Concept for executing management operations on components of application instances

Yannic Sowoidnich

Course of Study: Softwaretechnik

Examiner: Prof. Dr. Dr. h. c. Frank Leymann
Supervisor: M.Sc. Karoline Saatkamp

Commenced: April 1, 2019
Completed: October 1, 2019
Abstract

A large field of technologies exist for orchestrating cloud applications. Many of them focus on automated deployment techniques, rather than continuous management of application instances. Executing operations for deploying applications is different from executing management operations, due to their dependencies to the application state. Proper state management is important to guarantee valid execution of management operations.

Cloud providers such as Amazon have embedded functions for managing cloud applications, but they come with major drawbacks. They increase vendor-dependency and they do not support multi-cloud deployments.

Technologies like Chef, Puppet or Terraform work with declarative process models, which cannot be used for non-state-changing operations and they mostly only allow simple operations. It is impossible to execute more customized fine grained operations with those technologies. Also, most of these management tools only support executing operations on the whole application, not on specific components of the application.

The objective of this thesis is to find a way for executing management operations on running application instances by combining the information of the deployment model with the instance model of the application. The conceptual approach proposed in this thesis will consider and solve above addressed issues, as well as ensuring proper state management of application instances.

The practical feasibility of this concept is validated by a prototypical implementation based on the TOSCA standard and the OpenTOSCA ecosystem.
Contents

1 Introduction 17

2 Fundamentals 21
   2.1 Web service orchestration ........................................ 21
   2.2 TOSCA .................................................................... 22
   2.3 OpenTOSCA ................................................................. 24
   2.4 Declarative and imperative process models ....................... 28

3 State of the art 31
   3.1 Related work ............................................................... 32

4 Conceptual approach 35
   4.1 Discussion on application states .................................... 35
   4.2 Classification of management operations ......................... 39
   4.3 Executing operations .................................................... 41

5 Implementation of a prototype 43
   5.1 Additions and Changes to the TopologyModeler ............... 44
   5.2 Additions and Changes to the OpenTOSCA UI ................ 47
   5.3 Additions and Changes to the OpenTOSCA Container ........ 49

6 Conclusion and Outlook 53

Bibliography 57
List of Figures

2.1 Concept of service orchestration ........................................ 21
2.2 Concept of service choreography .................................... 22
2.3 Structure of a Service Template [TS13] .............................. 23
2.4 XML structure of an interface with a nested operation .......... 24
2.5 Overview of the OpenTOSCA Ecosystem (from [BKK+16]), slightly changed) . . 25
2.6 Model of the winery architecture [EWPD] .......................... 26
2.7 The MyTinyToDo application, modeled in the TopologyModeler ........ 27
2.8 TOSCA Container Architecture Overview from [OTC] .............. 28
3.1 Classification of orchestration techniques .......................... 33
4.1 Topology of the extended MyTinyToDo Application ............... 36
4.2 General approach to enrich operations with information for the induced state . . 38
5.1 Topology of the MyTinyToDo Application .......................... 44
5.2 Folder structure and meta informational files for Node Package Manager (NPM) library TopologyRenderer .................. 45
5.3 Detailed view of ServiceTemplate instance with integrated TopologyRenderer . . 47
5.4 Management operations of a DockerEngine instance ............ 48
5.5 NodeInstanceState is set to DELETED after successful execution 49
5.6 New OpenTOSCA Container API endpoints /topology and /managementoperation . . 50
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Topology and Orchestration Specification for Cloud Applications (TOSCA) lifecycle states from [TOSCA-Simple-Profile-YAML-v1.2]</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Pros and Cons of the <em>During Modeling Time (DMT)</em> and <em>During Runtime (DRT)</em> method</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>Characteristics of management operations that influence their handling</td>
<td>39</td>
</tr>
</tbody>
</table>
Listings

5.1 Excerpt from winery.component.ts ................................. 45
## List of Algorithms

5.1 Pseudo algorithm of `performManagementOperation()` in `NodeTemplateInstanceController.java` .......................... 51
Acronyms

API  Application Programming Interface. 27
BPMN  Business Process Model and Notation. 25
CSAR  Cloud Service Archive. 24
DMT  During Modeling Time. 9
DRT  During Runtime. 9
DTO  Data Transfer Objects. 49
GUI  Graphical User Interface. 25
IAAS  Architecture of Application Systems. 24
IPVS  Institute for Parallel and Distributed Systems. 24
JSON  JavaScript Object Notation. 44
KPI  Key Performance Indicators. 31
NPM  Node Package Manager. 7
OASIS  Organization for the Advancement of Structured Information Standards. 22
REST  Representational State Transfer. 25
SLA  Service Level Agreements. 31
SOAP  Simple Object Access Protocol. 27
TOSCA  Topology and Orchestration Specification for Cloud Applications. 9
UI  User Interface. 25
VM  Virtual Machine. 18
XaaS  Anything as a Service. 17
1 Introduction

Cloud computing solves many problems of the traditional IT infrastructure with its costly in-house servers and risky investments in dedicated hardware. Deploying applications in the cloud comes with great benefits. With the correct architecture (e.g. microservices [DGL+17]) they become very scalable, cheaper and easier to maintain, but cloud applications come with their own challenges [ZCB10] [JM12]. An often discussed problem is the vendor lock-in [OST14] [SHI+13]. Most applications are designed and explicitly programmed to fit into a specific vendor environment. Such applications are often not interoperable or portable and changing from one vendor to another becomes very pricy. Changes in the vendors infrastructure or adjustments of the pricing model of a vendor can cause severe problems for a customer due to the heavy dependencies to the environment [SHI+13].

There exist many technologies and third partie software using declarative deployment models to prevent vendor lock-in, popular examples are Puppet, Kubernetes or Chef [WBF+19]. Those technologies provide mechanisms and techniques for automated deployment and therefore share the same purpose, but they are quite different in supported features and in implementation details [WBF+19]. Not all of them support the most important deployment features such as single-, hybrid-, and multi-cloud deployments as well as Anything as a Service (XaaS) or can be extended and customized for further services [WBF+19]. Automated deployment technologies working with declarative deployment models to prevent vendor lock-in have been explored and discussed by many [BBK+14] [EBF+17], since they do not require much technical knowledge of the users. This happens by specifying how they final application should look like - technical details of how this result is achieved is left to the tool that processes the declarative deployment model.

Because most of these technologies only support declarative models and are mainly used for deploying cloud applications rather than managing them throughout their entire lifecycle, this work uses the TOSCA [TS13]. Contrary to the mentioned technologies the TOSCA specification is a standard and technologies using it do not need to introduce their own terminology or definitions. Also, the standard not only provides vendor independency but also improved flexibility, portability and interoperability for cloud applications. It supports declarative process models as well as imperative process models. Unlike declarative models, imperative models specify each step on a very fine grained level and require much technical knowledge, but have proven themselves to be much more customizable and more applicable for complex applications [EBF+17]. The TOSCA standard offers various concepts to enable flexible and easy management of distributed and compound applications [BBLS12]. TOSCA describes the structure of applications by deployment models that consist of the application’s components and their relationships. These deployment models also encapsulate respective management operations. Such operations can be run on the whole web application itself, or just a component of the application. A management operation can be anything and each operation comes with their own challenges. For example, an operation that is stopping a component is cumbersome because the internal state which often holds all business information is lost if no suitable state preserving mechanic is running in the background [HBKL19]. Another
example could be a `connectTo` operation that is connecting an application to a database. This operation would require some parameters (database credentials, database name) to work properly. Even large application deployment tests consist of small operations that together form the test operation [WBKL18]. To deploy or manage an application, it often is required to run deployment scripts, perform some manual tasks and (automated) operations must be executed, etc. [TS13]. This means for large scale applications, that there are a lot of management operations to be performed and the process models grow in complexity and size.

Automating and testing deployments is a well explored research field, however executing operations on running instances of an application still needs to be examined [HBS+19]. Performing operations on instances can behave differently and needs to be handled differently than operations during deployment for various reasons. Component state as well as the global application state play an important role for management operations. Most of the time deployment models are declarative, but some operations are not state changing and thus cannot be expressed with a declarative process model, therefore it is required to have an imperative approach. For managing applications it is often required to execute an operation on just a single component and not on the whole application itself. Executing these operations on application instances can lead to severe impacts for the whole system, or results in other components to be required to perform some additional management tasks. For example, when an application runs on a Virtual Machine (VM) and this VM gets stopped, the application is also stopped, since it cannot run anymore. Another challenge is to coordinate the simultaneous execution of management operations on many component instances. If not all application component instances are in the same state, it is very hard to guarantee the proper execution of each operation on each instance. If an operation impacts the global application state, some other queued operations possibly can't be performed anymore. All above mentioned reasons make managing an application different from deploying it.

There are some services and tools available, either by third party providers or the cloud providers themselves, to achieve successful orchestration of a full application. But the drawbacks of these services are clear: they often only work for well-defined orchestration tasks and they are completely encapsulated from the software that is deployed to the service [TBB+15]. They also mostly only work for declarative process models, rather than on imperative process models. There are several approaches tackling parts of these issues. Toffetti et al. presented an architecture for self-managing microservices [TBB+15], which tries to solve the aforementioned problems by choosing a novel software architecture that tightly couples the services with their corresponding management operations. While this approach may be very suitable for new projects that have to be implemented yet, it is not fitting for existing services. Above mentioned technologies like Chef, Kubernetes or Puppet often come with functionality to support runtime management of an application. But most of the time they only provide management operations like scaling components or changing their configurations [HBS+19]. Unfortunately, none of these approaches considers the current state of the application or its components, therefore potential errors can impact the functionality of the whole system.

All of the presented technologies for performing operations on application components share that they do not follow a unified and standardized specification. TOSCA solves this problem and also introduces management plans, which are chained management operations that work on the whole application. Usually, for single ad hoc operations on a component instances there exist no plans. Those plans could be generated, similar to the approach of generating deployment plans [BBK+14] or management plans [HBS+19]. The issue with generating plans for single operations is, that a
huge amount of management plans need to be generated automatically. Naturally, this will become very complex and confusing to handle and error-prone. Therefore it is important to have a concept to execute management operations on application instances and on its components. This concept should also consider the aforementioned problem of state handling of the application instances. The objective of this work is to develop such a concept to enable proper state handling and execution of management operations on single application components. State handling should consider the state of each component and the resulting global application state.

The concept for state handling is developed by classifying distinct classes of management operations and by examining states that instances of application components can reach. Also, dependencies amongst operations and the components state will be explored. Unlike deployment plans and management operations used for deployment, tasks that are executed on instances use the instance model and not the deployment model. One of the key factors of this approach is that it doesn’t only use the deployment model which was predefined at modeling time of the service but merges the information from the instance model with runtime data of the actual running instances of the application. The knowledge of the current runtime state of the application allows to execute management operations more efficiently, since the runtime state can differ significantly from what is specified in the deployment model. By merging the deployment model with the instance model, the objective of executing management operations on instances will be achieved. The proposed approach in this thesis is verified by a prototypical implementation, fully integrated into the OpenTOSCA ecosystem (see chapter 2.3).
2 Fundamentals

To fully understand the contributions of this work, the reader is required to have basic knowledge of web service orchestration in general (introduced in section 2.1) and more specifically about management operations. The concept developed in this work is tightly coupled to the TOSCA standard, therefore chapter 2.2 introduces the most relevant aspects. A prototypical implementation of the concept is done in the OpenTOSCA environment, a cloud runtime environment developed by the University of Stuttgart. Therefore, section 2.3 introduces an overview of this runtime together with the building parts that are important for managing application instances. Lastly, the motivation of this approach is also to not only allow management operations on declarative process models, but also to provide some imperative modeling. Consequentially, chapter 2.4 explains the differences between imperative and declarative approaches.

2.1 Web service orchestration

Loose coupling, microservices and service-oriented architecture are important building blocks in modern software development. Being able to invoke interoperable tasks and functions and to have them loosely coupled to many different application domains enables software developers to reuse their already written components. This reduces costly and time-consuming development of functionally redundant code and helps maintaining the software long-term. An important aspect of software built with this service-oriented approach is the management of its composite services. Either service orchestration or service choreography is used for that purpose [Pel03].

![Figure 2.1: Concept of service orchestration](image)

Service orchestration follows a fixed logic, where one centralized orchestrating component manages and coordinates the interactions of several other microservices. The orchestrator provides an interface for invoking services and returns their bundled and combined output. This technique
is used by developers to create composite applications out of different services and to support automation of business processes. Services are not programmed to interact or communicate with each other, aside from the orchestration service. Therefore, to produce the correct output, execution logic and messages are handled by the centralized orchestrating component [DD04]. In contrast to this model, there is service choreography which is also used to create composite applications from multiple services.

![Figure 2.2: Concept of service choreography](image)

Service choreography follows a concept where no centralized service coordinates the other services, but it follows a decentralized approach. All the participating services know about their interactions and the business logic that’s executed. The communication and interaction between services is defined by a set of rules and agreements between endpoints. So, the difference between service orchestration and service choreography lies within where the operating logic for interactions between application components resides [DD04]. The prototypical implementation of this thesis is done in the OpenTOSCA environment which follows the service orchestration approach. The main goal is to find an efficient method to execute management operations on live instances of some application. Therefore, it’s crucial to understand current approaches and methodologies that achieve the same goal. All of the common approaches fall in one of two categories. Either the operations are provided directly by the cloud vendor or they are provided by some third-party application. The problem with many of those solutions is, that they are often not able to execute any operation during runtime, but rather they need to have specific operations specified during modeling time, that can be executed later. The concrete approaches will get discussed in the next chapter.

### 2.2 TOSCA

The *Organization for the Advancement of Structured Information Standards (OASIS)* specified a standard called *TOSCA* in the year 2013. It enhances the portability and operational management of cloud applications [TS13]. The standard can be used to describe and define topologies of cloud applications with either the markup languages *YAML* (supported since 2016) or *XML* (supported since 2013). These topologies consist of the components of the modeled application, their relations...
2.2 TOSCA

Figure 2.3: Structure of a Service Template [TS13]

amongst them, as well as all the resources and information that is required to instantiate and orchestrate the software throughout its whole lifecycle. This section will not cover the whole TOSCA specification and explain the model in detail, but it will cover parts that are important to understand the concept explained in this paper.

An application described by the TOSCA metamodel would be modeled as a ServiceTemplate and represents the highest level of abstraction in the model. Since TOSCA is a descriptive language to define web applications, it helps to remember the nested nature of XML and YAML, so all other elements reside within the ServiceTemplate. Figure 2.3 shows a ServiceTemplate that has a TopologyTemplate inside. This defines the structure of a service. Plans reside on the same level of abstraction and are used to manage the services lifecycle with workflows. These process models are modeled with existing languages like BPEL or BPMN, rather than the standard having its own language. This is to keep the interoperability and portability high and makes the TOSCA standard more accessible.

A set of RelationshipTemplates and NodeTemplates is nested into the TopologyTemplate and together those templates form the Topology Model of a service, displayed as a graph. The nodes of this graph are called NodeTemplates and its edges are the RelationshipTemplates. Their properties and their interactions are defined by either their NodeTypes or their RelationshipTypes, meta information is defined by the templates themselves, such as usage constraints. The most important element in this specification for this work are the Operations of a NodeType. They are defined in the interfaces of a NodeType and can manipulate the component by providing some executable functionality to the node. The actual executables for these Operations are either provided as ImplementationArtifacts or DeploymentArtifacts. The former is used to implement interface operations of the NodeType (for running instances), such as start or stop operations. Whereas DeploymentArtifacts are the
executables needed for the instantiation of nodes, such as a Docker Container or a MySQLDB. A schematic representation of an Interface can be seen in figure 2.4. It defines three Operations called install, configure and uninstall. Next to its name property, an Operation can have some optional properties such as InputParameters and OutputParameters, that are used to configure the executable.

For a TOSCA runtime to be able to interpret a specified ServiceTemplate and its corresponding topology much information is required. To ensure the management and proper execution of a cloud application, not only the ServiceTemplate and the ServiceTopology are required, but all of its DeploymentArtifacts and ImplementationArtifacts have to be available to the underlying system [PCWTCSA]. For this task, a common data format is required. It needs to bundle all the necessary information and provide it to the runtime. Therefore, the TOSCA specification defined the Cloud Service Archive (CSAR), an archive format for transmitting this data [Sta13]. Since a CSAR is an archive file, it is zipped and usually compressed, allowing for quick data transmission, ensuring a fast deployment of the application [Sta13].

2.3 OpenTOSCA

The OpenTOSCA ecosystem is an open-source end-to-end toolchain for the deployment and management of TOSCA-defined applications. It was developed at the University of Stuttgart by the departments for Architecture of Application Systems (IAAS) and the Institute for Parallel and Distributed Systems (IPVS) and is supported by the german government. The ecosystem follows and implements the TOSCA standard and is an open-source tool with the purpose to model, orchestrate and supervise cloud applications [OTCOE]. Its greatest benefit is vendor-independency and its ability to support both, a declarative and an imperative approach to provision instances of the modeled applications [BKK+16]. This allows the users to choose and adjust the management of their applications by their capabilities and own needs, since the declarative style provides a low entry level and doesn’t require a huge knowledge about the processes behind the scenes but comes at the cost of losing some flexibility and control over the deployment flow. The imperative approach works vice versa, with high flexibility but requires more knowledge of the deployment flow [TOT].

The graphic depicted in figure 2.5 shows the three major building blocks of OpenTOSCA. The TOSCA modelling tool Winery, where users can model the components and relate them with management operations. The OpenTOSCA Container, which is the runtime that is used to instantiate
and manage the instances of a web-application. A self-service portal, called OpenTOSCA UI which allows users to use the functions provided by the runtime. All these three components will be explained in detail in the following subsections.

2.3.1 Winery

Graphical modeling of TOSCA topologies in the OpenTOSCA ecosystem are enabled by a web-based application called Winery. It has three basic Graphical User Interface (GUI) components to it, that are all required to define a complete model of a web application and a repository with a Representational State Transfer (REST) interface (see figure 2.6). The GUI of the Winery has been split into distinct components to increase the usability for users and make it more accessible for non-technical users [KBBL13]. This is achieved by splitting the modeling elements that are provided by the TOSCA standard into different categories. All elements that are related to visual topology modeling (like NodeTemplates or RelationshipTemplates) are modeled in the TopologyModeler component. All other elements are used to define metadata for the modeled (visual) elements such as configurations or types, are managed within the Type, Template and Artifact Management component (former called ElementManager) [KBBL13]. The third GUI component is the BPMN4TOSCA Plan Modeler, which offers a service to model some TOSCA plans with the Business Process Model and Notation (BPMN) notation. It is important to note, that not all BPMN elements and structures are supported, but rather the ones that are needed for TOSCA [EWPD]. The repository is used to manage and store the created TOSCA models. It also offers functionality to import or export existing CSAR files, to allow community-based work [KBBL13].

The TopologyModeler will be explained in a bit more detail in this section, since it plays an important role for the implemented prototype. It has been modified, so that its rendering functionality got encapsulated and could be exported to other projects, namely the OpenTOSCA User Interface (UI), which is described in a later chapter. An application topology gets created with the TopologyModeler
by dragging the required node type from a dropdown palette to the editing area, where it will become an actual node in the topology graph [KBBL13]. The node itself then can be annotated and populated with additional information, like the node’s properties, its requirements or its capabilities and much more information that is needed to efficiently manage and run the node instance later. By clicking on a node, some basic information is displayed in a sidebar to the node. Its name and the minimum and maximum instances can be modified there.

Also, additional elements can be created by clicking on the node and choosing from a dropdown menu. Those RelationshipTemplates become the edges in the topology graph and define the relation between the node components. Figure 2.7 shows the MyTinyToDo application, with all the mentioned elements of the TopologyModeler. On the top side of the editing area resides a navigation, that allows to enable or disable the visibility of certain information in the model (properties, deployment artifacts, etc.). The TopologyModeler also supports some layouting algorithms, to improve the understandability of the displayed graph and to help users with the modeling.

### 2.3.2 OpenTOSCA Container

The OpenTOSCA Container is the actual runtime of the OpenTOSCA project and provides the core functionalities of the ecosystem. Those key functions are to run plans and management operations, manage the state of the application or provide utility functions to other components, such as validating XML data [BBH+13]. These tasks revolve around importing CSARS and interpreting its contents in different stages of the web applications lifecycle. Therefore, a very modularized architectural pattern has been chosen to implement the OpenTOSCA Container, so that there is a component for each dedicated task to provide extended flexibility and extensibility. Java and the OSGi framework have been used to match those requirements. Figure 2.8 shows an architectural overview of the container with its main components. The Engine, consisting of the PlanEngine and the ImplementationArtifactEngine, is responsible for executing and processing management plans.
2.3 OpenTOSCA

Figure 2.7: The MyTinyToDo application, modeled in the TopologyModeler

and invoking and binding services that are required for the instantiation for the web application [BBH+13]. One of the special features of the OpenTOSCA project is to be able to provision instances in either a declarative or an imperative way. This is partly possible because of the PlanBuilder. It can either execute existing build plans from a CSAR or is able to generate them from a given application topology [OTC]. The processed build plans can install, deploy or provision the necessary parts of a TopologyTemplate and will be injected back into the CSAR for further processing.

The most important components to this work are the the ContainerAPI and the ManagementBus. The bus allows the invocation of management operations and acts as mediator between many different components of the OpenTOSCA Container [OTCSI]. It supports different Application Programming Interface (API)’s such as REST, OSGi and Simple Object Access Protocol (SOAP). The ContainerAPI exposes many utility functions to the ContainerUI, such as getting or storing data of models or instances.

2.3.3 OpenTOSCA UI

The OpenTOSCA runtime environment formerly implemented a self-service portal called Vinothek, a simple graphical interface that allowed users to interact with the functionality in a convenient way [OTCSI]. It is the predecessor of the OpenTOSCA UI, a much larger and more complex graphical user interface, created with the modern Angular framework. The UI enables easy installing,
configuring and provisioning of applications. It also provides functionality to see the instances of applications and interact with them, though this functionality is not fully extended yet. One of the contributions of this work is to enable more interaction with the application instances by performing management operations on their respective components, since the UI allows only performing management tasks for the whole application via management plans so far. Since the user interface is based on a classical web-based client-server architecture, no additional software is required on client side and makes it easy to use out of the box [OTCSI]. The communication happens via the containers RESTful API mentioned in section 2.3.2.

### 2.4 Declarative and imperative process models

Most technologies that provide automated deployment functionality use a declarative approach, where the specification describes the desired state of the system and the application takes all necessary actions to get the system into the required state [HAW11]. The biggest advantages of declarative management are that the outcome of the transition, the final state [HAW11], is well defined and known, since it was specified, and that the user has not to define the underlying logic to produce the desired outcome, which is a time-consuming, error-prone task [BBK+14]. Another advantage of declarative approaches is their intrinsic extensibility. They work by defining constraints and a desired outcome, rather than concrete steps to take to produce the result. Therefore, declarative approaches are quite flexible since with an imperative approach the whole execution logic for a new functionality had to be defined [BBK+14].
This is because imperative process models work by specifying the concrete steps on how to execute an operation, rather than specifying the result. The commonly used languages BPEL and BPMN are used to model imperative process models. Breitenbücher et al. [BBK+14] state that imperative management of cloud resources is required if the applications get too complex or if the application developers are required to specify a certain set of steps for the application to take. However, Fahland et al. don’t see evidence that imperative approaches deliver larger and more complex process models [FLM+09]. They conclude that some applications are more imperative (or declarative) to a higher or lesser degree than others [FLM+09].

Because imperative and declarative approaches each have their own strengths and weaknesses, the OpenTOSCA environment where the prototype is built in, supports both [BBK+14]. This enables users to define what they want without having to care about technical detail, but also gives them access to it if its needed. By extending the OpenTOSCA runtime environment by the approach proposed in this work for managing application components, with the environments property to support imperative and declarative provisioning, the prototype will become a novel and possibly potential tool for future research.
3 State of the art

Over recent years, many approaches have been developed to orchestrate web-services to ensure maximum flexibility on hosting applications in different cloud environments. The most common major differences lie in their automated deployment technology and in their degree of being vendor-independent and in their ease-of-use. Therefore, section 3.1 will introduce different orchestration categories, which were derived from observations of deployment automation technologies made by Wurster et al. [WBF+19] and common architecture of cloud applications, that are introduced in this section. These categories will be used to sort in existing approaches for the problems that this work tries to solve and to show how the concept proposed in this work is different.

Wurster et al. did a survey on the most commonly used technologies for automatic deployment for web applications and came up with three classes: General-Purpose (GP), Provider-Specific (ProvS) and Platform-Specific (PlatS) [WBF+19]. Those groups have been formed upon the ability of the considered technologies to satisfy and provide a set of features and mechanisms that have been defined by the research team. The software should be able to provide single-, hybrid-, and multi-cloud deployments, offer support for multiple cloud offerings (anything as a service; XaaS) and it should be able to specify the deployment on a very granular level [WBF+19]. Technologies that could fulfill all deployment features and mechanisms have been placed in the General-Purpose category. The Provider-Specific technologies support most of the features and mechanisms but are only capable of doing single cloud deployments (in the environment of their respective provider), hence the name. PlatS, the last of these categories is restricted by the cloud delivery model and they are required to have specific platform bundles for realizing and instantiating components [WBF+19].

Orchestrating cloud applications is not only relying on automated deployment techniques, they need to be managed continuously throughout their entire lifecycle. To fit not only automated deployment technology, but rather whole orchestration systems like OpenTOSCA, additional quality metrics must be added to the existing categories introduced by Wurster et al. and those metrics have to be observed by the monitoring ecosystem. Monitoring the state of a web application is important to adjust the provided resources to the incoming load dynamically. Transient errors on components can be detected and faced by restarting or recreating the component to improve resiliency against system failures [TBB+15]. Many current cloud orchestration products offer automated scaling and provisioning based on measurable Key Performance Indicators (KPI) to ensure that the Service Level Agreements (SLA) between the customer and the cloud vendor are met, but this technology is rarely used to provide fully automated health management for the cloud application and often requires human intervention. This is because the main purpose of these management systems is to shorten delivery times and to ensure a proper technical execution [WBKL18]. Correct technical execution but faulty system behavior is not uncommon, especially for distributed system components [WBKL18]. This can lead to some unwanted side-effects like increased cost trough system failure.
and loss of reputation. A common pattern to tackle this issue and to prevent negative impact, is to make the web application *Cloud-Enabled* or build it *Cloud-Native* [TBB+15]. This increases scalability and reduces failure, because there are backup instances running.

Cloud orchestration software should offer a solution for this problem, but since it is today's practice that management functionality is either provided as infrastructural functionality by the cloud vendor or as third-party software, there is a big gap between the managed application and the managing tool [TBB+15]. Current orchestrating software uses a generic approach to be able to run and manage many different applications at the same time. This work will contribute by exploring possible solutions for component state monitoring and management, to move the managed application closer to the underlying management functionalities, to ultimately improve failure resilience and overall system health.

### 3.1 Related work

Based on the idea of Wurster et al. [WBF+19] to categorize automated deployment techniques, an attempt is made to find some categories for orchestration tools and techniques. They are intended to give the reader a better understanding of today's state of the art and where to put the prototypical implementation of this work. Three types of orchestration have been worked out (see figure 3.1):

*Cloud-Provider Orchestration*, where all management functionality is provided by the cloud provider, which increases the degree of vendor lock-in and multi-cloud deployments are not supported. Since they provide the runtime environment for the deployed cloud application, infrastructural health checks for the system are implemented, which increases system robustness and stability. Amazon Web Services (AWS) is an example for this type of orchestration. The drawbacks of *Cloud-Provider Orchestration* are that they only support simple scaling and configuration operations, but no custom, fine-grained operations on specific parts of the application.

*Third-Party Orchestration* has the advantage of being independent of the cloud vendor and therefore supports deployments and management functionality in different cloud environments. It reduces vendor lock-in but comes with the cost, that the management software is not built on top of the underlying infrastructure and therefore cannot always provide the same robustness in this regard. Also, the functionality must work in many different environments and needs to be configured more precisely which increases complexity and can reduce usability of the software. Automated deployment technologies with management support such as *Chef* or *Terraform* are fitting into this category. Their disadvantage is, that they mostly only support lifecycle operations and do not allow more complex operations to be run on specific application instances. There exist some approaches that use declarative process models to generate management plans from them, to automate the management of applications that fit into the *Third-Party Orchestration* category. They use planlets and annotations or enrich the deployment model with additional *Feature Component Types* to automate the management of applications [BBKL13] [HBS+19]. Generating plans is also done by Harzenetter et al. with the focus on state management, which is also an objective for this work [HBKL19]. Another approach is presented by Wurster et al. where they enrich deployment models with deployment tests, that can be executed during runtime of the application, to ensure a valid system state [WBKL18]. These approaches all use a declarative process model and generate workflows or plans. For the issue discussed in this work, executing management operations on a fine-grained level, i.e. on application component instances, this would generate a large amount of
plans and would become confusing. Also, there exist operations that do not trigger a state change and therefore it is impossible to model them with declarative behaviour and a concept is needed to execute them independent from the type of the process model.

The last category, *Built-in Orchestration*, is not really used yet and represents an ideal of where orchestration could be. It is the idea of the deployed web application dictating how to manage it, providing management functionality of its own, as well as keeping track of its own health. This way, management of an application would not be separated anymore from the application itself and could be more efficient, since the programmers could think of possible scenarios and could implement suiting management behavior and crisis management. The only approach that is fitting for this category, is introduced by Toffetti et al. where they propose a novel architecture for cloud applications, that enables scalable and resilient self-management [TBB+15].
4 Conceptual approach

As it was briefly mentioned in former chapters, it is hard to tightly couple automated management processes to applications after the development is finished. If key aspects for operating an application in production were not considered during development it is hardly possible to extend the application afterwards [WBB+14]. Therefore, it is important to consider various management operations during modeling time of the cloud application, because every operation has an impact on the global state of the application instance. But this is not just one-directional: a component state or the global state can impact the behaviour of an operation or it can even deny the operation. These interactions depend heavily on the management tasks and on the components of the respective web application. These circumstances will be addressed within the following concept.

To get a better understanding of this, the following figure 4.1 shows the topology of the MyTinyToDo application, which is a ToDo-List management application. This application runs in a Docker Container, which is executed by a Docker Engine that runs on a Ubuntu VM. The application is connected to a MySQL-DB that also runs on a VM. Both virtual machines are hosted on the OpenStack platform. Because all of the components in the topology are connected and dependent on each other, management operations have to be executed with caution. If a VM is stopped, all components that are hosted on it will stop working too. Also, an operation like connectToDatabase on the application is dependant on the MySQL-DB to be running, else this operation is not executed successfully.

The following sections examine how and into what states an application can get into and what strategies can be applied to ensure some valid and stable state management. Also, an attempt is made to classify management operations, to get a better understanding on their impact on the application. These classes are used to derive strategies to check if it is safe to execute a task or not. There exist differences between executing management operations on single nodes and groups of nodes. Those will be examined and discussed, to see if the strategies from the previous chapters can be applied in both cases.

All proposed and discussed approaches in the following sections have been prototypically implemented and tested for its applicability. The last section will provide the optimal theoretical approach for performing management operations on node instances.

4.1 Discussion on application states

The TOSCA specification provides a list of lifecycle states to describe a node instance’s state. The standard specifies the states so that they can either be transitional or permanent [TOSCA-Simple-Profile-YAML-v1.2]. While permanent states can only transition out of their state with a respective operation, transitional states have been induced by such and will change to a permanent state eventually, after the operation has finished. An example for this would be a create operation that
is executed to create the *DockerEngine* in figure 4.1. While the *DockerEngine* is created and instantiated, the state of it will be *Creating*, which is a transitional state and will end eventually. It will transition into the *Created* state, which is permanent as long as no other operation is executed on the component. Table 4.1 shows all node states that are known to the *TOSCA* metamodel.

Those states and their transitions follow a fixed set of rules and they are only meant to be induced by their corresponding lifecycle operations, which are introduced in the next section. However, there may be scenarios where the standard *TOSCA* states are not enough and should be extended. This would be the case, if a non-lifecycle operation was performed on a node, such as a *connectToDatabase* operation from the *MyTinyToDoDockerContainer* seen in figure 4.1. Currently, there is no transitional state available that could be used for that scenario, since a *working* state is missing, as well as a *waiting* state for a node that is dependent on some other node to finish its task. But this issue is not only true for transitional phases, but for permanent states as well. Some applications might need a more precise *error* state and split it into more categories, i.e. *fault, error*
4.1 Discussion on application states

<table>
<thead>
<tr>
<th>Value</th>
<th>Transitional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>No</td>
<td>Node is not yet created.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Node only exists as a template definition</td>
</tr>
<tr>
<td>Creating</td>
<td>Yes</td>
<td>Node is transitioning from initial state to created state</td>
</tr>
<tr>
<td>Created</td>
<td>No</td>
<td>Node software has been installed</td>
</tr>
<tr>
<td>Configuring</td>
<td>Yes</td>
<td>Node is transitioning from created state to configured state</td>
</tr>
<tr>
<td>Configured</td>
<td>No</td>
<td>Node has been configured prior to being used</td>
</tr>
<tr>
<td>Starting</td>
<td>Yes</td>
<td>Node is transitioning from configured state to started state</td>
</tr>
<tr>
<td>Started</td>
<td>No</td>
<td>Node is started</td>
</tr>
<tr>
<td>Stopping</td>
<td>Yes</td>
<td>Node is transitioning from its current state to a configured state</td>
</tr>
<tr>
<td>Deleting</td>
<td>Yes</td>
<td>Node is transitioning from its current state to one where it is deleted and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>its state is no longer tracked by the instance model</td>
</tr>
<tr>
<td>Error</td>
<td>No</td>
<td>Node is in an error state</td>
</tr>
</tbody>
</table>

Table 4.1: TOSCA lifecycle states from [TOSCA-Simple-Profile-YAML-v1.2]

and failure. These states could be used if the data in the database was not properly synchronized with the data available to the application. The approach proposed in this work addresses the introduction and handling of additional states to the application.

There are some major challenges with introducing new states to the specification that should be considered. The quickest way would be to just extend the existing list with predefined states, so it can suit a specific scenario. The drawbacks of this approach would be manyfold. Specifying additional states would deviate from the standard’s lifecycle phases and render the specification unnecessary (TOSCA Conformity). Also, if the developers are not thoughtful enough during modeling time, there could be states missing during runtime because they were forgotten to add to the model (State Extendibility). This comes with another issue: the state transitions and the operations which induce the state had also to be specified perfectly, because it is hard to change that during runtime (Transition Extendibility). To challenge these problems, two procedures that center around the idea of management operations carrying information about the state that they are inducing were created. Both approaches share the general workflow shown in figure 4.2.

The graphic 4.2 shows the communication between a Node 1 and an orchestrating component (which is called ManagementBus, it’s the name of the corresponding component in the OpenTOSCA ecosystem). After sending the operation, the ManagementBus receives and executes the given task. If the operation was performed successfully, a new state for the node instance is set and Node 1 will be informed about that, so that the visual representation can be updated. If the operation was not successful, the state of the node instance does not change. This needs to be reconsidered in future
work, since this behavior could cause some bugs if the operation failed after changing something in the system. A transitional error state could be a suitable solution, in which the component would run some tests to check functionality. After test completion it could transition into a permanent error state or into its last valid state. Another approach could be to enrich all operations with more than one state – one for the success case, another one for failure.

The main differences for the two approaches that were developed in this work are in the timing of when an operation gets enriched by the state information and how. The first method adds state information during runtime, which is why it will be referenced as DRT in the following passages. The operation is enriched with state information by the user during runtime, right before execution. This provides great flexibility for the applications state management, but also requires the users to know exactly into what state the operation will set the node instance. This approach would cover all mentioned problems State and Transition Extendibility as well as TOSCA Conformity perfectly. The Extendibility is secured because every operation carries information about the state and implicitly about how to transition to it. The Conformity is covered, since the method does not need to change any TOSCA models and works as a wrapper to the standard. Drawbacks of the DRT method are that additional data needs to be sent to the orchestrating component (state information) and it would compromise the overall logic behind the orchestrating unit. Usually, single nodes don’t know the whole system nor the operational logic behind it – this is information exclusive to the orchestrator. Consequently, this approach would be self-contained, just like a CSAR. The orchestrator wouldn’t know about the upcoming state for any operation besides lifecycle operations. This prevents the system from checking if an operation impacts the whole application or not before the user ordered it. This behaviour is not optimal because the user needs to be aware of the impact of the operation (some operation could set the state of an important node to configured/stopped).

To cover this issue, the DMT method was created. It enhances the TOSCA NodeOperation definition by a property called nextState. This enables the developers to define the states and transitions during modeling time of the application and increases the usability of those functions during runtime, because no further knowledge is required then. It also reduces the amount of data that needs to be sent to the orchestrating component over the network (in comparison to DRT) and enables the orchestrator to black- or whitelist specific operations, depending on their impact on the global

---

Figure 4.2: General approach to enrich operations with information for the induced state
4.2 Classification of management operations

A management operation is a task that is performed by some node instance(s) in the service template instance (in the OpenTOSCA environment). This can range from starting or stopping a component, up to a bash script running on a virtual server deleting unused files to free some space. Despite every operation being unique in the steps to complete the task successfully, they still share some characteristics (see table 4.3) amongst each other. Those attributes play a big role in how some tasks can impact the whole cloud application and also define how they are affected by other influences. To find a valid procedure that can handle all management methods, a classification of operations was created, based on their dependencies and impacts on the node instance state and their impacts on the application wide state.

### Table 4.2: Pros and Cons of the DMT and DRT method

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRT</td>
<td></td>
</tr>
<tr>
<td>- State extensibility</td>
<td>- Less usability</td>
</tr>
<tr>
<td>- Transition extensibility</td>
<td>- More network traffic</td>
</tr>
<tr>
<td>- TOSCA conform</td>
<td>- Orchestrator learns about &quot;NextState&quot;,</td>
</tr>
<tr>
<td></td>
<td>right before executing it</td>
</tr>
<tr>
<td>- Good flexibility</td>
<td></td>
</tr>
<tr>
<td>- Easy to implement</td>
<td></td>
</tr>
<tr>
<td>DMT</td>
<td></td>
</tr>
<tr>
<td>- State extensibility</td>
<td>- Less flexibility</td>
</tr>
<tr>
<td>- Transition extensibility</td>
<td>- Not TOSCA conform</td>
</tr>
<tr>
<td>- Good usability</td>
<td>- Harder to implement</td>
</tr>
<tr>
<td>- Less network traffic</td>
<td></td>
</tr>
<tr>
<td>- Orchestrator knows about &quot;NextState&quot;</td>
<td></td>
</tr>
</tbody>
</table>

### Table 4.3: Characteristics of management operations that influence their handling

<table>
<thead>
<tr>
<th>Dependent on other components state</th>
<th>Dependent on own state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Impact on own state</td>
<td>Impact on other components state</td>
</tr>
</tbody>
</table>

When comparing these two approaches, it comes to a trade-off of equally important attributes and it really depends on the scenario, which one is to favor. If the users know what they are doing and usability is not that big of an issue, the DRT approach should be used. If the users are no experts on the field or if its very important to work with the well-defined TOSCA standard, the second approach should be chosen. A third alternative that could be viable is a hybrid of those approaches, where the states and transitions get defined at modeling time but can still be modified by experienced users during runtime. This needs more investigation and can be looked at in future work.
Table 4.3 shows attributes of management operations that influence the procedure of how they are dealt with and processed by the orchestrating component. The first characteristic is, that an operation can be dependent on the state of the component on which it is executed. The second one classifies operations that are dependent on the state of other components (i.e. the global application state). Also, management operations can have an impact on either the components state on which they are executed on, or on other components, this is reflected with the last two characteristics in table 4.3. To evaluate the important properties, the standard lifecycle operations of TOSCA [TOSCA-Simple-Profile-YAML-v1.2] have been used:

- Create
- Configure
- Start
- Stop
- Delete

These operations induce the states presented in chapter 4.1 and they follow a standard sequence for starting or stopping a node [TOSCA-Simple-Profile-YAML-v1.2]. This means, that most operations are dependent on the state of the node they are executed on. To derive the other classes of management tasks, the topology seen in figure 4.1 served as big enough model to cover all important aspects. Creating the MyTinyToDoDockerContainer application without having the DockerEngine node instance running would not be possible and therefore the Create operation for the MyTinyToDoDockerContainer is dependent on the DockerEngine component. As a consequence, all operations that either have the dependent on other components state or dependent on own components state should not be available until the dependent component has reached a state where it is safe to execute the operation.

For the third attribute impact on own state it is very trivial, since every task impacts the current state (even though it might just be a transient state, which will end eventually).

More important to consider is the impact on other components state-characteristic that an operation can have. This is the case for the Stop operation, if executed on a node that has other nodes depending on it, like the Ubuntu VM in figure 4.1. If it got stopped, the whole application would stop working. In this scenario, the dependent nodes would be removed by force, since when the VM is not running, the other nodes can’t be working. But there are scenarios where the dependent nodes would still be active, for example if the MySQL-DB got stopped. The application would not be working correctly, but the other nodes would still be running.

For both scenarios, the orchestrator is required to run a procedure to ensure that all components get shut down correctly and that the global application state is the desired one by the user. For all four cases, a management task is relying on the state of the node where the operation is executed, and on the other components. For the depends on tasks, it is the current state, for the rest it’s the future state of a node. Correspondingly the orchestrating component needs a way to check the relations between all components to find possible impacts and dependencies. This is no problem, since the TOSCA specification provides RelationshipTemplates and together with the NodeTemplates of a topology, they form a graph that can be traversed.
A much larger issue is for the orchestrator to know, from which state it can transition and from which it is not allowed, for a specific operation. This is analogue for the impact on other nodes: which operation has such an impact on other nodes, that they need to perform some operations as well, to ensure a valid global state? The most common solution to this kind of problem would be to implement some state machine or a set of rules, that provide a pattern to use for every well-defined operation. Disadvantages of this approach would be that it is hardcoded, very generic for a fixed set of operations and not extendible during runtime. This approach would only work for known operations, such as the TOSCA lifecycle tasks. Another idea is to enrich operations (like the approaches in chapter 4.1) with rules that clarify, in which situation the task can be executed and how to deal with the impact. But for this use-case, that approach is to complex and error-prone for the users and also shares the negative aspects with the methods presented in the previous chapter.

If resources, implementing time and cost wasn’t a problem, the most optimal solution would be to simulate the whole system and execute the operation, to see if it performs well and to see its impact.

Each of the proposed methods has its own disadvantages and none is truly satisfying. Consequently, the right choice depends on the situation. The hardcoded pattern approach is the easiest one to implement and requires the least resources, but it is very inflexible. The second approach should only be used, if the users are experts and if its required, to be able to handle unknown operations. The last approach is the most impracticable and resource heavy, but it provides the greatest flexibility and will produce the least errors.

4.3 Executing operations

Executing an operation on a single component is straight forward. Usually, there is two scenarios to consider. In the first scenario, the component itself exposes some interface for the required task and this interface will get called to execute it. The other scenario consists of an operation that is executed by a different node but the one from which it was invoked. A good example for this situation would be a run script operation, where a script for some component (e.g. the DockerEngine from figure 4.1) is invoked, but the actual script does run on the underlying Ubuntu VM.

The reason why this is important, is that it requires another node to execute the operation and makes it therefore dependent of other components. There is also another case where operations are reliant on other node instances. When the management operation is not atomic but consists of various steps, that include more than one component, it is also dependent from other nodes in its execution. Such tasks can be represented with business modeling languages like BPEL or BPMN. In the worst case, operations that are dependent on other components can’t be executed or will get delayed if the node they rely on is not available. Performing functions on groups of the same node instance can help to overcome this problem and will be discussed in chapter 6.

The following chapter describes the implementation of the prototype that was created alongside with this work. Chapter 4 introduced some strategies for state management, state transitions and when to perform management operations, on single instances and on groups of node instances. The most optimal implementation would use the DMT approach from chapter 4.1 for state management, along with the simulation approach from chapter 4.2, which would work perfectly on single node
instances. Due to time constraints, technical issues and missing usability, some derivations of those strategies had to be applied to the prototype. These will be discussed in the next chapter in greater detail.
5 Implementation of a prototype

As mentioned in the previous chapters, it was a goal of this work to prove the concepts and approaches discussed in this paper by a prototypical implementation of them. This implementation was created by extending the source code of the existing OpenTOSCA ecosystem - it was not possible to provide the expansion as a plug-in since the changes were manyfold and in various places of the existing code. There are three main components to which changes have been made to. These will be described in detail in the subsequent sections. The TopologyModeler, has been restructured and refactored, so that it was possible to use it as a library for the OpenTOSCA UI, which is called TopologyRenderer in the following. Other major changes and additions have been made to the OpenTOSCA UI, so that the TopologyRenderer could be fully included and that management operations could be executed on application instances. All logic responsible for executing and performing tasks on the node instances has been implemented in the OpenTOSCA Container.

To prove and test the concepts introduced in this work, some application components where required to execute management operations on. Therefore, the MyTinyToDo ServiceTemplate has been used throughout the entire development process to ensure consistent and comparable results. The ServiceTemplate can be seen in figure 5.1. It consists of the DockerEngine, which is responsible for managing and running the MyTinyToDoDockerContainer. This relation is expressed by the HostedOn-Arrow in the graphic. Both NodeTemplates have properties that are required for the NodeInstances. Some of these properties have to be set during the provisioning of the instance. These are all fields in the figure that have the getInput command as value.

Each of the presented NodeTemplates has a well defined set of operations that can be executed on the provisioned NodeInstances. The MyTinyToDoDockerContainer has two interfaces with the following operations:

http://opentosca.org/interfaces/connections: connectTo

ContainerManagementInterface: runScript, transferFile

For the DockerEngine it is also two interfaces:

InterFaceDockerEngine: startContainer, removeContainer

http://www.example.com/interfaces/lifecycle: install, configure, uninstall

Some of those operations contain parameters that need to be set in order for the required function to work. For the testing of the created prototype the lifecycle operations have been mainly used. This has the reason that they are more easy to test and to confirm, since they often don’t need additional information provided by parameters of those operations.
5 Implementation of a prototype

Figure 5.1: Topology of the MyTinyToDo Application

5.1 Additions and Changes to the TopologyModeler

To execute management operations on node instances, it is necessary to have a graphical representa-
tion of all components. This representation should contain necessary state information and properties
of the running instances, as well as an option to display and select management operations for a
specific node instance. Because the OpenTOSCA ecosystem already provided the TopologyModeler
which has functionality to render NodeTemplates within a complete ServiceTemplate, it was the first
step to create a library that could be reused within the OpenTOSCA UI. As it was mentioned in
chapter 2.3.3, the OpenTOSCA UI is built with Angular, which uses the NPM to manage and install
dependencies and libraries to your project. To create a NPM Package the project needs to follow a
specific folder and file structure and it must contain some meta-information.

Figure 5.2 shows the refactored folder structure and the configurational files. The public_api.ts
defines the Angular Components which are accessible from the outside. All yellow marked JavaScript
Object Notation (JSON) files in the graphic contain information about how to compile, which
dependencies should be installed with the library etc. Under the path projects > topologyrenderer
> src > lib it can be seen that there is not only the TopologyModeler to be exported, but
also the apps ToscaManagement and WorkflowModeler. The rendering functionality is solely
implemented within the TopologyModeler but the other two apps had to be exported as well,
since the TopologyModeler had some dependencies to those apps and it was not possible to
5.1 Additions and Changes to the TopologyModeler

Figure 5.2: Folder structure and meta informational files for NPM library TopologyRenderer

resolve those without major code changes. Because the library should not contain the modeling functionality, just the rendering capabilities of the TopologyModeler, some changes had to be done to the existing TopologyModeler Component. Usually, it would work without any parameters since this was the Root Component of the Angular application. In the TopologyModeler the necessary data is loaded and provided with backend calls. This behaviour had to be prevented when data is passed directly to the component. Therefore the logic was adjusted, so that when the input variable topologyModelerData.topologyTemplate contains data, no backend calls will be made. To improve the usability of the input parameter, the variable contains another attribute topologyModelerData.configuration.isReadonly. This enables the TopologyModeler to retrieve TopologyTemplates from the backend but only for displaying purposes. When the input is configured in "rendering only" mode, the function initiateLocalRendering() is called, which can be seen in listing 5.1.

```javascript
1  initiateLocalRendering(tmData: TopologyModelerInputDataFormat, tEntityTypes: EntityTypesModel): void {
2      const nodeTemplateArray: Array<TNodeTemplate> = tmData.topologyTemplate.nodeTemplates;
3      const relationshipTemplateArray: Array<TRelationshipTemplate> = tmData.topologyTemplate.relationshipTemplates;
```
5 Implementation of a prototype

Listing 5.1: Excerpt from winery.component.ts

This function populates all variables with data that would have been gotten from the server. The excerpt shows the data structure which is expected to be used for the input variable. The data mainly consist of these attributes:

- nodeTemplates
- relationshipTemplates
- visuals
- readonlyPropertyDefinitionType

The first three items in this list contain information about the nodes, the relationships and their visual styles. The last property is a variable that determines the PropertyDefinitionType in read only mode. This is required, because usually there will be a backend call that determines that type, in another component. Many components had to be reworked like this, but the procedure is always like in the main component (disabling backend calls and populating necessary variables). The biggest challenge for this library to work in the OpenTOSCA UI was to merge their Redux Stores. A Redux Store is an object holding the state tree of the web application. This enables very powerful state management inside applications, because stores use well defined rules on how to change states. The TopologyModeler and the OpenTOSCA UI both use a redux store, but they use different libraries to implement them. By design, an application should only have one store configured, since it is a global object holding the state. Therefore including the TopologyModeler in the OpenTOSCA UI required to merge these stores to ensure that the application is properly working. This was only possible by recreating the state transitions and the possible states from the TopologyModeler in the OpenTOSCA UI store and by disabling the store in the TopologyModeler. A better solution could not be found since this would have required massive changes to the existing store in the
5.2 Additions and Changes to the OpenTOSCA UI

As it was mentioned in the last section the OpenTOSCA UI got the TopologyRenderer library installed. The existing store had to be extended to work with the new library. Figure 5.3 shows the detailed view of a node instance from the MyTinyToDo ServiceTemplate with the integrated TopologyRenderer.

As the graphic shows, there is no sidebar visible, which would allow to drag&drop new elements to the canvas. Also other editing features are disabled, such as the node properties (greyed out in the graphic) or creating new relations between nodes. When a node instance is clicked, a sidebar pops up and displays additional information (just like in the regular TopologyModeler). This information is loaded from the server and is a mixture of NodeTemplate information and node instance data. The backend calls get made when the entire view is loaded and the TopologyRenderer will not be available until all data has been gathered from the endpoint. This ensures proper functionality, else it could come to bugs when the user tries to work with the TopologyRenderer. A Open Management Operation Modal - Button has been added to the UI. It was a hard decision if this button with its underlying functionality should be added to the OpenTOSCA UI or to the TopologyModeler. The main reason why it got added to the UI was that it was cleaner to separate this functionality from the rendering functionality of the library. Also the OpenTOSCA UI holds all information from the environment and therefore knows the required API endpoints. When a node instance is selected and the new button is pressed, a modal with operations and their respective interfaces open up. The modal and all newly added UI-elements have been created with the existing design in mind, so that
they will integrate smoothly into the environment. Figure 5.4 shows this modal with operations for the *DockerEngine*. If no node instance is selected, the modal will still pop up and display operations from all components in the topology. The UI will run the operation on the correct components if one of them is selected and executed.

Pressing the *Next*-Button will open a similar looking modal which contains input fields for the parameters of the selected management operation. The *OpenTOSCA UI* will try to fill in the required (static) parameters of the management task. Static parameters are values that won’t change for a specific node instance, such as their *ID* or the *ContainerName*. This happens by matching the parameter names with information that is gathered from the server. This is not perfect for every information, since the server does not store all data in just one place. Also, sometimes the parameter name does not match the name of the variable as it is stored on the server. But for the prototypical implementation this solution works good enough to provide the user sufficient usability.

For dynamic parameters, they are required to be filled in by the user, i.e. the port on which the application should be running, or the database password to which the application should connect to. If the operation has no parameters, the modal will still show up and the user needs to confirm the execution, by clicking the *Next* button. After the user confirmed running the operation, the modal will close and the user can work while the task is executed asynchronously on the server. When a response is received, the *OpenTOSCA UI* will update its view and the state of the node instance will change accordingly.
5.3 Additions and Changes to the OpenTOSCA Container

In chapter 4.1 the DRM approach was introduced to be the most optimal. This would have involved to enrich operations with state information during runtime by the user. This was implemented into the prototype, but it turned out to be impractical. It required too much knowledge about the operation and its possible states and it was impractical on the server side, since the OpenTOSCA Container only expects lifecycle states at this moment and the whole logic that evolves around state management of the server would have needed to be reworked. So whenever a user would have sent a non-lifecycle state, the server would have not accepted it, so this was taken out for better usability. Figure 2.3.3 shows the updated state after successful execution of an operation. If the operation fails, the user will get the error response from the server - this can be improved by translating the technical messages to a better to understand language, so the user knows why the operation is failing.

5.3 Additions and Changes to the OpenTOSCA Container

Displaying models of node instances with combined information from the model and with their actual runtime data, requires an interface where this data is taken from. In the OpenTOSCA runtime environment the OpenTOSCA Container is responsible for providing data to the other services with an API. To match the new functionality of executing management operations on application components, this API has been enhanced by new endpoints, which can be seen in figure 5.6.

A new GET endpoint has been added which gets called by the UI. It delivers node instance and relationship instance data combined with data from the NodeTemplates. Most importantly, it contains the interfaces with all their operations mapped to the nodes and the current lifecycle state of the components. New Data Transfer Objects (DTO) have been created to hold this information while it gets sent to the OpenTOSCA UI. There this information will be used to populate and update the TopologyRenderer.

The /managementoperation endpoint is used to call and execute management operations. It works as a wrapper for the ManagementBus, which is an interface for executing buildplans and operations during deployment and provisioning of the webapplication.
This wrapper works as it is shown in figure 5.1. It is a POST-request and it takes the parameters that get sent via the request-body and transfers them to the management bus. The ManagementBus will execute the operation specified in the request on the selected node instance. The ManagementBus had to be modified to allow external invocation, since it was not possible due to cross-site scripting restrictions. If the operation fails, there will be an error message created and sent back to the user. After successful invocation of the management task, the wrapper function will set the state of the node instance to either its previous state (if no lifecycle operation was executed), or to a new state (if it was a lifecycle operation). It was said, that the state of the operation gets send with it, so that the OpenTOSCA Container knows into which state the operation sets the node instance, if it was no lifecycle operation. This code has been commented out, since it was impractical for the user and also had some technical restrictions (the setState() function could only take a valid lifecycle state as parameter and it was not clear how it would have impacted the whole system if this was changed).

In chapter 4.2 it was mentioned that when an important component stops working (due to an executed operation), that dependent components should be shut down accordingly. The OpenTOSCA-system can find out about dependent components by traversing the TopologyTemplate with its nodes and relations in a convenient way. More problematic is to find out which operation has to be performed to shut down a dependent component. This task is not easy, since all node instances have their own interfaces and lifecycles and there is no standard naming convention for all different nodes. If this problem could be solved, the user could get a preview on what components were affected by the

---

**Figure 5.6:** New OpenTOSCA Container API endpoints /topology and /managementoperation
Algorithm 5.1 Pseudo algorithm of `performManagementOperation()` in `NodeTemplateInstanceController.java`

```java
public Response performManagementOperation(requestBody) {
    // Read the parameters for the management operations from the request body
    parameters = getParams(requestBody);
    // Determine if its a lifecycle operation and set future state accordingly to the operation
    // else set it to state that got sent with the operation (code is currently commented out, since its not practical enough)
    if (getInterface(requestBody)) == "lifecycle" {
        if (getOperation(request_body) == "uninstall") {
            nextState = "DELETED";
        } elseif (...) {
            ...
        }
    } elseif (...){
        ...
    }
    // Call managementbus with operation and its parameters
    executeOperation(parameters);
    // If execution is completed without errors set "nextState" and send response.OK
    if (executionSuccessful) {
        setNodeInstanceState(nextState);
        return Response.ok();
    }
    // Else leave current state and send error code
    else {
        return Response.error();
    }
}
```

selected management operation and they were shut down properly. Right now, only the selected component will execute the specified operation and all dependent node instances will be affected by the result in an uncontrolled way. This is to be discussed in the chapter 6 along with other improvements that can be made to the approach and the prototypical implementation.
6 Conclusion and Outlook

The goal of this thesis was to explore ways around the problem of executing management operations on instances of application components. Another goal was to find improvements which are tightly coupled to the application, to ensure overall system health and decrease the gap between management software and the hosted web application. Due to the fact that the prototype was developed in the OpenTOSCA ecosystem, it was a requirement to ensure conformity with the TOSCA Standard. This work tried to achieve those goals in a generic fashion, so that it would be possible to execute any management operation on any component. The approaches to solve the stated problems have been developed by examining the core problems and splitting it to many parts. Those parts namely where state handling of running instances, the operation type with its impact to the system and the dependencies amongst operations. By solving the core problems of each of those parts and putting them together, whole concepts for executing management operations on application components could be developed. Some of the created concepts and approaches have been implemented to the OpenTOSCA ecosystem in a prototypical manner, to validate them and to test their practical use. Certain concepts haven’t been applicable either due to time constraints and to high implementation effort, or due to missing usability for the user.

The final approach used in the prototype is applicable for all operations on any node component instance. The state management for these operations is only available for lifecycle operations defined by the TOSCA Standard. For other operations no satisfying method could be implemented for state handling. The approach that has been implemented with the prototype modified and added code to the following components: the OpenTOSCA UI, the OpenTOSCA Container and the Winery (TopologyModeler).

The TopologyModeler has been modified, so that it could be exported as a library, the TopologyRenderer. Those changes involved some restyling of the css, so that the TopologyRenderer can smoothly integrate anywhere. Other major changes have been applied to the project structure as well as to its configuration to export it as a NPM Package. By adding some functionality to feed the component with data from outside and to use it only as a viewer of this data by disabling modifying functionality, the TopologyRenderer was complete.

The OpenTOSCA UI integrated the TopologyRenderer into its existing code as a library. This allows to display instances of components from a web application, along with additional data. Those instances will be enriched with runtime information as well as with data from modeling time. Consequently, this allows the user to make sophisticated decisions for if an operation should be executed or not. The UI will support the user by gathering the required information from the database whenever this is possible, to allow good usability. Also, the OpenTOSCA UI has been enhanced with functionality to select a node from the TopologyRenderer and to execute an operation on it.
This operation gets sent to the OpenTOSCA Containers REST API and is processed at a new endpoint that has been created for this. To ensure proper processing of the operation, the endpoint forms an interface and connects the already existing ManagementBus with some state handling functionality. The ManagementBus had to be modified to allow proper execution of operations on single application components. All modifications have been made by extending the existing codebase directly, not in form of a plugin. This had to be done since many changes affected the core functionality of the OpenTOSCA Container and the OpenTOSCA UI.

Implementing the prototype showed that executing management operations on node instances is possible within the OpenTOSCA environment. But doing this in a generic way is very hard and troublesome and usually reduces either usability or increases the complexity and the implementation effort. The biggest challenges will be described in the next chapter and possible attempts to solving those.

Outlook

Future work can consider many different aspects, some more theoretical, other more technical. Technical improvements could be made to the TopologyRenderer by refactoring the existing code to resolve the dependencies to other winery components. The library would get smaller and would not contain unnecessary code, unimportant for rendering topologies. The redux store should be built with the same technologies as the redux store from the OpenTOSCA UI. This would enable the library to be used as a true (and generic) plugin that requires no modifications on the app that integrates it. Also, more modifications could be made to the TopologyModeler, since it still displays buttons that are not required when it is used as a renderer.

Improvements to the OpenTOSCA UI and the OpenTOSCA Container would revolve around better usability by providing the user with previews, showing how an operation would affect the system, or by translating the technical error messages to a more understandable language. Nodes in the UI could be greyed out, if they are not usable anymore because a dependent component changed its state to deleted or stopped. These greyed out nodes should be shut down before a dependent component is stopped. Therefore a solution is needed to detect the correct operation of a dependent node to stop it. This is because there exist differences, since every node has its own interface with its own operations and there is no standard naming convention which would imply the resulting state of an operation. Currently, the prototype doesn’t change the state of a component if the operation failed. This behaviour could also be reworked by thinking of concepts like transitional error states, which will resolve eventually after some system checks.

The Built-In Orchestration mentioned in chapter 3.1 could be explored more by building software that integrates management functionality directly into the software, so that it does not rely on third-party software or vendor specific solutions. Also, a hybrid version of the proposed DRT and DMT approaches could be developed to further enhance execution of management operations.

Lastly, TOSCA specifies a concept for grouping application components together. TOSCA allows to group NodeTemplates together, to have a single entity that can be managed more easily and efficiently [TOSCA-Simple-Profile-YAML-v1.2]. This allows for effective scaling, since the orchestration tool can apply policies and changes to all components inside the predefined group. This can also be very helpful for executing node operations, but there are some things to consider.
The standard specifies a group of various *NodeTemplates* that are unique, but operations get performed on instances of *NodeTemplates*. Executing tasks on a group of different node instances is very challenging, because they do not necessarily share the same interfaces or operations and it would take a lot of effort to find out which operations they have in common. Only lifecycle operations could be used, since it is a common interface defined by TOSCA.

A much more accessible case is when those instances are all derived from the same *NodeTemplate*. They would share a lot of their base attributes, such as node operations, relationships and their topology. Without having to wonder if they share the same interfaces, performing operations on groups of instances can help to manage them more efficiently. A considerable advantage would be that if an operation would fail on one instance, the system would still have more instances that keep the system running. In this case, different strategies could be applied – the system could ask for confirmation if it should continue with the operation on the other copies, or it could do some error handling for the failed instance first.

Having more than one node instance of the same template would work similar to the *simulation* approach proposed in section 4.2. While there are great advantages, there still exist some issues, that need to be overcome with this approach. The orchestrator would need good emergency strategies to communicate the failure of a component to the other components and to distribute the additional workload to the remaining nodes, to ensure further availability of the web service and additional future work needs to be done on that topic.
Bibliography


All links were last followed on September 21, 2019.
Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature