DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

FAIR Data - The data value chain of biotechnological processes

Alexandru Mardale

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

FAIR Data - The data value chain of biotechnological processes

FAIR Data - Die Datenwertschöpfungskette biotechnologischer Prozesse

Author:	Alexandru Mardale
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Prof. Dr.-Ing. Dirk Weuster-Botz, M. Sc. Lukas Bromig
Submission Date:	16.05.2022

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Bucharest, 16.05.2022 Alexandru Mardale

Acknowledgments

Thanks to Lukas Bromig for his support throughout the development. Thanks to my family.

Abstract

Among other things, laboratory management implies management of experiments, experiment data and devices. With various experiments that need to be performed, devices that come with different interfaces and large amounts of data, manually managing a laboratory becomes infeasible.

While some of these aspects have already been automated as part of the SiLA 2 Manager, data operations are not currently automated. The purpose of this project is to automate the data retrieval, storage and processing operations and, at the same time, make them customizable. The main idea of customization is that it should be possible to specify what data from what devices is relevant for an experiment, where to store this data and how to process it. Retrieval and storage should be performed for the duration of an experiment, while data processing should be executed at the end of an experiment.

The contribution of this project mainly comes in the form of the Data acquisition service that is integrated into the SiLA 2 Manager. This new service will handle customization and automation of data retrieval and storage. Data processing will be handled by an existing application called KNIME. Experiment scheduling already exists in the SiLA 2 Manager. This was modified to also provide scheduling for the data operations. The Frontend service, which is the user interface, was also updated to cover the newly implemented functionalities.

Some of the FAIR data principles, that describe data qualities for reusability, have also been implemented. However, the reason for not having implemented all of them is that laboratory data reusability is a long term goal.

The source code can be found here:
https://gitlab.com/lukas.bromig/sila2_manager/-/tree/alex-mardale-thesis

Contents

1 Introduction

Modern laboratories are becoming more and more complex as technology evolves. Additionally, the domain of activity of the laboratory (such as biochemical engineering, in which this project was developed, but is not limited to) also increases in complexity. On top of this, laboratories are generally subject to high costs and are part of competitive domains. Therefore it is particularly of high importance to have proper and efficient management of the environment and this is not a trivial job, especially with an increasing size of the laboratory.

In order to provide an overview of a possible lab environment and the activities performed in it, I will take our laboratory as an example, while noting that what will be discussed here is not limited to our particular case.

One of the main activities done is to execute experiments with the purpose of identifying new processes and improving current processes for producing different substances. These processes have a lot of things that need to be considered, such as the different steps to be taken, the varying parameters for these steps, initial substances and so on. Therefore it is the case that multiple experiments need to be executed under different conditions such that a better and better process can be constructed.

As mentioned before, experiments are generally complex and there are multiple steps that need to be executed. This often implies the usage of multiple laboratory equipments. To provide some examples, one device could handle the distribution of a substance into multiple containers, another could be responsible for the heating of substances, while another could perform measurements of different properties, such as temperature, concentration. The point here is that multiple laboratory devices need to be coordinated as part of carrying out an experiment.

The main issue that can be identified here comes from the fact that experiments generally last entire days with devices being used for a time in the range of hours to days. Because of this, the manual approach of having a user start a specific experiment step has some great disadvantages in the sense that the previous step might finish (and allow the current step to start) at a time when there is nobody present in the laboratory, for example, during the night or during the weekend. As a result, the experiment will be blocked and can only continue when a user authorized to handle the experiment is back in the laboratory. A large amount of time is wasted in this situation. There is an even worse scenario, in which after a specific step has finished the next one must be started with little delay, otherwise the experiment is compromized.

Considering the above mentioned time constraints, it is reasonable to assume that a solution would be to automate the execution of the different steps and the usage of the different devices in order to reduce the large amount of downtime that can arise during the execution of an experiment as a result of the considerable user involvement required. It is true, however, that there are actions that canot be automated and need to be performed manually, such as cleaning the containers that are no longer required in the experiment. This can be considered as very little user involvement and can be tolerated, especially when compared to the case with no experiment automation.

An additional advantage that comes from automating experiments and reducing user involvement is a reduction in errors introduced by users. While errors can still appear as a result of incorrect experiment input data or settings, problems during execution are much less likely to occur, especially if the experiment execution data is simply reused (and possibly slightly modified) from a previous successful run of the same experiment.

Having discussed automatic execution of experiments, questions that arise are what sort of devices do we have and how is communication with devices done? What appears to be the case in general with laboratory equipment is that there is no provider that produces all of the required devices. Most probably, in a laboratory there will be devices from multiple manufacturers that need to be controlled as part of the same experiment. What is of concern here is that devices will have different interfaces and it is infeasible to attempt to control them directly through their own interfaces. Moreover, even if there is initial work put into controlling some devices directly, when new devices will be added they will require additional work. Considering this, what would be useful to have is a system than can work regardless of the device type and can handle all devices in the same way through a common interface.

With the relevant aspects regarding experiment execution being highlighted through the above points, it is now time to turn our attention to the results of the experiments and the produced data. What sort of data is produced, how do we specify what is relevant to us, how do we actually retrieve the data from the laboratory devices, how and where do we store it, what do we want to do with it, how do we precess it are questions that immediately arise when considering why we do experiments in the first place.

Most of the data we are interested in is in the form of time-series data, as we are generally concerned with the evolution of some properties over time. Additionally, we are also interested in metadata, which is basically information about various aspects of the experiment, the devices, the parameters used and so on. On the other hand, not all data produced by a device is relevant to the experiment, therefore, as noted above, a mechanism to specify what is of interest poses a great advantage. Retrieving data from devices could be performed through the common interface discussed before, but what might be relevant is how often we do this. On top of all of this, to achieve the purpose of an experiment, it might be the case that some complex processing needs to be done.

As with experiment execution, data related activities (retrieval, storage, processing) could greatly benefit from undergoing a transition from manual to automatic execution. It is the case, especially, with metadata, as it is particularly difficult for a user to manually keep track of it (for example, accurately keeping track of the version of every device involved in an experiment).

In the following section, existing possible solutions for some of the points considered in this section are discussed, together with what could be used to cover the rest of the points.

2 Background and Related Work

2.1 SiLA

SiLA [26] is an international organization with the purpose of developing standards for common device interfaces. The aim is to allow rapid integration of various laboratory devices produced by different manufacturers with the final goal of enabling lab automation and making it more efficient.

The acronym "SiLA" stands for Standardization in Lab Automation.

2.2 SiLA standard

The SiLA 1.x is the first of such standards developed by the SiLA consortium, however it was developed using XML and SOAP and because of this it is considered limited. For this reason, the details of the SiLA 1.x standard will not be discussed.

2.3 SiLA 2 standard

Learning from the mistakes of the SiLA 1.x standard, the SiLA consortium has developed the new SiLA 2 standard [29]. SiLA 2 aims to offer a framework for the control of laboratory devices and management of device data [24]. Compared to SiLA 1.x, in which the focus is put on device types, SiLA 2 considers and treats every device in the laboratory as a service, with the purpose of putting an emphasis on behaviour and functionality.

The functional specification of devices is written in a common way through the FDL (Feature Definition Language), which allows devices to describe and share their behaviour in a standardized way. Considering that everything is treated as a service, the way to communicate with a device is through the use of a client-server architecture. The two important parts are the SiLA Client and the SiLA Server.

The SiLA Server runs on top of a laboratory device with the purpose of providing a common interface, that is used for all devices. Regardless of the type of device or the device manufacturer, devices can be accessed in the same way, the only thing that differs is the functionality that is available based on the type of device.

Nonetheless, regardless of the device type or implementation, a SiLA Server will have some specific properties, such as a UUID, name, version - basically, general information about itself.

A SiLA Client is the way through which the functionality offered by a SiLA Server is accessed, as can be seen in Figure 2.1.

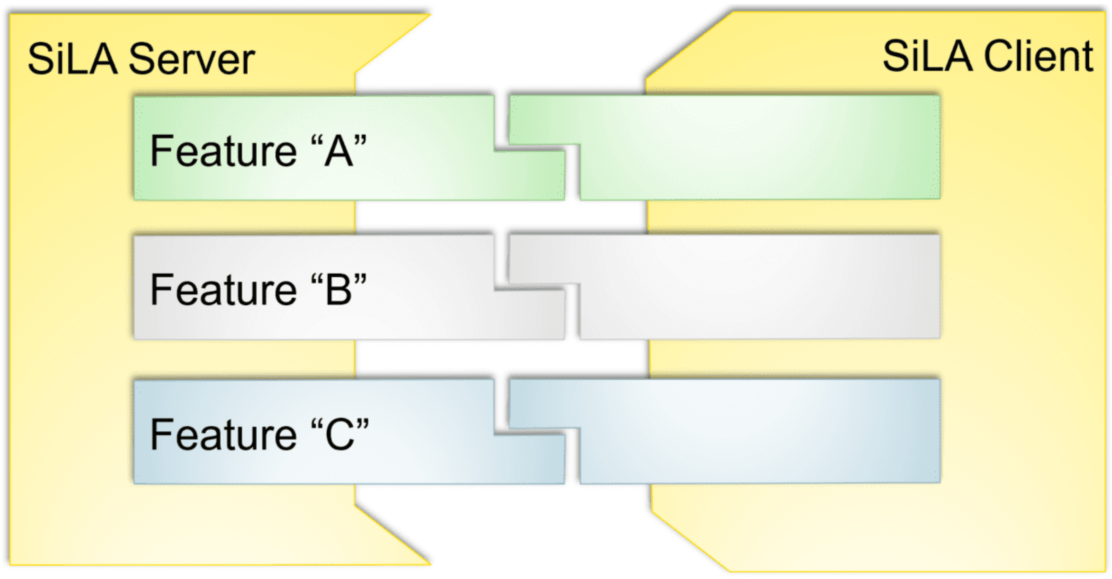

Figure 2.1: An overview of a SiLA Client and SiLA Server connection exposing some features of a service; Author: SiLA - https://sila-standard.com/
`https://drive.google.com/file/d/1CP_MTFmGZ89kkSfIXtPNQly97U1rcnuP/view`

As SiLA 2 is a standard that focuses on providing a common interface, it does not, therefore, enforce any sort of technology to use for implementation. The only thing it enforces is the interface. Considering this, the SiLA Client and SiLA Server can be implemented in any programming language. To aid in the integration of the SiLA 2 standard in existing systems that may be built using specific technologies and programming languages, the SiLA consortium provides implementations of the SiLA 2 standard in a variety of programming languages, such as Java, Python, C++, which are publicly available [30].

2.3.1 Feature

As can be seen from Figure 2.1, the functionality exposed by a SiLA services comes in the form of features. A feature in the context of the SiLA 2 standard is a group of closely related functions that the service can perform. A feature additionally has some information of its own, such as identifier, name, description, version and so on.

Different SiLA Servers can implement different features, depending on what functionalities the underlying laboratory device provides. However, according to the SiLA Specification [28], there is one feature that all SiLA Servers must implement and this is the 'SiLA Service' feature (as can be seen in Figure 2.2). The reason is that it offers general information about the SiLA Server and about the other features provided by the server.

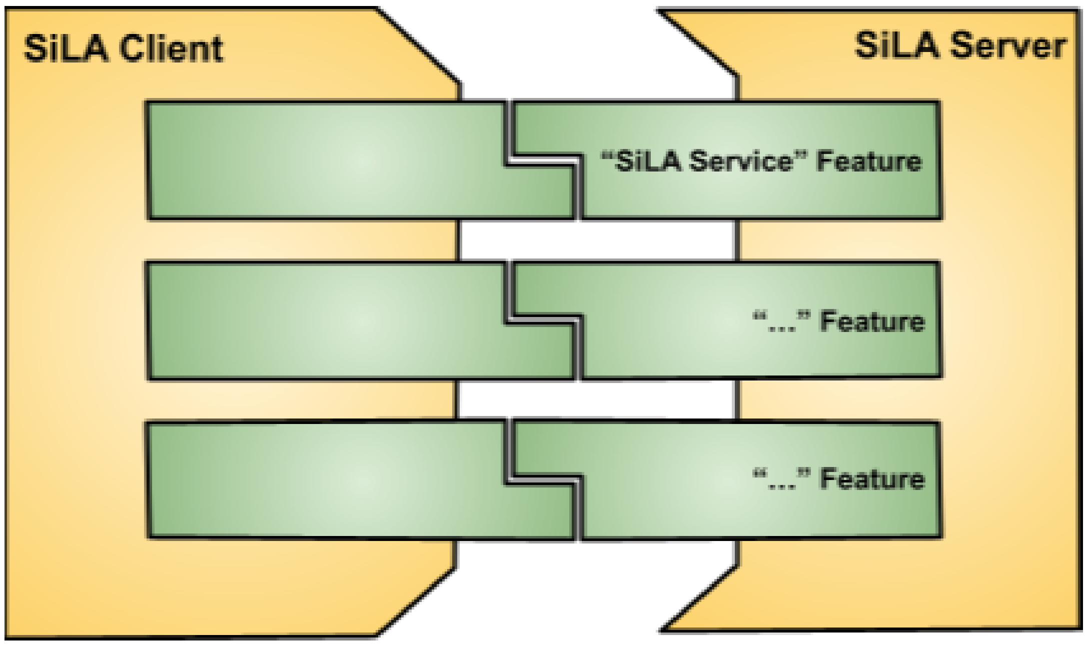

Figure 2.2: A SiLA Server that must expose the 'SiLA Service' feature and can additionally expose other features; Author: SiLA - https://sila-standard.com/
`https://drive.google.com/file/d/1CP_MTFmGZ89kkSfIXtPNQly97U1rcnuP/view`

As mentioned before, a feature contains functions that a SiLA service can execute.

These functions can be of one of two types, command or property, and have one of two modes of execution, observable or unobservable.

To note that commands can be compared to an action, while properties can be compared to the retrieval/interrogation of a value/aspect/property on the SiLA Server. While not particularly the best choice, ‘function’ will be used as an encompassing term for commands and properties.

Command

A command is a function on a SiLA Server that can be associated with performing an action on the server. What is particular about this type of function is that it can take input parameters (one, multiple or none), can provide responses after its execution and can provide intermediate responses during its execution. Additionally, it has some information of its own, such as identifier, name, description.

Unobservable command

An unobservable command is a command that does not provide monitoring of the execution and simply returns a final result.

Observable command

An observable command is a command that allows monitoring of the execution, provides intermediate responses through a data stream for the duration of the execution and returns a final result.

Property

A property is a function on a SiLA Server that can be associated with the retrieval of an aspect on the server. Executing such a function does not change any data on the server, data can only be changed through actions (commands). A property does not take any input parameters and returns only one response (the aspect that was queried on the server).

Unobservable property

An unobservable property is a property that can be interrogated once at any time by a client, but cannot be subscribed to. The current value is simply returned.

Observable property

An observable property is a property that can still be read once at any time by a client, but additionally allows subscription, in which case any time the property changes its value is provided through a data stream.

2.4 SiLA 2 Manager

The SiLA 2 Manager [27] is an existing project that is currently developed at the Technical University of Munich in which the work that is done as part of this thesis is integrated into.

This implementation of the SiLA 2 Manager was based on a simpler and smaller version developed as a proof of concept as part of an Interdisciplinary Project.

It contains implementations for some of the points discussed in Introduction, such as service managent, automatic execution of experiments.

While The SiLA 2 Manager contains some other parts as well, the most relevant ones are described below.

2.4.1 Service manager

The Service manager part of the SiLA 2 Manager handles automatic discovery and management of SiLA servers running in the network and handles SiLA Client logic in order to allow straightforward operations. Among others, it provides functionalities to view existing services, view the features and functions of a service, execute functions of a service.

2.4.2 Workflow designer

The SiLA 2 Manager also allows, through the Workflow designer, management of workflows, which describe experiment execution: what devices should be used, what should each edvice do and additional relevant information.

2.4.3 Workflow scheduler

As can be inferred from the name, the Workflow scheduler handles scheduling of experiments for specific times and ensures experiment execution when the given time is reached.

2.5 Data

Having discussed execution of functions and experiments, proceeding to discuss the resulting data naturally comes next.

As mentioned before, executing functions returns results in one way or another, but this is not particularly relevant at this point. What is relevant, however, is how and where we store this data and what we do with it afterwards.

2.5.1 FAIR Principles

The FAIR Principles [37][10] are a set of guidelines concerned with scientific data management and its improvement with respect to some properties. Each of these data properties has multiple, more detailed points, but only general aspects will be discussed here. The specific points will be discussed in Implementation when and where relevant.

Findability

The Findability principle states that data and metadata should be easy to find and uniquely identify.

Accessibility

Users are provided with the information on how data and metadata can be accessed. Data and metadata can possibly be secured, so this information should be offered as well.

Interoperability

Data and metadata can be integrated with other data and can be easily operated on by different applications.

Reusability

One of the most important principles, data and metadata should be well-described such that they can confidently be reused. In other words, other need to know where the data came from, under what conditions it was generated and other information that is relevant in deciding if the data is appropriate for the new application.

2.5.2 AnIML

'AnIML' stands for Analytical Information Markup Language [2][25][24] and is a standard based on XML for chemistry and biology data. Additionally, it has the advantage of implementing the FAIR principles.

The most important parts of an AnIML document are the Sample Declarations and Measurement Data.

The Sample Declarations keep track of the samples used in an experiment and, in case of derived samples, from what sample they originated. Additionally, alongside every sample it is possible to store arbitrary parameters.

The Measurement Data contains a list of experiment steps, each of those containing data about the applied technique, the samples used, the parameters used and the resulting data.

An AnIML document is based on a set of rules that is described below. The advantage of these rules is the fact that they are written in the form of an XML schema, therefore making it particularly easy to check the validity of an AnIML document.

AnIML Core Schema

The AnIML Core Schema describes the general structure of an AnIML document, indicating what elements may appear at the top level, the most relevant being the sample set and the experiment step set.

AnIML Technique Schema

The AnIML Technique Schema enforces the structure of AnIML Technique Definition Documents.

AnIML Technique Definition Document

An AnIML Technique Definition Document is a template for a specific technique, describing the parameters and data required by it.

To put technique rules in other words, a Technique Schema describes how you can write a template for a technique, while a Technique Schema is the actual template for a technique.

While the two schemas are provided by the developers of AnIML, Technique Definition Documents are custom templates that are written for specific techniques.

Decision against AnIML

At the start of the research done as part of this thesis AnIML showed great promise, but, having looked further into it, the conclusion was reached that the amount of resources available for it was limited.

Considering this, the decision was made not to proceed with the implementation of AnIML.

2.5.3 Processing

As part of the initial research different possibilities of doing data processing were investigated. The two that stood out the most were KNIME [17][4] (Konstanz Information Miner) and Riffyn Nexus [23]. Unfortunately, there was no scientific study to do a comparison between the two tools.

Riffyn Nexus

It allows creation of dataflows, either in a drag-and-drop manner or through a dataflow creation tool, with the purpose of allowing easy, fast and custom creation of data processing techniques.

However, the decision was ultimately made to choose KNIME as the data analytics tool, as a result of its better docummentation and support and larger interactive community.

KNIME

KNIME is a modular data analytics tool that allows the creation of data processing flows by combining different modules. There is a large amount of modules provided, either by KNIME or created by the community. One considerable advantage of KNIME is that custom modules can be created, therefore if a specific processing technique does not already exist, it can be implemented. However, many common data processing algorithms already have KNIME module implementations.

Besides actually processing the data, there needs to be a way to import and export it. For this purpose, a large variety of modules exist for different input and output modes, some of the most common being CSV files, various databases, PDF files, images.

A very simple example of a dataflow can be viewed in Figure 2.3. A CSV Reader will input the data from a CSV file, then a Row Filter will select some of the rows in the dataset and finally the resulting data is exported in the form of a PNG report by the Data to Report module. Each of these modules has some settings of its own, for example the CSV Reader will have the path of the CSV file, the Row Filter will have some rules based on which to filter and the Data to Report module will have the type and size of the exported file.

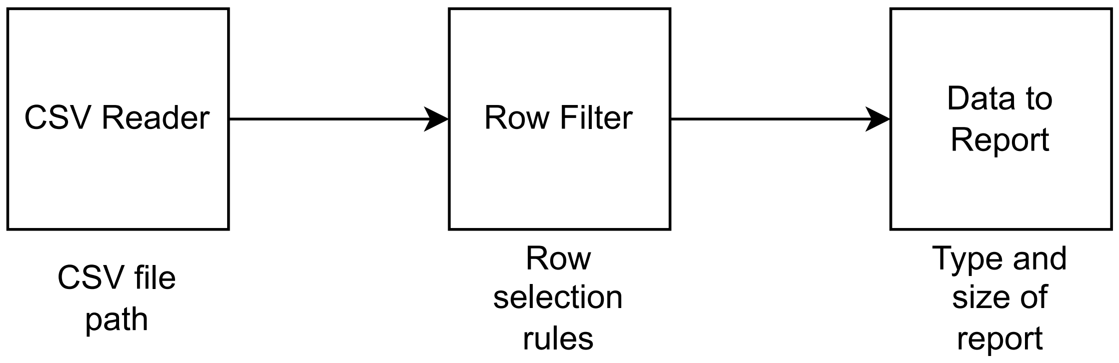

Figure 2.3: A simple example of a KNIME dataflow

The point of showing this process is to emphasize the fact that each of these modules is a standalone piece of functionality that can be combined with other modules, as long as the interfaces match. Modules are basically the building blocks of a dataflow.

Considering our simple example, the CSV Reader could be replaced by a module that takes data from a database, the Row Filter could be replaced by multiple, more complex operations, and the Data to Report module could be replaced by a module that writes the data back into a database. One of the greates advantages of KNIME is that everything is customizable to make it fit better your own requirements and to be more appropriate for your environment and your experimental setup.

Additionally, [12] identifies the interoperability property of KNIME dataflows in terms of operating systems and provides additional concrete examples of KNIME dataflows for various applications.

KNIME provides two major tools, the KNIME Analytics Platform [18] and the KNIME Server [19].

KNIME Analytics Platform

It is the tool that allows local creation, management and execution of dataflows. It provides an interface through which individual modules can be combined.

KNIME Server

It is a server that allows centralized storage, management, scheduling and control of dataflows. The main advantage of the KNIME Server is that it provides a REST API that enables programatic actions on dataflows.

KNIME and SiLA

To tie in with the previously discussed SiLA 2 standard, a cooperation [34] between KNIME and SiLA has already taken place with the purpose to provide a proof of concept for the processing of data from SiLA services through KNIME dataflows.

KNIME and FAIR

As KNIME is used to handle data, this relates to the FAIR principles. A discussion on the FAIR principles, how to make data FAIR and how each of the principles is implemented with KNIME is provided on the KNIME blog [36].

3 Motivation

This chapter will discuss the motivation behind the thesis, focusing on the drawbacks of the current laboratory processes, the improvements that are considered, and will present one of the main use cases of an automated laboratory and the smaller tasks involved in it.

3.1 Current laboratory drawbacks

While some of the points related to this section have been touched on in Introduction, they will be reiterated here for the purpose of clarity.

A normal laboratory environment has the purpose of supporting a large variety of experiments in a specific field of study. In order to do this, laboratory equipment is required. Additionally, it is reasonable to assume that different types of experiments require different types of equipment. Moreover, as the laboratory increases in size, experiment variety and number of experiments to be executed in parallel, the required number of devices and of different types of devices increases as well. One can imagine how complex the task of managing such a laboratory becomes.

Regarding the actual experiment execution, experiments generally take time in the range of hours to days, with different devices being used in different experiment steps for parts of this time. The main concern with a manual approach for this process is that an experiment step might be finished at a time when nobody is present in the laboratory, such as at night or during the weekend and the next step can only be started when someone is present. Two issues arise here, one being that a lot of the experiment time might be wasted in an idle status, and the second being that some experiment steps might not even support delay in execution start and must be performed immediately or shortly after the end of the previous step.

Having experiments that make use of devices, an additional issue comes from the variety of devices that will, inevitably, come from various manufacturers. This translates to having different interfaces for devices and, therefore, increased complexity in device communication.

On top of this, the set of experiments and the set of devices are not fixed and it should be possible to easily extend them (i.e., create new experiments and add new devices).

With the current laboratory process there is a lot of user involvement and this makes it, inherently, error-prone. Manually setting experiments, ensuring device communication and handling data will inevitably cause errors at some point, especially with an increasing size of the laboratory. The amount of data to be handled and the complexity also increase, both in the sense of recording metadata (such as experiment details, device versions) and in the sense of recording and processing the actual experiment results. At some point it becomes infeasible to do this manually.

3.2 Improvements

To overcome the drawbacks of a normal laboratory, improvements will be presented here. The ones that are already implemented will be shortly mentioned, with the focus being on the ones implemented as part of this thesis.

3.2.1 Common interface

A common interface for devices needs to be provided such that the complexity of communicating with arbitrary devices is reduced. For this purpose, an already existing solution was chosen in the form of the SiLA 2 standard and was integrated into the SiLA 2 Manager.

3.2.2 Laboratory management

Managing the large amount of devices available in a laboratory implies certain functionalities, such as tracking the available devices and adding new ones, viewing their information and offered functions, executing their functions. For this purpose, the Service manager was implemented into the SiLA 2 Manager and provides a central point of control.

3.2.3 Experiment creation and execution

Easily creating experiments, automating their execution and interaction with devices and reducing user errors are some of the main advantages of an automated laboratory and are handled by the Workflow designer.

3.2.4 Experiment scheduling

Scheduling experiments to run at specific times and removing the user involvement of manually starting them allows experiments to be started at any time, even if there is nobody in the laboratory. This functionality is already offered by the Workflow scheduler.

3.2.5 Data

Data-related activities are the main focus of this thesis. They are the parts that did not already exist and have been implemented under the newly create Data Acquisition service of the SiLA 2 Manager.

Automatic data retrieval

With the existing implementation of the SiLA 2 Manager, experiment data is not handled. As part of the experiment execution, some actions are done, but their results are not retrieved or handled in any way.

This is highly concerning, as one of the main reasons we do experiments in the first place is to actually look at their results. Therefore it is of great importance to have a mechanism in place that allows the automatic retrieval of data from devices.

The actual data that we are interested in comes in the form of some property, such as temperature, concentration. As the information we are concerned with varies over time, we want to retrieve data at various intervals over the duration of the experiment, thus ending up with our actual data being time-series data.

Additionally, we also consider metadata, this generally being, in our case, data about the devices. Compared to actual data, metadata can be stored only once, as it relates to some information that does not vary over the duration of the experiment.

With data and matadata we are referring to information that is retrieved from devices. On top of these two types of data, we are also interested in having some custom data, that is simply relevant data provided by the user.

Having discussed the above points about data, the next logical step is to provide a way of informing the automated data retrieval system about what and how to retrieve for specific experiments. Mostly, this boils down to specifying what functions of what devices we are interested in querying, what is actual data and how often to query it, what is metadata, and providing the required parameters for the function calls. The system should check if the provided information is actually valid and that the functions can be executed.

Automatic data storage

With data retrieval out of the way, we are now interested in automatically storing the data as soon as it was retrieved. The question that arises here is where do we store the data? One solution would be to have a specific database for this, but it has been identified as a benefit to be able to customize where certain data is stored.

Therefore, the system should allow a user to add the details of multiple databases and appropriately select based on the data that is to be stored. It should be checked, however, that the details provided are actually valid.

Automatic data processing

While having stored the data might be enough for certain experiments, others might require some processing to be performed in order to extract some additional information. Such as everything related to data so far, the processing that is applied to experiment data should be customizable. Users should be able to manage dataflows and provide one of those to an experiment, such that the dataflow is executed at the end of the experiment.

3.3 Use case

This thesis focuses on one main aspect, which is the one related to data operations. The point is to have the data operations, namely retrieval, storage and processing, automated.

From the user's perspective, because of having automated data operations, there is no manual execution of these steps. What the user has to do is customize these operations for specific experiments.

We will further refer to a custom data specification as protocol, to a custom database specification as database and to a custom data processing as dataflow.

The main use case that is implemented in this thesis is shown in Figure 3.1. Basically, the user has to provide to an experiment the data-related settings based on which automatic data retrieval, storage and processing will be performed. This can be further devided into the smaller use cases of managing protocols, databases and dataflows. These use cases will be discussed in more detail.

3.3.1 Protocol management

Regarding protocol management, there are some operations that a user should be able to perform. They are mentioned below and are also shown in Figure 3.2.

First of all, it should be possible to create a protocol. What this actually means and what information is required will be discussed. Very similar to creation, the user should be able to edit an existing protocol. A protocol can also be deleted. Finally, it should be possible to view one specific protocol and also a list of all protocols.

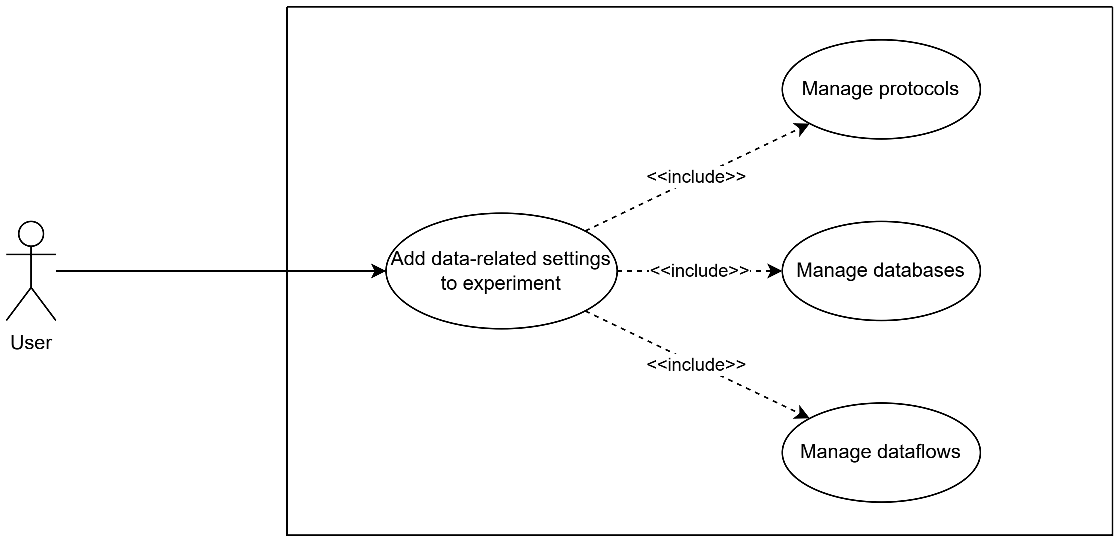

Figure 3.1: The main use case

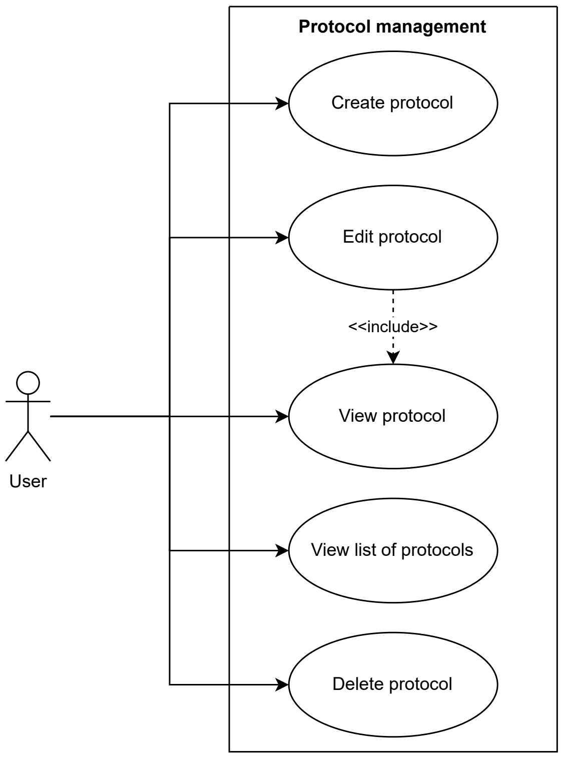

Figure 3.2: A use case diagram for protocol management

Going back to protocol creation, it is relevant to note the decision that the information in a protocol describes what data we are interested in from a specific SiLA Service. If data from multiple services is required, multiple protocols should be created. Additionally, multiple protocols can be created for the same service, with the purpose of allowing the user to request different data from the same service in different experiments.

The user should be able to select a SiLA Service for the protocol. In order to do this, the existing services must be viewed. After a service has been selected, the available features can be viewed. Selecting a feature then allows the user to see the functions for that feature. The user can then select a function and add it to the protocol. After this step, the parameters for the function can be provided, the interval can be set and the function can be marked as either actual data or metadata. Multiple functions can be added. Custom data can also be added to the protocol by providing a name and a value for every piece of custom data.

Protocol creation is also shown in Figure 3.3.

Updating a protocol is quite similar to creating a protocol, with the only addition that the user starts with an existing protocol instead of an empty one.

3.3.2 Database management

As mentioned before, a user should be able to manage where data is stored. To this purpose, database management should allow the expected operations of creating, updating, viewing, deleting a database and viewing a list of all databases.

For database creation things are a bit simpler compared to protocols, as here the user has to simply fill in some fields. The name of the database to be used inside of an actual running database instance has to be provided. An address and a port are required for the database instance to be located. The user must also provide authentication details in the form of a username and a password. Finally, a retention policy must be specified.

This information about data creation is additionally shown in Figure 3.4.

Database updating is similar to database creation, just that the fields to complete are initially populated with the details of the database that is to be updated.

3.3.3 Dataflow management

As with the other two types of objects, a user should be able to perform basic operations on dataflows: creation, update, deletion, viewing of a specific dataflow and viewing of a list of all dataflows.

The structure of a dataflow can vary depending on the tool that is used, therefore the specifics of dataflows will be discussed under Implementation.

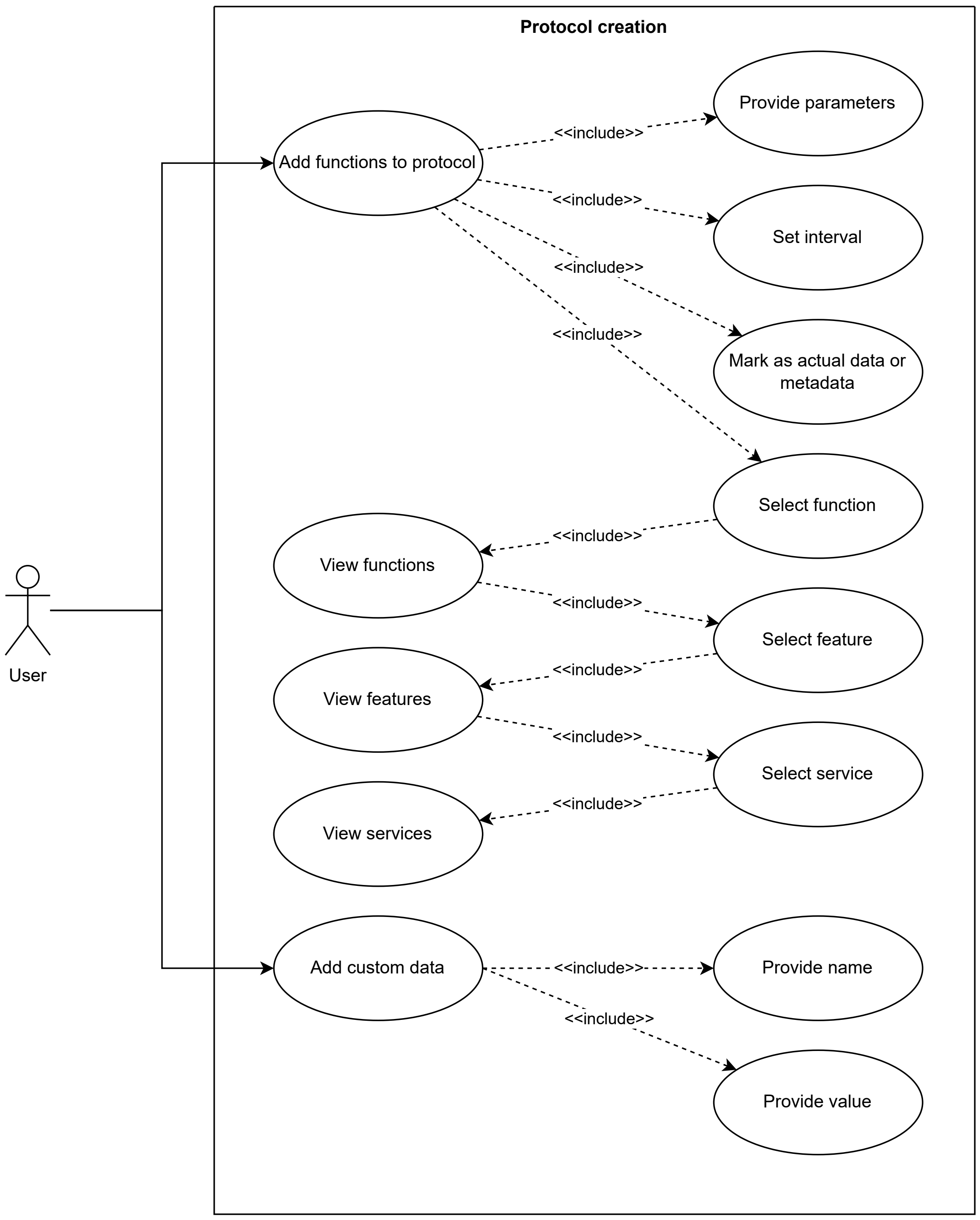

Figure 3.3: A use case diagram for protocol creation. The main goal is to add functions and custom data to a protocol. Adding functions is further broken down into the chain of required steps

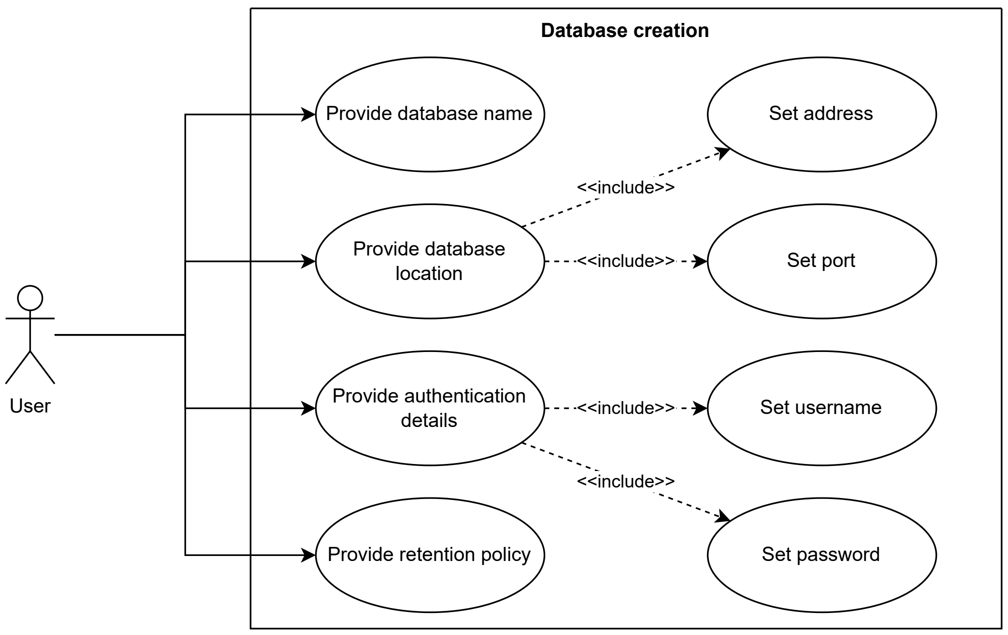

Figure 3.4: A use case diagram for database creation

3.3.4 Selection of experiment data-related settings

A user should be able to specify how data should be handled (retrieved, stored, processed) for an experiment. A dataflow can be provided to an experiment. Protocols and databases are provided in pairs, such that it is possible to store data from different protocols in different databases.

3.3.5 Data retrieval and storage process management

As mentioned before, data retrieval and storage will be performed for the duration of an experiment. A user should be able to stop and restart these processes.

4 Design

This chapter will present the design decisions that went into creating the project.

First, the overall architecture will be discussed and its advantages will be presented. This will also show how the integration with the SiLA 2 Manager will be done.

Afterwards, the functionalities the system should offer are highlighted, without going into the implementation details.

The way in which those functionalities are exposed is further discussed.

A general mechanism for the communication between the different parts of the SiLA 2 Manager is required and it will be presented here. The focus will be on the general concept and not its implementation.

Additionally, there are a few aspects that are fixed, considering what the project should be able to do, and these will be briefly identified.

4.1 Architecture

This section will focus on architecture in the sense of how the newly implemented part will be integrated into the larger SiLA 2 Manager.

The two most common architectural styles are monolithic architecture and microservice architecture.

4.1.1 Monolithic architecture

This type of architecture is more traditional and manifests itself in the form of having everything grouped together. One advantage that this approach has is that it is easier to setup and manage. However, a monolithic application that grows very large tends to get difficult understand and modify. Additionally, if a specific part of the application becomes a bottleneck, it is not possible to simply scale that part, the entire application needs to be scaled. Moreover, if one part is updated, the entire system must be restarted, which slows down application startup. Furthermore, encountering a bug in one part of the code might stop the execution of the entire application.

What can be noticed is that the term 'part' has been used a lot above. Its main purpose was to refer to groups of closely related functionalities. The point here is that we intuitively group functionalities together, but under a monolithic architecture parts can affect eachother and the entire application in undesirable ways, regardless of how related they are.

A solution to the issues of the monolithic architecture comes in the form of a microservice architecture, in which an application is actually separated into multiple parts, called microservices.

4.1.2 Microservice architecture

In a microservice architecture related functionalities are grouped together and separated from others. This separation is done in the form of having multiple microservices. One thing to note is that every microservice will have its own database.

One disadvantage is the fact that the setup and management of a larger application composed of multiple microservices becomes a bit more complicated.

However, it comes with anumber of advantages. Every microservice can be individually managed and developed independently of the other microservices. Microservices can be deployed independently, which allows them to be developed using different technlogies. Additionally, restarting one microservice does not require a restart of an entire application. Moreover, individual microservices can be scaled independently as required.

For the reasons described above, a microservice architecture has been chosen for development.

We will refer to the group of newly implemented functionalities as the Data acquisition service. A few changes were required in some other services of the SiLA 2 Manager and the Frontend service was updated to offer access to the functionalities of the Data acquisition service, but these will be discussed in more detail in Implementation.

4.2 Data acquisition service

The Data acquisition service will allow management of protocols, as described in Motivation. The important aspect to note here is that communication with the Service Manager will be required, such that information about the SiLA services can be presented to the user when managing protocols. Additionally, execution of SiLA functions will be attempted in order to verify that a protocol can be executed.

Database management functionality will be provided. Interaction with an actual database instance will be required, with the purpose of verifying that data can actually be stored given some database details.

Dataflow management will mostly be provided through KNIME, as it already offers the functionalities that we require.

Additionally, the Data acquisition service will handle data retrieval and storage that will be executed for the duration of an experiment. The Workflow scheduler service will need to interact with the Data acquisition service and KNIME in order to start data retrieval, storage and processing at the appropriate times.

The interactions between the different services and components of the SiLA 2 Manager are also illustrated in Figure 4.1.

4.3 Frontend

As seen in Figure 4.1, there is additionally a Frontend service that is basically a user interface to allow interaction with the available functionalities offered by the other services. This already existed in the SiLA 2 Manager, but will be updated to cover the new Data acquisition service and its interactions with other services.

4.4 REST API

The Data acquisition service will expose its functionalities in the form of a REST (REpresentational State Transfer) API (Application Programming Interface). This is also how communication with the other services will be done.

An API is an interface that exposes and provides information about the functionalities offered by an application, a service. It acts as a contract between a client and a server. A REST API is an API that conforms to REST.

REST is an architectural style that provides standards for communication. One of the standards it proposes is making it possible to uniquely identify every resource on the server, which is something we are concerned with, as users will need to interact with multiple resources. Manipulation of resources through representations is another standard according to which the actual representation of a resource (a piece of data) can be used to make modifications on the resource.

Additionally, it implies the usage of a client-server architecture. This is beneficial to us, as separating the service and the user interface allows us to better manage both.

Moreover, REST is stateless. This means that no client information is kept on the server between different requests from the client. Therefore, every request that is received by the server must contain all necessary information to be processed.

The above aspects of REST will be implemented such that the advantages they provide will be obtained. However, there are aspects of REST that will be omitted, such as HATEOAS (hypermedia as the engine of application state) and cacheability.

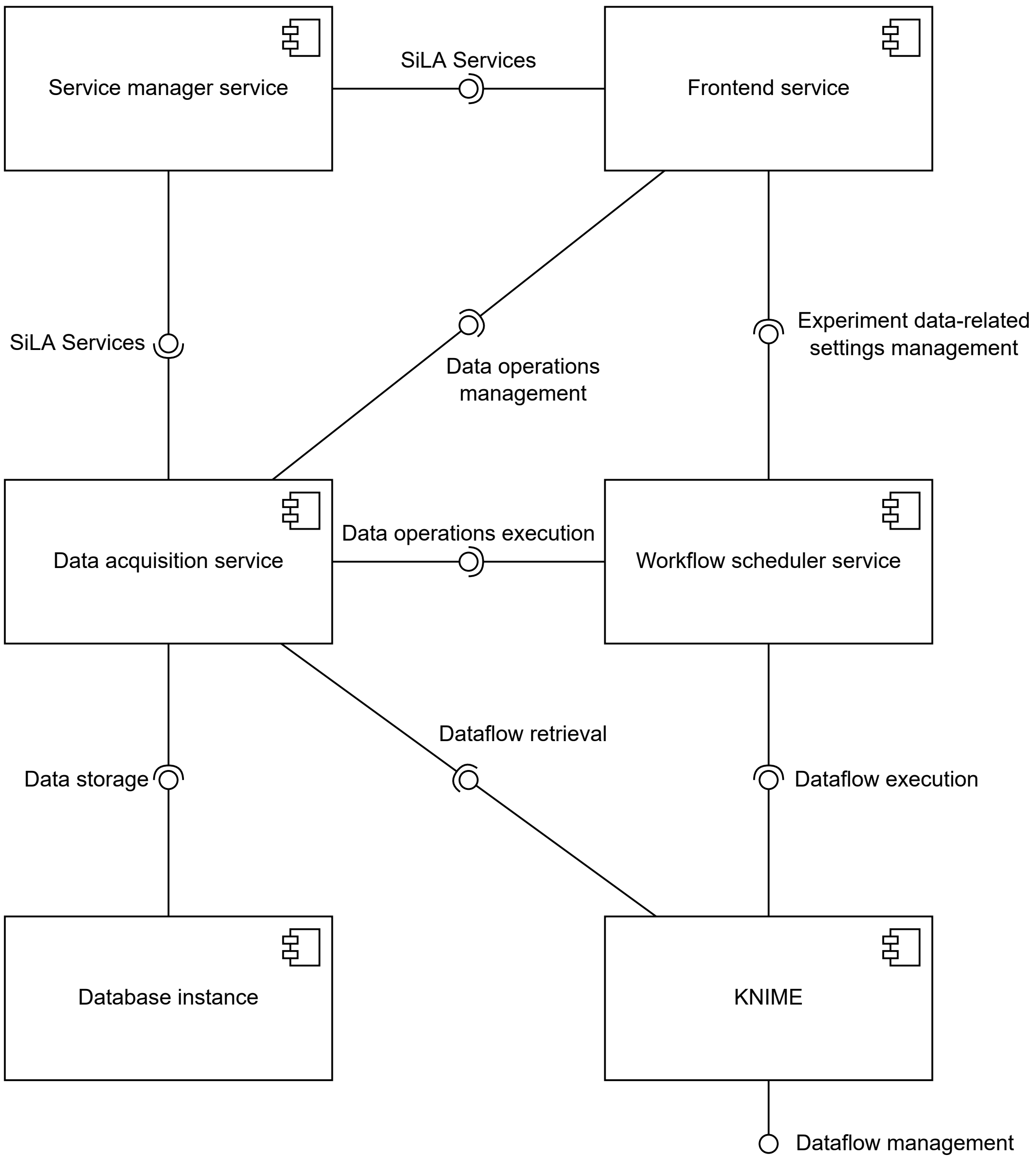

Figure 4.1: Interactions between the different components of the SiLA 2 Manager

4.5 API Gateway pattern

Requiring the client to know and keep track of the location of every microservice it makes use of increases the complexity of maintaining the client.

Additionally, doing user authentication in every microservice is not the best approach, as this implies that every microservice needs to have information about all the existing users.

Considering the above drawbacks of having a direct communication between client and microsevices, the solution comes in the form of the API Gateway pattern. This basically means having an entity between the client and the microservices that centrally handles authentication and mapping of client requests to the appropriate microservices. It is also shown in Figure 4.2.

In our project, the API Gateway exists in the form of an actual microservice, the Backend gateway service, to which all client requests are sent. Here they are authenticated, then forwarded to the appropriate microservice. The response of a microservice is sent back to the Backend gateway and from here it is sent to the client.

4.6 Storage

As mentioned before, there are a few aspects that are fixed given the requirements of the system. The first of these is data storage, given that one of the main requirements is to store experiment data. On top of experiment data, in order for the application to fulfill its functions, it is necessary to store additional data, such as protocols.

What is different for experiment data compared to application data is that experiment data will be in the form of time-series data: we will have measurments of some proeprties that are recorded at specific times and have some values.

Therefore, not only do we require to store data, we actually have different types of data that are more appropriate for specific database types, bacause of this two database types will be used. The more traditional relational database will be used to store application data and a time-series database will be used for experiment data.

4.7 Websockets

An additional aspect that is fixed is the usage of websockets. This is required because of how the Service Manager allows data retrieval for observable commands and properties: upon request, it opens a stream of data, a websocket, through which information is sent.

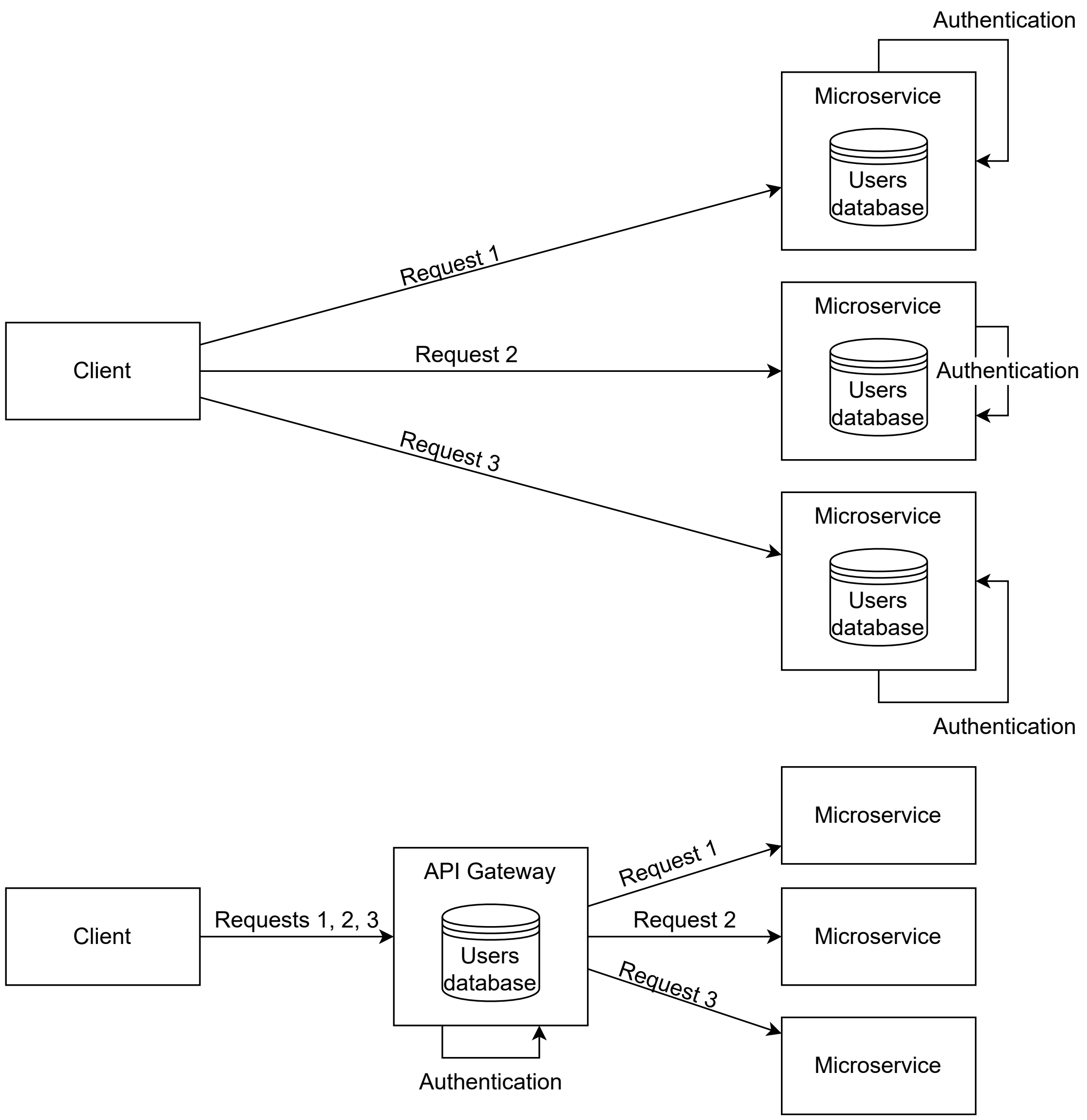

Figure 4.2: Overview of direct client-microservices communication (top) and client-microservices communication through an API Gateway (bottom)

5 Implementation

This chapter will identify the technologies used for development and will go into the implementation details of the Data acquisition service and into the changes erquired in the other services.

5.1 Server implementation

For the implementation of the server Python [22] and the FastAPI [11] framework were used.

Python is one of the most commonly used programming languages for web application development. However, it is not just Python that is used, as managing certain aspects, such as security and routing requests to specific endpoints, is highly complex. Therefore, we make use of a framework that takes care of all these operations that are common to web applications in general.

The framework of choice for this project is FastAPI as it is designed to be easy to use and provides, among others, the following features: authentication, data validation and seralization, automatic documentation.

5.1.1 Endpoints HTTP method best practices

As mentioned before, REST endpoints will be implemented to provide functionalities and allow operations on the resources of the Data acquisition service. These endpoints will be required to have HTTP methods assigned to them. The most common methods and the ones we will use are GET, POST, PUT and DELETE.

The HTTP methods will be assigned to endpoints depending on what sort of operations an endpoint will perform on resources. The best practices for HTTP method assignment are as described below.

If an endpoint will only retrieve data and not modify anything, it will be of type GET. An endpoint that will create a new resource on the server will have the POST HTTP method. Updating an existing resource will have type PUT. Finally, an endpoint that deletes a resource from the server will have type DELETE.

5.2 Storage

As described before, there are two types of data that need to be stored. Those two types are application data, such as protocols and databases, and experiment data, that comes in the form of time-series data. Because each of these two types of data is more appropriate for a specific type of database, we will use a relational database for application data and a time-series database for experiment data.

5.2.1 Application data

With our application data having a well defined structure and being, in some sense, more standard, it is more suitable for a relational database. For our project, the choice was made to use PostgreSQL [20], as it is one of the best ranked relational databases [32][6], it is open source and provides good reliability, scalability and security.

ORM

ORM stands for Object-Relational Mapping and is a technique that allows database interactions in an object-oriented manner. This helps reduce the amount and complexity of code that needs to be written by abstracting the low-level database queries.

ORM is also used to refer to the tool, the Mapper used to accommodate the technique. Having decided on using Python, it is necessary to choose an ORM that can work with it.

For our project SQLAlchemy [31] was chosen as it is an open source, well-known and industry grade ORM whose capabilities have been proven in large organizations.

5.2.2 Experiment data

Experiment data differs from application data in the sense that measurements might have various responses and metadata attached to them, so experiment data does not have such a strict structure as application data.

Moreover, with experiment data we are interested in the evolution of some aspect over time, therefore we will have time-series data, meaning that there will be additional focus on the time aspect of measurements.

It is important to choose the most appropriate type of database for data and, considering the above aspects, the one that suits our needs the best is a time-series database. The choice for this aspect of the project has been InfluxDB [14], as it is open source and is the top ranked time-series database [7]. The version we will use is InfluxDB 1.7 [15].

The way InfluxDB records data is, first of all, by storing the timestamps of data, as this is the key principle of a time-series database.

Additionally, there are fields that store actual data in the form of key-value pairs (basically, the name of the data and its value). It is important to note that fields are required, therefore we cannot have an entry in InfluxDB without a field. Also, they are not indexed, meaning that queries on fields will be slower.

InfluxDB also has tags that store metadata, again as key-value pairs. They are not mandatory in order to have an entry. Compared to fields, tags are, however, indexed. Therefore, queries on tags perform better.

The timestamp, fields and tags are stored in what InfluxDB calls a 'measurement'. A measurement is similar to a table in a relational database.

Data in InfluxDB also has a retention policy asosciated with it that specifies how long it is stored.

And the final concept, InfluxDB can have multiple databases. A database can contain the information the was described above.

InfluxDB does not enforce a specific schema, making it possible to store information with varying structure. This is an advantage considering that experiment data can have various names for metadata and responses.

Using InfluxDB helps us cover the A1.1 and A1.2 principles of FAIR data that state the communications protocol for data retrieval should be open, free, universally implementable and allow for authentication and authorization. InfluxDB covers this by providing data retrieval through an HTTP API that allows authenticaion.

Web interface

The creators of InfluxDB have also developed Chronograf [5], a tool that provides a web user interface for visualization of data.

5.3 Microservice template

Considering the above used technologies for backend development, a template [13] was identified that allows an easy initial setup for a new microservice using Python, FastAPI and PostgreSQL.

Additionally, it provides a Vue frontend, however the decision was made to use Angular for the frontend as the rest of the SiLA 2 Manager already uses it.

5.4 Frontend

As mentioned, the frontend for the new Data acquisition service will be developed in Angular [1] to allow straightforward integration with the rest of the interface for the

SiLA 2 Manager, also developed in Angular. Additionally, it is open source and is one of the most used web development frameworks [33].

The frontend is also a separate microservice.

5.5 Microservice deployment

Easy deployment of microservices is required, both in a production environment and during development operations.

For this purpose Docker [8] was chosen that allows deployment of services in containers that can run in any environment. Compared to virtual machines, which not only contain application code, but also contain an operating system, Docker containers leverage resources of the host operating system which allows them to only contain application code. Therefore, this has the advantages of making them smaller in size and faster to execute. Additionally, they require fewer resources than virtual machines.

Additionally, Docker provides a tool, Docker Compose [9], that allows simple execution of applications composed of multiple services in Docker containers. This is used to execute the SiLA 2 Manager, of which the Data acquisition service will be a part of.

5.6 Authentication

As mentioned before, the API Gateway pattern will be made use of in the form of the Backend gateway. As this will be the entry point for the backend microservices, this is where authentication will be done, instead of doing it in every microservice and thus requiring multiple copies of the users database.

If authentication is successful, the received request will have the user information attached to it and it will then be forwarded to the appropriate microservice.

To provide some additional information, authentication will be token based. A user will be required to first log in, after which a token will be generated for that user. This token must be added to requests. When receiving a request, the Backend gateway will either accept or reject the request based on the validity of the provided token. Of course, if no token is provided, the request will be rejected.

Regarding user credentials, passwords are never stored. What is actually stored when the user sets a password is the hash of that password. How logging in works is that the user provides a password, this is then hashed and compared against the value the server has. If the hashed values match the user is logged in, otherwise the password provided by the user at log in was not correct.

This mechanism of using the hash of a password instead of the actual password has the benefit that, in case of a security breach, the actual passwords of users are not compromized.

5.7 Authorization

The difference between authentication and authorization is that authentication verifies the identity of a user, while authorization verifies whether the user is allowed to perform a certain action.

As it is the case that most actions need to be authorized based on information that is only available in a microservice, the authorization process will take place in the microservice that is responsible for handling a specific request. Having it done in the Backend gateway would increase its complexity and the number of requests exchanged, thus going beyond its intended purpose of being a simple microservice that provides mechanisms for authentication and request mapping.

To provide a quick example for choosing not to do authorization centrally in the Backend gateway and instead do it in the respective microservice we can choose the action of updating a protocol that should only be permitted to the original creator of the protocol. Authorizing the operation in the Backend gateway would require retrieving the owner of the protocol from the Data acquisition service, comparing it with the user that sent the request and then, if authorized, forward the request to the Data acquisition service. This implies unnecessary requests, when it is possible to forward the request to the Data acquisition service, which holds the information about protocols, and let it compare the incoming user with the owner of the protocol on which an update is attempted. This comparison is also shown in Figure 5.1. To note that in both cases the protocol will be retrieved by the Data acquisition service from its database.

5.8 Data acquisition service

To continue, the implementation details of the Data acquisition service will be presented.

5.8.1 Application data classes

First, the application data classes will be discussed. The ones we require are protocol, together with the other classes (such as service, feature and so on) that are part of it, and database.

The way in thich the template we are using works regarding data classes is that we need two types of classes for the same concept (for example, for a protocol), those two being schema and model.

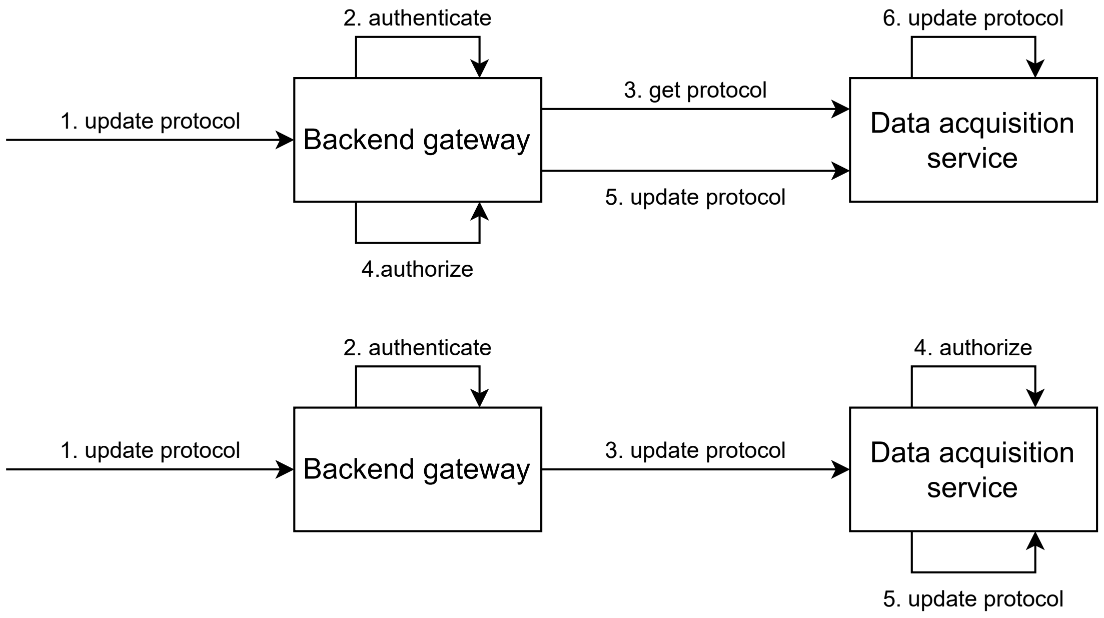

Figure 5.1: Comparison of authorization done in the Backend gateway (top) and in the responsible microservice (bottom)

Models

Models are required for the use with the the ORM, SQLAlchemy, to allow database interactions using Python objects. Additionally, they define the structure of the database. They provide information such as the name of the columns, their types, primary and foreign keys, relationships between the different tables. For our implementation, the primary keys will be numbers and they will be automatically generated.

Schemas

Moreover, we use Pydantic [21] models (to which we will refer as schemas, with the purpose of diferentiating from the SQLAlchemy database models) that we need for data validation and serialization when receiving input from and providing output to the endpoints. Basically, Pydantic helps us abstract the task of converting objects to and from JSON for communication between microservices.

There is also the case that for one concept we have multiple schemas. The reason for this is to differentiate between what is expected for an object to contain based on the different operations that are performed. The general structure is that we will have an ObjectCreate class that specifies what attributes are required as input for object creation, an ObjectUpdate class that specifies what attributes are required as input for object update and an Object class that specifies what the attributes the object returned as result of an operation should have. In general, an ObjectCreate class will enforce the presence of certain attributes that the object cannot exist without, while ObjectUpdate will allow them to not be specified for the reason of leaving them unchanged. An Object class will attempt to return all the information available about an object, even if some of it may be undefined. It will additionally contain the id the object has in the database.

Common information

Our application data classes will also record the id of the owner, the original creator, as to only allow certain operation to this user. Moreover, they will also contain the title of the object and the email address of the owner, however these will only be used for display purposes. The title will simply help the user know what object the application refers to, instead of just relying on ids. The email address of the owner is also just informative. Additionally, storing this in every object will make it simpler to show the owner without the need of actually querying the user from the Backend gatewy. Therefore, this saves us sending a request at the cost of a small increase in object size.

The general structure of the application data classes we will use (regardless of models and schemas) is shown in Figure 5.2 and will be discussed in more detail. It varies a bit between models and schemas and between the different schemas in a few ways, some of which are described above. To go through the most important ones, models have ids and, in case of objects contained by some other objects, references to the parent objects in the form of foreign keys and allow certain relationships and operations to be defined. Additionally, contained objects do not record owner id and email address as this is information already recorded by the topmost object.

Protocol

A Protocol is what will be used to specify what information we are interested in retrieving and storing from a SiLA Service.

In order to fulfill its purpose, a Protocol will have an id to allow for unique identification when performing various operations on it. A title will be used for display purposes to offer the user richer information, but otherwise will have no functional purpose. The id of the owner will be recorded in order to only allow certain operations to be performed by this user. Similar to the title, the email address of the owner is only informational and does not serve a functional purpose. Additionally, storing this allows us to save a request (the one of retrieving user information from the Backend

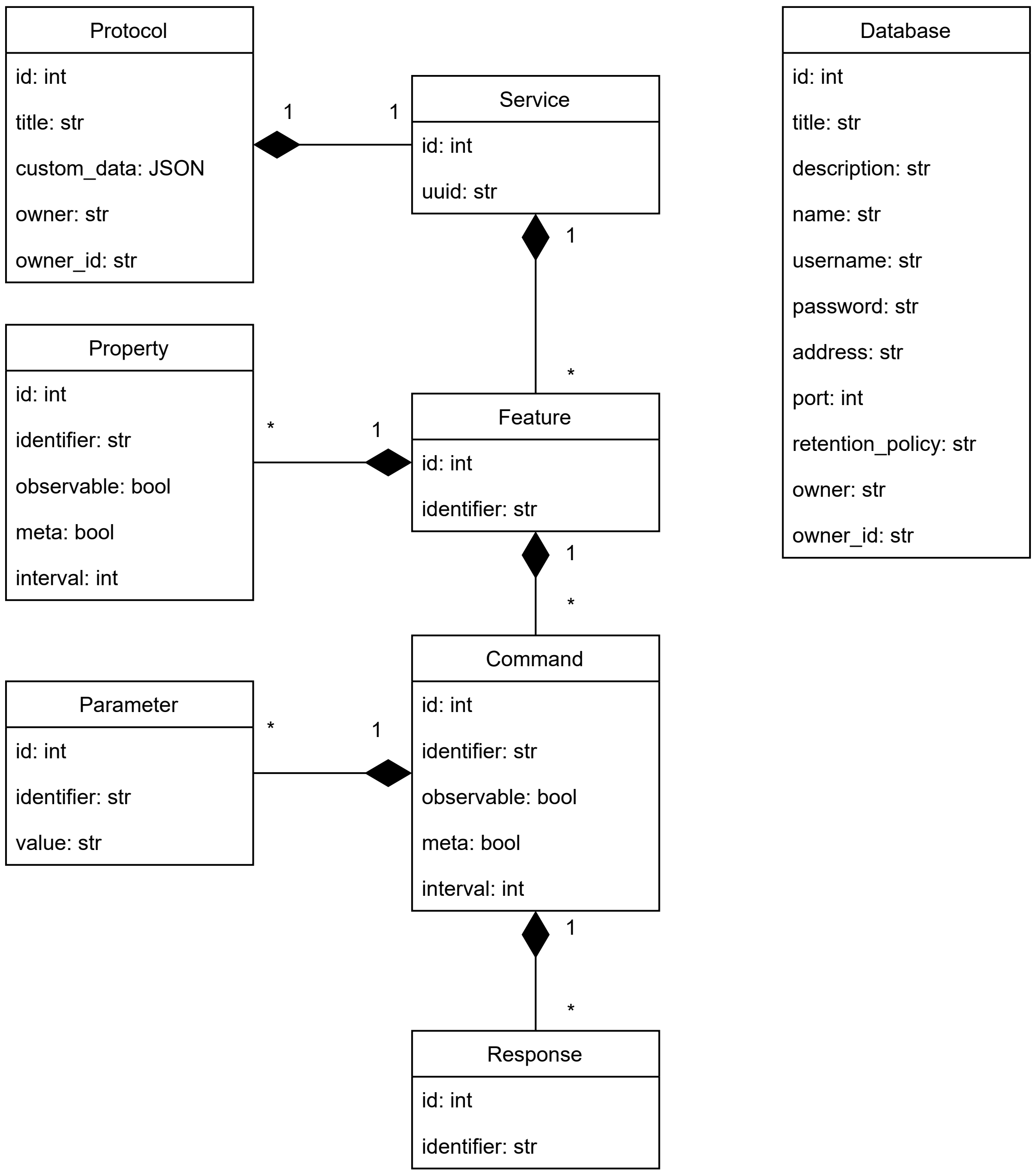

Figure 5.2: Structure of the application data classes

gateway).

Having gone past the more general information, a Protocol also records custom data that we are interested in storing. It also contains a Service object that will further solve the task of identifying what information from a service is relevant.

What is important to mention here is that the relationship between a Protocol and a Service contains a cascading delete orphan operation. This means that when a Protocol is deleted, the services it contains will also be deleted. It also means that on Protocol update, if a Service is left without a parent (it is no longer part of a Protocol), it will also be deleted.

There is one issue to discuss here. As described before, schemas are used for interaction with requests and models are used for interaction with the database. As it would be expected, at some point we need to convert between a model and a schema. While this works as expected for simple objects, an issue has been identified that such schema-model conversions do not complete successfully for nested objects. Because of this, it was required to implement a mechanism to execute the conversion of protocols with all the objects they contain.

Service

A Service is used to identify the SiLA Service from which we want to retrieve and store information as part of the protocol. As expected, it has an id for unique identification. Furthermore, it records the uuid of the SiLA Service (the Service Manager uses a service's uuid to identify the service on which to perform an operation). Additionally, a Service contains a list of features that are relevant for us.

The reason for not using the service's uuid as unique identification of a Service is that different protocols might require information from the same SiLA Service, but different features are of interest, therefore one single Service is not enough to represent this.

The relationship between Service and Feature also contains a cascading delete orphan operation, similar to the one between a Protocol and a Service.

Feature

A Feature will contain information about a SiLA Feature that we are interested in. Same as before, an id is present. The identifier of the SiLA Feature is also recorded. The reason for not using the identifier as a primary key is the same as not using a Service uuid as the primary key, which is described above.

Commands and Properties we are interested in are contained in a Feature. Same as before, cascading delete orphan operations are put in place for Feature-Command and

Feature-Property relationships.

Property

A Property identifies a SiLA Property that is to be executed. It will record an id, the identifier of the SiLA Property and whether the SiLA Property is observable or unobservable. It also informs whether the SiLA Property should be treated as metadata or actual data, and this provided by the user. Finally, a Property will also have an interval that specifies how often data should be queried and stored. However, this is only relevant in case of unobservable and actual data.

Command

A Command will contain the same information as a Property, but in this case regarding a SiLA Command. On top of this, SiLA Commands also have parameters and responses, therefore a Command will contain a list of parameters and a list of responses.

Cascading delete orphan operations also exist for the Command-Parameter and Command-Response relationships.

Parameter

A Parameter is basically used to provide a value for a parameter used in the execution of a SiLA Command. It will have an id and record the identifier of the SiLA Parameter. Additionally, it will have a value to be given to the parameter upon command execution.

Database

A Database is used to store the details of the InfluxDB database to be used for storage after retrieval of experiment data. Same as the other application data, a Database will have an id, a title, an owner id and email address that have the same purposes as in the other application data. Additionally, also to provide the user with richer information and without having actual functionality, a Database will have a description.

To go into what information about an InfluxDB database is relevant, we have the name of the database to use inside of an instance of InfluxDB. The location of the database, in terms of IP address and port, is recorded. The username and password required for authentication are contained here as well. Finally, the retention policy to apply to the stored data is provided.

Database status

There is an additional class that we make use of which only comes in the form of a schema and not that of a database, as it is not information that is required to be persisted. This class is DatabaseStatus and we use it to return the status of a database. Given that an InfluxDB instance is external to the Data acquisition service, its status can change at any time and therefore needs to be interrogated on the spot. This is why it does not require persistance.

This class specifies whether or not the information in a Database object can actually be used for experiment data storage in an InfluxDB instance. This is simply reported in the form of a boolean value. Additionally, the DatabaseStatus class also contains a field for a message, with the purpose of providing additional infomation.

5.8.2 Functionality

This section will present the implementation details of the functionalities offered by the Data acquisition service.

Protocol management

Protocol management functionality is provided through endpoints that allow the expected operations of creating, reading, updating and deleting protocols.

Retrieve list of protocols

A GET endpoint is offered that allows retrieval of multiple protocols. In order to not return all of the protocols and avoid impacting application performance, as the list could be extremely large, it is possible to specify how many protocols are requested and how many to skip from the list of those that can be offered.

The protocols that can be offered to a user are either those that the user has created, or all of them in case the user is a superuser.

The respective protocols, as described in the criteria above, are retrieved from the database and returned.

Retrieve one protocol

It is also possible to request the retrieval of a single Protocol specified by id through another GET endpoint. This will take as input the id of the Protocol to be retrieved.

It will be attempted to retrieve the Protocol specified by id from the database. In case it doesn’t exist, an HTTP exception containing this information will be raised.

If the specified Protocol does exist, it will only be provided to the requesting user if the user is a superuser or the creator of the protocol. If this is not the case, another exception will be raised with the information that the user does not have enough permissions.

Protocol deletion

A Protocol can also be deleted through the execution of a DELETE endpoint that takes the id of the Protocol to delete as input.

All the steps from the retrieval of one protocol are performed, with the addition that if the user has permissions to operate on the respective Protocol, this will be deleted and then returned. As part of the deletion, the objects contained in a protocol (such as service, feature and so on) will also be deleted as a result of the cascading delete orphan operation put on the relationships between the objects.

Protocol creation

Protocol creation is a more complex operation, mainly because the information provided needs to be checked against the actual information in the Service Manager and it needs to be verified that function execution is possible given the provided function information.

It is provided by a POST endpoint that takes as input a Protocol. First, it is checked that the information provided in the Protocol is valid and matches information available in the Service Manager. Some examples of what is checked are does this service actually exist in the Service Manager, does this command actually exist, does it actually have these parameters. Afterwards it is attempted to execute the functions specified in the protocol to ensure that execution is possible for the provided function information. How function execution actually happens will be later discussed in more detail.

If all of the above is successful, a new Protocol is created with the user that sent the request as the owner and then returned.

The process is also shown in Figure 5.3.

Protocol update

This operation is exposed in the form of a PUT endpoint. Updating a protocol is similar to creating a protocol with respect to input parameters and actions performed. The id of the Protocol to update must also be specified. What is additional to protocol update is, first of all, that a Protocol with the specified id must already exist in the database and, secondly, the user that sent the request must be a superuser or the owner of the Protocol.

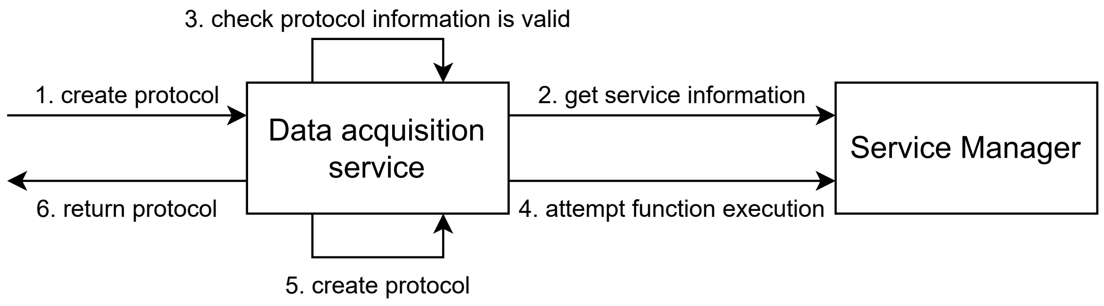
3. check protocol information is valid
1. create protocol
Data acquisition service
2. get service information
Service Manager
6. return protocol
4. attempt function execution
5. create protocol

Figure 5.3: Steps required for protocol creation

Existing objects that are not anymore part of the Protocol as a result of the update are automatically deleted.

Database management

Database operations follow the same pattern, flow, as the operations for Protocol management. The differences come in the form of the checks that need to be done as part of Database creation and update.

For databases it is required to check that the given information can actually be used to store data. This means, first of all, that an instance of InfluxDB is actually running at the provided IP address and port. Then it is important to check that authentication is successful for the provided username and password and that the user has admin rights. Finally, the information provided in the retention policy field must either match an existing retention policy or be successfully used in creating a new retention policy in the database (inside of InfluxDB) specified by the name field in the Database object.

The process for Database creation is shown in Figure 5.4.

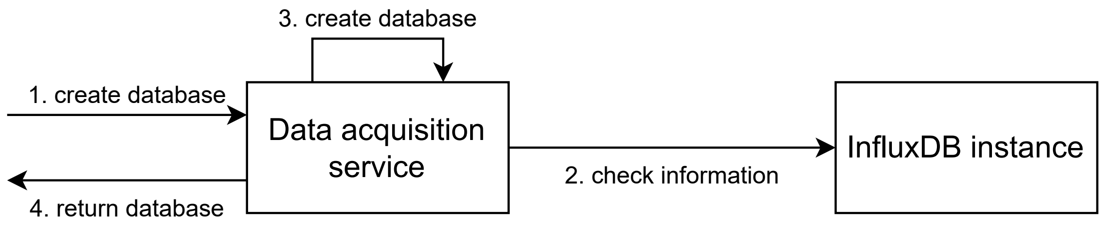

Figure 5.4: Steps required for database creation

In order to do all these database related checks we make use of the InfluxDB-Python [16] library that allows interactions with an InfluxDB instance.

Database status interrogation

There is an additional operation that can be performed as part of Database Management and this is the interrogation of the status of a Database.

This is exposed through a GET endpoint that takes as input the id of the Database on which to perform the interrogation. First, it is required to actually retrieve the respective Database object from the storage of the Data acquisition service. The Database specified by id needs to actually exist and the user must be authorized to perform operations on it. Then, the same checks as for database creation and update are performed and a DatabaseStatus object is created and returned.

This process is shown in Figure 5.5.

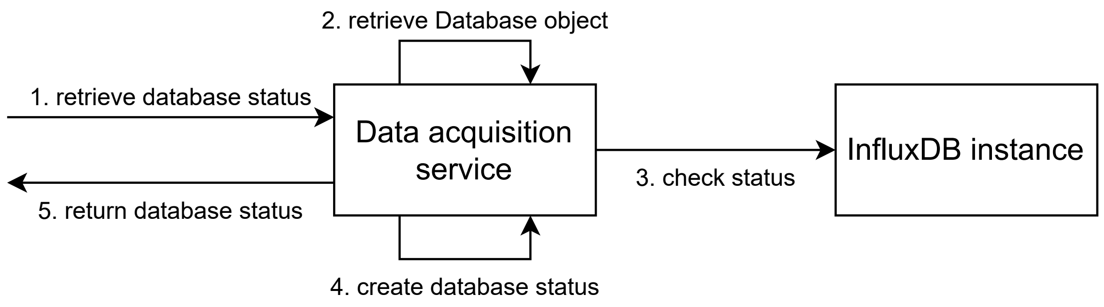
2. retrieve Database object
1. retrieve database status
Data acquisition service
InfluxDB instance
3. check status
5. return database status
4. create database status

Figure 5.5: Steps required for database status interrogation

Dataflow management

The only part of dataflow management that is performed by the Data acquisition service is the retrieval of the list of available dataflows. This is offered through a GET endpoint that makes a GET request to the KNIME server that provides the available dataflows. This is shown in Figure 5.6.

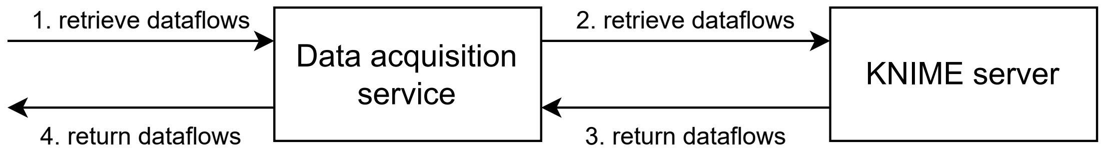

Figure 5.6: Dataflow retrieval

The rest of the Dataflow management operations are provided directly through KNIME, as it already offers what we require and therefore it was not needed to reimplement the functionality. The KNIME server offers storage, management and execution of dataflows. It provides endpoints for certain operations. The KNIME Analytics Platform is an application that acts as a user interface to interact with the server.

Data retrieval and storage

This is the most important part of the Data acquisition service that actually allows automating data retrieval and storage in a laboratory environment. This functionality is exposed in the form of a POST endpoint that takes as input a list of pairs of protocol and database, an experiment id and the id of the user responsible for the experiment. The reason that multiple protocols are needed is that we might want to handle data from multiple services and a protocol can only handle data from one service. Additionally, we need to pair every protocol with a databse for the purpose of allowing data from different protocols to be stored in different databases.

First of all, functions from all protocols are separated into four categories based on whether they are observable or unobservable and metadata or actual data. Execution based on interval is only required for unobservable actual data, as observable execution behaves differently and does not require an interval and regarding metadata, we only want it stored once.

For unobservable actual data we require to separate functions based on the intervals at which they are to be executed repeatedly.

We have implemented a method that performs one execution of unobservable SiLA functions and storage of the responses. The functions are executed through a call to an endpoint of the Service Manager. For command execution we will need to provide values to the parameters and specify the responses we want. Properties do not have parameters or multiple responses. Upon receiving the response, this will be stored in the specified database through the use of the InfluxDB-Python library. If execution of a function is not successful, this will be recorded in the database. If any other error occurs, such as the database is not available anymore, it will be logged.

Now that we have a method to retrieve and store unobservable data, we need a way to repeatedly execute it based on the different intervals specified for functions. Two main possibilities have been identified for this. One of them is to have Docker containers that execute the above discussed method. However, creating an entire container for the execution of a single method is somewhat wasteful. Therefore, the second possibility, which was also chosen for implementation, is to use APScheduler [3] (Advanced Python Scheduler). It offers functionality to execute code at specific intervals and at specific times. The one we are interested in is interval-based execution.

Therefore, for periodic retrieval and storage of unobservable actual data we instruct APScheduler to execute our method at a specific interval. We do this for every interval.

For unobservable metadata we simply use APScheduler to execute the method once.

For observable actual data we need to proceed in a different way. The way the Service Manager offers execution of observable functions is that it first takes a request to start the function, opens a data stream and then returns the name of the data stream. Afterwards, it is possible to connect to this data stream through which the Service Manager will send data. In the Data acquisition service we have another method that handles this. It calls the endpoint in the Service Manager to request the creation of the data stream. Then, the received name is used to connect to the data stream. For this we use the websockets [35] library for Python in order to create a client that allows us to accept the data that comes through the data stream from the Service Manager. When new data is received, it is stored in the respective InfluxDB instance. If an error occurs, but the InfluxDB instance is still available, we store this information. Other errors that occur are logged.

The last type of information to retrieve and store is observable metadata. For observable properties it is possible to treat them as unobservable properties and simply ask for one unobservable execution with the method described above. Observable commands do not support this and therefore it is needed to first call the Service Manager in order to open a websocket, connect to that websocket, accept and store the first piece of information received and then close the websocket, as for observable functions we are only interested in recording the value once.

To go into the details of what we actually store after the execution of the functions, first of all we are interested in the experiment the data is part of. We are also interested in the protocol, service, feature and function identifiers for the purpose of knowing where the data came from. The user responsible for the experiment is alse recorded. For commands we also store the used parameters. The information described so far is stored as tags. As for fields, this is where we store the actual data that came from command execution or, in case of errors, we store these. The database to use inside of an InfluxDB instance, the name of the measurement and the retention policy also need to be specified.

Considering the additional data we store together with the responses returned by function execution and other metadata retrieved from functions, the F2 principle of FAIR data is covered. This principle states that data should be described with rich metadata.

Unobservable and observable function execution is shown in Figure 5.7.

We also need to store the custom data specified in the protocol, but this is the most straightforward operation because we already have the information we want to store and there is no need to retrieve it from somewhere else. It simply needs to be stored.

Finnaly, we also offer a GET endpoint that takes as input the id of an experiment and stops the data retrieval and storage process for this experiment.

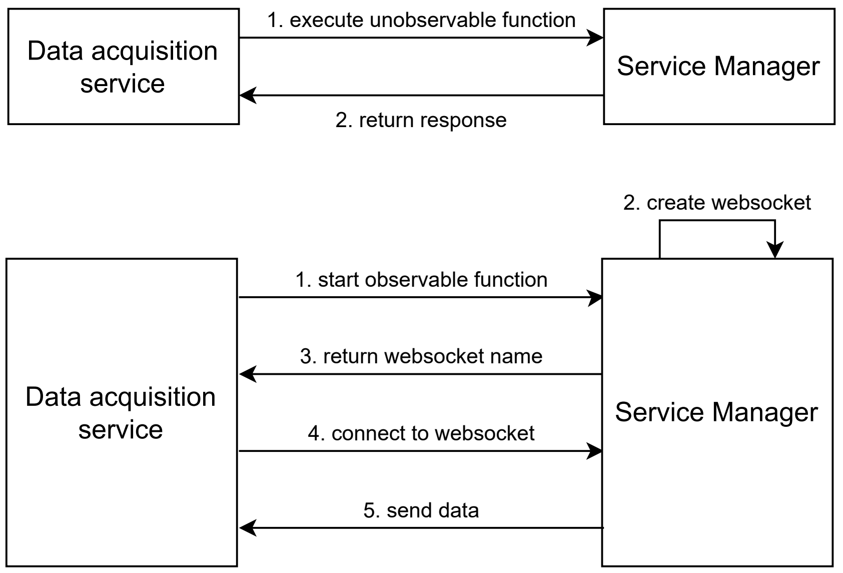

Figure 5.7: Execution of unobservable function (top) and observable function (bottom)

5.9 Scheduling

As mentioned before, it is required to perform data retrieval and storage for the duration of an experiment and dataflow execution at the end of an experiment. The scheduling mechanism is external to the Data acquisition service and is already implemented in the Workflow scheduler service as part of experiment execution. This service also handles some of the existing experiment settings. Protocols, databses and dataflows need to be added.

What we need to do, first of all, is to add new fields to the experiment classes. We will add a field that can store a list of pairs of protocol and database. For this to work, we are also required to add classes for Protocol and Database in the Workflow scheduler service. We also need to add a field to the experiment class that will store the path to a dataflow.

Then, on experiment creation, a list of pairs of protocol id and database id and a dataflow path can also be provided. Note that in this case we have protocol id and databse id, instead of the actual objects. This is because requests will be made to the Data acquisition service in order to retrieve the actual protocols and databases, that will then be added to the experiment.

The reason for providing protocol and database ids instead of the actual objects is because if we would receive an object we wouldn't know if the information in it is valid. Retrieving protocols and databases by id from the Data acquisition service ensures they are corect because the Data acquisition service checks this on creation and update.

The reason we also store protocols and databases in the Workflow scheduler service at the time of experiment creation is because we need to fix the information in protocols and databases so that it doesn't change. In this way we can make sure that what is described in protocols and databases at the time of experiment creation stays the same until experiment execution. Otherwise, someone might come and change protocols and databases between experiment creation and experiment execution, thus we will end up with different information than what we selected initially.

Additionally, the Workflow scheduler service already handles experiment scheduling and execution at specific times. What we need to add is a call to the Data acquisition service at the beginning of the experiment in order to start the data retrieval and storage process. At the end of an experiment we need to make a call to the Data acquisition service to stop the process and a call to the KNIME server to start the dataflow. This scheduling is also shown in Figure 5.8.

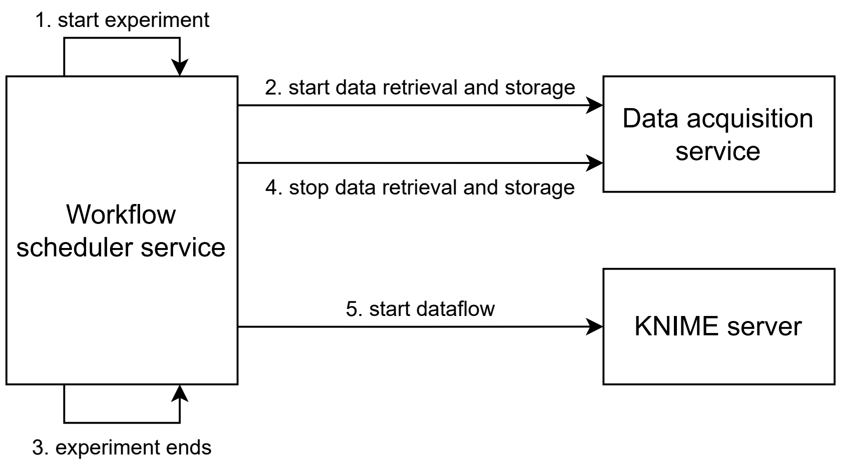
1. start experiment
2. start data retrieval and storage
Data acquisition service
4. stop data retrieval and storage
Workflow scheduler service
5. start dataflow
KNIME server
3. experiment ends

Figure 5.8: Scheduling of data retrieval, storage and processing

5.10 Backend gateway

The additions that were required for the Backend gateway mainly came in the form of mapping of requests to the new Data acquisition service and application data classes for validation and serialization of data of the Data acquisition service.

5.11 Frontend

The Frontend is another microservice that aready existed in the SiLA 2 Manager. It is a user interface that offers easy interaction with the functionalities of the other microservices.

What was needed to implement here, on top of what already existed, was protocol, database and dataflow management and settings for experiments.

Angular allows creation of data classes. We implemented such classes for protocol and database, that are similar to the data classes in the Data acquisition service.

As for interaction with the Data acquisition service endpoints, some service classes were implemented that provide methods to call the various endpoints.

Regarding the actual user interface, it allows viewing the KNIME dataflows and some information about them. Additionally, it provides links to their documentation page on the KNIME server.

For databases, the expected operations of database management are provided. A page displaying a list of databases and their information is offered. From here it is possible delete databases and access the Chronograf web interface (note that this is external to the SiLA 2 Manager and is not guaranteed to be running). Additionally, from here it is possible to go to the update page of a database. Database update and creation are very similar, with the difference that on database creation empty fields are provided and on database update the fields are pre-filled with the information of the database to update.

For protocols the interface is very similar. One of the differences is the fact that on the page displaying a list of protocols it is possible to expand a protocol by clicking on it to view all its information. This is done in order to not clutter the page, as protocols can have quite a lot of information. Additionally, protocol creation and update is a bit more complex. For databases, the user simply filled in some fields. For protocols, it is the case that for some inputs the user can only select from some existing options. The user can only select a service from the available services in the Service Manager. For this reason, the Frontend must perform a request to the Service Manager to retrieve the SiLA services. After a service is selected, the list of the available features for this service is presented to the user. This also need to be retrieved from the Service Manager. Afterwards, a list of available functions (retrieved from the Service Manager) for a feature is presented to the user. This process is presented in Figure 5.9.

There is already a page that allows providing some settings to an experiment. We additionally need to implement the selection of protocols, databases and dataflow. Protocols and databases will be added in pairs.

In order to add a pair of protocol and database to an experiment, the user must select a protocol from the existing ones and a database from the existing ones. Multiple pairs

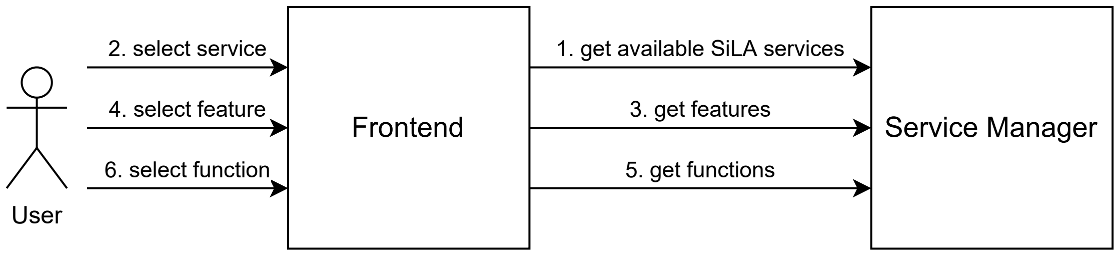

Figure 5.9: Frontend selection of function for protocol

can be added.

A dataflow is simply selected from the existing dataflows.

6 Evaluation

In order to check the correct execution of data retrieval, storage and processing, multiple runs were performed. Multiple SiLA services have been started to check retrieval of data from multiple sources. Multiple protocols were created to check that every protocol is executed correctly. Various functions and intervals were used inside of the protocols. Custom data was also provided. Multiple instances of InfluxDB were started and multiple Database objects were created to check that data is correctly stored in the specified place. Dataflows were also created and provided.

Multiple runs of the process were executed successfully and correctly.

There is one important aspect to note. At the time of writing, observable function execution was still in experimantal state and was not integrated into the development branch. Therefore, execution of observable functions as part of the data retrieval process was only possible to be verified separately with the experimental version of the Service Manager endpoint to execute observable functions. As a stable version for observable function execution is not available in the development branch, the part of the Data acquisition service that performs retrieval and storage of observable function data is currently disabled. However, the way in which observable function execution is provided by the Service Manager is not expected to change from the experimental to the future stable version.

7 Conclusion

Manually managing a laboratory, the experiments that take place and the data resulting from experiments is a highly complex task. The overall SiLA 2 Manager aims at providing laboratory automation. This thesis was focused on the development of a specific part of the SiLA 2 Manager, namely the Data acquisition service that has the purpose of automating data operations.

The three main data operations that we are concerned with are retrieval, storage and processing. The Data acquisition service allows a user to specify what data from what devices is relevant for an experiment, so that this data is automatically retrieved during the experiment. Additionally, the user can also specify where the data should be automatically stored after it is being retrieved. Moreover, processing of data can be defined beforehand and then automatically executed at the end of an experiment.

In other words, the Data acquisition service aims at providing a uniform interface for customizing data operations regardless of experiment and device type. Additionally, it automates the execution of the data operations.

The source code can be found here:
https://gitlab.com/lukas.bromig/sila2_manager/-/tree/alex-mardale-thesis

8 Outlook

There are a few aspects that could be implemented or improved and these will be discussed here.

8.1 Redundant schema checks

At the moment data validation for a request is performed both in the Backend gateway and in the responsible microservice. While this does not particularly affect the performance of the application, it could be considered redundant. Additionally, it also puts a bit more responsability on the Backend gateway, while its intended purpose is to provide authentication and request mapping.

However, the double data validations are performed throughout the entire SiLA 2 Manager and for consistency were also included in the Data acquisition services. Changing this would require a decision at the level of the entire SiLA 2 Manager.

8.2 Model duplication

In order to perform data validation in the Backend gateway it is required to have the data models there as well, in addition to having them in the respective microservices. This leads to code duplication and increased complexity when models are changed, as modifications are required in multiple places. Additional model duplication occurs in our case because of also saving protocols and databases in the Workflow scheduler service to ensure the information is fixed.

A mechanism for sharing data models between microservices would be required in this case. However, this is also a decision to be made at the level of the entire SiLA 2 Manager.

There is, additionally, duplication between the different data models of a microsevice. This occurs because of recording object id, owner id and email address at the level of individual models. These are attributes shared by most of the models. A possible solution would be to extract a base class, from which the data models can be subclassed, that contains the common attributes, so they don't have to be mentioned in every data class.

8.3 Frontend

The user interface is another aspect that could be improved, the main reason being my lack of experience with frontend development.

There are parts of the user interface that can be improved in terms of how they look, clarity. Additionally, when requesting some user inputs, validation of these inputs is minimal. To note that more consistent validation is done, however, in the backend. Finally, errors from the backend are not very well handled in the frontend, generally the errors are displayed directly.

8.4 FAIR data

While the title of the thesis refers to the FAIR data principles, not all of these principles were implemented in this project. Implementing all of the principles is more of a long term goal. At some point in the future it might be desired to make the data produced by the laboratory public for the purpose of reusability. However, at the moment, the entire SiLA 2 Manager is still under development.

The main purpose of this project was to provide customizable and automated data operations.

8.5 AnIML

It could be decided that AnIML is implemented in the future, this will also help with covering the FAIR data principles.

List of Figures

List of Tables

Bibliography

[1] *Angular*. Accessed: 2022-05-05. URL: `https://angular.io/`.

[2] *AnIML*. Accessed: 2022-04-30. URL: `https://www.animl.org/`.

[3] *APScheduler*. Accessed: 2022-05-08. URL: `https://apscheduler.readthedocs.io/en/3.x/userguide.html`.

[4] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, K. Thiel, and B. Wiswedel. "KNIME - the Konstanz Information Miner: Version 2.0 and Beyond." In: *SIGKDD Explor. Newsl.* 11.1 (Nov. 2009), pp. 26–31. ISSN: 1931-0145. DOI: `10.1145/1656274.1656280`.

[5] *Chronograf*. Accessed: 2022-05-05. URL: `https://www.influxdata.com/time-series-platform/chronograf/`.

[6] *DB-Engines Ranking of Relational DBMS*. Accessed: 2022-05-05. URL: `https://db-engines.com/en/ranking/relational+dbms`.

[7] *DB-Engines Ranking of Time Series DBMS*. Accessed: 2022-05-05. URL: `https://db-engines.com/en/ranking/time+series+dbms`.

[8] *Docker*. Accessed: 2022-05-05. URL: `https://www.docker.com/`.

[9] *Docker Compose*. Accessed: 2022-05-05. URL: `https://docs.docker.com/compose/`.

[10] *FAIR Principles*. Accessed: 2022-04-30. URL: `https://www.go-fair.org/fair-principles/`.

[11] *FastAPI*. Accessed: 2022-05-04. URL: `https://fastapi.tiangolo.com/`.

[12] A. Fillbrunn, C. Dietz, J. Pfeuffer, R. Rahn, G. A. Landrum, and M. R. Berthold. "KNIME for reproducible cross-domain analysis of life science data." In: *Journal of Biotechnology* 261 (2017). Bioinformatics Solutions for Big Data Analysis in Life Sciences presented by the German Network for Bioinformatics Infrastructure, pp. 149–156. ISSN: 0168-1656. DOI: `https://doi.org/10.1016/j.jbiotec.2017.07.028`.

[13] *Full Stack FastAPI and PostgreSQL - Base Project Generator*. Accessed: 2022-05-05. URL: `https://github.com/tiangolo/full-stack-fastapi-postgresql`.

[14] *InfluxDB*. Accessed: 2022-05-05. URL: https://www.influxdata.com/products/influxdb-overview/.

[15] *InfluxDB 1.7 documentation*. Accessed: 2022-05-05. URL: https://docs.influxdata.com/influxdb/v1.7/.

[16] *InfluxDB-Python*. Accessed: 2022-05-08. URL: https://influxdb-python.readthedocs.io/en/latest/.

[17] *KNIME*. Accessed: 2022-05-01. URL: https://www.knime.com/.

[18] *KNIME Analytics Platform*. Accessed: 2022-05-01. URL: https://www.knime.com/knime-analytics-platform.

[19] *KNIME Server*. Accessed: 2022-05-01. URL: https://www.knime.com/knime-server.

[20] *PostgreSQL*. Accessed: 2022-05-05. URL: https://www.postgresql.org/.

[21] *Pydantic*. Accessed: 2022-05-07. URL: https://pydantic-docs.helpmanual.io/.

[22] *Python*. Accessed: 2022-05-04. URL: https://www.python.org/.

[23] *Riffyn Nexus*. Accessed: 2022-05-01. URL: https://riffyn.com/.

[24] B. Schäfer. "Data Exchange in the Laboratory of the Future." In: *A glimpse at AnIML and SiLA Wiley Analytical Science, Hoboken, NJ: https://analyticalscience.wiley.com/do/10.1002/gitlab.17* (2018).

[25] B. A. Schäfer, D. Poetz, and G. W. Kramer. "Documenting laboratory workflows using the analytical information markup language." In: *JALA: Journal of the Association for Laboratory Automation* 9.6 (2004), pp. 375–381. DOI: 10.1016/j.jala.2004.10.003.

[26] *SiLA*. Accessed: 2022-04-30. URL: https://sila-standard.com/.

[27] *SiLA 2 Manager*. Accessed: 2022-04-30. URL: https://gitlab.com/lukas.bromig/sila2_manager/-/tree/idp-main/.

[28] *SiLA 2 Part (A) - Overview, Concepts and Core Specification*. Accessed: 2022-04-30. URL: https://docs.google.com/document/d/1nGGEwbx45ZpKeKYH18VnNysREbr1EXH6FqlCo03yASM/edit#heading=h.fqwkxeeqgcb4.

[29] *SiLA 2 standard*. Accessed: 2022-04-30. URL: https://sila-standard.com/standards/.

[30] *SiLA 2 standard implementations*. Accessed: 2022-04-30. URL: https://gitlab.com/SiLA2.

[31] *SQLAlchemy*. Accessed: 2022-05-05. URL: https://www.sqlalchemy.org/.

[32] *Stack Overflow 2020 Developer Survey - Databases*. Accessed: 2022-05-05. URL: `https://insights.stackoverflow.com/survey/2020#technology-databases-all-respondents4`.

[33] *Stack Overflow 2020 Developer Survey - Web Frameworks*. Accessed: 2022-05-05. URL: `https://insights.stackoverflow.com/survey/2020#technology-web-frameworks`.

[34] *User-friendly End-to-End Lab Automation in Action*. Accessed: 2022-05-01. URL: `https://www.knime.com/blog/user-friendly-end-to-end-lab-automation-in-action`.

[35] *websockets*. Accessed: 2022-05-08. URL: `https://websockets.readthedocs.io/en/stable/`.

[36] *What are the FAIR guiding principles and how to FAIRify your data*. Accessed: 2022-05-01. URL: `https://www.knime.com/blog/fair-guiding-principles-and-how-to-fairify-your-data`.

[37] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, et al. "The FAIR Guiding Principles for scientific data management and stewardship." In: *Scientific data* 3.1 (2016), pp. 1–9. DOI: `10.1038/sdata.2016.18`.