
**Development of an application programming interface for launching and
managing dynamic networks of distributed software**

BACHELOR'S THESIS

for the degree

BACHELOR OF ENGINEERING

of the course Informationstechnik

at the Baden-Wuerttemberg Cooperative State University Mannheim

by

Sebastian Nocke

Submission on September 16, 2022

Processing Period:	01.07.21 – 17.09.21
Student id, course:	9944135, TINF19IT1
Department:	Institut für Softwaretechnologie: Intelligente und Verteilte Systeme
Apprenticing company:	Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR)
Company's supervisor:	Niklas Först
University's reviewer:	Prof. Dr. Holger Hofmann

Declaration

I hereby assure you that I have written my bachelor's thesis on the

SUBJECT

Development of an application programming interface for launching and managing dynamic networks of distributed software

independently and that I have not used any other sources and aids than those indicated.

I also assure you that the electronic version submitted is the same as the printed version.*

* if both versions are required.

Köln, September 16, 2022

Contents

Listings	1
1. Introduction	3
2. Preliminaries	4
2.1. Distributed computing	4
2.1.1. RCE	5
2.2. Containerization	7
2.3. Kubernetes	8
2.3.1. Volumes	9
2.4. Microservices	10
2.5. REST API	11
2.5.1. Requests	12
2.6. Quarkus	13
2.7. Cloud system environment	15
3. Requirements and interface definition	17
3.1. Requirements	17
3.1.1. Namespace management	18
3.1.2. Catalog management	18
3.1.3. Network management and interaction	19
3.2. Interface	20
3.2.1. Example API calls	21
4. Concept of the API	27
4.1. Splitting functionality	27
4.1.1. Hierarchy	29
4.2. Kubernetes setup	31
5. Implementation	33
5.1. Catalog-Service	33
5.1.1. Program catalog	35
5.1.2. File catalog	37

5.1.3.	Persistent storage	39
5.2.	Instance-Manager	40
5.2.1.	Instance creation	41
5.2.2.	Specific-Instance-Manager	41
5.2.3.	Communication	43
5.3.	Frontend-Service	45
5.3.1.	Catalog forwarding	45
5.3.2.	Network forwarding	47
5.3.3.	Namespace forwarding	47
6.	Conclusion	49
6.1.	Current state	49
6.2.	Future prospect	50

List of Figures

2.1. View of an RCE workflow [10]	6
2.2. Network View of RCE [10]	7
2.3. Components of Kubernetes	9
3.1. Example GET request	23
3.2. Example POST request	24
3.3. Example POST request	25
3.4. Example DELETE request	26
4.1. Diagram of the service structure	30
4.2. Visualization of the Kubernetes setup	32
5.1. Catalog-Service UML class diagram	34
5.2. Instance-Manager UML class diagram	42
5.3. Frontend-Service UML class diagram	46

Listings

2.1. Quarkus simple REST handler example	14
2.2. Quarkus complex REST handler example	15
5.1. Stream operation example	36

Abstract

Creating networks of distributed software is a complex laborious task. This is due to the fact that many aspects have to be manually configured. Each instance of a program has to be configured and created. Tools like *Kubernetes* make it easier to create such networks as it provides the ability to create networks on a single computer. *Kubernetes* however does not reduce the amount of manual work that has to be done by a user. Additionally the communication with instances that are deployed inside a network can be tedious as not every program running in such a network may be exposed for external access.

To tackle this problem create an API that reduces the amount of work that has to be done by a user to create distributed software networks. The goal of this API is to make it possible to configure, create, and manage networks with minimal effort. Furthermore the API is supposed to make the process of running such networks more safe by strictly controlling which programs can be run via the API.

1. Introduction

The development of distributed software involves several difficulties. One of these difficulties is testing the software and ensuring the intended behavior is met. Methods of testing that are used on non distributed software can be used for testing distributed software, but will not cover some aspects of the software. The distributed aspect of the software needs special attention in the testing process.

Distributed software that is deployed to a network of computers that are interconnected. In order to ensure the intended behavior of a software in a network is given, it has to be tested. However this testing is not trivial, as a network of the software, that is to be tested, has to be created and a test workload has to be run. Manually creating such networks is a tedious and time consuming task that can be virtually impossible for software, that is designed to run in networks of hundreds of instances. The amount of manual effort that is required for the testing in a networked environment implies, that this kind of testing is performed less often than other testing methods.

The example of distributed software testing shows that an API for simple and automated creation of these networks has real application. In addition to the creation of distributed software network the API will be able to interact with the instances in a network. These capabilities ensure, that the API can be used to create automated tests for distributed software.

We implement an API that addresses the previously mentioned aspects. This is done in the scope of this bachelor thesis.

2. Preliminaries

This chapter describes the preliminaries for this work. In Section 2.1 we introduce the concept of distributed computing followed by a presentation of *RCE* in Subsection 2.1.1, which is the software, that will be used to test the API. We establish the principle of containerization in Section 2.2 before showing the software *Kubernetes* in Section 2.3. Section 2.4 will describe the idea of microservices. Next we present the *Quarkus* framework that we use for the implementation in Section 2.6. Following this we introduce the concept of a *REST* API in Section 2.5. At last we present the server infrastructure that is used for deployment of the project in Section 2.7.

2.1. Distributed computing

Distributed computing is done by using software that runs on a distributed system. Distributed systems are a collection of processors that are connected via a network. Some common features of a distributed system are no common physical clock, no shared memory, geographical separation, as well as autonomy and heterogeneity [9]. The missing of a common physical clock is an elementary component of a distributed system and can introduce disruption and asynchrony to the system. The fact that the system does not share memory leads to information sharing via messages. This approach is also well suited to circumvent the lack of a common system clock. Geographical separation of processors is a common property but is not necessary for a system to be seen as a distributed system. Some distributed system can cover vast distances while others have all of their processors within meters of each others. Autonomy and heterogeneity describe the fact, that the individual processors of the

system can differ, e.g., do not have the same speed. This also extends to the topic of operating systems as they too can be different. The execution of a program on such a system that uses more than one processor is called distributed computing.

The reasons for using distributed computing include resource sharing, enhanced reliability and increased performance to cost ratio [9]. Distributed systems enable resource sharing by making software or data available that is located on other processors. The usage of distributed computing enhances the reliability because resources and programs can be provided by replicas in case the original source is not accessible. This behaviour increases the availability of resources, ensures the integrity of them, and provides fault-tolerance. The effectiveness of these characteristics depends on the needs and the implementation of a distributed system. For example a system that has to provide information where its authenticity is vital while access to the data is not always required, the data integrity and fault-tolerance may be higher prioritized than the availability of the system.

2.1.1. RCE

RCE (Remote Component Environment) is a workflow-driven integration environment. It is created for design and simulation of complex, multi-disciplinary systems, such as aircraft or ships. The flexibility is achieved by allowing users to integrate their own design and simulation tools. *RCE* allows the creation of automated workflows that consist of the integrated tools. Workflows may be executed on a distributed network of *RCE* instances. An example of a workflow is shown in Figure 2.1 For example the design and evaluation for a new type of airplane with different working groups with different know-how like aerodynamics or aeroelastics can be supported by a workflow with different simulation tools to simulate the airplane as a whole. In order to manage such a project, individual tools are chained together so that simulation results are used as input data for other simulations. *RCE* is an open-source software that was developed by the Fraunhofer SCAI and German Aerospace Center (DLR) from 2006 through 2010. Since 2010, DLR has been developing the software on its own. It is based on the Eclipse Rich Client Platform [3] [10].

2.1. Distributed computing

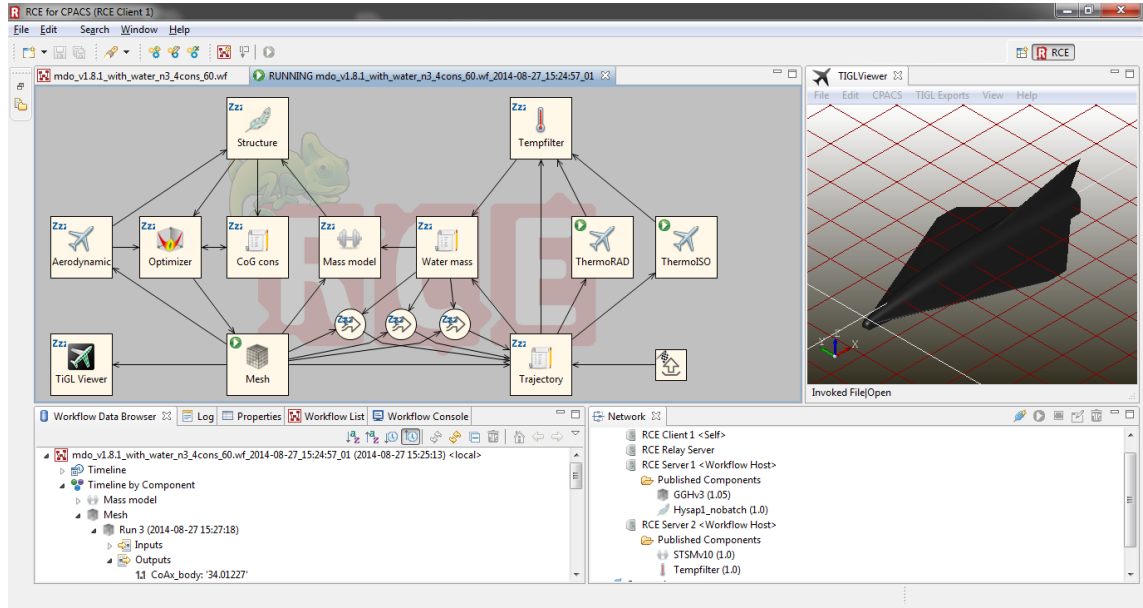


Figure 2.1.: View of an RCE workflow [10]

RCE has a wide range of features, but only the ones that are relevant to this work will be presented. This feature is the distributed execution of a workflow. For this multiple *RCE* instances have to be connected in a network. The instances of a network can then publish tools that can be used to construct workflows that use these. In Figure 2.2 we show a screenshot of the *Network View* which is a view in *RCE* that shows connected instances and the workflows that they publishing to the network. When a user runs a workflow that incorporates tools published by other instances the tool will be run by the instance publishing it. This results in *RCE* executing a workflow in a distributed system. *RCE* will orchestrate the execution of the tools on different instances. This includes starting a tool only when all its inputs are provided and collecting its output in order to pass it to a tool that depends on that data.

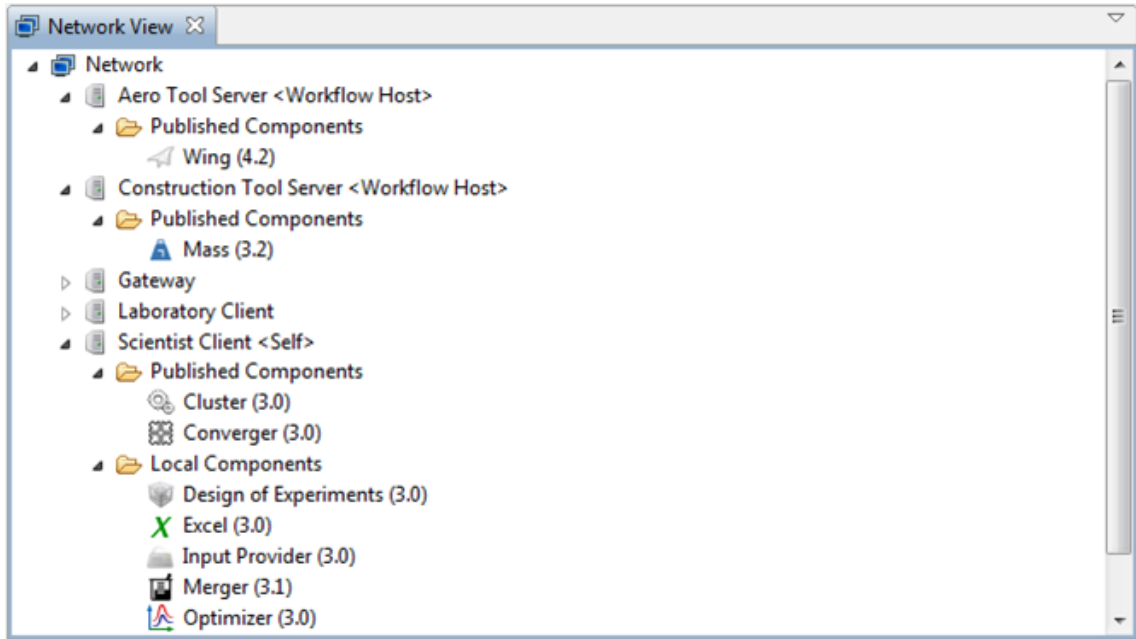


Figure 2.2.: Network View of RCE [10]

2.2. Containerization

Containerization provides a modern alternative for virtualization. In the scope of cloud computing virtualization is used to manage system resources and enable the execution of different software while restricting access to other processes. Containerization is becoming a replacement for virtualization in the area of cloud computing [11]. Containerization offers most of the benefits of virtualization while providing additional benefits. Virtualization is based on the usage of a hypervisor that divides the resources of the system and runs virtual machines using a specified amount of the total system resources. A virtual machine provides an entire operating system as well as the program and its dependencies. Containerization works by running independent containers for each program. Containers do not provide the kernel of an operating system because the kernel of the host operating system is shared with them. A container runtime is used for managing containers on a system. Resources of the host system are provided on demand by the container runtime, opposed to the approach of virtualization, where resources are assigned statically by the hypervisor.

2.3. Kubernetes

The container runtime, also known as a container engine, creates a running container from an image file. Such an image contains the application itself, as well as its dependencies. As a result a container image is loaded with all the data that is necessary for it to be run by the container engine.

Container images are defined by a layered structure. Each layer is an image on its own. Every container image starts with a layer of binaries and libraries for its operating system. On top of that any amount of additional layers can make up a container image. Such a layer could for example be a specific *Linux* distribution or the image of an application like an Apache server. When creating a new container image the included layers can only be read. The top most layer is mounted as the writable container as shown in [11]. Creating a container image by building a stack of read-only image leads to a lightweight approach, as images can easily be changed.

One of the tools that provide the creation as well as the execution of container images is *Docker* [4]. *Docker* provides a container runtime, a tool for the creation of container images, and *Docker Hub* which is a database for sharing and pulling container images.

2.3. Kubernetes

Kubernetes is an open-source container orchestration system for automating software deployment, scaling, and management. It was originally designed by Google in 2014 before giving it to the Cloud Native Computing Foundation. Originally *Kubernetes* exclusively worked with *Docker*, but since 2016 it can interface with different container runtimes. *Kubernetes* is designed to be highly scalable by running on clusters using a master called the *Kubernetes* control plane, which interacts with the workers called Nodes of the cluster [2]. The structure of a *Kubernetes* cluster can be seen in Figure 2.3.

Kubernetes wraps the containers that are spawned in Pods. Pods consist of one or more containers inside it. Containers running in the same pod share their storage and network resources. Containers should be in the same pod, if they run tightly

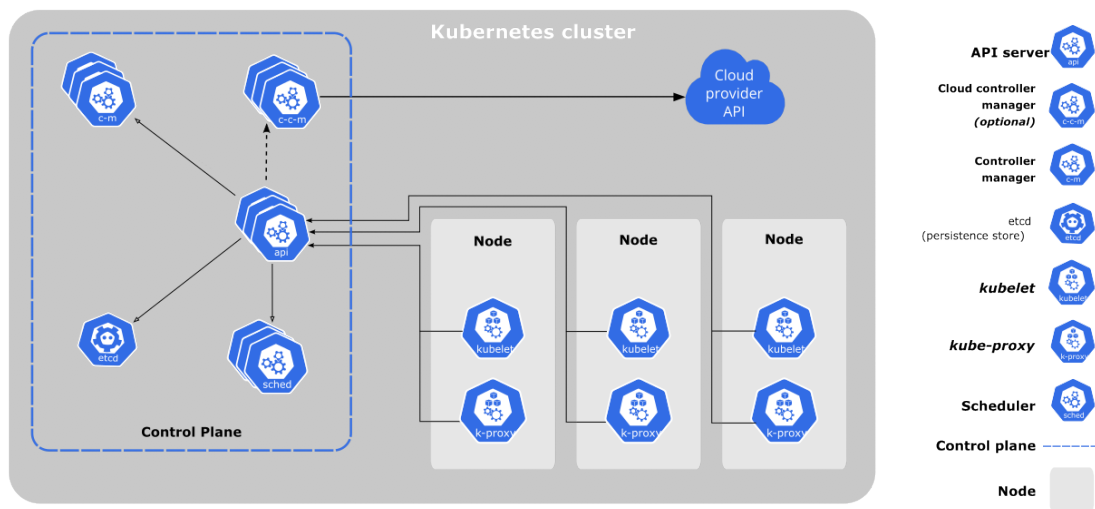


Figure 2.3.: Components of Kubernetes

coupled i.g. containers that depend on each other to fulfill their purpose. In order to run containerized applications using *Kubernetes* the user defines a deployment and applies it to the cluster. Deployments specify all the information that *Kubernetes* needs to know in order to create and manage the applications. These information include the container image of the applications, crash handling, exposing of ports, replicas and many more.

Kubernetes can create isolated groups of applications by using namespaces. Many resources like deployments for example have a namespace-based scope. As a result, such a resource must have a unique name inside the namespace, in that it exists, while a resource with the same name can be created in another namespace. Namespaces can be used to separate projects inside a cluster.

2.3.1. Volumes

Files that are written on disk by a container running inside a *Kubernetes* pod are not saved persistently. In case the pod crashes or is restarted in any other way, it returns to a clean state. To save data persistently, a storage that outlasts the lifetime of a pod is necessary. *Kubernetes* provides this kind of storage via volumes. A volume has to

be claimed using a *PersistentVolumeClaim*. The creation of a *PersistentVolumeClaim* binds a specified volume to it. Volume claims are namespace-scoped. A volume claim allows pods that are created in the same namespace access to the volume it is bound to. The NFS-volume (Network File System-volume) is a specific type of volume. It uses a network attached file system to provide persistent storage. Usage of an NFS-volume requires a preconfigured network attached file system to be accessible from the *Kubernetes* node.

2.4. Microservices

The microservice architecture does not have a universal definition. For this work we define a microservice as a service that has a single task and is capable of doing that task on its own. As the name suggests most microservices are small compared to other service architectures. This is a result of tackling one task per microservice. However a microservice is not defined by a small size of its binaries or its interface as some microservices can be larger given a bigger task to be handled by it. Microservices are often deployed as a cluster of multiple microservices that interact with each other using messages. Messages of microservices do not have a specific format, however the following a standard is recommended to increase the interoperability. In most real world use cases a single microservice is not capable of solving the requirements for that use case. This is the reason for microservices being deployed in a group with each of them solving a part of the total requirements. The microservice architecture addresses flexibility as well as scalability. Flexibility is improved, because a single microservice in a group can be modified and updated without having an influence on the other microservices. A microservice itself should be designed to easily be scalable thus resulting in the entire service structure being easily scalable [12].

2.5. REST API

A *REST* API is a special kind of API. It provides the interface by exposing endpoints that can be accessed by users via requests. We describe requests in detail in Section 2.5.1. As the name implies, this kind of API follows the constraints and philosophy of *REST* (Representational State Transfer) [5]. In the following we will present aspects that define *REST* APIs.

A *REST* API depends on the server-client architecture. The client-server architecture is based on a server providing service that users can interact with. Users start the interaction between themselves and the server by sending a request to the server. The server and more specifically the services that the user addresses will then either accept and handle the request or refuse it. Due to the fact that users can initiate communication with the server at any chosen time the services has to be accessible and thus running at all times. This is often realised by the service running in an endless loop waiting for requests from clients.

REST APIs are stateless which means that the server does not store a session state. Statelessness is an extension to the previously described server-client architecture. Each request that a user issues to the server has to contain all the information that is required for it to be executed. Users have to manage the session state on their own and cannot make usage of a context that is stored on the server. The usage of a stateless design has implications on multiple properties of the API. The visibility of the API is improved because the contents of a request fully describe the intention of the request. Reliability as well as scalability are improved because request are not required to be handled by the same instance of the service. This is a result of absence of a session state in the service. In the case of a fault of a service such as a crash there is no client specific data that has to be reconstructed. For the same reason the service can be supplied by multiple servers where requests are distributed by e.g. a load balancer thus creating a simple foundation for scalability.

A *REST* API can implement caching in order to improve the efficiency of network usage. The cache acts as an mediator between the user and the service. Services that use caching have to mark data of responses as either cacheable or non-cacheable.

2.5. *REST* API

Data that is marked as cacheable can be cached in a client side cache for following equivalent requests. Caching provides the ability to lessen the amount of requests that the services has to handle while introducing the risk of inaccurate and outdated data. Consequently the usage of a cache has to be used implemented in a balance of performance and efficiency gains opposed to the risk of the usage of imprecise data.

REST APIs make use of a uniform interface. This aspect of the API follows the concept of generality. It leads to a looser coupling of the implementations and the services they provide. The decoupling of the implementation from a services improves the ease of replacing the implementation. Beside these advantages the usage of a uniform interface can lead to a lower efficiency as data has to be formatted to the uniform format instead of using a service specific format that can be optimised for its needs. As *REST* is optimized for common cases of the Web and its hypermedia structure it does not provide an optimal interface for every task. The uniform interface of a *REST* API is defined by the identification of resources, manipulation of resources through representations, self-descriptive messages, and hypermedia as the engine of application state [5].

2.5.1. Requests

Users send requests to a *REST* API for creating, retrieving, updating, and deleting resources. This is implemented by the usage of different method types. The method types for requests are listed in the following [6].

GET Retrieval of a resource without modification in any way: Upon successfully execution a GET request produces a response with the status code 200 with its response body containing the requested data. A GET request can fail due to the requested resource not being found which will return status code 404 or due to an incorrectly formed request.

POST Creation of a new subordinate resource: A POST request creates a new resource that is subordinate to the path defined by the request. On successful completion a response with status code 201 should be returned that also contains a description of the resource as its body.

PUT Primarily updating an existing resource or creation if it does not exist: If a PUT request creates a new resource it returns a response with status code 201. In case such a request is used to update a resource the status code of the response should be 200.

DELETE Deletion of a resource: Successfully deleting a resource returns status code 200 in case the resource has been deleted or 202 in case its deletion has been queued. Calling a DELETE request on a resource that does not exist results in a response with status code 404.

PATCH Partially updating a resource: A PATCH request is used to update part of an already existing resource and returns a response with status code 200. PATCH requests will fail with status code 404 if the resource whose updating is requested does not exist.

2.6. Quarkus

Quarkus is a framework build for the Java programming language. It is developed under the open source Apache License version 2.0. The purpose of *Quarkus* is to move Java away from its typical monolithic application structure into the cloud environment of containers and *Kubernetes*. Using *Quarkus* developers can create cloud-native applications with native *Kubernetes* support [8].

Quarkus aims to make the development of cloud-native applications efficient and easy. This is done by providing tools, libraries, and extensions. *Kubernetes* is tightly integrated into *Quarkus* making it easy to deploy applications into a *Kubernetes* cluster. Through this integration the generation of *Kubernetes* resources can be done automatically. Container images of applications developed using *Quarkus* can also be automatically built and deployed. Developing applications is supported using two types of coding styles. The imperative coding style that is most common when using the Java programming language. Imperative programming is a style of programming where instructions are defined in an order. Execution of a program using the imperative programming style is done by the order in which the instructions

are defined. The other programming style that is supported is reactive programming. Reactive programming is a declarative style in which handlers for events are defined. The concept builds on an application running and waiting for events in order to trigger the corresponding handlers. Event handlers are executed asynchronously which enables the application to listen to new events while an event handler is executed. The events for reactive programming in *Quarkus* are REST requests. A simple example of a REST request handler in *Quarkus* is shown in Listing 2.1. Listing 2.2 depicts a more complex handler configuration.

```
1  class ServiceProvider {
2
3      @GET
4      @Path("/v0/simple")
5      public Response restHandler() {
6          return Response.ok("Simple response").build();
7      }
8  }
```

Listing 2.1: Quarkus simple REST handler example

The code example shown in Listing 2.1 would provide a handler for *REST* GET request at the “/v0/simple” endpoint. This request does not receive any sort of parameters and will always return a response with the status code 200 and contains the message “Simple response”.

The complex example shown in Listing 2.2 is an implementation of a *REST* DELETE request. The endpoint for this request handler is put together by two separate *Path* annotations. *Path* annotations that are specified for a class are combined with annotations for methods inside that class thus resulting in the endpoint for the request being “/v0/complex/resourceName” where *resourceName* is a parameter of the request. Parameters are made accessible to the method by the *PathParam* annotation. For the sake of keeping the code example short the two methods *resourceExists* and *deleteResource* are used but their implementation is not shown. As the names of the methods state the first one would check if the resource specified

by the parameter exists while the second one would delete the resource. The response contains the status code 200 and the message “Resource deleted” in a *JSON* formatted entity as long as the resource exists. If however the resource does not exist the status code is 400 and the message “Resource does not exist” is sent.

```
1  @Path("/v0")
2  class ServiceProvider {
3
4      @DELETE
5      @Produces(MediaType.APPLICATION_JSON)
6      @Path("/complex/{resourceName}")
7      public Response restHandler(@PathParam("
           resourceName") String resourceIdentyer) {
8          if (resourceExists(resourceIdentifyer)) {
9              deleteResource(resourceIdentifyer);
10             retrun Response.ok("{ message: \"Resource
                deleted\" }").build();
11         } else {
12             return Response.status(400).entity("{ message:
                \"Resource does not exist\" }").build();
13         }
14     }
15 }
```

Listing 2.2: Quarkus complex REST handler example

2.7. Cloud system environment

In this section we will present the server cluster that is used for the deployment of this project. These servers are internal to the DLR network. They can only be accessed from that network.

The cluster consists of eleven computers that are deployed together. Only one of these computers is exposed and acts as an admin and access point to the server cluster. In order to gain access to the rest of the cluster, it has to be used as a jump server. It is also the only node of the cluster, that has access to the internet. Thus the admin node is used to install software onto the nodes that do not have internet connectivity. The IT-infrastructure forbids access to the internet on the other nodes as a part of security decisions.

The ten servers that are disconnect form the internet are configured differently and are named correspondingly. Seven nodes are setup for CPU compute power, each of them being equipped with four CPUs for a total of 64 cores and 128 threads. The nodes are called *cloud-cpu1* through *cloud-cpu7*. The remaining three nodes are configured for GPU compute power, large amounts of storage, and fast storage which are called *cloud-gpu*, *cloud-storage-big*, and *cloud-storage-fast* respectively. These nodes are configured with hardware according to their designated tasks. The two nodes *cloud-storage-big* and *cloud-storage-fast* are configured to provide network attached storage to the other nodes of the server cluster.

Orchestration and configuration of the server cluster is done using *Ansible* [1]. It is used on the admin node in order to install software on the other nodes to enable the rest of the cluster to be disconnect from the network. We will not go into detail about *Ansible* because it is not used directly for this work.

3. Requirements and interface definition

In this chapter we will discuss the requirements of the API in Section 3.1. Following this we will define the interface in Section 3.2.

3.1. Requirements

In this section we define the requirements of the API and establish the scope of this project. We have elicited these requirements via unstructured interviews with prospective users of the interface. Due to the time constraints of this project, we have opted to omit a more structured approach to requirements engineering. Moreover, since the resulting interface will not be publicly available it will be rather straightforward to adapt the interface to future additional requirements. We split up the requirements for this project. We do this in order to tackle separate topics one by one. The topics are namespace management, catalog management and instance management. These topics cover the creation of “frames” for the networks of software to live in, making software and files available to these frames, and executing the software in respectively. Each of the following subsection will address one of these topics. We address them separately, because they tackle different problems of the API and will be implemented separately.

In the following sections we will talk about programs, services, and instances. We now define the meanings of these terms for this work. A program is a software that

can be run by the API. Services are the pieces of software that make up the API itself. Instances describe a service or program when it is running.

3.1.1. Namespace management

This subsection addresses the management of the *Kubernetes* namespaces that the API will use to host the networks of distributed software.

Namespaces and their management are crucial to the API, because they enable the creation of separated networks. Separated networks are necessary in order to create multiple networks at the same time. Namespaces will have to be created and deleted by requests. Additionally a mapping of the managed namespaces to reference ids must be implemented. A reference id is name that is optionally set by a user when creating a namespace. This mapping will help avoid naming conflicts and will help with automated workflows by referring to namespaces using the reference id. Furthermore the ability to assign ingress ports to namespaces will have to be supported. Ingress ports allow external access to applications running inside the namespace. The management of namespaces requires using the *Kubernetes* API.

3.1.2. Catalog management

This subsection describes the requirements for managing the catalog. The purpose of the catalog is providing data that is accessible for all namespaces. The catalog is supposed to provide a layer of security. It does so by providing the only way for programs and files to be added into networks via the API. For this to be effective, the namespaces that are managed by the API should not be interacted with manually. The catalog can be divided into the two topics of files and programs. We will start with explaining the basic goal of the catalog itself. Following that we will describe the requirements of the file catalog with the program catalog being discussed afterwards.

File catalog

The purpose of the file catalog is to provide a collection of files that can be used by programs, that will be running via the API. To keep the effort for this project in a suitable scope the structure of the file catalog is kept simple. All files contained in the catalog will be exposed at once to a program. The files that are provided via the catalog should not be modifiable by applications running in the networks of the API. It is also not permitted that an application running via the API can write files into the folder of the catalog.

Program catalog

The API uses the program catalog to control the programs that can be in its namespaces. Each entry for a program in the catalog contains multiple information. These comprise the name of the program that will be used to create an instance of it and the path to the container image for that program. Additionally each entry contains information about the interactions that the program provides, which are shell commands, ssh commands, and *REST* API. The entire information that is stored in the program catalog uses the *JSON* format. Finally, each program needs a service that takes care of the creation of the instances. The path for the container image for that service is also part of each entry in the catalog.

3.1.3. Network management and interaction

The API creates namespaces that will further be used by the API to create networks of distributed software. Users only interact with the namespaces indirectly through the API. Users will add instances of of distributed software via the API to its namespaces. Using this part of the API, users can create networks of distributed software dynamically without any static configuration. As previously mentioned in Section 3.1.2 only programs that are registered in the program catalog can be instantiated inside a network. In addition to the instantiation of programs the ability for users to interact with running instances has to be given. Three kinds of

communication are required for the API to provide a general set of communication. These will be the execution of shell commands inside the running container, executing commands via ssh as well as using a *REST* API that an application may provide. Not every application will provide every kind of communication, as mentioned in Section 3.1.2. Thus the kind of communication, for example the execution of a shell command, can only be executed if the program supports it.

This concludes the definition of the requirements. We now define the interface and the requests for this project.

3.2. Interface

In this section we will define the interface, that will be used by users of the API. The interface that will be defined in this section is the external interface of the API. The splitting up of functionality that we discuss in Section 4.1 causes different interfaces for internal calls of the API. We discuss further detail on the difference of internal and external interfaces in that section as well. We will discuss further detail on the difference of internal and external interfaces section as well.

In general the interface that we will be defining will be that of a *REST* API as stated in Section 2.5. This means, that multiple endpoints exist, the requests types can be GET, POST, or DELETE, and each request contains all the data that is required for it to be executed. For the simplicity of the further definition the protocol, host and port on which the API will be accessible will not be mentioned. Every possible API call will be presented in the following. The definitions will contain the path, method type, parameters, possible responses and the body of POST requests. Defined API endpoints are divided into three different groups. Belonging to a group is defined by the second element of the path of an API endpoint. It will always be one of the following three values.

- namespace
- catalog

- network

3.2.1. Example API calls

In this subsection some API calls will be shown as examples. A single GET and DELETE request are shown while two POST requests are used as an example. The remaining API call definitions can be found in Appendix A.

Figure 3.1 is an example of a GET request to the API. This request contains no parameters and returns information about the namespaces that are managed by the API. The response of this request, as well as any other request to the API will return data in the “application/json” format. For the request the response will contain a field called “data”, which is a list that contains the information of the namespaces that were created using the API.

Figure 3.2 shows the definition of the POST request that will be used to upload a file to the file catalog. The body of this POST request is using the format “multipart/form-data”. This format is used because it provides the ability to upload files. In addition to the file itself a name for it has to be specified too. Executing this API call may lead to different responses.

If the request was successful, the response comprises the fields “data” and “info”. The former contains a list of all registered files, while the latter contains a message stating, that the file has been added to the catalog. The request could fail for two different reasons, that will return different responses. One reason for failure is that a file with the specified name already exists in the file catalog. The second reason for failure is that an error occurs while writing the file to the folder of the file catalog. Each of these failures returns a response that contain an “info” field, which will contain a human-readable message describe the failure reason.

We show a second POST request in 3.3. This request executes a shell command inside the container of a running application in a network. This request contains the network id and the name of the addressed instance as path parameters. Since this request is a POST request it contains a body as well. The body comprises the

3.2. Interface

command to be executed, the arguments to that command, and a timeout in seconds for the termination of the command. If the execution of the shell command succeeds a response will be sent containing the standard output of the command in the “data” field. This API call has two possible reasons to return a failure response. These reasons are a timeout of the shell command and providing a resource that does not exist. Resources that must be provided for this command are the network and the instance. If the specified network does not exist or the network does exist but does not contain the specified instance, the response will specify which of these was the case in the “info” field. In case the shell command took more time to execute than was specified, the response will contain a message in the “info” field stating that the shell command timed out.

The last API call that will be presented is of the method type DELETE (Figure 3.4). This request is used for the removal of programs from the program catalog. Determining which program will be deleted is done by using the parameter “programName”. For this request there are two possible responses. If a program with the specified name exists it will be deleted and the response returns a list of all the remaining programs in the “data” field. Additionally the “info” field contains a message stating that the program has been deleted. This is done to provide the failure reason in a human-readable format. However, if no program with the given name is found, a response is returned that contains a message in the “info” field that states that the program does not exist.

get	/v0/namespaces	get list of all namespaces
Response		application/json
200	ok	
<pre>1 { 2 "data": [3 { 4 "index": 1, 5 "name": "testnet-0", 6 "referenceId": "refName", 7 "ingressPorts": [] 8 }, 9 { 10 "index": 1, 11 "name": "testnet-1", 12 "referenceId": "otherRefName", 13 "ingressPorts": [] 14 }, 15] 16 }</pre>		

Figure 3.1.: Example GET request

3.2. Interface

post	/v0/catalog/files <i>upload new file</i>
Body	multipart/form-data
<pre>1 fileName=someFileName 2 file=@local/path/to/file</pre>	
Response	application/json
200 ok	<pre>1 { 2 "data": [3 { "name": "file-0" }, 4 { "name": "someFileName" } 5], 6 "info": [7 "File 'someFileName' added" 8] 9 }</pre>
400 file already exists	<pre>1 { 2 "info": ["File 'someFileName' already exists"] 3 }</pre>
500 internal error	<pre>1 { 2 "info": ["Failed writing to disk"] 3 }</pre>

Figure 3.2.: Example POST request

3.2. Interface

post	/v0/network/{networkId}/instances/{instanceName}/shell <i>execute shell command in instance</i>
Parameter	
networkId	id of a network
instanceName	name for the new instance
Body	application/json
<pre>1 { 2 "command": "ls", 3 "arguments": "/home", 4 "timeout": 5 5 }</pre>	
Response	application/json
200	ok
<pre>1 { 2 "data": "Result of the shell command" 3 }</pre>	
400	command timeout
<pre>1 { 2 "info": ["Shell command timed out"] 3 }</pre>	
404	instance or network not found
<pre>1 { 2 "info": ["Network {networkId} not found"] 3 }</pre>	

Figure 3.3.: Example POST request

delete	/v0/catalog/programs/{programName} <i>delete a program</i>
Parameter	
programName	name of the program to be deleted
Response	
application/json	
200	ok
<pre>1 { 2 "data": [3 { "name": "program-0", ... }, 4 { "name": "program-1", ... } 5], 6 "info": [7 "Program 'someProgram' deleted" 8] 9 }</pre>	
404	program not found
<pre>1 { 2 "info": [3 "Program does not exist" 4] 5 }</pre>	

Figure 3.4.: Example DELETE request

4. Concept of the API

In this chapter we will discuss the concept of the API. This begins with presenting the way that tasks and functionality are split up in different microservices in Section 4.1. By splitting up the functionality of the API into multiple services a hierarchy emerges, that will be shown in Subsection 4.1.1. At last the structure and the setup of *Kubernetes* will be explained in Section 4.2. *Kubernetes* is necessary for the API as it provides the ability to manage namespaces that will be used to contain networks of distributed software.

4.1. Splitting functionality

The requirements of this work can be divided into three different parts as discussed in Section 3.1. As a result of this division of requirements the implementation will also be split into multiple parts. Each of these parts will be its own microservice as we described in Section 2.4. Implementing each set of requirements in its own microservice leads to a clear division of responsibilities. With each microservice managing a subset of the API cross communication of the services will be necessary. In the following we define the names for the services and the topic that the services will take over.

Catalog-Service Management of the file and program catalog: The Catalog-Service holds a collection of programs and files for the API. It manages adding and removing entries.

4.1. Splitting functionality

Namespace-Manager Management of the namespace that will be used by the API:

The Namespace-Manager provides the ability to create and delete namespaces. Additionally ingress ports for namespaces can be allocated.

Instance-Manager Management and interaction of instance in a network: The Instance-Manager handles the instantiation of programs in networks of the API. Furthermore it enables the communication with instances in a network.

A problem that occurs through splitting the project into multiple services is, that Users that would need to access multiple APIs. This problem can be alleviated by creating an additional services whose job it is to forward API calls to the specific microservices. The service that will take this role will be named *Frontend-Service*. Another challenge that the API faces comes from the individual needs of different applications upon instantiation. Programs are configured using different systems like configuration files. Configuration files or other means of configuring programs upon start requires individual tasks to be executed. These task may be the creation of a configuration file and placing it at a specific location or exposing ports to the network, that are required by an application. This challenge leads to the decision of including a type of service that is referred to as *Specific-Instance-Manager*. For this work, the only task that a *Specific-Instance-Manager* service is assigned to help with is the creation of instances of a specific application. As a consequence of this, a *Specific-Instance-Manager* is required for each program. The two newly established services are summarized in the following.

Frontend-Service Forwarding calls to sub services: Provides a single interface for users and forwards requests to the sub services.

Specific-Instance-Manager Instantiation of a specific program: Individual for every program that can be run by the API. Its job is to configure a program upon startup.

The services that we previously introduced are shown in Figure 4.1.

4.1.1. Hierarchy

The microservices are structured in a hierarchy. This hierarchy is not strictly enforced but serves as a reference for the calls that will be made between the services in most cases. Furthermore the hierarchy as it is shown in Figure 4.1 sets out the part of the API that is exposed for external access. Only the *Frontend-Service* is exposed to external access. It is the only exposed service, in order to have a single access point for the API.

The second layer of services contains the three services *Instance-Manager*, *Catalog-Service* and *Namespace-Manager*. These are the services, that will receive the forwarded calls from the *Frontend-Service*. In contrast to the *Instance-Manager* the *Catalog-Service* and the *Namespace-Manager* will always be present as one running instance of the services each. Thus the communication to these services are always addressing the same service instance. The *Instance-Manager* service differs from that scheme by having a running instance inside every namespace that is created by the API. The Reasoning for that design decision is given in section 4.2.

A third layer to the hierarchy only exists for the *Instance-Manager* in the form of *Specific-Instance-Manager*. It represents a collection of services that are also located inside the namespaces of the API. They receive a forwarded call from the *Instance-Manager* in case a new instance of the program that the *Specific-Instance-Manager* is designed for is requested. Consider, e.g., a user who wants to create a new network, add a program to that network, create a new instance of that program, and execute a shell command in the container executing that instance. To do so, they first have to request the creation of a namespace via the API. This call, as well as every other call created by the user are directed at the *Frontend-Service*. The request to create a namespace is forwarded to the *Namespace-Manager* by the *Frontend-Service*. Afterwards they have to create a request for the instantiation of the program in the previously create namespace. This request is forwarded to the *Instance-Manager* running in that network which requests information about the program for the *Catalog-Service* before forwarding the request to the *Specific-Instance-Manager* for that program. The *Specific-Instance-Manager* then creates the requested instance. Finally they have to send a request for the execution of the shell command to the

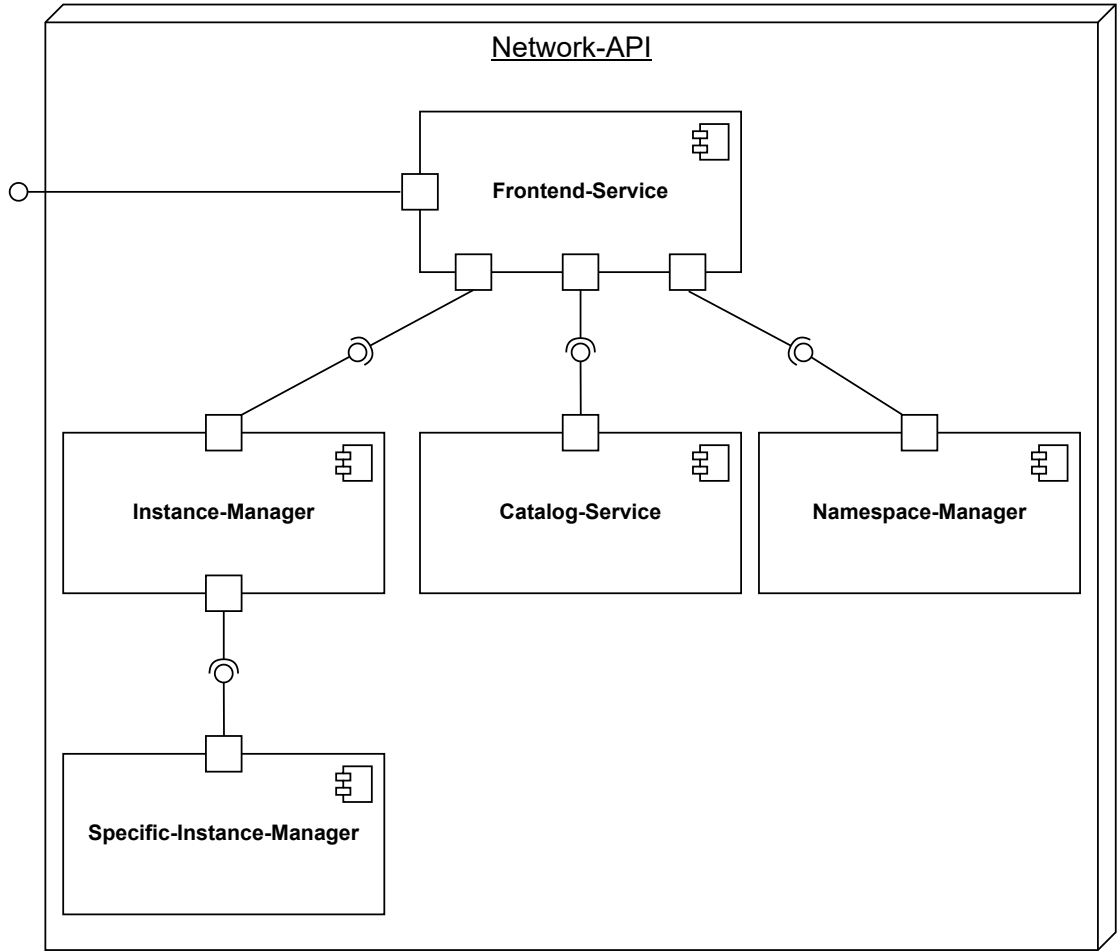


Figure 4.1.: Diagram of the service structure

API, which get forwarded to the *Instance-Manager*. The *Instance-Manager* will then issue the execution of the shell command in program instance.

As previously mentioned, the hierarchy is not strictly enforced. This behaviour is necessary, because the *Instance-Manager* has to be able to request data from the *Catalog-Service*. In the scope of this work, the request of the *Instance-Manager* to the *Catalog-Service* is the only communication of services that are on the same layer. One aspect of the hierarchy, that is always given is, that a services from a lower layer never sends a request to one in a higher layer. This means, that the requests of services in different layers is always are always one directional. Communication in

that direction is carried out by request from the higher layer with a response from the service in a lower layer.

4.2. Kubernetes setup

We deploy the project on a server cluster that is internal to the DLR network as explained in Section 2.7. We use one of the servers, namely *cloud-cpu5* as the *Kubernetes* node as shown in Figure 4.2. As we explained in Section 2.7, the server setup has a server called *cloud-storage-big* that provides an NFS storage to the *Kubernetes* node. The API makes the provided storage usable for applications running inside the *Kubernetes* cluster by creating a *PersistentVolume* as described in Section 2.3.1. As mentioned in 3.1.1, the API creates namespaces. The namespaces that are created by the API follow the naming scheme of beginning with *testnet-*, followed by a self incrementing index that are automatically supplied by the API of the namespace. Examples for these namespaces can be seen in Figure 3.1.1. The API itself is also located in a namespace of the *Kubernetes* node. When deploying the API, developer can choose the name of the namespace that will house it freely. For the sake of readability, we will call this namespace *network-api* for the remainder of this work.

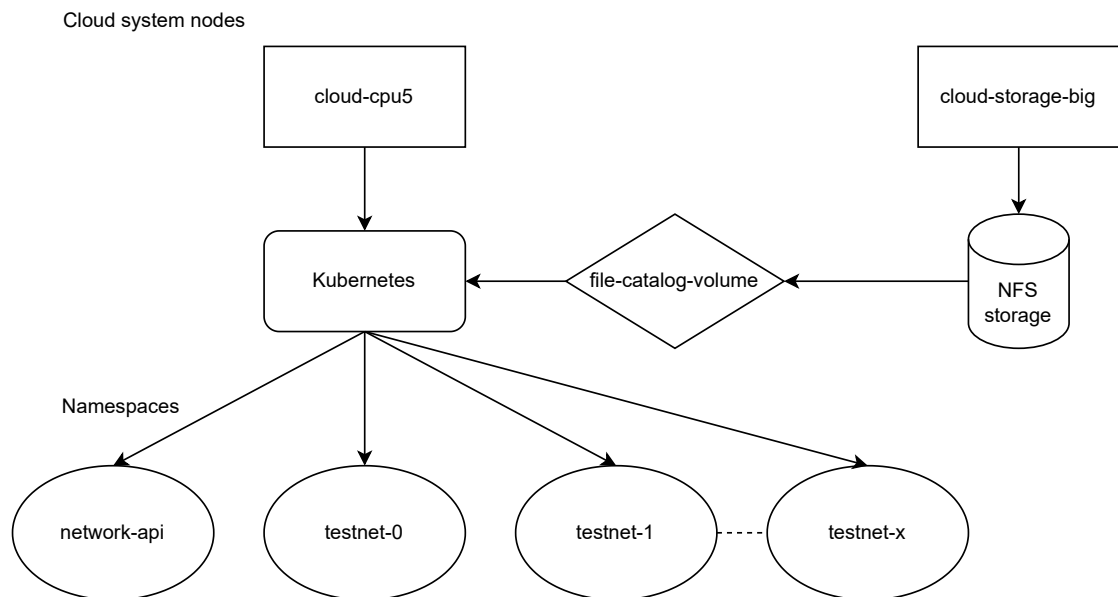


Figure 4.2.: Visualization of the Kubernetes setup

5. Implementation

In this chapter we will discuss the implementation of the API. First we explain the implementation of the *Catalog-Service* in section 5.1. Following this in Section 5.2 we show the implementation of the *Instance-Manager*. At last we discuss the implementation of the *Frontend-Service* in Section 5.3. Notice that we do not talk about the implementation of the *Namespace-Manager* as the service has been implemented before this work.

5.1. Catalog-Service

We started the implementation of the *Catalog-Service* by dissecting the requirements for it. The job of the *Catalog-Service* is to provide a collection of programs to the API that can be run by it as well as providing a set of files to programs running in the API. This leads to the creation of multiple classes that are depicted in Figure 5.1. The design is based on splitting up the functionality of the *Catalog-Service* into their own classes. A single class is responsible for receiving the requests for the service. This class named *CatalogServiceAPI*. The specific tasks of creating and managing the program and file catalog are done by a specific class each. These classes are the *FileCatalog* and the *ProgramCatalog* respectively. The job of the *CatalogServiceAPI* class is the separation of the interface of the service from the actual implementation. In the following we will first describe the implementation of the *ProgramCatalog* class and its dependencies in Subsection 5.1.1. Afterwards we discuss the implementation of the *FileCatalog* class and its dependencies in Subsection 5.1.2.

5.1. Catalog-Service

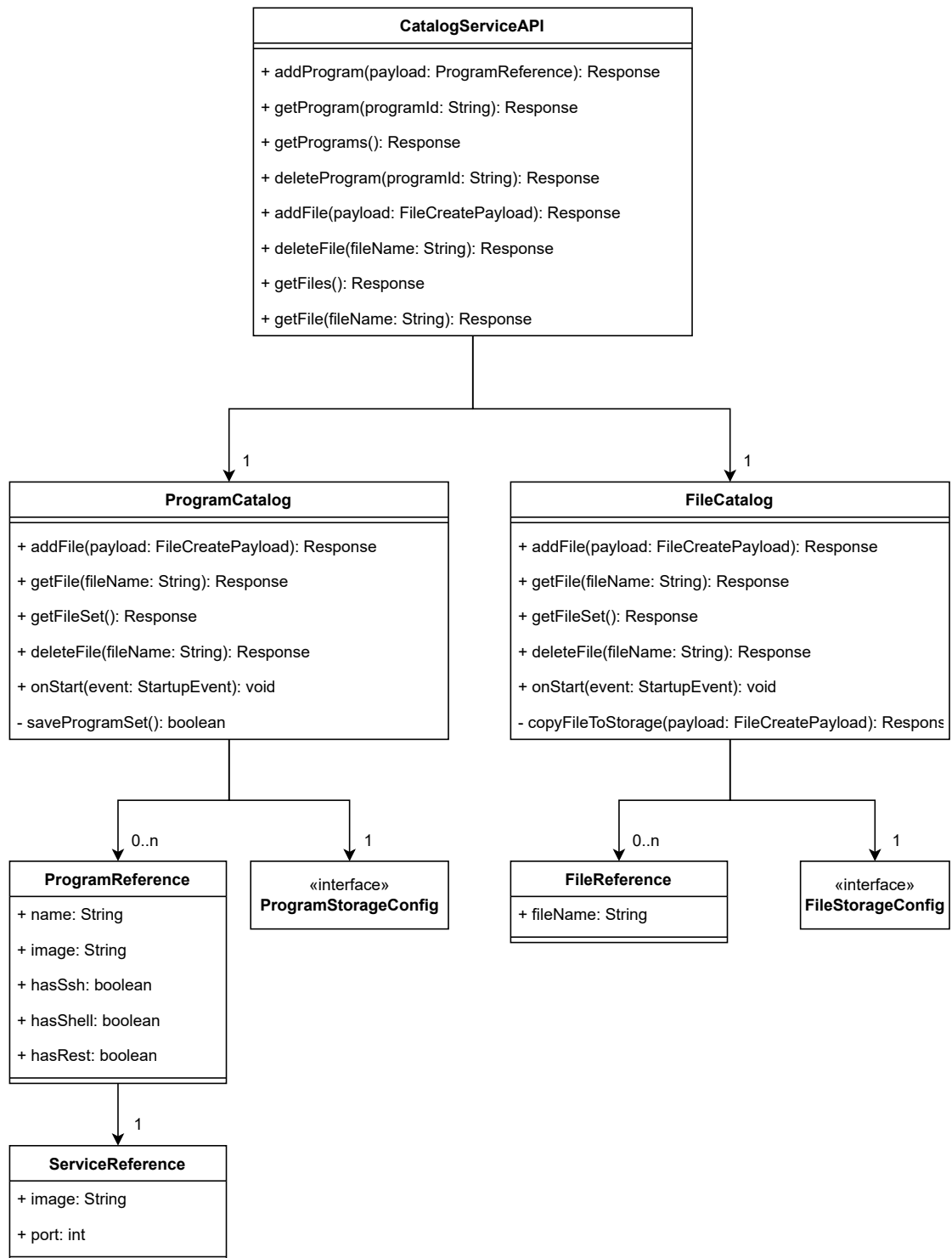


Figure 5.1.: Catalog-Service UML class diagram

5.1.1. Program catalog

The program catalog is the part of the *Catalog-Service* that manages the programs for the API. As we show in Figure 5.1 the *ProgramCatalog* class depends on the class *ProgramReference* which by itself depends on the *ServiceReference* class. Additionally the *ProgramCatalog* has a dependency on the *ProgramStorageConfig* interface. We created these two classes as data containers only without any functionality in the form of methods. The data stored in these classes has to be serialized. We will later go into the reason for the need of the serialization. For the serialization we will use a *JSON* serializer that is provided by *Quarkus resteasy-reactive-jackson* extension. In order for the serialization to be possible the fields of the classes have to be marked by the *JsonProperty* annotations. The annotation does not only mark the field as data that will be serialized but also specifies the name for the field that will store the data. For this the annotation has to be defined with the name for the field as shown in the following: `@JsonProperty("fieldName")`. The name specified does not have to match the name of the field that it marks however in this work this will always be the case. We do not provide an implementation for the *ProgramStorageConfig* interface as it is used to inject data from a *Kubernetes* configmap into the application. This is done by marking the interface via the *ConfigMapping* annotation and using the *Inject* annotation at the variable declaration in the *ProgramCatalog* class. The value provided by the interface is the mount path for the *Kubernetes* volume that will be used to store the information of the programs in the program catalog. As we described in Section 2.3 data saved to the file system of a pod running in *Kubernetes* is deleted when the container crashes or restarts. In Subsection 5.1.3 we discuss how we tackled that problem and explain the reasoning behind having the *ProgramStorageConfig* interface.

The *ProgramCatalog* stores the information of the programs that were added to it in the *Set* data structure. A *Set* is used as the data only has to be stored without the need for a name mapping or an order for the entries. For accessing the data of the program catalog or modifying it the class provides five methods. In the following we will briefly explain the purpose and the functionality of the methods. Additionally we note which API call to the *Catalog-Service* will lead to the execution of which

method.

The *addProgram* method takes over the task of adding a new program to the catalog. For this the method receives an instance of the *ProgramReference* class as a parameter. This method as well as all the other methods that interact with the catalog return an instance of the *Response* class which represents the response for a REST request. Inside the *addProgram* method we first test if an entry with the same name as defined in the program that should be added exists. This is done by stream operations on the *Set* that store the programs of the catalog as shown as an example in Listing 5.1

```
1  boolean containsProgramWithName = programSet.stream
    ().anyMatch(programReference -> programReference.
        name.equals(newProgram.name));
```

Listing 5.1: Stream operation example

In case a program with the same name already exists the method returns a *Response* instance with the code 400. If on the other hand the stream operation does not find a program with the same name in the catalog the method will continue with adding the new program. As we later discuss in Subsection 5.1.3 we also have to save the set of programs after adding a new one. This is done by the private method *saveProgramSet* which will also be a topic of the Subsection 5.1.3. The saving of the program set done by the previously mentioned method can either fail or succeed. In order to know whether the saving process has succeeded or not the *saveProgramSet* method returns a boolean value. That value is true if saving was performed successfully and false if an error occurred. When saving succeeds a response with code 200 is returned, otherwise the code will be 500 to indicate the error.

The *getProgram* method receives the name of a program as a parameter. This method is called when requesting the information of a single program. Similar to the stream operation that is used in the previously described method this one uses a stream operation on the set containing the program information in order to retrieve a program, that has the name specified by the parameter of the method. The result of that operation is of the type *Optional<ProgramReference>*. The *Optional<T>* type

of Java is a container that may or may not contain an instance of the class *T* which in our case is *ProgramReference*. An *Optional<T>* can be checked on whether it holds data or not using its *isPresent* method. We use this to determine if a program with the specified name exists or not. In the case of the data being present the method return this data and the status code 200 as its response. If the data is not present a response with the status code 404 is return which is generally acknowledged as the status code for a resource not existing.

The *getProgramSet* method is called when requesting information of all the programs of the program catalog. This method is simple in comparison to the ones we presented previously. Its simplicity is due to the fact that it always returns a response with the status code 200 and the data of the set containing the programs of the catalog in the *JSON* format.

The *CatalogServiceAPI* calls the *deleteProgram* method upon receiving a request for the removal of a program from the catalog. The structure of the implementation is almost equal to that of the *getProgram* method. This is the case because for a program to be deleted we first have to check that it actually exist, resulting in the same stream operation and conditional behaviour depending on its result. At this point the *deleteProgram* method differs from the *getProgram* method by removing the entry from the set and calling the method to save the data. Similar to the *addProgram* method this method sends different responses depending on whether the set could be saved or not. The response that the method return in case the specified program does not exist has the status code 400.

This concludes the methods of the *ProgramCatalog* class that the *CatalogServiceAPI* calls when receiving requests for the program catalog provided by the *Catalog-Service*. In Subsection 5.1.3 we discuss the last method (*saveProgramSet*) of the *ProgramCatalog* class.

5.1.2. File catalog

Figure 5.1 shows the *FileCatalog* class and its dependencies. The dependencies of the class are the *FileReference* class, the *FileCreatePayload* class, as well as the

FileStorageConfig interface. We use *FileStorageConfig* interface in the same way as the *ProgramStorageConfig* interface that we discussed in Subsection 5.1.2. The *FileReference* class is a data container for the name of a file with no additional functionality. For adding files to the catalog we use the *FileCreatePayload* class. The *FileStorageConfig* interface provides a path to a mounted *Kubernetes* volume. This volume is used to store the files that the catalog manages. As described in Section 3.2 we use the “multipart/form-data” method type to upload files to the file catalog. In order to represent the data of these requests we use the *FileCreatePayload* class. For *Quarkus* to be able to format the data of these requests into the *FileCreatePayload* class the fields of the class have to be marked with the *RestForm* annotation. *Quarkus* uses the names of the marked fields for the mapping of the parameters to map the fields of the request to the fields of the class. The implementation of the *FileCatalog* class is similar to the implementation of the *ProgramCatalog* class. We use a set of *FileReferences* to keep track of the files that have been added to the catalog. The class implements methods for adding files to the catalog, retrieving a list of all files, retrieving the contents of a single file, as well as deleting a file. In the following we present the implementations of these methods.

The *addFile* method is used to add a new file to the catalog. It receives an instance of the aforementioned *FileCreatePayload* class as a parameter. Analog to the implementation of the *addProgram* method of the *ProgramCatalog* class the method checks whether a file with the same name already exists. If no file with the same name exists the method goes on by trying to save the file to a *Kubernetes* volume. Saving files to the volume is done using the *copyFileToStorage* method which will be discussed Subsection 5.1.3. The method returns an instance of the *Response* class stating whether saving the file was successful or not. In case the file was saved successfully an instance of the *FileReference* class containing the name for the added file is added to the set of *FileReferences* and the method returns an instance of the *Response* class containing the status code 200. If the file could not be saved to the *Kubernetes* volume the response of the *copyFileToStorage* method is returned;

The *getFile* method get called by the *CatalogServiceAPI* class when it receives a REST GET request for a file of the catalog. The name of the requested file is passed

to the method as a parameter. First the method checks whether a file with the given name is an element of the catalog. In case the file does not exist the method returns a *Response* with the status code 404. If the file is an element of the catalog the method continues to read the data of the requested file and returns it in a *Response* containing the status code 200.

The *getFileSet* returns the set of files that have been added to the file catalog. The *CatalogServiceAPI* class calls this method when a REST GET request for the set of files is requested. This method always returns a *Response* with the status code 200 and the set of *FileReferences*.

The last method of the *FileCatalog* class that is called by the *CatalogServiceAPI* class is the *deleteFile* method. This method implements the deletion of a file from the catalog. The method starts by verifying that the file that is specified by the parameter of the method is stored in the catalog. If the file is part of the catalog the method tries to delete the file from the *Kubernetes* volume. On successful deletion of the file its entry in the set is also removed and the method returns a *Response* with the status code 200. In case the file is not part of the catalog the method returns a *Response* with status code 404 stating that the specified file does not exist.

5.1.3. Persistent storage

Persistent storage volumes are a necessity for the *Catalog-Service* in order to achieve persistent entries in the catalog. If for example all the data of the *Catalog-Service* were to be saved in the file system of the container every time it restarts the data would be lost. This is due to *Kubernetes* always starting a container in a clean state as it is defined in a Deployment. In the *Kubernetes* environment there are multiple events that can trigger a Pod and thus a container to be stopped or restarted both by a user or automatically. A user for example can stop a Pod by deleting the Deployment that manages the containerized application. As an example for *Kubernetes* itself stopping and some times also restarting a Pod automatically the crash of an containerized application triggers this behaviour. Whether *Kubernetes*

only stops the Pod or additionally restarts it is depending on the restart policy defined in the Deployment for that Pod.

Saving the information of the programs in the *Kubernetes* volume is done by the *saveProgramSet* method of the *ProgramCatalog* class as mentioned in Subsection 5.1.1. For saving the set of programs of the catalog we first create an instance of the *File* class. Upon its creation we specify the path of the file to use the mount path that is provided by the *ProgramStorageConfig* interface and a constant file name. In this work we call this file “programs.json”. Mapping the set of programs to a *JSON* string and saving it is done using the *ObjectMapper* class provided by the *resteasy-reactive-jackson* extension. We use its method *writeValue* which takes the set of programs and the previously mentioned *File* instance. This method has to be wrapped in a try-catch block as it can fail due to an *IOException*. If an exception is encountered the process of saving the data was not successful and the method returns *false* as an indication for this, otherwise it returns the value *true*.

The files of the file catalog are written to *Kubernetes* volume as mentioned in Subsection 5.1.2. We implemented this functionality in the *copyFileToStorage* method of the *FileCatalog* class. For this we provide the method with an instance of the *FileCreatePayload* class as a parameter. We start by creating the path for the file in the mounted *Kubernetes* volume. This is done by combining the mount path provided by the *FileStorageConfig* interface and the name provided by the *FileCreatePayload* parameter. We then copy the uploaded file to the previously constructed path. This operation is surrounded by a try-catch block as it can fail with an *IOException*. When the copy operation succeeds the method return a *Response* with the status code 200, otherwise the status code is 500. The *Kubernetes* volume that is used to store the files is exposed to the namespaces managed by the API. Through this the files are made accessible to the programs managed by the API.

5.2. Instance-Manager

For the implementation of the *Instance-Manager* we started by splitting up the functionality that it should provide into multiple classes. We present the result of

the partitioning of responsibilities in the UML class diagram in Figure 5.2. The *InstanceManagerAPI* handles the REST API calls and forwards the requests to four different classes. The four classes are the *ProgramForwarder*, the *SshCommunicator*, the *ShellCommunicator*, and the *RestCommunicator*. The *ProgramForwarder* implements the creation of instances in the network in which the *Instance-Manager* is deployed while the other three implement the communication with instances. First we discuss the creation of instances in a network in Subsection 5.2.1 before we explain the communication with instances in Subsection 5.2.3. At last we present the concept of *Specific-Instance-Managers* for the API in Subsection 5.2.2.

5.2.1. Instance creation

The creation of instances is done by the *ProgramForwarder* class. This class has dependencies to the classes *CreationPayload*, *ProgramReference*, and the interface *CatalogServiceConfig*. The *ProgramReference* class itself depends on the *ServiceReference* class. These two classes mirror the data of the program data of the *Catalog-Service* as described in Section 5.1. This data is necessary as it provides the data that is used to create an instance of a program. When users request the creation of an instance of a program the *ProgramForwarder* class requests the information for that program from the *Catalog-Service*. In order to address the *Catalog-Service* the class needs the URL for it. This URL is provided by the aforementioned *CatalogServiceConfig* interface. The interface is configured to provide data that is stored in a *Kubernetes* configmap as the *ProgramStorageConfig* interface discussed in Subsection 5.1.2. Creating new instances is done by sending requests to a *Specific-Instance-Manager* for the program that the instantiation is requested for. As a payload for this we use the *CreationPayload* class which wraps the necessary data for the request. The details of this process are discussed in Subsection 5.2.2.

5.2.2. Specific-Instance-Manager

The *Instance-Manager* uses *Specific-Instance-Managers* for the creation of programs in its network. We will briefly discuss the working principle of a *Specific-Instance-*

5.2. Instance-Manager

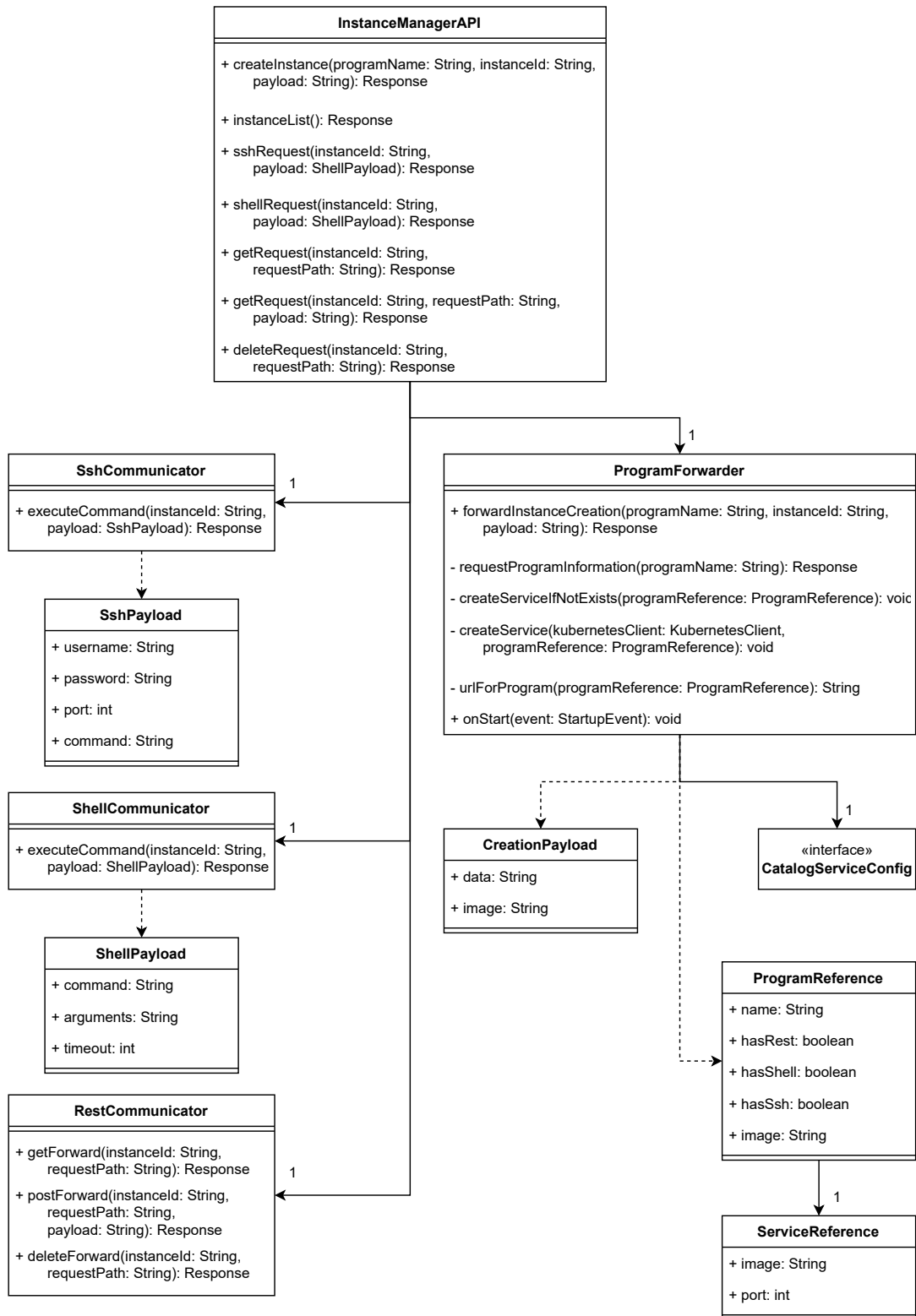


Figure 5.2.: Instance-Manager UML class diagram

Manager as each of these services has to be highly adapted to the program that it is designed for. We do this by using the *RCE-Instance-Manager* that we implemented within the scope of this work in order to be able to create instance of *RCE*, the distributed software we presented in Section 2.1.1. An *Instance-Manager* creates *Specific-Instance-Managers* on demand in its own network. For example this means that an *RCE-Instance-Manager* is created on the first time an *Instance-Manager* receives a request for the creation of an *RCE* instance. After the *RCE-Instance-Manager* is created or it already exists the *Instance-Manager* forwards a request for the creation of an *RCE* instance to the *RCE-Instance-Manager*. We will not go into detail of the implemented of the *RCE-Instance-Manager* as it is specific to *RCE* and its only job is to create instances. In order for the API to be able to create instances of other programs developers have to create *Specific-Instance-Managers* for these.

5.2.3. Communication

We implement the communication to instances running in a network via three classes. These classes are the *SshCommunicator*, *ShellCommunicator*, and *RestCommunicator*. Each of these is responsible for one method of communication.

The *SshCommunicator* provides the ability to send commands to a program via an ssh connection. In order to fulfill this task the class provides the *executeCommand* method. For this the method takes two parameters which are the id for the instance on which the command will be executed as well as an instance of the *SshPayload* class. We use the *SshPayload* class as the container for the data that is necessary for a command to be executed. These values are the username for the SSH account, the password for the user, the SSH port for the program, and the command. For the method to be able to make an SSH connection to a program we use the *JSch* library [7]. We use the *Session* and *ChannelExec* classes for the execution of the command. The method starts with creating a *Session* in order to connect to the program followed by opening the *ChannelExec* which is used to run the command itself. After setting up the *Session* and the *ChannelExec* we wait for the *ChannelExec* to disconnect for the program. Upon successful execution of the SSH command the method returns

a *Response* with the status code 200 and the result of the SSH command. The execution can fail due to an *JSchException* or an *InterruptedException*. In case one of these exceptions is triggered the method returns a *Response* with status code 500.

The *ShellCommunicator* implements the functionality of executing shell commands inside a container. This is done by invoking the *executeCommand* method. Similar to the method we previously discussed it provides two parameters. One of these is the id of the instance in which the command will be executed while the other parameter is an instance of the *ShellPayload* class. The functionality of this class is analog to the *SshPayload* class while providing a set of fields specific to the execution of shell commands. These fields are the shell command, arguments for the command, and a timeout in seconds. In the *executeCommand* method we use *Kubernetes* client API provided by *Quarkus* for the execution of the specified shell command. For this we create a *CountDownLatch* this is used for asynchronous notification of the shell command finish status as we count it down when the execution succeeds or fails. Additionally we define two *ByteArrayOutputStreams* that are used to receive the standard output and the standard error of the shell command execution. Next we retrieve the Pod that is specified by the instance id using the *Kubernetes* client API. Following this we create an *ExecWatch* which executes the shell command. Before execution we configure the *ExecWatch* to use the previously defined *ByteArrayOutputStreams* for standard output and error. Additionally we add an anonymous *ExecListener* class to count down the *CountDownLatch* as previously described. This is done by implementing its methods *onClose* and *onFailure* and calling the *countDown* method of the *CountDownLatch* that we defined. After starting the execution of the shell command we use the *await* method of the *CountDownLatch* in order to wait until it is counted down or the timeout provided by parameters runs out. We store the boolean return value of this method call in a variable and check its value. In case the value is false the command execution timed out and we return a *Response* with status code 500 to the *InstanceManagerAPI*. If the value is true and the *await* method did not trigger an *InterruptedException* we return a *Response* with status code 200 that also contains the standard output and error of the command. A triggered exception leads to the method returning a *Response* with status code 500.

5.3. Frontend-Service

We implement the *Frontend-Service* by using multiple classes. We show the UML class diagram for the *Frontend-Service* implementation in Figure 5.3. The *FrontendServiceAPI* class receives the REST API calls and forwards these calls to three classes that implement the actual functionality for the calls. These three classes are the *CatalogForwarder*, *NamespaceForwarder*, and *NetworkForwarder*. Additionally the *SubServices* interface is used by the *FrontendServiceAPI* class. This interface has no implementation as it provides information of a *Kubernetes* configmap in the same way the *ProgramStorageConfig* interface of the *Catalog-Service*. The interface provides URLs for the sub services that the *Frontend-Service* communicates with. Their usage will be discussed later while presenting the three classes that implement the communication. We forward the API calls from the *FrontendServiceAPI* class to the other classes by calling specific methods on them. Each of the classes that handle the communication provide methods for processing GET, POST, and DELETE requests. The *FrontendServiceAPI* class calls the corresponding methods when receiving requests. For the call forwarding classes to work the *FrontendServiceAPI* class sets the URL for each of the forwarding classes on the startup of the service. We implement an *onStart* method that has a *StartupEvent* as a parameter which is marked by an *Observes* annotation. This mechanism is provided by *Quarkus* in order to implement startup tasks of the service. We set the URLs for the sub services using setter methods provided by the three classes that implement the communication.

5.3.1. Catalog forwarding

The implementation of the *CatalogForwarder* class is by design not complex. This is due to the fact that it forwards the calls to the *Catalog-Service* without any additional functionality. We call the *Catalog-Service* using the *Client* class provided by the Java JAX-RS library which is part of the *Quarkus* framework. We use the client by constructing requests by combining the URL to the *Catalog-Service* and the path of the requests. For each *REST* method type the *CatalogForwarder* has a specific method for handling them. Following this the methods make the request

5.3. Frontend-Service

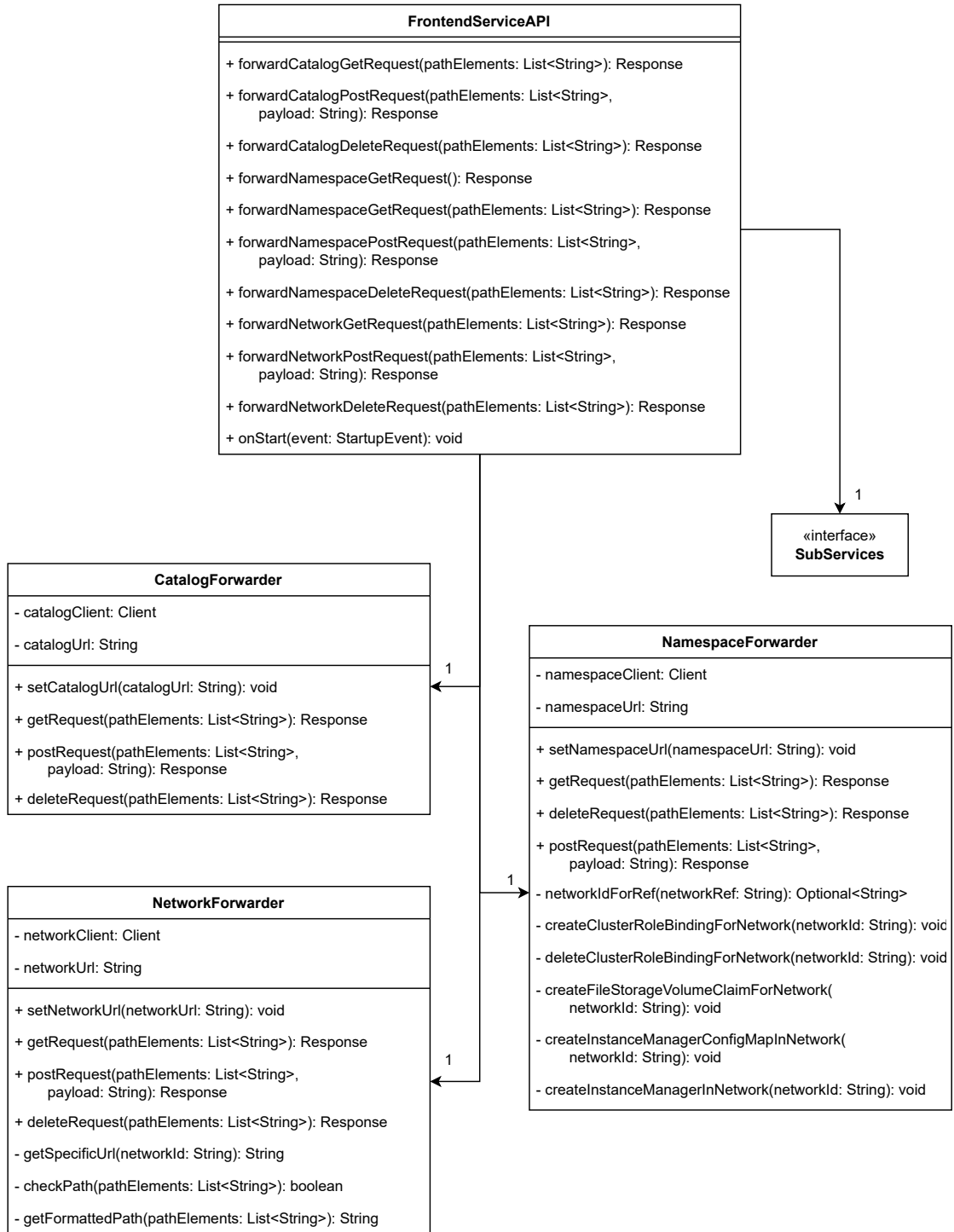


Figure 5.3.: Frontend-Service UML class diagram

using the different method types corresponding to the type they handle. Additionally the method handling the POST request also adds the payload to the request. The *FrontendServiceAPI* provides the path to the method using the parameter of the method while for POST requests there is an additional parameter for the payload. The requests return a response which the methods return to the *FrontendServiceAPI*.

5.3.2. Network forwarding

The implementation of the *NetworkForwarder* is similar to the implementation of the *CatalogForwarder*. We also implement three methods where each of these forwards one type of the supported REST method types (GET, POST, DELETE). The key difference for the forwarding of network requests is that the URL for these has to be constructed in a different way. The reason for this is that *Instance-Managers* are running inside the namespaces that they manage. As we will explain in Subsection 5.3.3 each *Instance-Manager* is exposed by a *Kubernetes Service*. The *Service* exposes an *Instance-Manager* and makes it addressable using a DNS address. The DNS address for the *Instance-Managers* has the format of “instance-manager.namespace.svc.cluster.local”. This is also the value that the *NetworkForwarder* receives from the *FrontendServiceAPI* class. In order to address the *Instance-Manager* we extract the namespace defined in the request path and substitute it into the URL. For this we implemented the *getSpecificUrl* method. Following this we format the request path to fit the format required by the *Instance-Manager*. Specifically this means removing the namespace and the “instance” element of the request path. We combine the constructed URL with the formatted path to construct the complete URL for the forwarded request. With this complete URL we issue requests and return their results in the same way as discussed in Subsection 5.3.1.

5.3.3. Namespace forwarding

The implementation of the *NamespaceForwarder* is mostly the same as the implementation of the *CatalogForwarder*. It is adapted to forward its calls to the

Namespace-Manager and contains additional logic to create an *Instance-Manager* in a newly created namespace and setting up *Kubernetes* to provide functionality to the namespace. We implement this in the method that handle POST and DELETE requests. First we forward the request to the *Namespace-Manager* followed by checking whether the request failed or succeeded. In case a request created a new namespace we follow this by creating an *Instance-Manager* in this namespace, creating a *ClusterRoleBinding* in order to provide the *Instance-Manager* its necessary privileges as well as creating a new *Kubernetes* volume pointing to the storage of the file catalog in order to make it accessible for this namespace. If a DELETE request successfully deleted a namespace the *Instance-Manager* as well as the *Kubernetes* volume are automatically deleted as these are namespace scoped objects however the *ClusterRoleBinding* has to be manually removed. We create and delete the previously described objects each in their own method. We will not discuss them in detail as these methods do not contain any logic but only use the *Kubernetes* client API in order to create or remove the previously stated resources.

6. Conclusion

In this chapter we evaluate the current state of the project in Section 6.1. Following this we discuss possible extensions for the future in Section 6.2.

6.1. Current state

The testing of distributed software involves a lot of manual work. Automated networks for the testing provide many advantages. For this purpose we implemented an API as described in the following.

The API consists of three parts that each manage a separate aspect of the API. Due to this nature we decided to use the microservice architecture. This lead to the creation of the four microservices *Frontend-Service*, *Catalog-Service*, *Namespace-Manager*, and *Instance-Manager*. The *Namespace-Manager* was provided as it has been implemented before this work. We implemented the *Catalog-Service* and *Instance-Manager* first as these provide the functionality of the API. The goal of the *Frontend-Service* is to provide a single interface for users to work with the API. We implemented this by forwarding calls made to the *Frontend-Service* to their corresponding services. The *Catalog-Service* supplies programs that can be run via the API as well as provide a set of files that can be used by the running programs. For the entries of the *Catalog-Service* to be persistent we make use of *Kubernetes* volumes. In each namespace that the API manages gets its own *Instance-Manager* service running inside of it. Using the *Instance-Manager* itself users can communication with instances running in that namespace using shell commands, ssh commands, or a *REST* API if a program provides it. We realize the instantiation of programs

by creating a *Specific-Instance-Manager* for each program that is registered in the *Catalog-Service*. We use *Specific-Instance-Managers* as the creation of instances may vary from program to program. Some programs may require configuration data at specific locations or have to have some specific ports exposed to the *Kubernetes* cluster. The necessary flexibility for such processes is provided by having individual *Specific-Instance-Managers*.

Kubernetes cluster administrators can setup the API to be usable without much limitation. An administrator has to setup the API as it needs permissions that administrators have to give it. After an administrator has setup the API any user of the *Kubernetes* cluster can use it. This is due to the fact that we implemented the API to receive configuration data from *Kubernetes* configmaps. In its current state the name of the namespace that is used to house the API can be chosen freely. The same applies to the *Kubernetes* volumes that the *Catalog-Service* uses for persistent storage. We made the choice to make the setup of the API configurable in order to make it easily deployable in different *Kubernetes* clusters without many conditions to the cluster.

6.2. Future prospect

We see the possibility for authentication in the API in order to limit the access of users to different parts of the API. The most obvious part of the API that would benefit from limiting access is the *Catalog-Service*. This is due to the fact that the *Catalog-Service* is supposed to provide security by providing a selection of programs and files that can be used via the API. Currently every user can add and remove files and programs to the *Catalog-Service*. We would increase the security provided by the *Catalog-Service* by allowing only a selected set of users to make these operations. Additionally the API could benefit by adding the possibility of making authentication necessary for networks of the API. Through this unwanted access and modification to a network can be prohibited. This may be desirable for networks as modification of it by other users may be unexpected and can lead to the misbehaviour of the programs running in it.

The usability of the API can be increased by simplifying the use of the communication with programs running in networks. As an example for the SSH communication the API could create mappings for every program that supports it and is running in a network. Such a mapping could contain the standard SSH port that will be used for SSH as well as a user and its password or public key. This would lead to a reduction in parameters that have to be transmitted by an SSH communication request leading to simplified usage for the users. A similar mapping could also be applied to the REST communication in order to remove the need to supply the port of the REST API supplied by a program. Another functionality that leads to an improved user experience is connected to *Kubernetes* ingress ports. Ingress ports can be added to a namespace in order to make a service running in it accessible from outside of the *Kubernetes* cluster itself. The *Namespace-Manager* provides the ability to add ingress ports to a newly created namespace. However if a user wants to expose a service running in a namespace of the API that has an ingress port the service has to be manually configured to be exposed on the same port as the ingress port. This process could be made easier for the user by making a mapping for ingress ports of a namespace. Providing such a mapping would enable a user to specify that a program that is started should use the port that is associated with an ingress port.

Bibliography

- [1] Red Hat Ansible. *Ansible is Simple IT Automation*. URL: <https://www.ansible.com/>. (accessed: 07.09.2022).
- [2] The Kubernetes Authors. *Kubernetes*. URL: <https://www.kubernetes.io/>. (accessed: 04.08.2022).
- [3] Brigitte Boden et al. *RCE: An Integration Environment for Engineering and Science*. 2019. DOI: 10.48550/ARXIV.1908.03461. URL: <https://arxiv.org/abs/1908.03461>.
- [4] Docker. *Docker - Home*. URL: <https://www.docker.com/>. (accessed: 07.09.2022).
- [5] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. Irvine, CA: University of California, 2000.
- [6] Lokesh Gupta. *HTTP Methods - REST API Tutorial*. URL: <https://www.restfulapi.net/http-methods/>. (accessed: 07.09.2022).
- [7] Inc. JCraft. *JSch - Java Secure Channel*. URL: <https://www.jcraft.com/jsch/>. (accessed: 07.09.2022).
- [8] Jekyll. *Quarkus - Supersonic Subatomic Java*. URL: <https://www.quarkus.io/>. (accessed: 08.09.2022).
- [9] Ajay D Kshemkalyani and Mukesh Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2011.
- [10] Deutsches Zentrum für Luft- und Raumfahrt e.V. *RCE*. URL: <https://www.rcenvironment.de/>. (accessed: 15.09.2022).
- [11] Claus Pahl. "Containerization and the PaaS Cloud". In: *IEEE Cloud Computing* 2.3 (2015), pp. 24–31. DOI: 10.1109/MCC.2015.51.

- [12] Dharmendra Shadija, Mo Rezai, and Richard Hill. “Towards an understanding of microservices”. In: *2017 23rd International Conference on Automation and Computing (ICAC)*. 2017, pp. 1–6. DOI: 10.23919/ICoNAC.2017.8082018.

Appendix A.

API calls

post	/v0/namespaces <i>create new namespace</i>
Body	application/json
<pre>1 { 2 "ingressPortCount": 0, 3 "referenceId": "someReferenceName" 4 }</pre>	
Response	application/json
200 ok	
<pre>1 { 2 "data": { 3 "index": 2, 4 "name": "testnet-2", 5 "referenceId": "someReferenceName", 6 "ingressPorts": [] 7 }, 8 "info": [9 "Available ingress ports (before): [21000, 10 21001, 21002, 21003, ...]", 11 "Creating namespace testnet-2" 12] 13 }</pre>	

delete	/v0/namespaces/id/{id} <i>delete namespace by id</i>
Parameter	
id	id of namespace
Response application/json	
200	ok
<pre>1 { 2 "info": [3 "Deleting namespace testnet-2" 4] 5 }</pre>	
404	namespace not found
<pre>1 { 2 "info": [3 "Namespace not found" 4] 5 }</pre>	

delete	/v0/namespaces/ref/{ref} <i>delete namespace by reference</i>
Parameter	
ref	reference of namespace
Response application/json	
200	ok
<pre>1 { 2 "info": [3 "Deleting namespace testnet-2" 4] 5 }</pre>	
404	namespace not found
<pre>1 { 2 "info": [3 "Namespace not found" 4] 5 }</pre>	

get	/v0/catalog/files	get list of files in the catalog
Response		application/json
200	ok	
<pre>1 { 2 "data": { 3 "files": [4 { 5 "name": "file-0" 6 }, 7 { 8 "name": "file-1", 9 } 10] 11 } 12 }</pre>		

delete	/v0/catalog/files/{fileName} <i>delete a file</i>
Parameter	
fileName	name of the file to be deleted
Response application/json	
200	ok
<pre> 1 { 2 "info": [3 "File 'fileName' deleted" 4] 5 }</pre>	
404	file not found
<pre> 1 { 2 "info": [3 "File 'fileName' does not exist" 4] 5 }</pre>	
500	internal error
<pre> 1 { 2 "info": [3 "Error deleting file 'fileName'" 4] 5 }</pre>	

get	/v0/catalog/programs <i>get list of programs in the catalog</i>
Response application/json	
200	ok
<pre> 1 { 2 "data": { 3 "programs": [4 { 5 "name": "program-0", 6 "image": "url.to/container/image", 7 "hasSsh": false, 8 "hasShell": true, 9 "hasRest": true, 10 "service": "url.to/service/image" 11 }, 12 { 13 "name": "program-1", 14 "image": "url.to/other/image", 15 "hasSsh": true, 16 "hasShell": true, 17 "hasRest": false, 18 "service": "url.to/some/service" 19 } 20] 21 } 22 }</pre>	

post	/v0/catalog/programs <i>upload new program</i>
Body	application/json
<pre> 1 { 2 "name": "someProgram", 3 "image": "url.to/container/image", 4 "hasSsh": true, 5 "hasShell": false, 6 "hasRest": false, 7 "service": "url.to/service/image" 8 }</pre>	
Response	application/json
200	ok
<pre> 1 { 2 "data": { 3 "programs": [4 { "name": "program-0", ... }, 5 { "name": "someProgram", ... } 6] 7 }, 8 "info": ["Program 'someProgram' added"] 9 }</pre>	
400	program already exists
<pre> 1 { 2 "info": ["Program 'someProgram' already exists"] 3 }</pre>	
500	internal error
<pre> 1 { 2 "info": ["Error saving program"] 3 }</pre>	

get	/v0/network/{networkId}/instances <i>get list of instances in a network</i>
Parameter	
networkId	id of a network
Response	application/json
200	ok
<pre> 1 { 2 "data": { 3 "instances": [4 { 5 "name": "instance-0", 6 "name": "instance-1", 7 } 8] 9 } 10 }</pre>	
404	network not found
<pre> 1 { 2 "info": [3 "Network not found" 4] 5 }</pre>	

post	/v0/network/{networkId}/instances/new/{programName}/{instanceName} <i>create new instance of a program in a namespace</i>
Parameter	
networkId	id of a network
programName	name of a program
instanceName	name for the new instance
Body	application/json
<pre> 1 { 2 "programInitData": "program initialization data" 3 }</pre>	
Response	application/json
200 ok	
<pre> 1 { 2 "info": [3 "Instance {instanceName} created" 4] 5 }</pre>	
400 instance creation failed	
<pre> 1 { 2 "info": [3 "Instance {instanceName} already exists" 4] 5 }</pre>	

post	/v0/network/{networkId}/instances/{instanceName}/ssh <i>execute ssh command in instance</i>
Parameter	
networkId	id of a network
instanceName	name for the new instance
Body	application/json
<pre>1 { 2 "username": "someUser", 3 "password": "examplePassword", 4 "port": 22, "command": "pwd" 5 }</pre>	
Response	application/json
200	ok
<pre>1 { 2 "data": { 3 "sshResult": "Result of the ssh command" 4 } 5 }</pre>	
404	instance not found
<pre>1 { 2 "info": [3 "Instance {instanceName} not found" 4] 5 }</pre>	
404	network not found
<pre>1 { 2 "info": [3 "Network not found" 4] 5 }</pre>	