Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Design and Implementation of an SAP HANA Cloud Configuration Handler

Cora Glaß
Matrikelnummer: 5200206
cora.glass@fu-berlin.de

Erstprüfer : Barry Linnert
Zweitprüfer: Prof. Dr.-Ing. Jochen Schiller

Berlin, October 11, 2020

Abstract

The SAP HANA is a highly configurable, in-memory database that just recently was migrated into the cloud. The resulting product is called SAP HANA Cloud which is a database-as-a-service (DbaaS). The used cloud infrastructures are Kubernetes clusters that are managed by SAP. The manual configuration of a SAP HANA risks the possibility of human errors or security issues. An undesired consequence of human errors might be service outages. The manual configuration also generates a workload that could be decreased by automating the configuration management of the SAP HANA. Therefore, this thesis deals with the design and the implementation of a microservice, named SAP HANA Cloud configuration handler. The handler is a prototype of a service that automates the SAP HANA configuration process. The greatest finding of this thesis is the following. Before designing a service that shall run in a complex environment that is supported and improved by multiple people, it is wise to spend the time to analyse the environment. This reveals dependencies that must be fulfilled as well as synergies that might be useful.

Zusammenfassung

SAP HANA ist eine hoch konfigurierbare In-Memory-Datenbank, welche vor Kurzem in die Cloud migriert wurde. Das daraus resultierende Produkt ist bekannt als SAP HANA Cloud, eine Database-as-a-Service (DbaaS). Die Cloud Infrastrukture basiert auf Kubernetes Clustern. Diese werden innerhalb der SAP verwaltet. Das manuelle Konfigurieren von SAP HANAs erhöht das Risiko von Service-Ausfällen. Des Weiteren ist die manuelle Konfiguration menschlicher Arbeitsaufwand, welcher durch die Automation einiger Schritte verringert werden kann. Daher beschäftigt sich diese Arbeit mit dem Design und der Implementation eines Microservices genannt SAP HANA Configuration Handler. Dieser Handler ist ein Prototyp eines Services, welcher die SAP HANA Konfiguration automatisiert. Der bedeutenste Fund der Arbeit lautet wie folgt. Bevor man einen Service entwirft, der in eine komplexe Umgebung integriert werden soll (welche von mehreren Personen unterstützt und verbessert wird), ist es ratsam, sich die Zeit zu nehmen die existierende Umgebung zu analysieren. Dadurch können Abhängigkeiten identifiziert werden sowie Synergien, welche eventuell zum Vorteil des Services genutzt werden können.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

October 11, 2020

Cora Glaß

Contents

1 Introduction

Not long ago, SAP released its new product SAP HANA Cloud[1]. In general, SAP HANA is an in-memory database management system that provides its users with fast access, querying and processing. SAP HANA has a large number of customers. One of them is Charité[2], a renowned university hospital in Berlin. With the help of SAP HANA, they built a scalable research platform[3]. SAP HANA Cloud is SAP's highly scalable cloud solution of SAP HANA and is currently available on Amazon Web Services (AWS) and Microsoft Azure. In the following, an SAP HANA Cloud landscape (in short: landscape) describes the infrastructure on which the SAP HANA Cloud instances (in the following called: HANA instance or instance) are running on. A HANA instance provides the user with many configuration parameters that can be set to satisfy the needs of customers having different requirements. The user and customer can be the same person. When operating a landscape with a lot of such highly configurable instances, it is necessary to maintain a general overview on how each instance or group of instances is configured. For example, the configuration of a HANA instance could be changed as a temporary workaround for a bug. This might for example avoid an outage. Usually, a bug would affect multiple HANA instances. Therefore, a group of multiple instances would have to be configured. In this case, we must know which instances belong to this group to be able to execute a rollback of their configurations after the bug is fixed. If the total amount of instances is rather low, they can be managed manually. This also means a subset of employees must be responsible for the configurations and have to access a productive landscape. This does introduce the possibility of human errors as well as the workload that could have been decreased or replaced by automation. An increasing amount of instances leads to the situation that the configuration handling gets more and more time-consuming and demands a continuous effort as e.g. maintaining an overview of all instances gets more complex. To reduce such manual workload, it can be efficient to use the site reliability engineering (SRE) approach of automating operations related work in form of e.g. a service [1].

Therefore, this bachelor thesis deals with the design and the implementation of a prototype of such a service, called SAP HANA Cloud configuration handler. The goal of my bachelor thesis is to improve the configuration management of HANA instances by introducing the following functionalities in form of the handler:

- secure configuration of instances without accessing the system manually
- configuration of multiple instances via one request
- on-demand generation of a configuration state overview of all HANA instances
- creation and application of configuration profiles that contain information of parameters and the value they should get set to

[1] https://www.sap.com/products/hana/cloud.html

[2] https://www.charite.de/en/

[3] For more information visit https://www.sap.com/documents/2020/01/7ef991e0-7e7d-0010-87a3-c30de2ffd8ff.html

The prototype shall only focus on the configuration of the HANAs and neither manage the provision and deletion of HANAs.

2 The Initial Architecture

This section gives an insight into the infrastructure on which HANA Cloud instances are running as well as on HANAs itself.

2.1 The Landscape

HANA instances are running in Kubernetes[4] clusters that are maintained by Gardener[5], an SAP product that creates, updates, scales and deletes Kubernetes clusters on-demand. These Kubernetes clusters are therefore SAP HANA Cloud landscapes. To understand the purpose of Kubernetes, we should briefly touch upon the topic of containerization. Containerization describes the principle of binding an application together with all its necessary resources to satisfy all dependencies. The structure is called container and can run on different computer environments. For example, if Docker[6] was used to build the container, more precisely the image which was used to create the container, it can run on different environments if these have the Docker engine installed. Kubernetes is a platform to orchestrate these containers. It binds one or a group of containers into a pod. A pod is a Kubernetes resource. In this context, a HANA instance is a pod (in the following also called HANA pod) and the HANA is a container that is running within the pod, as shown in Figure 2. The objective of Kubernetes clusters is to manage and monitor these pods that are running on cluster nodes. For a simplified overview see Figure 1[7]. One main goal of Kubernetes clusters is to be a mostly self-healing system. This means if a pod is not behaving as expected, stops to run, or if a cluster node goes down, the functionality of Kubernetes is to execute predefined routines, restart the pod, alert the operator of the landscape, or reschedule the pod upon another cluster node. Kubernetes provides its users with multiple resource types besides the pod. These structures are all part of the Kubernetes ecosystem. In general, Kubernetes resource types are each an abstraction of a configuration aspect.

The resource types are for example associated with the cluster, a pod, or used for security and/or network purposes. Resources associated with the cluster can define e.g. the number of nodes included in the cluster or the OS and OS version in use. Resources referring to the pod can declare which containers belong in one pod, how many replicas shall be available, and define the healthy behaviour of a container. Other resources are used to e.g. configure access permissions or for the management of sensitive data. It is also possible to create resource types yourself. They are called custom resources. It is possible to add labels to every kind of Kubernetes resource. They are saved in the metadata of the Kubernetes resource. Using a Kubernetes API client, such as the command-line tool kubectl[8], it is possible to select e.g. all pods that contain a specific label with a specific value.

[4] https://kubernetes.io/

[5] https://gardener.cloud/

[6] https://www.docker.com/

[7] For information about the Kubernetes node components see https://kubernetes.io/docs/concepts/overview/components/#node-components

[8] https://kubernetes.io/docs/reference/kubectl/

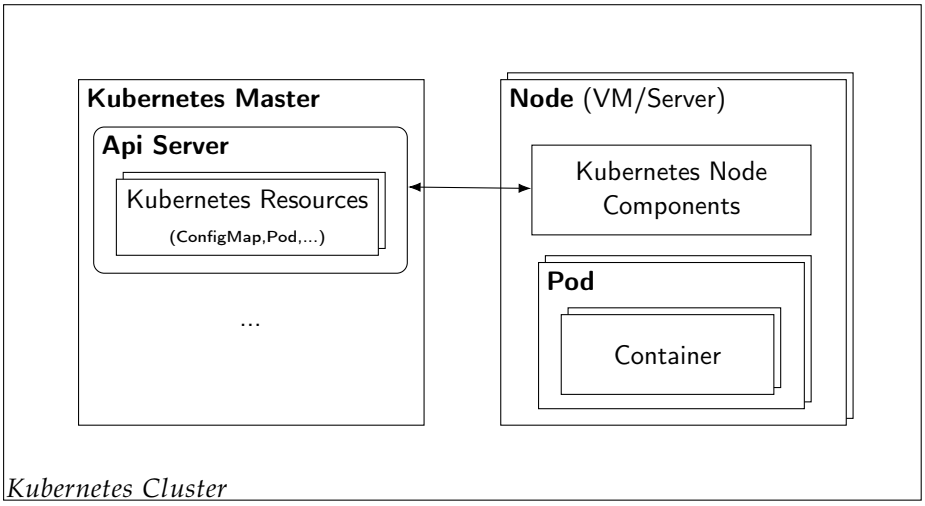

Figure 1: Simplified Overview of Kubernetes Cluster Components

2.2 The HANA

As mentioned before, HANA instances are pods. These pods contain each a container which is the actual HANA. A HANA consist of one system database and at least one tenant database (in short: SystemDB and TenantDB). The HANAs that are running in HANA Cloud instances have a general design, as shown in Figure 2. They consist of a SystemDB and exactly one TenantDB.

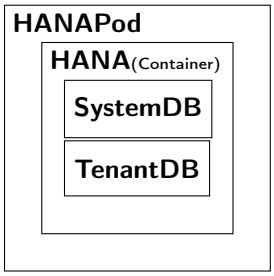

Figure 2: HANA Cloud Instance Architecture

The SystemDB is used for the system administration and the TenantDB is the database that is used by customers to save and manage their data. Therefore, customers have only access to the TenantDB of the HANA. SAP has only access to the TenantDB until the HANA is fully provisioned, and keeps permanently access to the SystemDB. In the context of this thesis, configuring a HANA means to adjust parameters of the SystemDB to either configure the system itself or the TenantDB.

The SystemDB contains different layers which make it possible to configure parameters system-specific or database-specific, thus configure the whole system or only the TenantDB. The layers of the SystemDB are called DEFAULT, SYSTEM, DATABASE and HOST. In case of multiple-host systems, the HOST layer is used to configure multiple hosts. The HANAs that are used in the HANA Cloud instances are no

multiple-host systems. Therefore, this layer will be not further discussed in this thesis.

The DEFAULT layer contains the default values of all available parameters. This would be very practical to use to roll back the configuration of a HANA to their default configuration. However, in Section 5.8 it is explained why these default values cannot be blindly used in our case.

The SYSTEM layer contains system-specific configuration parameters, which means the configuration of a parameter of this layer configures the whole system including the TenantDBs. If a parameter is not explicitly configured, the default value is set.

The DATABASE layer contains the database-specific parameters. This layer is also available in the TenantDB. Thanks to this layer, the SystemDB can configure the TenantDB. If a parameter is not explicitly configured, the value of the SYSTEM layer is applied.

As discussed, the configuration parameters are saved in the SystemDB and also partly in the TenantDB. Additionally, they are also saved in INI files that are located in the filesystem of the HANA container. The INI files are plain text files that are holding the parameters and values. The configuration can also be executed by adjusting the INI files and triggering a configuration of the SystemDB via a separate tool that is installed in the HANA.

The configuration parameters all follow a basic structure:
$INI - filename, layer, section, key : value$

3 Related Work

Kubernetes resources can be configured via the standard Kubernetes API. There are also tools, such as Helm[9] and kustomize[10], available to improve the configuration handling of Kubernetes resources. The parameters that I want to configure in my bachelor thesis are not part of the Kubernetes architecture, but are part of the HANA container. These parameters are located in the INI files that exist in the filesystem of the HANA container. After adjusting these files, the database must be reconfigured. The Kubernetes platform does not provide a service to specifically configure the HANA database, as it is not the purpose of Kubernetes to configure the resources and services within a pod, but at most to check if they are satisfying the predefined expected behaviour.

Ansible[11] is a software tool by Red Hat. It can e.g. provision, deploy, configure or orchestrate resources, such as clusters or containers. Ansible holds the necessary modules to easily build a service that can configure databases such as Mongodb, Mysql or Postgresql. However, modules for configuring HANA are not available yet. The usage of Ansible in the company landscapes would need some set-up work. Also, the module for the configuration of the HANA would have to be implemented. Moreover, Ansible would be a new infrastructure tool in the SAP HANA Cloud landscape. Implementing the service via a microservice approach is the common solution in the SAP HANA Cloud landscape and would require less integration effort compared to using Ansible.

There are two solutions provided by SAP to configure HANAs , SAP HANA Cockpit[12] and SAP HANA Studio[13]. SAP HANA Studio is deprecated, which means it is not any more explicitly supported and will not be improved any more. SAP HANA Cockpit was build to replace it. Both let the user request reconfigurations via a user interface. These tools are not only available to the employees, but also to the end-users of the HANA. The users must hold the right permissions profile to be able to request anything. In SAP HANA Cockpit it is possible to create configurations templates. The selection of parameters for the templates are limited. The templates can only contain parameters referring to one database layer. The SAP HANA Cockpit can only apply the template to one HANA at a time. An option to change an already existing template is not available. The configuration profile concept that shall be introduced with the SAP HANA Cloud configuration handler has a similar approach. However, the profiles shall provide the option so hold very different parameters from different layers. The handler shall also be able to apply the profiles to multiple HANAs per request. Furthermore, the adjusting of a profile must be possible and shall also trigger the reconfiguration for HANAs that are using this profile. The SAP HANA Studio does not support configuration templates or profiles, but it has the option to reset a

[9] https://helm.sh/

[10] https://kustomize.io/

[11] https://www.ansible.com/

[12] https://help.sap.com/viewer/product/SAP_HANA_COCKPIT/2.12.0.0/en-US

[13] https://help.sap.com/viewer/a2a49126a5c546a9864aae22c05c3d0e/2.0.00/en-US/c831c3bbbb571014901199718bf7edc5.html

parameter. Unfortunately, the reset mechanism resets the configuration parameters to the default values that are defined in the DEFAULT layer of the SystemDB. In Section 5.8 it is explained why this method is not applicable in the SAP HANA Cloud landscapes. In conclusion, both solutions provide the basic functionality to configure the HANA on-demand. Both solutions hold some interesting features. Still, for the specific use case that my prototype shall support there are yet some features missing in both solutions. The tools are available to external users. The handler will be a tool that shall only be used within the company by the operations team and a selection of developers. Furthermore, SAP HANA Studio is not even supported any more.

Multiple other database configuration tools are available that are not provided by SAP. The difference between them and the planned handler is that they do not support HANA as a database type. Also, the handler will be a service that is running in the same system as the databases. Most solutions are running on local machines or in separate systems.

4 Workflow

This section describes the workflow of designing and implementing the SAP HANA Cloud configuration handler. I started with defining the quality criteria that I wanted to focus on.

A software product should have a certain quality. To evaluate the quality of a software product meaningfully, evaluation criteria must be determined. These could be for example extracted from the requirements that are stated by the customers. The configuration handler is a service that is supposed to be used by cloud software developers as well as the operations team of SAP. This means the customers, and also the users, are employees of the SAP HANA Cloud Development department or of the SAP Cloud Operations unit. Therefore, I expect that the users have a background in programming or that they are at least familiar with using a command-line tool. I am working in a team that is part of the SAP HANA Cloud Development department. The team is implementing and maintaining the SAP HANA Cloud landscapes. The team already implemented services that are only used by the developers and the operations team of our company. Therefore, I used my own experience and the knowledge about the aspects that my team focused on while implementing those services. I divided them into criteria. This resulted in the following ones:

Usability
Since all users of this handler are expected to be at least familiar with the usage of a command-line tool, this criteria does not refer to user-friendly web interfaces. The usability can be evaluated by viewing how easy it is to use and to which degree the introduction of the service shortens and simplifies the workload of the developers and operations team in terms of the HANA configuration support.

Performance
Clearly, a good performance is always desired. The configuration is not a critical real-time application. Still, if HANAs must be reconfigured to avoid them being non-functional, the execution of the configuration should take a reasonable amount of time. The criteria describes the situation that the handler is designed to fulfil its purpose with a minimum of API communications and calculations. This shall avoid wasting precious CPU resources.

Security
It is always recommended that a software product has a degree of security. In the best case, the handler will be integrated into production landscapes at some point. Running in a production landscape, the handler can change parameters of HANAs that manage actual customer data. Therefore, the degree of security of the handler should be rather high in comparison to e.g. a monitoring service that only observes resources that are running in the landscape. The handler should only be granted permissions that are needed. Decisions about concepts that are used in the implementation of the handler should be, amongst other aspects, based on the mandatory permissions. This means avoiding concepts that need critical permissions, if possible.

Scalability
The landscapes are Kubernetes clusters, thus, they are distributed systems that can dynamically scale up and down in size. The scalability is an aspect that is valued in the majority of services which are running on such systems. Even though it is not expected that the handler will be used constantly and in high frequency, the possibility that it can scale up and down with the usage density or with the number of HANAs would be preferable, as this is an aspect that belongs to cloud-native programming.

After I defined the criteria, I thought about what the end result should be able to do. A discussion with my supervisor and another software architect resulted in the following use case:

The handler should be used for the reconfiguration of HANAs. If a HANA must be changed, this should be done without executing the configuration manually. An improvement would be if the configuration can be based on profiles. The profiles could hold parameters and their desired values. In critical situations, these profiles must be temporarily changed or created by the operations team. The configuration change in the profiles should also be forwarded to the HANAs that used the profile. It should also be possible to identify the state of a HANA. This includes if and how the HANA is reconfigured.

In the next step, I reviewed the general architecture of our landscapes and thought about the basic aspects of the handler that must be clarified. The questions that occurred in this process were afterwards discussed with my supervisor to make sure that the end result fulfils the requirements of the company. In the following the questions and answers are listed:

1. Where is the handler running?
   The handler is running in the landscape with the HANAs that it is supposed to configure. Otherwise, the handler would have to execute the reconfiguration from outside the cluster. This decision should minimize the communication between the cluster and the outside which is favourable for the security and performance aspects.

2. Are the profile definitions supposed to be managed in a central location?
   Yes, this would be good. They have to go through the release process which is also only possible at a central location.

3. Who has permissions to use the handler?
   The operations team and developers that have to support the HANAs.

4. Should HTTPS requests be used?
   This is desirable, but HTTP is sufficient for the prototype.

5. What can be expected to be known by the requester?
   It can be expected that the requester knows the HANA ID and image version of the HANA(s) that he/she wants to reconfigure.

After the first questions were discussed, I created mind maps and lists of the functionalities or features that would be mandatory to be included in the service for it to fulfil its main purpose. Before and while creating concepts on which, later on, the implementation of features or processes of the service was based, I brainstormed about challenges or disadvantages that might occur when going through with an idea or concept. This step was supposed to help me identify avoidable mistakes when making design decisions.

I have been working for two years in a company that is using an adaptation of SCRUM[14] as their software development framework. A basic concept in this framework is the iterative development process. Some might be more familiar with extreme programming[15]. Extreme programming is also based on the iterative concept. Due to being already familiar with SCRUM, I wanted to implement my service using an iterative process. The prototype started out as a simple service which was only able to run in the cluster. Successively, more and more functionalities and whole features were added. I decided to proceed in that way to make sure that I have a prototype that is working after each extension or improvement. The first real functionality that was added was the logging system. It displays messages and can be used to track errors and unexpected behaviours within the handler or during network issues. The first bigger step was to expose the service to the outside of the cluster. From this point onward, features were added and other decisions were made. The features and design decisions are explained in Section 5.

[14] https://www.scrum.org/resources/what-is-scrum
[15] http://www.extremeprogramming.org/

5 Design & Implementation

In this section, all implemented features of the handler are explained as well as the design approaches that were used. For each feature, other possible approaches will be mentioned each followed by reasoning why these approaches were not chosen. Lastly, the produced artefacts and documentation are stated. There are two preconditions for the handler to work. First of all, no HANA is any more manually configured after the introduction of the handler. Secondly, other services do not adjust the configurations of the HANA after it has been fully provisioned. The supervisor of my bachelor thesis at work found the first condition to be reasonable. The second condition is currently also fulfilled. Each subsection deals with a feature or an important functionality. Some subsections contain diagrams or sample code snippets for an easier understanding. Additionally, a representation of the final architecture is included in the Appendix (see Figure 5).

5.1 Availability from outside the cluster

The decision to make the handler available from outside the cluster was made for usability reasons. Doing so, it is possible to send an HTTP request to the handler. The requests are currently handled without authentication of the requester. Thus, the availability from outside the cluster is a security issue that requires an additional usage restriction to make sure that only requests are handled that are sent by users with the right permissions. Such a restriction is not yet introduced, but possible approaches to tackle this issue are described in Section 6.1.

There are multiple ways in Kubernetes to expose a service. The prototype is exposed externally using a load balancer. The load balancer is a Kubernetes service resource that owns an IP address and routes the traffic to a specific port of each Node, called NodePort. The handler is mapped to this specific NodePort. The handler can receive HTTP requests that are sent to the IP address of the load balancer. The set-up of a load balancer that is supporting HTTP traffic is rather simple. The load balancer service configuration contains a port and target port that must be set. The container of the handler must be configured to listen on the target port that was set. This can be configured in the Kubernetes pod or deployment resource of the handler. The disadvantage of load balancers is they can get costly. If a load balancer is created for each microservice that should be exposed, it is very pricey. The generated costs have multiple reasons. One reason is that every load balancer needs an IP address which must be bought or rented. Furthermore, switches are needed that must support the bandwidth of the load balancers and the load balancers need also additional CPU resources. A solution for this is to expose the microservice via an ingress on all nodes and to use one load balancer for all microservices. In the case of the prototype, I created a separate load balancer for learning purposes instead of using one that was already available in the landscapes.

Another solution is to use a ClusterIP which is also a Kubernetes resource and exposes the microservice on an internal IP address within the cluster. Additionally, a Kubernetes proxy must be open that connects the cluster with the external machine.

This solution was repeatedly rated as unfit for the use in productive landscapes. A separate proxy must be opened for each external machine that wants to reach the handler [2, 3]. Therefore, I decided to use the solution of exposing the handler via a load balancer.

5.2 Request Handling

I considered two concepts of handling requests. Either the handler checks continuously if a label or condition of a specific resource, e.g. HANA pod was changed (e.g. a condition that can go from `configured` to `configuration requested`). The resource should also contain additional configuration details that are supposed to be executed, such as parameters and values or a profile ID. This concept uses the polling method.

The other option is to implement the handler to act only on-demand. This means the handler does not execute any checks explicitly, but exists in the state of busy waiting until a request arrives at its endpoint. The polling method is rather common for microservices in our landscape which are working with conditions. I decided to use the on-demand based concept due to the fact that this concept is not using as much CPU resources as the polling concept would have. Furthermore, it is expected that the handler will not be used constantly, but still should react within a reasonable time after a request is executed. The polling concept would waste more CPU resources for the time frame where the handler is not in use.

5.3 ConfigurationProfiles

A ConfigurationProfile (in short: profile) is the concept of a resource that I designed as part of the prototype. Such a profile contains an ID, HANA configuration parameters, and the desired values. The profiles should be designed to fulfil the following conditions. First of all, the profiles should be able to exist in a Kubernetes cluster on their own. Meaning, the deployment of a profile should not depend on the existence of the HANA configuration handler or other services. Secondly, the profiles should be separate from the configuration handler which means they should be deployed as a separate artifact. Temporarily existing profiles are excluded from this precondition. This is mandatory because the profiles must go through the release process of the company without being tied to the configuration handler release. The solution was to create a separate repository that contains a Helm Chart[16] which defines the profiles. Helm[9] is a Kubernetes package manager. It can define, deploy and upgrade Kubernetes resources. A Chart is the package format of Helm. The profiles are defined as ConfigMaps. A ConfigMap is a Kubernetes resource that manages data in the form of a map that only supports strings as values. Additionally, labels or other metadata can be set in it. Due to the fact, that HANA containers can be built from different image versions (in the following also HANA version), the profile ID should indicate the HANA version that is supported by the profile. Therefore, the profile ID starts with the profile name that should indicate the purpose of the configuration, e.g. bug-A or workload-XL. The second part of the ID is the HANA version. Both

[16] https://helm.sh/docs/topics/charts/

parts are separated by a hyphen. ConfigMaps can exist in a Kubernetes cluster without any dependencies and the release process of the company supports the release of Helm Charts. Therefore, all conditions are satisfied. The usage of another Kubernetes resource, such as secrets, would have satisfied the conditions as well, but in the end, both would have been used similar to the ConfigMap. Same goes for defining a custom resource. The following shows a sample of a ConfigurationProfile for the HANA:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: sample-0.0.1
  namespace: default
  labels:
    type: hana-configuration-profile
    profile-name: sample
    profile-version: "0.0.1"
data:
  configurationspecs: |
    - filename: "global.ini"
      layer: "SYSTEM"
      section: "persistence"
      key: "log_segment_size_mb"
      value: "111"
    - filename: "nameserver.ini"
      layer: "SYSTEM"
      section: "auditing_csvtextfile"
      key: "max_files"
      value: "8"
```

Listing 1: Sample of a ConfigurationProfile

A different approach is not to locate the profiles in the cluster, but to provide them in a web tool with a user interface, such as SAP Cockpit, and send the parameters and the ID of the profile to the handler. However, user interfaces, such as SAP Cockpit's, are available for external users. Additionally, the workload generated by adjusting or generating a user interface does not hold much value. The users of the handler are expected to work confidently with command-line tools. Furthermore, the concepts of the profile rollback and update are based on the fact that the profiles exist in the cluster. So I decided to not further investigate this approach.

5.4 HANA Configuration Process

Multiple components are involved in the HANA configuration process. The relations of these components are roughly shown in Figure 3. For more detail see Figure 5 in the Appendix. The process starts with (1) a requester sending a configuration request to the HANA Configuration Handler. The requester can be a developer or operations engineer. A configuration request can explicitly state parameters and the values that they shall be set to. Otherwise, a ConfigurationProfile ID must be set. Additionally, there is an option to request the configuration of a whole set of HANAs instead of

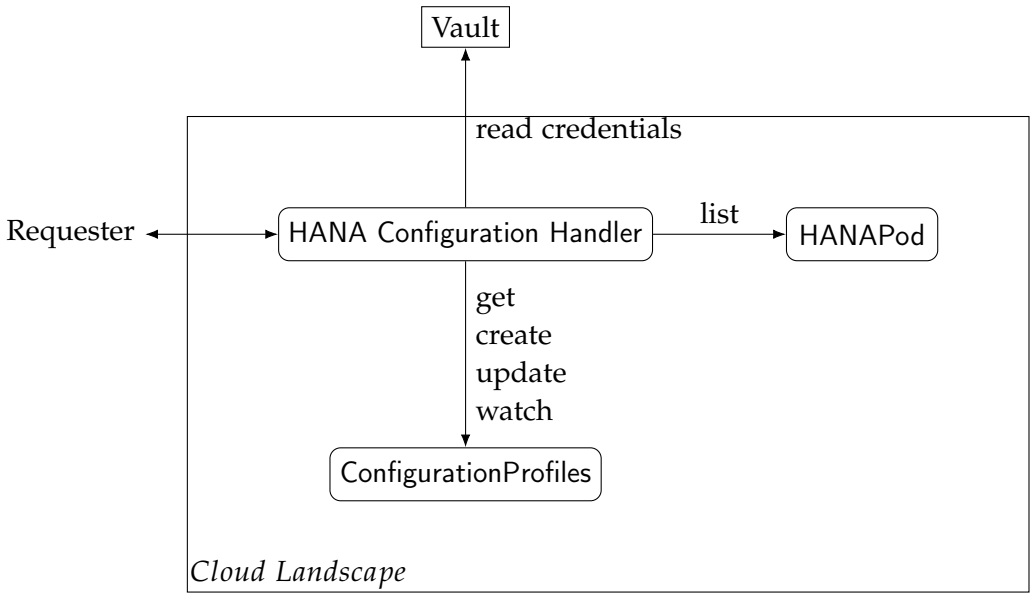

Figure 3: Relation of HANA Configuration Process Components

only one. Depending on the option in use, the request is more or less complex in its structure, as shown in Listing 2 and 3.

```
curl -d '{ "instancetype": "hana", "instanceid": "<HANAID>", "
    instanceversion": "<HANA-version>", "parameters": [ {"
    filename":"dummy.ini","layer":"SYSTEM","section":"dummy-
    section","key":"dummy-key","value":"2"}, {"filename":"foo.
    ini","layer":"SYSTEM","section":"foo-section","key":"foo-key
    ","value":"true"}]}' <external-ip>:<target-port>/configure
```

Listing 2: Request for the configuration of a single HANA with single parameters

```
curl -d '{ "instancetype": "hana", "instances":[ "<HANAID-1>",
    "<HANAID-2>"], "instanceversion": "<HANA-version>", "profile
    ": "<profile-name>" }' <external-ip>:<target-port>/configure
```

Listing 3: Request for the configuration of a set of HANAs with a ConfigurationProfile

(2) The load balancer routes the request to the HANA configuration handler. (3) The handler parses the request and validates it. Afterwards, it checks if the requested HANA instance or instances are present in the cluster. The process stops if a requested HANA instance is not found. For requests that contain a profile ID, the handler also checks if the ConfigurationProfile exists in the cluster. In case that the profile exists, the parameters and desired values are retrieved from the profile and are used as configuration parameters. In case that the previous steps were successfully executed, (4) the access credentials to the SystemDB, a password, must be retrieved. The password for the SystemDB is landscape-specific which means that one password is valid for all SystemDBs of HANAs that are running in the same landscape. This password is saved in a Vault[17]. This is a secret store provided by HashiCorp. A Vault

[17] https://www.vaultproject.io/

client is built within the handler to obtain the SystemDB password. The Vault client needs the permission to read the password from a specific path in the Vault. This permission is given to the handler via a ServiceAccount that had to be added in the Cluster configuration. The Kubernetes ServiceAccount resource is part of the role-based access permission (RBAC) concept. In the Kubernetes environment, the RBAC concept works with Roles, ClusterRoles and ServiceAccounts. A Role or ClusterRole defines permissions. These roles can be bound to a ServiceAccount. In that way, the ServiceAccount receives its permission. In the further course of this thesis, the process of configuring roles and binding them to a ServiceAccount is described as configuring the ServiceAccount. This shall make it easier to read. It is also used to grant the handler permissions to work with Kubernetes resources. The ServiceAccount can be configured to grant the permission to e.g. list pods. At this point in the process, the access credentials, configuration parameter, and hostnames are known. The hostname is a fixed prefix, which is internally known, plus the HANA ID.

Now, (5) for each HANA a SQL connection is successively created and the configuration statement is executed for each parameter. The SQL statement is structured as follows:

```
ALTER SYSTEM ALTER CONFIGURATION('<INI-filename>', '<layer>')
   SET ('<section>', '<key>')='<value>' WITH RECONFIGURE
```

Listing 4: Reconfiguration Statement

After each successful configuration, the connection is closed and (6) the `profile` label is added or, if already present, changed to the profile ID that was used (if no profile was used the value is `single-param`). After this, HANAs are configured, (7) the process is completed.

The `profile` label is used to retrieve an overview of how HANAs are configured. Kubectl can be used to get all HANAs that e.g. are configured with a specific profile by selecting those that own the `profile` label with the desired profile ID as value:

```
kubectl get pod -l profile=<profile-ID>
```

Listing 5: Sample command to get all HANAPods configured with a specific profile

The configuration of a HANA could also be executed in a different fashion. The SQL statement that the handler uses adjusts the parameters in the database and updates the INI file. The other approach is to update the INI file. Afterwards, the reconfiguration based on the INI files must be triggered. For this, there is a separate tool is present in the HANA.

Both solutions are valid, however, the second method requires the handler to enter into the HANA container to adjust the files and trigger the reconfiguration. To grant the handler the permissions to enter the HANA container, which means granting permission to exec the HANA pod, the RBAC concept would be used. Exec is short for execute. In this context, being able to exec a pod means to be able to execute

a command in a container of the pod or to enter the container[18]. Unfortunately, the RBAC can only grant permissions for a specific resource or a specific instance of this resource, if the name is known. This means the handler can be granted to either exec every pod or a pod with a specific name that must be known before the ServiceAccount is deployed. The amount of HANA pods and their names are not known, because they are generated dynamically. Therefore, the only possibility would be to grant the handler the permissions to exec all pods. This is not a very safe solution. Therefore the handler configures the HANAs via a SQL statement instead of entering the HANA containers.

5.5 Profile Creation/Update Process

In general, ConfigurationProfiles should be defined as a ConfigMap in a Helm Chart and go through the release process of the company. Changes on the ConfigurationProfile should also be executed in this way. Nevertheless, there are scenarios where the operations team has to add or adjust a ConfigurationProfile under time pressure. This means the whole release process should be skipped. The improvement is to enable the handler to do so. This makes it possible to create and adjust ConfigurationProfiles via an HTTP request that is sent to the handler. The possible request types are called `change` and `create`. They have the following structure:

```
curl -d '{ "requesttype": "<request-type>", "profilename": "<
    profile-name>", "profileversion": "<HANA-version>", "
    lifespan": 3, "parameters": [{"filename":"dummy.ini","layer
    ":"SYSTEM","section":"dummy-section","key":"dummy-key","
    value":"2"}] }' <external-ip>:8080/profile
```

Listing 6: Request for the Creation/Change of a ConfigurationProfile

The lifespan value is optional. It indicates the days that the profile or the adjustment in it is valid. After the lifespan runs out and in case of a creation request, the ConfigurationProfile will be deleted. The default lifespan value is set to 7 days. See Section 5.7 for more. Similar to the configuration process, the request is received by the handler and gets validated. Afterwards, the handler checks if the requested ConfigurationProfile exists. A creation request is not allowed to create a ConfigurationProfile that is already present. Change requests can only be executed for ConfigurationProfiles that are present. If the conditions are not fulfilled, the process stops. Otherwise, the configuration parameters of the requests are prepared to satisfy the profile convention and the ConfigurationProfile, which is a Kubernetes ConfigMap resource, will be either created or updated with the parameters. In case of a creation request, a label will be set to the ConfigMap that sets a date, based on the lifespan value, at which the profile will be deprecated. In the specific solution, this label is called `deletionDate`. In case of a change request, the handler overwrites the whole parameter section of the ConfigMap. Therefore, even if only one parameter shall be changed, the whole set of parameters that are already present in the profile have to be contained in the

[18]For an example of the exec command option see https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#exec

request. The mechanism of changing parameters separately in the ConfigMap is complex. I came to the conclusion that it does not hold much value for the current use cases.

5.6 HANA Configuration Update Process

This update feature is used in situations in which a ConfigurationProfile is changed via a re-deployment of the Helm Chart or via a change request. It is desired that all HANAs that used the updated profiles are reconfigured with the new ConfigurationProfile content. This avoids inconsistency in the configuration of HANAs. This means the following. Assume multiple HANAs are configured with the same ConfigurationProfile. Their configurations were executed at different points in time. Still, they should not show differences in their configuration. In short, their whole configuration should be equal.

The first important functionality is to detect changes in the ConfigurationProfiles. Due to the fact that changes can be executed independently from the handler, it is not efficient to simply execute a reconfiguration of HANAs right after the handling of a change request. The main point of the solution is, that the handler is watching the events of ConfigurationProfiles, more specifically ConfigMaps owning the `profile-name` label. This watching process is running in a thread in parallel to the HANA configuration handling. If the handler detects a change or deletion of a ConfigMap, the handling process is triggered. It is important that each triggered handling process is also running in a separate thread. In the specific solution, I used goroutines[19] which are light-weight threads. This avoids the situation that the handler still handles the reconfiguration of HANAs and misses a ConfigMap event that occurs in parallel to it.

The deletion of a ConfigMap triggers a different handling process than a ConfigMaps change would. In both cases, the first step is to check if the ConfigMap is actually a ConfigurationProfile. Only if the ConfigMap is a ConfigurationProfile, the handler proceeds. In case of a ConfigMap deletion event, all HANAs are obtained that use this profile and a rollback of their configuration to the default configuration is executed. The rollback is explained in Section 5.8. Afterwards, for each HANA the `profile-name` label will be set to `default`.

In case of a ConfigMap change event, the new content of the ConfigMap is obtained and all HANAs that used this profile get rolled back to their default configuration and then get reconfigured with the new configuration parameters. This completes the handling process. The thread/goroutine that handled the process gets terminated.

5.7 Profile Deprecation

The deprecation of a ConfigurationProfile indicates, that the profile should no longer exist in its current state. This can mean that the profile was changed by the handler and should be reset to the previous state, e.g. the state that a Helm Chart defined.

[19] https://golang.org/doc/effective_go.html#goroutines

Another scenario is that the profile was created by the handler and should be deleted. The specific implementation of the handler is currently only supporting the profile deletion and not the rollback. Profiles that are created via the handler should not exist longer than they absolutely must as they have not gone through the official release process of the company. They were only created as a temporary workaround for a bug or for similar reasons. As explained in the profile creation process, a `deletionDate` label is added to the profile in its creation phase. This label marks the date after which the profile should be deleted.

The profile deprecation feature frequently checks if a deprecated profile exists in the cluster. The lifespan that can be set in the profile creation request has to be between one day and a week. Therefore, I concluded that a check that gets triggered once or twice a day is sufficient. The feature is realized with a Kubernetes CronJob and a Python3 script. The configuration handler deploys the CronJob right after the handler is fully created. Each time that a profile is created it will be checked, that the CronJob is present. If the CronJob is missing, it gets redeployed. Otherwise, it gets simply updated to make sure that the definition of the CronJob is equal to the definition that is defined in the handler.

A CronJob has the option to create Kubernetes Jobs based on a schedule. In the case of the handler, it creates a job every day at 12 am and 12 pm while the cluster exists. The job creates a pod which is running a container that executes the Python3 script. This container holds permissions to build a Kubernetes API client that can list and delete ConfigMaps, as shown in Figure 4. The script obtains all ConfigMaps that contain a `deletionDate` label. These ConfigMaps are all ConfigurationProfiles, as they are the only ConfigMaps that may own such a label. Afterwards, it checks if the timestamp is older than the current one. If this is not the case it proceeds with checking the rest of the profiles. If a profile is found that holds a deprecated `deletionDate` label, the object of it will be added to a list. After all profiles are checked, the list is either empty and nothing has to be done or deprecated profiles were found. In that case, all profiles, that are associated with an object of the list, get deleted. This completes the job. The pod gets terminated and the CronJob creates the job again at a later point in time as it is scheduled.

5.8 Rollback of HANA Configurations

The rollback feature is needed in two scenarios. The first scenario is that a HANA is not in the default configuration and shall be reconfigured. The parameters that currently differ from the default configuration do not have to be included in the set of parameters that are requested to be configured. This means these parameters have to be reset to their default values before the reconfiguration can be executed. This makes sure that the configuration stays consistent. Meaning, HANAs that are configured with the same profile have the same configuration and not a single parameter is set to a different value. The other scenario is that the HANA is configured with a deprecated profile. After the profile is deleted the configuration of the HANAs must be rolled back to the default configuration to avoid having HANAs being configured based on profiles that do not exist any more.

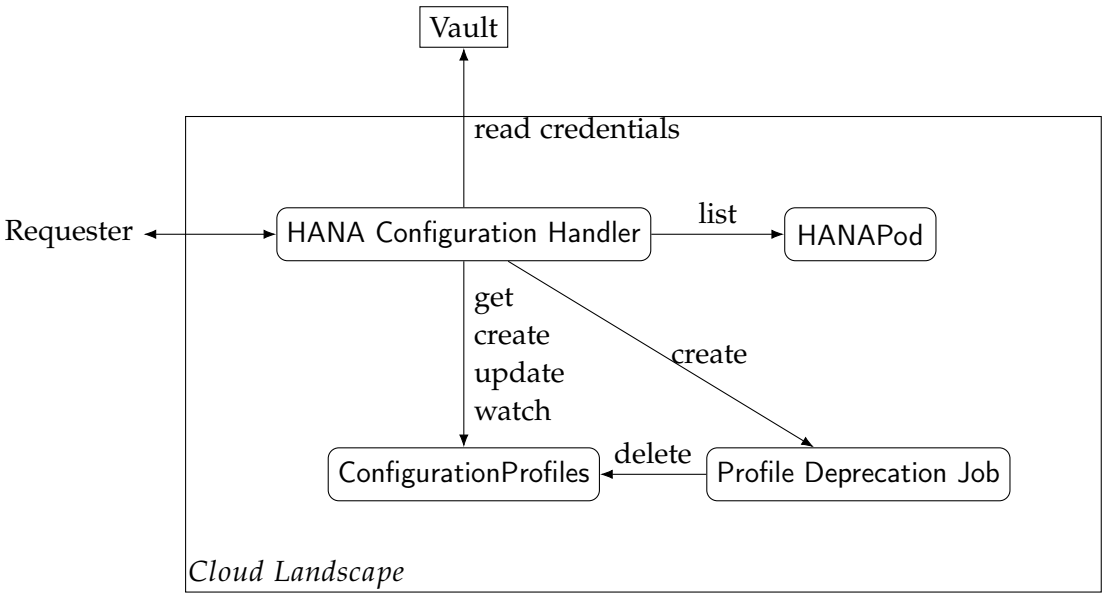

Figure 4: Relation Overview of the ConfigurationProfile Deprecation Feature Components

To make it possible to rollback a configuration the parameters that were adjusted and their default values must be known. The SystemDB contains a DEFAULT layer, that contains the default configuration. However, there are infrastructure services in place that reconfigure the HANA right before it is fully provisioned. Therefore, the alleged default configuration listed in the DEFAULT layer does not equal the default configuration that is desired by the HANA Cloud landscape. As a consequence, the data from the DEFAULT layer cannot be blindly used. Instead, when the HANA is reconfigured by the handler for the very first time, it retrieves the values of the parameters that are contained in the configuration request and saves them as the default values in a map. The map is structured as follows:

```
{ <hana-ID1>: [ <parameter1> : <value2>, <parameter2> : <value2
   >], <hana-ID2> : ...}
```

Listing 7: Sample Structure of the Default Value Map

It must be noted, that a parameter consists of multiple subparameters, as shown in Section 2.2. Therefore, the parameter is a struct of multiple values in the actual implementation.

The default values for a specific HANA can be retrieved by the HANA ID. The default values are saved for each HANA as we cannot be sure that a default configuration is valid for HANAs of different image versions. Saving the default values in a map in the handler means the data is saved in the memory of the handler. This does hold a big disadvantage that will be discussed at the and of this section. Now, if a HANA is about to be reconfigured, the handler checks if the HANA is in the default configuration. An indicator for this is, that the HANA pod does not own a `profile` label or that the `profile` label is set to `default`. If that is the case, a rollback must not

be executed before the reconfiguration. Otherwise, the default values are retrieved from the map and the HANA is configured based on the default values. After that, the handler proceeds with the actual reconfiguration.

The disadvantage of this solution is, that in case of a restart or the termination of the handler, all default values are lost. This means even after the handler is running again, a rollback of a HANA, that was configured by the handler before, is not possible due to the missing data. The values of the HANA that are already configured might be saved as default values as the handler does not check if the HANA was already configured. This is due to the fact, that the handler only saves the previous values of parameters that are configured. Optimally, when a HANA is reconfigured multiple times, the process consists of rolling back the parameters that were previously configured, saving the current values of the parameters that are requested to be changed and executing the configuration. If the handler would have been running the whole time, including during the HANA provisioning, it could be assumed that the saved values actually represent the desired default values. However, if the handler was at some point restarted or terminated, it cannot be assured that the requested configuration equals the actual configuration of a HANA. There might be parameters that were not correctly rolled back to the desired default. In short, the values of already configured parameters might be saved as default values due to failed rollbacks.

The current implementation of the rollback feature is only a temporary solution for the prototype and should be improved. Therefore, approaches to improve this feature are discussed in Section 6.2.

5.9 Artifacts and Documentation

The prototype is not one single service. It consists of the configuration handler, a CronJob and the profiles. The configuration handler and the definition of the CronJob are located in the same repository. Both, the handler and the CronJob are using a Docker image each that must be built and pushed before the deployment of the handler via a Helm Chart is possible. The ConfigurationProfiles are located in a Helm Chart in a separate repository. It is possible to create multiple Helm Charts and/or repositories for profiles as they are not depending on any services etc.

Additionally, each repository contains a simple documentation on how to deploy and use the artifacts that are contained in the repository. Lastly, separate documentation is available, which contains an implementation diary, the exposé of the bachelor thesis, concepts, problems and questions that occurred including answers and solutions. In summary, the prototype includes the following:

Docker Images
The first Docker image creates the configuration handler container. The other image contains the script that checks for deprecated profiles. It is used in the CronJob that is referred to in the profile deprecation feature.

Helm Charts
One Helm Chart contains all necessary definitions of Kubernetes resources to deploy

the configuration handler. This includes the deployment of the pod in which the configuration handler container is running as well as the ServiceAccount definition for the RBAC method and the load balancer service definition to expose the handler to the outside of the cluster. The second Helm Chart is a collection of profiles, thus, multiple ConfigMaps definitions.

Documentations
Each repository contains a simple documentation. A detailed documentation is found in a separate repository.

6 Future Work

The handler that was developed in this bachelor thesis leaves room for improvement. There are aspects in the implementation that are needed to be improved to release the handler into a production landscape. This includes using a more efficient approach for the rollback feature, the introduction of a restricted usage of the handler and the extension of the profile deprecation feature. Furthermore, there are also optional adjustments that can be made to increase the quality. This section discusses improvements and possible approaches that could be used to realize them. Note that there are also small improvements that are not discussed in this section as they do not hold any complex logic. For example, a check if the HANA is already configured with the profile that is requested to avoid unnecessary reconfigurations, or the switch from HTTP to HTTPS.

6.1 Restricted Usage of the Handler

The handler is exposed to the outside of the cluster. Currently, everyone with the right information can send a request to the handler. It is not validated if the requester holds any permissions to grant him the privileges to use the handler. Furthermore, there are supposed to be two user groups with different access rights. One user group shall only be granted to use the handler's reconfiguration abilities. The other group shall be able to also create and adjust configuration profiles if necessary. In the following two approaches will be presented that could be used to realize this feature.

The first approach can limit the access to the handler to a group of employees that already have the permissions to actually access HANAs. This means these employees have already the ability to configure the HANA manually. Viewing the current cluster access groups, a user that can exec HANA pods can also create, update and delete ConfigMaps and thus also ConfigurationProfiles. This means this solution grants permission only to users that would be explicitly allowed to use the handler, but there is no differentiation in the usage permissions. Therefore, it might be possible that users are excluded from this group even though they should have at least partial access to the handler. The logic of this approach is to expose the handler only inside the cluster. A pod can be created that contains a container that is permanently in an idle state. To use the handler, the user must enter the idle container. There he/she and can send a request to the internally exposed handler. The solution is simple, but the user experience might be compromised and some users might be unintentionally excluded from using the handler

The other approach would be to make use of a framework that is used in our company to manage the permissions of each employee. It can grant e.g. read and write permissions to Github[20] repositories or permissions to create a development cluster. By using the Kubernetes ServiceAccount resource and the framework, it is possible to grant each user specific permissions within the landscape. The user can request the right ServiceAccount in the framework. The permissions of the ServiceAccount can be forwarded to the handler. This means the handler does not use its own ServiceAc-

[20] https://github.com/

count, as it does currently, but the ServiceAccount of its user. This ensures that only those, who are explicitly allowed to use the handler, can do so. When a user does not own the right ServiceAccount the handler will fail in its tasks as it does not have the permissions to e.g. obtain pod resources or read the SystemDB password from the Vault.

6.2 Rollback Feature

The implemented rollback mechanism of the handler is rather naive. In Section 5.8, the disadvantages of the current implementation are discussed. The default values that are necessary to rollback the HANAs to their default configuration are currently saved in the memory of the handler. Therefore, if the handler is restarted or deleted, all default values are lost and the wrong data might be later on saved as default values.

The first possible approach is to save the default values to an independent resource, such as a ConfigMap or a PersistentVolume. Both are Kubernetes resources. In case of the restart or deletion of the handler, these resources still are able to exist in the cluster and when the handler is back up again it can retrieve the data from the resource without problems.

The second approach would be to define a configuration profile of each HANA version. They contain all configuration parameters and the default values. The handler would only save for each configured HANA or ConfigurationProfile (if it was used), the parameters that were configured in e.g. a ConfigMap. Then in case of a rollback, the handler retrieves the parameters that were configured from the ConfigMap and the default values from the right default configuration profile. The creation of the default configuration profile can be automated by a parsing script. The downside of this approach is that the default configuration profiles contain a lot of data that must be iterated through to find the right parameters. This will slow down the whole rollback process. Therefore, the first approach is preferable. A very different approach would be to work with the values of the DEFAULT layer of the SystemDB and provide a resource, such as a ConfigMap, in which configuration parameters can be set. The values from the resource would have priority over the values in the default layer. This method would require the developers of services that are adjusting the values of the HANA initially to update the resource. Otherwise, their configurations could be overwritten at some point. The rollback to values of the DEFAULT layer can be easily done with the following statement:

```
ALTER SYSTEM ALTER CONFIGURATION('<INI-filename>', '<SYSTEM|
    DATABASE>') UNSET ('<section>', '<key>') WITH RECONFIGURE
```

Listing 8: Rollback Statement

For this approach, the handler remembers for each HANA or profile the parameters that were changed, like in the approach that was discussed before. Then it would check if the resource contains any of these requested parameters. If that is the case, the handler will configure the HANA with the corresponding value from the resource. Otherwise, it plainly resets the value. The downside is that when a team of developers

releases a service that changes the default configuration without knowing about this mechanism, their desired default configuration will be overwritten at some point. The mechanism creates a new dependency in an already complex landscape.

6.3 Deprecation and Rollback of ConfigurationProfiles

The deprecation feature of the prototype only supports the deprecation of temporarily created profiles. The deprecation mechanism for temporarily changed profiles is an extension that should be introduced at some point. Profiles that are created via the handler own the `deletionDate` label with a timestamp value. For changes on profiles that are executed by the handler, the `rollBackDate` label must be added. Currently, this process is implemented but not executed as the actual support is not yet realized. The basis for the implementation of this feature is already in place. The CronJob that is already used for the Deprecation Feature can be extended. The script that is executed in its Jobs can be adjusted to not only check for profiles that own a deprecated `deletionDate` timestamp but also for a deprecated `rollBackDate` timestamp. Similar to how the script deals with a deprecated `deletionDate` label, a list is created of all profiles that own the deprecated `rollBackDate` label. After all available profiles are checked, a request should be sent to the handler. This request shall contain the IDs of all profiles that are supposed to be rolled back to its previous version. Note that it can happen that a profile is changed multiple times via the handler and not just once before it is rolled back. In that case, the profile should be rolled back to the version that is defined as the default version. This is the version that is defined in the Helm Chart from which it was actually deployed. For this, the handler must be extended to handle a new version of the profile request. Currently, only the request types `change` and `create` are available. The request structure and the new type can easily be implemented. The challenge is to know the version of the profile to which it shall be rolled back. To tackle this issue I selected one possible approach. Other approaches would include granting access permissions to the repositories that contain the Helm Charts in which the definition of the profiles are located. Due to aspects of the release process of the company, I decided not to further investigate these options. The approach I would like to implement in the future works as follows. We know, the profile saves all parameters as one text string in a map, as shown in Listing 1. The key to the parameters is `configurationspecs`. Now, if a change profile request is sent to the handler, it retrieves the profile that shall be changed. At this point, we can adjust the handling. Instead of just changing the profile the handler can obtain the string value of `configurationspecs` and save it under a new key, `rollbackversion`. Note, if the `rollbackversion` key already exists in the profile the new step can be skipped. The `rollbackversion` key is set only once and represents the version of the profile that was defined in a Helm Chart. If the profile is updated by the redeployment of a Helm Chart, the `rollbackversion` will be gone, thus, the new valid version defined by a Helm Chart will be identified as the version to which the profile shall be rolled back. Afterwards, the value of `configurationspecs` is changed as requested and the changes are applied to the profile that is running in the cluster as usual.

6.4 Watching Profile Events

The handler is watching the ConfigMap events that refer to changes in or the deletion of ConfigMaps that are identified as ConfigurationProfiles. As long as the handler is continuously running in the cluster, the method works fine, but as explained in the rollback feature, there might appear issues, if the handler is gone for some time. In case of a restart or the deletion of the handler, it might miss a change or the deletion of a profile. This would have the consequence that HANAs are still configured with a profile that does not exist any more or that was already changed. As the rollback of profiles, that are only supposed to be temporarily changed, is not yet supported by the handler I have not thought of a solution to identify this case. For the case that HANAs are configured with profiles that do not exist any more, I selected two possible approaches.

Either, a CronJob can be added that contains a Job that checks all HANAs using profiles, similar to the CronJob used for the profile deprecation. Or, a thread can be created in the handler that does the same. In both approaches, the following would be executed. First of all, the IDs of all profiles are retrieved that are running in the cluster. They could be saved in an array. Afterwards, all HANAs are checked that were configured with a profile. For each HANA it is checked if it is configured with a profile that owns an ID that is not in the list. If that is the case, the HANA should be rolled back. In the handler, the rollback function can be simply called. For the Job, created by the CronJob, a new request option must be introduced that requests the rollback of a HANA or group of HANAs. For both approaches, it is possible that a situation occurs in which a configuration and rollback request are sent at the same time. It might happen, that the rollback happens after the configuration request was executed. This would result in the HANA being in its default configuration state. Before implementing one of these approaches it should be discussed if such a situation can be tolerated.

6.5 Concurrency in the Configuration Process

Introducing concurrency into the configuration process is an optional improvement and might not be very meaningful for the use cases of the handler. The handler is written in Go. One of the advantages of this programming language is its goroutines. Goroutines are basically light weight threads that can be easily created. They are already used in the handler to watch for ConfigMap events and to handle them. The main use case of the handler is to only configure a single HANA. It is expected that the handler is not frequently used or by multiple employees at the same time, but rather in special cases. Still, if a situation occurs in which a bigger group of HANAs should be configured, it would improve the performance of the handler if the configuration of the HANAs is not executed successively but in parallel. The question is if this is an option that would hold great value or if it would only create race conditions or other complications. For the time being, it is not really necessary to realize this idea. If the usage of the handler changes to rather configuring multiple HANAs, the idea could be discussed further.

7 Challenges

The first challenge was to define a strict scope of what the prototype shall realize and what the written bachelor thesis shall discuss. Implementing a service, such as the configuration handler, it can be rather tempting to implement concepts and features that are growing in complexity but not much in value. Therefore, I needed to repeatedly remind myself to focus on the basic functionalities that the handler should contain. As this is a thesis written with the cooperation of SAP, I also had to make sure to not reveal internal details about the infrastructure or similar things. This turned out to be not as much of a complication as I first expected it to be.

The second major challenge was the integration of the configuration handler into an existing system. The most time-consuming example is the implementation of the SQL client. Its purpose is to connect to the SystemDB and send queries. While testing it, the problem occurred that even though the right SystemDB password, hostname and port were used no traffic that was generated by the handler was routed to the SystemDB. The reason was a proxy service, which only allows traffic from pods that own a specific label to be routed to the SystemDB. Working on a service that is running in a system that is supported and developed by multiple teams, it is quite hard to maintain an overview of all specifications, dependencies and necessities. Therefore, it is important to find the right documentations or find help from colleagues.

8 Evaluation

The evaluation is divided into three parts. The first part deals with the workflow of the design and implementation phase. The second part focuses on the prototype, and the third part rates the concepts that are presented in Section 6.

8.1 Workflow

The beginning of realizing the service was mainly used to identify possible challenges that would be avoidable. After creating a concept I reviewed it very critically to find possible flaws as early as possible. Doing this, I tried to make use of the knowledge that I collected in the last year working in the department of the company that is engaged in developing microservices in a Kubernetes environment. I believe the early and critical review of each concept did improve my decision making. Still, it was time not spent on implementing or researching. It is natural that choices will be changed in the implementing process as not all challenges can be identified in theory. However, in my opinion and experience, it was a smart choice to take the time to make the first and biggest decisions of a project. Therefore, I would not say that the time spent on the designing of the service was unreasonable.

For the implementation, using an iterative software development process was in this case efficient. Building a service that is supposed to run in a landscape that can change much over time, it is important to continuously check if the service is compatible with its designated environment. It happened a few times that network issues appeared within the development landscape. Due to the fact that I was able to repeatedly check that the handler was working, I could estimate if unexpected behaviour was caused by temporary issues in the landscape, or by the implementation of the handler itself. Thanks to that, I could continue adding features or functionalities and did not spend much time trying to solve issues that did not relate to the handler at all.

In summary, I am pleased with the execution of the design and implementation phase.

8.2 Prototype

The HANA Cloud configuration handler that was designed and implemented in the context of my bachelor thesis is a prototype. This means this handler is not supposed to be the final solution, but a tool to educate oneself about how to tackle the task of automating the reconfiguration of HANAs in a way that was requested. The evaluation will be based on the four quality criteria that are introduced in Section 4.

Usability
An HTTP request has to be sent to use the handler. It is expected from the user that he/she possesses all the necessary data to send a configuration, create or change request. The only information that must be separately obtained is the external IP address on which the handler is exposed. This process is not very complicated. In the 'how to' documentation of the handler it explained how the IP address is obtained

by executing one simple command. As soon as the user possesses all needed data, he/she is able to send a request to the handler. The data describing the task to the handler is sent in a JSON[21] map. The structure of the map is kept as simple as possible. The general structure is also explained in the 'how to' documentation. I showed the handler to four developers and one software architect at work. Their feedback concerning the request complexity was very positive. Comparing the usage of the handler with manually configuring one or multiple HANAs, the handler automates almost every step that otherwise would have been taken manually. The most common way to configure a HANA manually is the following. The SystemDB password for the right landscape must be obtained from the Vault. After that for each parameter that should be configured a SQL statement is sent and the password is entered each time. The other solution is to enter the HANA, which means the container. The INI file that contains the parameter that should be configured must be found in the filesystem. After all requested parameters are adjusted a tool must be used to trigger the reconfiguration in the SystemDB. This tool is already available in the HANA. Both ways consists of multiple steps and the execution time grows proportionally with a growth in the number of parameters that should be configured and also with the size of the group of HANAs that should be configured. The steps needed to use the handler are constant and only consist of getting the IP address and sending the request. Therefore, I conclude that with the introduction of the handler the workload for the developers and operations team will clearly decrease. Furthermore, the handler also logs meaningful events in the configuration process and for each request, an HTTP response informs about the configuration process. The response states that the configuration was executed successfully or otherwise that the request is invalid or errors occurred. Using this information, the user can fix the request or inform the maintainer of the handler about any issues while providing meaningful information.

Performance

The handler was designed not to execute redundant API calls. This resulted in higher complexity of the implementation. The purpose of functions is in some cases unclear. An example of that is a function that checks if a HANAPod with a specific HANA ID exists. See function `Exists` in the HANA client code snippet of the Appendix. This function would usually return a boolean. As the function executes an API call to retrieve all HANAPods, the function would also return an error object for debugging purposes. In this implementation, the handler also returns an object containing the HANAPod data if the HANAPod exists. The function is basically the combination of a check and a getter function. This programming style reduces the number of the API calls, however, it does make the programming code harder to understand which is a disadvantage for other developers who did not participate in the implementation. Another aspect of the performance of the handler is calculations. The handler does not execute a lot of calculations, but it does contain many for-loops to search in arrays for the right object. For an example of that see functions `hanaPodList` and `foundHanaPodInPodList` in the HANA client code snippet of the Appendix. Some of these loops would be made obsolete, if the API calls would be more detailed. Cur-

[21] https://www.json.org/json-en.html

rently, in cases where the handler wants to obtain a specific resource instance it does not send an API call to get one instance, but executes a list request to obtain a subset of the resource instances which contains the desired one. This is due to the fact that the `get` function that is supposed to return only one specific resource instance does not work with the data that is available. If this would be solved, multiple for loops could be removed which would improve the performance and decrease the CPU usage of the handler. Nonetheless, the current performance of the handler that it shows in a development landscape is already acceptable. This opinion was also supported by developers that I presented the handler to in an online meeting.

Security
Out of the three most important quality criteria, the implemented handler fell short in the security aspect. Design decisions were made carefully to build a handler that holds a minimum of critical permissions. An example of that would be the decision about the HANA configuration mechanism. The decision was made between the possibility to connect to the SystemDB via SQL or entering the container and adjusting INI files. For the second option, the handler would have to be granted the permissions to exec every pod that is running in the same cluster. It is not possible to grant the handler only permissions to exec only HANA pods due to the nature of the RBAC method. Granting the handler permissions to exec every pod would have created a great security risk and also more complexity in the configuration process. Therefore, the handler was designed to fulfil its purpose while having as little access rights as possible. However, the real security concern is not the handler itself but the access to it. Currently, there is no method introduced that restricts the usage of the handler. Everyone that has knowledge of the external IP address, on which the handler is exposed, can send a request to the handler. It is not checked if the requester actually is allowed to use the handler in any way. This is a concern that was not solved in the handler due to the limited time. This means, it is crucial that the usage of the handler must be restricted in a way, such as explained in Section 6.1, before it is introduced in a productive system. Another security risk that should be tackled in the future is that not all parameters are supposed to be changeable by every or any user. Some parameters result in e.g. more storage usage of a HANA. This is a costly consequence and should only be executable by the operations team and not every developer. In conclusion to the security aspect, though the handler is acceptable concerning the permissions that it holds, the access to it not limited enough.

Scalability
The handler saves the data in its memory that is needed for each HANA to roll back. This has the consequence, that it is not sufficient to run multiple replicas of the handler to scale up and down with the number of requests that are sent. Furthermore, the automated HANA configuration update process is based on the functionality of watching the events of the ConfigurationProfiles. If multiple replicas of the handler would be running while a profile was deleted or updated, this would result in a situation were every replica would try to roll back or update the configuration of the same HANAs. That means redundant and CPU resource-consuming work. However, the handler is designed to run in every landscape that hosts HANAs. This also means

that the handler scales up with the numbers of landscapes. A production landscape runs usually not more than a thousand HANAs at a time. Development landscapes run way fewer HANAs. For the handler to handle up to thousand entries as a return of an API call is not much computational power. A use case for this would be retrieving all HANA pods to find the HANA that shall be configured. Therefore, the scalability is in my opinion currently sufficient. If the scalability has to be improved this could be done by executing configurations of HANA groups concurrently.

In summary, the handler is scalable and thus cloud-native developed.

8.3 The Improvements

The improvement approaches would have positive and negative influences on the handler concerning the quality criteria. One of the most common tension fields is the relation between comfort and security. In this case, the comfort is equal to the usability criteria. The introduction of the restricted use of the handler increases the security in every realization, but can also decrease the usability aspect. The most comfortable and secure approach that was explained in Section 6.1 is to use the permission management framework that is already used in the company. This adds only one additional precondition that the user is associated with the right ServiceAccount. To do so, he/she requests it in the framework. Fortunately, every employee of the department has already used the framework and has, therefore, knowledge on how to send permission requests. The usage of the handler afterwards is not influenced in any obvious way. The approach is ideal as it does not decrease the quality of the user experience and uses a tool that is already widely used in the company. In most cases, it is preferable to reuse existing tools instead of introducing more complexity by using other tools.

The improvement approaches for the rollback feature all have the advantages, that the default configuration data is saved independently from the handler and also resolves one of the two mentioned issues why the handler cannot be duplicated. Each approach has also its disadvantages. As each approach is based on saving the data independently from the handler, the handler will have to obtain the data via API calls, with the exception of persistent volumes. This will increase the mandatory number of API calls and will influence the performance. Which approach is the most suitable depends on the work ethic between teams and the memory usage that is defined as reasonable for this feature. The rollback mechanism for the ConfigurationProfiles is rather simple. The only big issue is that in the worst case, this method doubles the memory usage of each profile. Additional work would have to be done in the handler to add the new request type, rollback. However, the workload is not too high as the handler is designed to be flexible and extensible. This was done by implementing it in a modular way by using separate packages as interfaces for different logical sections. Also, in this case, a lot of code can be reused.

The presented approach to add a safety net to the profile event watcher is simple and as a similar method is already used for the profile deprecation feature code can be adjusted and reused for this extension. This seems to be a simple approach, but might end up being the reason for undesired configurations, as explained in Section

6.4. The last improvement that is discussed is to introduce the concurrent configuration mechanism. The evaluation of the prototype also referred to this improvement. If the handler is frequently used to configure HANA groups of great size, the concurrent configuration process would improve the responsiveness of the handler efficiently.

9 Conclusion

> Good morning, and in case I don't see ya, good afternoon, good evening, and good night!
>
> Truman from The Truman Show (1998)

This thesis dealt with the design and implementation of a service that automates the configuration management of HANAs running in an existing Kubernetes environment. Yet, the main value does not lie in the implementation or design of the handler, but in the educational aspect of this project. Viewing the workflow, it can be noted that the first phase of brainstorming the mandatory functionalities and searching for possible challenges and issues did result in a positive outcome. Some might argue that also without the first phase, no bigger issues could have appeared. However, I already started to exclude some concepts in this phase of critically evaluating concepts that I created. This avoided situations that might have created some complications. The fact is that while implementing the handler, no critical issues appeared that blocked the development massively or risked not meeting the deadline for the bachelor thesis. Furthermore, to use an iterative development process turned out to be a wise choice as this made it possible to continuously check if the handler is feasible and allowed me, after each extension of the handler was fully implemented, to decide if I want to stop the development of the handler for the thesis. Using another process, such as a waterfall-based method, could have resulted in a handler that can only function if every planned feature is implemented. Therefore, I conclude that the workflow used in this thesis was efficient. Referring to the rest of the thesis, the prototype itself is a functional realization of its purpose. It does show room for improvement, but has the potential to be a helpful tool for the developers and operations team of the company. If desired, this handler could also be extended by other instance type adapters to also configure other instances besides HANA. Not all concepts are designed in a cloud-native way, an example is the current issue of not being able to scale the handler up and down. However, the handler is a prototype that can be further improved to be a service that can be used in production landscapes. Single features or modules can be simply replaced, in most cases, thanks to the usage of interfaces.

Finally, the most important piece of knowledge that I could obtain from this thesis is the following. When designing and implementing a tool or service which will be integrated into an existing environment, it is efficient to first examine the existing environment and the services and resources that are available in it. Doing this, it can be avoided that much effort and time is put in building resources or concepts that are already available in a slightly different fashion. For example, to create a concept to restrict the access to the handler even though the company already uses a permission management framework that has proven its worth and can also be used to realize the access restriction. The second advantage is that you do not spend much time on issues that resulted in dependencies within the system like explained in Section 7. In summary, in a big company with a complex environment, it is wise to spend

the time to analyse the environment, before designing the service that shall run in it, to know what dependencies to fulfil and to make use of synergies positively, when possible.

Bibliography

[1] What is sre (site reliability engineering)? https://www.redhat.com/en/topics/devops/what-is-sre. (accessed: 06.10.2020).

[2] Anit Buehrle. Kubernetes FAQ - How can I route traffic for Kubernetes on bare metal? https://www.weave.works/blog/kubernetes-faq-how-can-i-route-traffic-for-kubernetes-on-bare-metal. (accessed: 23.06.2020).

[3] Horacio Gonzalez. Getting external traffic into kubernetes – clusterip, nodeport, loadbalancer, and ingress. https://www.ovh.com/blog/getting-external-traffic-into-kubernetes-clusterip-nodeport-loadbalancer-and-ingress/. (accessed: 22.06.2020).

Glossary

Cluster is a a collection of virtual and/or physical machines behaving as one system.

Cluster Nodes is a single virtual or physical machine included in the cluster.

ConfigMap (Kubernetes resource type) is basically a map storing string keys and string values.

Helm is a Kubernetes package manager.

Helm Chart is the package format of Helm.

Kubernetes is a system to orchestrate containers.

Pod (Kubernetes resource type) is a collection of containers that are restricted to run on the same cluster node.

Rollback is the mechanism to return e.g. a configuration to a previous state.

A Appendix

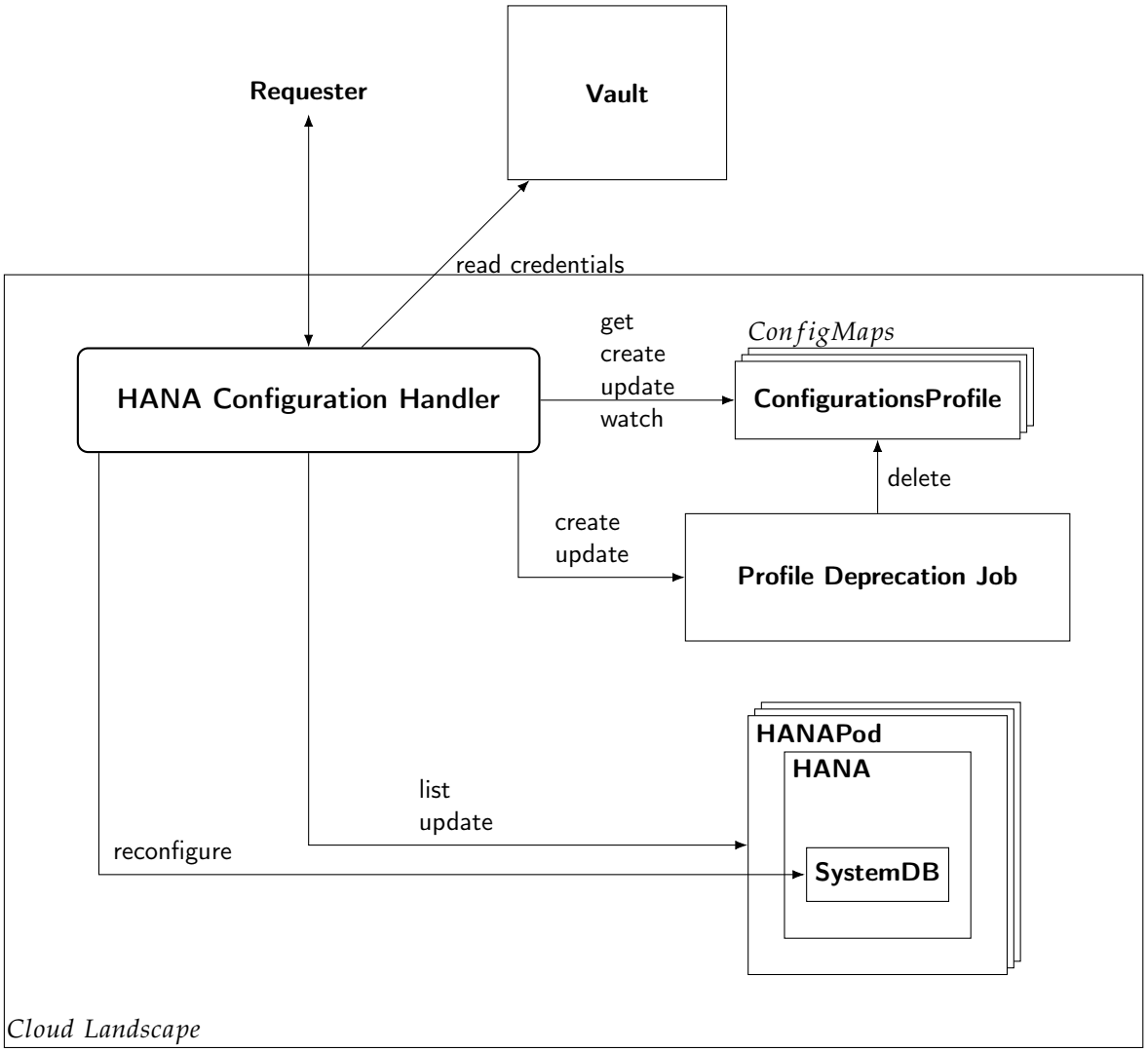

Figure 5: HANA Configuration Handler Architecture

Source code snippet of HANA client[22]:

```
package hana
...

// checks if HANAPod exists and returns it
func (hana *HANA) Exists(instanceID string) (bool, map[string]corev1.
    Pod) {
  podList, err := hana.hanaPodList()
  if err != nil {
    return false, map[string]corev1.Pod{}
  }

  ok, pod := hana.foundHanaPodInPodList(podList, instanceID)
```

[22] from hana.go

```
  if !ok {
    return ok, map[string]corev1.Pod{}
  }

  return ok, map[string]corev1.Pod{instanceID: pod}

}

// searchs for HANAPod 'instanceID' in list of HANAPods and returns it
func (hana *HANA) foundHanaPodInPodList(podList []corev1.Pod,
    instanceID string) (bool, corev1.Pod) {
  for _, pod := range podList {
    if compareInstanceIDAndPodName(pod.ObjectMeta.Name, instanceID) {
      hana.logger.Info("Pod for instance ID " + instanceID + " does
          exist :D")
      return true, pod
    }
  }
  hana.logger.Info("Pod for instance ID " + instanceID + " does not
      exist :O")
  //should this really be an error
  return false, corev1.Pod{}
}

// returns all HANAPods
func (hana *HANA) hanaPodList() ([]corev1.Pod, error) {
  label := "HanaInstallation"
  pods, err := hana.clientSet.CoreV1().Pods("").List(context.
      Background(), metav1.ListOptions{LabelSelector: label})
  if err != nil {
    return []corev1.Pod{}, err
  }
  podList := pods.Items
  if len(podList) == 0 {
    hana.logger.Info("No pods with label " + label + " were found :(")
    return []corev1.Pod{}, errors.New("No pods with label " + label +
        " were found")
  }
  return podList, nil
}

// HELPER
// checks if HANAPod belongs to HANA 'instanceid'
// pod names is the combination of the HANAID and a random generated
    suffix
func compareInstanceIDAndPodName(podname, instanceid string) bool {
  return strings.HasPrefix(podname, instanceid)
}

...
```