Management of polyglot persistent integrations with virtual administrators

Thomas Clauwaert

Supervisors: Prof. dr. ir. Filip De Turck, Dr. ir. Gregory Van Seghbroeck
Counsellors: ing. Merlijn Sebrechts, Dr. ir. Gregory Van Seghbroeck

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Information Engineering Technology

Department of Information Technology
Chair: Prof. dr. ir. Bart Dhoedt
Faculty of Engineering and Architecture
Academic year 2017-2018

GHENT
UNIVERSITY

Foreword

TODO

Enkele opmerkingen:

Aanpassingen template

- ik heb de paginanummering anders ingesteld (alles voor chapter 1 telt in romeinse cijfers) de template zelf kwam anders op bvb pagina 11 uit wat eigenlijk de eerste pagina was

- een glossary en lijst van afkortingen toevoegen?

- TODO lichte achtergrond voor listings

- British english or american english

- TODO check bibtex entry for beyond generic lifecycle en orchestrator conversation: distributed management of cloud applications once paper is published

- TODO voorblad opnieuw genereren en importeren

Template extended abstract

Thomas Clauwaert

Supervisor(s): Dr. todo, todo

Abstract—**Nulla ac nisl. Nullam urna nulla, ullamcorper in, interdum sit amet, gravida ut, risus. Aenean ac enim. In luctus. Phasellus eu quam vitae turpis viverra pellentesque. Duis feugiat felis ut enim. Phasellus pharetra, sem id porttitor sodales, magna nunc aliquet nibh, nec blandit nisl mauris at pede. Suspendisse risus risus, lobortis eget, semper at, imperdiet sit amet, quam. Quisque scelerisque dapibus nibh. Nam enim. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nunc ut metus. Ut metus justo, auctor at, ultrices eu, sagittis ut, purus. Aliquam aliquam.**

Keywords—**Todo, tengu, todo, todo**

I. Inleiding

DEZE thesis ...

Morbi tincidunt posuere arcu. Cras venenatis est vitae dolor. Vivamus scelerisque semper mi. Donec ipsum arcu, consequat scelerisque, viverra id, dictum at, metus. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut pede sem, tempus ut, porttitor bibendum, molestie eu, elit. Suspendisse potenti. Sed id lectus sit amet purus faucibus vehicula. Praesent sed sem non dui pharetra interdum. Nam viverra ultrices magna.

II. Extensies

Nunc velit augue, scelerisque dignissim, lobortis et, aliquam in, risus. In eu eros. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Curabitur vulputate elit viverra augue. Mauris fringilla, tortor sit amet malesuada mollis, sapien mi dapibus odio, ac imperdiet ligula enim eget nisl. Quisque vitae pede a pede aliquet suscipit. Phasellus tellus pede, viverra vestibulum, gravida id, laoreet in, justo. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Integer commodo luctus lectus. Mauris justo. Duis varius eros. Sed quam. Cras lacus eros, rutrum eget, varius quis, convallis iaculis, velit. Mauris imperdiet, metus at tristique venenatis, purus neque pellentesque mauris, a ultrices elit lacus nec tortor. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent malesuada. Nam lacus lectus, auctor sit amet, malesuada vel, elementum eget, metus. Duis neque pede, facilisis eget, egestas elementum, nonummy id, neque.

III. Methodologie

Sed gravida lectus ut purus. Morbi laoreet magna. Pellentesque eu wisi. Proin turpis. Integer sollicitudin augue nec dui. Fusce lectus. Vivamus faucibus nulla nec lacus. Integer diam. Pellentesque sodales, enim feugiat cursus volutpat, sem mauris dignissim mauris, quis consequat sem est fermentum ligula. Nullam justo lectus, condimentum sit amet, posuere a, fringilla mollis, felis. Morbi nulla nibh, pellentesque at, nonummy eu, sollicitudin nec, ipsum. Cras neque. Nunc augue. Nullam vitae quam id quam pulvinar blandit. Nunc sit amet orci. Aliquam erat elit, pharetra nec, aliquet a, gravida in, mi. Quisque urna enim, viverra quis, suscipit quis, tincidunt ut, sapien. Cras placerat consequat sem. Curabitur ac diam. Curabitur diam tortor, mollis et, viverra ac, tempus vel, metus.

IV. Technologieën

Sed gravida lectus ut purus. Morbi laoreet magna. Pellentesque eu wisi. Proin turpis. Integer sollicitudin augue nec dui. Fusce lectus. Vivamus faucibus nulla nec lacus. Integer diam. Pellentesque sodales, enim feugiat cursus volutpat, sem mauris dignissim mauris, quis consequat sem est fermentum ligula. Nullam justo lectus, condimentum sit amet, posuere a, fringilla mollis, felis. Morbi nulla nibh, pellentesque at, nonummy eu, sollicitudin nec, ipsum. Cras neque. Nunc augue. Nullam vitae quam id quam pulvinar blandit. Nunc sit amet orci. Aliquam erat elit, pharetra nec, aliquet a, gravida in, mi. Quisque urna enim, viverra quis, suscipit quis, tincidunt ut, sapien. Cras placerat consequat sem. Curabitur ac diam. Curabitur diam tortor, mollis et, viverra ac, tempus vel, metus.

Logo is te zien op figuur 1.[1]

Fig. 1. Tengu logo.

Nulla ac nisl. Nullam urna nulla, ullamcorper in, interdum sit amet, gravida ut, risus. Aenean ac enim. In luctus. Phasellus eu quam vitae turpis viverra pellentesque. Duis feugiat felis ut enim. Phasellus pharetra, sem id porttitor sodales, magna nunc aliquet nibh, nec blandit nisl mauris at pede. Suspendisse risus risus, lobortis eget, semper at, imperdiet sit amet, quam. Quisque scelerisque dapibus nibh. Nam enim. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nunc ut metus. Ut metus justo, auctor at, ultrices eu, sagittis ut, purus. Aliquam aliquam.

V. Architectuur

Nunc velit augue, scelerisque dignissim, lobortis et, aliquam in, risus. In eu eros. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Curabitur vulputate elit viverra augue. Mauris fringilla, tortor sit amet malesuada mollis, sapien mi dapibus odio, ac imperdiet ligula enim eget nisl. Quisque vitae pede a pede aliquet suscipit. Phasellus tellus pede, viverra vestibulum, gravida id, laoreet in, justo. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Integer commodo luctus lectus. Mauris justo. Duis varius eros. Sed quam. Cras lacus eros, rutrum eget, varius quis, convallis iaculis, velit. Mauris imperdiet, metus at tristique venenatis, purus neque pellentesque mauris, a ultrices elit lacus nec tortor. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent malesuada. Nam lacus lectus, auctor sit amet, malesuada vel, elementum eget, metus. Duis neque pede, facilisis eget, egestas elementum, nonummy id, neque.

VI. Conclusie

Nulla ac nisl. Nullam urna nulla, ullamcorper in, interdum sit amet, gravida ut, risus. Aenean ac enim. In luctus. Phasellus eu quam vitae turpis viverra pellentesque. Duis feugiat felis ut enim. Phasellus pharetra, sem id porttitor sodales, magna nunc aliquet nibh, nec blandit nisl mauris at pede. Suspendisse risus risus, lobortis eget, semper at, imperdiet sit amet, quam. Quisque scelerisque dapibus nibh. Nam enim. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nunc ut metus. Ut metus justo, auctor at, ultrices eu, sagittis ut, purus. Aliquam aliquam.

References

[1] *Logo*, https://www.tengu.io/

Contents

List of Figures

List of Tables

List of Listings

"Instead of IT Operations doing manual work that comes from work tickets, it enables developer productivity through APIs and self-serviced platforms that create environments, test and deploy code, monitor and display production telemetry, and so forth. By doing this, IT Operations become more like Development (as do QA and Infosec), engaged in product development, where the product is the platform that developers use to safely, quickly and securely test, deploy and run their IT services in production."

~The DevOps Handbook when talking about NoOps

1 Introduction

One of the most crucial subjects in IT, is data. Everything revolves around it. Never was there so much generated data as today. Names, addresses, account numbers of people or generated sensor values from IoT-devices. There is no denying that the world as we know it today can't function anymore without all that information stored in databases. In addition, it is not considered acceptable anymore to have significant downtime in any services and all stored data needs to be accessible at any time. In such a world that resolves primary around attained knowledge from that data, the lifecycle on how to store, process and analyse the bits and bytes is of great importance. Operation and system engineers have the difficult task to create, deploy and maintain all deployed software and all maintained hardware. In a lot of cases the central design considerations focus on application development, more than service operations which might reduce the quality of the cloud application. The "The (5+1) Architectural View Model for Cloud Applications" brings attention to all aspects of cloud applications [3]. Methodologies like Agile and DevOps but also outsource-approaches have gained popularity over the last few years as they provide better development cycles, automation and scalability [4]. This research focuses on a concept within these so-called philosophies called service orchestration and its possibilities when creating virtual (system) administrators. These virtual entities then aid data scientists or developers to a more easily configuration of their wanted environments without the need of a physical operations engineer.

1.1 Data scientists as a scarce resource

The digital and technical evolution amounted to ecosystems were different types of data became the central entity no matter what the actual business is. Guerra et al. [5] describes this phenomena where he emphasizes that even computer science and engineering courses lack the "pedagogical practice in face of a reality that has required multidisciplinary, multidimensional, global, and contextualized preparation". This shows the lack of data analysis subjects and exercises in programs that focus on programming, computational insight and other IT-related skills. In a paper called "Integrating NoSQL, Relational Database, and the Hadoop Ecosystem in an Interdisciplinary Project involving Big Data and Credit Card Transactions" Rodrigues et al. [6] shows how he tackled a real life use case concerning big data tools and technologies for a group of 60 graduate students over the course of 17 academic weeks. The paper shows the complexity of the Big data subject and the technical and mathematical requirements and tools before proper analysis can be performed (graduates need to understand the concepts of NoSQL, Hadoop, MapReduce and Hive before they are able to start performing analysis's).

The whole categorisation of "IT-people" blurred somewhat the last decennia. The job title "IT-consultant" or "IT officer" is gone. Most people employed in IT have a specific range of tools, languages and technologies they have affinity with, but an even bigger list of things they don't know anything about. Even "fullstack engineers" need help with certain tasks. It is not possible anymore to be an all-round IT-person who knows everything. That means that the tasks of person A (a data scientist) are not the same of person B (a developer) even though both exercise similar programming challenges. Person C (an operations engineer) focuses on his turn on completely different topics. All three people are needed and they all perform "IT tasks". This illustrates that their direct needs and work (in most cases) do not align even though they are working for the same company or team. In some cases, a developer might perform some operational tasks and might act as an operations engineer but for the sake of clearness throughout this research the distinction will be made. A developer and a data scientist in this written piece are not supposed to know the so-called "operations knowledge".

1.2 Virtual administrators & Operations knowledge

Before defining the problem and goal of this research two important concepts need to be defined. Virtual administrators and operations knowledge are key points crucial for understanding on what level this research focuses and why certain approaches are used.

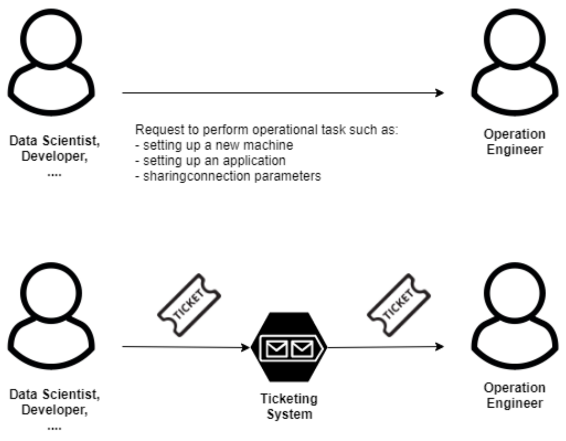

Data Scientist,
Developer,
....
Request to perform operational task such as:
- setting up a new machine
- setting up an application
- sharingconnection parameters
Operation Engineer
TICKET
TICKET
Ticketing System
Data Scientist,
Developer,
....
Operation Engineer

Figure 1.1: Traditional communications between operations and non-operations

1.2.1 Virtual administrators

To properly define the concept of a virtual administrator it might be interesting to look at some daily work cycles. These days most teams have real life administrators to perform operations. These operational tasks may vary from setting up and configuring new services and machines or monitoring existing applications. Some (often smaller) companies also make use of the possibility to outsource these tasks as they often want to focus on one specific thing without the cost and complexity of the tasks of a system administrator or network engineer. The same can be said about testing, quality assurance (QA), infosec, analytics... Figure 1.1 shows a common workflow on how non-operation minded people work together with system administrators. It is clear that there are at least two stakeholders. Because multiple people are involved, time management and planning becomes crucial. It doesn't occur frequently that people have the time and ability to perform operational tasks immediately, resulting in a slow process. The DevOps philosophy/approach introduced awareness and techniques to bundle forces between developers on the one side and operations engineers on the other. This way both teams can deal with the big differences between developing a piece of software and deploying or managing it. Both work together with the product as ultimate goal. This mindset is already a huge step forward and reduces the time it takes from *wanting* an operational task finished and actually *performing* it. The virtual administrator approach reduces this bottleneck completely. Figure 1.2 shows that an operation engineer can create a virtual administrator and provide this entity all the necessary tools in order to perform the tasks he otherwise needed to do. In this case a user (data scientist, developer, tester...) can make use of the virtual administrator at any time reducing the amount of stakeholders and time needed to get the requested operational tasks.

Imagine the (use) case where a group of developers (or data scientists, project managers, business analysts...) need some servers and applications (a database for example) for a new project. In a lot

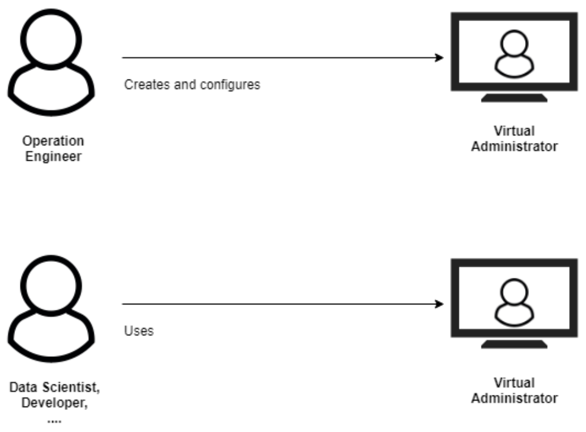


Figure 1.2: The concept of a virtual administrator

of cases either the developer has access to a database (or clean) server himself (and he needs to do everything himself) or he must properly ask an operations engineer or database administrator (DBA) to perform the steps that are necessary for the new project. These steps can for example include setting up a server, installing the software, creating users, sharing details... Note that in the first possibility the developer (or data scientist) requires specific knowledge and skills to perform these tasks. This knowledge is often not directly associated with his job as a developer. The idea of a virtual administrator is not taking away the job of the DBA or operations engineer but is rather the concept of creating a system, a tool, a way that focuses on performing the tasks in such a way that the developer can get what he wants in a time and cost-friendly way. Two key concepts come forth from this idea:

- No physical (other) person is needed or must intervene when setting up these servers, applications or services.

- No real operations knowledge (see below) must be known by the developer.

The virtual administrator handles all technical stuff and provides the requested services and details to the data scientist/developer in such a way that the data scientist/developer is ready to go and start carrying out his expertise without bothersome configuration issues.

A last theoretical approach is that things would really start to get interesting if virtual administrators interact with each other. These tools or services need to be built in such a way that they can be extended to different needs and that they can interact with each other for a fully fledged automated ecosystem.

1.2.2 Operations knowledge

The abstract concept of operations knowledge can be defined as the overall sum of specific technologies, their relations, configurations and limitations. This includes version numbers, technology-specific differences for the same (physical or logical) thing, configuration parameters... The following example might clarify this definition.

Imagine a team working on a webshop that sells items. The company wants to start performing some data analysis on the statistics and items of that webshop and they decide to put a team of data scientists to work. This team needs to setup their tools, configure the applications they want to use and they need to be able to get the data from the webshop. This task and how it is achieved is what falls under the domain of operations knowledge. The data scientists doesn't really care what database (MySQL, MongoDb, Cassandra...) or back- and front-end technology is used. All they want, is the ability to retrieve the data and work with it. In other areas though they might be a bit pickier as the data scientists might request some specific version (of a tool or technology) because of some specific feature. Note that the same exercise can be made with developers instead of data scientists. Developers generally don't care if you run your webserver on a Nginx- or an Apache-server or whether your database server runs on a MySQL or a Mariadb instance. This flexibility to request things when having the operations knowledge and at the same time being able to work on a higher (more abstract) level that doesn't require operations knowledge is not easily attained. With the help of virtual administrators (no operations knowledge) and operations engineers (with operations knowledge) a team should be able to have faster and easier deployment models without losing flexibility and power.

A small side note is the differences between relational database technologies (SQL) and non-relational ones. They are definitely important (these choices will be made by the team) but the exact setups, configurations and parameters are often only relevant for the people with "operations knowledge".

1.3 Problem statement

The previous section already briefly discussed that it isn't easy to find a lot of big data scientists with good hands-on experiences. Education institutions need the time to re-educate themselves and transition their courses to these subjects. In addition, the number of different topics to discuss in computer science and engineering studies keeps growing while there is only a fixed, limited amount of time. Therefore, it is difficult to find (or become) a data scientist that has the mathematical and statistical foundations and the skill set to program properly. A recent survey of Stack Overflow [1] shows that most people fall under the category "developer" (almost 60% identify themselves as Back-end developer). Job titles such as "Database administrator" (not even 15%), "System administrator (11,3%)", "De-

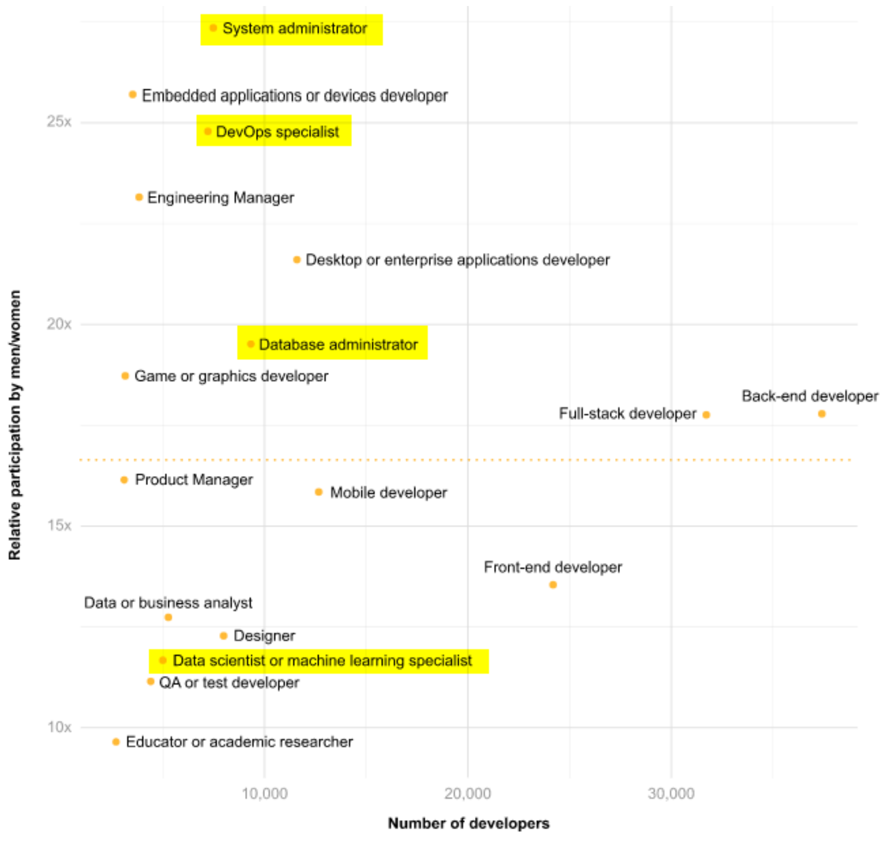


Figure 1.3: Stackoverflow survery 2018 [1]

vOps specialist (10,4%)" and "Data scientist or machine learning specialist" (7,7%) are less present and it is safe to assume that there are less people employed in these areas. Figure 1.3 illustrates the number differences on the X-axis. A data scientist (today) needs in a lot cases also the skills and knowledge to setup, install, configure their tools. Proper configurations and connections often cause a lot of frustrations and lost time. These operational tasks often need to be performed by another scarce group of people, the operation engineers. More frustrations may potentially arise when views differ. When you as a data scientist finally have reached the point where all tools, machines and services are configured correctly you might have lost the motivation to start. Things should be easier!

Configuration management tools like Chef, Puppet, Ansible... have grown over the last few years and more and more companies started to use them. This is already a huge step forward from the slowly manually steps one needed to take in a not so distant past. Tasks such as setting up a (virtual) machine, configuring and installing it shouldn't be performed manually anymore. All these (new) tools help the operation engineers in performing his tasks. These tools however are of no use for developers or data scientist who have no affinity with these operations. Services that offer preconfigured environments that hook into the big on-demand cloud computing platforms like Amazon AWS, Google Compute

Engine, DigitalOcean... are therefore definitely wanted.

1.4 Goal

The quote at the beginning of this chapter comes from “The DevOps Handbook, How to create world-class agility, reliability, & security in technology organizations” they finish with a call for action. They sell DevOps principles and patterns as a solution that “can help the creation of dynamic learning in organizations, achieving the amazing outcomes of a fast flow and world-class reliability and security, as well as increased competitiveness and employee satisfaction” [7]. Even though this is written from a DevOps point of view it can be directly mapped on the high-level goals and problems that are discussed in this research. When setting up tools, configurations and machines to work with, people without operations knowledge should be able to receive their requested operational entities faster and more easily. Things should be easier!

Aside from this “ultimate let's-change-the-world” goal, some smaller, more attainable, goals are exploring the possibility of virtual data administrators, their use cases, possibilities and limitations. Furthermore, the functional analysis of the OASIS TOSCA cloud modelling language concerning datastores and their relationships should give insight on how cloud based webservices and applications can be deployed, connected and managed more easily.

To answer some questions concerning implementing generic models that represent databases (see figure 1.4), some virtual administrators will be written to examine the proposed ideas and their feasibility in a tool called Juju. This iterative process (also represented on figure 1.4) tries to add more and more functionally and therefore flexibility for the user.

Finally there is a project called Tengu, created by Qrama a Ghent University / imec spin-off[1]. It is a platform build on top of Juju. Its goal is to provide all necessary tools and steps to get started with big data so that the customer doesn't need to bother with operations. Tengu acts as a virtual administrator where data scientists or developers can request operational tasks in such a way as if they would request it to a human person. If the virtual administrators working with generic database nodes in the technical implementation part of this research deem to be interesting and useful, they might get used by the Tengu-team as a starting point to provide virtual database administrator support inside Tengu.

[1]https://www.tengu.io/

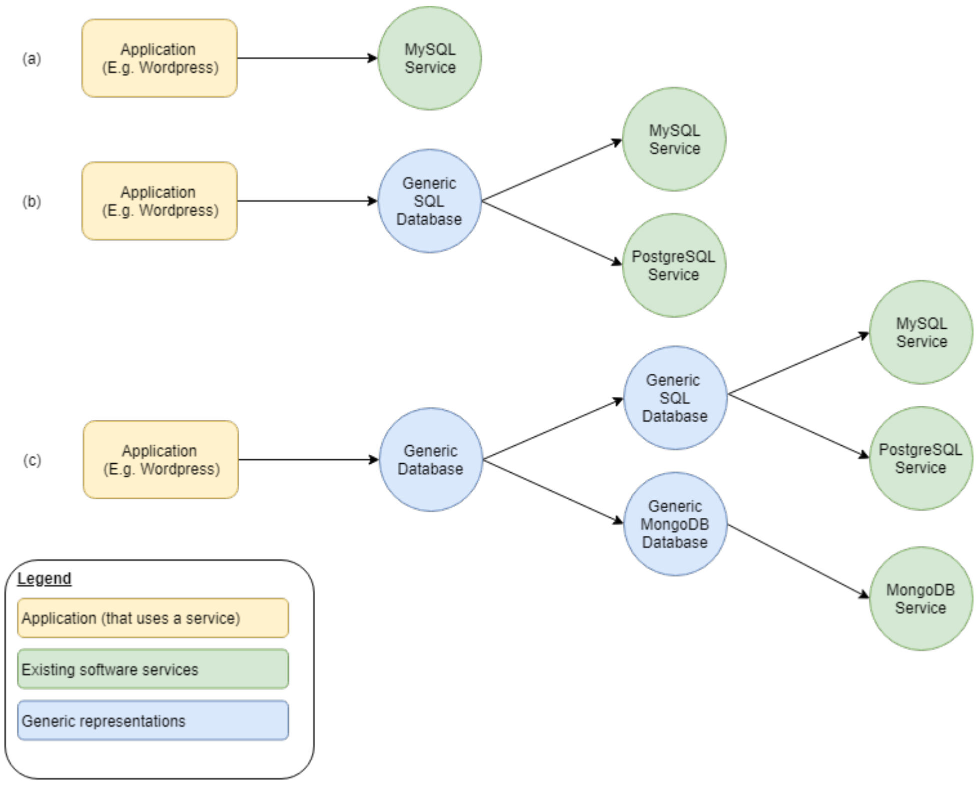


Figure 1.4: Modeling generic databases

1.5 Research questions

Because of the somewhat broad and yet narrow scope of this research it is interesting to define some big and some small research questions. Some focus on conceptualizing, others ask whether or not it is possible to implement some of these ideas and finally some are the goals discussed before formulated as questions.

- What kind of issues does service orchestration resolve and how do other approaches tackle these issues?
- What is the concept and scope of a virtual (data) administrator in this research conceptually, in OASIS TOSCA and in Juju?
- What answers can service orchestration and virtual administrators give in big data analysis, software development and operational tasks?
- Is it possible to create generic database models that give enough flexibility at design time?
- Is it possible to create virtual data administrators with the use of OASIS TOSCA?
- Is it possible to create virtual data administrators in the application orchestration tool Juju?

1.6 Overview

The next few chapters are organized as follows: chapter 2 will focus on existing tools and related work. It will define used concepts, languages and technologies such as OASIS TOSCA and Canonical's Juju. Some possible alternatives will be explored and some existing tools will be discussed as well and why they fall short. The similarities and differences between database technologies will also be defined as they will provide crucial information when defining generic databases. Chapter 3 will focus on a high functional level. Starting with some conceptual to-be scenario's and "wishful models". The rest of the chapter will follow an iterative process where each step defines a more generic solution working to the to-be situation following the OASIS TOSCA standard. A technical implementation, some proof of concepts, will be discussed in chapter 4. The different results in chapter 3 will be mapped onto Juju. These small proof of concepts will explore the practical use of virtual administrators and generic databases. Chapter 5 will discuss the results, limitations and possible future research or work. Finally chapter 6 wraps everything up in a conclusion.

"Knowledge is a treasure, but practice is the key to it."

~Lao Tzu

2 Background

This chapter will give an overview of some key components and concepts about operational tasks. A small overview of old and new approaches will be given accompanied by concepts such as Infrastructure as Code (IaC). Next, state of the art research around concerning virtual administrators and service orchestration will be thoroughly discussed. Next up the OASIS TOSCA modelling language and Canonical's Juju will be examined for respectively the functional analysis and technological implementation. Finally, for the sake of completeness some different type of database technologies are discussed and compared. From relational technologies such as MySQL or PostgreSQL to non-relational implementations such as MongoDB or Neo4j.

2.1 From infrastructure to infrastructure as code

When it comes down to setting up machines and applications or deploying developed software a lot of operational tasks used to be manual work. With virtualisation techniques and afterwards cloud computing possibilities other approaches were needed. More and more principles of software development found their way to infrastructure. "Infrastructure as Code" (IaC) is a term describing the act of setting up, managing and interacting with data centres or cloud computing models. The idea is that the configuration is written in files (typically YAML and JSON) on what the desired state should be for a specific machine (see listing 1 for an example). The concept of idempotence, meaning "that a deployment command always sets the target environment into the same configuration" [1], is very important. Configuration management tools such as CFEngine, Chef or Puppet can be seen as frameworks for IaC.

```
# execute 'apt-get update'
exec { 'apt-update':                    # exec resource named
↪ 'apt-update'
  command => '/usr/bin/apt-get update'  # command this resource
   ↪ will run
}

# install apache2 package
package { 'apache2':
  require => Exec['apt-update'],
  ensure => installed,
}

# ensure apache2 service is running
service { 'apache2':
  ensure => running,
}
```

Listing 1: Example of a Puppet manifest

IaC approaches and configuration management tools have improved the workflow for both developers and operation engineers. These tools however, only help when someone uses them who has "operations knowledge" (see section 1.2.2 Operations knowledge). Users can define certain parameters more easily and the deployment is faster but if the user does not know what to define and setup these tools wont help. In an introductory video about Juju and juju charms Jorge Castro talks about "service orchestration" as the next step. In his slideshow[2] he shows Figure 2.1 to illustrate the evolution. He states: "We see the next step being service orchestration, which is when you get to the level of scales when you are talking about thousands and hundreds of thousands of instances, you have to manage at the service level instead of the individual machine. You care about the individual machine but they become like CPU and RAM are today."

[1] https://www.visualstudio.com/learn/what-is-infrastructure-as-code/
[2] https://youtu.be/ByHDXFcz9Nk
[3] At time stamp 4:45

Figure 2.1: Representation of DevOps Evolution[3]

2.2 Service orchestration

An orchestrator in its simplest form is "a program" that interprets knowledge in the form of models or files and performs the necessary management actions in automated manner [8]. Orchestration should not be confused by a similar but different approach namely the "Web Service Choreography", a specification to define business processes with XML. Not everyone agrees on terminology and concepts[4] but the main difference lies in the management approach. Orchestration has a single entity ordering tasks and deciding things. With choreography there is no "conductor" (central management node), the "performers" (webservices for example) need to act on their own. A post on stackoverflow[5] visualises this nicely. Note though that because IT infrastructures can grow very complex one central orchestrator often lacks maintainability. Even meta-schedulers (decentralized scheduling) do not address the extensive needs in cloud modelling languages. To answer this issue Merlijn et al. propose a distributed orchestrator in the paper Distributed Service Orchestration: Eventually Consistent Cloud Operation and Integration [8].

The need for orchestration tools became apparent when knowledge reuse became a very requested phenomenon. Code reuse in software development in the form of libraries has been around for years. With an Infrastructure as Code approach it became clear that concepts such as abstraction and encapsulation became crucial in operations as well. In the paper Orchestrator Conversation: Distributed Management of Cloud Applications, Merlijn et al. [8] propose the orchestrator conversation. This approach should

[4]`https://www.infoq.com/news/2008/09/Orchestration`
[5]`https://stackoverflow.com/questions/4127241/orchestration-vs-choreography`

enable the reuse of knowledge.

2.3 OASIS TOSCA

"Topology and Orchestration Specification for Cloud Applications" or *TOSCA* REF3uitvolgende is an OASIS standard (first described in 2013) providing specifications to create self-contained cloud models that describe the topology of cloud applications alongside the management and orchestration in a workflow model [9]. To illustrate the concept of the TOSCA standard imagine following use case (based on slides from OpenTOSCA[6]: A small group of developers want to deploy a java WAR archive a Tomcat instance. The Tomcat server runs on an operating system (Ubuntu) which is hosted on a virtual machine on a cloud provider (AWS for example). TOSCA models these different entities and define capabilities (what do I *provide*) and requirements (what do I *need*). Properties are configuration parameter of a specific entity such as usernames, passwords, IP addresses, port numbers...

TODO afbeelding van dia 14 https://www.slideshare.net/OpenTOSCA/tosca-and-opentosca-tosca-introduction-and-opentosca-ecosystem-overview (kan slideshare als bron?)

At this point two workflows are possible. Imperative workflows and declarative workflows.

2.3.1 Imperative workflows

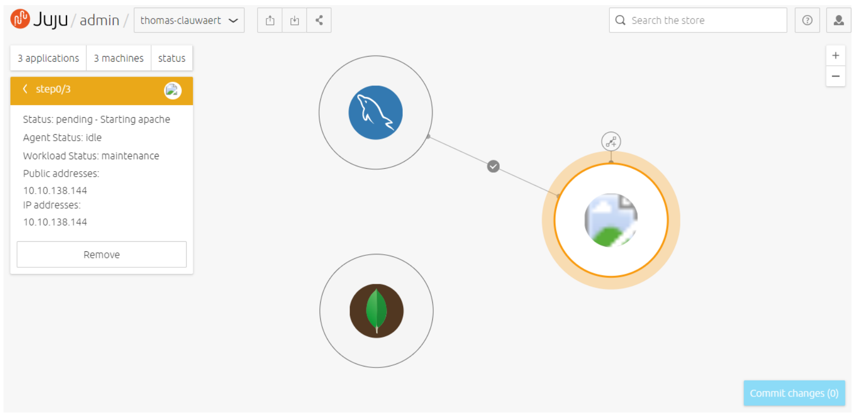
2.3.2 Declarative workflows

TODO declaratie + afbeelding simple yaml profile winery concepts

2.4 Juju

This section will give an extensive overview of the Juju platform. What Juju is, how it can be used and what goals it tries to achieve. Next, a summary is given on how Juju works under the hood. The main components such as charms, hooks and relations will be discussed and the new charms.reactive framework will be examined closely as it enables reusability and provides more and better compatibility [9].

[6] `https://www.slideshare.net/OpenTOSCA/tosca-and-opentosca-tosca-introduction-and-opentosca-ecosystem-overview`



Status: pending - Starting apache
Agent Status: idle
Workload Status: maintenance
Public addresses:
10.10.138.144
IP addresses:
10.10.138.144

Figure 2.2: Juju GUI

2.4.1 What is Juju

Canonical describes Juju as follows: *"Juju is a state-of-the-art, open source modelling tool for operating software in the cloud. Juju allows you to deploy, configure, manage, maintain, and scale cloud applications quickly and efficiently on public clouds, as well as on physical servers, OpenStack, and containers. You can use Juju from the command line or through its beautiful GUI"* [10]. Modern applications these days need multiple other services to work probably. Microservices, load balancers, worker and slave-nodes, caching tools... they all share multiapplication architecture. Even a website that uses a database consists of two different applications. Application modelling is the art of modelling the different applications with as goal to more easily manage (and scale) them.

When looking at the **Juju GUI** (see figure 2.2) it shows a visual representation of the different applications and how they are connected. This webapplication can be accessed by any user who has access to the model and who has a browser. The model represents an undirected graph where each node represents an application (and in a lot of cases a separate machine) and each vertex contains relation-specific details between the two applications. This visual level is clearly an example of encapsulating complexity for users.

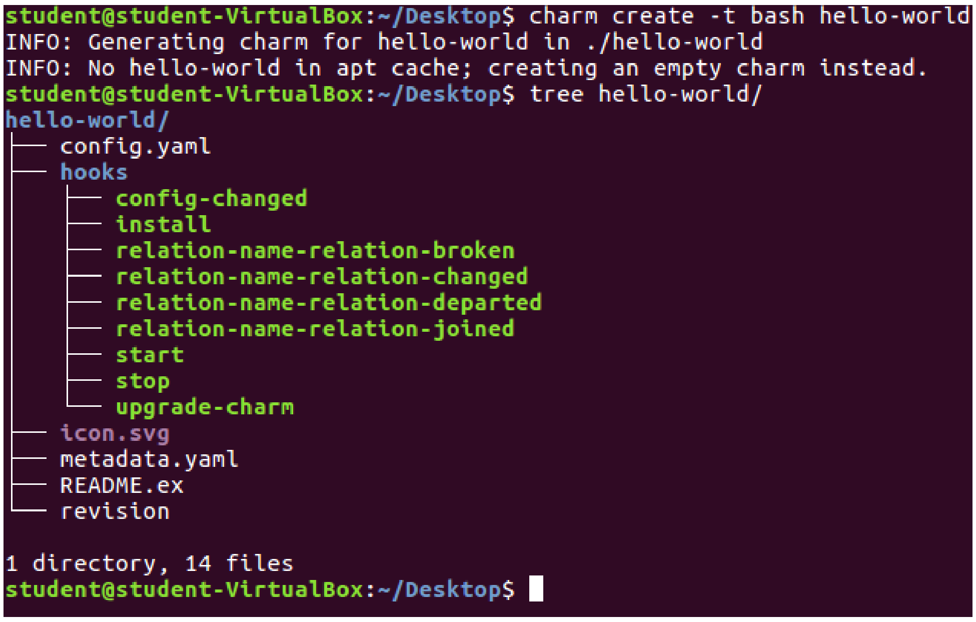
Juju uses **charms** and **bundles** to setup an infrastructure. Charms are the fundamental building blocks of Juju. They are a set of scripts for deploying an operating the application. Juju offers the possibility to write charms in any language (including existing configuration management tools such as Chef and Puppet). These charms are event-driven with as goal to reuse operational steps (or code) in different circumstances. If a team has multiple redundant setups for security or testing purposes for example the steps to configure them are similar if not completely the same. Bundles are collections of charms that are linked together. With the use of bundles a team can deploy a whole stack of technologies at once.

The great thing about these charms is that once they are written they provide a way of setting up systems without *"application-specific"* knowledge (hence operations knowledge). Things like dependencies, operational events like backups and updates can all be encapsulated in the charm. The stronger the knowledge of the application of a charm writer, the more options will be available and the more flexibility one can have when designing in the Juju GUI. Once done a user (data scientist) can then completely manage their infrastructure without bothering anyone else. In other words when the charms or bundles are written, Juju and its charms act as virtual administrators for the user.

While charms and bundles are the fundamental building blocks of Juju, a user gets confronted with some other concepts first. After installing Juju and optionally setting up the credentials for public cloud environments such as Amazon Web Services, Windows Azure or Google Compute Engine one can *"bootstrap"*. This means creating a controller for that specific cloud environment. A **juju controller** (which is also a machine) is the central communication and management node for a cloud environment.

Thanks to the controller it is then possible to create **Juju models**. A model is always associated with one controller. Models can be easily added, destroyed or modified by users. It is at this level that operations engineer also can invoke security by granting users only access to specific models. In this model charms can be added and linked together. This is the environment where "modelling an infrastructure" becomes possible through the use of Juju GUI.

Before continuing there is one big remark. Juju is built around the idea of application modelling. This means that the philosophy of Juju is fundamentally different (then the goals of this research) and the focus is thus somewhat different. Juju wants to model applications (and nothing else). Every charm gets deployed on a different machine (or container with subordinates) meaning that every charms actually "deploys" something. Modelling generic "entities" from a model "all the things" (or at least the things that seem interesting) perspective is not possible. Nodes that aren't fully fledged applications or instances but rather entities that represent some logical item that holds some sort of information (or mostly configuration) cannot exist on their own. A small disclaimer inside this disclaimer (but more on this in the functional analysis) is how far should one go "modelling all the things"? "Where do we draw the line?". Is the logical representation of a database enough or is there a need for nodes that represent tables or documents as well? When talking about database technologies, what about users, views, stored procedures, indices...?



```
student@student-VirtualBox:~/Desktop$ charm create -t bash hello-world
INFO: Generating charm for hello-world in ./hello-world
INFO: No hello-world in apt cache; creating an empty charm instead.
student@student-VirtualBox:~/Desktop$ tree hello-world/
hello-world/
├── config.yaml
├── hooks
│   ├── config-changed
│   ├── install
│   ├── relation-name-relation-broken
│   ├── relation-name-relation-changed
│   ├── relation-name-relation-departed
│   ├── relation-name-relation-joined
│   ├── start
│   ├── stop
│   └── upgrade-charm
├── icon.svg
├── metadata.yaml
├── README.ex
└── revision

1 directory, 14 files
student@student-VirtualBox:~/Desktop$
```

Figure 2.3: Charm (bash template) structure

2.4.2 Juju internals

Structure, hooks & relationships

As previously stated, with juju everything resolves around charms. Figure 2.3 shows the charm structure in its simplest form. The *config.yaml* (see listing 2 for an example that holds some options for a deployed http website) file holds the different options that will be accessible by the end user. The *icon.svg* is the image used in the Juju GUI to represent the service and the *README* should offer some explanation about the charm for other uses. *Revision* is optional and rather deprecated. The *metadata.yaml* is another important file. A simplified version of the Haproxy service is given in listing 3. The first few lines give some information but the "provides" and "requires" tags are crucial. They define how charms can interact and communicate with each other. Finally there is a folder called *hooks*. A hook is an executable file (written in any language that can be interpreted by an Ubuntu machine). These files will be called by the juju unit agent depending on different events. The hooks inform Juju what events happen and what actions the charm should take.

TODO lifecycle of hooks

```
options:
  website-name:
    type: string
    default: "My Website"
    description: "The title of your website"
  port-number:
    type: int
    default: 80
    description: "Port to run website on"
```

Listing 2: Example of a config.yaml file

```
name: haproxy
summary: "fast and reliable load balancing reverse proxy"
maintainers: [Juan Negron <juan@ubuntu.com>, Tom Haddon
↪ <tom.haddon@canonical.com>]
description:
 HAProxy is a TCP/HTTP reverse proxy ...
tags: ["cache-proxy"]
series:
  - trusty
  - ...
requires:
  reverseproxy:
    interface: http
provides:
  website:
    interface: http
...
```

Listing 3: Example of the Haproxy metadata.yaml file[7]

An example of a very basic *install-hook* is shown in listing 4. The language used is bash. It installs the apache software and deploys a basic website.

```
#!/bin/bash

set -eux

apt-get install apache2 -y
a2ensite 000-default
echo "<html><body>Hello World!</body></html>" >
↪ /var/www/html/index.html
service apache2 restart
```

[7] https://api.jujucharms.com/charmstore/v5/haproxy-43/archive/metadata.yaml

Listing 4: Example of an install hook

Reactive framework

In section REF the concept of "declarative workflows" were discussed. In the paper "Beyond Generic Lifecycles", Merlijn et al. discuss some limitations such as inflexibility or good support to reuse certain steps. The reactive framework is an answer to those shortcomings. They speak about "emergent workflows" using declarative flags and handlers [9]. Through the use of When decorators (annotations), a charm writer can easily define conditions whenever the framework should "react" (hence the name charms.reactive[8]).

Because the reactive framework offers more flexibility and reusability in the form of layers it should be the preferred method when writing charms. An example is given in listing 5.

```
@when('apache.available', 'mysql.availale')
def setup_app(mysql):
    render(source='configuration.php',
        target='/var/www/configuration.php',
        owner='www-data',
        perms='0o775',
        context={
            'db': mysql,
        })
     set_state('apache.start')
     status_set('maintenance', 'Starting apache')
```

Listing 5: Example of a handler in the reactive framework

TODO afbeelding architecture of charms.reactive framework

2.4.3 Conjure-up

Conjure-up[9] is a tool build on top of Juju. Its goal is to provide even less know-how and faster setting up times. With a mindset as "Start using your big software instead of learning how to deploy it." their focus aligns with the goals of this research. Because Conjure-up is nothing more than a layer on top of Juju (and therefore they use the Juju charms and bundles), this tool wont be used.

[8]https://charmsreactive.readthedocs.io/en/latest/
[9]https://conjure-up.io

2.4.4 Juju as a solution?

When looking back at the problem stated in section REF, it is interesting to look at the workflow of Juju users. Imagine the use case where a team has Juju installed and correctly configured. A data scientist wants to start performing some analysis. He needs two key aspects: his Big Data environment and the source of the data he needs to analyse. The setup of his tools (for example a Hadoop and Spark cluster) is something Juju can quite well in the user-friendly Juju GUI. Setting up the connection and relation between these tools and (non-)existing datastore units shows some more issues. When tackling this use case practically there are a few scenarios possible (we assume the data scientist is given a Juju model to work with):

1. The operation engineer has already predefined all data store related entities. What's left to do for the data scientist is setting up his tools and adding relationships.
2. The data scientist has access to charms from another model through cross model relations. This idea might be conceptually the most logical one as data is stored in the datastores by other services then what the data scientist uses.
3. The user doesn't want to interact with juju models at all (looking at tools that exist on top of Juju: Conjure-up, Tengu or custom frameworks). All he wants to say is: "setup and configure everything for me".
4. (Proposed) The data scientist (and/or the layers on top of Juju) model everything but through generic entities such as a charm representing a database instead of a concrete application.

It might be clear that approach 1 still requires some manual work from the operation engineer, something fundamentally against the goal of this research. Approach 2 seems very promising but the Juju GUI offers no support whatsoever for cross model relations meaning the visual representation for non-technical users is of no help. In addition, there is still need for an entity in the model of the data scientist representing the datastore (probably in the form of a generic database) or this approach becomes similar to approach 1. Also note here that while Juju has its limitations, Juju focusses on modelling applications and applications only. Approach 3 is the ultimate goal for both the data scientist and the operation engineer if it can all work in an automated way. Point 4 is the proposed idea of the virtual data administrators filling the gap and making approach 3 possible.

Overview of things to keep in mind concerning Juju:

- Juju is more powerful than the Juju GUI, but using all its capabilities requires some more knowledge of the CLI on one hand and some experience of Juju on the other.

Table 2.1: Juju terms

Concept	Meaning	Example
Cloud	Resource that provides machines	AWS, MAAS, LXD
Controller	Initial cloud instance that functions as central management node	-
Model	A model has 1 controller and is the playfield for deploying applications	-
Charm	The sum of instructions needed to install and configure applications on machines	MySQL, Wordpress
Bundle	A collection of charms	wiki-simple
Machine	An instance of the cloud (mostly has an IP)	-
Unit	Deployed software	-
Relation	The concept of connecting multiple units through interfaces	-
Agent	Software on a juju machine to keep track of state changes	-

- Juju focusses on application modelling. Every charm represents a machine (every subordinate represents a container) and that machine runs a service. Modelling more (or other) things that represent other ideas or concepts than applications is something out of the scope of Juju.
- Juju and its charms allows many languages but the reactive framework REF seem to be the preferred way. REF

2.4.5 Alternatives

There are a lot of tools out there each with their own strengths and weaknesses. Most ease the operational activities for an operations engineer or focus on automating these tasks (see section 2.1). This is not the area where Juju shines. Juju provides the flexibility of its virtual administrator and the availability of reusable charms and bundles. When looking at other tools to replace Juju and that operate at the same application modelling level some characteristics need to be defined as metrics:

- What happens (what is the lifecycle) when two applications need to be connected?
- Are users able to work, deploy with the tool without knowing operations knowledge?
- Does an application request one specific other application or does it allow generic types?
- Is it possible to reuse parts of existing models or setups?
- Is it possible to connect to one and the same database with multiple applications?

When looking at these questions it already becomes clear that most configuration management tools or PaaS (Platform-as-a-Service) like solutions do not provide enough flexibility or possibilities. The need for tools that work on the level of service orchestration (see section 2.2 Service orchestration) and offer that sort of possibilities is crucial.

TODO alien4vloud+terraform, mcollective ... https://zeebe.io/

2.5 Database technologies

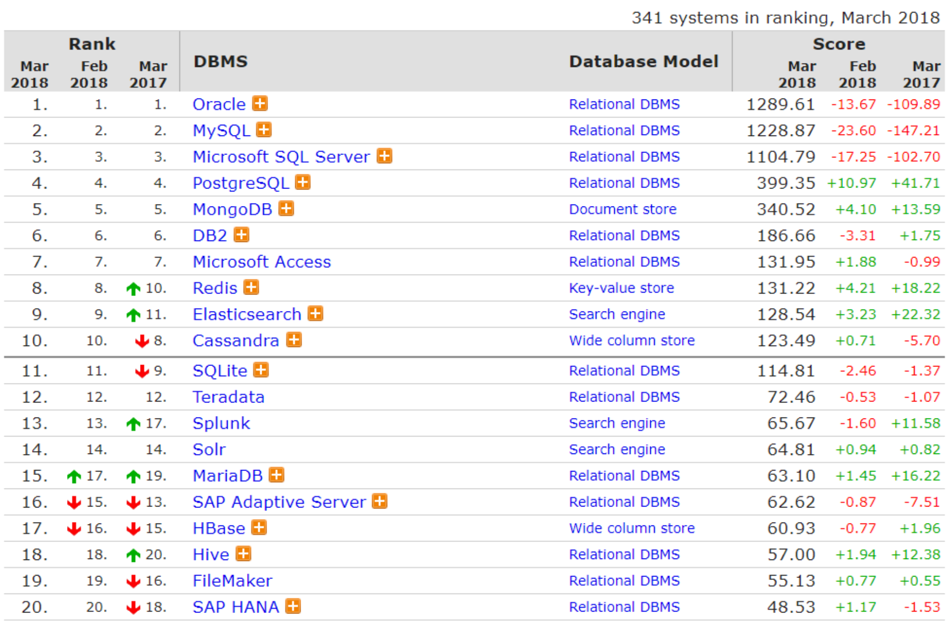
A database is defined as an organized collection of data [10]. Unlike a database a datastore allows a broader range of storage systems like for example the distributed data stores Apache Cassandra. Throughout this dissertation the terms database and datastore will be used interchangeably referring to one and the same concept: storing data in some way or another. Some concepts and terms in the database domain are universal such as users or indexes while others are unique to the type of technology like stored procedures and views.

There are two big different types of categorizations when it comes to database technologies. There are the relational (SQL) defined technologies and the non-relation (NoSQL) type of implementations. The SQL-technologies share some concepts as table (relations), rows (records) and columns (fields). The NoSQL-technologies allow different classifications such as key-value store (Redis) systems or graph approaches (Neo4J) for example. Note that while the term "NoSQL" is relative young the concept of non relational database has been around for quite some time now[11]. Figure 2.4 gives an overview of the most popular technologies in March 2018.

2.5.1 Relational (SQL)

In 1970 E. F. Codd proposed the concept of a relational database [11]. The main language to communicate with a relational database is the querying language SQL (Structured Query Language) hence why relational database systems are often referred to as SQL databases. Note that a lot of "dialects" exist in this commonly used language depending on the type of technology and what version is running. Oracle and Microsoft's SQL Server were probably the best examples but as time went by support for most features found their way to both technologies (but sometimes slightly different). The core SQL language stays fundamentally the same just like the general concepts used in every relational database. Table 2.2 gives an overview of some frequently used terms and their meaning. The next sections will examine some technologies highlighting their key (unique) characteristics.

[10]https://www.merriam-webster.com/dictionary/database
[11]http://www.leavcom.com/pdf/NoSQL.pdf
[12]Source: https://db-engines.com/en/ranking



341 systems in ranking, March 2018

Rank			DBMS	Database Model	Score		
Mar 2018	Feb 2018	Mar 2017			Mar 2018	Feb 2018	Mar 2017
1.	1.	1.	Oracle	Relational DBMS	1289.61	-13.67	-109.89
2.	2.	2.	MySQL	Relational DBMS	1228.87	-23.60	-147.21
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1104.79	-17.25	-102.70
4.	4.	4.	PostgreSQL	Relational DBMS	399.35	+10.97	+41.71
5.	5.	5.	MongoDB	Document store	340.52	+4.10	+13.59
6.	6.	6.	DB2	Relational DBMS	186.66	-3.31	+1.75
7.	7.	7.	Microsoft Access	Relational DBMS	131.95	+1.88	-0.99
8.	8.	↑ 10.	Redis	Key-value store	131.22	+4.21	+18.22
9.	9.	↑ 11.	Elasticsearch	Search engine	128.54	+3.23	+22.32
10.	10.	↓ 8.	Cassandra	Wide column store	123.49	+0.71	-5.70
11.	11.	↓ 9.	SQLite	Relational DBMS	114.81	-2.46	-1.37
12.	12.	12.	Teradata	Relational DBMS	72.46	-0.53	-1.07
13.	13.	↑ 17.	Splunk	Search engine	65.67	-1.60	+11.58
14.	14.	14.	Solr	Search engine	64.81	+0.94	+0.82
15.	↑ 17.	↑ 19.	MariaDB	Relational DBMS	63.10	+1.45	+16.22
16.	↓ 15.	↓ 13.	SAP Adaptive Server	Relational DBMS	62.62	-0.87	-7.51
17.	↓ 16.	↓ 15.	HBase	Wide column store	60.93	-0.77	+1.96
18.	18.	↑ 20.	Hive	Relational DBMS	57.00	+1.94	+12.38
19.	19.	↓ 16.	FileMaker	Relational DBMS	55.13	+0.77	+0.55
20.	20.	↓ 18.	SAP HANA	Relational DBMS	48.53	+1.17	-1.53

Figure 2.4: Most popular technologies in March 2018[12]

Table 2.2: Relational database and SQL terms

Relational database term	**SQL term**	**Meaning**
Relation	Table	Structured collection consisting of columns and rows
Record	Row	Collection of fields, representing a single item
Field	Column	One specific, labeled attribute of a record
Unique key	Primary key	Unique defined attribute

MySQL

TODO Some info on mysql and key features

Mariadb

TODO

PostgreSQL

TODO

2.5.2 Non-relational (NoSQL)

TODO

Mongodb

TODO

Cassandra

TODO

Neo4j

TODO

When talking about databases and datastores, the general idea is a place where some data is stored. The act of storing data has been around for more than fifty years dating back to the navigational Database Management System (DBMS). Because most navigational databases were used on mainframes and with the COBOL language their popularity is close to nothing [13].

[13]`https://db-engines.com/en/ranking/navigational+dbms`

The next step in database history was the relation DBMS. The idea of using tables with a limited amount of records gave birth to the relational model. Codd's twelve rules [12] show the requirements of a DBMS before it gets promoted to a RDBMS. The differences and similarities between RDBMS will illustrate what characteristics a generic database should have before calling it a "generic relational database" or "generic SQL database".

Finally, there are the NoSQL and NewSQL group of technologies. "Not only SQL" technologies are often used in scenarios where the traditional tables (and its relations) fall short. Streaming services and Big Data solutions are often use cases where NoSQL shows its power.

2.5.3 Summary

TODO overzicht (tabel) van alle technologieën met hun belangrijkste karakteristieken.

2.6 Other tools & technologies

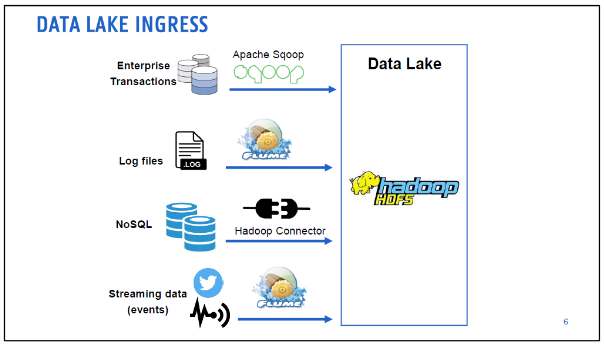
For the sake of completeness this research sometimes refers to other technologies or tools. This section gives a brief overview and explanation of tools not directly relevant to this research but which are referred to in concepts, use cases or examples.

2.6.1 Big Data

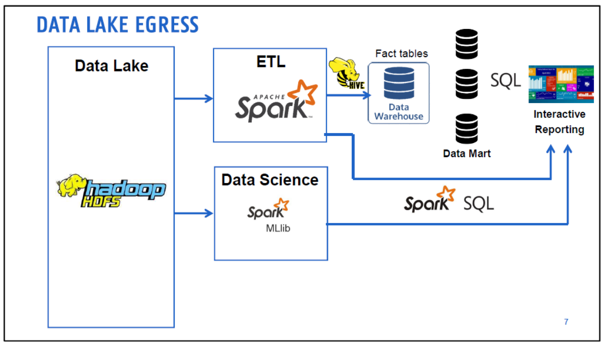
This research isn't about big data or data analysis but it is interesting to illustrate some aspects in this ecosystem as it will show the different tools a data scientist needs when working with big data. Furthermore (as stated in goal REF), this research focuses primarily on data scientists as users.

First, a key concept in big data is the concept of a data lake. The idea is to create a centralized pool where all different types of data is gathered (from relational and non-relational databases to unstructured data such as emails or documents). The Hadoop Distributed File System (HDFS) acts as such a data lake. Tools like Sqoop, Flume or Kafka (all from Apache) help importing data into HDFS. Spark and Hive on the other hand provide a way to read from the Data Lake. Figures 2.5a and 2.5b from the course "System Design" by P. Simoens illustrate how these technologies can be used together [2].

While these tools are mainly focused to gain knowledge from the big data set a little thought experiment could lead to some interesting use cases concerning transforming from one database technology



(a) Putting data in the data lake



(b) Getting data from the data lake

Figure 2.5: The Hadoop Distributed File System (HDFS) ecosystem [2]

to another. Imagine filling an empty lake with relational data from a Mysql-server. With the use of MapReduce or Spark it would be theoretically possible to pull the data in a modified form so that it would fit a MongoDB structure for example. This lies beyond the scope of this research. For more information concerning this topic consult the doctorate of Thomas Vanhove called “Management of Polyglot Persistent Environments for Low Latency Data Access in Big Data” [13].

2.7 Conclusion

This section summarises briefly this chapter. The main objective is to answer the big complexity of operation knowledge for (primarily) data scientists and to some degree developers, project managers, infosec, QA, business... As these types of people already have different concerns it is not easy to find or construct a one tool fits all. Some will want to visualise everything (in a BPMN kind of way) while others want only the crucial information (only applications like Juju). Finally, there are multiple datastore technologies that can be used. Most have certain strengths in one area but shouldn’t be used in another. More and more application-stacks tend to become polyglot persistent (see REF) meaning that they use multiple database technologies together. There are differences and similarities between these groups and these will be crucial when encapsulating configuration-specific parameters and building generic database models.

TODO TOSCA limitations

TODO Juju limitations

The different database technologies will need to act on the same level with the same concepts and the same terms. Support for features will need to be implemented through iterations when possible and with a sharp eye on not breaking previous workflows on other technologies.

Finally, existing applications and or big data technologies should be able to use the new proposed generic databases in OASIS TOSCA conceptually and implementation wise with the proposed Juju charms.

3
Functional specification

In this chapter a conceptual approach and outline is given of the generic database concept. First some terms are clarified for a proper understanding with the help of the OASIS TOSCA standard. Next, the generic database concept is discussed through an example use case. Afterwards, a clear definition illustrates that the generic database only works on the operational side of services leading into an explanation about certain design choices. Finally, some remarks wrap up this chapter.

3.1 Terms and visualisations

3.1.1 Application modelling

Through this thesis, the idea of application modelling is followed to visualise software stacks or infrastructure ecosystems. In these models, nodes represent applications or services and the vertices represent the relations between them. Graphs are usually used in computer science and mathematics to represent data types that are related to each other in one way or another. Table 3.1 summarises the different terms used throughout this thesis as these will be used interchangeably. Note that the OASIS TOSCA standard, models more than only applications and services. Unless otherwise stated, each graph used in this research, represents an application model that omits entities such as machines or operating systems. An example of such an application model is given in figure 3.1 where a Wordpress

Table 3.1: Relation between the conceptual terms and how it is visualised

Conceptual	Visualised	Meaning	Remarks
Application model	Graph	The full software stack with all its components and underlying relations	A full ecosystem of software components.
Service/ Application	Node	A software component providing functionality	Acts as a self-providing virtual administrator.
Relation	Vertex	The relation between one or more services	This often indicates shared data between the services.

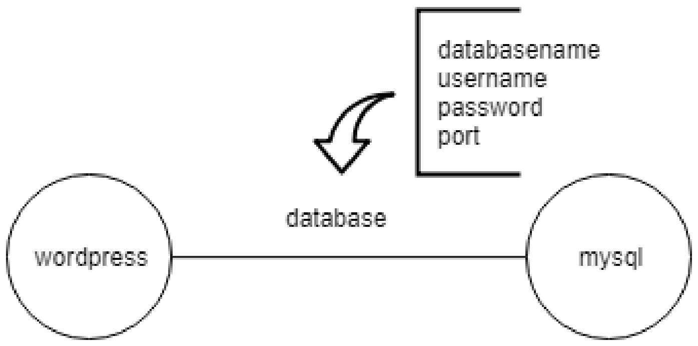


Figure 3.1: Example application model of Wordpress and MySQL

application needs a database to work with. The graph therefore represents two services. The Wordpress application itself and a database technology service (here MySQL) that can provide a database. The relationship between the two applications show a shared entity, in this case this a database which has several attributes such as the databasename, the username, the password and the port number. In other words the necessary details to establish a proper connection to the database.

3.1.2 OASIS TOSCA

Section 2.3 gave a brief introduction of the OASIS TOSCA standard language, explaining concepts such as node templates, relationship templates and topology templates to describe topologies of cloud based web services. Note the similarities between the application model showed before and the OASIS TOSCA standard. This thesis takes the best of two worlds by using the same concepts and visual guidelines of OASIS TOSCA where possible but in a loosely way to keep things as simple and clear as possible. Therefore no YAML-like code will be presented and other details of the standard, for the sake of simplicity, might be omitted.

No better way to clarify this then by presenting an example. Figure 3.2, directly taken from the TOSCA Simple Profile document (section 2.4, page 19), shows an example of a logical diagram meeting the

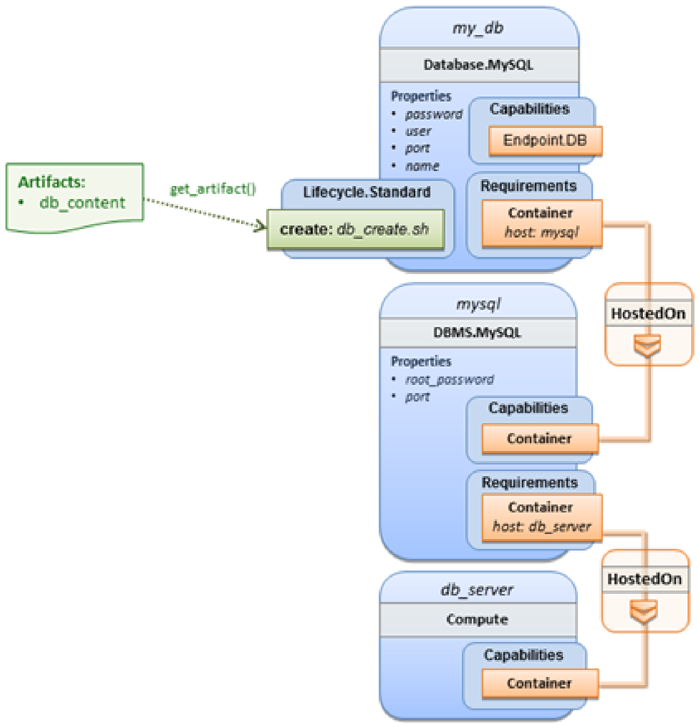


Figure 3.2: Logical diagram example of OASIS TOSCA standard

requirements of the OASIS TOSCA standard. Note the three different nodes, each with a "Capabilities" and optionally "Requirements" section. These characteristics work like Lego pieces, offering and requiring structures to fit together. This means that one node requiring "X" can hook into a node offering (capabilities) "X". In figure 3.2 this is for example the MySQL host (in this case a container). The Database.MySQL node has this in the requirements section whereas the DBMS.MySQL node provides this in the capabilities section.

Figure 3.1 is based on the very same idea and concept. The Wordpress node has a "requires" which is identical to the "capabilities" of the MySQL node. Nodes that indicate the host or DBMS system are left behind for clarity and the attributes of the database connection are put on top of the relation. This leads to smaller and clearer graphs but with the same conceptual idea of the OASIS TOSCA standard.

3.2 Example use case: company X

Before properly defining the generic database concept, an example use case will be examined. With this scenario in mind, the goal and example usage should become clear. Company X decides to invest in the creation of a web shop. The developers creating the online application decided to use two different database technologies. All user-related information of the customers will be stored on a Post-

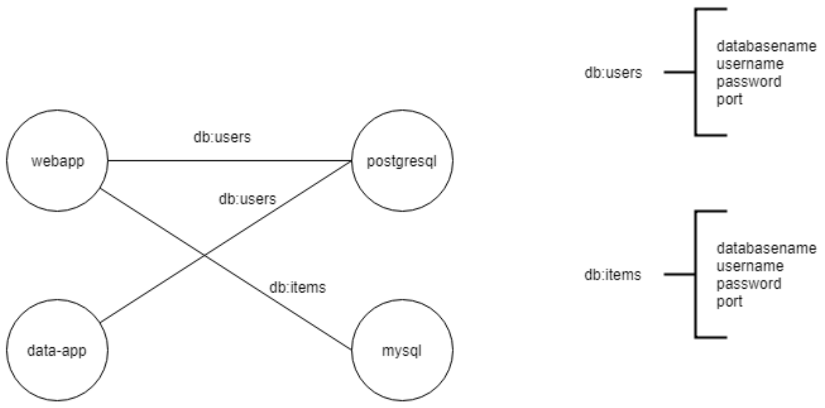


Figure 3.3: Application model of use case: company X

greSQL database and all the available items of the shop will use a MySQL one. Because company X wants to analyse the web shop extensively, they also decide to create an application that will perform statistical analysis. The database containing all user-related information needs to be accessible for this application as well. From now on the web shop webservice will be referred to as "webshop" and the statistical application as "data-app". The two databases are named as "users" and "items" respectively. Figure 3.3 shows a simplified OASIS TOSCA application model of this use case. The graph shows that the (topology) model consists of four nodes and three relationships in this case.

The different steps for setting up this use case could be summarised as follows:

- Creation of the webshop and data-app applications.
- Deployment and configuration of a PostgreSQL and MongoDB server.
- Creation of both the users and items databases.
- Deployment and configuration (including database connection details) of the webshop and data-app applications.

3.3 The generic database

3.3.1 Definition

Section 1.2.1 touched the concept of virtual administrators. The idea of a tool that would automate most manual processes concerning system administration tasks, is the fundamental starting point of the generic database concept. A, rather abstract, definition of the generic database could therefore be:

> “A virtual administrator that handles the operational tasks, such as setting up a database server, the creation of databases and sharing connection details, of a database administrator regardless of the database technology.”

In other words, the generic database would automate all operational steps from the moment a request is made. Once these operational steps are finished the generic database is no longer generic but becomes concrete and holds certain properties such as databasename, username, password and the port number. The name “generic” refers to the database’s ability to offer support for polyglot heterogeneous database technologies. Note the similarities with the “tosca.nodes.Database” definition from the TOSCA standard in the OASIS TOSCA Simple Profile (p208). There are however some key differences between the TOSCA conceptual defined database and the generic database presented here. The TOSCA database assumes it’s “hosted on” a node of the type “RDBMS” illustrating that the database knows at any time what type of database technology is used. This is not the case with the generic database. In addition, the generic database is first considered to be a service with the ability to provide a database. Only after a request, the generic database is considered a similar, concrete, database just like the TOSCA database.

Note that the generic database service, by definition, only functions on the operational level. Just like all graphs (application models) presented here, all nodes and their relationships are representations concerning system administration, deployment, management and monitoring. They do not reflect the workflow or topologies from the application or services themselves. If service A has a direct relation with service B and B with C on an operations level, then it is entirely possible for service A to communicate with C from the applications point of view. This wil become clear at the end of this chapter. More information about this in section 3.7.

3.3.2 Design choices

When designing infrastructures, services or applications, certain choices need to be made. These choices are crucial and often determine the usage, capabilities and limitations of a certain service. The generic database service, as defined in the previous section, is considered to represent either no database (still generic and available to fulfil a request) or one database (concrete). This choice, for an atomic-like structure, came from the idea that all other (possible) definitions were either meaningless or still possible with the generic database as is. Four approaches were examined and are summarised in table 3.2.

The second definition needs a bit of explanation. It sounds complexer then initially intended. One of the first ideas when approaching the generic database concept was to look at the idea of encapsulation and inheritance from the object-oriented programming principles. Database technologies could be put together in a hierarchy as shown on figure 3.4 in the form of a small example. A possible thinking path was then the idea to start at the bottom and add more and more support creating new services, or

Table 3.2: Different (possible) definitions for the generic database concept

	Name	**What**	**Remarks**
Def. 1	The atomic generic database	1 service equals 1 database	This is the chosen definition
Def. 2	Multiple generic databases in a hierarchy of subdivisions	1 service equals 1 database but there is a distinction between services	There is no benefit or reason to use a sql-generic-database over a generic-database
Def. 3	Generic database manager	The idea that a requesting service needs n databases. 1 service equals n databases	Possible, create a new service that uses Def. 1
Def. 4	Global generic database manager	The idea that every database is represented by the service. 1 service equals all databases	Also possible with Def. 1

starting at the top and making sure all necessary features were present. This approach was not further researched as there is no reason why anyone would want to use the NoSQL generic database service for example over the generic database service at the root of the tree.

The (atomic) generic database (from now on generic database) is therefore an intermediate service between a requesting service and a providing database technology service. It acts as a proxy between the two. When designing an application model any requesting service or application in need of a database could therefore connect to the generic database without any constraints. The generic database will take care of all necessary actions such as creating the database and sharing the connection details. The generic database is therefore by definition a virtual administrator.

3.4 Possible situations

The generic database represents one database and one only. This means that if an application needs ten different databases, ten different generic databases will exist in the application model (the graph). This results in the following situations:

- Situation A: A 1-on-1 relation between requester node and generic database node. This is the most trivial graph. In this case one service needs one database.
- Situation B: A n-on-1 relation between requester node and generic database node. In this case multiple requester nodes want a connection to one and the same generic database. This means that 1 requesting service did a request for a database and n other requesting services want to connect to this very database. The connection details are therefore the same.

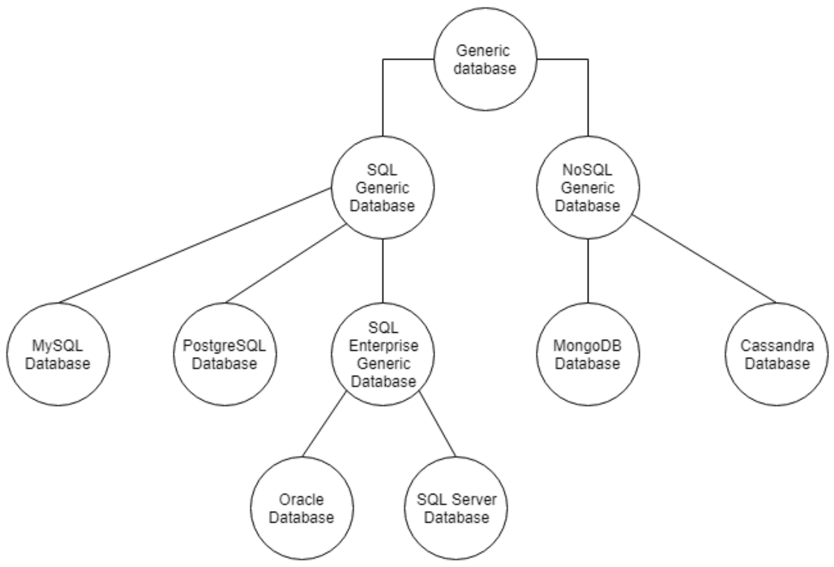


Figure 3.4: Example hierarchy or categorization of database technologies

- Situation C: A 1-on-n relation between requester node and generic database node. This situation is reached when 1 requesting service needs multiple databases.
- Situation D: A n-on-n relation between requester node and generic database node. This is a combination of the previous 2 situations. A requesting service needs multiple databases and these databases are also used by various other services.

3.5 Use case revisited

The use case of company X can be remodelled with generic databases. This use case is an example of situation D. The users database needs to be accessible for two services (the webshop and data-app applications) and the webshop requires two databases (users and items). Creating the application model with generic databases results in a graph as show in figure 3.5.

3.6 The generic database under the hood

Thanks to the use case the inner workings of the generic database should be given more form. The next few paragraphs will use diagrams to illustrate what the generic database service should do and how it interacts with the other components.

Figures 3.6 and 3.7 show BPMN (= Business Process Model and Notation) models. The three lakes

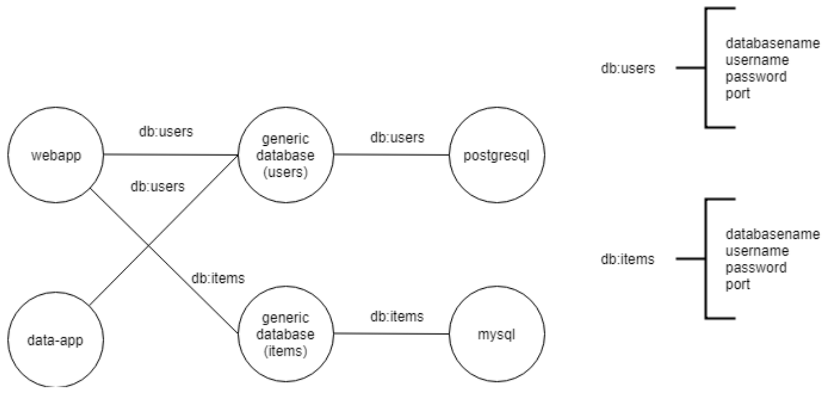


Figure 3.5: Application model of use case: company X with generic databases

represent the three different services. Figure 3.8 is the simplest model but assumes that the database technology service is up and running and ready for use while figure 3.9 also takes this uncertainty into account. In other words, in the model of figure 3.7 the generic database also manages the database technology whereas in figure 3.6 the generic database only performs requests. Both are implementations of the generic database concept but with different preconditions and therefore functionalities.

Figures 3.8 and REF 3.9 show a similar workflow with sequence diagrams. The same distinction is made as before. First with the simplest possible model with figure REF and then a more complete diagram with figure REF.

3.7 Remarks, discussion & limitations

This chapter introduced the idea of a generic database virtual administrator in a descriptive manner. The idea of having a virtual administrator taking care of all operational tasks for a service (or another virtual administrator) offers workflows that are easy to use. There are however some important remarks and caveats concerning the generic database concept.

It is important to realise that the generic database concept only works on the operational level of the infrastructure. This chapter used application models and graphs to illustrate workflows and communication models. They only represent however the “operations” side of applications, meaning that the generic database is not relevant in the so-called application topology. Once a requesting node (such as the webshop application in the use case) receives the connection details of a database, all interactions are directly with the database and not through the generic database (in contrast of what might seem intuitive when looking at the graphs). There is a distinct difference between the operations perspective and the application workflow perspective. A developer or an application don’t need knowledge about the generic developer whatsoever. No special actions or setups should be required as the generic

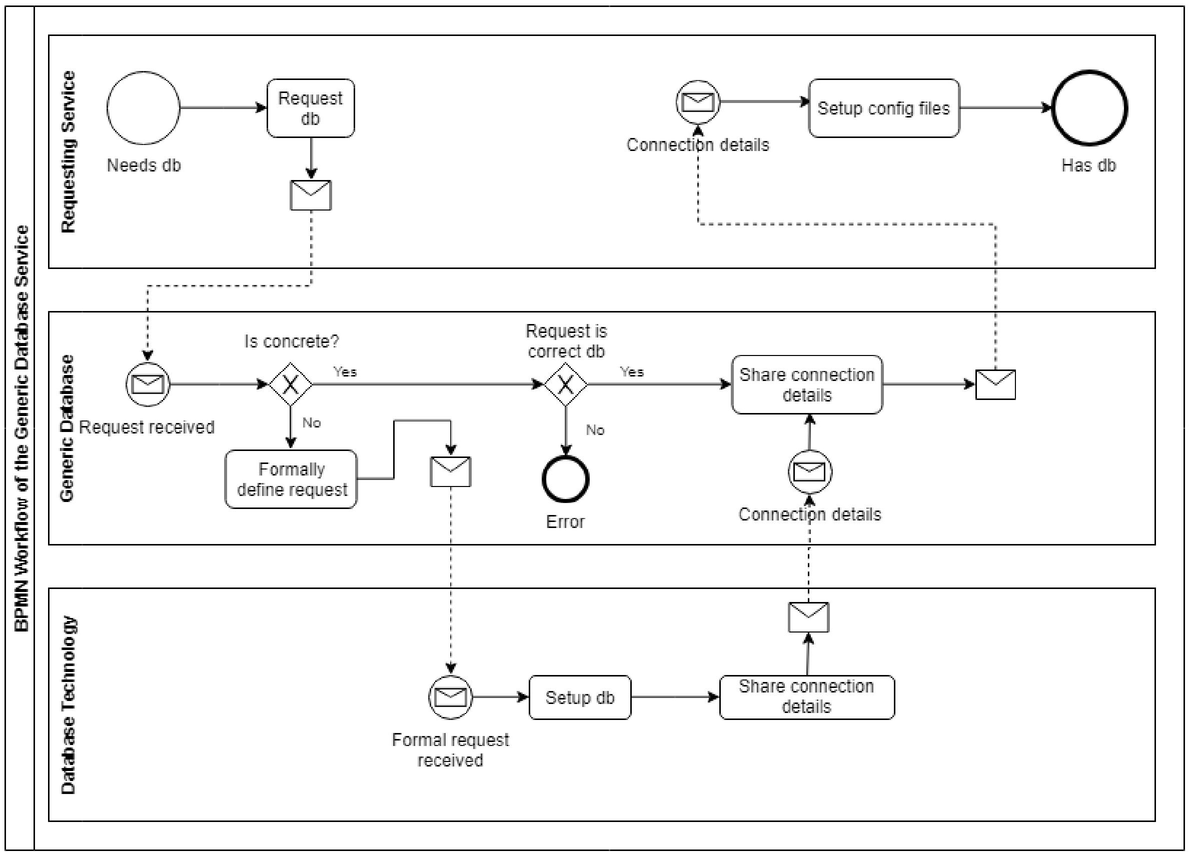


Figure 3.6: BPMN diagram of the generic database service workflow

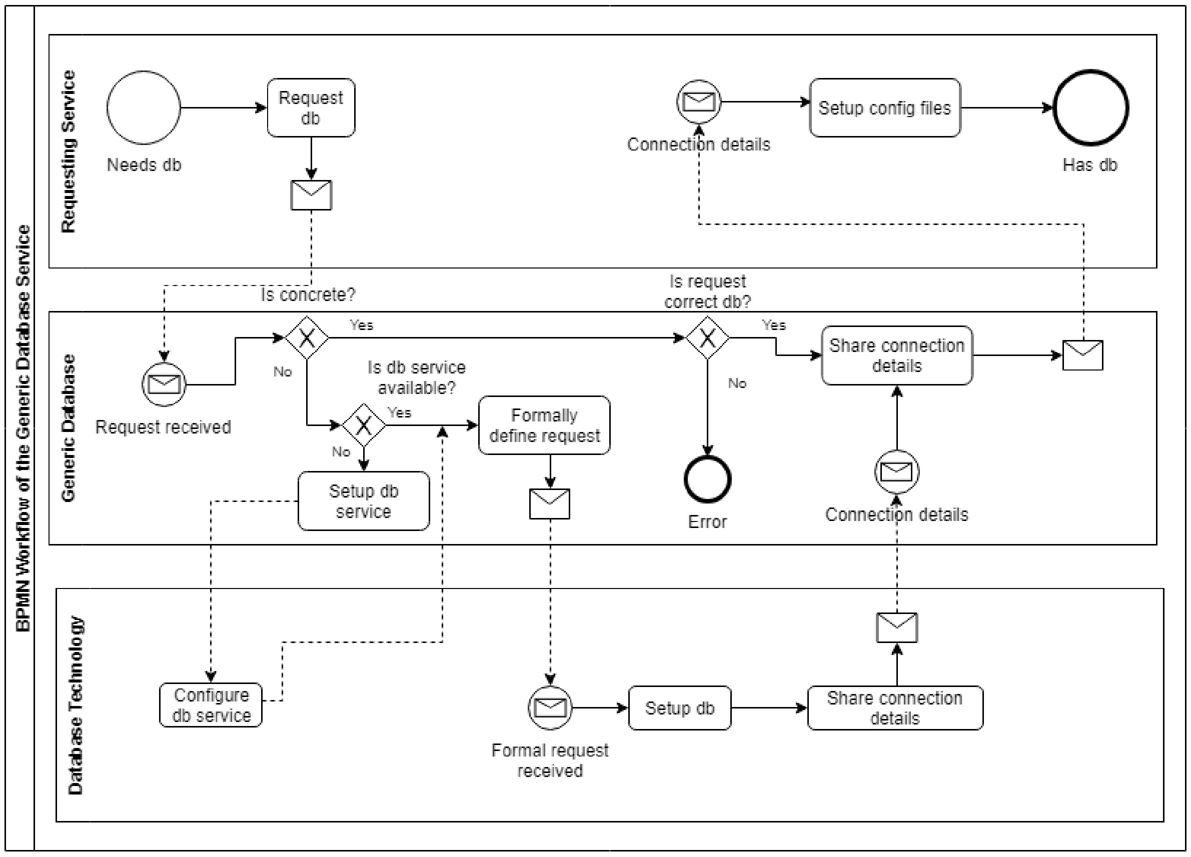


Figure 3.7: BPMN diagram of a more complete generic database service workflow

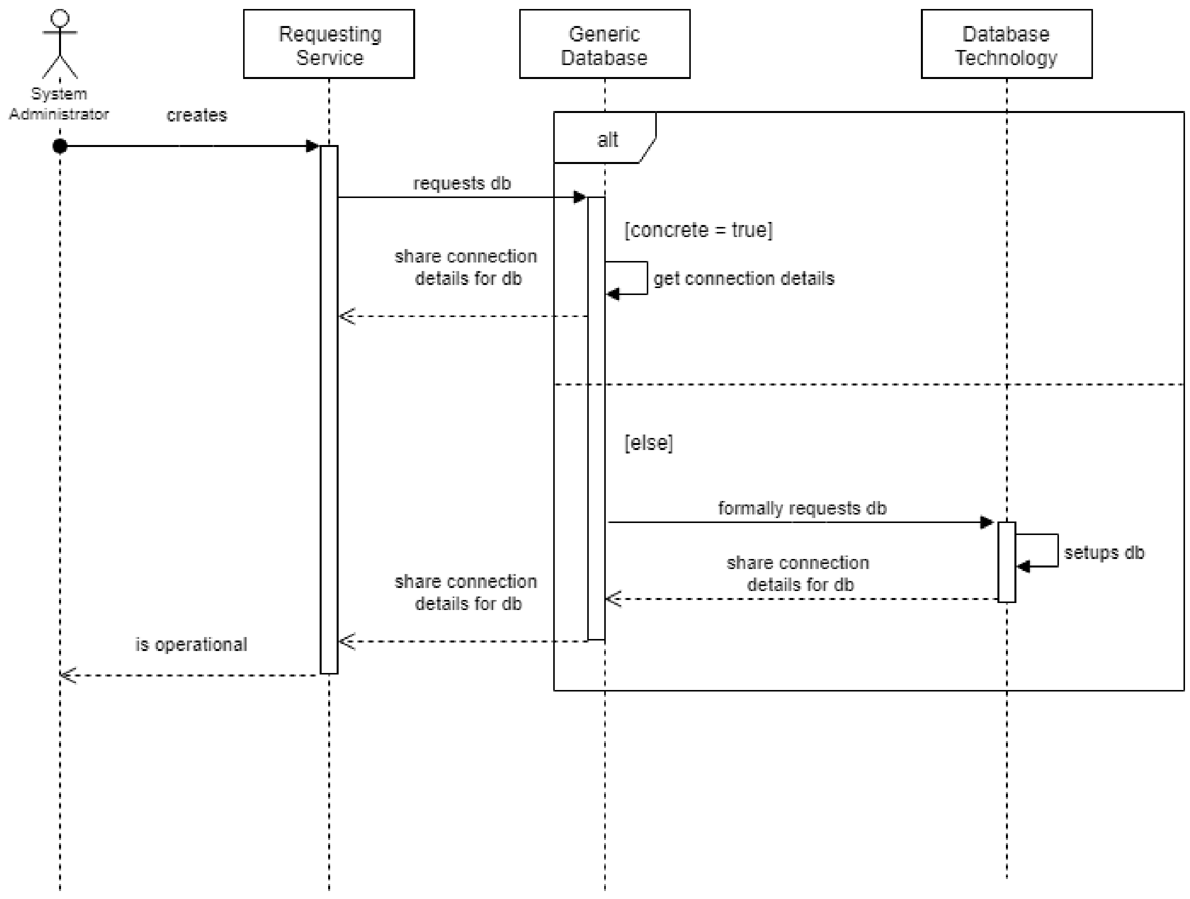


Figure 3.8: Sequence diagram of (basic) generic database service workflow

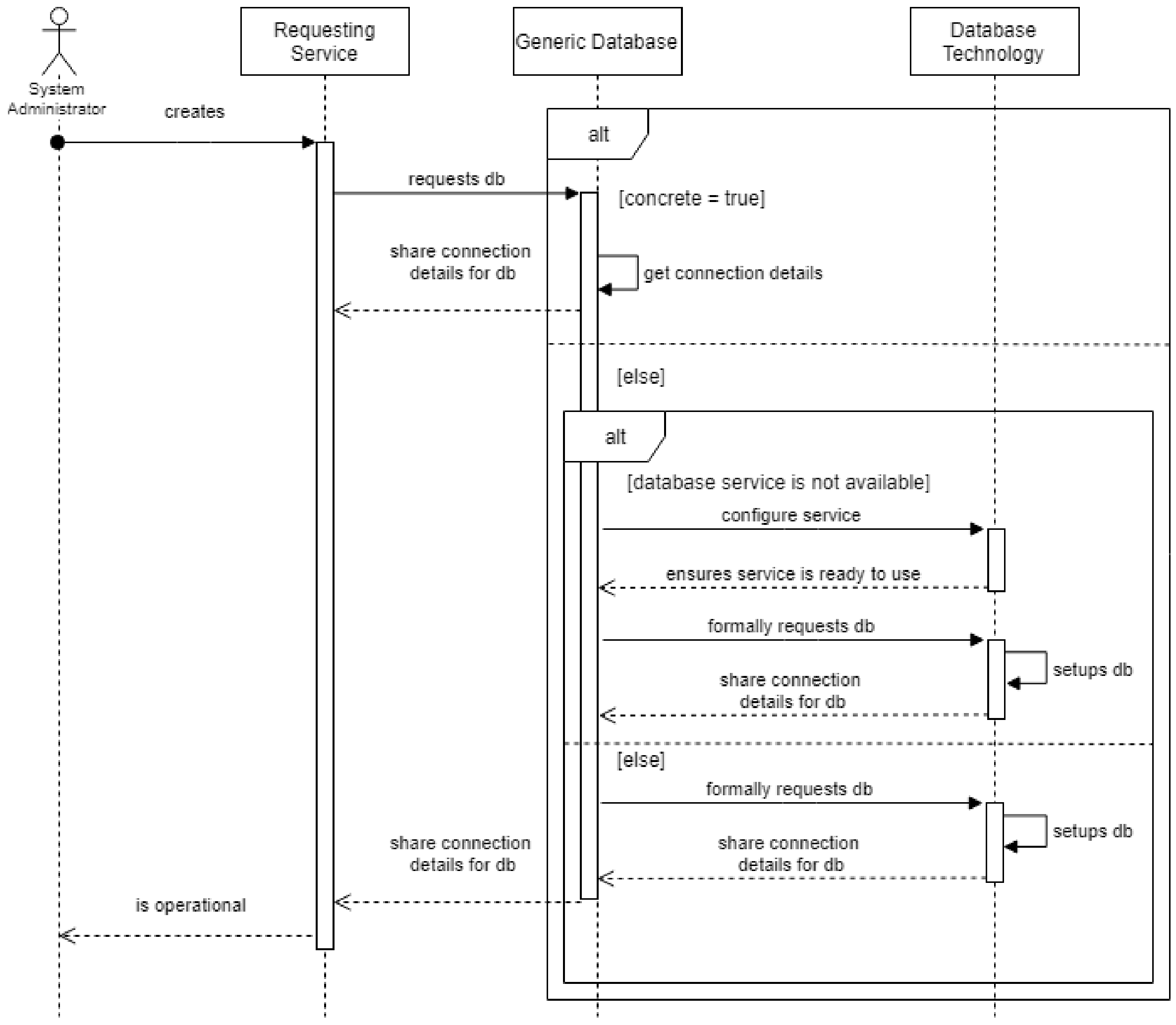


Figure 3.9: Sequence diagram of a more complete generic database service workflow

database service is not present in the workflow of the applications using it. This is an interesting characteristic but it might be contra intuitive and weird at first. Once again there is a distinction between the operations side and the inner workings of the applications.

The formal definition of the generic database does not describe how the setup and underlying relation between database technologies is done. If a generic database represents a MySQL database after a request, the generic database needs access to or communication with a MySQL service. In what way this is achieved is free to choose. Maybe the generic database service and MySQL are both hosted on the same machines, maybe in different containers or maybe they are two completely different and independent machines. A fully fledged generic database service would allow all these possibilities.

A requesting service still requests a database for a specific technology. This means that the virtual administrator of this requesting application needs the knowledge of that specific technology. An interesting feature would therefore be a service that would help teams choose in deciding what type of database technology would be best fit for their use case (or application). Such a feature would look at certain requirements or needs and determine what database technology should be used. This could be implemented on top of the generic database service or as a service that would communicate (request) the generic database. With such a service the database specific knowledge (or requirements) of a service would be completely gone and all would be automated by virtual administrators. This lies however beyond the scope of this research.

The generic database as proposed here, deliberately doesn't limit features of the generic database. In the section about design choices it is stated that the generic database at least needs a way of requesting databases or derivatives (for example a collection in MongoDB) and its properties such as hostname, databasename, password and port. But what about schema's (or for example keyspaces for Cassandra), views, triggers or other database defined elements? In addition, does the generic database need to offer support for SQL-queries? A way to offer backup support? All these questions reside in a grey zone. While useful, these features not necessarily have their place in the generic database service. Therefore, it would be interesting to have an additional service (node in the graph) with a different relation or connection (vertex in the graph) that would offer these features. This way the generic database is used for the deployment part of the database and the other service is responsible for operational tasks on the database. Depending on the use case teams could opt for implementing them together.

Finally, a big remark about the performance and expansion caused by the generic database service. By introducing the generic database nodes onto the application model, the graph itself became bigger. More services, maybe more machines, and therefore more resources are needed to make it work. If a few applications need ten databases then ten generic database services will need to be deployed, increasing the complexity of the application model. This may or may not be a reason to use the generic database node or not. A user should therefore reflect and think about the use case. The virtual administrator is easy to use and can model databases in the application model, but comes at the cost of more resources and a more complex model.

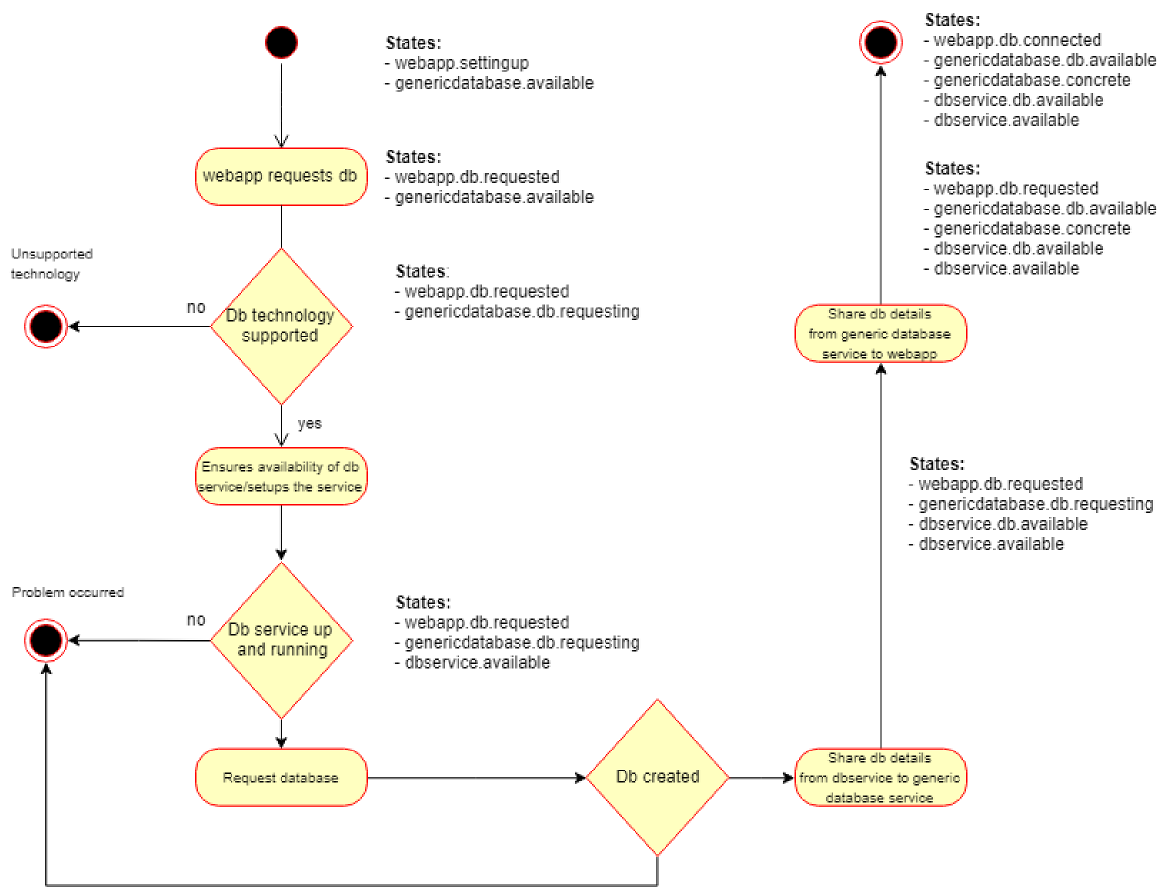


Figure 3.10: Flowchart with states for webapp and generic database interactions

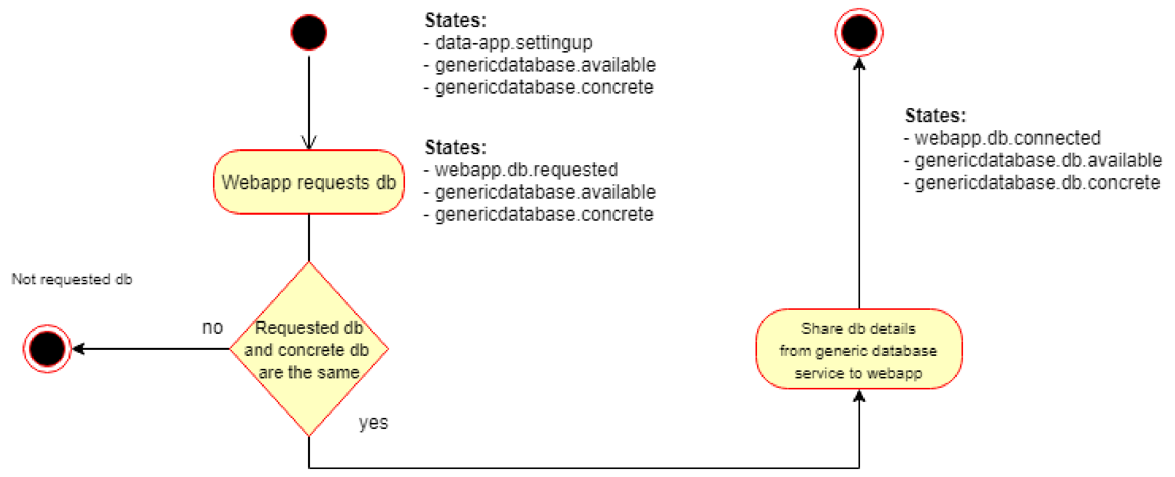

States:
- data-app.settingup
- genericdatabase.available
- genericdatabase.concrete
Webapp requests db
States:
- webapp.db.requested
- genericdatabase.available
- genericdatabase.concrete
Not requested db
no
Requested db and concrete db are the same
yes
Share db details from generic database service to webapp
States:
- webapp.db.connected
- genericdatabase.db.available
- genericdatabase.db.concrete

Figure 3.11: Flowchart with states for data-app and generic database interactions

4 Technical implementation

A technical implementation and tangible product is the core of this chapter. The concept of the generic database as described in chapter 3 (REF) will be implemented in the service orchestration tool Juju (examined in chapter 2 (REF)). Not only the generic database idea but the whole structure of the previous chapter is recycled to map all conceptual steps towards an implementation. First the different terms and visualisations of the conceptual specifications will be translated to their Juju counterparts. Next, the use case of company X will be shortly described without the use of the generic database concept. Next an example implementation will explain how the theoretical ideas can be constructed in Juju along with certain design choices. The same use case will be revisited once again with the use of the constructed generic database. This chapter concludes with some limitations, caveats and future improvements specific for the created service.

4.1 Juju specific terms

This chapter will use the terms as defined by Juju. Refer to chapter 2 REF for an in-depth explanation. Table REF gives an overview of the different things relevant in this chapter. TABLE

4.2 Example use case: company X

Looking back at the use case (webshop in need of 2 databases, one of which is also used by another data analysis application) from section REF, the following approach could be used by a Juju user:

1. Determining what database technologies is needed.
2. Creation of the webshop and data analysis app.
3. Creating 2 charms: one for the webshop and one for the data analysis app.
4. Deployment of (existing) PostgreSQL and MongoDB charms from the charm store.
5. Deployment of the (self-written) charms for the webshop and data analysis app.
6. Adding relations accordingly.
7. Manual intervention to make sure multiple charms can access one and the same database.

The final model in Juju would look something like figure REF. Note the final step. It is, at design level in Juju, impossible for multiple charms to access the same database. This is a result of the implementation of the existing interface layers and database technology charms. They are configured in such a way that new incoming relations create new databases. The whole goal of an easy-to-use application modelling tool becomes inadequate as manual steps are still required. The generic database charm (generic-database-charm from now on) to the rescue.

4.3 The generic-database-charm

4.3.1 Design Choices

When creating a charm, a clear concept of the service is needed for optimal choices. One of the first questions that rises is what type of charm the generic-database-charm needs to be. Juju offers support for regular charms or subordinates. The generic database represents a database. At first glance a subordinate seems the suitable choice as we want to waste as little resources as possible. There are however reasons why a regular charm is more interesting (and therefore chosen in this implementation):

- Subordinates only exist for the lifetime of their principal service (this is a regular charm in whose container the subordinate service would run). This means that it is impossible to model the database without a requesting charm. In the use case of company X this would mean that no generic-database-charm can be created without the webapplication and/or data analysis app. In addition, the generic-database-charm would be gone if the principal service would be destroyed.

- The use of subordinates would also result in another interesting feature being lost. Regular charms can be on "stand-by". This means that they can be deployed and be ready for use beforehand with custom configurations where needed. Note that the implementation in this research therefore made the choice to create a regular charm. There are however use cases where a subordinate would work just as good and would be less resource-demanding.

4.3.2 Structure and workflow

The basic idea of the generic database workflow is ... in Juju this is implemented through flags.

4.4 Use case revisited

4.5 Remarks, limitations & future work

The generic-database-charm offers an easy to use way to request databases for charm authors. Through simply adding a relation to the generic database, adding a decorator and requesting the database, the necessary knowledge is reduced to the minimum and the request process becomes as simple as a plain question. There are however some remarks that can be made. First, all remarks as presented in REF still apply as these were remarks on the concept. For instance, the additional resources of the generic-database-charm (a complete new machine for the representation of a database is a harsh thing to do). Furthermore, a charm author who has the knowledge of setting up and connecting databases using database technology charms wouldn't directly benefit from the generic-database-charm. The manual steps necessary when multiple applications need connection to the same database becomes however obsolete with the generic-database-charm. On the same note, non-Juju users might not benefit from the generic-database-charm as the decision to use the Juju tool is most likely not determined by this charm or use case.

The generic-database-charm as implemented in this chapter isn't the one and only solution and should be looked at as a minimal working example. It illustrates the implementation of the generic database concept and workings through a use case. A fully-fledged service, usable in more situations, would need extra features. The following list could serve as a to-do list for future work on this charm:

- Add support for more database technologies.
- Add support for at-demand-time deployment. Right now, the generic-database-charm assumes that the database technology charm is deployed and ready for use. This feature would allow the

generic-database-charm to spin up the required database technology charm accordingly. Either on the same machine or as a complete standalone service.

- Add (global) support for more request functionalities. Things like custom database names, usernames, passwords are not always accessible by every database technology charm or interface layer. A uniform interface that would allow this feature for all supported database technologies, would be a nice thing to have. This would also lead towards a more optimal request workflow as there is no distinction anymore for the interface layer between "requesting" (here setting up) a database and connecting to an already concrete generic database. If every request comes with a databasename then the charm needs to "setup" (start the process of creating) the database if the generic-database-charm is not concrete. In the other scenario the charm needs to "connect" (share the connection details.
- A new service (charm) functioning as a generic database manager (as introduced in REF) would also be a welcoming feature. A requesting service could then ask for multiple databases in one go. The manager would use the generic-database-charms to make it happen.

Example of a listing for now

```python
from charms.reactive import set_flag, clear_flag, when
from charms.reactive.helpers import any_file_changed
from charmhelpers.core import templating, hookenv

@when('db.database.available', 'config.set.admin-pass')
def render_config(pgsql):
    templating.render('app-config.j2', '/etc/app.conf', {
        'db_conn': pgsql.connection_string(),
        'admin_pass': hookenv.config('admin-pass'),
    })
    if any_file_changed(['/etc/app.conf']):
        set_flag('myapp.restart')

@when('myapp.restart')
def restart_service():
    hookenv.service_restart('myapp')
    clear_flag('myapp.restart')
```

Listing 6: Example of a listing

5
Results and Discussion

5.1 Functional level

5.2 Technical implementation

5.3 Discussion

5.4 Future work

6
Conclusion

Bibliography

[1] S. Overflow, "Stack Overflow Developer Survey 2018," 2018. [Online]. Available: https://insights.stackoverflow.com/survey/2018/

[2] P. Simoens, "System Design Course," 2017, university Ghent.

[3] M. Hamdaqa and L. Tahvildari, "The (5+1) architectural view model for cloud applications," in *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '14. Riverton, NJ, USA: IBM Corp., 2014, pp. 46–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=2735522.2735530

[4] M. Abramow, "How DevOps and Agile Development Can Drive Digital Transformation," 2017. [Online]. Available: http://www.oracle.com/us/corporate/profit/big-ideas/072417-mabramow-3839318.html

[5] V. d. C. Guerra, E. Segeti, F. Hino, F. Kfouri, L. F. S. Mialaret, L. A. V. Dias, and A. M. d. Cunha, "Interdisciplinarity and agile development: A case study on graduate courses," in *Proceedings of the 2014 11th International Conference on Information Technology: New Generations*, ser. ITNG '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 622–623. [Online]. Available: http://dx.doi.org/10.1109/ITNG.2014.49

[6] R. A. Rodrigues, L. A. L. Filho, G. S. Gonçalves, L. F. S. Mialaret, A. M. da Cunha, and L. A. V. Dias, "Integrating nosql, relational database, and the hadoop ecosystem in an interdisciplinary project involving big data and credit card transactions," in *Information Technology - New Generations*, S. Latifi, Ed. Cham: Springer International Publishing, 2018, pp. 443–451.

[7] G. Kim, J. Humble, P. Debois, and J. Willis, *The DevOps Handbook, How to create world-class agility, reliability, & security in technology organizations*, 1st ed. IT Revolution Press, 2016.

[8] M. Sebrechts, G. V. Seghbroeck, T. Wauters, B. Volckaert, and F. D. Turck, "Orchestrator conversation: Distributed management of cloud applications," 2018.

[9] M. S. ..., "Beyond generic lifecycles," 2018.

[10] C. Ltd, "What is Juju?" 2017. [Online]. Available: https://jujucharms.com/docs/stable/about-juju

[11] E. F. Codd, “A relational model of data for large shared data banks,” *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970. [Online]. Available: http://doi.acm.org/10.1145/362384.362685

[12] ——, *The Relational Model for Database Management: Version 2.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[13] T. Vanhove, “Management of polyglot persistent environments for low latency data access in big data,” Ph.D. dissertation, Ghent University, 2018.

Appendices

Appendix A - Juju Roadmap

TODO

This will have an overview of interesting commands and it will be a getting started with juju explaining all steps and an in depth guide.