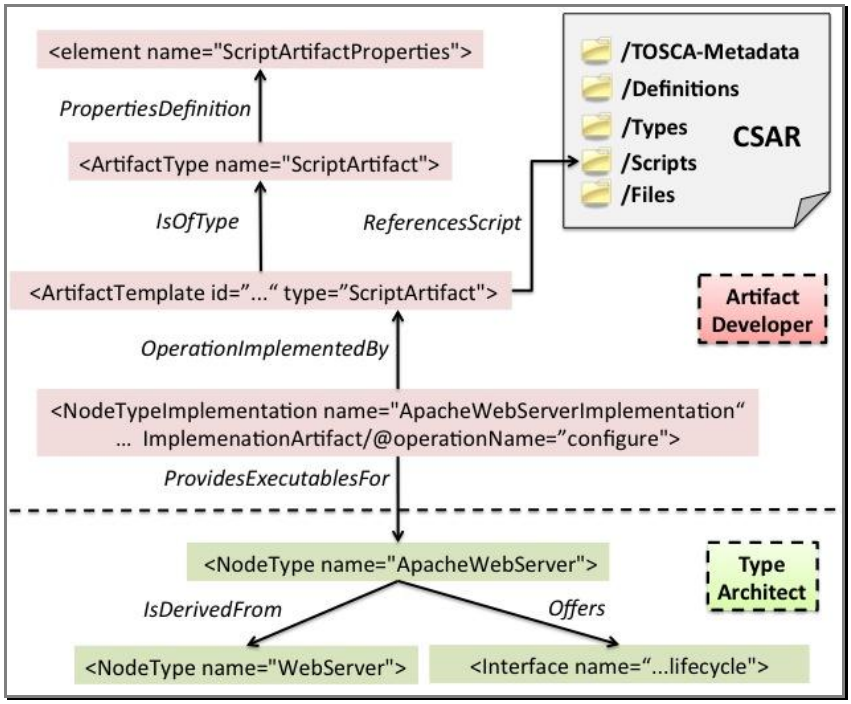

FIGURE 12 - KEY DEFINITIONS FOR TOSCA ARTIFACTS AND THEIR RELATIONSHIPS

4.1 Defining Artifact Types

A TOSCA *Artifact Type* represents the kind of an installable or the kind of an executable. For example, whether the installable is an image or zip file, or the executable is a script or a JEE component.

In order to define the installables and executables required to provision and manage a cloud application, the artifact developer needs to define (or make use of already existing) artifact types. Artifact types are independent of any particular application domain and define the kind of artifacts, e.g. that a particular artifact is a script or is an EJB. Artifact types also define the properties of artifact types. These properties are assumed to be invariant, i.e. an artifact implementation that is based on an artifact type specifies values for the properties that are independent from the concrete use of the artifact implementation. When an artifact implementation is actually used in an implementation artifact or a deployment artifact, additional individual variant information can be added to the invariant data in the artifact specific content field of the corresponding implementation artifact or deployment artifact.

The following (non-normative) `ScriptArtifactProperties` element describes the invariant metadata of an implementation artifacts or a deployment artifact that is a script: a script (according to the `ScriptArtifactProperties` element) always has "ScriptLanguage" and a "PrimaryScript" elements.

```
<xs:element name="ScriptArtifactProperties">
 <xs:complexType>
  <xs:complexContent>
   <xs:extension base="tScriptArtifactProperties"/>
  </xs:complexContent>
 </xs:complexType>
</xs:element>

<xs:complexType name="tScriptArtifactProperties">
 <xs:complexContent>
  <xs:extension base="tExtensibleElements">
   <xs:sequence>
    <xs:element name="ScriptLanguage" type="xs:anyURI"/>
    <xs:element name="PrimaryScript" type="xs:string"/>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

The `ScriptArtifactProperties` element is referenced within the following "ScriptArtifact" `ArtifactType` as part of the `PropertiesDefinition` element's `element` attribute. This artifact type is derived from the "RootArtifactType" representing the common definitional items of all artifact types in our sample domain.

```
 <ArtifactType name="ScriptArtifact">
   <documentation>Script Artifact</documentation>
   <DerivedFrom typeRef="RootArtifactType"/>
   <PropertiesDefinition element="ScriptArtifactProperties"/>
 </ArtifactType>
```

4.2 Defining Artifact Templates

A TOSCA *Artifact Template* represents a concrete executable or installable. It provides actual properties of the installable or executable (like the language of a script) and a reference where to find the executable or the installable (like a link into a CSAR).

An `ArtifactTemplate` is based (by means of its `type` attribute) on an artifact type from which it gets the kind of properties the values of which have to be specified for the concrete executable or installable in its `Properties` element. The concrete executable or installable itself is referenced in its `ArtifactReference` element: typically, the reference will be to an object of the CSAR containing the artifact template itself, or via a dereferencable URL pointing to the location where the artifact can be downloaded.

The following `ArtifactTemplate` specifies the information about the "configure.sh" script. The properties of this script say that it is a shell script (via the 'sh' value of the `ScriptLanguage` element), and that its primary script is "scripts/ApacheWebServer/configure.sh" (via the `PrimaryScript` element). The `reference` attribute of the `ArtifactReference` element points to the "scripts/ApacheWebServer"

directory of the CSAR including the artifact template; path expressions are always interpreted as starting from the CSAR as the root of the directory tree. The `Include` element, nested in the `ArtifactReference` element, specifies that the artifact template consists of exactly the "configure.sh" script of that directory.

```
<ArtifactTemplate id="314"
                  type="ns1:ScriptArtifact">
  <Properties>
    <ns1:ScriptArtifactProperties
      xmlns:ns1="http://www.example.com/ToscaBaseTypes"
      xmlns="http://www.example.com/ToscaBaseTypes">
      <ScriptLanguage>sh</ScriptLanguage>
      <PrimaryScript>scripts/ApacheWebServer/configure.sh</PrimaryScript>
    </ns1:ScriptArtifactProperties>
  </Properties>
  <ArtifactReferences>
    <ArtifactReference reference="scripts/ApacheWebServer">
      <Include pattern="configure.sh"/>
    </ArtifactReference>
  </ArtifactReferences>
</ArtifactTemplate>
```

Let us take a look at the pseudo-code for a different TOSCA *Artifact Template* (listed below). It is an artifact template that consists of two scripts, the "connectToDatabase.sh" script and the "runSilentInstall.sh" script. Both scripts are shell scripts (have an 'sh' value in the `ScriptLanguage` element), and the "connectToDatabase.sh" script is the primary script (specified in the `PrimaryScript` element); and the "runSilentInstall.sh" script is needed by the "connectToDatabase.sh script" under certain conditions (you need to take a look at the contents of the corresponding script to understand those conditions). The `ArtifactReference` element points to the "scripts/MySQLDatabaseConnection" directory of the including CSAR (via its `reference` attribute) and selects the "connectToDatabase.sh" script and the "runSilentInstall.sh" script from that directory (by means of the nested `Include` elements).

```
<ArtifactTemplate id="271"
                  type="ns1:ScriptArtifact">
  <Properties>
    <ns1:ScriptArtifactProperties
      xmlns:ns1="http://www.exemple.com/ToscaBaseTypes"
      xmlns="http://www.example.com/ToscaBaseTypes">
    <ScriptLanguage>sh</ScriptLanguage>
    <PrimaryScript>scripts/MySQLDatabaseConnection/connectToDatabase.sh
    </PrimaryScript>
    </ns1:ScriptArtifactProperties>
  </Properties>
  <ArtifactReferences>
    <ArtifactReference reference="scripts/MySQLDatabaseConnection">
      <Include pattern="connectToDatabase.sh"/>
      <Include pattern="runSilentInstall.sh"/>
    </ArtifactReference>
  </ArtifactReferences>
</ArtifactTemplate>
```


4.3 Providing Implementations

A TOSCA *Node Type Implementation* or *Relationship Type Implementation* provided the executables of the operations of the interfaces of the corresponding node or relationship types, respectively, as well as the required installables. The executables of the operations are defined as *Implementation Artifacts*, and the installables are defined as *Deployment Artifacts*: a node type implementation (or relationship type implementation) bundles these artifacts into a single unit.

The following `NodeTypeImplementation` bundles the `ImplementationArtifacts` for all the operations of the "ApacheWebServer" node type (as referenced by its `nodeType` attribute) and also provides a reference to an installable, in this case an "OSPackage", as part of its `DeploymentArtifact`. For each `Operation` that appears in the referenced `nodeType`'s `Interface` element, the corresponding `NodeTypeImplementation` must provide a matching `ImplementationArtifact` element. Specifically, each node type's `Operation` has a `name` attribute and this attribute's value helps us locate the corresponding `ImplementationArtifact` which will have an `operationName` attribute with the same value. Once we locate the matching `ImplementationArtifact` via this method, we can use artifact's `artifactRef` and `artifactType` attributes to locate the actual executables.

In the sample code below, each `ImplementationArtifact` provides executables for the lifecycle interface of the "ApacheWebServer" node type (e.g. "install", "configure", "start", "stop" and "uninstall"). Looking at ImplementationArtifact for the "configure" operation (as indicated by its `operationName` attribute), we see that its executable is identified by the `artifactRef` attribute's value "314" which matches some `ArtifactTemplate` (defined elsewhere) that has an `id` attribute also with the value "314".. The `ArtifactTemplate` referenced by its `id`, in this case, would provide the details about a "configure.sh" shell script which implements the "configure" operation

> Note: Although the values for the `artifactRef` of the `NodeTypeImplemention` and the `id` of the `ArtifactTemplate` are simple numbers in our examples, in actual practice these would be UUIDs to prevent collisions of artifact references when composing and publishing service templates for real world consumption.

Finally, the `DeploymentArtifact` element of the node type implementation refers to the artifact template with identifier "23". The referenced installable is an operating system package as indicated by the value of its `artifactType` attribute.

```
<NodeTypeImplementation name="ApacheWebServerImplementation"
                        nodeType="tns:ApacheWebServer">
  <ImplementationArtifacts>
    <ImplementationArtifact
      artifactRef="11"
      artifactType="ns1:ScriptArtifact"
      interfaceName="http://www.example.com/interfaces/lifecycle"
      operationName="install"/>
    <ImplementationArtifact
      artifactRef="314"
      artifactType="ns1:ScriptArtifact"
      interfaceName="http://www.example.com/interfaces/lifecycle"
```



```
      operationName="configure"/>
    <ImplementationArtifact
      artifactRef="13"
      artifactType="ns1:ScriptArtifact"
      interfaceName="http://www.example.com/interfaces/lifecycle"
      operationName="start"/>
    <ImplementationArtifact
      artifactRef="17"
      artifactType="ns1:ScriptArtifact"
      interfaceName="http://www.example.com/interfaces/lifecycle"
      operationName="stop"/>
    <ImplementationArtifact
      artifactRef="19"
      artifactType="ns1:ScriptArtifact"
      interfaceName="http://www.example.com/interfaces/lifecycle"
      operationName="uninstall"/>
  </ImplementationArtifacts>
  <DeploymentArtifacts>
    <DeploymentArtifact
      artifactRef="23"
      artifactType="ns1:OsPackageArtifact"
      name="http-packages"/>
  </DeploymentArtifacts>
</NodeTypeImplementation>
```

Relationship types may have interfaces defined too: namely interfaces that can act on the source of the relationship or on the target of the relationship. Thus, relationship types with interfaces must be realized by corresponding `RelationShipType` elements.

The following `RelationshipTypeImplementation` element provides the implementation artifact for the "connectTo" operation of the "ConnectsTo" interface of the "MySQLDatabaseConnection" relationship type. The corresponding `ImplementationArtifact` element refers to the artifact template with a reference identifier (i.e. the `artifactRef` attribute) of "271". This artifact template bundles the scripts implementing the ability to connect to a particular database (see before).

```
<RelationshipTypeImplementation
      name="MySQLDatabaseConnectionImplementation"
      relationshipType="tns:MySQLDatabaseConnection">
  <ImplementationArtifacts>
    <ImplementationArtifact
      artifactRef="271"
      artifactType="ns1:ScriptArtifact"
      interfaceName="http://www.example.com/ToscaBaseTypes/ConnectsTo"
      operationName="connectTo"/>
  </ImplementationArtifacts>
</RelationshipTypeImplementation>
```

4.3.1 Coping With Environment-Specific Implementations

Implementation artifacts or deployment artifacts may depend on specific features they assume to be available in an environment they are deployed in. For example, the executable of an implementation artifact may make use of specific APIs in order to realize the operation it is associated with. Such dependencies can be expressed by a set of `RequiredContainerFeature` elements. Each such element denotes an individual requirement by means of a URI. An environment that processes a service template uses these

features to determine the node type implementation of a certain node type that is most appropriate for the environment.

```
  <NodeTypeImplementation
name="ApacheWebServerSpecialImplementation"
                        nodeType="tns:ApacheWebServer">
    <RequiredContainerFeatures>
      <RequiredContainerFeature
        feature="http://www.example.com/RequiredFeatures/MyImageLibrary"/>
    </RequiredContainerFeatures>
    <ImplementationArtifacts>
          ...
    </ImplementationArtifacts>
    <DeploymentArtifacts>
          ...
    </DeploymentArtifacts>
  </NodeTypeImplementation>
```


5 What Cloud Service Providers Should Know About TOSCA

5.1 Adaptation to Particular Cloud Providers

While TOSCA's goal is the definition of cloud services in a portable manner, i.e. independent of the particular building blocks used in the actual cloud environment of a provider implementing a TOSCA compliant cloud, there is a gap to bridge between the service definition itself and the concrete infrastructure components to be used to run the instances of the cloud service. The closer the elements of the service definition are related to the infrastructure, the harder it is to keep them described in a portable way. The approach of TOSCA is to use appropriate and well defined abstractions to allow portable definitions. Examples of such abstractions are J2EE application servers, which – with different implementations – provide standardized abstractions to be used by compliant applications, allowing them to ignore potentially significantly differing aspects in the underlying operating system.

While this example shows, that these abstractions are typically used in hierarchical layers (with the application being a layer on top of the application server, itself being a layer on top of the operating system, which itself is an abstraction layer on top of the hardware), it also gets clear this abstraction is more difficult to achieve as you get closer to particular hardware or infrastructure, because the abstractions and associated standards are not so well known and common, or despite of a well-defined, common standard, the actually used implementation might vary significantly.

To solve this issue, some aspects or elements of a TOSCA cloud service definition need to be mapped to the concrete elements used in the deployment of a service provider. TOSCA foresees a few ways to achieve this mapping:

5.1.1 Deployment of implementation artifacts and deployment artifacts

In order to allow a service definition to accommodate for different elements in the environment, the CSAR might contain so-called implementation artifacts or deployment artifacts.

Deployment artifacts essentially contain (software) elements, which are needed to implement the cloud service, for example they may contain the necessary J2EE modules for an application, or a particular software application. These software elements might be dependent on the actually available underlying abstractions, therefore the CSAR may contain multiple variations of the artifact to match the actually used abstractions, if needed for a particular CSP. For example, a CSAR might contain as a deployment artifact multiple different images (or references to those) for the different target cloud provider environments containing the "same" software content (e.g. Apache).

TOSCA *Implementation Artifacts* are routines which are needed for executing activities in the course of managing the cloud service, for example to instantiate a service (aka “provisioning”), adapting it to the used environment. Again, one CSAR may contain multiple variations of an implementation artifact to address different infrastructure elements or services available in the concrete deployment environment. As an example, the provisioning of additional storage capacity independent of the VM image might use different variations of an implementation artifact, e.g. one using OpenStack compliant interfaces, another using Amazon EC2 compliant interfaces (as different abstractions) depending on the underlying provider infrastructure.

More information about deployment artifacts and implementation artifacts is available in Section 4.

6 What Application Architects Should Know About TOSCA

This chapter introduces a cloud application that will serve as the "hello world" example for incrementally describing some essential TOSCA concepts against a practical, real-world application and expressing them using the TOSCA standard. This example may, at first glance appear simple; however, it allows us to introduce some powerful capabilities that TOSCA enables for the application architect. We will emphasize the declarative approach in defining the application throughout this example in order to highlight the interoperability and reusability possibilities of modeling using TOSCA.

First, we will show how TOSCA modeling can describe the logical groupings (parts) of an application that should be deployed, configured and managed together as "tiers". This same grouping concept can also be used to communicate the scaling policies that apply to each tier so that they can be independently scaled by cloud service providers to accommodate variations in consumer demand.

Specifically, the cloud application in this example will consist of two tiers. One tier will describe a typical Customer Relationship Management (CRM) web application which we will call the "web application tier". The other tier will describe a SQL Database, or "database tier", which stores the actual customer relational data the CRM application will connect to and use.

We will first show how the "database tier" can be described as layered, granular set of service components, or TOSCA nodes, each declaring the hosting capabilities they offer which can be matched to the requirements of other layers by the provider. We then take the cloud developer through creating the granular description of the web application stack (or "tier") in a manner that permits them to be portable across different cloud service providers (even those that may offer different underlying service models).

Finally, we show how to connect the "web application tier" to the database within the "database tier" using a "ConnectsTo" relationship type that has its own connection-specific requirements and capabilities for matching node templates and which itself can also carry a set of configurable connection properties.

6.1 Single-Tier MySQL Database for our SugarCRM Web Application

Most every web application interacts with a Database Management System (DMBS) that serves as the service layer on-top of a database that holds the domain-specific data the application is concerned with in its "day to day" operations. This section begins by showing how TOSCA can be used to model this common application dependency, specifically using a MySQL database, so that it can be interoperably deployed and referenced on clouds that support TOSCA. We then show how we can customize this generalized "database stack" into a "database tier" that our example's *SugarCRM Web Application* tier (see Section 6.2) can eventually reference from its own TOSCA types, templates and artifacts.

6.1.1 Required Node Types

When modeling using TOSCA, one of the first things you would need to do is to identify the logical set of component services that the application is both composed of as well as those services that the application relies upon for deployment, installation, execution and other lifecycle operations. Each of these components would be represented as a TOSCA node type each exporting an associated, configurable set of properties (or settings) as well as their invocable operations.

Of course, the node types used to describe these components and their dependencies may be fully defined within a single TOSCA service template itself; however, TOSCA permits node types in one service template to reference nodes types defined externally by some other party. These external types could be developed and published by infrastructure, platform, middleware and software service providers so they are made widely available to cloud application architects. This is done through the importing of *Definitions* documents as described in Section 3.5 "Modular Design of Service Templates".

6.1.1.1 Define the Required Base Node Types

The first step a cloud developer should take when modeling their cloud application is to identify the basic set of components services their application is comprised of. Most web applications require some form of database along with the necessary components to deploy, host and manage it over its lifecycle. The cloud application developer would then need to describe these basic components, their interfaces and properties as TOSCA *Node Types*.

The following table describes the basic or "base" node types that we will use to compose a typical database components "stack" using TOSCA:

TABLE 2 – SINGLE-TIER MYSQL DATABASE EXAMPLE'S BASE NODE TYPES

Base Node Type Name	Description
Database	Represents an abstract *Database* along with basic structure (schema), properties and operations.
Database Management Service (DBMS)	Represents an abstract *Database Management Service (DBMS)* and its basic operations.
Operating System (OS)	Represents an abstract notion of an *Operating System* service (or *Platform Service*).
Server	Represents an abstract notion of a "compute" service along with its properties and operations. For example, this could be further developed to describe a Virtual Machine (VM), a Virtual Processing Unit (VPU) or an actual CPU when describing the TOSCA cloud application to a cloud service

	provider.
Tier	Represents an abstract grouping concept for other node types, typically used to describe a stack of software (nodes) that should be managed and/or scaled together "as a *Tier*".

These "base" node types derive from the TOSCA schema's "root node" and would be represented in pseudo-XML as follows:

```
<NodeType name="Database">
  <documentation>A basic Database</documentation>
  <DerivedFrom typeRef="tns:RootNodeType"/>
  ...
</NodeType>

<NodeType name="DBMS">
  <documentation> A basic Database Management System </documentation>
  <DerivedFrom typeRef="tns:RootNodeType"/>
  ...
</NodeType>

<NodeType name="OperatingSystem">
  <documentation> A basic operating system/platform </documentation>
  <DerivedFrom typeRef="tns:RootNodeType"/>
  ...
</NodeType>

<NodeType name="Server">
  <documentation> A basic cloud compute resource</documentation>
  <DerivedFrom typeRef="tns:RootNodeType"/>
  ...
</NodeType>

<NodeType name="Tier">
  <documentation>Tier</documentation>
  <DerivedFrom typeRef="tns:RootNodeType"/>
  ...
</NodeType>
```

Again, here we simply show the basic type definitions; other required elements of the TOSCA `NodeType` element's definition are discussed in later sections.

As you can see, many of these base node types which will serve the basic building blocks for our example's database tier might also be used when modeling web applications as well. In fact, in the following section (section 6.2) will show how the "Tier", "Server", and "Operating System" base node types are indeed referenced again as the "building blocks" for modeling the web application components for a "web application tier".

Additionally, these "base" node types allow us to conceptually "collapse" complex middleware service models into basic, well-understood abstract node types cloud service providers might accept and understand when deploying interoperable TOSCA applications. This concept would allow developers to focus on modeling their application's specific components, dependencies and other details (in a declarative way) while not immediately concerning themselves with all

the possible implementations cloud service providers may choose to employ when orchestrating their application.

In the next sections, we will show how the application architect can easily use these base types to derive more specific (and even customized) node types that further describe the granular details for their actual database components and supporting services.

As the TOSCA standard evolves, these "base" application node types would ideally be offered as commonly defined TOSCA types (within a TOSCA *Definitions* document) and their orchestration would be well-understood by multiple cloud service providers that encounter them. These node types could include additional normative information including basic properties, capabilities and operations common to services of that type. For example, any derivation of the *DBMS* base node type might include some standardized set of properties, interfaces and policy requirements common to most database management services, as well as exporting a typical database service container's hosting capabilities.

6.1.1.2 Define the Required Specific Node Types

Now that we have identified the "base" node types for modeling a database tier, the next step for the developer would be to extend these types to more clearly represent the specific middleware (and other software) our application actually uses. These new types would derive directly from the "base" types we defined above and provide this next level of detail that specialize them for "specific" software. To exhibit this concept, let's further assume that the company's web application persists its customer data in a MySQL database which needs to be described using a "specific" node type that builds upon the "base" database node type.

The following table describes the specific node types that would be needed to support a MySQL database example along with the base types they extend from:

TABLE 3 – SINGLE-TIER MYSQL DATABASE EXAMPLE'S SPECIFIC NODE TYPES

Specific Node Type Name	Extends Node Type	Description
MySQL (DBMS)	**Database Management Service (DBMS)**	Represents a specialized MySQL Database Management Service (i.e. a specialized "DBMS" node type).
MySQL Database	**Database**	Represents a node type "Database" node type specialized for MySQL.

The pseudo-XML code sample below shows how these special node types for the MySQL database service and MySQL database components (i.e. the "MySQLDBMS" and "MySQLDatabase" node types respectively) might be defined. These are shown to derive from the base node types we introduced in the previous section (using the `typeRef` attribute where

"ns1" represents the fully-qualified `targetnamespace` for those "base" node types and their *Definitions* document):

```
<NodeType name="MySQLDBMS">
  <documentation>MySQL</documentation>
  <DerivedFrom typeRef="ns1:DBMS"/>
  <PropertiesDefinition element="tns:MySQLProperties"/>
  <Interfaces>
    <Interface name="http://www.example.com/interfaces/lifecycle">
      <Operation name="install"/>
      <Operation name="configure"/>
      <Operation name="start"/>
      <Operation name="stop"/>
      <Operation name="uninstall"/>
    </Interface>
  </Interfaces>
</NodeType>

<NodeType name="MySQLDatabase">
  <documentation>MySQL Database</documentation>
  <DerivedFrom typeRef="ns1:Database"/>
</NodeType>
```

It is envisioned, that as TOSCA gains adoption, middleware and cloud service providers would develop, publish and maintain normative "specialized" node type *Definitions* documents that describe their component services using TOSCA. Additionally, their types would be derived from standardized TOSCA "base" node types (as described in the previous section). Creating a TOSCA type ecosystem in this manner would further enable cloud application developers to more easily model and compose simple, interoperable cloud applications as TOSCA service templates.

These types, "specific" to MySQL databases, could be published, made available and maintained within a TOSCA *Definitions* document that is offered by the entity that governs MySQL software components and services.

6.1.1.3 Define the Required "Custom" Node Types

The cloud developer, having the necessary set of both "base" and "specific" node types defined for a database tier, is now able to further extend and "customize" them to provide the final details needed for their application. These "custom" node types might include even more properties, interfaces and other details needed to make it suitable for use by their actual application. In our example, the company's application specifically uses an open source Customer Relationship Management (CRM) solution, such as SugarCRM, which may need a suitably customized "MySQL Database" node type for proper operation.

The following table describes the customization of the specific "MySQL Database" node type into one "customized" for a SugarCRM database:

TABLE 4 – SINGLE-TIER MYSQL DATABASE EXAMPLE'S CUSTOM NODE TYPES

Custom Node Type Name	Extends Specific Node Type	Description

SugarCRM Database	**MySQL Database**	Represents the database that is designed to work with the company's SugarCRM application which is a custom derivation of the *"MySQL Database"* type.

The XML for this "custom" `NodeType` definition might look something like:

```
<NodeType name="SugarCRMDatabase">
  <documentation>SugarCRM Database</documentation>
  <DerivedFrom typeRef="ns1:MySQLDatabase"/>
  <PropertiesDefinition element="tns:SugarCRMDatabaseProperties"/>
  <Interfaces>
    <Interface name=" http://www.example.com/interfaces/lifecycle ">
      <Operation name="install"/>
      <Operation name="start"/>
      <Operation name="uninstall"/>
    </Interface>
  </Interfaces>
</NodeType>
```

6.1.1.4 Node Type Inheritance (Base, Specific, Custom)

If we were to look at the node type inheritance hierarchy for the base, specific and custom node types identified above for the "database tier" of our example, it would look as follows:

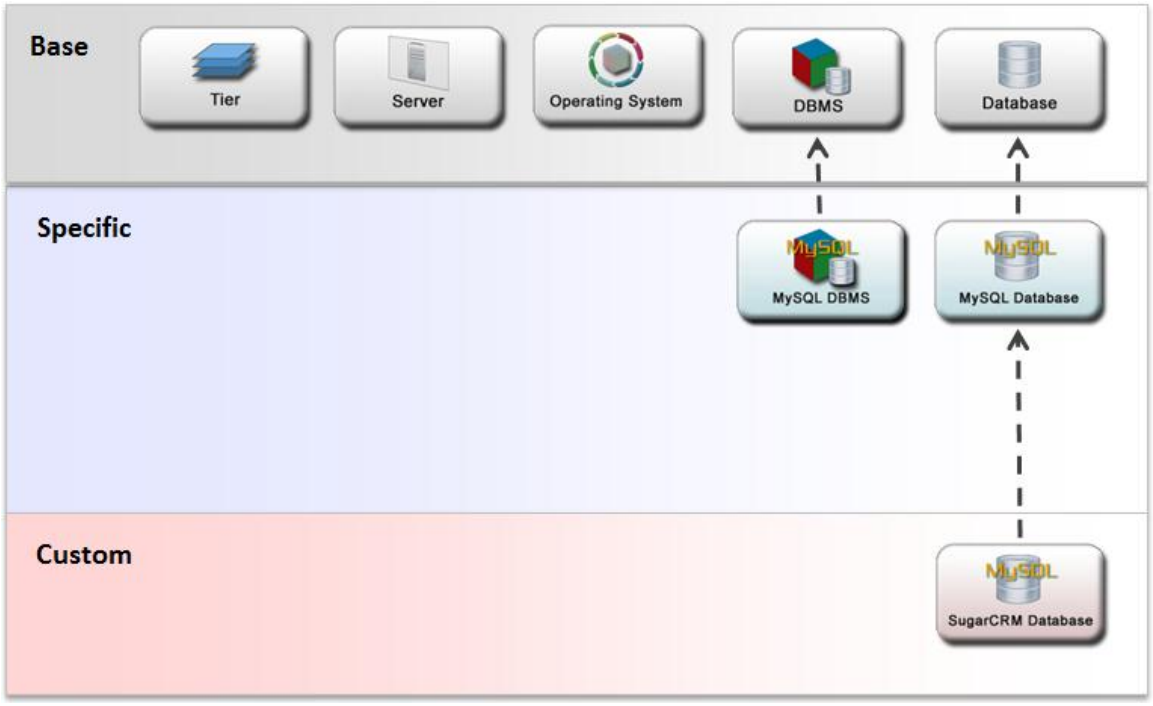

FIGURE 13 - NODE TYPE INHERITANCE FOR A SUGARCRM DATABASE TIER

6.1.2 Turning Node Types into Node Templates

Node types, by themselves simply describe the properties, operations, requirements and capabilities representative of that class of services or software. They are not composable as

TOSCA models; instead TOSCA *Node Templates* are used to turn TOSCA *Node Types* into modelable entities that can be instantiated with specific properties, etc. and related to other node templates to describe, in our current example, the overall SugarCRM MySQL database tier's topology.

The pseudo-XML for the node templates for the database tier's node types would look something like this:

```
  <!– namespaces for the external node type definitions described above -->

  <!– Namespaces for imported TOSCA NodeType Definitions documents -->

  xmlns:ns1="http://www.example.com/ToscaBaseTypes"
  xmlns:ns2="http://www.example.com/ToscaSpecificTypes"
  xmlns:ns3="http://www.example.com/SugarCRMCustomTypes"

  <!-- Define the node templates for the "Database Tier" -->

  <NodeTemplate id="DatabaseTier" name="Database Tier" type="ns1:Tier">
  </NodeTemplate>

  <NodeTemplate id="VmMySql" name="VM for MySQL" type="ns1:Server">
    <Properties>
      <ns1:ServerProperties>
        <NumCpus>1</NumCpus>
        <Memory>1024</Memory>
        <Disk>10</Disk>
      </ns1:ServerProperties>
    </Properties>
  </NodeTemplate>

  <NodeTemplate id="OsMySQL" name="OS for MySQL"
type="ns1:OperatingSystem">
  </NodeTemplate>

  <NodeTemplate id="MySql" name="MySQL" type="ns2:MySQL">
    <Properties>
      <ns2:MySQLProperties>
        <RootPassword>password</RootPassword>
      </ns2:MySQLProperties>
    </Properties>
  </NodeTemplate>

  <NodeTemplate id="SugarCrmDb" name="SugarCRM DB"
type="ns3:SugarCRMDatabase">
    <Properties>
      <ns3:SugarCRMDatabaseProperties>
        <DBName>sugardb</DBName>
        <DBUser>sugaradmin</DBUser>
        <DBPassword>sugaradmin</DBPassword>
        <mySqlPort>3306</mySqlPort>
      </ns3:SugarCRMDatabaseProperties>
    </Properties>
  </NodeTemplate>
```

As you can see, looking at the SugarCRM database's `NodeTemplate` (i.e. "SugarCrmDb") which is based upon the "SugarCRMDatabase" `NodeType` detailed earlier, we see that the template includes specific property settings that the application developer has provided that describes settings to be applied to that server when orchestrated by the TOSCA processing environment:


```
<NodeTemplate id="SugarCrmDb" name="SugarCRM DB"
              type="ns3:SugarCRMDatabase">
  <Properties>
    <ns3:SugarCRMDatabaseProperties>
      <DBName>sugardb</DBName>
      <DBUser>sugaradmin</DBUser>
      <DBPassword>sugaradmin</DBPassword>
      <mySqlPort>3306</mySqlPort>
    </ns3:SugarCRMDatabaseProperties>
  </Properties>
</NodeTemplate>
```

These properties are conveyed to the cloud provider's underlying implementation container software such that the application developer need not understand the implementation details of any particular provider's database container.

Of course, we also need to create node templates for the "custom" node types of our application each with their own custom property settings:

```
<NodeTemplate id="SugarCrmApp" name="SugarCRM App"
              type="ns3:SugarCRMApplication">
  <Properties>
     <ns3:SugarCRMApplicationProperties>
       <SugarCRMKey>somekey</SugarCRMKey>
       <AdminUser>admin</AdminUser>
       <AdminPassword>admin</AdminPassword>
       <DBexists>false</DBexists>
    </ns3:SugarCRMApplicationProperties>
  </Properties>
  ...
</NodeTemplate>

<NodeTemplate id="SugarCrmDb" name="SugarCRM DB"
              type="ns3:SugarCRMDatabase">
  <Properties>
    <ns3:SugarCRMDatabaseProperties>
       <DBName>sugardb</DBName>
       <DBUser>sugaradmin</DBUser>
       <DBPassword>sugaradmin</DBPassword>
       <mySqlPort>3306</mySqlPort>
    </ns3:SugarCRMDatabaseProperties>
  </Properties>
  ...
</NodeTemplate>
```

6.1.3 Required Artifact Types

In order to actually deploy and install a cloud application using a TOSCA service template, the application architect would also need to describe the actual scripts, files, software packages and other types of artifacts that would be used during these first stages of an application lifecycle.

6.1.3.1 Define Required Base Artifact Types

The following table lists what is viewed as some of the common "base" artifact types that are necessary for fully describing our SugarCRM application that could be understood by multiple service providers:

TABLE 5 – SINGLE-TIER MYSQL DATABASE EXAMPLE'S BASE ARTIFACT TYPES

Base Artifact Type Name	Description
File Artifact	Represents artifacts that contain generalized data or metadata that is somehow used or required during an application's lifecycle and encapsulated into a single file.
Script Artifact	Represents artifacts are typically files that encapsulate commands, macros and other instructions that are executed (or interpreted) to perform some operation. These files are often authored using various script programming languages designed for these purposes.
Archive Artifact	Represents artifacts that contain a collection of files that are packaged together for storage or transport between (deployment) locations. Archive artifacts usually contain additional metadata about the files it contains such as a file manifest, filesystem layout and access control information.
Package Artifact	Represents artifacts that contain a collection of files that comprise a complete software application or service which are packaged together for convenient distribution, deployment and/or installation.

These types would look something like this in pseudo-XML using the TOSCA standard:

```
<ArtifactType name="FileArtifact">
  <documentation>File Artifact</documentation>
  <DerivedFrom typeRef="tns:RootArtifactType"/>
</ArtifactType>

<ArtifactType name="ScriptArtifact">
  <documentation>Script Artifact</documentation>
  <DerivedFrom typeRef="tns:RootArtifactType"/>
  <PropertiesDefinition element="tns:ScriptArtifactProperties"/>
</ArtifactType>

<ArtifactType name="ArchiveArtifact">
  <documentation>Archive Artifact</documentation>
  <DerivedFrom typeRef="tns:RootArtifactType"/>
  <PropertiesDefinition element="tns:ArchiveArtifactProperties"/>
</ArtifactType>

<ArtifactType name="PackageArtifact">
  <documentation>Package Artifact</documentation>
  <DerivedFrom typeRef="tns:RootArtifactType"/>
  <PropertiesDefinition element="tns:PackageArtifactProperties"/>
</ArtifactType>
```

Much like "base" node types, these "base" artifact types extend from a TOSCA `RootArtifactType` and they themselves could also be extended to create specialized or custom artifact types. These four base artifact types defined above are suitable for conceivably describing most kinds of software-related artifacts (not just for our current database tier). For example, a *Web Application Archive* (or WAR file) could be defined as a specialized

"*ArchiveArtifact"* type which in turn could be used to create custom application types by web application developers and vendors.

6.1.4 Turning Artifact Types into Artifact Templates

In order to model artifacts in TOSCA, we have to create TOSCA *Artifact Templates* from the *Artifact Types* we have defined above. The following example code shows how an installation script used to install the MySQL database would be represented as an artifact template:

```
<ArtifactTemplate id="uid:at-example1234"
                  type="ns1:ScriptArtifact">
  <Properties>
    <ns1:ScriptArtifactProperties
      xmlns:ns1="http://www.example.com/ToscaBaseTypes">
      <ScriptLanguage>sh</ScriptLanguage>
      <PrimaryScript>scripts/MySQL/install.sh</PrimaryScript>
    </ns1:ScriptArtifactProperties>
  </Properties>
  <ArtifactReferences>
    <ArtifactReference reference="scripts/MySQL">
      <Include pattern="install.sh"/>
    </ArtifactReference>
  </ArtifactReferences>

</ArtifactTemplate>
```

As you can see, this template provides the values for the `ScriptArtifactProperties` and `ArtifactReference` elements that are customized to describe the `ScriptLanguage` as "sh" (an abbreviation for the "shell" script and also its processing utility) along with the name of the script's file and relative location within the CSAR file.

6.1.5 Required Relationship Types

Having identified the set of nodes and artifacts required to express our database tier, we now must be able to describe how these TOSCA modelable components would relate to each other in a cloud deployment. For this purpose, TOSCA introduces the notion of *Relationship Types* which define the "edges" of a TOSCA topology that when turned into TOSCA *Relationship Templates* are able to describe the logical relationships and other dependencies between the application's node templates.

6.1.5.1 Required Base Relationship Types

The database components from our SugarCRM example require just a single base relationship type that provides us the means to express their relationships to one another conceptually in a cloud deployment. This fundamental relationship we need to describe expresses the notion of "containment". In other words, it describes how one node depends on another node to provide its necessary services to manage and host nodes of its type during their lifecycle. This relationship type we will simply call "HostedOn". Specifically from our example, the "SugarCRM Database*"* node would be "HostedOn" the "MySQL DBMS" node which in turn is hosted on an "Operating System" node.

The table below highlights the base relationship type we need to describe the fundamental concept of "containment" when modeling with TOSCA:

TABLE 6 – SINGLE-TIER MYSQL DATABASE EXAMPLE'S BASE RELATIONSHIP TYPES

Base Relationship Type Name	Description
HostedOn	Represents a hosting relationship between two nodes in a service template.

The definition for this base relationship types would look something like this:

```
<RelationshipType name="HostedOn">
  <documentation>Hosted on</documentation>
  <DerivedFrom typeRef="tns:RootRelationshipType"/>
  <ValidSource typeRef="tns:ContainerRequirement"/>
  <ValidTarget typeRef="tns:ContainerCapability"/>
</RelationshipType>
```

6.1.5.2 Specific and Custom Relationship Types

Of course, the base relationship type show above can also be extended (as we showed for node and artifact types above) to create middleware and application (vendor) specific types. Again, these extensions would include additional properties needed to relate components for those respective software offerings and services. For example, the base "HostedOn" type can be extended to create a specific "MySQLDatabaseHostedOnMySQL" `RelationshipType` to better describe the requirements and property settings needed to realize a containment relationship between "MySQLDBMS" and "MySQLDatabase" node templates versus some other middleware provider's database.

```
<RelationshipType name="MySQLDatabaseHostedOnMySQL">
  <documentation>Hosted on</documentation>
  <DerivedFrom typeRef="ns1:HostedOn"/>
  <SourceInterfaces>
    <Interface name="http://www.example.com/ToscaBaseTypes/HostedOn">
      <Operation name="hostOn"/>
    </Interface>
  </SourceInterfaces>
  <ValidSource typeRef="tns:MySQLDatabaseContainerRequirement"/>
  <ValidTarget typeRef="tns:MySQLDatabaseContainerCapability"/>
</RelationshipType>
```

6.1.6 Turning Relationship Types into Relationship Templates

As we discussed for TOSCA *Node Types*, which are turned into TOSCA *Node Templates* so that they can be modeled, TOSCA *Relationship Types* also must be turned into TOSCA *Relationship Templates* in order to use them in a TOSCA service model. *Relationship Templates* represents the "edges" of a TOSCA model and represents an instance of relationship type that it references as part of its definition along with specific property values used during orchestration of the template.

A relationship template, besides being based upon a relationship type, also describes the valid node source and target *Node Templates* these relationships are designed to connect (i.e. via the `SourceElement` and `TargetElement` elements).

For example, the `RelationshipTemplate` shown below is designed to relate our example's "SugarCRM Database" `NodeTemplate` to our "MySQL DMBS" `NodeTemplate`. Specifically, this relationship template would be able to connect a "source" node template that declares it requires a "SugarCrmDb_container" to another node template that exports the capability of containing "MySql_databases":

```
<RelationshipTemplate id="SugarCrmDb_HostedOn_MySql"
  name="hosted on" type="ns2:MySQLDatabaseHostedOnMySQL">
  <SourceElement ref="SugarCrmDb_container"/>
  <TargetElement ref="MySql_databases"/>
</RelationshipTemplate>
```

As we can see the `NodeTemplate` for the "SugarCrmDb" would declare a `Requirement` for a "MySQLDatabaseContainerRequirement" type identified as "SugarCrmDb_container" (which is referenced in the `SourceElement` of the `RelationshipTemplate` shown above):

```
<NodeTemplate id="SugarCrmDb" name="SugarCRM DB"
              type="ns3:SugarCRMDatabase">
  <Properties>
    ...
  </Properties>
  <Requirements>
    <Requirement id="SugarCrmDb_container" name="container"
                 type="ns2:MySQLDatabaseContainerRequirement"/>
  </Requirements>
  <Capabilities>
    ...
  </Capabilities>
</NodeTemplate>
```

And the corresponding "MySQL" `NodeTemplate` would have a `Capability` element that indeed exports a "MySQLDatabaseContainerCapability" type indicating it can host from "MySql_databases" (i.e. MySQL type database nodes):

```
<NodeTemplate id="MySql" name="MySQL" type="ns2:MySQL">
  <Properties>
    <ns2:MySQLProperties>
      ...
    </ns2:MySQLProperties>
  </Properties>
  <Requirements>
    ...
  </Requirements>
  <Capabilities>
    <Capability id="MySql_databases" name="databases"
                type="ns2:MySQLDatabaseContainerCapability"/>
  </Capabilities>
</NodeTemplate>
```

Note: The TOSCA specification enables application architects the means to design their service template to permit providers the ability to either select from a set of similar nodes types that provide the same functionality. For example, the

database tier could include two types of "DBMS" node types allowing the provider to choose the best match for their infrastructure.

6.2 Two-Tier SugarCRM Web Application Example

This section will describe how a cloud application developer would model a basic, two-tier web application using TOSCA. We will describe how to define the types and templates for the nodes, artifacts and relationships needed to describe the components of a "web application tier". Then we will provide a means to connect the "web application tier" to the "database tier" that was fully developed in previous section (i.e. Section 6.1, "Introduction") and already customized for our SugarCRM web application. Finally, we will show how to package all these into a TOSCA CSAR file ready for deployment and installation.

6.2.1 Required Node Types

This section will parallel the same activities that we described when developing the required node types for our database tier (see Section 6.1.1). However, we will attempt to feature only the new types, artifacts, relationships and concepts that are necessary to compose the "web application tier" for our example application.

6.2.1.1 Define the Required Base Node Types

The cloud developer's would again identify the basic set of component nodes needed to describe the significant parts of the web application stack (as we did in Section 6.1.1.1 for the database tier's "base" node types). We would need to define or reference the following "base" TOSCA *Node Types* in order to have the types needed to eventually model a basic web application tier:

TABLE 7 – SUGARCRM WEB APPLICATION EXAMPLE'S BASE NODE TYPES

Base Node Type Name	Description
Web Application	Represents an abstract web application along with basic properties and operations.
Web Server	Represents an abstract service that is capable of hosting and providing management operations for one or more web applications.
Operating System (OS)	*This is the same "base" type as described in Section 6.1.1.1.*
Server	*This is the same "base" type as described in Section 6.1.1.1.*

Tier	*This is the same "base" type as described in Section 6.1.1.1.*

Once again, these base node types would derive from the TOSCA schema's "root node" and be represented in pseudo-XML as follows:

```
<NodeType name="WebApplication">
  <documentation> A basic Web Application </documentation>
  <DerivedFrom typeRef="tns:RootNodeType"/>
  …
</NodeType>

<NodeType name="WebServer">
  <documentation> A basic Web Server </documentation>
  <DerivedFrom typeRef="tns:RootNodeType"/>
  …
</NodeType>

<NodeType name="OperatingSystem">
  <documentation> A basic operating system/platform </documentation>
  <DerivedFrom typeRef="tns:RootNodeType"/>
  …
</NodeType>

<NodeType name="Server">
  <documentation> A basic cloud compute resource</documentation>
  <DerivedFrom typeRef="tns:RootNodeType"/>
  …
</NodeType>
```

The other elements of the node type are discussed in later sections.

> Remark: Ideally, as mentioned above in Section 6.1.1.1, the definitions and treatment for these "base" node types would ideally be standardized and agreed upon by multiple *cloud service providers* so that *cloud application developers*, using the TOSCA standard, could truly compose interoperable TOSCA service templates without worrying about underlying provider implementations of these base services.
>
> For example, any derivation of the "Server" base node type would be expected to have some common properties and definitions of its own service requirements and hosting capabilities by any CSP.

In the following example, we show the "Server" `NodeType` referencing an externally defined set of standardized properties, declaring that it requires a service provider to have a valid "container" to "host" its component (i.e. a "HostedOn" relationship) and that it itself provides the capability to "host" an operating system container (or node).

```
<NodeType name="Server">
  <documentation> A basic cloud compute/execution resource</documentation>
  <DerivedFrom typeRef="tns:RootNodeType"/>
    <PropertiesDefinition element="tns:ServerProperties"/>
    <RequirementDefinitions>
      <RequirementDefinition lowerBound="0" name="container"
        requirementType="tns:ServerContainerRequirement"
upperBound="1"/>
```



```
      …
    </RequirementDefinitions>
    <CapabilityDefinitions>
      <CapabilityDefinition
        capabilityType="tns:OperatingSystemContainerCapability"
        lowerBound="0" name="os" upperBound="1"/>
      …
    </CapabilityDefinitions>
…
</NodeType>
```

Where the externally provided "ServerProperties" complex type would be defined as follows:

```
<xs:complexType name="ServerProperties">
  <xs:sequence>
    <xs:element default="1" name="NumCpus">
      <xs:annotation>
        <xs:documentation xml:lang="en">Number of CPUs</xs:documentation>
      </xs:annotation>
      <xs:simpleType>
        <xs:restriction base="xs:int">
          <xs:enumeration value="1"/>
          <xs:enumeration value="2"/>
          <xs:enumeration value="4"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
    <xs:element name="Memory" type="xs:int">
      <xs:annotation>
        <xs:documentation xml:lang="en">Memory size (in MB)</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="Disk" type="xs:int">
      <xs:annotation>
        <xs:documentation xml:lang="en">Disk size (in GB)</xs:documentation>
      </xs:annotation>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

6.2.1.2 Define the Required Specific Node Types

Let's further assume, for our example, that this company's application specifically uses an open source Customer Relationship Management (CRM) solution, such as SugarCRM which is hosted by an Apache web server (perhaps running on a Linux operating system distribution of some kind). This SugarCRM web application would persist its customer data in a MySQL database much like the one we described earlier in this chapter. Readers will recognize these software components as a typical open-source-based, middleware stack used to host many web application today in the. Sometimes this is referred to as a "***LAMPs stack***" which stands for the component software (Linux, Apache, MySQL and an Apache PHP module) to support the web application.

The following table shows these specific node types needed to describe our web application stack along with the base types they extend:

TABLE 8 – SUGARCRM WEB APPLICATION EXAMPLE'S SPECIFIC NODE TYPES

Specific Node Type Name	Extends Node Type	Description
Apache Web Server	**Web Server**	Represents an Apache specialization of the basic "*Web Server"* node type including any additional properties, operations and capabilities specific to Apache.
Apache Web Application	**Web Application**	Represents an Apache specialization of the basic "*Web Application"* node type including any additional properties, operations and capabilities specific to Apache.
Apache Module	***TOSCA Root***	In this case, the middleware service itself is composed of optional functional components that can be represented as their own node types. Specifically this node represents an Apache-specific node type that describes software modules which are understood and managed by Apache web servers.
Apache PHP Module	**Apache Module**	Represents an "*Apache Module"* node type that specializes in providing PHP Hypertext Processor functionality to "Apache Web Application" nodes.

Below we show how these "specific" *Node Types* for the SugarCRM, Apache and MySQL service related components would appear as pseudo-XML:

```
<NodeType name="ApacheWebServer">
  <documentation>Apache Web Server</documentation>
  <DerivedFrom typeRef="ns1:WebServer"/>
  <PropertiesDefinition element="tns:ApacheWebServerProperties"/>
  <Interfaces>
    <Interface name="http://www.example.com/lifecycle">
      <Operation name="install"/>
      <Operation name="configure"/>
      <Operation name="start"/>
      <Operation name="stop"/>
      <Operation name="uninstall"/>
    </Interface>
  </Interfaces>
</NodeType>

<NodeType name="ApacheWebApplication">
  <documentation>Apache Web Application</documentation>
  <DerivedFrom typeRef="ns1:WebApplication"/>
</NodeType>

<NodeType name="ApacheModule">
  <documentation>Apache Module</documentation>
  <DerivedFrom typeRef="ns1:RootNodeType"/>
</NodeType>
```



```
<NodeType name="ApachePHPModule">
  <documentation>Apache PHP Module</documentation>
  <DerivedFrom typeRef="tns:ApacheModule"/>
  <Interfaces>
    <Interface name="http://www.example.com/lifecycle2">
      <Operation name="start"/>
      <Operation name="uninstall"/>
    </Interface>
  </Interfaces>
</NodeType>
```

As you can see, we also introduced a new node type that derives from the TOSCA root node type for "Apache Module". TOSCA models permit middleware software providers, like Apache, to describe the components of their own specific software as TOSCA models themselves; this enabling even better granular orchestration of these constituent components by CSPs using the TOSCA standard.

6.2.1.3 Define the Required Custom Node Types

The cloud developer, having the necessary set of both "base" and "specific" node types defined for a typical web application stack, is now able to simply extend the appropriate "specific" node types to describe the customizations needed to "stand up" the company's actual SugarCRM application:

TABLE 9 – SUGARCRM WEB APPLICATION EXAMPLE'S CUSTOM NODE TYPES

Custom Node Type Name	Extends Specific Node Type	Description
SugarCRM Application	**Apache Web Application**	Represents the company's actual SugarCRM application service which is a custom derivation of the "*Apache Web Application"* node type.
SugarCRM Database	**MySQL Database**	Represents the database that is designed to work with the company's SugarCRM application which is a custom derivation of the "*MySQL Database"* type.

The pseudo-XML for these custom `NodeTypes` would be something like this:

```
<NodeType name="SugarCRMApplication">
  <documentation>SugarCRM Application</documentation>
  <DerivedFrom typeRef="ns1:ApacheWebApplication"/>
  <PropertiesDefinition element="tns:SugarCRMApplicationProperties"/>
  <Interfaces>
    <Interface name=" http://www.example.com/lifecycle3">
      <Operation name="install"/>
      <Operation name="configure"/>
      <Operation name="start"/>
      <Operation name="uninstall"/>
    </Interface>
  </Interfaces>
```



```
</NodeType>

<NodeType name="SugarCRMDatabase">
  <documentation>SugarCRM Database</documentation>
  <DerivedFrom typeRef="ns1:MySQLDatabase"/>
  <PropertiesDefinition element="tns:SugarCRMDatabaseProperties"/>
  <Interfaces>
    <Interface name=" http://www.example.com/lifecycle4">
      <Operation name="install"/>
      <Operation name="start"/>
      <Operation name="uninstall"/>
    </Interface>
  </Interfaces>
</NodeType>
```

6.2.1.4 Node Type Inheritance (Base, Specific, Custom)

If we were to look at the node type inheritance hierarchy for the base, specific and custom node types identified above for the "web application tier" of our example, it would look as follows:

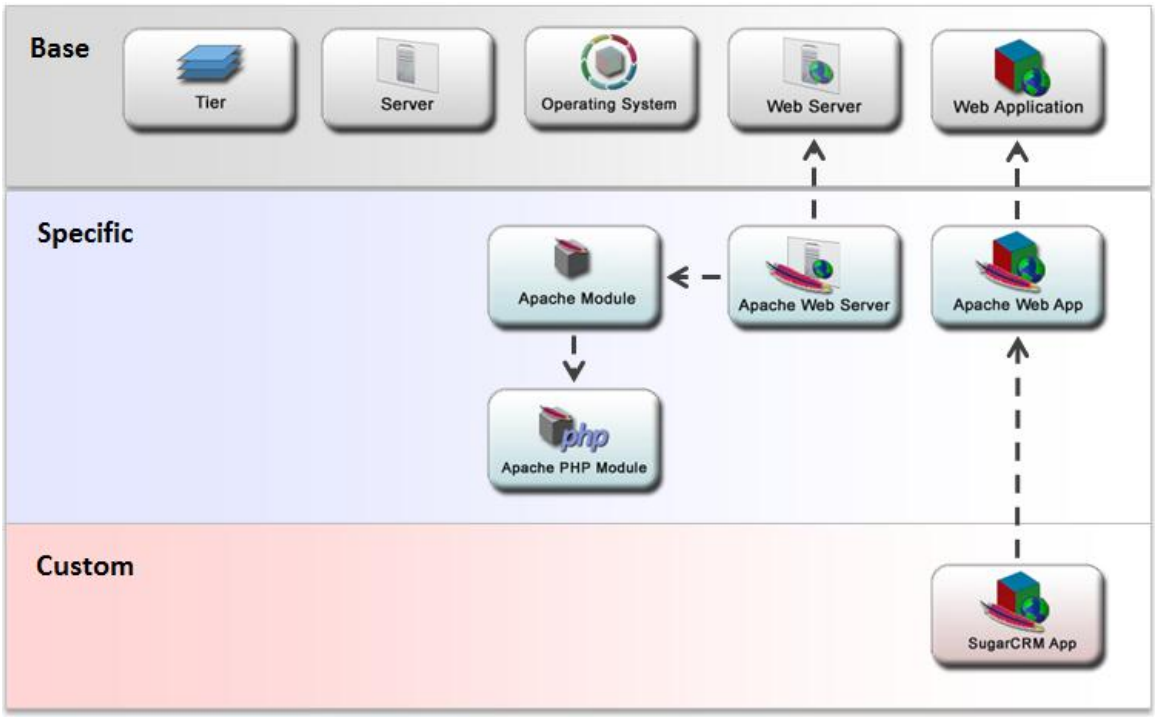

FIGURE 14 - NODE TYPE INHERITANCE FOR A SUGARCRM WEB APPLICATION TIER

6.2.2 Turning Node Types into Node Templates

Node types, by themselves simply describe the properties, operations, requirements and capabilities representative of that class of services or software. They are not composable and cannot be modeled; instead TOSCA Node Templates are used to turn Node Types into component entities that can be instantiated with specific properties, etc. and related to other Node Templates to describe the overall application service's topology.

The pseudo-XML for the node templates for "specific" node types would look something like this:

```
  <!– Namespaces for imported TOSCA NodeType Definitions documents -->

  xmlns:ns1="http://www.example.com/ToscaBaseTypes"
  xmlns:ns2="http://www.example.com/ToscaSpecificTypes"
  xmlns:ns3="http://www.example.com/SugarCRMCustomTypes"

  <!-- Define the node templates for the "Web Tier" -->

  <NodeTemplate id="WebTier" name="Web Tier"
                type="ns1:Tier">
  </NodeTemplate>

  <NodeTemplate id="VmApache" name="VM for Apache Web Server"
                type="ns1:Server">
    <Properties>
      <ns1:ServerProperties>
        <NumCpus>1</NumCpus>
        <Memory>1024</Memory>
        <Disk>10</Disk>
      </ns1:ServerProperties>
    </Properties>
  </NodeTemplate>

  <NodeTemplate id="OsApache" name="OS for Apache"
type="ns1:OperatingSystem">
  </NodeTemplate>

  <NodeTemplate id="ApacheWebServer" name="Apache Web Server"
                type="ns2:ApacheWebServer">
    <Properties>
      <ns2:ApacheWebServerProperties>
        <httpdport>80</httpdport>
      </ns2:ApacheWebServerProperties>
    </Properties>
  </NodeTemplate>

  <NodeTemplate id="SugarCrmApp" name="SugarCRM App"
                type="ns3:SugarCRMApplication">
    <Properties>
      <ns3:SugarCRMApplicationProperties>
        <SugarCRMKey>dummy</SugarCRMKey>
        <AdminUser>admin</AdminUser>
        <AdminPassword>admin</AdminPassword>
      </ns3:SugarCRMApplicationProperties>
    </Properties>
  </NodeTemplate>

  <NodeTemplate id="PhpModule" name="PHP Module"
                type="ns2:ApachePHPModule">
  </NodeTemplate>
```

As you can see, looking at the web server's `NodeTemplate` (i.e. "VmApache") which is based upon the "Server" `NodeType` as described in the previous section, we see that the template includes specific property settings that the application developer has provided that describes settings to be applied to an actual server at a CSP when orchestrated:


```
  <NodeTemplate id="VmApache" name="VM for Apache Web Server"
type="ns1:Server">
    <Properties>
      <ns1:ServerProperties>
       <NumCpus>1</NumCpus>
       <Memory>1024</Memory>
       <Disk>10</Disk>
      </ns1:ServerProperties>
    </Properties>
    ...
  </NodeTemplate>
```

These properties are conveyed to the cloud provider's underlying implementation container software such that the application developer need not understand the implementation details of any particular provider which supports interoperability.

As before, we also need to create node templates for the "custom" node types of our application each with their own custom property settings:

```
  <NodeTemplate id="SugarCrmApp" name="SugarCRM App"
type="ns3:SugarCRMApplication">
    <Properties>
       <ns3:SugarCRMApplicationProperties>
         <SugarCRMKey>somekey</SugarCRMKey>
         <AdminUser>admin</AdminUser>
         <AdminPassword>admin</AdminPassword>
         <DBexists>false</DBexists>
      </ns3:SugarCRMApplicationProperties>
    </Properties>
    ...
  </NodeTemplate>

  <NodeTemplate id="SugarCrmDb" name="SugarCRM DB"
type="ns3:SugarCRMDatabase">
    <Properties>
      <ns3:SugarCRMDatabaseProperties>
         <DBName>sugardb</DBName>
         <DBUser>sugaradmin</DBUser>
         <DBPassword>sugaradmin</DBPassword>
         <mySqlPort>3306</mySqlPort>
      </ns3:SugarCRMDatabaseProperties>
    </Properties>
    ...
  </NodeTemplate>
```

6.2.3 Required Artifact Types

In order to actually deploy and install the web application using a TOSCA service template, the application architect would also need to describe the actual scripts, files, software packages and other types of artifacts that would be used to deploy, install the actual software components for our web application stack.

6.2.3.1 Define Required Base Artifact Types

The same set of base artifact types that we listed and described in Section 6.1.3.1 when we walked through architecting a MySQL "database tier" (i.e. the artifact types "*File Artifact*", "*Script Artifact*", "*Archive Artifact*" and "*Package Artifact*") are the same ones we will use for architecting for deriving the necessary set of artifacts for our web application tier.

6.2.4 Turning Artifact Types into Artifact Templates

The service template for our SugarCRM web application (tier) example would include many different kinds of artifacts that would be needed to not only deploy and install the component software (such as packages, archives, configuration files and policy documents), but also files used at various stages of these components' respective lifecycles and corresponding operations (such as scripts). Below we show an example of a "script" artifact that CSP would use during the "install" operation for the "ApacheWebServer" node:

```
<ArtifactTemplate id="uid:install-xxx" type="ns1:ScriptArtifact">
  <Properties>
    <ns1:ScriptArtifactProperties>
      <ScriptLanguage>sh</ScriptLanguage>
      <PrimaryScript>scripts/ApacheWebServer/install.sh</PrimaryScript>
    </ns1:ScriptArtifactProperties>
  </Properties>
  <ArtifactReferences>
    <ArtifactReference reference="scripts/ApacheWebServer">
      <Include pattern="install.sh"/>
    </ArtifactReference>
  </ArtifactReferences>
</ArtifactTemplate>
```

6.2.5 Required Relationship Types

6.2.5.1 Required Base Relationship Types

In Section 6.1.5.1 "Required Base Relationship Types", we established a base relationship type that describes "containment" which was named "HostedOn". This type is of course essential for describing the essential hosting relationships between the component nodes of our web application stack. However, our SugarCRM example application introduces the need for two additional basic relationships to fully describe the relationships of our application's model.

The first relationship type we need to add to our set of required base types would describe a service that is available through a network endpoint accessible from the cloud's network and is required during some stage of another service's lifecycle. This "ConnectsTo" relationship could either represent a continuous or periodic network connection IP based (typically using an HTTP(s) protocol). From our example, our *SugarCRM Web Application* would "ConnectsTo" the *SugarCRM Database* (our custom MySQL database) over an HTTP connection.

The second type we introduce here is a relationship that describes a more generalized dependency between nodes that we will simple name "DependsOn". This relationship is exhibited by the *SugarCRM Web Application* node which "depends on" the *Apache PHP Module* in order to execute.

The table below highlights the names and descriptions for these additional required “base” relationship types for convenience:

TABLE 10 – SUGARCRM WEB APPLICATION EXAMPLE'S BASE RELATIONSHIP TYPES

Base Relationship Type Name	Description
ConnectsTo	Represents a network connection between two nodes in a service template.
DependsOn	Represents a general dependency relationship between two nodes in a service template.

The definitions for these base relationship types would look something like this:

```
<RelationshipType name="ConnectsTo">
  <documentation>ConnectsTo</documentation>
  <DerivedFrom typeRef="tns:RootRelationshipType"/>
  <ValidSource typeRef="tns:EndpointRequirement"/>
  <ValidTarget typeRef="tns:EndpointCapability"/>
</RelationshipType>

<RelationshipType name="DependsOn">
  <documentation>Depends on</documentation>
  <DerivedFrom typeRef="tns:RootRelationshipType"/>
  <ValidSource typeRef="tns:FeatureRequirement"/>
  <ValidTarget typeRef="tns:FeatureCapability"/>
</RelationshipType>
```

6.2.5.2 Specific and Custom Relationship Types

Of course, the base relationship types shown above in Table 10 can also be extended to create middleware and application (vendor) specific types that have additional properties available for those respective software offerings and services. As was shown in Section 6.1.5.2 when the base “HostedOn” relationship type was used to derive a specific “MySQLDatabaseHostedOnMySQL” relationship type, we can define similar “HostedOn” relationship types for each of the logical containments between node templates for our web application stack. For example, the “Apache PHP Module” would be logically hosted by the “Apache Wen Server”:

```
    <RelationshipTemplate id="PhpModule_HostedOn_Apache"
      name="hosted on" type="ns2:ApacheModuleHostedOnApache">
      <SourceElement ref="PhpModule container"/>
      <TargetElement ref="ApacheWebServer_modules"/>
    </RelationshipTemplate>
```

The base “ConnectsTo” relationship type would be extended to create a “MySQLDatabaseConnection” to better describe the requirements and property settings needed to realize a connection to our desired “MySQL Database” node template versus some other kind of database.

```
<RelationshipType name="MySQLDatabaseConnection">
  <documentation>Connects on</documentation>
```



```
    <DerivedFrom typeRef="ns1:ConnectsTo"/>
    <SourceInterfaces>
      <Interface name="http://www.example.com/ToscaBaseTypes/ConnectsTo">
        <Operation name="connectTo"/>
      </Interface>
    </SourceInterfaces>
    <ValidSource typeRef="tns:MySQLDatabaseEndpointRequirement"/>
    <ValidTarget typeRef="tns:MySQLDatabaseEndpointCapability"/>
  </RelationshipType>
```

6.2.6 Turning Relationship Types into Relationship Templates

As we discussed for TOSCA Node Types, which are turned into TOSCA Node Templates so that they can be modeled, TOSCA Relationship Types also must be turned into TOSCA Relationship Templates in order to use them in a TOSCA service model. The Relationship Template represents the "edges" of a TOSCA model and represents an instance of relationship type that it references as part of its definition along with specific property values used during orchestration of the template.

A relationship template, besides being based upon a relationship type, also describes the valid node source and target Node Templates these relationships are designed to connect (i.e. via the SourceElement and TargetElement elements).

For example, the *RelationshipTemplate* shown below is designed to connect our example's SugarCRM database *NodeTemplate* to our SugarCRM application *NodeTemplate*. Specifically, it can connect "source" node templates that declare they require a named "SugarCrmApp_database" connection to "target" node templates that declare they are capable of accepting "SugarCrmDb_clients":

```
  <RelationshipTemplate id="SugarCrmApp_ConnectsTo_SugarCrmDb"
    name="connects to" type="ns2:MySQLDatabaseConnection">
    <SourceElement ref="SugarCrmApp_database"/>
    <TargetElement ref="SugarCrmDb clients"/>
  </RelationshipTemplate>
```

As we can see the NodeTemplate for the "SugarCrmApp" would declare a *Requirement* for a "MySQLDatabaseEndpointRequirement" type identified as "SugarCrmApp_database" (which is referenced in the *SourceElement* of the *RelationshipTemplate* shown above):

```
      <NodeTemplate id="SugarCrmApp" name="SugarCRM App"
                    type="ns3:SugarCRMApplication">
        ...
        <Requirements>
          <Requirement id="SugarCrmApp phpRuntime" name="phpRuntime"
type="ns2:PHPRuntimeRequirement"/>
          <Requirement id="SugarCrmApp_database" name="database"
type="ns2:MySQLDatabaseEndpointRequirement"/>
          <Requirement id="SugarCrmApp container" name="container"
type="ns2:ApacheWebApplicationContainerRequirement"/>
        </Requirements>
      </NodeTemplate>
```

And the corresponding "SugarCrmDb" NodeTemplate would have a *Capability* element that indeed exports a "MySQLDatabaseEndpointCapability" indicating it can accept connections from "SugarCrmDb_clients":

```
  <NodeTemplate id="SugarCrmDb" name="SugarCRM DB"
type="ns3:SugarCRMDatabase">
    ...
    <Requirements>
      <Requirement id="SugarCrmDb_container" name="container"
type="ns2:MySQLDatabaseContainerRequirement"/>
    </Requirements>
    <Capabilities>
    <Capability id="SugarCrmDb_clients" name="clients"
type="ns2:MySQLDatabaseEndpointCapability"/>
    </Capabilities>

  </NodeTemplate>
```

In addition, we can describe how our "SugarCRM Web Application" node template "DependsOn" an Apache PHP Module in order to properly run the SugarCRM application itself:

```
      <RelationshipTemplate id="SugarCrmApp_DependsOn_PhpModule"
        name="depends on" type="ns1:DependsOn">
        <SourceElement ref="SugarCrmApp_phpRuntime"/>
        <TargetElement ref="PhpModule_phpApps"/>
      </RelationshipTemplate>
```

6.2.7 Creating the Cloud Service Archive (CSAR)

Describing exhaustively the numerous types, templates for the nodes, relationships and artifacts that would be defined and connected to one another to realize a complete TOSCA service template in a "Primer" would be counterproductive. However, we wish to show how the CSAR file would be structured given the expectation for such a set of files and artifacts.

Our example's TOSCA *Service Template* (which contains the complete two-tier model of our SugarCRM application) would have many XML schema files, *Definitions* documents, and artifacts (files) that would be packaged together to create a CSAR file. The CSAR file "SugarCRM-MySQL-Example.CSAR" (named to match our example) would have the following directory structure along with samples of files that contain the types and definitions we described throughout Section 6:

```
SugarCRM-MySQL-Example.CSAR
  /TOSCA-Metadata
    /TOSCA.meta
  /types
     /Artifacts.xsd
     /ToscaBaseTypes.xsd
     /ToscaSpecificTypes.xsd
     /SugarCRMCustomTypes.xsd
  /Definitions
    /ToscaBaseTypes-Definitions.xml
    /ToscaSpecificTypes-Definitions.xml
    /SugarCRMCustomTypes-Definitions.xml
    /SugarCRM-Definitions.xml
  /files
    ... (subdirectories would include various software packages/archives)
  /scripts
    /ApacheModuleHostedOnApache
```



```
    /ApachePHPModule
    /ApacheWebServer
    /MySQL
    /MySQLDatabaseHostedOnMySQL
    /SugarCRMApplication
    /SugarCRMDatabase
```

And the file “TOSCA.meta” contains:

```
TOSCA-Meta-File-Version: 1.0
CSAR-Version: 1.0
Created-By: OASIS TOSCA Interop SC

Name: Definitions/TOSCABaseTypes-Definitions.xml
Content-Type: application/vnd.oasis.tosca.definitions

Name: Definitions/TOSCASpecificTypes-Definitions.xml
Content-Type: application/vnd.oasis.tosca.definitions

Name: Definitions/SugarCRMCustomTypes-Definitions.xml
Content-Type: application/vnd.oasis.tosca.definitions

Name: Definitions/SugarCRM-Definitions.xml
Content-Type: application/vnd.oasis.tosca.definitions
```


7 Moving Virtual Machines to the TOSCA World

A developer's first experience with hosting an application in the Cloud may be simply packaging up an existing application and moving it to a virtual machine (VM) that's hosted remotely (either within their enterprise or on a public Cloud). Typically, the easiest way to do this is to create an image (e.g. an ISO image) of an entire machine and then unpackaging it onto a Cloud-hosted virtual machine.

This section describes how virtual machines can be deployed via TOSCA.

7.1 Deploying a New Virtual Machine (VM)

In this scenario, the deployment artifact being described by a TOSCA CSAR file will be a virtual machine image. The container into which it will be deployed will be a Machine (as defined by CIMI).

7.1.1 Required Node Types

This scenario requires the cloud service provider (CSP) to define one `NodeType` that will be supported by its TOSCA implementation. This `NodeType`, called a “Machine”, defines the virtual hardware characteristics of the new virtual machine - the actual properties of the “Machine” will be in a complex type called "MachineTemplate". Below is an example of what a cloud service provider might advertise:

```
<NodeType name="Machine">
  <documentation> A new virtual machine as defined by CIMI </documention>
  <DerivedFrom typeRef="tns:RootNodeType"/>
  <PropertiesDefinition element="tns:MachineTemplate"/>
</NodeType>
```

Where "MachineTemplate" is defined as follows:

```
<Definitions id="CSPTypes" targetNamespace="http://mycsp.com/toscaTypes"
  xmlns="http://docs.oasis-open.org/tosca/ns/2011/12"
  xmlns:tbase="http://example.com/ToscaBaseTypes"
  xmlns:tns="http://mycsp.com/toscaTypes">

  <Types>

    <xs:complexType name="MachineTemplate">
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="description" type="xs:string"/>
        <xs:complexType name="MachineConfiguration">
          <xs:sequence>
            <xs:element name="cpu" type="xs:integer"/>
            <xs:element name="memory" type="xs:integer"/>
            <xs:complexType name="disk" minOccurs="0" maxOccurs="unbounded">
            <xs:sequence>
              <xs:element name="capacity" type="xs:integer"/>
              <xs:element name="format" type="xs:string"/>
              <xs:element name="initialLocation" type="xs:string"/>
            </xs:sequence>
            </xs:complextType>
```



```
            <xs:element name="cpuArch" type="xs:string" minOccurs="0"/>
            <xs:element name="cpuSpeed" type="xs:integer" minOccurs="0"/>
          </xs:sequence>
        </xs:complexType>
      </xs:sequence>
    </xs:complexType>

  </Types>

  <NodeType name="Machine">
    <documentation> A new virtual machine as defined by CIMI </documention>
    <DerivedFrom typeRef="tbase:RootNodeType"/>
    <PropertiesDefinition element="tns:MachineTemplate"/>
  </NodeType>

</Definitions>
```

7.1.2 Creating the Service Template xml file

Given the above `NodeType`, a service template XML file, called "Machine.xml", can then be created by the application developer, filling in all of the properties of the new machine as well as including a reference to the ISO image file containing the application:

```
<Definitions xmlns="http://docs.oasis-open.org/tosca/ns/2011/12"
    xmlns:csp="http://mycsp.com/toscaTypes"
    xmlns:tns="http://example.com/myApp"
    targetNamespace="http://example.com/myApp" >

  <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
          namespace="http://mycsp.com/tosca/toscaTypes" />

  <ArtifactTemplate id="myImage" type="csp:ISOImageArtifact">
    <ArtifactReferences>
      <ArtifactReference reference="http://example.com/myISOs/machine1.ISO"/>
    </ArtifactReferences>
  </ArtifactTemplate>

  <ServiceTemplate name="MyFirstMachine">
    <TopologyTemplate>
      ...
      <NodeTemplate type="csp:Machine">
        <Properties>
          <csp:MachineTemplate xmlns="http://mycsp.com/toscaTypes">
            <name> MyMachine </name>
            <description> My First Machine </description>
            <MachineConfiguration>
              <cpu>4</cpu>
              <memory>64000</memory>
              <disk>
                <capacity>512000</capacity>
                <format>NTFS</format>
                <initialLocation>C:</initialLocation>
              </disk>
            </MachineConfiguration>
          </csp:MachineTemplate>
        </Properties>
        <DeploymentArtifact
          artifactRef="tns:myImage"
          artifactType="csp:ISOImageArtifact" />
      </NodeTemplate>
      ...
    </TopologyTemplate>
  </ServiceTemplate>
```

```
</Definitions>
```

In this XML file there are a couple of things being defined:

- In the Import statement there is no `Location` attribute specified. In this case the location of the imported *Definitions* (document) will be inherently known by the Cloud Service Provider.

- In the `Definitions` element itself a couple of XML namespaces are defined. The default namespace is defined as the one from the TOSCA specification itself, this tells us that most (but not all) of the XML elements are defined by the TOSCA specification. The "csp" namespace will be used for XML elements, or `NodeTypes`, that are defined by the cloud service provider and will be available for the application to use. The "tns" namespace are for application owned/defined entities - such as the name of the service template.

- Next is the `ArtifactTemplate` element. This element defines the type of artifact (in this case an ISO image) and includes a reference to where the image can be retrieved - in this case "http://example.com/myISO/machine1.ISO". How this ISO file was created and placed at this URL is out of scope for this document.

- And finally in the `ServiceTemplate` element we bind the ISO image to a particular “Machine”. In order to do this we need two pieces of information. First, we need the configuration parameters of the new “Machine” that is to be created. In this case, the application is asking for a new virtual machine with 4 CPUs, 64 megabytes of memory and one ephemeral disk (with 512 megabytes, formatted with NTFS, available as the "C" drive). Notice that the XML elements that define the characteristics of the “Machine” are not in the same XML namespace as the rest of the XML document. Since the cloud service provider, and not the TOSCA specification, defined what a "Machine" `NodeType` looked like, those elements are in the "csp" namespace.

  Second, we need to tell the cloud service provider which ISO file to use when creating the new machine - in other words, we need to reference the deployment artifact we defined at the top of the XML file.

Creating the Cloud Service Archive (CSAR) file

This service template can now be used to create a CSAR file, called “myFirstMachine.CSAR”, with the following structure:

```
myFirstMachine.CSAR
  /TOSCA-Metadata
    /TOSCA.meta
  /Definitions
    /Machine.xml
```

Where “TOSCA.meta” contains:


```
TOSCA-Meta-File-Version: 1.0
CSAR-Version: 1.0
Created-By: Joe Smith
```

This CSAR file can then be given to a TOSCA provider which will then deploy the application on a new virtual machine.

8 How TOSCA Works with Other Cloud Standards

8.1 Mapping TOSCA to DMTF OVF

The deployment artifact of a node (i.e. a TOSCA *Node Template* or a *Node Type Implementation*) in a TOSCA *Service Template* can be represented by an image definition such as an OVF package. If OVF is used by a node, it means that the node is deployed on a virtual system or a component (OVF's "product") running in a virtual system, as defined in the OVF package.

8.1.1 Use Case One: OVF Package for Single Virtual System

Consider a web application deployment requires an application server. The application server is implemented by a virtual system installing Linux OS and App Container. The virtual system is defined in an OVF descriptor file and an image installing SUSE Linux and Tomcat is included in the OVF package.

Accordingly, developers define two `NodeType`*s* in TOSCA for the application server and the Web application respectively. The `NodeTypeImplementation` for the `NodeType` of the application server provides the deployment artifact "appServer.ova" to materialize instance of the particular `NodeTemplate` referring the `NodeType` of the application server. The `NodeTypeImplementation` also provides the implementation artifact "serverMgt.war" to implement the interface operations of the `NodeType`. The service template topology is shown in Figure 15.

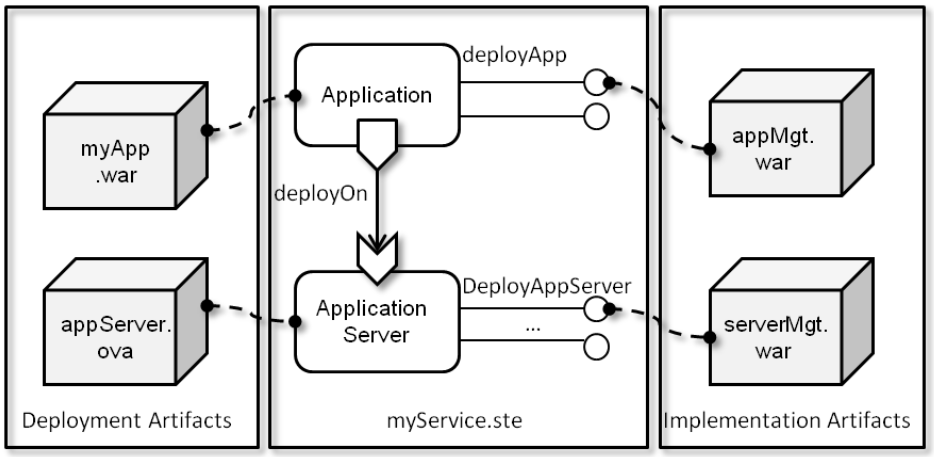

FIGURE 15 - SAMPLE SERVICE TOPOLOGY FOR OVF USE CASE 1

The `NodeType` for the application server is defined within the `Definitions` element of a *Definitions* document identified as "MyNodeTypes" within the target namespace "http://www.example.com/SampleNodeTypes". Thus, by importing the corresponding namespace into another *Definitions* document, the `NodeType` is available for use in the other document.

```
<Definitions id="MyNodeTypes" name="My Node Types"
             targetNamespace="http://www.example.com/SampleNodeTypes">

  <NodeType name="ApplicationServer"
            targetNamespace="http://www.example.com/SampleNodeTypes">
    <Interfaces>
       <Interface name="MyAppServerInterface">
          <Operation name="DeployAppServer">
       </Interface>
    </Interfaces>
  </NodeType>
</Definitions>
```

The following steps should be followed in order to take an OVF package as the deployment artifact for the `NodeTemplate` of the `NodeType` "ApplicationServer".

8.1.1.1 Step One. Defining the ArtifactType that can be used for describing OVF packages as deployable artifacts.

Like the `NodeType`, the `ArtifactType` is also defined in a *Definitions* document with the `id` attribute value "MyArtifactTypes" within the target namespace "http://www.example.com/SampleArtifactTypes". Thus, by importing the corresponding namespace into another *Definitions* document, the `ArtifactType` is available for use in the other document.

```
<Definitions id="MyArtifactTypes" name="My Artifact Types"
  targetNamespace="http://www.example.com/SampleArtifactTypes"
  xmlns:mnt="http://www.example.com/BaseArtifactTypes"
  xmlns:map="http://www.example.com/SampleArtifactProperties>

  <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
          namespace="http://www.example.com/ToscaBaseTypes"/>

  <Import importType="http://www.w3.org/2001/XMLSchema"
          namespace="http://www.example.com/SampleArtifactProperties"/>

  <ArtifactType name="OVFPackage">
    <DerivedFrom typeRef="ba:VMPackage"/>
    <PropertiesDefinition element="map:OVFPackageProperties"/>
  </ArtifactType>
</Definitions>
```

8.1.1.2 Step Two. Defining the Artifact Template referring to the ArtifactType "OVFPackage".

The "OVFPackage" `ArtifactType` is defined in another *Definitions* document with the `id` of "MyArtifacts" within the target namespace "http://www.example.com/SampleArtifacts". Thus, by importing the corresponding namespace into another *Definitions* document, the `ArtifactTemplate` is available for use in the other document.

```
<Definitions id="MyArtifacts" name="My Artifacts"
  targetNamespace="http://www.example.com/SampleArtifacts"
  xmlns:sat="http://www.example.com/SampleArtifactTypes">

  <Import namespace="http://www.example.com/SampleArtifactTypes"
          location="http://www.example.com/
                              Types/MyArtifactTypes.tosca"
          importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>
```



```
  <ArtifactTemplate id="MyOVFInstallable"
                    name="My OVF Installable"
                    Type="sat:OVFPackage">
    <ArtifactReferences>
       <ArtifactReference reference="files/appServer.ova"/>
    </ArtifactReferences>
  <ArtifactTemplate>

</Definitions>
```

8.1.1.3 Step Three. Defining the NodeTypeImplementation "MyImpls".

The following code block defines an implementation of the `NodeType` "ApplicationServer".

```
<Definitions id="MyImpls" name="My Implementations"
  targetNamespace="http://www.example.com/SampleImplementations"
  xmlns:snt="http://www.example.com/SampleNodeTypes"
  xmlns:sat="http://www.example.com/SampleArtifactTypes"
  xmlns:sa="http://www.example.com/SampleArtifacts"
  xmlns:ba="http://www.example.com/BaseArtifactTypes">

  <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
          namespace="http://www.example.com/BaseArtifactTypes"/>

  <Import namespace="http://www.example.com/SampleArtifactTypes"
          location="http://www.example.com/
                          Types/MyArtifactTypes.tosca"
          importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>

  <Import namespace="http://www.example.com/SampleArtifacts"
          location="http://www.example.com/
                          Artifacts/MyArtifacts.tosca"
          importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>

  <Import namespace="http://www.example.com/SampleNodeTypes"
          location="http://www.example.com/
                          Types/MyNodeTypes.tosca"
          importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>

  <NodeTypeImplementation name="MyAppServerImpls"
                          nodeType="snt:ApplicationServer">
    <ImplementationArtifacts>
      <ImplementationArtifact name="MyAppServerManagement"
                              interfaceName="MyAppServerInterface"
                              artifactType="ba:WARFile">
          files/serverMgt.war
      </ImplementationArtifact>
    </ImplementationArtifacts>

    <DeploymentArtifacts>
      <DeploymentArtifact name="MyOVFPackage"
                          artifactType="sat:OVFPackage"
                          artifactRef="sa:MyOVFInstallable">
      </DeploymentArtifact>
    </DeploymentArtifacts>

  </NodeTypeImplementation>

</Definitions>
```

8.1.2 Use Case Two. OVF Package for Multiple Virtual Systems

OVF package may contain multiple VM images which are *Deployment Artifacts* for multiple *Node Type Implementations* or *Node Templates*, hints should be given so that TOSCA container can map the right VM image with a node type implementation or template.

Consider a web application deployment requires an Application server and a DB server. So, two VM images are needed, where one image installing Linux OS and Apache and the other one installing Linux OS and MySQL. Therefore, in an OVF descriptor file, these two images are described by two `VirtualSystem` elements and included in one `VirtualSystemCollection` element as follows:

```
<VirtualSystemCollection ovf:id="multi-tier-app">
   <Info>A collection of virtual machines</Info>
   <Name>Multi-tiered Appliance</Name>
   <SomeSection>
       <!-- Additional section content -->
   </SomeSection>
   <VirtualSystem ovf:id="appServer">
       <Info>A virtual machine installing Suse Linux and Tomcat</Info>
       <Name>Application Server</Name>
       <!-- Additional sections -->
  </VirtualSystem>
  <VirtualSystem ovf:id="dbServer">
       <Info>A virtual machine installing Suse Linux and MySQL</Info>
       <Name>DB Server</Name>
       <!-- Additional sections -->
  </VirtualSystem>
</VirtualSystemCollection>
```

Accordingly, developers define two `NodeType`*s* in TOSCA for the "Application Server" and the "Database Server" respectively. The `NodeTypeImplementations` for these two `NodeType`*s* refer the same OVF package as the deployment artifact. The service topology is shown in Figure 16.

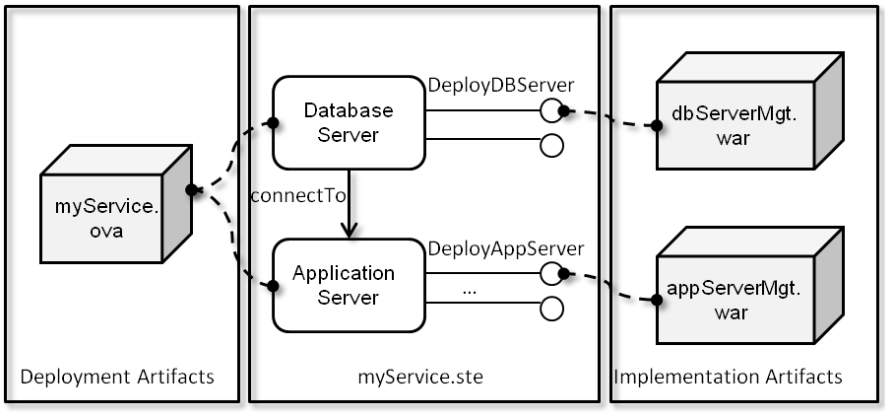

FIGURE 16 - SAMPLE SERVICE TOPOLOGY FOR OVF USE CASE 2

The `NodeTypes` for the "Application Server" and "Database Server" are defined in a *Definitions* document "MyNodeTypes" within the target namespace "http://www.example.com/SampleNodeTypes". Thus, by importing the corresponding

namespace into another *Definitions* document, the `NodeTypes` are available for use in the other document.

```
<Definitions id="MyNodeTypes" name="My Node Types"
  targetNamespace="http://www.example.com/SampleNodeTypes">

  <NodeType name="ApplicationServer"
            targetNamespace="http://www.example.com/SampleNodeTypes">
    <Interfaces>
       <Interface name="MyAppServerInterface">
          <Operation name="DeployAppServer">
       </Interface>
    </Interfaces>
  </NodeType>

  <NodeType name="DBServer"
            targetNamespace="http://www.example.com/SampleNodeTypes">
    <Interfaces>
       <Interface name="MyDBServerInterface">
          <Operation name="DeployDBServer">
          <Operation name="AcquireNetworkAddress">
       </Interface>
    </Interfaces>
  </NodeType>

</Definitions>
```

Since the deployment artifacts to implement these two `NodeTypes` are packaged in a single OVF package, the instances for these two nodes can be instantiated by importing the OVF package only once. There may be interaction between these two nodes. For example, the application server may acquire the network address of the database server in order to connect to it. The TOSCA container should know which VM is for the application server and which one is for the database server after successfully importing the OVF package.

The `ArtifactTemplates` are defined in a *Definitions* document "MyArtifacts" within the target namespace "http://www.example.com/SampleArtifacts". Thus, by importing the corresponding namespace into another *Definitions* document, the `ArtifactTemplates` are available for use in the other document.

```
<Definitions id="MyArtifacts" name="My Artifacts"
  targetNamespace="http://www.example.com/SampleArtifacts"
  xmlns:sat="http://www.example.com/SampleArtifactTypes">

  <Import namespace="http://www.example.com/SampleArtifactTypes"
          location="http://www.example.com/
                          Types/MyArtifactTypes.tosca"
          importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>

  <ArtifactTemplate id="AppServerInstallable"
                    name="Application Server Installable"
                    Type="sat:OVFPackage">
     <ArtifactReferences>
         <ArtifactReference reference="files/myService.ova"/>
            <Include pattern="appserver.img"/>
     </ArtifactReferences>
  <ArtifactTemplate>

  <ArtifactTemplate id="DBServerInstallable"
                    name="Database Server Installable"
                    Type="sat:OVFPackage">
```



```
      <ArtifactReferences>
          <ArtifactReference reference="files/ myService.ova"/>
              <Include pattern="dbserver.img"/>
      </ArtifactReferences>
  <ArtifactTemplate>
</Definitions>
```

By specifying the VM image in the `ArtifactTemplates` and referring to them in the `DeploymentArtifacts` of `NodeTypeImplementations`, the TOSCA container can be aware of the role of each VM. The `NodeTypeImplementations` for `NodeTypes` can be defined as follows:

```
<Definitions id="MyImpls" name="My Implementations"
  targetNamespace="http://www.example.com/SampleImplementations"
  xmlns:snt="http://www.example.com/SampleNodeTypes"
  xmlns:sat="http://www.example.com/SampleArtifactTypes"
  xmlns:sa="http://www.example.com/SampleArtifacts"
  xmlns:ba="http://www.example.com/BaseArtifactTypes">

  <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
          namespace="http://www.example.com/ToscaBaseTypes"/>

  <Import namespace="http://www.example.com/SampleArtifactTypes"
          location="http://www.example.com/Types/MyArtifactTypes.tosca"
          importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>

  <Import namespace="http://www.example.com/SampleArtifacts"
          location="http://www.example.com/Artifacts/MyArtifacts.tosca"
          importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>

  <Import namespace="http://www.example.com/SampleNodeTypes"
          location="http://www.example.com/Types/MyNodeTypes.tosca"
          importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>

  <NodeTypeImplementation name="MyAppServerImpls"
                          nodeType="snt:ApplicationServer">

    <DeploymentArtifacts>
       <DeploymentArtifact name="AppServerDA"
                           artifactType="sat:OVFPackage"
                           artifactRef="sa:AppServerInstallable">
       </DeploymentArtifact>
    </DeploymentArtifacts>

  </NodeTypeImplementation>

  <NodeTypeImplementation name="MyDBServerImpls"
                          nodeType="snt:DBServer">

    <DeploymentArtifacts>
       <DeploymentArtifact name="DBServerDA"
                           artifactType="sat:OVFPackage"
                           artifactRef="sa:DBServerInstallable">
       </DeploymentArtifact>
    </DeploymentArtifacts>

  </NodeTypeImplementation>

</Definitions>
```

Based on the above *Definitions* document, the TOSCA *container* can determine that the VM made from “appserver.img” is the application server and the VM made from “dbserver.img” is the database server. When the operation “AcquireNetworkAddress” defined in the `NodeType`

"DBServer" is performed, the TOSCA container can access the correct IP Address of the VM for the database server.

Appendix A. Acknowledgments

The following individuals have participated in the creation of this document and are gratefully acknowledged by the TOSCA TC:

Contributors by Section:

Section 1: Matt Rutkowski, IBM
Section 2: Adolf Hohl, NetApp; Frank Leymann, IBM
Section 3: Frank Leymann, IBM
Section 4: Frank Leymann, IBM
Section 5: Dietmar Noll, IBM; Frank Leymann, IBM
Section 6: Matt Rutkowski, IBM
Section 7: Doug Davis, IBM; Thomas Spatzier, IBM
Section 8: Paul Zhang, Huawei; Marvin Waschke, CA

Appendix B. Terminology & Acronyms

B.1 Acronyms

Acronym	Phrase
BPMN	Business Process Model and Notation
CIMI	Cloud Infrastructure Management Interface, a DMTF specification.
CRM	Customer Relationship Management
CSAR	Cloud Service Archive, a TOSCA specified format
DMTF	Distributed Management Task Force
EC2	Elastic Compute Cloud, an Amazon Web Service (AWS).
EJB	Enterprise JavaBeans
FTP	File Transfer Protocol
J2EE	Java 2 Platform, Enterprise Edition
JAR	Java ARchive
NTFS	New Technology File System
OVF	Open Virtualization Format
PHP	PHP: Hypertext Preprocessor, a recursive acronym.
QoS	Quality of Service
RPM	RPM Package Manager, a packaging format used on some Linux distributions.
TAR	Unix-style archive file format; originally developed for tape archives, but has been adopted for additional uses.
UID, UUID	Unique Identifier, and Universally Unique Identifier
VM	Virtual Machine

Appendix C. Revision History

Revision	Date	Editor	Changes Made
WD1, Rev. 01	2012-11-01	Matt Rutkowski, IBM	Initial Version, with Table of Contents (TOC).
WD1, Rev. 02	2012-11-04	Frank Leymann, IBM	Addition of initial Chapter 2 content.
WD1, Rev. 03	2012-11-05	Adolf Hohl, NetApp	Addition of remainder of Chapter 2 content.
WD1, Rev. 04	2012-11-09	Matt Rutkowski, IBM	Cleanup and edits for Chapters 1 & 2. Added Terminology and Acronyms Appendix.
WD1, Rev. 05	2012-11-15	Frank Leymann, IBM; Matt Rutkowski, IBM	Frank added explanation of declarative vs. imperative models. Also, added the section sketching the TOSCA environment with corresponding figures. Matt preformed more edits/cleanup.
WD1, Rev. 06	2012-11-19	Matt Rutkowski, IBM	Added “Statement of Purpose” to Introduction (Chapter 1).
WD1, Rev. 07	2012-11-26	Matt Rutkowski, IBM	Authored contents of Section 3.1 plus general editing of grammar and content for document consistency.
WD2, Rev. 01	2012-12-05	Paul Zhang, Huawei; Marv Waschke, CA	Added contents for Section 8.1 “Mapping TOSCA to DMTF OVF”.
WD2, Rev. 02	2012-12-10	Matt Rutkowski, IBM	Addition of Section 7 “What cloud providers should know…” along with edits and comments. Fixed style problems with Section 8.1 merge, fixed code samples, and updated document fields, minor edits.
WD2, Rev. 03	2012-12-11	Matt Rutkowski, IBM	Comprehensive editing of Section 2 contents. Minor edits to other sections.

WD2, Rev. 04	2012-12-12	Frank Leymann, IBM; Matt Rutkowski, IBM	Merged in Section 6 “What Artifact developers should know…”. Fixed all figure captions and added “List of Figures” table to top of document. Minor edits to Section 6.
WD3, Rev. 01	2012-12-13	Matt Rutkowski, IBM	Created “clean” version based upon WD2, Rev. 04 as WD3, Rev. 01 as approved by the TOSCA. TC on 2012-12-13.
WD3, Rev. 02; Rev. 03	2013-01-09; 2013-01-10	Matt Rutkowski, IBM; Frank Leymann, IBM; Doug Davis, IBM	Merged in additional initial content for sections 3 (Matt), 4 (Doug) and complete section 5 content from Frank. Added non-normative references for TOSCA spec. and referenced open source projects. Added new acronyms/definitions.
WD4, Rev 02	2013-01-16	Matt Rutkowski, IBM; Frank Leymann, IBM, Doug Davis, IBM	Matt: Complete top-down edit, all styles, keywords, tables, figures, reference all fixed and made consistent. Reorganization of chapters to discuss all “abstract” concepts before “concrete” examples. Reauthored text to fit new section “flow”. Inclusion of Chapter 4 Artifact Developers” from Frank Leymann and review/edit of its contents. Merged additional content from Doug Davis for migrating “multi-VM” applications in Chapter 7.
WD5, Rev1	2013-01-17	Matt Rutkowski, IBM	Clean version reviewed and approved by TC during TOSCA TC meeting 2013-01-17
WD5, Rev2e	2013-01-22	Matt Rutkowski, IBM	Rewrite of section 6 to split DB tier from Web App tier and adjust tables/figures/code examples to match.
WD5, Rev2f	2013-01-23	Matt Rutkowski, IBM	Merged updates for Ch.4 from Doug Davis and Section 3.4.3 from Frank Leymann. Addressed comments from Frank for Section 6 where still applicable.
WD5, Rev3	2013-01-24	Matt Rutkowski, IBM	Addressed all legacy comments/outstanding work items based upon resolutions/plans discussed on

			editor's call during top-down review. Also addressed updates to section 7 from Doug Davis/Thomas Spatzier and comments from Frank Leymann on section 6 contents. Removed Section 7.2 at the request of Doug Davis. Matt's Edits/Revisions and remaining content for section 6. Matt's
WD6, Rev1	2013-01-24	Matt Rutkowski, IBM	TC approved the creation of WD6, Rev1 during the OASIS TOSCA TC call on 2013-01-24 based upon WD5, Rev3b.
WD6, Rev2	2013-01-30; 2013-01-31	Matt Rutkowski, IBM	Addressed comments and corrections from Gerd Breiter, Doug Davis and Frank Leymann. Also, some other corrections/edits based upon another top-down re-read of the final document.

