University of Southern Denmark

Master Thesis

Software Engineering

Safe Programming of a Cryopreservation Access Policy Framework

Niels Hvid
nhvid13@student.sdu.dk

Jens Andersen
jeand17@student.sdu.dk

Supervisor: Ulrik Pagh Schultz

June 1st, 2019

Abstract

Guaranteeing that access policies are enforced can be a challenge, there are many requirements to what can be expressed in policies and how to enforce them. In this thesis, a look at how policies can be expressed and enforced in a medical freezer context is explored. The freezer uses REST-based services and these should be protected by policies.

To guarantee that the policies guarding the exposed REST resources are enforced, the REST architecture is investigated. The result is a domain-specific language that generates and configures a policy framework. The policy framework is called PInt and is also developed as part of this thesis.

The results of creating the DSL and the associated policy framework PInt, is a proof of concept that successfully enforce policies on a RESTful web service. The example is fully generated from the DSL. The example showcases PInts ease of integration in REST-based systems, as well as the simplicity of creating policies with the PInt DSL. This is due to many language features that support the creation of policies and configuration of the generated PInt policiy framework.

Keywords: Domain-Specific language, Policy, Policy enforcement, REST, KAoS, HTTP, Authorization, Access Control, Role-based, Entity-based, Obligation-based, FFU, Intermediary.

Acknowledgments

We would like to express our greatest gratitude and appreciation to our supervisor Ulrik Pagh Schultz for his guidance and counseling throughout the project as well as his valuable feedback on the DSL, Java web services, and the structure of the report. We would like to extend our appreciation to William Appleton Coolidge, the lead developer at the FFU project, for being available whenever we had questions regarding the FFU system.

Contents

1 | Introduction

Laboratories at both University of Southern Denmark(SDU) and Odense University Hospital(OUH) use freezers designed to store blood and tissue cells. These freezers operate at extremely low temperatures (-86C). As a result, opening and closing the freezer regularly negatively affects the stored contents with inconsistent temperatures and freezer burn. It is not uncommon to retake blood samples because the original sample is lost or stored at insufficient temperatures. The design of the freezer is similar to consumer freezers: the entire front opens to allow access to all contents. This raises some security concerns, different researchers have unnecessary access to other researcher's samples, and there is no easy way to regulate or even log which researchers accesses what samples. In addition the laboratories struggle with freezers filling up with unlabelled samples. Administrators have no easy way of knowing if these samples are part of an important ongoing research project, or simply forgotten by the researchers from past projects. As a result the administrators are left with no choice but installing a new freezer next to the old ones.

In an attempt to address these issues, a Future Freezing Unit (FFU) is currently being developed in a research and innovation project coordinated by SDU. The freezer features an airlock in which researchers can place their sample, the freezer then takes care of storing the sample.

It is desired to be able to regulate access to samples stored in the FFU. The FFU developers wants to implement role-based access control (RBAC) on the FFU, this leaves the FFU developers in need of a policy framework that can be implemented on the FFU to enforce these policies. Since the end goal is to sell the FFU as a commercial product, it is desired to be able to configure the system with new policies.
Since the organizational structure can vary a lot from company to company, and the task of specifying the policies can be complicated and time consuming, it is desired to develop an easy way to configure the policy framework to work with different companies.

In this thesis, we analyze the FFU project, and develop a new policy framework. The framework is generated from a Domain-Specific Language (DSL). The company is modelled in the DSL and then the desired policies are written, this makes it easy to adapt the system to fit in different companies with different policies. Three types of policies can be expressed: role-based, entity-based, and obligation-based.

The generated framework is a Java-based web service that mediates all communication between a client and a server. Only authorized requests are forwarded. The service supports the HTTP methods: `GET`, `POST`, `PUT`, and `DELETE`. All information in the body and header as well as cookies and session IDs are passed along between the client and server.

Reading Guide

There are two parts to this thesis: this report, and the developed DSL that generates the PInt framework.

The contents of the report is divided into chapters, sections, and subsections.

In Chapter 2 relevant background information is provided to give the reader the necessary knowledge to read and understand the report. In Chapter 3 the context of the FFU project is described and the challenges that the FFU developers are facing is analyzed. The analysis results in a requirement specification for this thesis. The requirements are found in Section 3.4. In Chapter 4 the developed DSL is presented at an abstract level. Here we explain the core functionality and reasoning behind the decisions made during development of the project. A more technical explanation of the DSL along with code snippets that documents the behavior of the generated Java web service is provided in Chapter 5. In Chapter 6 we evaluate the results, here we compare the developed framework with KAoS. In Chapter 7 we present future work, and discuss features that could further improve the DSL and generated framework. Lastly a final conclusion on the project is made.

All files associated with this project:

- A digital copy of this report
- The defined grammar of the DSL (.xtext)
- All generators of the DSL and custom scoping and validation files (.xtend)
- Example of a generated PInt framework (.java)
- License for usage of the code
- Code documentation

can be found on the GitHub repository at github.com/NielsH12/PInt or by scanning the QR code below.

`https://github.com/NielsH12/PInt`

2 | Background

This chapter will introduce the concepts and methods used in the the project. First an introduction to access control systems in Section 2.1 followed by an overview of computer security models in Section 2.2. Section 2.3 presents the policy framework KAoS and explores some of its details. REST is explained in Section 2.4 in close relation to how HTTP works as well. Finally what a DSL is, and how to develop one is explained in Section 2.5.

2.1 Access Control Systems

An access control system is a system that allows you to manage, monitor, and maintain access to a resource [1]. Resources can be anything such as a physical space in commercial buildings, electronic resources like printers and networks, or digital files in a database. The idea behind access control systems is to provide easy access for authorized people while restricting access for unauthorized people. The set of rules that restrict and grant access is referred to as policies. Access control systems consists of three basic components: user-facing, admin-facing, and infrastructure.

User-facing

The user-facing component consists of the part of the interface that handles the authentication of the user. In the case of commercial buildings this is typically a keycard and a card reader placed next to the door. The keycard stores an ID number of the employee and represents the credentials for the employee. When the keycard is scanned on the card reader, the ID number is read and sent to a server that handles access requests. If the given ID is allowed access to that door, the door is unlocked. In the case of software systems, the user-facing component of an access control system is typically a login screen where the user enters a username and password to authenticate themselves. The user-facing component is not the entire user interface, but encompasses only the part that handles authentication, sometimes two-factor authentication is used (see Section 2.2).

Admin-facing

The admin-facing component is where the administrator of the system sets the parameters and circumstances of who is allowed access to which resources. The admin-facing component can be in the form of a dashboard in which the administrator can view and edit policies, or it can be fully automated. An automated administration requires access to a directory of the employees and automatically updates when a new employee is added. The admin-facing component is not use case specific, which means that the dashboard which the administrator is presented with, can look very similar regardless of whether the resource being governed is a physical building or digital files.

Infrastructure

Access control infrastructure are the physical components that deny or allow access. In the case of employee access to commercial buildings, the infrastructure consists of electronic locks on the doors as well as a central processing unit that communicates with the locks and card readers in the system. This unit would often be located in the server room, as every component has a connection, wired or wireless, to this unit. In the case of software systems, no physical locks are needed to complete the system. The infrastructure is simply the logic in the code that determines what resources to show and hide from the user.

2.1.1 Access Policies

Access policies are the set of rules that restrict and grant access to a resource in an access control system. The task of specifying when people or machines have access to a given resource quickly becomes a cumbersome task. As organizations grow in size and age, the complexity of maintaining access policies grow exponentially therefore different techniques to deal with these issues have been developed. A commonly used technique for specifying access policies is role-based access control (RBAC). RBAC is used by the majority of enterprises with more than 500 employees[2]. RBAC group permissions by roles typically related to a position in an organisation, which allows administrators to quickly modify groups with new permissions expressed in access policies. Another benefit of access control systems is that they provide a separation between the resource and the access rules. This allows administrators to change existing rules or specify new rules without the need to restart the resources being governed.

Even though access control systems have been used for many years, the task of maintaining (adapting to changes) such a system is still a huge task and can rarely be automated. Therefore it is important that system administrators are able to effectively manage the

access policies. Administrators often run into issues of conflicting or duplicated policies, issues that are also typical in other software areas, e.g. handling merge conflicts. Even though there are many merge conflict handling tools, there are still many cases where a human needs to make the final decisions of a conflict.

2.2 Computer Security Models

Computer security models are used to specify and enforce policies in computer systems. Many different schemes exists for configuring such policies. In particular, role-based and capability-based access control are interesting ways of implementing policies. Some different computer security models are described in the following subsections.

2.2.1 Trust boundary

A trust boundary is a term used in computer security systems to describe a portion of a system that is trusted and a portion of a system that is not trusted [3]. For example in a client server model, the client is typically outside the trust boundary, while most, if not all, the components on the server is trusted. this means that the data received from the client is untrusted i.e. outside the trust boundary. It is a good idea to have a clearly defined trust boundary, so it is always known if extra precautions need to be taken when communicating with another component. A potential attacker is presumed to start in the untrusted part of the system.

2.2.2 Authentication

Although authentication and authorization sounds very similar, in practice they are two very different processes.

Authentication is the process of verifying the identity of a user. Typically authentication is achieved by presenting a system with some credentials, these credentials are then cross checked in a database of all registered users. A credential is an attestation that the user is given when created in the system. Examples of credentials include username and passwords, pin codes, ID badges, and keys.

Some systems employ two-factor authentication where the users must be in possession of two sets of credentials in order to be authenticated by the system. The most widely known examples of two-factor authentication is when signing in to online services. Google for example employs an optional two-factor authentication where the user first supplies a valid

username and password, and then either gets an email with an additional password or a notification on their smartphone where they are required to confirm their sign in. Likewise the Danish government has a two-factor authentication system known as NemID where the user first supplies a username and password. Then a "challenge response" procedure is started where the system gives a 4 digit key and expects the correct 6 digit response from the user.

2.2.3 Authorization

Authorization, in contrast, is only done after the user is authenticated. Authorization is the process of verifying that an authenticated user has access permission to a given resource or action.

The task of specifying which users have which permissions is a large subject in the field of computer science. Many strategies exist, some of which are explained in the following sections.

2.2.4 Role-Based Access Control

In role-based access control, a number of roles are created in the system, and policies for each role is specified. Examples could be user, developer, and administrator each with their own set of permissions. One could imagine the administrator to have read/write access to all resources within the system, while the user has more limited access and the developer is somewhere in between.

Role-based access control is a powerful way to express policies because it gives administrators the ability to express policies in the way they view the organization instead of translating it to accommodate an access control mechanism.

Once all roles and accompanying policies have been specified by the administrator, it is very easy to assign employees the correct roles, many companies does this automatically by linking the access control system to a database of their employees.
Once all roles and associated policies are configured, people can be assigned to these roles. A person can possess more than one role at the same time, people can also possess no roles at all. Role-based policies makes it easy for companies to handle employee permissions. All employees in the same department should have the same permissions, thus, simply assigning the same role to them achieves that goal.

2.2.5 Capability-Based Access

Another way of authorizing actions is with the implementation of a capability. A capability is an unforgeable token that represents a set of access permissions. Capabilities are typically used in operating systems as a solution to the confused deputy problem where users could gain elevated permissions due to ambient authority [4].

In practice, systems that use capability-based security, generates the capabilities and pass a value that references the capability to the user. When the user wish to perform an action, the value that references the capability is passed to the server, the server then checks that the given capability grants permission to the desired action. Since the capability itself is never given to the user, it is not possible to add or modify any of the permissions contained in the capability.

In capability-based access systems, once a valid capability is presented to the server, access is given without requiring any further authorization. Capability-based systems also allow users (or user processes in the case of operating systems) to pass a capability between one another to give each other access to a resource.

2.2.6 Access Control Lists

An access control list (ACL) is a list attached to a resource which contains information about which users have access permission to the resource as well as what actions they are authorized to perform. An example of an ACL on a file in a computer system with users Alice, Bob, and Charlie:

```
Alice:  Read, Write; Bob:  Read;
```

In this case, Alice has permission to both read and write to the file, Bob can only read it, and Charlie has no access to the file at all.

2.2.7 Access Control Matrix

An access control matrix is an abstract model of access policies in an access control system. It consists of a matrix that defines a set of allowed actions for every user on every resource in a computer system.

The matrix contains a row for each user, and a column for each resource in the system. A given cell, that represents a user-resource pair, lists all the allowed actions that particular user is allowed to perform on that particular resource. Table 2.1 is an example of such matrix.

	File 1	**File 2**	**File 3**
Alice	Read, Write	Read	
Bob	Read	Read, Write	Read
Charlie		Read	Read, Write

Table 2.1: Access Control Matrix

Access control matrices are merely used to model policies since a literal implementation would consume an excessive amount of memory.

As can be seen, many different strategies exists for implementing access control mechanisms. In particular, role-based access control is interesting for its simplicity. KAoS is one example of an implementation of a policy framework that uses role-based access control.

2.3 Knowledgeable agent-oriented system (KAoS)

KAoS is a policy management framework with focus on specification, management, conflict resolution, and enforcement of policies [5]. The original paper presenting KAoS was published by Seattle University[6] in 1994. The approach was an agent-based policy framework designed with the goal of separating the agent and the domain characteristics. The framework has seen continuous development and numerous articles have been published [5, 7, 8]. Florida Institute for Human & Machine Cognition has published a significant amount of research on their development of the framework [9].

A common challenge when designing a policy management framework is making it flexible enough to handle many different use cases and domains. Many policy management frameworks have been developed with a specific use case in mind like network management[8] or access control[10]. KAoS is designed to fit many different scenarios, like web services, distributed computing platforms, and industrial and military applications[11]. While being flexible enough to be easily adapted to new scenarios as well. This is achieved partly through rich policy semantics which are easily extensible and partly through a layered architecture with a Common Service Interface (CSI).

To better understand what KAoS does, a look into the general operation of KAoS will be useful. On an operational level there are three main components of the KAoS framework, these are the Directory Service, Agents, and Guards [12]. The Directory Service (DS) is the core of the system and holds the current state of the system, e.g. policies, history, and registered agents. Next are the agents which can be robots, web services or applications. It is the agents that policies are enforced upon. To enforce policies guards are implemented

locally as a decision point. The guards contain the policies relevant to its area of implementation. The enforcement of the policy decision is application-specific as the guard only provides a generic way to determine authorization.

The mentioned operational level of KAoS is a small part of the KAoS framework, the framework comes with a large range of additional abilities. To get a grasp of the KAoS framework as a whole we provide an overview of the three-layered architecture.

2.3.1 Three-layered architecture

The basic components of the KAoS framework are the three layers of the architecture, namely: human interface layer, policy management layer, and the policy monitoring and enforcement layer [7, 8] see Figure 2.1. These layers are built on requirements of modularity and extensibility and can all be extended with end user plugins. The first layer is the Human Interface Layer and handles policy creation, managements, etc. This layer is managed with the KAoS Policy Administration Tool (KPAT), which is a graphical user interface based on hypertext. Next is the Policy Management Layer, responsible for monitoring and enforcement. This layer is where the DS is located. The state of the system or the current policy is kept and updated here. Whenever a guard needs to be implemented or updated, they will get their policies from the DS. The final layer is the Monitoring and Enforcement Layer and here the guards can be found.

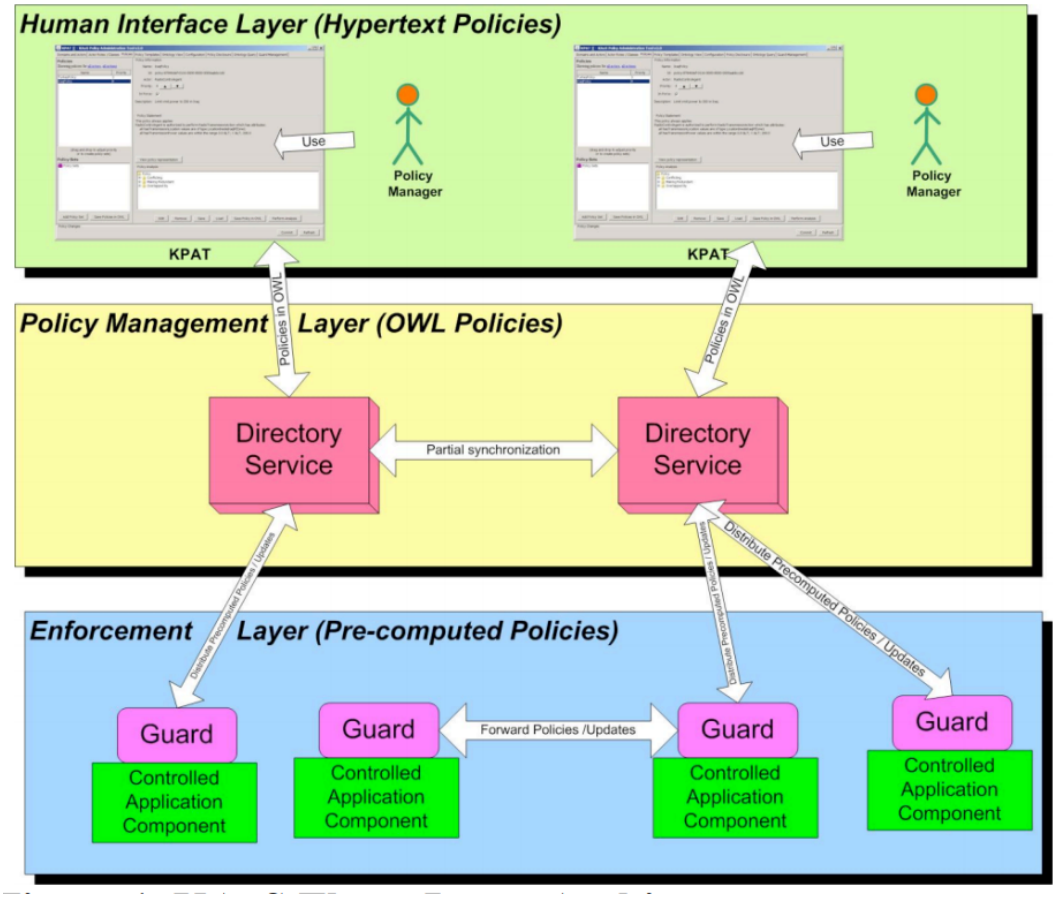

Figure 2.1: The three-layered architecture of KAoS [13]

2.3.2 KAoS policies

In the KAoS user guide, policies are defined as follows: "an enforceable, well specified constraint on the performance of a machine-executable action by a subject in a given situation" [12].

- **Enforceable:** In principle, an action controlled by a policy must be of the sort that it can be prevented, monitored, or enabled by the system infrastructure;
- **Well-specified:** Policies are well-defined declarative descriptions;
- **Constraint on the performance:** The objective of a policy is to ensure, with or without the knowledge or cooperation of the entity being governed, that the policy administrator's intent is carried out with respect to whether or not the specified policy governed action takes place;
- **Subject:** The subject is either a human being, a hardware or software component,

or a group of such entities;

- **Situation:** Policy applicability may be determined by a variety of preconditions and contextual factors" [12].

KAoS distinguishes between two types of policies: authorization and obligation. These two types are the basic building blocks and with them, more sophisticated policies can be constructed. The actions an agent is permitted to do, is specified with either a positive or negative policy, e.g. the agent is authorized to, or not authorized to perform an action. This is what KAoS defines as authorization and are mainly used to prohibit certain actions. Obligations are very similar to authorization policies, the main difference is that they require an action before a trigger condition. In the Figure 2.2 and 2.3, the two policy types are demonstrated. Note: in KAoS, actor and agent refer to the same entity.[12]:

Authorization

Figure 2.2: Authorization policy [12]

Obligation

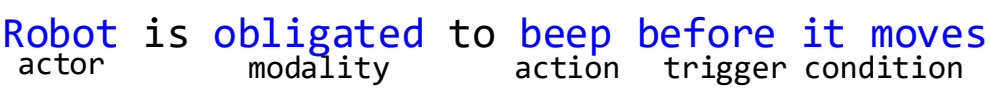

Figure 2.3: Obligation policy [12]

2.3.3 Policy Reasoning

If we look at the simple KAoS policy example from Figure 2.2, we see an authorization policy that allows Bob to send messages. In the case where this general case does not apply, KAoS allows the creation of a new policy that overrules the first policy. To overrule a policy, KAoS uses an integer as a priority level; the higher the integer, the higher priority it has.

It is also possible to add context to a policy, for instance if one wishes to only allow communications with people from the same group (GroupA). The policy would then state that Bob is only authorized to perform a `Communication Action` that has the destination of GroupA (see Figure 2.4).

```
Bob is authorized to send message which has attributes:
    all destination values are in the set (groupA)
```

Figure 2.4: Context on an authorization policy [12]

2.3.4 Creating Actors

In KAoS, policy enforcement is implemented by extending `KAoSActorImpl` and registering the actor with KAoS. Actors can be robots, web services, grid services, or anything on which you would like to apply policies. Registering an actor requires a name, list of domains to register in and a desired transport method. When an actor is registered, a guard is created on the local platform to provide a local policy decision point, which enables policy checking. When this is done, it is possible to connect to the DS and register actions that can be performed. At this point, both the actor and actions should show up in KPAT and it is now possible to create policies.

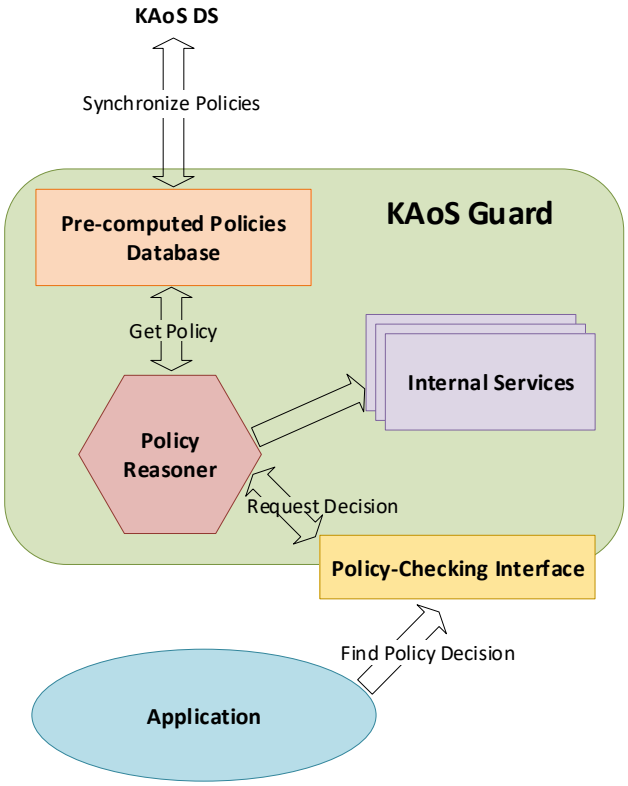

Figure 2.5: Simplified "Conceptual Architecture of the KAoS Guard" [14]

Guards are updated by talking to the DS, but do not need connectivity to operate, see Figure 2.5. Guards have their own policy reasoner logic but it is important to know that, the guard do not enforce policies they only provide a policy decision. It is the responsibility

of the actor to enforce the policy decision. If a guard denies an authorization check it will throw a `KAoSSecurityException`, which must be handled by the actor, hence if no exceptions are thrown it means the authorization was approved.

2.4 REST

REST (Representational State Transfer) is an architectural style for creating interoperability between computer systems and exposing resources over the internet [15].

REST uses Unique Resource Identifiers (URI) to reference resources and communicates over HTTP (see Section 2.4.1 and 2.4.2).

REST defines a set of constraints that a RESTful web service (RWS) must conform to:

Client-Server architecture
The Client-Server architecture is a well known model in the field of data communication. The server exposes a resource or service to one or more clients. The client requests access to the resource. Clients do not expose resources to the server, therefore it is also the client that initiates a communication session.

Statelessness
This is a key constraint on a RESTful web service. It constrains the interaction between a client and a server, such that no context about the client is stored on the server between calls. All information needed to handle a call must be included in the call itself and not implied by the context.

Cacheability
Responses by a server must define whether they are cacheable or not to prevent outdated information on the client due to caching on the client itself or on any intermediaries that may exist between the client and the server.

Layered system
Any number of intermediaries can exist between the client and the server. This is necessary due to the large amount of nodes that make up the internet. Another use of the layered design is load balancing, given the stateless nature of REST, an intermediary acting as a load balancer can split the load between multiple servers. Lastly an intermediary can also enforce security policies.

Uniform Interface
A central feature that distinguishes REST from other network-based APIs is the uniform

interface that sets a global standard for how resources are identified and manipulated.

2.4.1 URI

In REST, resources are identified by their Uniform Resource Identifier (URI). A URI is a string of characters that identifies a specific resource on a specific web service. A URI specifies both the scheme, host name, and path to the resource being manipulated (See Figure 2.6. Universal Resource Locator (URL) is one of the most common types of URI [16].

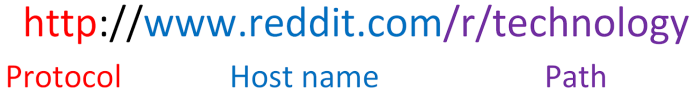

Figure 2.6: URI

2.4.2 HTTP

Hypertext Transfer Protocol (HTTP) is a request-response protocol where a client sends a request to a server, the request contains a request method, protocol version, and a URI identifying the desired resource. The server responds with a status line, protocol version, and a success or error code. The protocol operates on the application layer and assumes the underlying transport layer protocol to be reliable, usually TCP is used, but other protocols that guarantee reliability can be used [17].

One or more intermediaries can be present between the client and the server. There are three common types of intermediaries: proxy, gateway, and tunnel.

Proxy:
A proxy is a forwarding agent that receives requests for URIs and rewrites all or part of the message and forwards the request to the server identified by the URI.

Gateway:
A gateway connects different nodes on a network, but does not alter the messages. A gateway may be present between a client and a server without any of them knowing it.

Tunnel:
A tunnel is used to create a link between two nodes in a restricted network connectivity like a firewall.

HTTP defines a set of methods that the client can invoke, the method is used by the client to indicate the type of action it wishes to perform on the resource. The most common methods used by REST are:

- **GET:** Used to retrieve a representation of a resource, GET should not modify a resource.
- **PUT:** Used to replace a target resource with the contents of the payload.
- **POST:** Used to submit an entity to the target resource.
- **DELETE:** Used to delete the target resource.

Since REST is web-based, and communication between entities typically uses HTTP, there is no need for a specific programming language. In fact, a server and client does not even need to use the same language to be able to communicate.

2.5 Domain-Specific Languages

Martin Fowlers definition of a Domain-Specific Language (DSL): "a computer programming language of limited expressiveness focused on a particular domain." [18, p. 33].

Domain-specific languages are languages designed for a specific target domain. Domain-specific languages allows the user to write high level, highly functional and highly readable programs in very few lines of code, within the domain. This is in great contrast to General-Purpose Languages (GPL) which allows the user to express almost any functionality although at a lower level and with more lines of code. An example of a DSL is HTML which is a markup language designed to design web pages.

A great strength of a DSL is the ability to port to different platforms. Say for example a company specializes in making questionnaires. By using a DSL to write the questionnaire, the company can quickly and efficiently write the questions as desired by the customer and later export it to a webpage, a mobile app, or a printed document. If a change is desired by the customer after the export, it is only required to change the questionnaire in the DSL, and the change will apply to all exports.

DSLs give you a high level language with a large range of functionality with relative few lines of code compared to a GPL. The tradeoff is that the language will only be usefull in the domain it is made for. Therefore the syntax used in the language will be very specialized, which is what enables very high functionality and very readable code.

Potentially enabling people who do not know how to program a GPL, to program the more simple DSL. This can be very valuable because the domain experts are now more directly involved in the development. Even if the domain experts can not create code with the DSL, there is a good chance that they are able to read the DSL code, and thereby engage in the process of verifying whether or not, the software is matching the their needs. The limited expressiveness is why it is called domain-specific, as the language is developed to express behavior within a domain, which also means that it will not be useful outside of the domain. With a clear focus within a domain, the language becomes very useful, making the limited language a worthwhile investment.

What does a DSL look like then? A good example could be the languages used at a Starbucks when ordering coffee, which Martin Fowler uses [18, p. 36]; "Venti, half-caf, nonfat, no-foam, no-whip latte". Reading this outside the context of a Starbucks, it will not make much sense, but within the domain it is very efficient to use. The limited expressiveness of a DSL makes it harder to say wrong things and easier to see when you have made an error. However the mentioned Starbucks example is actually not a DSL by Martin Fowler's definition, as the Starbucks-language is not interpreted by a computer. The language must be implemented as a computer program before it can be called a DSL. But even doing that is not enough as the implementation must also have a language-like flow [18]. The difference of a normal API and a DSL with language-like flow, is illustrated in Figure 2.7 and 2.8.

```
1  Question q1 = new Question("Q1", "Do you have any working
       ↪ experience with robots?", BOOLEAN, null);
2  Require r1 = new Require("Q1", YES);
3  Question q2 = new Question("Q2", "How many years?", INT, r1);
4  Question q3 = new Question("Q3", "Do you have any education
       ↪ relevant for this job?", BOOLEAN);
5  Require r2 = new Require("Q3", YES);
6  Question q4 = new Question("Q4", "Title of education?", STRING
       ↪ , r2);
7
8  Questionnaire q = new Questionnaire
9  q.add(q1);
10 q.add(q2);
11 q.add(q3);
12 q.add(q4);
```

Figure 2.7: Questionnaire API

The code of Figure 2.7 is made with a GPL, created for ordering new custom made questionnaires. Now the same code are restructured with method chaining in Figure 2.8, here a more language-like use of the same GPL is used. This code can be read and interpreted by

a computer. The code in Figure 2.8 have the language-like flow to it as described before.

```
questionnaire()
  .question()
    .name("Q1")
    .text("Do you have any working experience with robots?")
    .answerType(BOOLEAN)
  .question()
    .name("Q2")
    .text("How many years?")
    .answerType(INT)
    .require("Q1", YES)
  .question()
    .name("Q3")
    .text("Do you have any education relevant for this job?")
    .answerType(BOOLEAN)
  .question()
    .name("Q4")
    .text("Title of education?")
    .answerType(STRING)
    .require("Q3", YES)
  .end();
```

Figure 2.8: Internal Questionnaire DSL - method chaining

The code in Figure 2.8 is an internal DSL. Internal DSLs uses a host language but with patterns like method chaining, you can create your own language with a defined syntax and grammar. Making it easier to read, as for instance data types and declarations are obscured. Although internal DSLs have many uses and are quick to implement, they are always constrained by the syntactic structure of the host language. Opposite to internal DSLs are external DSLs, which provides a greater syntactic freedom, enabling the ability to create any syntax that is needed. As an example Figure 2.9 shows an external DSL for creating questionnaires, in the same questionnaire domain as the earlier examples.

```
1 Q1 "Do you have any working experience with robots?" boolean
2 Q2 "How many years?" int require(Q1 = yes)
3 Q3 "Do you have any education relevant for this job?" boolean
4 Q4 "Title of education?" string require(Q3 = yes)
```

Figure 2.9: External Questionnaire DSL

The code in Figure 2.9 shows how to declare the same questions from earlier, in a questionnaire. On the first line the question starts with an ID, then the actual question and in the end an answer type. Line 2 shows an additional feature, the `require` keyword, in this language it means that before question Q2 is shown you must answer "yes" to question Q1.

A common use for DSLs is to generate code based on the DSL script. The mentioned

questionnaire DSL is developed to generate the questionnaire as a web page written with HTML, CSS, and JS. By using the questionnaire DSL one can quickly generate a questionnaire without any knowledge of HTML, CSS, or JS. In the case of a salesman of web-based questionnaires, you can use the language to quickly generate a questionnaire or you could even let your customers write some code themselves.

2.5.1 How to develop a DSL

Using an existing DSL can be easy enough but how do you make your own? The general idea is to make a DSL script, which is then parsed into a metamodel, populating the model. Creating an instance of the metamodel, which then can be used to generate code in a target language. As illustrated in Figure 2.10. Note that the metamodel in this figure is called a semantic model. Metamodel and semantic model means the same and can be used interchangeably, in this report, we use the term metamodel.

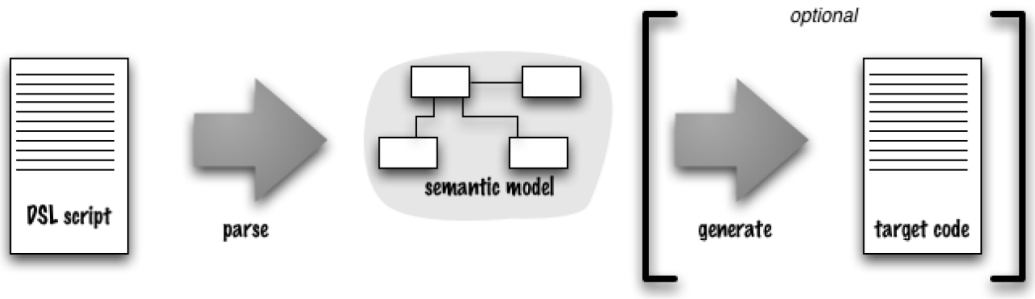

Figure 2.10: The overall architecture of DSL processing [18]

Referencing the example from before in the questionnaire language. The metamodel would be a model of all possible combinations of questions in a questionnaire:

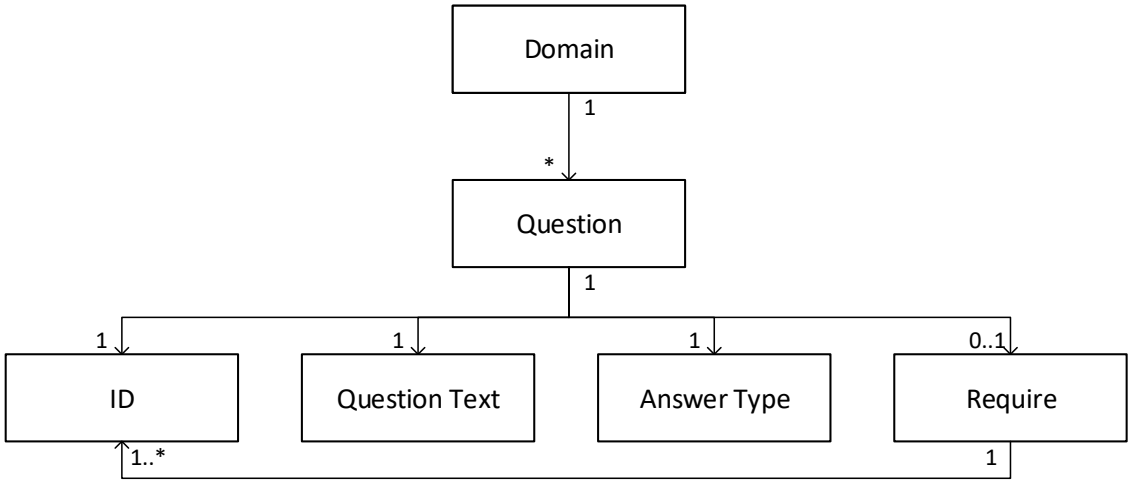

Figure 2.11: Questionnaire metamodel

A domain-specific language, as the name suggest, is a language and most languages have their own syntax and grammar. The grammar defines a set of rules which describe how a stream of text is turned into a syntax tree. The syntax tree is then used by the parser to create an instance of the metamodel. In the case of the questionnaire example, the instance of the metamodel would be a list of question elements, each with properties of ID, Question Text, Answer Type, and Require. In most cases you will then use this model to generate code, using the list of question elements the generator will then iterate the list and generate HTML, CSS, and JS code for each of the questions. E.g. for a boolean answer type it would be a radio button or in the case of string it would just be a text field. It is important to note the difference between the grammar and the semantics, a grammar should dictate that, 3 + 6 is valid, whereas the semantic is does it mean 9 or 36?

Parsing

Parsing is the process of taking a set of symbols and making sure that they conform to the grammar of the language. Before parsing can begin, the characters have to be divided into a sequence of tokens. Tokens can be, among others, keywords (like 'class' in java) or identifiers (like Q1 from the the questionnaire language from earlier), this process is called lexical analysis. When the sequence of tokens is obtained, the parsing can begin. When using the Xtext framework, the parser is generated when the language is created. For the most part it is not necessary to be aware of the workings of the parser. Though there are some cases where this becomes relevant, such as left recursion, associativity or precedence, this will be explained in Section 2.5.2.

DSL Advantages and Disadvantages

Advantages

Generating code is faster than writing it yourself, improving productivity. If you can trust your generator it will be faster and easier to read and write your code. "*The limited expressiveness of DSLs makes it harder to say wrong things and easier to see when you have made an error*" [18, p. 33]. The metamodel is also a great way to create a common language between the engineers and the domain experts when developing a DSL. You can even incorporate terms, frequently used by the domain experts, in the domain language.

Disadvantages

The most obvious disadvantage is the initial cost to develop a domain-specific language, it can also further complicate the cost of maintenance. Since a new language is being developed, this language needs to be learned by the users, which of course is also a cost.

2.5.2 Language Workbenches

Working with DSLs, especially external DSLs can quickly become complicated, as you are creating a new language, a parser for the language, and in most cases also a generator. Changes to one of these parts can involve having to change the other parts as well. To combat this a set of tools can be used, Fowler calls these "Language Workbenches" [19]. Language workbenches are designed to help people create new DSLs, together with high-quality tooling required to use those DSLs effectively. Some of the common characteristics of such language workbenches are:

- **Metamodel Schema:** Helps to define the data structures composing the metamodel.
- **DSL Editing Environment:** Is used to facilitate the programming experience while writing a DSL.
- **Metamodel Behavior:** Determines what the DSL script does by creating the metamodel using code generation in most of the cases.

Xtext

> *"Xtext is a framework for development of programming languages and domain-specific languages. With Xtext you define your language using a powerful grammar language. As a result you get a full infrastructure, including parser, linker, typechecker, compiler as well as editing support for Eclipse, any editor that sup-*

ports the Language Server Protocol and your favorite web browser."

Quote 2.1: Eclipse.org/xtext [20]

In Xtext you define a grammar, see Figure 2.12 for an example. At the very top, the root element is located. The root element: `Questionnaire` contains a variable `questions` that can hold zero to many objects of the type `Question`. The '`*`' character at the end denotes the cardinality rule, additional rules are '`?`' which means optional and '`+`' which means at least once. To only assign a single value, '`=`' is used instead of '`+=`'. In the Question element an `ID` must be written. An `ID` is an inherited rule which only allows a limited set of characters such as (`'a'..'z', '0'..'9'`), similar rules exist for strings, integers, etc. These kind of rules used in Figure 2.12 are called Backus-Naur Form(BNF). BNF is a way of writing grammars and can be imagined as a long list of production rules. There are some extensions to the BNF, one being Extended Backus-Naur Form(EBNF) which basically means that you can use carnality as described earlier. Xtext uses an EBNF which allows you to use the cardinality described.

```
1 Questionnaire:
2     questions+=Question*;
3
4 Question:
5     name=ID text=STRING answerType=Type (Require)?;
```

Figure 2.12: Grammar example

This grammar enables you to write scripts like the one in Figure 2.13. As the cardinality are set to `zero to many` you can write multiple Question elements as long as they are valid Question elements.

```
1 Q5 "Do you have any working experience with customers?"
   ↪ boolean
2 Q6 "What kind of experience do you have" string require(Q5 =
   ↪ yes)
```

Figure 2.13: Language example

A very useful feature in Xtext is referencing, in the questionnaire language there is a reference in `Require` element to the ID of another question. A reference is denoted with the use of square brackets, see Figure 2.14 where the rule of `Require` is shown.

```
Require:
   require '(' ref=[Question] '=' answerTypeResponses=
      ↪ AnswerTypeResponse ')'
```

Figure 2.14: Xtext role for a Require element

Left recursive grammars

The parser that is created when using Xtext, uses a top-down strategy. Top-down parsers can not handle left recursion due to the top-down approach. This means that if for instance, you would like to create a grammar where you can recursively add numbers together, you would end up in an infinite loop. To combat left recursion a transformation called left factoring is used, see Figure 2.15.

```
// Left recursion
Expression:
   Expression '+' Expression |
   Expression '-' Expression |
   INT
;
// Left factored
Expression:
    INT ('+' Expression)? |
    INT ('-' Expression)?
;
```

Figure 2.15: Left factoring

Bottom-up parsers on the other hand do not have the problem of left recursion but instead run into a problem of operator precedence. Operator precedence is the set of rules that determines which rule to execute first when evaluating a mathematical expression. With the Xtext parser the issue of operator precedence is implicitly solved with left factoring. Operator precedence with Xtext can be seen in Figure 2.16.

```
Addition :
    Multiplication ('+' Multiplication)*
;

Multiplication:
    INT ('*' INT)*
;
```

Figure 2.16: Operator precedence

Another issue to pay attention to is associativity, which define rules for what happens when operators have the same precedence. For mathematical expressions we typically wish to

have a left-associative grammar.

A left-associative grammar means that the expression is evaluated from left to right. A simple example can be the mathematical expression `4+3-2`. Addition and subtraction share the same operator precedence, which means the order in which the expression is evaluated depends on the associativity of the parser. If the parser uses top-down approach, the syntax tree would be created as shown in Figure 2.17.

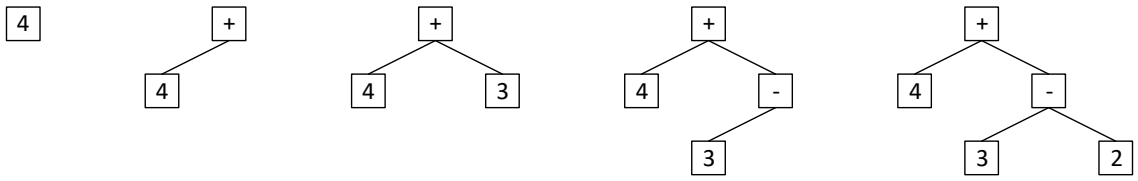

Figure 2.17: Syntax tree of `4+3-2` with *right associative* grammar

To ensure left associativity with a top-down parser. A *tree rewrite action* can be used. As shown in Figure 2.18 the `left=current` takes the current object and stores it in the left branch of the syntax tree and continues. The result is clearly shown in step 4 in Figure 2.19.

```
1 Exp returns Expression:
2     Factor ( ('+' {Add.left=current} | '-' {Sub.left=current})
          ↪  right=Factor)*
3 ;
4
5 Factor returns Expression:
6     Primitive ( ('*' {Mul.left=current} | '/' {Div.left=
          ↪ current}) right=Primitive)*
7 ;
```

Figure 2.18: Operator precedence

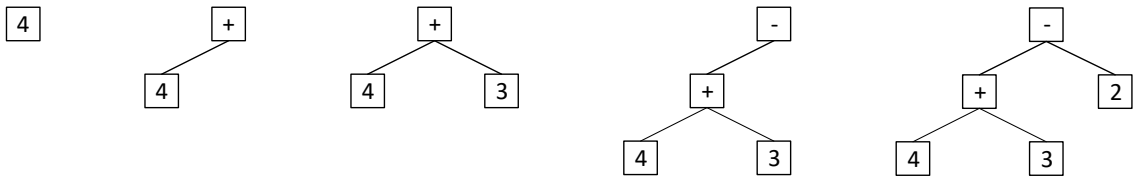

Figure 2.19: Syntax tree of `4+3-2` with *left associative* grammar

3 | Analysis

This chapter analyzes the use of access control mechanisms in the FFU system. First we summarize the current state of cryopreservation freezers at SDU and OUH and motivate the development of a Future Freezing Unit (FFU). An analysis of the FFU will expose some concerns regarding the security of the system. A set of requirements to address these concerns will be derived from the analysis.

3.1 Future Freezing Unit

The Future Freezing Unit is a new type of freezer, designed to be a self-contained unit, that strives to preserve samples better than conventional freezers and enforce access control mechanisms on its stored contents [21].

3.1.1 Motivation

Personal medicine tailored specifically to an individual's genes is a growing trend that, in the future, will be much more common than it is today. Freezing units for cryopreservation of biological cells, blood, and tissue for research and medical use, operate at extremely low temperatures (-86C). In today's freezer, when a researcher wants to retrieve or insert a sample, the entire freezer is opened and all samples are exposed to warm humid air from the outside environment (see Figure 3.1). As a result, opening and closing the freezer regularly, negatively affects the stored contents with inconsistent temperatures and icing. It is not uncommon for researchers to have to retake blood samples because the original sample is lost or stored at insufficient temperatures. Furthermore, storage of personal blood- and tissue samples also raises some security and privacy risks. Researchers currently have needless access to each others samples and could easily retrieve, swap, or modify other samples, either intentional or by mistake. Which could have severe consequences for the patients or the research project. Lastly, the lack of organization within the freezer raises another issue: when researchers forget a sample or leave it inside the freezer after they

are done with their research, the samples stay in there indefinitely. No other researcher knows if the sample is still part of a research project or has been forgotten in there. When the freezer is full, the administrators are left with no choice but to buy a new freezer and install it next to the old one.

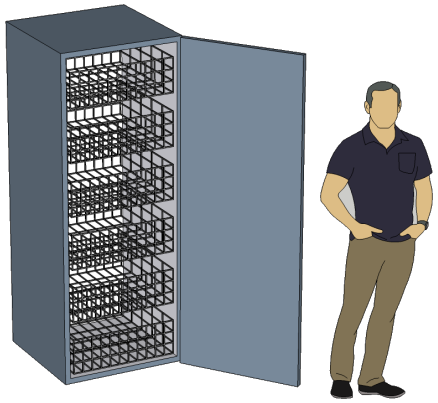

Figure 3.1: Traditional Freezing Unit

For these reasons it is desired to build a new freezing unit that addresses these issues and enables a future with personal medicine: a Future Freezing Unit (FFU).

Large, warehouse-sized biobanks with airlocks and robotic transport systems already exist, but these systems are large, extremely expensive, and very complex. The FFU is designed to be a smaller self contained unit that can fit inside laboratories, small clinics, and other similar facilities with the need of storing biological samples where constructing a fully sized biobank is not feasible.

3.1.2 Design

The concept of the FFU is quite simple: the conventional front door, where the user opens the entire freezer and is granted access to all its contents, is replaced with an airlock. The airlock is serviced by a robotic framework that moves contents to and from the airlock. This way the temperature is kept much more consistent inside the freezer and the amount of moist air from the outside environment that enters the freezer, causing icing, is severely reduced (see Figure 3.2).

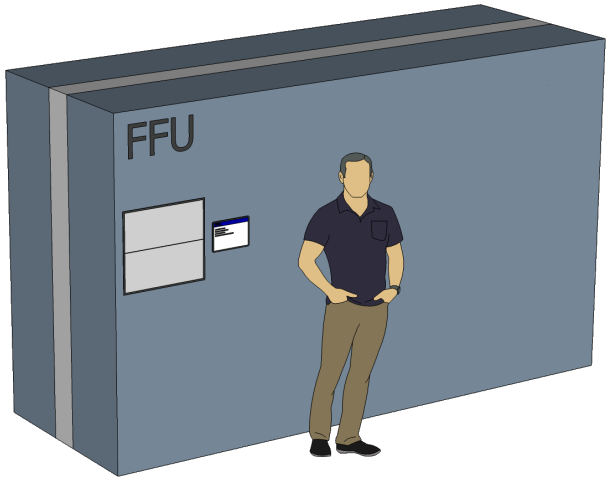

Figure 3.2: Future Freezing Unit

A Human Machine Interface (HMI) is placed somewhere outside the freezer, likely in the form of a tablet, where users can look up the contents of the freezer and request samples to be delivered to the airlock.

Inside the freezer, a computer system runs a database that keeps track of all samples in the freezer. An access control system ensures that only people with proper authorization can access samples and data in the system.

3.1.3 Analysis

Building the FFU itself has shown to introduce many challenges for both Electrical- Mechanical- and Software Engineers. For Electrical- and Mechanical Engineers, the low temperature of -86C is a challenge. Most electrical components and integrated circuits are not designed to work below -50C. Physical properties tend to change at extreme temperatures, even something as simple as copper cables becomes inflexible and brittle due to the brittle-to-ductile transition.

From the perspective of a Software Engineer the challenges relate more to the architecture of the system. The goal is to release the FFU as a commercial product to other corporations and even other industries.

This goal of releasing FFU to other industries means that the system needs to be customized to the needs of other customers than SDU and OUH. It is desired to have an easy way to configure the system and a way for the FFU developers to show the configuration to the customers. Examples of different configuration needs could be featuring different hierarchical structures, different access rules, and maybe even complex access rule conditions,

like time of day, expirations, or defining prerequisites for a given action to be allowed. The current software architecture of FFU is analyzed in Section 3.1.3. While the access control mechanism in FFU is analyzed in Section 3.2.

The FFU needs to be certified for medical use if the system is to be used in the medical sector, e.g. for storing medicine or blood used for transfusion. The process and requirements for certifying a system for medical use is a large and complex topic, and varies depending on where the product is to be released. Medical approval of software is covered in more detail in Section 3.3.

Software

The FFU software consists of many layers and services, the main parts of the system are the user interface, the service layer, the physical layer, the policy framework, and a database. Figure 3.3 illustrates the main parts of the system, a more detailed description of these will be given in this section.

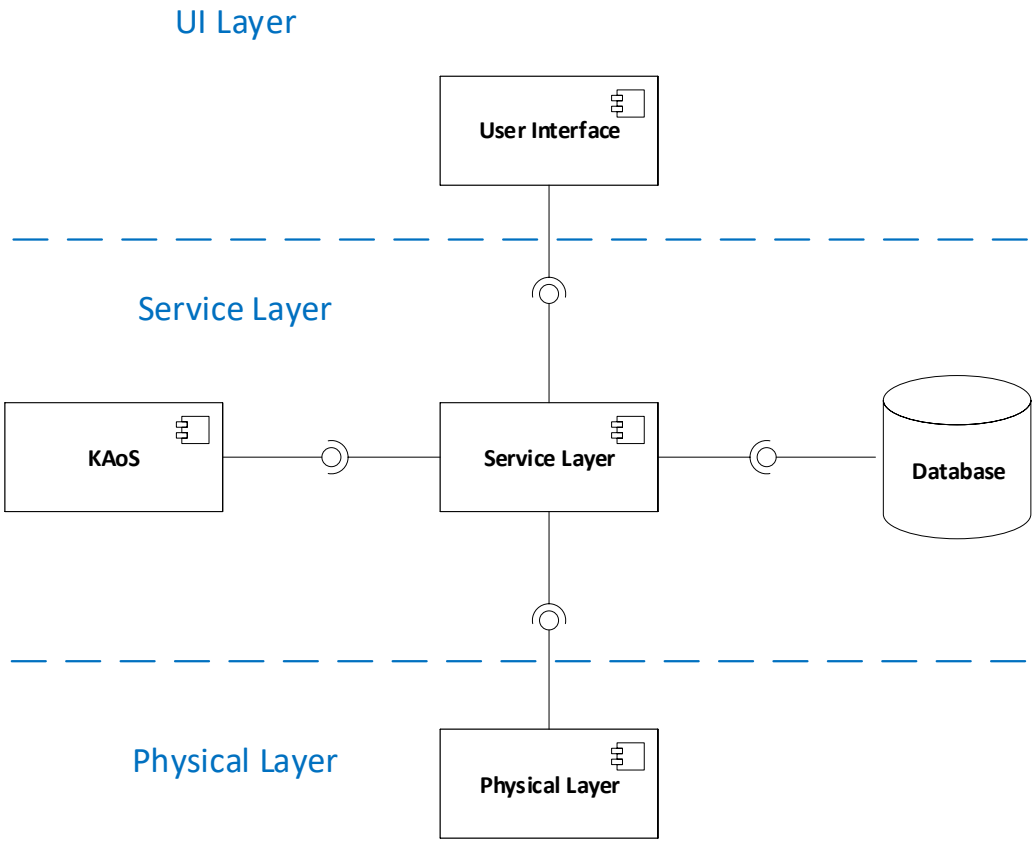

Figure 3.3: FFU component diagram

User Interface

The user interface is the component that enables humans to interact with the freezer, the component is still under development. It is not yet determined which user interface platforms that should be supported. The general idea is to have a mounted tablet on the freezer for daily operations. But for further maintenance a PC with an administrator system will also be provided. How the daily tablet UI and the administrator UI will communicate with the Freezer and its service layer is still to be determined.

Physical Layer

The physical layer operates the mechanical mechanisms of the freezer such as the robotic framework, refrigeration and airlock of the system. It is very low level and sends simple operations such as "`move robot gripper to position x, y`" [22]. The physical layer is not context aware and knows nothing about what is in the freezer or the logical steps for taking out a sample from the freezer. It is the service layer that holds the logic and instructs the physical layer to activate some predefined operations.

Database

The database holds most of the data used in the system such as sample location in the freezer and relevant data about the samples. Samples are stored in sets in the database. There is a distinction between physicalsets and logicalsets [23]. The physical location of a sample inside the freezer is saved in the database as a physicalset. You have to ask the database where physicalsets are located as the physical layer does not hold this information. A logicalset is a set of samples that are not stored in the same location inside the freezer.

KAoS

As described in Section 2.3 KAoS is a large policy management framework developed to support a wide variety of use cases. In the FFU system the main components of KAoS that will be used are the guards, and the developed REST interface for policy configuration on top of the original KAoS service. The components can also be seen in Figure 3.4. Further analysis of the use of KAoS can be found in Section 3.2.

Service Layer

The service layer is by far the most complicated layer in the system. It services all incoming requests from the user interface, including requests to retrieve and insert samples to the freezer. The service layer updates all samples and users in the database, newly submitted samples are also stored. Lastly it translates requests to retrieve a certain box into x, y

commands that the physical layer can understand. The service layer also exposes a REST interface where administrators can query information about samples and registered users. The service layer ties together all the other components, looking at Table 3.1 the sequence of calls involved in creating a new box in the system can be seen. The service layer acts as the central controller in the system. The service layer is responsible for performing authorization checks with KAoS before performing an action. It is also the responsibility of the service layer to catch exceptions thrown by KAoS in the event that a request is not authorized.

Create new box			
Step	**Agent**	**Service**	**Description**
1	user	app	Reads box cover guid and fills out all metadata in HMI. ENTER
2	app	node	POST physicalspecimenset with GUID
3	node	kaos	Authorization check
4	node	kaos	Quota check
5	node	db	Decrement and update allocations
6	node	node	Update physicalspecimenset JSON (extentHistory, @ID), update biostoreextent external
7	node	db	Insert box
8	node	app	Return response as per db version

Table 3.1: Create new box [24]

If we look at step 6 in Table 3.1, there is a call from “node” to “node”, this is because there are multiple services in the service layer. In the Figure 3.4 a more detailed model of the FFU system is shown including some of the services in the service layer.

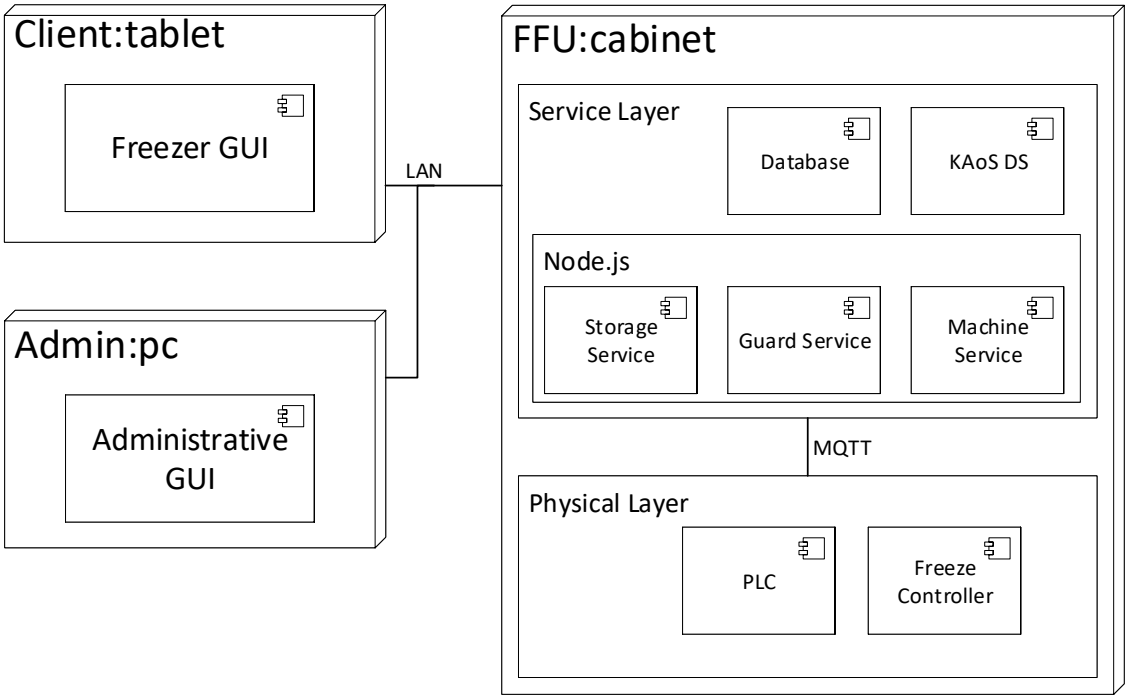

Figure 3.4: FFU Setup

Looking at Figure 3.4 KAoS are present two places: the guard and the KAoS directory service. The guard is the policy decision point, every time an authorization has to be made a guard implementation must be used. The second place KAoS is represented is the directory service, which keeps all created policies. Currently the directory service is managed through KPAT. The FFU developers wanted the ability to manage the policies through a REST interface instead of KPAT, thus IHMC is tasked with implementing a REST interface to manage the policies.

In KAoS, policies are implemented by extending `KAoSActorImpl` and registering the agent with the KAoS directory service. Once registered it is possible to ask permission for different actions. If an action is not authorized, a `KAoSSecurityException` is thrown. It is up to the class extending `KAoSActorImpl` to catch and handle the exception. KAoS itself does not prevent the class from executing the action anyway, it only says whether the action is authorized or not. In the FFU system, the service layer is responsible for extending the `KAoSActorImpl` and obeying the response. This also means that a bug in the service layer would mean that policies might not be upheld.

The FFU system is still subject to change as it is still in development, more services could be added or the communication channels could change. The layered architecture of the FFU system features a very clean separation between the different layers. It is desirable to have

a clean separation between the policy framework and the remaining system components. As seen in Figure 3.3 the KAoS component is also separated from the other components.

3.2 KAoS and Access Policies

The FFU developers are faced with some requirements they need to fulfill when implementing a policy framework. Robust enforcement of policies and ensuring only authorized people have access to the samples stored in the freezer needs to be implemented. The decision to go with a policy-based approach was made by the FFU developers as shown in Quote 3.1.

Many frameworks already exist to configure and enforce policies. The FFU developers has chosen to go with KAoS for its maturity and simplicity as shown in Quote 3.2.

> *"The ability to set specific properties, conditions, and actions on individual storage items constitutes storage automation.*
>
> *...*
>
> *The system concept that is adopted here is to use a policy based design throughout for the realization of properties, conditions, and actions. Policies are thus the format of the mechanism by which the system will realize self-management."*

Quote 3.1: FFU System Concepts [25]

> *"The targeted policy service is the Institute for Human & Machine Cognition's (IHMC) Knowledge Agent-Oriented System (KAoS) Policy Services Framework that is selected due to its maturity, success, simplicity, and features. This framework was originally developed by Boeing and IHMC and has been applied in a variety of areas including military and security applications."*

Quote 3.2: FFU System Concepts [25]

As described in Section 2.3, KAoS is a proprietary framework developed by The Institute for Human & Machine Cognition (IHMC). The framework relies on their own policy administration tool to create what they call policy templates and policy instances. The FFU developers wanted to interact with the framework using REST which was not originally possible. Upon obtaining a license to the software, IHMC also extended the framework to allow interaction with it using REST. Unfortunately, during this thesis, it was discovered that the REST implementation was flawed and needs further development from IHMC.

A significant amount of time was spent trying to work around the issue but due to time limitations we had to exclude the use of KAoS in this thesis. As an alternative, the focus will be to develop a similar framework with similar functionality. This framework needs to be easily configurable, work with REST, and allow the expression of policies in a flexible manner to allow for the customization that FFU requires.

KAoS is a role-based access policy framework (RBAC), and allows users to express policies in almost human-like terms. More information about KAoS' policy-definitions can be found in Section 2.3. KAoS' use of RBAC is compatible with the requirements of the FFU project, the FFU documentation in fact states that the system should: "*secure access with role and user based authorization*" [25].

The goal is to customize the FFU to fit any needs the customer might have. Different customers may have different needs, one could imagine a company that only needs role-based access control, where certain roles have certain permissions. This is relatively straightforward to implement. But another company may need to express rules that relate to a sample rather than a role. One could imagine a rule that dictates that old samples can not be taken out of the freezer as frequently. A detailed user analysis with potential customers to identify needs for different policy types is needed but also outside the scope of this thesis. KAoS is very flexible in the policies that can be expressed. KAoS excels at expressing even complex policies in a way that makes them very readable. As described in Section 2.3, KAoS features two basic types of policies: authorization-based and obligation-based. Each of which can be used as the foundation to construct more sophisticated policies, that also relate to the context and the attributes of an action. An integer serves as the priority for the policy in case there are conflicting policies. Inspired by the two types of basic policies expressible in KAoS (authorization- and obligation-based) the developed policy framework in this thesis is desired to be able to express the same two types of policies.

As mentioned, it is desired to have an easy way to share the configuration with the customers. The customers are domain experts in their field, and can easily validate a configuration if they can understand it. It is not expected that the customers can validate a configuration based on source code. Therefore it is desired to implement a mechanism that is easier to interpret than source code in a general-purpose programming language.

Domain-specific languages are often used within a custom domain, where a configuration needs to change but the domain remains largely the same. DSLs are typically also easier to read, because the domain specificity allows more understandable keywords to be used than in general-purpose languages. Furthermore the domain specificity means that much more

functionality can be described in fewer lines of code, which again increases readability. A DSL is a good approach to implement an easy way to configure the system and domain experts might be able to validate a configuration based on the DSL.

3.3 Certifying a system for medical use

To sell a product as a medical devices you have to obtain a CE-marking for your product. Medical devices are products which are used to diagnose, prevent, relieve or treat a disease, disability, injury, etc., the full definition of a medical device in Denmark can be found in the "Medical Device Order § 1.1" [26].

It is important to know that the product needs to be CE-marked for the use cases that the manufacturer *intents* the product to be used for, and not what the buyers actually use the product for. This means that the FFU developers could market and sell the freezer as a regular freezer for storing food, and then they would not have to worry about obtaining CE-marking for medical use. However, obtaining the CE-marking for medical use, greatly increases the value of the FFU and is desired by the FFU developers to have.

Obtaining a certification for medical devices all depends on what classification your device will have. In the European Union there are four basic classes for medicals devices, Class I, Class IIA, Class IIB and Class III [27], these classes are based on what risk the device will pose when in use.

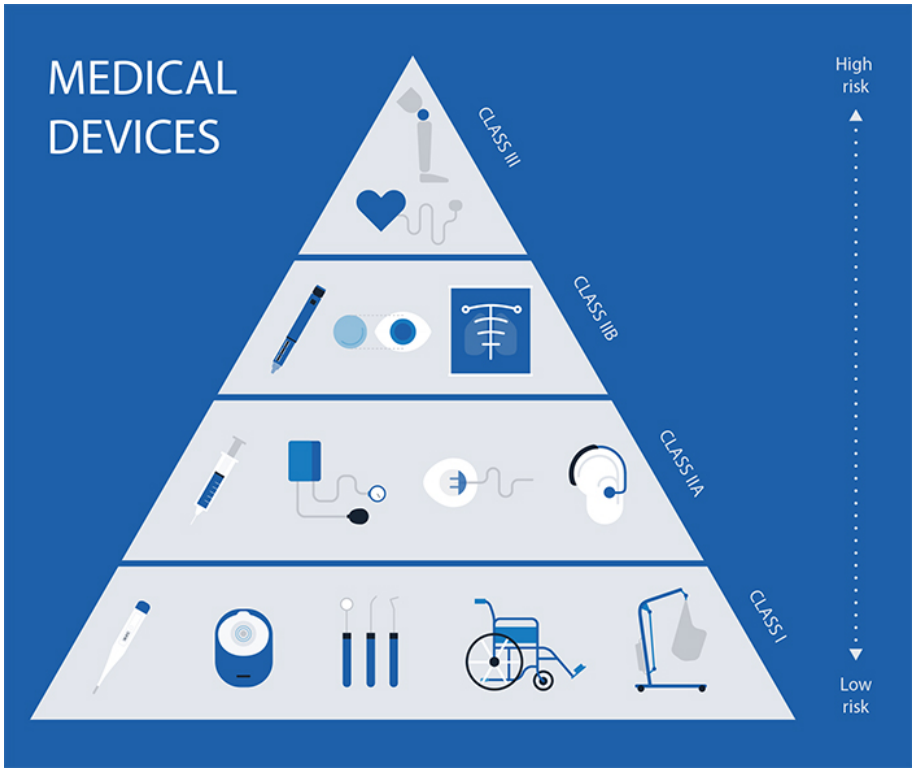

Figure 3.5: Risk classes for medical devices [27]

Class I is associated with the lowest risk and Class III have the highest risk involved, see Figure 3.5. When classifying a device one generally looks at whether or not the device will be used as invasive equipment. Invasive equipment partially or completely penetrates the body either through a body opening or through the body surface. If the equipment is non-invasive it will typically be classified as Class I, e.g. a wheel chair. Invasive equipment are classified based on different criteria such as the duration of contact with the patient, the degree of invasiveness and the part of the body affected by the use of the device [26] appendix IX. In the case of FFU the classification falls under the Class IIa if the stored material is to reenter the body again. If the stored content is not to reenter the body, the freezer will be classified as a Class I device, according to the Danish Medical Device Order see Quote 3.3.

> *"Rule 2:*
> *All non-invasive devices intended for the channeling or storage of blood, body fluids or tissues, liquids or gases for infusion, administration or introduction into the body are in Class IIa:*
>
> – *...*
> – *if it is intended to be used for storing or channeling blood or other body*

fluids or for storing organs, parts of organs or body tissues in all other cases, it is in Class I.

– *...*"

Quote 3.3: Translated from: Danish Medical Device Order [26]

When developing software for use in a medical device, including those classified as Class I and Class IIa, the development process, under the medical directive MDD 93/42 / EEC, can roughly be divided into eight steps. Figure 3.6 outlines the CE-marking process for medical devices. The eight steps are compared to a traditional development process. Additionally, as illustrated, quality management and risk analysis must be conducted and documented throughout the device development.

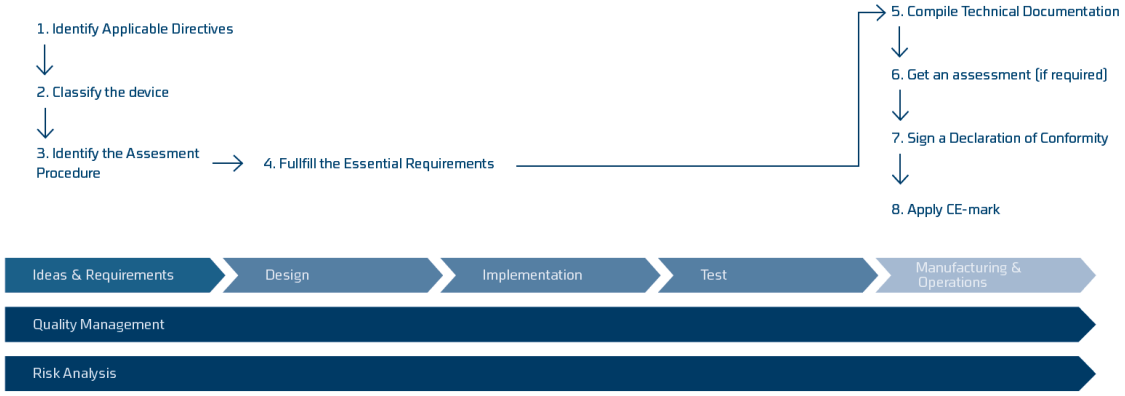

Figure 3.6: CE-marking process for medical devices [28]

Out of these eight steps, there are roughly five key elements of the essential requirements for medical software, when applying for CE-marking [28]:

- Use a quality management system based on ISO 13485.
- Use risk assessment throughout the development of the product and its entire presence on the market, in accordance to ISO 14971, IEC 62304 and IEC 62366-1.
- Implement a lifecycle model containing processes for the software development, error handling and updates, as described in IEC 62304 and IEC 82304-1.
- The product, including software, has to be evaluated in a clinical evaluation, as described in MEDDEV 2.7/1.
- The manufacturer must monitor the product after marketing so that errors and inexpediencies are detected and handled. For applications, procedures for handling errors must be established so that relevant authorities are informed and errors are rectified quickly and efficiently.

The main task in preparing a product, that includes software for CE-marking certification for medical use, is the documentation. Minimizing the amount of code that requires full documentation, including risk analysis and quality management, would greatly reduce the resources needed. Looking into what is needed to comply with all the mentioned standards is beyond the scope of this thesis.

However, if the developed policy framework could guarantee that policies are always enforced, regardless of the state of the higher layers in the system, then the trust boundary would be reduced. We hypothesize that a reduction in the trust boundary would also reduce the amount of code that needs risk analysis and quality management. Which would benefit the FFU developers in obtaining CE-marking for their system.

3.4 Requirements

The analysis of the project has narrowed down the exact task to be completed in this project and revealed a number of requirements. In this section we first summarize the analysis and highlight the findings that translates to the requirements. A complete list of the requirements can be seen in Table 3.2.

Project

This thesis will explore the possibilities of implementing a policy framework in a DSL with similar functionality to KAoS. The framework should enforce policies on protected resources, be accessible using REST, work with different company structure and policy types, and lastly it should be easy to share the configuration with domain experts.

DSL

Creating policies that can be shared and verified by domain experts requires some freedom in how the language is designed. Therefore it is deemed necessary to create the DSL as an external DSL, since an internal DSL will be too constrained by the host language. Using a DSL for code generation will also make it possible to separate the implementation of policies from how they are expressed.

Management

The FFU developers wants the ability to query, instantiate, modify, and remove policies using REST. Thus the framework should feature policy management using REST.

Architecture

The software architecture of FFU features a layered structure, with clean separation between each layer. It is desired to have a clean separation between the access control mechanism and the remaining components in the system. The clean separation should also make it easy to integrate with other systems. If it can be guaranteed that a bug in the upper layers can not result in violating policies, then the amount of code that needs to comply with the requirements for obtaining the necessary CE-marking for medical use, can be greatly reduced.

Trust boundary

It is desired to design the policy framework such that the trust boundary is as small as possible. The reason for this is a hypothesis that states that the smaller the trust boundary is, the easier it will be for the FFU developers to obtain CE-marking for medical use.

KAoS implementation

The FFU developers have chosen to work with KAoS for policy creation and enforcement on FFU. In the implementation of KAoS, the service layer holds the responsibility of complying with the decisions given by the KAoS guards.

Policies

Inspired by KAoS, the framework should be able to express authorization-based and obligation-based policies. The authorization-based policies are separated into two types of authorization policies: role-based and entity-based. A role-based policy authorizes a given role to a given action, while an entity-policy authorizes interaction with a given entity:

- **Role-based policies:** Regular role-based authorization
- **Entity-based policies:** Entity-based authorization
- **Obligation-based policies:** Requires certain actions to be performed before performing the action guarded by this policy

With these three types of policies, it is possible to express a wide range of policies, that fit most, currently defined use cases of the FFU.

ID	Description
1. DSL:	
1a	Readable by domain experts
1b	Implemented as an external DSL
2. Policies:	
2a	Model different hierarchical company structures
2b	Express role-based policies
2c	Express entity-based policies
2d	Express obligation-based policies
3. System integration:	
3a	Easily integratable in new systems
3b	Clear separation between policy enforcement and the rest of the system
3c	The implementation must guarantee policy enforcement
3d	Manage policies in the framework using REST

Table 3.2: Requirements

4 | PInt: Policy-enforcing Intermediary

In this chapter, we present a new role-based access policy framework: PInt. First, in Section 4.1 we give an introduction to the framework. Then in Section 4.2 we explain how policies are enforced on REST calls. Section 4.3 explains the need to include the concept of organizations when implementing PInt. In Section 4.4 we explain the elements in the domain-specific language and how they work together. Section 4.5 explains the different types of policies that can be expressed in the DSL. Lastly in Section 4.6 we discuss if PInt assists the FFU developers in obtaining CE-marking for medical use.

4.1 PInt

PInt is programmed using a domain-specific language that generates a Java-based intermediary that applies policies on HTTP requests. PInt is written with the FFU in mind, but it is designed to be flexible and easy to configure to work in any REST-based system in need of policy-enforcement. Based on the declarative policy specifications, PInt automatically generates a REST-based intermediary service that mediates the interactions between a client and a server.

PInt is designed around the architectural style defined by REST, the main reason for this is the layered architecture that REST enforces, more information about REST in PInt can be found in Section 4.2.

In order to express more complex policies, PInt needs a model of the organization it is implemented in. For this reason PInt defines an interface that it uses to get information about the organization. More information about why the concept of organizations are necessary and how they are used can be found in Section 4.3.

When implementing PInt, all resources exposed by the server are modelled in the DSL,

PInt will then expose exactly the same resources and forward the calls from the client to the server. Any response from the server is sent back to the client, see Figure 4.1.

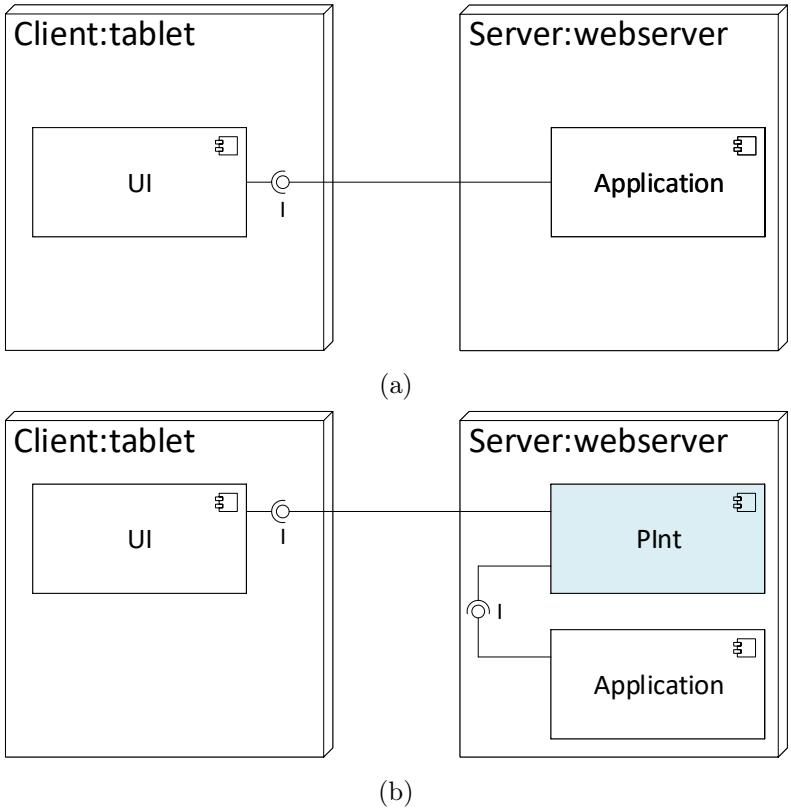

Figure 4.1: Deployment diagram of a web server with and without PInt

The DSL serves three main purposes:

1. Define all resources exposed by the server
2. Model the roles and entities in the system
3. Define policies that should be applied to the resources

More information about the DSL can be found in Section 4.4

The core functionality offered by PInt is the enforcement of policies. Currently three types of policies can be expressed in the DSL: role-based, obligation-based, and entity-based policies. More information about the type of policies and how they are written can be found in Section 4.5.

4.2 Enforcing Policies with REST

In this section, the implementation of enforcement of policies on REST calls between a client and server is explained. REST was chosen to be able to fulfill Requirement 3a and 3b, because the architectural style defined by REST dictates a layered structure.

4.2.1 Trust boundaries

Requirement 3c which dictates that policies must be enforced, requires a correct implementation of KAoS. If bugs are present in the class that extends `KAoSActorImpl`, then the policies enforced by KAoS might not be guaranteed. This means that the code in the class that extends `KAoSActorImpl` needs to be trusted and is thus inside the trust boundary.

Instead of returning whether the action is authorized or not, the policy framework itself could perform the action. This removes the need to trust that the client obeys the authorization. However, since the system, that implements the policy framework, could be written in any general-purpose programming language, it can be a challenge to make the policy framework perform the action instead of the client that made the request.

The language independency and layered architectural style of REST makes it a perfect candidate to base PInt on. With REST, the language of the policy enforcing mechanism is completely independent from the remaining system, while the components maintain their ability to communicate.

REST is also layered by design and any number of intermediaries can exist between two endpoints, this makes it easy to insert a policy enforcing intermediary. The intermediary authorizes the REST call and only forwards the call if it is authorized. By intercepting all communication and only forwarding authorized calls, it is possible to guarantee that no matter what happens at the client, all policies are enforced at all times. This places the client outside trust boundary.

4.2.2 Easy Integration

The layered structure of REST allows PInt to be inserted with almost full transparency, the upper layer will barely notice the existence of PInt. In a best case scenario, the only thing that needs to be changed at the client, is the URI that the calls are sent to. In a worst case scenario, the client needs to be modified to include an ID to identify the user and an ID to identify the entity.

The core of PInt is the web server that receives and processes the REST calls from the

client. The web server is implemented using Apache Tomcat and running servlets that are implemented using JAX-RS. The behavior of the web server is quite simple. Each REST call made to PInt needs to be authorized, and if successful, forwarded to the intended recipient, see Figure 4.2.

```
1 Procedure: Forward Rest Call
2     if(user is authorized) then
3         Forward request
4     else
5         Return Error
6     end
7 end
```

Figure 4.2: PInt web server behavior

Most of the functionality that makes PInt work is neatly packed away in a class called Shield. A detailed explanation of the Shield class as well as how policies are stored and actions are authenticated will be covered in much more detail in Section 5.1.

4.3 Roles and Organizations

The Future Freezing Unit is expected to operate in many different settings, with people of different roles and different permissions. With role-based action control, it is possible to assign roles to large groups of people and give them the same set of permissions. The role a person has, could be based on their position within the company or based on pay grades.

However, this also introduces a problem: a person with a given role, say; researcher, needs permission to retrieve and insert his own samples in the freezer, but does not need access to other researchers samples. It is desired to enable a person to have permission for only their own samples, not all samples in the freezer. Since the REST implementation defines the entire physical layer as a single REST endpoint, and enforces policies on all calls to this resource all the same, there is not an easy way to distinguish between the samples that belong to the researcher and those that does not. In other words, there is no way of defining a subset of samples inside the freezer as accessible to the researcher while restricting access to other samples.

To solve this problem while retaining the model of the freezer, PInt is built around the concept of organizations:

Unlike regular role-based access control, where a person has a globally defined role. In PInt, a person only has a role inside a specific organization. A person can have a role in

multiple organizations, but at most one role in each organization.

Each entity in the system (in case of the FFU this would be a sample) exists inside an organization. Storing different samples in different organizations enables the separation between different researchers and their projects. The organizations are hierarchical, so a person in a parent organization has the same permissions in the child organization. This is useful for instance to create supervisors or administrators who need the same set of permissions in multiple organizations.

Example

To better understand the importance of organizations let us look at a small example:

A small company, that deals with samples, installs an FFU, the company has three types (roles) of employees:

- Researchers
  - These are the primary actors in the system, they interact with the samples in their laboratory and should have full access to their own work.
- Assistants
  - These are secondary actors in the system. Assistants relieve the burden on the researchers by inserting samples back into the freezer when they are done with them.
- Supervisors
  - These are passive actors in the system. Supervisors in this company can only query information about the samples, they cannot retrieve or insert samples in the freezer.

The role-based policies for these three roles would look something like this:

- `Researchers can retrieve, insert, and query samples`
- `Assistants can insert samples`
- `Supervisors can query samples`

Let us say that the company has two research teams each with one researcher and one assistant. Both research teams share a single supervisor.

With regular role-based access control, both researchers would have access to each others samples, which is undesired. This is where organizations come into play.

Figure 4.3 shows how the research teams are grouped into separate organizations. Each

organization holds the samples associated with it, and also holds information about people with certain roles in the organization:

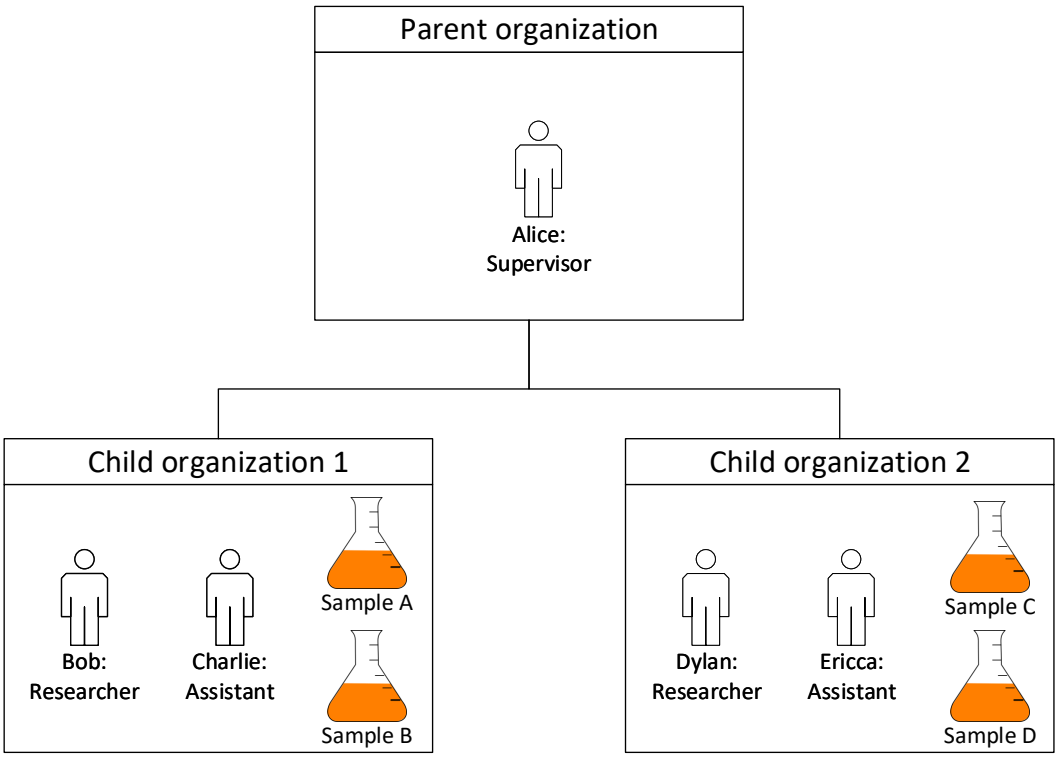

Figure 4.3: Example organization

Alice, has the role of a supervisor in the parent organization, and as a result she is also a supervisor in the two child organizations. Alice has permission to query information about all the stored samples.

Bob, who is a researcher in Child organization 1 can retrieve, insert, and query sample A and B, but has no authority to perform any action on sample C and D and vice versa for Dylan who is a researcher in Child organization 2.

Charlie, who is an assistant can insert sample A and B, but has no authorization to perform any action on sample C and D and vice versa for Ericca who is an assistant in Child organization 2.

When PInt authorizes requests, it needs to know the role of the user in the organization that holds the entity. PInt provides an interface to obtain role information. The interface defines two methods that are used to obtain information about the user and the entity.

```
String getRoleByEntity(String UserID, String EntityID);
String getRoleByOrganization(String UserID, String
    ↪ OrganizationID);
```

Figure 4.4: Interface to get role

These methods, takes the UserID and either the EntityID or the OrganizationID and returns the role the user has in the organization. In the case where the user does not have a role in the organization, the method looks recursively up through any potential parent organizations. If no roles are found then `null` is returned.

Since the best way to infer a person's role in an organization can vary from system to system. It makes sense to generate an interface and manually write this method when implementing PInt in the system.

Let us imagine the company from the previous example has a relational database. A sensible way to store their hierarchical structure could be in a relational database with a table Person holding information about the employees and a table Organization with information about the organizations as shown in Figure 4.5.

`Person` and `Organization` could then be linked with a table `PersonHasRoleInOrganization` which holds information about which employee have what roles, and in which organizations.

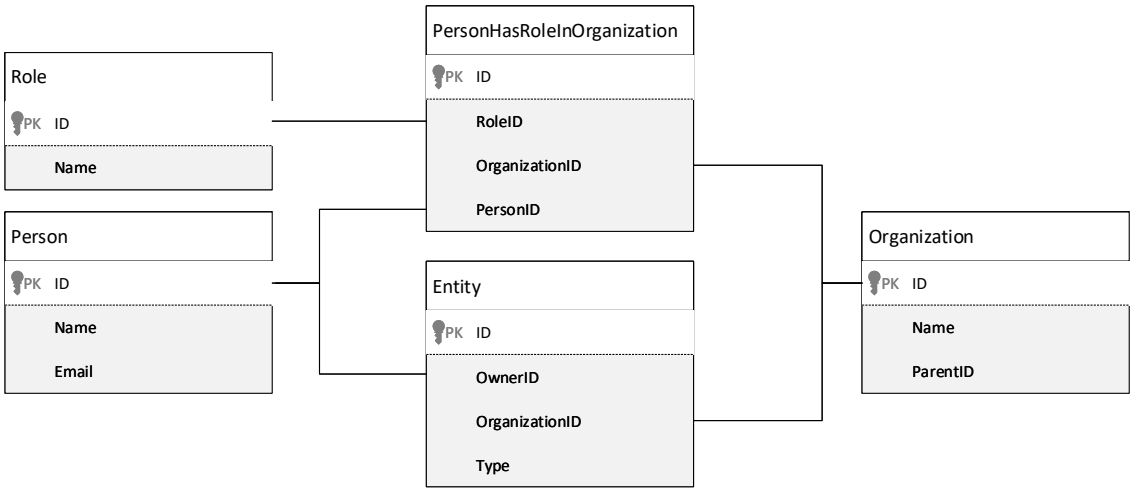

Figure 4.5: Relational database example

If the company kept their employees and roles organized in a database like this, then the interface to find the role given an EntityID and a PersonID could be an SQL statement that looks like Figure 4.6:

```
SELECT role FROM PersonsHasRoleInOrganization
    LEFT JOIN Role AS R
        ON R.ID = PersonsHasRoleInOrganization.RoleID
    LEFT JOIN Organization AS ORG
        ON ORG.ID = PersonsHasRoleInOrganization.
           ↪ OrganizationID
WHERE personID = userID AND organizationID = organizationID
```

Figure 4.6: SQL statement to find role

With this code, it is possible to determine the role a person has in the organization that holds the entity which the action is performed on. Once the action, role, and entity is known, it is possible to perform the policy check that determines whether to execute the action or not.

4.4 Domain-Specific Language

The basic idea behind using DSL, is to separate the language that the domain experts have to read, from the actual implementation of policies. This saves domain experts from having to read or understand the implementation code.

In this section the DSL developed for PInt is presented. An overview of the elements is shown in a metamodel in Section 4.4.2 along with a short description of each element. The design of the language and the decisions made when developing it, as well as an example of the language can be found in Section 4.4.3.

The DSL introduces the concepts of endpoints, roles, and entities. These concepts are explained in Section 4.4.4, 4.4.5, and 4.4.6. The concept of policies are also introduced in the DSL, but this is a larger topic, and is explained in more detail in Section 4.5.

4.4.1 Considerations

As Requirement 1b states, the DSL will be an external DSL because of the language freedom it gives. External DSLs always revolve around a metamodel, creating an instance of this model is done by writing code in the domain-specific language and then populating the metamodel from this code. With a populated metamodel, code generation can be done. In this thesis, Xtext will be used to create the grammar for the policy language. By using Xtext, a parser generator is also provided and is thus not needed to develop. When using Xtext, the metamodel and the grammar are defined simultaneously. Code generation is subsequently done by writing a code generator in Xtend. In summary, Section 4.4 will

explore how the PInt policy language was developed and in Chapter 5 the semantic meaning of the language in the form of a code generator will be explained.

4.4.2 Metamodel

The PInt DSL is designed to basically concern itself with three main areas, (1) model the exposed resources by the web server, (2) model the roles and entities in the company, (3) express the policies that should be enforced.

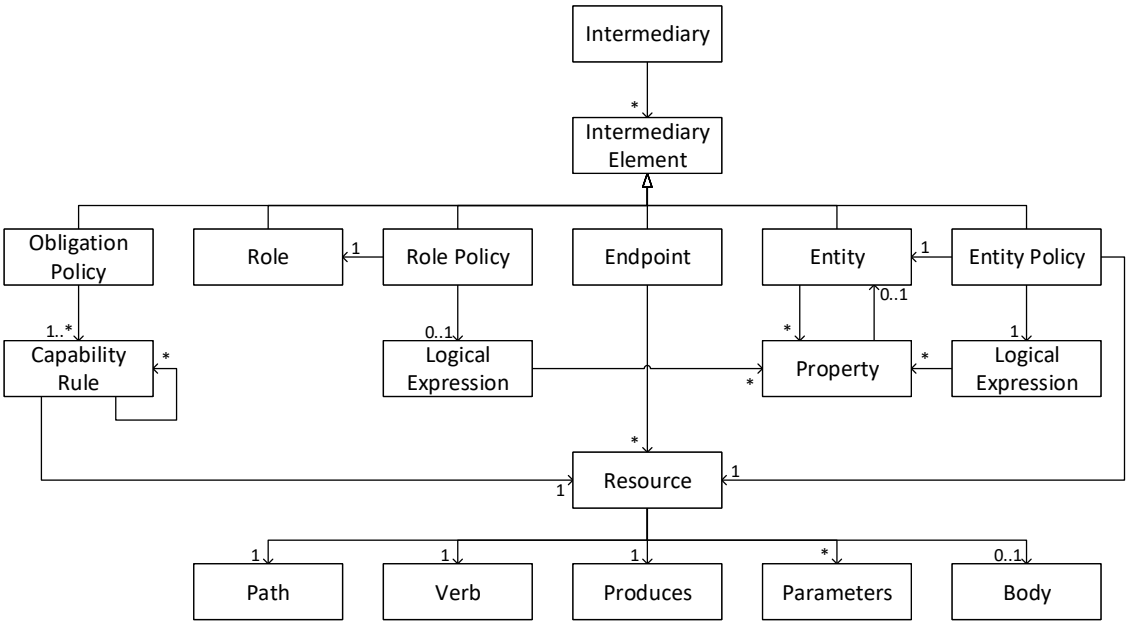

Figure 4.7: PInt Metamodel

Requirement 2b, 2c, and 2d states what policies should be expressible in the DSL. Expressing policies is done with three types of policies; obligation policy, role policy, and entity policy. Policies are used when receiving a HTTP request and are checked before the request is forwarded. The difference between the policies are what information they use to base the authorization on. The authorization process will be explained further in Section 4.5, as well as a more detailed explanation of their difference. For now the three type of policies are used as follows: role policies, as standard role-based access control; obligation policies, as a capability required sequence of HTTP requests; and entity policies, as specific rules on the state of an entity. In the metamodel in Figure 4.7 we see that the root element: `Intermediary` consists of six core elements: `Obligation Policy`, `Role`, `Role Policy`, `Endpoint`, `Entity`, and `Entity Policy`:

Root element

The `Intermediary` is the root element of the language, in this element, an array of `Intermediary Elements` is stored. The `Intermediary Element` stores the six core elements. The reason for storing the core elements in a parent element is to give more flexibility to the user of the language, as this removes the need of defining the elements in a specific order.

Core elements

The `Endpoint` element is used to model the web server. Endpoint elements contains the URL that REST calls should be forwarded to. Each `Endpoint` also contains a number of `Resource` elements, these are used to model the exposed resources of the server. When defining a `Resource` the user must specify the path, HTTP method, and the return type of the call. This information is stored in the `Path`, `Verb`, and `Produces` element. Lastly it is possible to define any query parameters and a JSON object in the HTTP body. Defining this is optional, but if policies rely on this information, it needs to be defined here.

The `Entity` element is used to model the entities that is interacted with in the system. In the case of the FFU the entities could for example be "sample". Each entity has a number of `Property` elements stored in them. These are values that can be used by policies. A property of an element can be a new element, this way it is possible to nest entities. As an example, we could imagine defining a `Person` entity, with properties such as name, address, and email. Next we could define a `Sample` entity with a property owner of type `Person`.

The `Entity Policy` element is used to specify policies that apply based on the property of the entity. For this reason, a `Logical Expression` is required when making an entity policy. Entity policies are defined in the language with the keyword `Require`. The `Logical Expression` is essentially a boolean expression that specifies whether to allow the action on the entity or not. The boolean expression can include properties from the entity on which the policy applies.

The `Role` element is used to define the different roles in the system. These are used to apply role-based policies.

The `Role Policy` element gives a role access to a resources on an endpoint. If no logical expression is defined, then the role always has access to the resource. If a logical expression is defined, then the expression is evaluated and used to decide whether to give access or not. The logical expression under the role policy is very similar to the logical expression under the entity policy. However, in role policies, it is possible to also include values in

the query parameter map and HTTP body of the call as part of the boolean expression.

The `Obligation Policy` element is used to define the obligation-based policies. This is done with the keyword `capabilities` followed by a number of resources represented in a tree-like structure. This explains what actions are needed to perform other actions in the system.

4.4.3 Language Design

As mentioned, the PInt DSL is designed to handle three main areas: model the web server, model the company, and define the policies. Modelling of the web server and company is done by developers or sales people in the FFU project. It is presumed that when the FFU is to be sold, the setup will be done by an FFU representative, based on the customers requirement. Policies on the other hand is supposed to be read by the customers.

Setup design

The grammar design for Endpoints are inspired by the OpenAPI Specification (OAS). OAS is a standard for describing REST APIs made by a consortium of industry experts [29]. As PInt is an intermediary it makes perfect sense to model the REST system in which PInt should be inserted, with a common understandable description. However OAS can express almost all details imaginable within the domain of REST and far more than is needed for PInt. That is why it is only inspired by the OAS at this stage. By creating a PInt specific grammar a lot of noise is removed. Looking at Figure 4.8 a small but complete example, of a PInt DSL instance is illustrated. Endpoints are the first element in the example and should look familiar to OAS. Next is entities and roles, they are designed to look like variables and objects from popular GPL, like Java.

Policy design

Employees of a hospital, such as doctors, should be able to read and verify the policies in PInt. No user analysis have been conducted at this stage of the FFU project, hence no knowledge of existing domain vocabulary is known. Due to the lack of knowledge of the user, the policies are structured similarly as an English sentence. KAoS uses the same approach when creating policies, as previously seen in Section 2.3.2. This of course does not grantee that the users understands PInt policies. Ensuring language readability for domain experts, is a significant task and deemed out of scope of this thesis. However with the right tools and an external DSL, making changes to the language to accommodate requirements of a later user analysis, will be fairly cheap and easy to do. Fulfilling Requirement 1a

is therefore only partially attempted in this thesis. Looking at Figure 4.8 policies can be found in the last part of the figure. Obligation-based policies are modelled as a tree structure, representing sequences of allowed resources. The role- and entity-based policies are designed as English sentences, see for instance Line 43 in Figure 4.8.

```
endpoint Freezer{
    url:"http://ffu.freezer.com"
    resource insert {
        path: "insert"
        verb: PUT
        produces: plain
        body: {
            bloodtype: string
        }
    }
    resource retrieve {
        path: "retrieve"
        verb: GET
        produces: json
        parameters: "xPos" int "yPos" int
    }
    resource querySample {
        path: "querysample"
        verb: GET
        produces: json
    }
}

entity Sample {
    identifier string sampleID
    string owner
    date accessed
}

role Supervisor
role Assistant
role Researcher

// Obligation-based policies
    capabilities {
        Freezer.querySample {
            Freezer.retrieve
        }
    }

//Role-based policies
    // Researcher policies
    rolepolicy : Researcher can access Freezer.querySample
    rolepolicy : Researcher can access Freezer.retrieve
    rolepolicy : Researcher can access Freezer.insert

    // Assistant policies
    rolepolicy : Assistant can access Freezer.querySample
    rolepolicy : Assistant can access Freezer.insert  if (
       ↪ StringCompare(body.bloodtype, "AB+"))

    // Supervisor policies
    rolepolicy : Supervisor can access Freezer.querySample

// Entity-based policies
    require : Sample ( DaysBetween(accessed, today) > 2) for
       ↪ Freezer.retrieve
```

Figure 4.8: Full example of policy DSL

4.4.4 Endpoints

PInt can have many endpoints, to whom HTTP requests are forwarded. When modelling the endpoints, you must define a name, URL, and resources. The URL defines where the request will be forwarded to. Resources are the different resources exposed at the Endpoint, each of these resources can then have different policies.

```
endpoint Freezer{
    url:"http://ffu.freezer.com"
    resource insert {
        path: "insert"
        verb: PUT
        produces: plain
    }
    resource retrieve {
        path: "retrieve"
        verb: GET
        produces: json
        parameters: "xPos" int "yPos" int
    }
    ...
}
```

Figure 4.9: Example of Endpoints

Resources must have a name, verb, path, and produces. Parameters and body are optional. In Figure 4.9 an endpoint named Freezer is created, the URL to the endpoint is defined and two resources are defined: insert and retrieve. The insert resources have a path "insert" (full path: http://ffu.freezer.com/insert). The verb is the HTTP method and defines what method should be used, in Figure 4.9 the resource "insert" expects the PUT method. Produces defines the expected response media type, in this case it is JSON, other examples could be text or plain. In Figure 4.10 another example of an endpoint is illustrating the use of a body element. Using a body enables policies to include values from the HTTP body in their logical expression.

```
endpoint SampleDatabase{
    url:"http://ffu.database.com"
    resource createblood {
        path: "blood"
        verb: POST
        produces: plain
        body: {
            Sample{
                serialnumber: int
                owner: string
                accessed: string
            }
            bloodtype: string
        }
    }
}
```

Figure 4.10: Example of Endpoints with body

4.4.5 Roles

Roles are simply defined with the keyword role, followed by a name, see Figure 4.11. They are afterwards used to express which roles have access to which resources. However the PInt intermediary can not trust an incoming request to provide the role, and therefore must obtain this on its own, by a trusted source. How this is done will be demonstrated later.

```
role Researcher
role Assistant
role Supervisor
```

Figure 4.11: Example of roles

4.4.6 Entities

When modeling entities, a name and an identifier variable must be provided. The identifier is needed by the framework to know how to get this entity when used in an entity policy, more details on this in Section 4.5.2. Each entity can then have one to many properties, of the types; int, string, float, boolean, and date. In Figure 4.12 an entity by the name `Sample` is created with an identifier `sampleID` of type string. Looking at `Sample` there are two properties besides the identifier, one of the type string and one of type date. Each property must also have a unique name. Properties can also be a reference to another entity, enabling some code reuse. In Figure 4.12 the entities `Urine` and `Blood` have a property of the type `Sample` which refers to the entity `Sample`. This means that `Urine` and `Blood` also contains the data in `Sample`.

```
entity Sample {
    identifier string sampleID
    string owner
    date accessed
}

entity Urine {
    identifier string urineID
    Sample sampledata
    int Protein
    int Glucose
    float pH
}

entity Blood {
    identifier string bloodID
    Sample sampledata
    int Redbloodcells
    int Whitebloodcells
    int Bacteria
}
```

Figure 4.12: Entity example, extended with Urine and Blood entities

4.5 Policies

The core functionality of PInt is the creation and enforcement of policies. Requirement 2b, 2c, and 2d defines three types of policies (role-based, entity-based, and obligation-based) that must be expressible in the DSL. How each of the policies are written in the DSL is explained in the following sections.

4.5.1 Role-based policy

Role-based policies are the most fundamental policy in PInt. All calls to PInt are first verified with role policies, if no authorizing role-policy exists, then the request is denied. If an authorizing policy exists, then PInt continues to verify entity and obligation policies.

As mentioned in Section 4.3, PInt also introduces the concept of organizations in order to achieve separation between users with the same role, but working on different projects.

In practice, a role policy simply links a role (like Researcher) with a resource at an endpoint (like Freezer.retrieve). This link authorizes people with the role `Researcher` to access the resource `retrieve`. It is also possible to make a resource accessible to everyone with the keyword `everyone`. If a policy uses this keyword, calls that request this action is not verified with role-based policies.

In the DSL, a role-based policy is created with the keyword `rolepolicy` followed by the

policy. An example a role-based policy can be seen in Figure 4.13.

```
rolepolicy: Researcher can access Freezer.retrieve
```

Figure 4.13: Role policy example

When defining the endpoint in the DSL, it is possible to define arguments that the query parameter map contains. It is possible to enforce role-based policies that require values in the query parameter map to meet certain conditions. As an example, let's say the Freezer endpoint is defined as in Figure 4.14.

```
endpoint Freezer{
    url:"http://ffu.freezer.com"

    resource retrieve {
        path: "retrieve"
        verb: GET
        produces: json
        parameters: "xPos" int "yPos" int
    }
}
```

Figure 4.14: Freezer endpoint

A role-based policy can now require certain values of `xPos`. This is done by adding the keyword `if` at the end of the policy followed by a logical expression. The logical expression is very similar to boolean expressions in most GPL. The policy in Figure 4.15 allows Researchers to call retrieve, if and only if, the parameter `xPos` is equal to 2.

```
rolepolicy: Researcher can access Freezer.retrieve if(xPos ==
    ↪ 2)
```

Figure 4.15: Role policy example

In addition to the query parameter map, it is also possible to define values in the body of the HTTP request. Just like it is possible to require certain conditions to be met from values in the query parameter map, it is also possible to require values from the body to meet certain conditions.

4.5.2 Entity-based policy

In the case of a freezer like the one developed in the FFU project, one could imagine that a sample in the freezer might have some restrictions on how often they can be retrieved

from the freezer. To handle this, entity-based policies were created. Entity-based policies put restrictions on the access to resources by looking at the state of the entity. The state of the entity must be provided through the interface generated by PInt. This enables PInt to request the state of an entity and checking its properties and compare to the relevant policy. Looking at Figure 4.16 an entity policy starts with the keyword `require` followed by the entity that the policy applies to. In the parentheses a boolean expression is defined. In this case there is a date comparison between the date accessed and the current date, the difference in days have to be greater than 2. At the end, the resource that is protected by this policy is written. In this case, the entity policy dictates that samples need to be in the freezer for more than two days before they can be retrieved again.

```
require : Sample ( DaysBetween(accessed, today) > 2) for
    ↪ Freezer.retrieve
```

Figure 4.16: Example of an entity policy

4.5.3 Obligation-based policy

Obligation-based policies requires the user to make certain HTTP requests before being allowed to request the resource protected by the obligation policy. This requires PInt to remember what calls have previously been made by which user, or give the user a way to prove that they have indeed made the required call to be allowed to make their next intended call. In operating systems, a capability is sometimes used by processes to prove their authorization to specific resources. This approach is suitable to solve this problem and has been implemented in the DSL.

```
capabilities {
    SampleDatabase.get{
        Freezer.retrieve{
            SampleDatabase.retrieve
        }
    }

    SampleDatabase.findEmptySlot{
        Freezer.insert{
            SampleDatabase.insert
        }
        Freezer.move
    }
}
```

Figure 4.17: Extended example of a Obligation policy

In Figure 4.17 two obligation policies are created. All obligation policies are contained in

the `capabilities` keyword. The first policy specifies that `SampleDatabase.get` gives a capability that authorizes the user to call `Freezer.retrieve`. Nested under `Freezer.retrieve`, is a new obligation policy that specifies that this call returns a capability that authorizes the user to call `SampleDatabase.retrieve`.

In summary, the first obligation policy defines a sequence of calls that needs to be called in the order as shown in Figure 4.18.

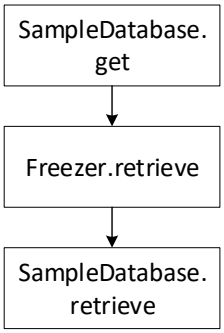

Figure 4.18: Obligation-based policy

The second obligation policy works in a similar way. There is one exception though, the first call returns a capability that can authorize two different calls. Only one of the calls can be made with the capability. As a result, instead of a sequence of calls, the policy is more similar to a tree of valid calls as seen in Figure 4.19

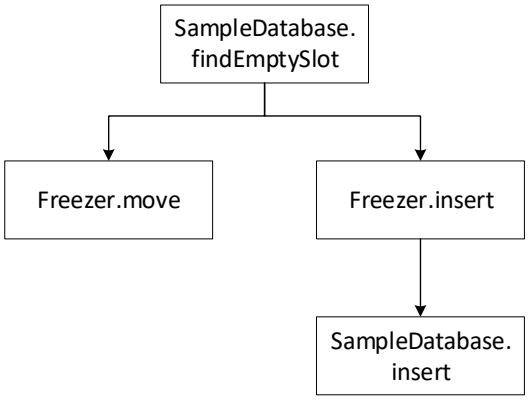

Figure 4.19: Obligation-based policy

4.6 PInt in the FFU system

The requirements for obtaining medical certification does not require specific coding standards or similar constraints on the structure of the code. Instead, the focus is on quality management, risk assessment, and life-cycle modelling.

These documentations are required of the FFU developers to use when writing the software that runs on the system. Thus, simplifying the process of incorporating a policy framework in the system will assist the developers in completing the project and obtaining the required certification. Furthermore if PInt can reduce the trust boundary to exclude a portion of the code base, then completing the risk assessment would also be simplified for the FFU developers.

PInt is a stand alone framework that can be used in any system in many scenarios. Hence PInt is not an integrated part of the FFU system, the documentation requirements does not apply when developing the framework itself. Instead, the requirements only apply when implementing PInt in the FFU system.

In an effort to simplify the process of obtaining a CE-marking for the FFU developers it was desired to reduce the trust boundary of the code base as much as possible. An effective way to accomplish this is to insert the policy enforcement between two layers in the system and denying all unauthorized communication between them. This completely eliminates the need to trust the code above the policy enforcement. The only requirement is that authentication is reliable and that the policy enforcement itself is trusted.

5 | Implementation

In this chapter, we explain the technical aspects of the implementation in more detail. The chapter is twofold. First we explain the generated code: a Java-based web server that receives HTTP requests, applies policies, and forwards them if they are authorized. Secondly we explain the domain-specific language: the grammar, defined in Xtext and the associated generator, written in Xtend.

5.1 The PInt Web Service

The PInt web service is the generated Java-based intermediary. All the endpoints are defined in the PInt class. The Shield class is responsible for authorizing all calls, and the ForwardClient forwards the calls to the intended recipient if the Shield class authorizes the call.

The class diagram shown in Figure 5.1 describes the PInt web service. The notation «X» is used to show methods and attributes that are generated for multiple values of X depending on what is written in the DSL.

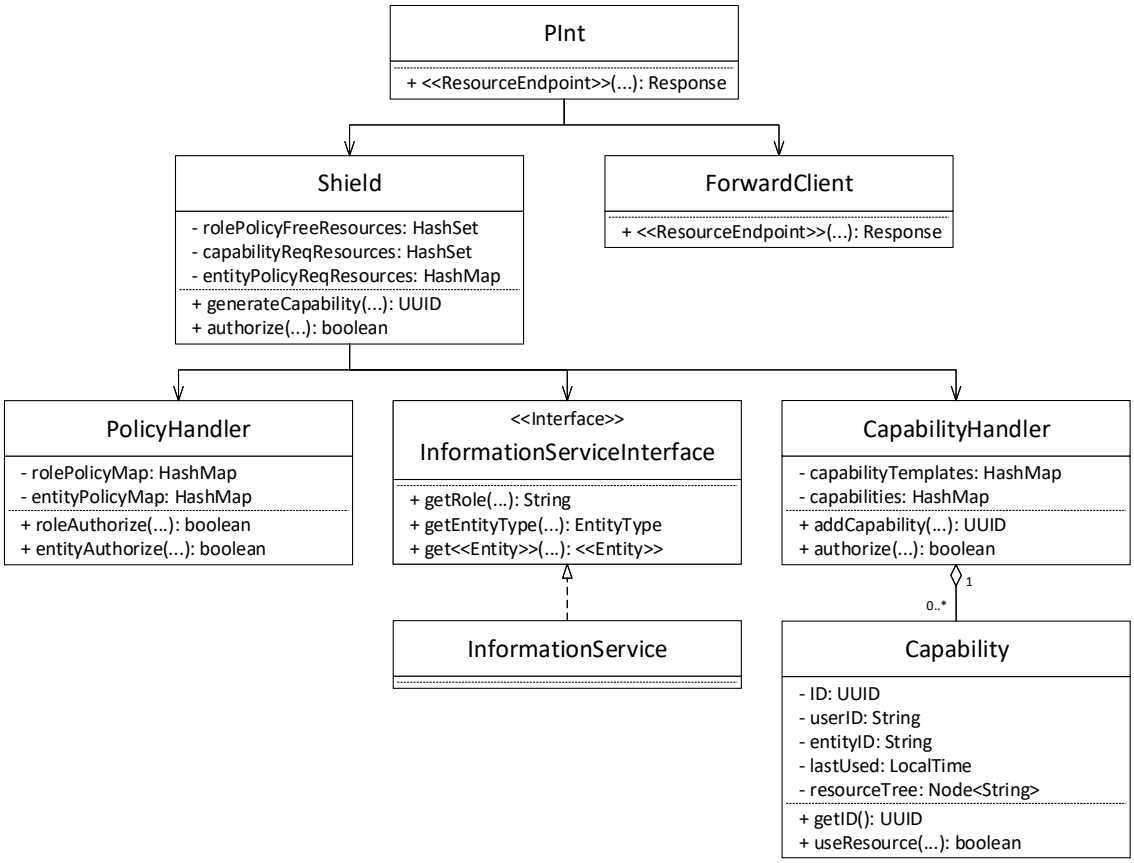

Figure 5.1: Class Diagram: The notation «X» denotes a method or attribute generated for multiple values of X

Figure 5.2 shows a simple sequence diagram showing the overall behavior when handling a HTTP request from a client and its path to the server and back to the client again. The authorization steps are handled by the class Shield and its subclasses.

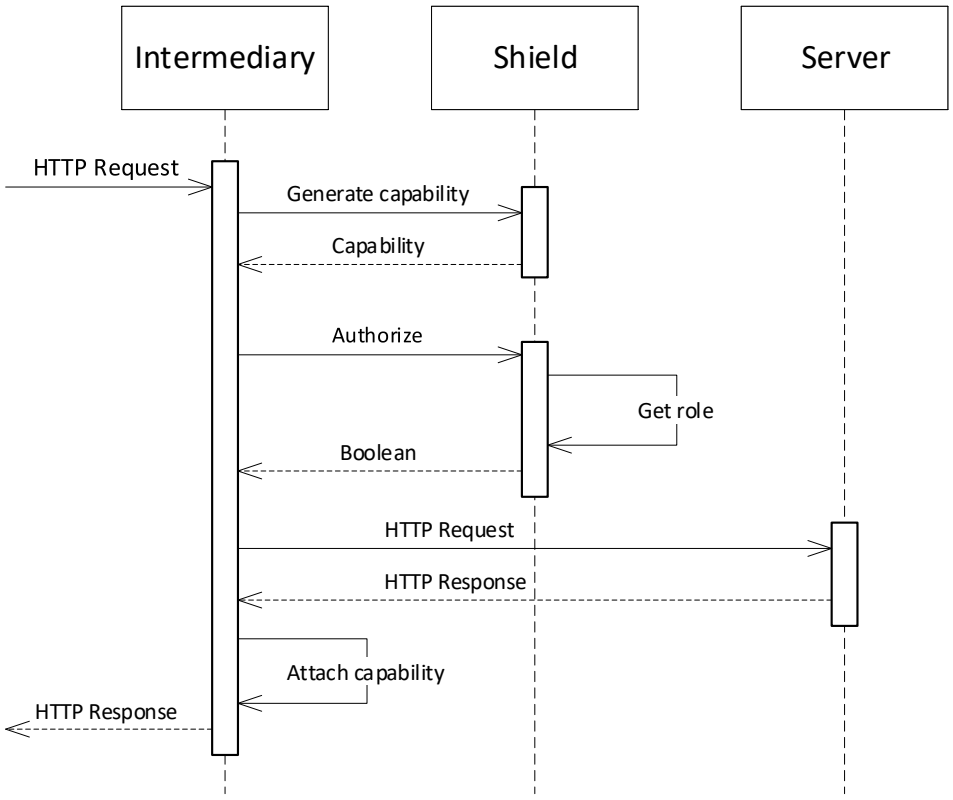

Figure 5.2: Sequence diagram of the overall behavior of PInt

5.1.1 PInt

The PInt class is the main class of the system. It runs the web server, which is implemented using the JAX-WS framework and Apache Tomcat.

The class exposes the resources defined in the DSL. Figure 5.3 is an example of the creation of a resource. If the resource returns a capability, the capability is created (see Line 10 in Figure 5.3) and the reference to the capability is added to the response header. The request is authorized using the Shield class. If the Shield class authorizes the request, the request is forwarded and the response is returned as shown in Line 15. If the request is not authorized, the Shield class returns a proper error to the user. If the Shield class encounters an unexpected error, an INTERNAL_SERVER_ERROR is thrown instead as shown in Line 18.

```
@Path("querysample/{EntityID}")
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response querySampleFreezer (@Context UriInfo info,
    ↪ String body, @Context HttpHeaders headers, @PathParam("
    ↪ EntityID") String entityID){

    if(!entityID.equals(String.valueOf(info.getQueryParameters
        ↪ ().getFirst("EntityID")))){
        throw new WebApplicationException(Response.Status.
            ↪ BAD_REQUEST);
    }

    UUID CapabilityID = shield.generateCapability("Freezer/
        ↪ querySample", info.getQueryParameters(), body, false)
        ↪ ;

    // Check policies
    if (shield.authorize("Freezer/querySample", info.
        ↪ getQueryParameters(), body)){
        // Forward request
        return Response.fromResponse(IntermediaryClient.
            ↪ FreezerGETquerySample(info, body, headers)).
            ↪ header("Capability", CapabilityID).build();
    };

    throw new WebApplicationException(Response.Status.
        ↪ INTERNAL_SERVER_ERROR);

}
```

Figure 5.3: PInt: Resource

5.1.2 ForwardClient

The ForwardClient class builds the HTTP request and forwards it to the intended web server, afterwards the reply from the web server is returned to the user. A method is generated for each endpoint-resource pair. An example of such method can be seen in Figure 5.4.

```
public static Response FreezerPOSTauthorize(UriInfo info,
    ↪ String body, HttpHeaders headers){

     Invocation.Builder builder = client
          .target(REST_URI_Freezer)
          .path("biostore/authenticate/login")
          .request()
          .header("Content-Type", "application/json");

          for (Map.Entry<String, Cookie> entry : headers.
             ↪ getCookies().entrySet()){
              builder.cookie(entry.getValue());
          }

          return builder.post(Entity.json(body));
}
```

Figure 5.4: PInt: ForwardClient

5.1.3 Shield

The Shield class manages all the functionality needed to enforce policies. However, the Shield class does not itself contain the functionality to make policy decisions, but instead relies on member classes to do this. The PolicyHandler evaluates role-based and entity-based policies while CapabilityHandler evaluates obligation-based policies. The Shield class holds information about which resources are guarded by policies, and which resources are publicly available.

The Shield class has two HashSets: The first HashSet stores all the resources that are publicly accessible. A resource is made public with the keyword `everyone` in the DSL. The second HashSet stores all the resources that require a capability to be accessed.

In addition to the two HashSets, the Shield class also has a HashMap. The HashMap links an entity with a HashSet of resources. This is used to check whether the requested resource is protected by an entity-based policy.

Since the Shield class is responsible for all authorization, the class has a method `generate Cabability` used by the PInt class to create cababilities. The Shield class provides an authorize method that authorizes a request, the method checks the HashSets and HashMap for the types of policies that should be checked, and makes the necessary checks, see Figure 5.5

```
public boolean authorize(String resource, MultivaluedMap<
    ↪ String, String> QPmap, String body){

    String UserID = QPmap.getFirst("UserID");
    String EntityID = QPmap.getFirst("EntityID");
    String role = getRoleByEntity(UserID, EntityID);

    // Check role policy
    if(!rolePolicyFreeResources.contains(resource)){
        if(!policyHandler.roleAuthorize(role, resource, QPmap,
            ↪  body)){
            throw new WebApplicationException("RP: Permission
                ↪ denied", Response.Status.FORBIDDEN);
        }
    }

    // Check entity policy
    String entityType = informationService.getEntityType(QPmap
        ↪ );
    if(entityPolicyRequiringResources.get(entityType).contains
        ↪ (resource)){
        if(!policyHandler.entityAuthorize(entityType, resource
            ↪ , QPmap)){
            throw new WebApplicationException("EP: Permission
                ↪ denied", Response.Status.FORBIDDEN);
        }
    }

    // Check capability
    if(capabilityRequiringResources.contains(resource)){
        String tempCapID = QPmap.getFirst("CapabilityID");
        if (UserID == null || tempCapID == null){
            throw new WebApplicationException("OP: Missing
                ↪ input", Response.Status.BAD_REQUEST);
        }

        UUID CapabilityID = UUID.fromString(tempCapID);
        if(!capabilityHandler.authorize(UserID, CapabilityID,
            ↪ resource)){
            throw new WebApplicationException("OP: Invalid
                ↪ capability", Response.Status.FORBIDDEN);
        }
    }
    return true;
}
```

Figure 5.5: Shield: authorize method

5.1.4 PolicyHandler

The PolicyHandler class handles all role- and entity-based policies. The storage and lookup of these policies is achieved in very similar ways: 2D HashMaps.

The rolePolicyMap is a HashMap that links a role with a new HashMap. The new HashMap links a resource with an implementation of Condition. The Condition interface holds a single method: evaluate. The reason behind the Condition interface compared to just

a boolean expression is the ability to parse arguments. If the map contains just boolean expressions, it is not possible to parse arguments to it. The Condition interface features an evaluate method that takes the query parameter map and HTTP body as arguments. When instantiating the policymap this method is implemented and the use of these arguments is possible.

With this implementation, authorizing for example `"Doctor"` with resource `"Freezer/insert"` is achieved with a simple lookup:

```
1 return rolePolicyMap.get("Doctor").get("Freezer/insert").
    ↪ evaluate(map, body);
```

Figure 5.6: rolePolicyMap lookup

The entityPolicyMap is very similar, here the outer HashMap links an entity to a new HashMap, the new HashMap links a resource with an implementation of condition.

The entityPolicyMap and rolePolicyMap is instantiated and populated using lambda expressions, this makes both the Xtend code and the generated Java code much cleaner. See Figure 5.7 for an example of an instantiation of a 2D HashMap of role policies. On Line 2, a HashMap with all policies that applies to the Assistant is stored. On Line 3 a policy for the retrieve resource that applies to the Assistant is stored, this policy requires that the value "xPos" in the query parameter map is 10.

```
1  private HashMap<String, HashMap<String, RoleCondition>>
     ↪ rolePolicyMap = new HashMap<String, HashMap<String,
     ↪ RoleCondition>>() {{
2      put("Assistant", new HashMap<String, RoleCondition>(){{
3          put("Freezer/retrieve", (map, body) -> (Integer.
               ↪ parseInt(map.getFirst("xPos")) == 10));
4      }});
5
6      put("Doctor", new HashMap<String, RoleCondition>(){{
7          put("Freezer/retrieve", (map, body) -> ((Integer.
               ↪ parseInt(map.getFirst("xPos")) >= 2) && (Integer.
               ↪ parseInt(map.getFirst("yPos")) >= 0)));
8          put("Freezer/insert", (map, body) -> true);
9      }});
10 }};
```

Figure 5.7: Role Policy Map

5.1.5 CapabilityHandler

The CapabilityHandler manages all obligation-based policies.

A representation of all the policies are stored in the HashMap `capabilityTemplates`. The HashMap links a resource with a tree datastructure representation of all the resources, that are accessible after requesting the original resource. An example of the instantiation of the capabilityTemplates is seen in Figure 5.8.

```
1  private HashMap<String, Node<String>> capabilityTemplates =
       ↪ new HashMap<String, Node<String>>(){{
2      put("BoxDB/get", new Node<String>("BoxDB/get"){{
3          addChild(new Node<String>("Freezer/retrieve"){{
4              addChild(new Node<String>("BoxDB/retrieve"){{
5              }});
6          }});
7      }});
8      put("BoxDB/findEmptySlot", new Node<String>("BoxDB/
           ↪ findEmptySlot"){{
9          addChild(new Node<String>("Freezer/insert"){{
10             addChild(new Node<String>("BoxDB/insert"){{
11             }});
12         }});
13     }});
14 }};
```

Figure 5.8: Capability Templates Map

When a resource that produces a capability is requested, a new capability is created and stored in another HashMap called `capabilities`. This HashMap links the ID of a capability with the capability itself. Only the ID of the capability is returned to the caller, this prevents modification of the capability to gain access to other resources.

The authorize method, shown in Figure 5.9, looks for the capability ID in the capabilities HashMap, if a capability is found, the capability's `useResource` method is called. This method is descriped in Section 5.1.6. The boolean return value is whether the call is authorized or not.

The capabilities have a default lifetime of 5 minutes. This value can easily be changed by modifying the Java code. Expired capabilities are automatically removed to avoid excess memory use.

```
public boolean authorize(String UserID, UUID CapabilityID,
    ↪ String resource){
    if(!capabilities.containsKey(CapabilityID)){
        return false;
    }

    boolean result =  capabilities.get(CapabilityID).
       ↪ useResource(UserID, resource);
    return result;
}
```

Figure 5.9: Capability authorize

5.1.6 Capability

The capability class holds all the information about a capability: its own ID, the ID of the user with permission to use it, the ID of the entity it can be used on, and the tree data structure of all the resources accessible with the capability.

The capability features a method `useResource` (shown in Figure 5.10) which takes all the IDs and the requested resource as arguments. First all the IDs are compared with the ones stored locally, if they all match, the requested resource is compared with the tree. If any of the child nodes of the root has the resource, the request is authorized, and the subtree under the requested resource is saved as the new tree.

```
boolean useResource(String UserID, String resource){
    if(!UserID.equals(this.userID)){
        return false;
    }

    Node<String> t = this.resourceTree.useResource(resource);
    if (t != null){
        this.resourceTree = t;
        return true;
    }
    return false;
}
```

Figure 5.10: Capability useResource

5.1.7 InformationService interface

The last class is an implementation of the InformationServiceInterface. The interface defines a number of methods that needs to be defined when implementing PInt. The methods are used to get information about the system it is implemented in. This can vary a lot from system to system, and as a result, it does not make sense to generate this functionality.

The methods are `getRoleByEntity()`, `getRoleByOrganization()`, `getEntityType()`, and `get«Entity»()`.

The `getRoleByEntity()` takes UserID and EntityID and should return what role the user has in the organization containing the given entity.

The `getRoleByOrganization()` takes UserID and OrganizationID and should return what role the user has in that organization.

The `getEntityType()` takes an EntityID and should return what type of entity it is.

A `get«Entity»()` method is generated for each entity type defined in the DSL. A class for each entity type is also generated, with attributes that matches the entities as defined in the DSL. The `get«Entity»()` method should return an instance of the given class.

5.2 Domain-specific language

This section will show how the developed PInt DSL is implemented with Xtext. Followed by an explanation of how the code generation is achieved. In the previous section the PInt web server was presented, many of its classes have parts that are denoted as generated. How these parts are generated will be shown in Section 5.3.1. Generation is executed based on an instance of the metamodel, how the metamodel and the associated grammar is created with Xtext, is presented Section 5.3.

5.3 PInt grammar

Due to the model-driven approach of Xtext which links text patterns with the model, the rules that define the PInt grammar are closely related to the metamodel (Figure 4.7). When using Xtext the first thing to do is defining a grammar, this is done with a syntax similar to EBNF. Based on the grammar, a language infrastructure will be generated by Xtext. This infrastructure consists of a textual editor, the Ecore-metamodel, and the Java API for easy model access. This section will only look at how the PInt grammar is constructed.

Looking at endpoints again, the Xtext implementation of endpoints can be seen in Figure 5.11 and 5.12. As it was shown with the PInt DSL the endpoint must start with the keyword `endpoint` followed by a name. The name is actually an ID that is saved in a variable called name. ID means that other elements of the same type can not have the same ID, note that this is by default not enabled in the default Xtext projects. The variable `resources` contains all the elements of resource, saved as a list. In Figure 5.12 we see

the implementation of resource. Resource contains some optional rules, e.g. `parameters` surrounded by a parentheses and ended with the '?' character. At the `body` variable we see the use of a reference with the square brackets around `Entity`. The full Xtext grammar can be seen in Appendix A.

```
Endpoint:
    'endpoint' name=ID '{'
    'URI' ':' uri = STRING
    resources += Resource*
    '}'
;
```

Figure 5.11: Xtext Endpoint

```
Resource:
    'resource' name=ID '{'
    'path' ':' path = STRING
    'verb' ':' httepVerb = HttepVerbs
    'produces' ':' product = produceType
    ('parameters' ':' queryParam+=QueryParam*)?
    ('body' ':' body=[Entity])?
    '}'
;
```

Figure 5.12: Xtext Resource

Left recursive grammars

As Xtext uses a LL(*) algorithm which means that the grammar is going to be left recursive by default. When nested functionality is required, left factoring must be implemented. Figure 5.13 shows how the left factoring is implemented for logic expressions in the PInt grammar. Additionally Figure 5.13 also shows how precedence is obtained in the grammar, where AND(&&) have a higher precedence than OR(||).

```
LogicExp returns Proposition:
    Conjunction ('||' {OR.left=current} right=Conjunction)*
;

Conjunction returns Proposition:
    Condition ('&&' {AND.left=current} right=Condition)*
;

Condition returns Proposition:
     Comparison | StringComparison
;

Comparison:
    left=Exp op=RelationalOp right=Exp
;
```

Figure 5.13: Logic Expression

5.3.1 Code generation

The reason for using code generation is, as mentioned before, to separate the language from the implementation. In particular making it possible to write the policies in one way and the implementation in another. This is achieved by generating the policies in Java code based on PInt code.

The code generation in PInt is done with the toolset that Xtext provides. The main tool used for code generation is Xtend which provides useful mechanisms for writing code generators, in particular multiline template expressions. The essential elements that needs to be generated is the incoming REST calls, entities, policies, and the outgoing REST calls. There are many classes in the PInt framework, that are more or less persistent regardless of the configuration. However many classes also change depending on the configuration, that is why using template expressions for code generation is valuable. As the persistent code can be mixed with varying code.

Endpoint generation example

Looking at the PInt code from earlier (Figure 4.9 in Section 4.4.4), we see that the endpoint 'Freezer' contains a resource called 'insert'. Figure 5.14 shows some of the generated java code, based on the PInt DSL code.

```
@Path("Freezer/insert")
@PUT
@Produces(MediaType.TEXT_PLAIN)
public Response insertFreezer (...)
```

Figure 5.14: Java example of resource

Generating the code in Figure 5.14 is achieved with the Xtend code shown in Figure 5.15. The Xtend generator, first finds all the domain elements of the type endpoint, then loops through each endpoint. Each endpoint's resources are also looped through. The generation then starts on the resource element, first the path is inserted on Line 3. Line 4 inserts the HTTP verb, Line 5 the media type and finally on Line 6 the name of the method is inserted by combining the names. Code outside of the guillemets («») in the template expression is inserted directly in the Java file.

```
1 «FOR endpoint : domain.domainelements.filter(Endpoint)»
2    «FOR resource : endpoint.resources»
3         @Path("«endpoint.name»/«resource.name»")
4         @resource«.httpVerb»
5         @Produces(MediaType.«mediaTypeMapper(resource.product)
             ↪ »)
6         Public «resource.name»«endpoint.name»(...
```

Figure 5.15: Xtend example of resource generator

Generation in general

All the code generation in the PInt framework is done in the same manner as the endpoints example above. The instance of the metamodel is traversed and for each element, code is inserted in the template expression. There are in total seven different main template expressions that generates the different classes that are used in PInt.

5.3.2 Validation

Entity policies in PInt are relates to the state of an entity. This means that POST methods, that create a new instance of an entity, can not have an entity policy. To validate that no such policy is created the use of validation is implemented. The code that do this check can be seen in Figure 5.16.

```
1 @Check
2 def checkEntityPolicyIsNotUsed(EntityPolicy entityPolicy) {
3     if(entityPolicy.restResource.httpVerb == "POST"){
4         warning('EntityPolicies can not be used on POST
             ↪ methods', PIntPackage.Literals.
             ↪ ENTITY_POLICY__REST_RESOURCE)
5     }
6 }
```

Figure 5.16: Xtend validation example

5.3.3 Scoping

To enable the use of cross-referencing the right scope must be provided. For instance in a role-based policy, a reference to the role object is used, as you do not create the role object in the role policy itself. When using Xtext, the interface IScopeProvider is responsible for providing scopes. In PInt there is a global scope by default, but this is not always desirable. In Figure 5.17 an example of a scope implementation written in Xtend can be seen. This scope rule restricts the available query parameters in a role-policy.

```
1 if (context instanceof RoleRequire){
2     var rolePolicy = EcoreUtil2.getContainerOfType(context,
      ↪ RolePolicy);
3     return Scopes.scopeFor(rolePolicy.restResource.queryParam)
4 }
```

Figure 5.17: Xtend role scoping example

6 | Evaluation

In this chapter an evaluation of PInt will be conducted. First a feature comparison between KAoS and PInt is made. Following is a comparison between the established requirements and PInt. To test the functionally of PInt a range of unit tests are implemented. These tests are presented in Section 6.3. To show that the developed PInt framework can be inserted in a system like the FFU system, a proof of concept implementation of PInt is shown in Section 6.4. Lastly the scalability of PInt is discussed in Section 6.5.

6.1 Comparison with KAoS

PInt was developed as an alternative to KAoS. KAoS was the chosen policy framework for the FFU project, but unfortunately it was incomplete. The original plan was to create a DSL for configuring policies in KAoS. Instead this thesis investigated the possibility of using a DSL to fully generate a policy framework, instead of configuring an existing one. To evaluate the PInt framework a comparison with the features of KAoS and PInt will be conducted.

KAoS features that PInt lacks

Richer policy semantics:
The semantics used to express policies in KAoS are richer than the semantics in PInt. Specifically the conditions in PInts which looks very similar to boolean expressions in GPL which might be difficult for domain experts to read.

Positive and negative modality:
In KAoS it is possible to both express actions that are authorized and unauthorized. This makes KAoS more flexible in terms of expressiveness. In PInt all requests are rejected unless there is a role-policy that specifically authorizes the request.

Policy overruling:
In KAoS, it is possible to write overlapping and (in some cases) conflicting policies. This issue is solved with the use of priorities to the policies. The policy with the higher priority overrules the lower priority policy. KPAT includes functionality to detect and present conflicting policies of same priority to the user, which makes it easy to handle conflicts.

Modify policies:
KPAT in KAoS allows the administrator to view, create and modify policies after the system is initiated. In PInt, the policies are written before the system is started. It is not possible to modify the policies after system launch.

Detailed violation description:
In KAoS, when a request fails to get authorized, the user can get information about which policy exactly that caused the rejection. In PInt, when a request is rejected, the user is served a HTTP status code 403: FORBIDDEN.

PInt features that KAoS lacks

Language independency:
PInt can be implemented in systems written in any language. The only requirement is that the system exposes their resources via REST. KAoS is implemented by extending the `KAoSActorImpl` class written in Java.

Clean separation:
PInt acts as an intermediary, and is very loosely coupled with the rest of the system, this makes for a very clean separation between itself and the remaining system. Implementing KAoS is a bit more intricate as you have to modify the existing code to extend the `KAoSActorImpl` class.

Smaller trust boundary:
In PInt, the client is placed outside the trust boundary, which means you do not have to trust the client. In KAoS, the implementation requires a modification of the client, and you need to make sure the implementation is correct, and that the client obeys the decisions given by KAoS.

6.2 Comparison with Requirements

A set of requirements was derived after the analysis in Chapter 3. The requirements related to both the design of the DSL and the functionality of the generated framework.

DSL

Requirement 1a specifies that the DSL should be readable by domain experts, this means that customers can verify that the configuration of PInt is correct and that the policies matches their needs.

Measuring readability and complexity of code is a difficult task, but comparing the PInt DSL with KAoS shows quite similar semantic structure of the policies. Though policies with conditions look more like boolean statements in a GPL, and this might not be easy for domain experts to read.

Requirement 1b specifies that the DSL should be implemented as an external DSL, this was chosen to aid the fulfillment of requirement 1a, as readability usually is easier to achieve in an external DSL. PInt is implemented in an external DSL meaning the requirement is fulfilled.

Policies

Requirement 2a specifies modelling different hierarchical company structures. PInt does not itself contain a model of the structure (organizations) of the company, but provides an interface to get information about this.

Requirement 2b - 2d specifies the three types of policies (role- entity- and obligation-based) that PInt should feature. Role- and entity-based policies are implemented in the `PolicyHandler` class, and are stored in a HashMap. Obligation-based policies are implemented differently, this is due to the stateless nature of REST. Obligation-based policies are implemented with capabilities that are returned to the client. If a client wishes to access a resource protected by an obligation policy, it is the responsibility of the client to provide a capability that servers as a proof that the requiring REST calls were made prior to this call.

System integration

Requirement 3a and 3b which specifies easy integratability and clear separation between the policy enforcement and the remaining system is fulfilled almost automatically with the decision to use REST. As described in Section 6.4 the steps involved in integrating PInt

in a system are simple and involve minimal change to the existing system.

Requirement 3c is also fulfilled thanks to the use of REST. In PInt the HTTP request is simply not forwarded if the call is not authorized. This ensures that policies are always enforced even if bugs are present in the client. This is in sharp contrast to KAoS that relies on correct extension of the `KAoSActorImpl` class on the client.

The last requirement, 3d, represents the desire from the FFU developers to be able to manage the policies using REST. PInt does not feature any type of query, modification or creation of policies after the system is launched. This could be implemented in the future. More information about features that could be implemented in the future are discussed in Section 7.1.

6.3 Unit tests

As PInt was built from scratch and underwent numerous large refactorings of most of the code base. The idea of using test-driven development would do more harm than good as the tests would also have to be refactored every time the rest of the code was refactored. Instead, tests were written afterwards and served to verify the correct behavior in edge cases of different classes. Having implemented the unit tests, they will however serve to prevent unintentional breaking changes in the further development on PInt. The unit tests are created with the use of the testing framework TestNG and the mocking framework Mockito. There are 30 unit tests covering the three classes; Shield, CapabilityHandler, and PolicyHandler.

Testing the `PolicyHandler` class involves testing the policy checking based on the provided input. First step in running the tests, is the setup phase, the method `setup` (see Figure 6.1) is called before each test, providing a clean setup each time. In this case, the `setUp` method mocks an instance of `MultivaluedMap`. When running the actual test, methods used by the `map` can then be stubbed. The result of a stubbed method can be defined with the method `thenReturn`. On Line 4, an instance of the PolicyHandler is created with the test instances of `rolePolicyMap` and `entityPolicyMap`.

```
@BeforeMethod
public void setup() {
    map = mock(MultivaluedMap.class);
    policyHandler = new PolicyHandler(rolePolicyMap,
        ↪ entityPolicyMap, informationService);
}
```

Figure 6.1: Setup method in Shield test

The first actual test of the `PolicyHandler` class can be seen in Figure 6.2, this test verifies that a `role`, in this case "doctor", with the right permissions are allowed when trying to access the resource "Freezer/retrieve".

```
@Test
public void Role_with_ActionPermission_returnsTrue() {
    final boolean actual = policyHandler.roleAuthorize("Doctor
        ↪ ", "Freezer/retrieve", null);
    Assert.assertTrue(actual);
}
```

Figure 6.2: Simple test of PolicyHandler

The test of the CapabilityHandler is very similarly to the tests of the `PolicyHandler`, it simply tests input with the expected output. The `Shield` class on the other hand tests that the right procedure is followed, meaning for instance, if a resource is public, then no role policy check should be performed.

6.4 Integration with FFU - Proof of Concept

Currently the FFU system is implemented as a client–server model, which means that the simplest integration with the current FFU system would be to act as an intermediary between the client and the server. In this section the necessary steps involved in integrating PInt in the current FFU system will be demonstrated. The client software is still in development, instead the API development environment Postman is used to send the HTTP requests. As mentioned, minor changes needs to be made to the client before PInt can be used in a system. These changes are easily applied in Postman to to implement PInt. The steps needed to integrate PInt in an existing system are the following:

1. Incoming HTTP requests must include a UserID, EntityID, and OrganizationID in the query parameter map.
2. An implementation of the `InformationServiceInterface` must be written.
3. Endpoints, resources, and roles must be modelled in the PInt DSL.

4. Define policies in the PInt DSL.
5. Point the client to the PInt intermediary service. (Make sure that the server only accepts requests from the PInt service).
6. (Optional) If obligation-based policies are needed, capabilities must be handled by the client and added in the header of subsequent requests.

Step 1. is easy to do with Postman and should also be easy to implement in the client as well. The information required is closely tied to the existing request structure. The UserID is known as it is already required for users to login when using the client. OrganizationID is tied to the user as well and presumed to be available. EntityID is needed for all PUT, GET, and DELETE requests and in the case of FFU, these are already provided as a path parameter, see Figure 6.3. Which means that it simply needs to be inserted as a query parameter.

```
1 http://ffu.dk/biostore/logicalsets/{{logicalsetsID}}
```

Figure 6.3: Logicalsets URI

Step 2. requires an implementation of the interface and is very specific from system to system. To showcase the PInt integration in the FFU system, a simple implementation of the interface has been created. The `getRole` method is made to look for the creator of the entity by the EntityID, the creator's organization is then used as the organization in which the entity is a part of. The user is then found by the UserID, cross checking the user's groups with the organization of the entity. If there is a match the role associated with the group is returned. This implementation is all made with the existing REST resources the FFU API exposes. With this implementation, PInt can now find the role of the user sending the request and apply policies.

Step 3. involves modelling the FFU REST API with PInt. The requests in Table 6.1 is modelled in PInt.

HTTP verbs	**Name**	**Incl. body**	**Path**
POST	authenticate	no	biostore/authenticate/login
GET	physicalsets	no	biostore/physicalsets
PUT	physicalsets	yes	biostore/physicalsets
POST	physicalsets	yes	biostore/physicalsets

Table 6.1: FFU REST API

The first request is the authenticate request and it must be used before a client can start to interact with the FFU server. Authenticate will return key-value pairs as cookies and are used for creating a session with the FFU server. Next we have a GET, PUT, and POST for a physicalsets. The domain-specific code that models the server can be seen in Figure 6.4. Additionally the modelling of roles and the one entity used, can be seen in the same figure.

Step 4. is creating policies, in this example four policies are created, see Figure 6.4. Line 38 shows the public policy making all users able to authorize a session with the server. Line 40 allows Oberservers to access the resource get. Line 44 shows a policy that dictates that the PUT request must have a containerSize of 81 or 64, if the containerSpec is c or a respectively.

Step 5. is easy to do with Postman and should also be easy to implement in the client as well. It would also be wise to modify the server to reject requests directly from the client, this is necessary to avoid unauthorized requests.

PInt is now ready to enforce the specified policies. For instance the `FFU.get` request should return all the physicalstes, for users with the role Observer. However, if the user have the role Researcher, PInt should return a `HTTP Status 403 - Forbidden status`.

```
endpoint FFU{
    url:"http://tek-ffu-h0a.tek.sdu.dk:80/"

    resource authorize {
        path: "biostore/authenticate/login"
        verb: POST
        produces: json
    }

    resource get {
        path:"biostore/physicalsets"
        verb: GET
        produces: json
    }

    resource post {
        path: "biostore/physicalsets"
        verb: POST
        produces: json
        body: {
            containerSize: int
        }
    }

    resource put {
        path: "biostore/physicalsets"
        verb: PUT
        produces: json
        body: {
            containerSize: int
        }
    }
}

role Researcher
role Observer

rolepolicy: everyone can access FFU.authorize

rolepolicy: Observer can access FFU.get
rolepolicy: Researcher can access FFU.post
rolepolicy: Researcher can access FFU.put if(
    (StringCompare(body.containerSpec, "c") && body.
       ↪ containerSize == 81) ||
    (StringCompare(body.containerSpec, "a") && body.
       ↪ containerSize == 64) )
```

Figure 6.4: Demo of FFU as modelled in PInt

6.5 Scalability

An important thing to take into account if you consider using PInt to enforce policies on your system, is the performance of the framework. How much does the framework increase response time? How many requests per second can it handle? What options are available to scale the framework up if needed?

The response time of the framework depends on the implementation of the interface. The role- and obligation-based policies are resolved just by looking up values in a HashMap. Entity-based policies require PInt to get the state of the entity using the interface, this adds a delay that depends on the implementation. The framework generated in Figure 6.4 takes less than 1ms to authorize role-based policies. The interface used in this example uses REST calls to get information about the entities. Since HTTP uses TCP to communicate, and TCP has a quite elaborate 3-way handshake to initiate connections, this adds around 300ms to the response time of the framework. However, an implementation that uses a faster protocol to get entity information would lower the response time significantly.

Since PInt acts as a web server that simply forwards calls. It is possible to launch multiple instances of these and run them in parallel, a load balancer can then be installed in front that distributes load between the servers. This means that with enough hardware, PInt is theoretically speaking infinitely scalable.

7 | Perspectives

In this chapter we present some ideas for features that could be implemented in PInt in the future. Then we make a final conclusion on the project as a whole.

7.1 Future Work

PInt is a very capable framework, with sophisticated features and a clean implementation. However, there are plenty of features that are not implemented that would improve the applicability of the framework a lot. Most notably the policy management with REST calls as mentioned in Requirement 3d. In this section, some features that could be useful to implement in PInt in the futures are presented along with a discussion on how they could be implemented on a conceptual level.

Trigger policies

A type of policy that could be nice to implement is trigger-based policies. Essentially, the framework would make a call if a policy specifies it. This call could be to make a notification to either the user or an administrator. In the FFU system it could also be to eject samples with an exceeded expiration date.

Like all other policies, trigger policies would need to be evaluated whenever a request is made to PInt. But unlike other policies, trigger policies would also need to be evaluated at regular intervals to handle policies that relate to time. The best way to implement trigger policies would be to have some kind of event system, that handles and processes events, and checks if trigger policies are activated.

REST policy management

A very desirable feature of KAoS is their policy management capabilities. KAoS has a directory service which handles policies at run time. KPAT can then be used to modify

existing policies and create new ones. In PInt's current state, policies can not change after compilation. Adding the ability for PInt to be able to do so would be very beneficial.

PInt currently stores all policies in 2D HashMaps while HashSets stores information about public resources. Modifying the HashMaps and HashSets could lead to inconsistency between them, which would result in unexpected behavior of the framework. Instead, the best approach to implementing this would be to store policies in a different data structure (maybe a database), and then instantiate the HashMaps from this.

Produces and consumes

The `@Produces` and `@Consumes` annotations when developing RESTful web services is a way for the server to communicate what media type it accepts and what media type the response is. In both cases, it is possible to give an array of supported media types. The inclusion of the `@Produces` and `@Consumes` annotations in PInt is very limited. PInt only supports the `@Produces` annotation, and only one media type for each resource can be defined. This is fine for the FFU where the communication between the client and server is limited to just a single tablet with custom software to communicate with the FFU. But in other systems, where the client and server has less knowledge about each other, being able to negotiate media types for exchanging information might be useful.

Implementing support for `@Produces` and `@Consumes` annotations in PInt should be very simple. Just add a rule in the DSL for consumes, and allow multiple media types in both consumes and produces to be specified.

More readable conditions

The policies in PInt are quite similar to KAoS in terms of readability. However, the conditions at the end of a policy are more similar to a boolean expression in a GPL. A future improvement of PInt could be to improve the readability of these expression.

An improvement of the readability could be implemented with inspiration from KAoS. In KAoS the conditions are implemented with the keywords `which has attributes:` followed by the properties that should have certain values.

Path parameters

In RESTful web services, the path parameter is often used to specify the ID of the resource being accessed e.g. `"www.ffu.dk/biostore/{{someID}}"` where `someID` is a path parameter. In PInt, the ID is retrieved from the query parameter map instead. Being able to

specify where to retrieve the ID would be a useful feature in PInt, as this would minimize the amount of refactoring of the client that is needed to implement PInt.

Multiple files

When modelling the organization and entities, and specifying policies in the same file, the file can quickly become quite long and disorganized. Especially for larger organizations. Splitting the model of the company and entities into a file separate from the policies would alleviate this issue. Furthermore it would also be simpler to present the policies to the domain experts.

Contradicting conditions

Many IDEs have support for detecting if boolean expressions always return false, in other words, if they contradict themselves. An example could be `(x < 5 && x > 7)`. Implementing this for conditions in the DSL would help avoid creating these contradicting policy conditions.

Authentication

The process of authenticating the user has been placed outside the scope of this thesis. However, to fully utilize the features of PInt and trust the requests sent by the client, the implementation of authentication would be needed. Currently, the client is responsible for inserting the UserID in the query parameter map which is used to deduct the role and perform authorization against the policies in the system. But there is nothing that validates that the UserID supplied by the user is the correct ID. In other words, if a user (A) knows the UserID of another user (B) in the system, then user A can pretend to be user B without submitting any further information.

7.2 Conclusion

In this thesis, the policy framework PInt has been developed. PInt is a Java-based intermediary that applies policies on HTTP requests. Based on the declarative policy specifications, PInt automatically generates a REST-based intermediary service that mediates the interactions between a client and a server.

In practice PInt is really two projects, the PInt intermediary software and the DSL that generates it. The developed intermediary has shown how policies can be enforced in a REST environment. This is achieved by inserting PInt between the client and server which

stands in contrast with KAoS. When using KAoS, the client is part of the trust boundary, whereas by using PInt the client can be put outside of the trust boundary. By having the client outside of the trust boundary, it can be guaranteed that policy enforcement is done with or without the cooperation of the client.

The second area of the thesis was developing the DSL for generating the PInt framework. By creating the PInt DSL, a language for modeling policies and the system PInt should be insert into, has been made easy. Additionally, the grammar for creating policies has been simplified to a human like grammar and not a general-purpose programming language.

PInt is developed in close relation with the FFU use case, hence the REST-based approach, but PInt is also designed with the intent of being useful in other contexts. In practice this means that PInt can be inserted in REST-based systems that have resources that needs policy enforcement, where the concepts of UserID, EntityID, organization and roles are applicable. The ease of integration is shown with a proof of concept example, where PInt is inserted between the client and the FFU server and its corresponding REST resources. Several policies are created and the successful enforcement of these is achieved.

With the fulfillment of almost all the requirements and a working policy-enforcing intermediary, we conclude that the project has been a success. Regardless of whether PInt is implemented on the FFU system or not, the project has been a learning experience for us, giving us knowledge that we can use in our future endeavours.

Bibliography

[1] "Introduction to Access Control Systems." [Online]. Available: http://www.silvaconsultants.com/introduction-to-access-control-systems.html

[2] M. P. Gallaher, A. C. O. Connor, and B. Kropp, "The Economic Impact of Role-Based Access Control."

[3] P. Stavroulakis and M. Stamp, *Handbook of Information and Communication Security*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[4] A. Spiessens, "Patterns of safe collaboration," Universite catholique de Louvain, Tech. Rep., 2007.

[5] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott, "KAoS policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement," *Proceedings - POLICY 2003: IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pp. 93–96, 2003.

[6] N. J. George, R. J. Jasper, M. Rosman Lafever, K. M. Morrison, D. B. Rosenthal, S. R. Tockey, J. D. Woolley, J. M. Bradshaw, G. A. Boy, and P. D. Holm, "KAoS: A Knowledgeable Agent-oriented System," *Association for the Advancement of Artificial Intelligence*, pp. 24–30, 1994. [Online]. Available: http://aaaipress.org/Papers/Symposia/Spring/1994/SS-94-03/SS94-03-004.pdf

[7] A. Uszok, J. M. Bradshaw, J. Lott, M. Breedy, L. Bunch, P. Feltovich, M. Johnson, and H. Jung, "New developments in ontology-based policy management: Increasing the practicality and comprehensiveness of KAoS," *Proceedings - 2008 IEEE Workshop on Policies for Distributed Systems and Networks, POLICY 2008*, pp. 145–152, 2008.

[8] A. Uszok, J. M. Bradshaw, J. Lott, M. Johnson, M. Breedy, M. Vignati, K. Whittaker, K. Jakubowski, J. Bowcock, and D. Apgard, "Toward a flexible ontology-based policy

approach for network operations using the KAoS framework," *Proceedings - IEEE Military Communications Conference MILCOM*, pp. 1108–1114, 2011.

[9] Bradshaw, J.M. A. Uszok, M. Breedy, L. Bunch, T.C. Eskridge, P.J. Feltovich, M. Johnson, J. Lott and M. Vignati., "The KAoS Policy Services Framework," *Eighth Cyber Security and Information Intelligence Research Workshop (CSIIRW 2013). Oak Ridge, TN: Oak Ridge National Labs*, 2013.

[10] Y. Demchenko, L. Gommans, A. Tokmakoff, and R. van Buuren, "Policy Based Access Control in Dynamic Grid-based Collaborative Environment," *International Symposium on Collaborative Technologies and Systems (CTS'06)*, pp. 64–73, 2006. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1644117

[11] M. Johnson, J. M. Bradshaw, H. Jung, N. Suri, and M. Carvalho, "Policy management across multiple platforms and application domains," *Proceedings - 2008 IEEE Workshop on Policies for Distributed Systems and Networks, POLICY 2008*, pp. 197–202, 2008.

[12] "KAoS Policy Services Framework: User Guide," 2013. [Online]. Available: http://ontology.ihmc.us/KAoS/KAoSUsersGuide.pdf

[13] J. M. Bradshaw, P. J. Feltovich, M. J. Johnson, L. Bunch, M. R. Breedy, T. Eskridge, H. Jung, J. Lott, and A. Uszok, "Coordination in human-agent-robot teamwork," *2008 International Symposium on Collaborative Technologies and Systems, CTS'08*, no. June 2014, pp. 467–476, 2008.

[14] Bradshaw, J.M. A. Uszok, M. Breedy, L. Bunch, T.C. Eskridge, P.J. Feltovich, M. Johnson, J. Lott and M. Vignati., "The KAoS Policy Services Framework," *Eighth Cyber Security and Information Intelligence Research Workshop (CSIIRW 2013). Oak Ridge, TN: Oak Ridge National Labs*, 2013.

[15] R. T. Fielding, "REST: architectural styles and the design of network-based software architectures," Doctoral dissertation, University of California, Irvine, 2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[16] https://www.w3.org/TR/uri-clarification/. [Online]. Available: https://www.w3.org/TR/uri-clarification/

[17] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Rfc 2616, hypertext transfer protocol – http/1.1, section 1.4," 1999.

[Online]. Available: http://www.rfc.net/rfc2616.html

[18] M. Fowler, *Domain Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.

[19] G. Bocewicz and Z. A. Banaszak, "Declarative approach to cyclic steady state space refinement: periodic process scheduling," *The International Journal of Advanced Manufacturing Technology*, vol. 67, no. 1, pp. 137–155, Jul 2013. [Online]. Available: https://doi.org/10.1007/s00170-013-4760-0

[20] LANGUAGE ENGINEERING FOR EVERYONE! [Online]. Available: https://www.eclipse.org/Xtext/

[21] (2017) Robotfryser skal holde styr på vores gener. [Online]. Available: https://innovationsfonden.dk/da/nyheder-presse-og-job/robotfryser-skal-holde-styr-pa-vores-gener

[22] D. Memo and F. S.-a. Interface, "2252 – Design Memo FFU ADS FFU Service-Automation Interface specification in ADS 2252 – Design Memo," University of Southern Denmark, Tech. Rep., 2015.

[23] W. A. Coolidge, "2252 – Design Memo FFU Logical Service," University of Southern Denmark, Tech. Rep., 2015.

[24] ——, "2252 – Design Memo, Storage service implementation," University of Southern Denmark, Tech. Rep., 2015.

[25] ——, "2252 – FFU System Concepts," University of Southern Denmark, Tech. Rep., 2015.

[26] Sundheds- og Ældreministeriet, "Bekendtgørelse om medicinsk udstyr," pp. 42–43, 2006.

[27] (2019) Medical devices. [Online]. Available: https://laegemiddelstyrelsen.dk/en/devices/

[28] K. Madsen, "Positive følgevirkninger ved ce-maerket software som medicinsk udstyr."

[29] OpenAPIs. [Online]. Available: https://www.openapis.org/about

Appendices

A | Full Xtext grammar

```
grammar dk.sdu.ffu.pint.Pint with org.eclipse.xtext.common.Terminals
generate pint "http://www.sdu.dk/ffu/pint/Pint"

Intermediary:
        elements+=Element*
;
Element:
        Endpoint | Entity | Capability | Role | RolePolicy | EntityPolicy
;
Capability:
        'capabilities' '{'  capabilities += CapabilityRule+ '}'
;
CapabilityRule:
        restResource = [RestResource | FQN] '{'
                subRestResource += CapabilityRestResource*
        '}'
;
Entity:
        'entity' name=ID '{' ('uri' ':' uri=STRING)? 'identifier' idType=Type  idName=ID properties += Property* '}'
;
Property:
```

```
        type=(Type|EntityRef) name=ID
;
Type:
        ({tString}'string' | {tInt}'int' | {tFloat}'float' | {tBoolean}'boolean' | {tDate}'date' | {tTime}'time')
;
EntityRef:
        ref=[Entity]
;
CapabilityRestResource:
        restResource = [RestResource | FQN] ('{' nestedCapabilityRestResources += CapabilityRestResource* '}')?
;
Endpoint:
        'endpoint' name=ID '{'
        'url' ':' uri = STRING
        restResources += RestResource*
        '}'
;
FQN: ID ("." ID)*;
RestResource:
        'resource' name=ID '{'

        'path' ':' path = STRING
        'verb' ':' httpVerb = HttpVerbs
        'produces' ':' product = produceType
        ('parameters' ':' queryParam+=QueryParam*)?
        ('body' ':' httpBody=HttpBody)?
        '}'
;
HttpVerbs:
         'GET' | 'POST' | 'PUT' | 'DELETE'
;
QueryParam:
        name=STRING type=QueryType
;
QueryType:
```

```
        'int'| 'string'
;
produceType:
        'html' | 'plain' | 'json'
;
RoleType:
        roleRefs  = RoleRef | {Everyone}'everyone'
;
RoleRef:
        ref = [Role]
;
Role:
        'role' name=ID
;
RolePolicy:
        'rolepolicy' ':' role=RoleType 'can' 'access' restResource=[RestResource|FQN] require=RoleRequire?
;
EntityPolicy:
        'require' ':' entity=[Entity] require=Require 'for' restResource=[RestResource|FQN]
;
Require:
        '(' requirement=LogicExp ')'
;
RoleRequire:
        'if' '(' requirement=rLogicExp ')'
;
LogicExp returns Proposition:
        Conjunction ('||' {OR.left=current} right=Conjunction)*
;
Conjunction returns Proposition:
        Condition ('&&' {AND.left=current} right=Condition)*
;
Condition returns Proposition:
         Comparison | StringComparison
;
```

```
Comparison:
        left=Exp op=RelationalOp right=Exp
;
RelationalOp:
        {RelEQ} '==' | {RelLT} '<' | {RelGT} '>' | {RelLTE} '<=' | {RelGTE} '>=' | {RelNEQ} '!='
;
Exp returns Expression:
        Factor ( ('+' {Add.left=current} | '-' {Sub.left=current}) right=Factor)*
;
Factor returns Expression:
        Primitive ( ('*' {Mul.left=current} | '/' {Div.left=current}) right=Primitive)*
;
Primitive returns Expression:
        IntExp | EntityProperty | Bool | DateComparison | '(' logicExp=LogicExp ')'// | StringComparison
;
DateComparison:
        'DaysBetween' '(' left=DateExp ',' right=DateExp ')'
;
DateExp:
        EntityProperty | {today}'today'
;
EntityProperty:
        entityPropertyRef=EntityPropertyRef
;
EntityPropertyRef:
         propertyRef=[Property] ('.'ref=EntityPropertyRef)?
;
IntExp:
        value=INT
;
StringComparison:
        'StringCompare' '(' left=StringValue ',' right=StringValue ')'
;
StringValue:
        StringPrim | EntityProperty | QueryParamRef | BodyRef
```

```
;
rStringComparison:
        'StringCompare' '(' left=rStringValue ',' right=rStringValue ')'
;
rStringValue:
        StringPrim | QueryParamRef | BodyRef
;
StringPrim:
        value=STRING
;
Bool:
        bool=('true' | 'false')
;
rLogicExp returns Proposition:
        rConjunction ('||' {OR.left=current} right=rConjunction)*
;
rConjunction returns Proposition:
        rCondition ('&&' {AND.left=current} right=rCondition)*
;
rCondition returns Proposition:
         rComparison | rStringComparison
;
rComparison:
        left=rExp (op=RelationalOp right=rExp)?
;
rExp returns Expression:
        rFactor ( ('+' {Add.left=current} | '-' {Sub.left=current}) right=rFactor)*
;
rFactor returns Expression:
        rPrimitive ( ('*' {Mul.left=current} | '/' {Div.left=current}) right=rPrimitive)*
;
rPrimitive returns Expression:
        IntExp | '(' logicExp=rLogicExp ')' | QueryParamRef | Bool | BodyRef
;
HttpBody:
```

```
        '{' jsonElements += JsonElement+ '}'
;
JsonElement:
        JsonObject | KvPair
;
JsonObject:
        name = ID ':' '{' jsonElements += JsonElement+ '}'
;
KvPair:
        name = ID ':' type = JsonType
;
JsonType:
        'int'| 'string'
;
QueryParamRef:
        'queryparameter' '.' ref=[QueryParam]
;
BodyRef:
        'body'  jsonRef=TreeElement
;
TreeElement:
        LeafElement | BranchElement
;
BranchElement:
        '.' jsonObjectRef=[JsonObject] child=TreeElement
;
LeafElement:
        '.' kvPairRef=[KvPair]
;
```